Osval Antonio Montesinos López
Abelardo Montesinos López
José Crossa

# Multivariate Statistical Machine Learning Methods for Genomic Prediction

*Foreword by* Fred van Eeuwijk

OPEN ACCESS

# Multivariate Statistical Machine Learning Methods for Genomic Prediction

Osval Antonio Montesinos López •
Abelardo Montesinos López • José Crossa

# Multivariate Statistical Machine Learning Methods for Genomic Prediction

Foreword by
Fred van Euwijk

Osval Antonio Montesinos López
Facultad de Telemática
University of Colima
Colima, México

Abelardo Montesinos López
Departamento de Matemáticas
University of Guadalajara
Guadalajara, México

José Crossa
Biometrics and Statistics Unit
CIMMYT
Texcoco, Estado de México, México

Colegio de Postgraduados
Montecillos, Estado de México, México

ISBN 978-3-030-89009-4 ISBN 978-3-030-89010-0 (eBook)
https://doi.org/10.1007/978-3-030-89010-0

© The Editor(s) (if applicable) and The Author(s) 2022
**Open Access** This book is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.
The images or other third party material in this book are included in the book's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the book's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

*"This book is a prime example of CIMMYT's commitment to scientific advancement, and comprehensive and inclusive knowledge sharing and dissemination. The book presents novel models and methods for genomic selection theory and aims to facilitate their adoption and use by publicly funded researchers and practitioners of National Agricultural Research Extension Systems (NARES) and universities across the Global South. The objectives of the authors is to offer an exhaustive overview of the current state of the art to give access to the different new models, methods, and techniques available to breeders who often struggle with limited resources and/or practical constraints when implementing genomics-assisted selection. This aspiration would not be possible without the continuous support of CIMMYT's partners and donors who have funded non-profit frontier research for the benefit of millions of farmers and low-income communities worldwide. In this regard, it could not be more fitting for this book to be published as an open access resource for the international plant breeding community to benefit from. I trust that this publication will become a mandatory reference in the field of genomics-assisted breeding and that it will greatly contribute to accelerate the development and deployment of resource efficient and nutritious crops for a hotter and drier world."*

**Bram Govaerts,** Director General, a.i.
CIMMYT

# Foreword

In the field of plant breeding, the concept of prediction has always played a key role. For a long time, breeders tried to predict superior genetic performance, genotypic value or breeding value, depending on the mating system of the crop, from observations on a phenotype of interest with estimators that relied on the ratio of genetic to phenotypic variance. This ratio was usually the heritability on a line mean basis. When molecular markers became available, around 1990, it did not take too long before predictions of genotypic values were created from linear mixed models containing design or predictor matrices built on marker information. In plants, a paper by Rex Bernardo in 1994 (*Crop Science*, 34) was probably the first to propose this. A similar idea, under the name of genomic prediction and selection, revolutionized animal breeding after the publication of a famous paper by Theo Meuwissen, Ben Hayes, and Mike Goddard in *Genetics* 157, in 2001.

In contrast to its immediate success in animal breeding, in plant breeding the introduction and absorption of genomic prediction took longer. A reason for this difference between animal and plant breeding can be that for many traits in plant breeding it was not that difficult to identify useful quantitative trait loci (QTLs), whereas in animal breeding the detection of QTLs was more cumbersome. Therefore, in plant breeding some progress was possible by manipulating QTL alleles in marker-assisted selection procedures. Another reason for the slower implementation of genomic prediction in plants is the smaller order of magnitude for the training sets. In animal breeding 1000s or 10,000s are common; in plants 1000 is an outlier at the high end. A further and major complication in genomic prediction procedures for plants is the ubiquity of genotype by environment interaction.

Many methods have been proposed for genomic prediction over the last 20 years. Because these methods aim at using large numbers of SNPs (or other markers/ sequence information) for prediction of genotypic values, the number of predictors is typically higher, and often a lot higher, than the number of genotypes with phenotypes, i.e., the size of the training set. Therefore, some form of regularization is necessary in the estimation procedure for the SNP effects. This regularization can come in ways familiar to plant breeders. The classical linear mixed model for

prediction of genotypic values, which uses line mean heritability and random genotypic effects that are structured by pedigree relations, can be generalized to a similar linear mixed with genotypic effects structured by marker relationships between genotypes. In that case, we move from classical best linear unbiased predictions, or BLUPs, to genome-enabled BLUPs, or G-BLUPs.

The G-BLUP has become the workhorse of genomic prediction in animal and plant breeding. For computational reasons, whenever the number of genotypes is less than the number of markers, a mixed model formulation that structures relationships between genotypes from marker profiles is more attractive than a formulation inserting a design matrix containing the SNPs itself. The latter formulation is a form of Ridge regression, and its predictions are called RR-BLUPs. G-BLUP and RR-BLUP can be motivated from a shrinkage perspective within a mixed model context, but alternatively, motivations from a penalized regression or Bayesian context are possible. RR-BLUP and G-BLUP use a Ridge penalty, while they can also be understood from the use of a single Normal prior on the SNP effects. At this point, it is obvious that all kinds of other penalties and priors have been proposed in genomic prediction, with the idea of inserting information on the genetic architecture of the trait. For example, penalties and priors can induce a degree of sparsity, like in Lasso-type penalties and Laplace/double exponential priors. Further, different assumptions can be made on the distribution of the (non-null) SNP effects. In another direction, SNP effects can be assigned to belong to different distributions with different variances, i.e., different magnitudes of SNP effects.

The tool kit for predicting genotypic values, or better, expected phenotypic performance from marker profiles, was quickly filled with linear mixed models (G-BLUP, RR-BLUP), penalized regressions (Ridge, Lasso), and Bayesian methods (A, B, Cπ, R). An excellent review of all these methods was produced by de los Campos et al. in *Genetics* 193, 2013. In addition to the above “statistical” prediction methods, all kinds of machine learning techniques like support vector machines, random forests, and deep learners are proposed for genomic prediction. A good review and comparison of statistical and machine learning methods was offered by Azodi et al., *G3*, 2019. A somewhat reassuring and equally disappointing conclusion from this paper and many other papers on comparison of genomic prediction methods is that prediction accuracies do not change very much between methods.

The authors of the book that you have in front of you now played an important role in the evaluation of a wide spectrum of prediction methods in plants, starting from G-BLUP and extending to deep learning. Their work included real wheat and maize data from the well-known international breeding institute CIMMYT, the international center for the improvement of maize and wheat. They were also instrumental in generalizing genomic prediction methods to multiple traits and environments. The route chosen here was via a generalization of multi-trait and multi-environment mixed models by defining a structure for the random genotype × trait effects and genotype × environment effects as a Kronecker product of structuring matrices on genotypes on the one hand and traits or environments on the

other hand. For the genotypes, the relationships were defined by markers; for the traits or environments, unstructured or factor analytic variance-covariance matrices can be chosen. When explicit environmental characterizations are available, relationships between environments can also be based on environmental similarities, as explained by Jarquin et al. in *Theoretical and Applied Genetics* 127, 2014. As another major contribution, the book authors showed how to do genomic prediction for binary traits, categorical traits, and count data. The way forward for such data is via the use of Bayesian implementations of generalized linear (mixed) models.

A remarkable achievement of the current book is that the authors have been able to present a full spectrum of genomic prediction methods for plant data. The book gives the theory and practice of genomic predictions for single and multiple environments, single and multiple traits, for continuous data as well as for binary, categorical, and count data. For the "traditional statistical methods," mixed models are explained, alongside penalized regressions and Bayesian methods. For the machine learning techniques, we find kernel methods, support vector machines, random forests, and all kinds of old and modern machine and deep learners. In between we see functional regression. The theoretical treatments of the methods are rather advanced for average plant breeders, but the book simultaneously offers a lot of introductory material on the philosophy behind statistical learning, bias-variance trade-offs, training and test sets, cross validation, matrices, penalties, priors, kernels, trees, and deep learners that allows less statistically trained readers to get some flavor of most of the techniques. Another positive point on the usability of the book is the ample inclusion of R programs to do the analyses yourself. The central packages for prediction are BGLR, BMTME, glmnet, and Keras, with the first two closely connected to the authors of this book. Some worry may be expressed on the computational feasibility of the Bayesian implementations for larger data sets, i.e., 100s of genotypes and 30 or more traits/trials.

The book offers more than just methods for genomic prediction. Most of the prediction methods can easily be adapted to other types of prediction, where the target traits remain the same, while the inputs are not SNPs, but metabolites, proteins, gene expressions, methylations, and further omics data. Different types of predictors can also be combined as in kernel methods, where different types of inputs serve to define multiple kernels, which translate to (transformed) genotype $\times$ genotype similarity matrices that structure genotypic random effects. For example, two random genotypic effects may be defined, with a SNP-based relationship matrix structuring the first random genotypic effect and a metabolite-based relationship matrix structuring the second genotypic effect. Another example consists in adding a dominance kernel to a mixed model that already contains a kernel for the additive effects. To include epistatic effects of all orders in addition to additive effects, Reproducing Kernel Hilbert Space models can be used. For the use of high-throughput phenotyping information as predictors for phenotypic traits, functional regression methods are described. The use of images for prediction of plant traits is facilitated by deep learners.

This book is encyclopedic in its approach to prediction in plant breeding. About any feasible method is described with its statistical properties, examples, and

annotated software. Most if not all prevailing applications are covered. Readability and usefulness are improved by the fact that chapters are to a large extent self-contained. The book is a unique reference manual for people with an academic or professional interest in prediction methods in plants.

Wageningen, The Netherlands  Fred van Eeuwijk
1 September 2021

# Preface

In plant breeding, prediction models are essential for the successful implementation of genomic selection (GS), which is considered a predictive methodology used to train models with a reference training population containing known phenotypic and genotypic data and then perform predictions for a testing data set that only contains genomic data. However, because a universal model is nonexistent (free lunch theorem), it is necessary to evaluate many models for a particular data set and subsequently choose the best option for each particular situation. Thus, a great variety of statistical models are used for prediction in GS. The GS genome-based statistical machine prediction models face several complexities, as the choice of the model depends on multiple factors such as the genetic architecture of the trait, marker density, sample size, the span of linkage disequilibrium, and the genotype $\times$ environment interaction.

In the field of plant science, the most popular statistical learning model is the linear mixed model, which uses Henderson's equations to find the best linear unbiased estimates (BLUEs) for fixed effects, the best linear unbiased predictors (BLUPs) for random effects, and the Bayesian counterpart of this model from which the authors developed different versions that they coined into the so-called Bayesian Alphabet Methods [Bayesian Ridge Regression (BRR), BayesA, BayesB, BayesC, Bayes Lasso (BL), etc.]. Several machine learning methods are available, including deep learning, random forest, and support vector machines. Random forest and support vector machines are very easy to implement since they require few hyperparameters to be tuned and quickly provide very competitive predictions.

The methods arising from statistical and machine learning fields are called statistical machine learning methods.

All of the statistical machine learning methods require highly preprocessed inputs (feature engineering) to produce reasonable predictions. In other words, most statistical machine learning methods need more user intervention to preprocess inputs, which must be done manually. Chapter 1 assesses the use of big data for prediction and estimation through statistical machine learning and its applications in agriculture and genetics in general, and it provides key elements of genomic selection and its

potential for plant improvement. Chapter 2 details the data preparation necessary for implementing statistical machine learning methods for genomic selection. We present tools for cleaning, imputing, and detecting minor and major allele frequency computation, marker recodification, marker frequency of heterogeneous, frequency of NAs, and methods for computing the genomic relationship matrix. Chapter 3 provides details of the linear multiple regression model with gradient descendent methods described for learning the parameters under this model. Penalized linear multiple regression is derived for Ridge and Lasso penalties.

This book provides several examples in every chapter, including the necessary R codes, thus facilitating rapid comprehension and use.

Chapter 4 describes the overfitting phenomenon that occurs when a statistical machine learning model thoroughly learns the noise as well as the signal that is present in the training data, and the underfitted model when only a few predictors are included in the statistical machine learning model that represents the complete structure of the data pattern poorly. In Chapter 5, we explain the linear mixed model framework by exploring three methods of parameter estimation (maximum likelihood, EM algorithm and REML) and illustrate how genomic-enabled predictions are performed under this framework. The use of linear mixed models in Chap. 5 is illustrated for single-trait and multi-trait linear mixed models, and the R codes for performing the analyses are provided.

Chapter 6 describes the Bayesian paradigm for parameter estimation of several Bayesian linear regressions used in genomic-enabled prediction and present in various examples of predictors implemented in the Bayesian Generalized Linear Regression (BGLR) library for continuous response variables when including main effects (of environments and genotypes) as well as interaction terms related to genotype-environment interaction. The classic and Bayesian prediction paradigms for categorical and count data are delineated in Chap. 7; examples were implemented in the BGLR and glmnet library, where penalized multinomial logistic regression and penalized Poisson regression are given.

A notable semiparametric machine learning model is introduced in Chap. 8, where some loss functions under a fixed model framework with examples of Gaussian, binary, and categorical response variables are provided. We illustrate the use of mixed models with kernels including examples for multi-trait RKHS regression methods for continuous response variables.

We point out the origin and popularity of support vector machine methods in Chap. 9 and highlight their derivations by the maximum margin classifier and the support vector classifier, explaining the fundamentals for building the different kernel methods that are allowed in the support vector machine. In Chap. 10, we go through the foundations of artificial neural networks and deep learning methods, drawing attention to the main inspirations for how the methods of deep learning are built by defining the activation function and its role in capturing nonlinear patterns in the input data. Chapter 11 presents elements for implementing deep neural networks (deep learning, DL) for continuous outcomes. Several practical examples with plant breeding data for implementing deep neural networks in the keras library are outlined. The efficiency of artificial neural networks and deep learning methods is

extended to genome-based prediction in keras for binary, categorical, and mixed outcomes under feed-forward networks (Chap. 12). Deep learning is not frequently used in genome-based prediction, as its superiority in predicting performance of unobserved individuals over conventional statistical machine learning methods is still unclear. Nevertheless, few applications of DL for genomic-enabled prediction show that it improves the selection of candidate genotypes at an early stage in the breeding programs, as well as improving the understanding of the complex biological processes involved in the relationship between phenotypes and genotypes.

We provide several examples from plant breeding experiments including genomic data. Chapter 13 presents a formal motivation for Convolution Neural Networks (CNNs) that shows the advantages of this topology compared to feed-forward networks for processing images. Several practical examples with plant breeding data are provided using CNNs under two scenarios: (a) one-dimensional input data and (b) two-dimensional input data. We also give examples illustrating how to tune the hyperparameters in the CNNs.

Modern phenomics methods are able to use hyperspectral cameras to provide hundreds of reflectance data points at discrete narrow bands in many environments and at many stages of crop development. Phenotyping technology can now be used to quickly and accurately obtain data on agronomic traits based on extensive investments and advancements in plant phenotyping technologies. The main objective of HTP is to reduce the cost of data per plot and increase genomic-enabled prediction accuracy early in the crop-growing season. Chapter 14 presents the theoretical fundamentals and practical issues of using functional regression in the context of phenomic (images) and genomic-(dense molecular markers) enabled predictions. Functional regression represents data by means of basis functions. We derived a Bayesian version of functional regression and explain details of its implementation.

Chapter 15 provides a description of the random forest algorithm, the main hyperparameters that need to be tuned, as well as the different splitting rules that are key for its implementation. Several examples are provided for training random forest models with different types of response variables using plant breeding and genome-based prediction. A random forest algorithm for multivariate outcomes is provided, and its most popular splitting rules are also explained.

Colima, México — Osval Antonio Montesinos López
Guadalajara, México — Abelardo Montesinos López
Texcoco, Estado de México, México — José Crossa

# Acknowledgments

The work described in this book is the result of more than 20 years of research on statistical and quantitative genetic sciences for developing models and methods for improving genome-enabled predictions. Most of the data used in this study was produced and provided by CIMMYT and diverse research partners. We are inspired and grateful for their tremendous commitment to advancing and applying science for public good and specially for improvising the livelihood of the low resource farmers.

The authors are grateful to the past and present CIMMYT Directors General, Deputy Directors of Research, Deputy Directors of Administration, Directors of Research Programs, and other Administration offices and Laboratories of CIMMYT for their continuous and firm support of biometrical genetics, and statistics research, training, and service in support of CIMMYT's mission: "maize and wheat science for improved livelihoods."

This work was made possible with support from the CGIAR Research Programs on Wheat and Maize (wheat.org, maize.org), and many funders including Australia, United Kingdom (DFID), USA (USAID), South Africa, China, Mexico (SAGARPA), Canada, India, Korea, Norway, Switzerland, France, Japan, New Zealand, Sweden, and the World Bank. We thank the financial support of the Mexico Government through out MASAGRO and several other regional projects and close collaboration with numerous Mexican researchers.

We acknowledge the financial support provided by the (1) Bill and Melinda Gates Foundation (INV-003439 BMGF/FCDO Accelerating Genetic Gains in Maize and Wheat for Improved Livelihoods [AG2MW]) as well as (2) USAID projects (Amend. No. 9 MTO 069033, USAID-CIMMYT Wheat/AGGMW, AGG-Maize Supplementary Project, AGG [Stress Tolerant Maize for Africa]).

Very special recognition is given to Bill and Melinda Gates Foundation for providing the Open Access fee of this book.

We are also thankful for the financial support provided by the (1) Foundations for Research Levy on Agricultural Products (FFL) and the Agricultural Agreement Research Fund (JA) in Norway through NFR grant 267806, (2) Sveriges Llantbruksuniversitet (Swedish University of Agricultural Sciences) Department of

Plant Breeding, Sundsvägen 10, 23053 Alnarp, Sweden, (3) CIMMYT CRP, (4) the Consejo Nacional de Tecnología y Ciencia (CONACYT) of México, (5) Universidad de Colima of Mexico, and (6) Universidad de Guadalajara of Mexico.

We highly appreciate and thank the several students at the Universidad de Colima, students and professors from the Colegio de Post-Graduados and the Universidad de Guadalajara who tested and made suggestions on early version of the material covered in the book; their many useful suggestions had a direct impact on the current organization and the technical content of the book.

Finally, we wish to express our deep thanks to Prof. Dr. Fred van Eeuwijk, who carefully read, corrected, modified, and suggested changes for each of the chapter of this book. We introduced many important suggestions and additions based on Prof. Dr. Fred van Eeuwijk's extensive scientific knowledge.

# Contents

**1 General Elements of Genomic Selection and Statistical Learning** . . . . . 1
1.1 Data as a Powerful Weapon . . . . . 1
1.2 Genomic Selection . . . . . 3
1.2.1 Concepts of Genomic Selection . . . . . 4
1.2.2 Why Is Statistical Machine Learning a Key Element of Genomic Selection? . . . . . 6
1.3 Modeling Basics . . . . . 8
1.3.1 What Is a Statistical Machine Learning Model? . . . . . 8
1.3.2 The Two Cultures of Model Building: Prediction Versus Inference . . . . . 9
1.3.3 Types of Statistical Machine Learning Models and Model Effects . . . . . 11
1.4 Matrix Algebra Review . . . . . 17
1.5 Statistical Data Types . . . . . 25
1.5.1 Data Types . . . . . 25
1.5.2 Multivariate Data Types . . . . . 28
1.6 Types of Learning . . . . . 28
1.6.1 Definition and Examples of Supervised Learning . . . . . 29
1.6.2 Definitions and Examples of Unsupervised Learning . . . . . 32
1.6.3 Definition and Examples of Semi-Supervised Learning . . . . . 33
References . . . . . 33

**2 Preprocessing Tools for Data Preparation** . . . . . 35
2.1 Fixed or Random Effects . . . . . 35
2.2 BLUEs and BLUPs . . . . . 36
2.3 Marker Depuration . . . . . 43
2.4 Methods to Compute the Genomic Relationship Matrix . . . . . 49

2.5 Genomic Breeding Values and Their Estimation . . . . . . . . . . . 52
2.6 Normalization Methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
2.7 General Suggestions for Removing or Adding Inputs . . . . . . . . 58
2.8 Principal Component Analysis as a Compression Method . . . . 63
Appendix 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
Appendix 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69

**3 Elements for Building Supervised Statistical Machine Learning Models** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
3.1 Definition of a Linear Multiple Regression Model . . . . . . . . . . 71
3.2 Fitting a Linear Multiple Regression Model via the Ordinary Least Square (OLS) Method . . . . . . . . . . . . . 71
3.3 Fitting the Linear Multiple Regression Model via the Maximum Likelihood (ML) Method . . . . . . . . . . . . . . 75
3.4 Fitting the Linear Multiple Regression Model via the Gradient Descent (GD) Method . . . . . . . . . . . . . . . . . . 76
3.5 Advantages and Disadvantages of Standard Linear Regression Models (OLS and MLR) . . . . . . . . . . . . . . . . . . . . 80
3.6 Regularized Linear Multiple Regression Model . . . . . . . . . . . . 81
3.6.1 Ridge Regression . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
3.6.2 Lasso Regression . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
3.7 Logistic Regression . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 98
3.7.1 Logistic Ridge Regression . . . . . . . . . . . . . . . . . . . . . . 100
3.7.2 Lasso Logistic Regression . . . . . . . . . . . . . . . . . . . . . . 102
Appendix 1: R Code for Ridge Regression Used in Example 2 . . . . . . . 104
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107

**4 Overfitting, Model Tuning, and Evaluation of Prediction Performance** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
4.1 The Problem of Overfitting and Underfitting . . . . . . . . . . . . . . 109
4.2 The Trade-Off Between Prediction Accuracy and Model Interpretability . . . . . . . . . . . . . . . . . . . . . . . . . . 111
4.3 Cross-validation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115
4.3.1 The Single Hold-Out Set Approach . . . . . . . . . . . . . . . 115
4.3.2 The *k*-Fold Cross-validation . . . . . . . . . . . . . . . . . . . . 116
4.3.3 The Leave-One-Out Cross-validation . . . . . . . . . . . . . . 117
4.3.4 The Leave-m-Out Cross-validation . . . . . . . . . . . . . . . . 117
4.3.5 Random Cross-validation . . . . . . . . . . . . . . . . . . . . . . 118
4.3.6 The Leave-One-Group-Out Cross-validation . . . . . . . . . 118
4.3.7 Bootstrap Cross-validation . . . . . . . . . . . . . . . . . . . . . 119
4.3.8 Incomplete Block Cross-validation . . . . . . . . . . . . . . . 120
4.3.9 Random Cross-validation with Blocks . . . . . . . . . . . . . 121
4.3.10 Other Options and General Comments on Cross-validation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122

4.4 Model Tuning . . . 124
4.4.1 Why Is Model Tuning Important? . . . 126
4.4.2 Methods for Hyperparameter Tuning (Grid Search, Random Search, etc.) . . . 127
4.5 Metrics for the Evaluation of Prediction Performance . . . 128
4.5.1 Quantitative Measures of Prediction Performance . . . 129
4.5.2 Binary and Ordinal Measures of Prediction Performance . . . 131
4.5.3 Count Measures of Prediction Performance . . . 137
References . . . 138

**5 Linear Mixed Models** . . . 141
5.1 General of Linear Mixed Models . . . 141
5.2 Estimation of the Linear Mixed Model . . . 142
5.2.1 Maximum Likelihood Estimation . . . 142
5.3 Linear Mixed Models in Genomic Prediction . . . 148
5.4 Illustrative Examples of the Univariate LMM . . . 148
5.5 Multi-trait Genomic Linear Mixed-Effects Models . . . 152
5.6 Final Comments . . . 157
Appendix 1 . . . 158
Appendix 2 . . . 158
Appendix 3 . . . 159
Appendix 4 . . . 159
Appendix 5 . . . 160
Appendix 6 . . . 163
Appendix 7 . . . 165
References . . . 168

**6 Bayesian Genomic Linear Regression** . . . 171
6.1 Bayes Theorem and Bayesian Linear Regression . . . 171
6.2 Bayesian Genome-Based Ridge Regression . . . 172
6.3 Bayesian GBLUP Genomic Model . . . 176
6.4 Genomic-Enabled Prediction BayesA Model . . . 178
6.5 Genomic-Enabled Prediction BayesB and BayesC Models . . . 180
6.6 Genomic-Enabled Prediction Bayesian Lasso Model . . . 184
6.7 Extended Predictor in Bayesian Genomic Regression Models . . . 186
6.8 Bayesian Genomic Multi-trait Linear Regression Model . . . 188
6.8.1 Genomic Multi-trait Linear Model . . . 190
6.9 Bayesian Genomic Multi-trait and Multi-environment Model (BMTME) . . . 195
Appendix 1 . . . 198
Appendix 2: Setting Hyperparameters for the Prior Distributions of the BRR Model . . . 199
Appendix 3: R Code Example 1 . . . 200

Appendix 4: R Code Example 2 . . . 202
Appendix 5 . . . 204
R Code Example 3 . . . 204
R Code for Example 4 . . . 206
References . . . 207

**7 Bayesian and Classical Prediction Models for Categorical and Count Data . . . 209**
7.1 Introduction . . . 209
7.2 Bayesian Ordinal Regression Model . . . 209
7.2.1 Illustrative Examples . . . 216
7.3 Ordinal Logistic Regression . . . 221
7.4 Penalized Multinomial Logistic Regression . . . 225
7.4.1 Illustrative Examples for Multinomial Penalized Logistic Regression . . . 228
7.5 Penalized Poisson Regression . . . 232
7.6 Final Comments . . . 235
Appendix 1 . . . 236
Appendix 2 . . . 238
Appendix 3 . . . 240
Appendix 4 (Example 4) . . . 242
Appendix 5 . . . 244
Appendix 6 . . . 246
References . . . 248

**8 Reproducing Kernel Hilbert Spaces Regression and Classification Methods . . . 251**
8.1 The Reproducing Kernel Hilbert Spaces (RKHS) . . . 251
8.2 Generalized Kernel Model . . . 253
8.2.1 Parameter Estimation Under the Frequentist Paradigm . . . 253
8.2.2 Kernels . . . 255
8.2.3 Kernel Trick . . . 257
8.2.4 Popular Kernel Functions . . . 260
8.2.5 A Two Separate Step Process for Building Kernels . . . 269
8.3 Kernel Methods for Gaussian Response Variables . . . 269
8.4 Kernel Methods for Binary Response Variables . . . 271
8.5 Kernel Methods for Categorical Response Variables . . . 274
8.6 The Linear Mixed Model with Kernels . . . 274
8.7 Hyperparameter Tuning for Building the Kernels . . . 278
8.8 Bayesian Kernel Methods . . . 280
8.8.1 Extended Predictor Under the Bayesian Kernel BLUP . . . 283
8.8.2 Extended Predictor Under the Bayesian Kernel BLUP with a Binary Response Variable . . . 286

8.8.3 Extended Predictor Under the Bayesian Kernel BLUP with a Categorical Response Variable . . . . . . . 287
8.9 Multi-trait Bayesian Kernel . . . . . . . . . . . . . . . . . . . . . . . . . 288
8.10 Kernel Compression Methods . . . . . . . . . . . . . . . . . . . . . . . 289
8.10.1 Extended Predictor Under the Approximate Kernel Method . . . . . . . . . . . . . . . . . . . . . . . . . . . 294
8.11 Final Comments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 297
Appendix 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 298
Appendix 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 301
Appendix 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 305
Appendix 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 306
Appendix 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 308
Appendix 6 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 311
Appendix 7 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 316
Appendix 8 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 320
Appendix 9 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 325
Appendix 10 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 329
Appendix 11 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 331
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 334

**9 Support Vector Machines and Support Vector Regression . . . . . . . 337**
9.1 Introduction to Support Vector Machine . . . . . . . . . . . . . . . . . 337
9.2 Hyperplane . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 338
9.3 Maximum Margin Classifier . . . . . . . . . . . . . . . . . . . . . . . . . 340
9.3.1 Derivation of the Maximum Margin Classifier . . . . . . 343
9.3.2 Wolfe Dual . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 346
9.4 Derivation of the Support Vector Classifier . . . . . . . . . . . . . . 354
9.5 Support Vector Machine . . . . . . . . . . . . . . . . . . . . . . . . . . . 357
9.5.1 One-Versus-One Classification . . . . . . . . . . . . . . . . 361
9.5.2 One-Versus-All Classification . . . . . . . . . . . . . . . . . 361
9.6 Support Vector Regression . . . . . . . . . . . . . . . . . . . . . . . . . . 369
Appendix 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 371
Appendix 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 373
Appendix 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 375
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 377

**10 Fundamentals of Artificial Neural Networks and Deep Learning . . 379**
10.1 The Inspiration for the Neural Network Model . . . . . . . . . . . . 379
10.2 The Building Blocks of Artificial Neural Networks . . . . . . . . . 382
10.3 Activation Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 387
10.3.1 Linear . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 388
10.3.2 Rectifier Linear Unit (ReLU) . . . . . . . . . . . . . . . . . . 388
10.3.3 Leaky ReLU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 389
10.3.4 Sigmoid . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 390
10.3.5 Softmax . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 390
10.3.6 Tanh . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 391

10.4 The Universal Approximation Theorem . . . . . . . . . . . . . . . . . 392
10.5 Artificial Neural Network Topologies . . . . . . . . . . . . . . . . . . . 393
10.6 Successful Applications of ANN and DL . . . . . . . . . . . . . . . . 396
10.7 Loss Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 399
10.7.1 Loss Functions for Continuous Outcomes . . . . . . . . . 400
10.7.2 Loss Functions for Binary and Ordinal Outcomes . . . . 401
10.7.3 Regularized Loss Functions . . . . . . . . . . . . . . . . . . . . 402
10.7.4 Early Stopping Method of Training . . . . . . . . . . . . . . 405
10.8 The King Algorithm for Training Artificial Neural Networks: Backpropagation . . . . . . . . . . . . . . . . . . . . . . . . . 407
10.8.1 Backpropagation Algorithm: Online Version . . . . . . . 412
10.8.2 Illustrative Example 10.1: A Hand Computation . . . . . 413
10.8.3 Illustrative Example 10.2—By Hand Computation . . . 418
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 424

**11 Artificial Neural Networks and Deep Learning for Genomic Prediction of Continuous Outcomes** . . . . . . . . . . . . . . . . . . . . . . . 427
11.1 Hyperparameters to Be Tuned in ANN and DL . . . . . . . . . . . . 427
11.1.1 Network Topology . . . . . . . . . . . . . . . . . . . . . . . . . . 427
11.1.2 Activation Functions . . . . . . . . . . . . . . . . . . . . . . . . . 428
11.1.3 Loss Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 428
11.1.4 Number of Hidden Layers . . . . . . . . . . . . . . . . . . . . . 428
11.1.5 Number of Neurons in Each Layer . . . . . . . . . . . . . . 430
11.1.6 Regularization Type . . . . . . . . . . . . . . . . . . . . . . . . . 431
11.1.7 Learning Rate . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 432
11.1.8 Number of Epochs and Number of Batches . . . . . . . . 433
11.1.9 Normalization Scheme for Input Data . . . . . . . . . . . . 434
11.2 Popular DL Frameworks . . . . . . . . . . . . . . . . . . . . . . . . . . . . 435
11.3 Optimizers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 436
11.4 Illustrative Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 438
Appendix 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 459
Appendix 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 462
Appendix 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 466
Appendix 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 467
Appendix 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 471
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 476

**12 Artificial Neural Networks and Deep Learning for Genomic Prediction of Binary, Ordinal, and Mixed Outcomes** . . . . . . . . . . . . 477
12.1 Training DNN with Binary Outcomes . . . . . . . . . . . . . . . . . . . 477
12.2 Training DNN with Categorical (Ordinal) Outcomes . . . . . . . . 482
12.3 Training DNN with Count Outcomes . . . . . . . . . . . . . . . . . . . 486
12.4 Training DNN with Multivariate Outcomes . . . . . . . . . . . . . . . 490
12.4.1 DNN with Multivariate Continuous Outcomes . . . . . . 490
12.4.2 DNN with Multivariate Binary Outcomes . . . . . . . . . . 493

12.4.3 DNN with Multivariate Ordinal Outcomes . . . . . . . . . 498
12.4.4 DNN with Multivariate Count Outcomes . . . . . . . . . . 501
12.4.5 DNN with Multivariate Mixed Outcomes . . . . . . . . . . 504
Appendix 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 507
Appendix 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 512
Appendix 3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 517
Appendix 4 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 521
Appendix 5 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 526
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 531

**13 Convolutional Neural Networks** . . . . . . . . . . . . . . . . . . . . . . 533
13.1 The Importance of Convolutional Neural Networks . . . . . . . . . 533
13.2 Tensors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 534
13.3 Convolution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 539
13.4 Pooling . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 542
13.5 Convolutional Operation for 1D Tensor for Sequence Data . . . 545
13.6 Motivation of CNN . . . . . . . . . . . . . . . . . . . . . . . . . . 546
13.7 Why Are CNNs Preferred over Feedforward Deep Neural Networks for Processing Images? . . . . . . . . . . . . . . . . . . . . . 548
13.8 Illustrative Examples . . . . . . . . . . . . . . . . . . . . . . . . . 554
13.9 2D Convolution Example . . . . . . . . . . . . . . . . . . . . . . . 560
13.10 Critics of Deep Learning . . . . . . . . . . . . . . . . . . . . . . . 566
Appendix 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 568
Appendix 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 572
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 576

**14 Functional Regression** . . . . . . . . . . . . . . . . . . . . . . . . . . 579
14.1 Principles of Functional Linear Regression Analyses . . . . . . . . 579
14.2 Basis Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 584
14.2.1 Fourier Basis . . . . . . . . . . . . . . . . . . . . . . . . . 584
14.2.2 B-Spline Basis . . . . . . . . . . . . . . . . . . . . . . . . 585
14.3 Illustrative Examples . . . . . . . . . . . . . . . . . . . . . . . . . 589
14.4 Functional Regression with a Smoothed Coefficient Function . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 598
14.5 Bayesian Estimation of the Functional Regression . . . . . . . . . . 604
Appendix 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 611
Appendix 2 (Example 14.4) . . . . . . . . . . . . . . . . . . . . . . . . 617
Appendix 3 (Example 14.5) . . . . . . . . . . . . . . . . . . . . . . . . 621
Appendix 4 (Example 14.6) . . . . . . . . . . . . . . . . . . . . . . . . 626
References . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 631

**15 Random Forest for Genomic Prediction** . . . . . . . . . . . . . . . . . 633
15.1 Motivation of Random Forest . . . . . . . . . . . . . . . . . . . . . 633
15.2 Decision Trees . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 634
15.3 Random Forest . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 637

15.4 RF Algorithm for Continuous, Binary, and Categorical Response Variables . . . . . 639
15.4.1 Splitting Rules . . . . . 641
15.5 RF Algorithm for Count Response Variables . . . . . 650
15.6 RF Algorithm for Multivariate Response Variables . . . . . 655
15.7 Final Comments . . . . . 660
Appendix 1 . . . . . 662
Appendix 2 . . . . . 664
Appendix 3 . . . . . 665
Appendix 4 . . . . . 669
Appendix 5 . . . . . 672
Appendix 6 . . . . . 676
References . . . . . 680

**Index** . . . . . 683

# Chapter 1
# General Elements of Genomic Selection and Statistical Learning

## 1.1 Data as a Powerful Weapon

Thanks to advances in digital technologies like electronic devices and networks, it is possible to automatize and digitalize many jobs, processes, and services, which are generating huge quantities of data. These "big data" are transmitted, collected, aggregated, and analyzed to deliver deep insights into processes and human behavior. For this reason, data are called the new oil, since "data are to this century what oil was for the last century"—that is, a driver for change, growth, and success. While statistical and machine learning algorithms extract information from raw data, information can be used to create knowledge, knowledge leads to understanding, and understanding leads to wisdom (Sejnowski 2018). We have the tools and expertise to collect data from diverse sources and in any format, which is the cornerstone of a modern data strategy that can unleash the power of artificial intelligence. Every single day we are creating around 2.5 quintillion bytes of data (McKinsey Global Institute 2016). This means that almost 90% of the data in the world has been generated over the last 2 years. This unprecedented capacity to generate data has increased connectivity and global data flows through numerous sources like tweets, YouTube, blogs, sensors, internet, Google, emails, pictures, etc. For example, Google processes more than 40,000 searches every second (and 3.5 billion searches per day), 456,000 tweets are sent, and 4,146,600 YouTube videos are watched per minute, and every minute, 154,200 Skype calls are made, 156 million emails are sent, 16 million text messages are written, etc. In other words, the amount of data is becoming bigger and bigger (big data) day by day in terms of volume, velocity, variety, veracity, and "value."

The nature of international trade is being radically transformed by global data flows, which are creating new opportunities for businesses to participate in the global economy. The following are some ways that these data flows are transforming international trade: (a) businesses can use the internet (i.e., digital platforms) to export goods; (b) services can be purchased and consumed online; (c) data collection

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_1

and analysis are allowing new services (often also provided online) to add value to exported goods; and (d) global data flows underpin global value chains, creating new opportunities for participation.

According to estimates, by 2020, 15–20% of the gross domestic product (GDP) of countries all over the world will be based on data flows. Companies that adopt big data practices are expected to increase their productivity by 5–10% compared to companies that do not, and big data practices could add 1.9% to Europe's GDP between 2014 and 2020. According to McKinsey Global Institute (2016) estimates, big data could generate an additional $3 trillion in value every year in some industries. Of this, $1.3 trillion would benefit the United States. Although these benefits do not directly affect the GDP or people's personal income, they indirectly help to improve the quality of life.

But once we have collected a large amount of data, the next question is: how to make sense of it? A lot of businesses and people are collecting data, but few are extracting useful knowledge from them. As mentioned above, nowadays there have been significant advances for measuring and collecting data; however, obtaining knowledge from (making sense of) these collected data is still very hard and challenging, since there are few people in the market with the expertise and training needed to extract knowledge from huge amounts of data. For this reason, to make sense of data, new disciplines were recently created, such as data science (the commercial name of statistics), business intelligence and analytics, that use a combination of statistics, computing, machine learning, and business, among other domains, to analyze and discover useful patterns to improve the decision-making process. In general, these tools help to (1) rationalize decisions and the decision-making process, (2) minimize errors and deal with a range of scenarios, (3) get a better understanding of the situation at hand (due to the availability of historical data), (4) assess alternative solutions (based on key performance indicators), and (5) map them against the best possible outcome (benchmarking), which helps make decision-making more agile. For this reason, data analysis has been called "the sexiest job of the twenty-first century." For example, big data jobs in the United States are estimated to be 500,000. But the McKinsey Global Institute (2016) estimates that there is still a shortage of between 140,000 and 190,000 workers with a background in statistics, computer engineering, and other applied fields, and that 1.5 million managers are needed to evaluate and make decisions on big data. This means that 50–60% of the required staff was lacking in the year 2018 in the United States alone (Dean 2018).

Some areas and cases where statistical and machine learning techniques have been successfully applied to create knowledge and make sense of data are given next. For example, in astronomy these techniques have been used for the classification of exoplanets using thousands of images taken of these celestial bodies. Banks use them to decide if a client will be granted credit or not, using as predictors many socioeconomic variables they ask of clients. Banks also use them to detect credit card fraud. On the internet, they are used to classify emails as spam or ham (not spam) based on previous emails and the text they contain. In genomic selection, they are used to predict grain yield (or another trait) of non-phenotyped plants using

information, including thousands of markers and environmental data. In Google, they are used for recommending books, movies, products, etc., using previous data on the characteristics (age, gender, location, etc.) of the people who have used these services. They are also used in self-driving cars, that is, cars capable of sensing their environment and moving with little or no human input, based on thousands of images and information from sensors that perceive their surroundings. These examples give more evidence that the appropriate use of data is a powerful weapon for getting knowledge of the target population.

However, as is true of any new technology, this data-related technology can be used against society. One example is the Facebook–Cambridge Analytica data scandal that was a major political scandal in early 2018 when it was revealed that Cambridge Analytica had harvested personal data from the Facebook profiles of millions of people without their consent and used them for political purposes. This scandal was described as a watershed moment in the public understanding of personal data and precipitated a massive fall in Facebook's stock price and calls for tighter regulation of tech companies' use of data. For these reasons, some experts believe that governments have a responsibility to create and enforce rules on data privacy, since data are a powerful weapon, and weapons should be controlled, and because privacy is a fundamental human right. All these are very important to avoid the weaponization of data against people and society.

To take advantage of the massive data collected in Genomic Selection (GS) and many other domains, it is really important to train people in statistical machine learning methods and related areas to perform precise prediction, extract useful knowledge, and find hidden data patterns. This means that experts in statistical machine learning methods should be able to identify the statistical or machine learning method that is most relevant to a given problem, since there is no universal method that works well for all data sets, cleans the original data, implements these methods in statistical machine learning software, and interprets the output of statistical machine learning methods correctly to translate the big data collected into insights and operational value quickly and accurately.

## 1.2 Genomic Selection

Plant breeding is a key scientific area for increasing the food production required to feed the people of our planet. The key step in plant breeding is selection, and conventional breeding is based on phenotypic selection. Breeders choose good offspring using their experience and the observed phenotypes of crops, so as to achieve genetic improvement of target traits (Wang et al. 2018). Thanks to this area (and related areas of science), the genetic gain nowadays has reached a near-linear increase of 1% in grain yield yearly (Oury et al. 2012; Fischer et al. 2014). However, a linear increase of at least 2% is needed to cope with the 2% yearly increase in the world population, which relies heavily on wheat products as a source of food (FAO 2011). For this reason, genomic selection (GS) is now being implemented in many

plant breeding programs around the world. GS consists of genotyping (markers) and phenotyping individuals in the reference (training) population and, with the help of statistical machine learning models, predicting the phenotypes or breeding values of the candidates for selection in the testing (evaluation) population that were only genotyped. GS is revolutionizing plant breeding because it is not limited to traits determined by a few major genes and allows using a statistical machine learning model to establish the associations between markers and phenotypes and also to make predictions of non-phenotyped individuals that help make a more comprehensive and reliable selection of candidate individuals. In this way, it is essential for accelerating genetic progress in crop breeding (Montesinos-López et al. 2019).

### *1.2.1 Concepts of Genomic Selection*

The development of different molecular marker systems that started in the 1980s drastically increased the total number of polymorphic markers available to breeders and molecular biologists in general. The single nucleotide polymorphism (SNP) that has been intensively used in QTL discovery is perhaps the most popular high-throughput genotyping system (Crossa et al. 2017). Initially, by applying marker-assisted selection (MAS), molecular markers were integrated with traditional phenotypic selection. In the context of simple traits, MAS consists of selecting individuals with QTL-associated markers with major effects; markers not significantly associated with a trait are not used (Crossa et al. 2017). However, after many attempts to improve complex quantitative traits by using QTL-associated markers, there is not enough evidence that this method really can be helpful in practical breeding programs due to the difficulty of finding the same QTL across multiple environments (due to QTL $\times$ environment interaction) or in different genetic backgrounds (Bernardo 2016). Due to this difficulty of the MAS approach, in the early 2000s, an approach called association mapping appeared with the purpose of overcoming the insufficient power of linkage analysis, thus facilitating the detection of marker–trait associations in non-biparental populations and fine-mapping chromosome segments with high recombination rates (Crossa et al. 2017). However, even the fine-mapping approach was unable to increase the power to detect rare variants that may be associated with economically important traits.

For this reason, Meuwissen et al. (2001) proposed the GS methodology (that was initially used in animal science), which is different from association mapping and QTL analysis, since GS simultaneously uses all the molecular markers available in a training data set for building a prediction model; then, with the output of the trained model, predictions are performed for new candidate individuals not included in the training data set, but only if genotypic information is available for those candidate individuals. This means that the goal of GS is to predict breeding and/or genetic values. Because GS is implemented in a two-stage process, to successfully implement it, the data must be divided into a training (TRN) and a testing (TST) set, as can be observed in Fig. 1.1. The training set is used in the first stage, while the testing set

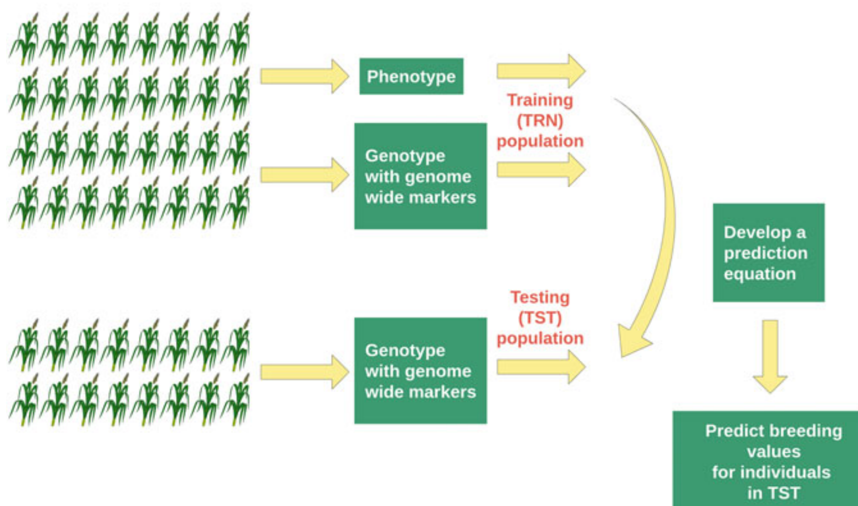



**Fig. 1.1** Schematic representation of TRN and TST sets required for implementing GS (Crossa et al. 2017)

is used in the second stage. The main characteristics of the training set are (a) it combines molecular (independent variables) and phenotypic (dependent variables) data and (b) it contains enough observations (lines) and predictors (molecular data) to be able to train a statistical machine learning model with high generalized power (able to predict data not used in the training process) to predict new lines. The main characteristic of the testing set is that it only contains genotypic data (markers) for a sample of observations (lines) and the goal is to predict the phenotypic or breeding values of lines that have been genotyped but not phenotyped.

The two basic populations in a GS program are shown in Fig. 1.1: the training (TRN) data whose genotype and phenotype are known and the testing (TST) data whose phenotypic values are to be predicted using their genotypic information. GS substitutes phenotyping for a few selection cycles. Some advantages of GS over traditional (phenotypic) selection are that it: (a) reduces costs, in part by saving the resources required for extensive phenotyping, (b) saves time needed for variety development by reducing the cycle length, (c) has the ability to substantially increase the selection intensity, thus providing scenarios for capturing greater gain per unit time, (d) makes it possible to select traits that are very difficult to measure, and (e) can improve the accuracy of the selection process. Of course, successful implementation of GS depends strongly on the quality of the training and testing sets.

GS has great potential for quickly improving complex traits with low heritability, as well as significantly reducing the cost of line and hybrid development. Certainly, GS could also be employed for simple traits with higher heritability than complex traits, and high genomic prediction (GP) accuracy is expected. Application of GS in

plant breeding could be limited by some of the following factors: (i) genotyping cost, (ii) unclear guidelines about where in the breeding program GS could be efficiently applied (Crossa et al. 2017), (iii) insufficient number of lines or animals in the reference (training) population, (iv) insufficient number of SNPs in the panels, and (v) the reference population contains very heterogeneous individuals (plants or animals).

### *1.2.2 Why Is Statistical Machine Learning a Key Element of Genomic Selection?*

GS is challenging and very interesting because it aims to improve crop productivity to satisfy humankind's need for food. Addressing the current challenges to increase crop productivity by improving the genetic makeup of plants and avoiding plant diseases is not new, but it is of paramount importance today to be able to increase crop productivity around the world without the need to increase the arable land. Statistical machine learning methods can help improve GS methodology, since they are able to make computers learn patterns that could be used for analysis, interpretation, prediction, and decision-making. These methods learn the relationships between the predictors and the target output using statistical and mathematical models that are implemented using computational tools to be able to predict (or explain) one or more dependent variables based on one or more independent variables in an efficient manner. However, to do this successfully, many real-world problems are only approximated using the statistical machine learning tools, by evaluating probabilistic distributions, and the decisions made using these models are supported by indicators like confidence intervals. However, the creation of models using probability distributions and indicators for evaluating prediction (or association) performance is a field of statistical machine learning, which is a branch of artificial intelligence, understanding as statistical machine learning the application of statistical methods to identify patterns in data using computers, but giving computers the ability to learn without being explicitly programmed (Samuel 1959). However, artificial intelligence is the field of science that creates machines or devices that can mimic intelligent behaviors.

As mentioned above, statistical machine learning allows learning the relationship between two types of information that are assumed to be related. Then one part of the information (input or independent variables) can be used to predict the information lacking (output or dependent variables) in the other using the learned relationship. The information we want to predict is defined as the response variable ($y$), while the information we use as input are the predictor variables ($\boldsymbol{X}$). Thanks to the continuous reduction in the cost of genotyping, GS nowadays is implemented in many crops around the world, which has caused the accumulation of large amounts of biological data that can be used for prediction of non-phenotyped plants and animals. However, GS implementation is still challenging, since the quality of the data (phenotypic and

genotypic) needs to be improved. Many times the genotypic information available is not enough to make high-quality predictions of the target trait, since the information available has a lot of noise. Also, since there is no universal best prediction model that can be used under all circumstances, a good understanding of statistical machine learning models is required to increase the efficiency of the selection process of the best candidate individuals with GS early in time. This is very important because one of the key components of genomic selection is the use of statistical machine learning models for the prediction of non-phenotyped individuals. For this reason, statistical machine learning tools have the power to help increase the potential of GS if more powerful statistical machine learning methods are developed, if the existing methods can deal with larger data sets, and if these methods can be automatized to perform the prediction process with only a limited knowledge of the subject.

For these reasons, statistical machine learning tools promise considerable benefits for GS and agriculture through their contribution to productivity growth and gain in the genetic makeup of plants and animals without the need to increase the arable land. At the same time, with the help of statistical machine learning tools, GS is deeply impacting the traditional way of selecting candidate individuals in plant and animal breeding. Since GS can reduce by at least half the time needed to select candidate individuals, it has been implemented in many crops around the globe and is radically changing the traditional way of developing new varieties and animals all over the world.

Although GS is not the dominant paradigm for developing new plants and animals, it has the potential to transform the way they are developed due to the following facts: (a) the massive amounts of data being generated in plant breeding programs are now available to train the statistical machine learning methods, (b) new technologies such as sensors, satellite technology, and robotics allow scientists to generate not only genomic data but also phenotypic data that can capture a lot of environmental and phenotypic information that can be used in the modeling process to increase the performance of statistical machine learning methods, (c) increased computational power now allows complex statistical machine learning models with larger data sets to be implemented in less time, and (d) there is now greater availability of user-friendly statistical machine learning software for implementing a great variety of statistical machine learning models.

However, there are still limitations for the successful implementation of GS with the help of statistical machine learning methods because much human effort is required to collect a good training data set for supervised learning. Although nowadays it is possible to measure a lot of independent variables (markers, environmental variables) due to the fact that the training set should be measured in real-world experiments conducted in different environments and locations, this is expensive and subject to nature's random variability. This means that GS data are hampered by issues such as multicollinearity among markers (adjacent markers are highly correlated) and by a problem that consists of having a small number of observations and a large number of independent variables (commonly known as "large $p$ small $n$"), which poses a statistical challenge. For this reason, obtaining data sets that are large and comprehensive enough to be used for training—for example,

creating or obtaining sufficient plant trial data to predict yield, plant height, grain quality, and presence or absence of disease outcomes more accurately—is also often challenging.

Another challenge is that of building statistical machine learning techniques that are able to generalize the unseen data, since statistical machine learning methods continue to have difficulty carrying their experiences from one set of circumstances to another. This is known as transfer learning, and it focuses on storing knowledge gained when training a particular machine learning algorithm and then using this stored knowledge for solving another related problem. In other words, transfer learning is still very challenging and occurs when a statistical machine learning model is trained to accomplish a certain task and then quickly apply that learning exercise to a different activity.

Another disadvantage is that even though today there are many software programs for implementing statistical machine learning tools for GS, the computational resources required for learning from moderate to large data sets are very expensive and most of the time it is not possible to implement them in commonly used computers, since servers with many cores and considerable computational resources are required. However, the rapid increase in computational power will change this situation in the coming years.

## 1.3 Modeling Basics

### *1.3.1 What Is a Statistical Machine Learning Model?*

A model is a simplified description, using mathematical tools, of the processes we think that give rise to the observations in a set of data. A model is deterministic if it explains (completely) the dependent variables based on the independent ones. In many real-world scenarios, this is not possible. Instead, statistical (or stochastic) models try to approximate exact solutions by evaluating probabilistic distributions. For this reason, a statistical model is expressed by an equation composed of a *systematic* (deterministic) and a *random part* (Stroup 2012) as given in the next equation:

$$y_i = f(\boldsymbol{x_i}) + \epsilon_i, \text{for } i = 1, 2, \ldots, n, \tag{1.1}$$

where $y_i$ represents the response variable in individual $i$ and $f(\boldsymbol{x}_i)$ is the systematic part of the model because it is *determined* by the explanatory variables (predictors). For these reasons, the systematic part of the statistical learning model is also called the deterministic part of the model, which gives rise to an unknown mathematical function ($f$) of $\boldsymbol{x}_i = x_{i1}, \ldots, x_{ip}$ not subject to random variability. $\epsilon_i$ is the $i$th random element (error term) which is independent of $\boldsymbol{x_i}$ and has mean zero. The $\epsilon_i$ term tells us that observations are assumed to vary at random about their mean, and it also defines the uniqueness of each individual. In theory (at least in some philosophical

domains), if we know the mechanism that gives rise to the uniqueness of each individual, we can write a completely deterministic model. However, this is rarely possible because we use probability distributions to characterize the observations measured in the individuals. Most of the time, the error term ($\epsilon_i$) is assumed to follow a normal distribution with mean zero and variance $\sigma^2$ (Stroup 2012).

As given in Eq. (1.1), the $f$ function that gives rise to the systematic part of a statistical learning model is not restricted to a unique input variable, but can be a function of many, or even thousands, of input variables. In general, the set of approaches for estimating $f$ is called statistical learning (James et al. 2013). Also, the functions that $f$ can take are very broad due to the huge variety of phenomena we want to predict and due to the fact that there is no universally superior $f$ that can be used for all processes. For this reason, to be able to perform good predictions out of sample data, many times we need to fit many models and then choose the one most likely to succeed with the help of cross-validation techniques. However, due to the fact that models are only a simplified picture of the true complex process that gives rise to the data at hand, many times it is very hard to find a good candidate model. For this reason, statistical machine learning provides a catalog of different models and algorithms from which we try to find the one that best fits our data, since there is no universally best model and because there is evidence that a set of assumptions that works well in one domain may work poorly in another—this is called the *no free lunch theorem* by Wolpert (1996). All these are in agreement with the famous aphorism, "all models are wrong, but some are useful," attributed to the British statistician George Box (October 18, 1919–March 28, 2013) who first mentioned this aphorism in his paper "Science and Statistics" published in the *Journal of the American Statistical Association* (Box 1976). As a result of the *no free lunch theorem*, we need to evaluate many models, algorithms, and sets of hyperparameters to find the best model in terms of prediction performance, speed of implementation, and degree of complexity. This book is concerned precisely with the appropriate combination of data, models, and algorithms needed to reach the best possible prediction performance.

### *1.3.2 The Two Cultures of Model Building: Prediction Versus Inference*

The term "two cultures" in statistical model building was coined by Breiman (2001) to explain the difference between the two goals for estimating $f$ in Eq. (1.1): prediction and inference. These definitions are provided in order to clarify the distinct scientific goals that follow inference and empirical predictions, respectively. A clear understanding and distinction between these two approaches is essential for the progress of scientific knowledge. Inference and predictive modeling reflect the process of using data and statistical (or data mining) methods for inferring or predicting, respectively. The term modeling is intentionally chosen over model to

highlight the entire process involved, from goal definition, study design, and data collection to scientific use (Breiman 2001).

**Prediction**

The prediction approach can be defined as the process of applying a statistical machine learning model or algorithm to data for the purpose of predicting new or future observations. For example, in plant breeding a set of inputs (marker information) and the outcome Y (disease resistance: yes or no) are available for some individuals, but for others only marker information is available. In this case, marker information can be used as a predictor and the disease status should be used as the response variable. When scientists are interested in predicting new plants not used to train the model, they simply want an accurate model to predict the response using the predictors. However, when scientists are interested in understanding the relationship between each individual predictor (marker) and the response variable, what they really want is a model for inference. Another example is when forest scientists are interested in developing models to predict the number of fire hotspots from an accumulated fuel dryness index, by vegetation type and region. In this context, it is obvious that scientists are interested in future predictions to improve decision-making in forest fire management. Another example is when an agro-industrial engineer is interested in developing an automated system for classifying mango species based on hundreds of mango images taken with digital cameras, mobile phones, etc. Here again it is clear that the best approach to build this system should be based on prediction modeling since the objective is the prediction of new mango species, not any of those used for training the model.

**Inference**

Many areas of science are devoted mainly to testing causal theories. Under this framework, scientists are interested in testing the validity of a theoretical causal relationship between the causal variables (X; underlying factors) and the measured variable (Y) using statistical machine learning models and collected data to test the causal hypotheses. The type of statistical machine learning models used for testing causal hypotheses are usually association-based models applied to observational data (Shmueli 2012). For example, regression models are one type of association-based models used for testing causal hypotheses. This practice is justified by the theory itself, which assumes the causality. In this context, the role of the theory is very strong and the reliance on data and statistical modeling is strictly through the lens of the theoretical model. The theory–data relationship varies in different fields. While the social sciences are very theory-heavy, in areas such as bioinformatics and natural language processing, the emphasis on a causal theory is much weaker. Hence, given this reality, Shmueli (2012) defined *explaining* as causal explanation and *explanatory modeling* as the use of statistical models for testing causal explanations.

Next, we provide some great examples used for testing causal hypotheses: for example, between 1911 and 1912, Austrian physicist Victor Hess made a series of ten balloon ascents to study why metal plates tend to charge spontaneously. At that

time, it was assumed that the cause was the presence, in small quantities, of radioactive materials in rocks. If this is the case, as one moves away from the ground, the tendency of metal plates to be charged should decrease. Hess brought with him three electroscopes, which are instruments composed basically of two metal plates enclosed in a glass sphere. When charging the plates, they separated one from the other. Hess observed that from a certain height, the three electroscopes tended to be charged to a greater extent. On August 7, 1912, together with a flight commander and a meteorologist, he made a 6-hour flight in which he ascended to more than 5000 m in height (Schuster 2014). Hess published his results in 1913, where he presented his conclusion that the cause of the charge of the electroscopes was radiation of cosmic origin that penetrates the atmosphere from above. The discovery of this cosmic radiation, for which Hess received the Nobel Prize in 1936, opened a new window for the study of the universe (Schuster 2014). It was in 1925 when American physicist Robert Andrew Millikan introduced the term "cosmic rays" to describe this radiation, and what it was made of was still unknown. In this example, it is clear that the goal of the analysis was association.

No one suspected that tobacco was a cause of lung tumors until the final decade of the nineteenth century. In 1898, Hermann Rottmann (a medical student) in Würzburg proposed that tobacco dust—not smoke—might be causing the elevated incidence of lung tumors among German tobacco workers. This was a mistake corrected by Adler (1912) who proposed that smoking might be to blame for the growing incidence of pulmonary tumors. Lung cancer was still a very rare disease; so rare, in fact, that medical professors, when confronted with a case, sometimes told their students they might never see another. However, in the 1920s, surgeons were already faced with a greater incidence of lung cancer, and they began to get confused about its possible causes. In general, smoking was blamed, along with asphalt dust from recently paved roads, industrial air pollution, and the latent effects of poisonous gas exposure during World War I or the global influenza pandemic of 1918–1919. These and many other theories were presented as possible explanations for the increase in lung cancer, until evidence from multiple research sources made it clear that tobacco was the main culprit (Proctor 2012). Here again it is clear that the goal of the analysis should be related to inference.

### *1.3.3 Types of Statistical Machine Learning Models and Model Effects*

#### 1.3.3.1 Types of Statistical Machine Learning Models

Statistical machine learning models are most commonly classified as parametric models, semiparametric models, and nonparametric models. Next, we define each type of statistical machine learning models and provide examples that help to understand each one.

***Parametric Model*** It is a type of statistical machine learning model in which all the predictors take predetermined forms with the response. Linear models (e.g., multiple regression: $y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \epsilon$), generalized linear models [Poisson regression: $E(y|x) = \exp(\beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3)$], and nonlinear models (nonlinear regression: $y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 e^{\beta_4 x_3} + \epsilon$) are examples of parametric statistical machine learning models because we know the function that describes the relationship between the response and the explanatory variables. These models are very easy to interpret but very inflexible.

***Nonparametric Model*** It is a type of statistical machine learning model in which none of the predictors take predetermined forms with the response but are constructed according to information derived from data. Two common statistical machine learning models are kernel regression and smoothing spline. Kernel regression estimates the conditional expectation of $y$ at a given value $x$ using a weighted filter on the data ($y = m(x) + \epsilon$, with $\widehat{m}(x_0) = \frac{\sum_{i=1}^{n} K\left(\frac{x_i - x_0}{h}\right) y_i}{\sum_{i=1}^{n} K\left(\frac{x_i - x_0}{h}\right)}$), where $h$ is the bandwidth (this estimator of $m(x)$ is called the *Nadaraya–Watson (NW) kernel estimator*) and $K$ is a kernel function. While smoothing splines minimize the sum of squared residuals plus a term which penalizes the roughness of the fit [$y = \beta_0 + \beta_1 x + \beta_2 x^2 + 3x^3 + \sum_{j=1}^{J} \beta_{1j} \left(x - \theta_j\right)_+^3$, where $(x - \theta_j)_+ = x - \theta_j$, $x > \theta_j$ and 0 otherwise], this model in brackets is a spline of degree 3 which is represented as a power series. These models are very difficult to interpret but are very flexible. Nonparametric statistical machine learning models differ from parametric models in that the shape of the functional relationships between the response (dependent) and the explanatory (independent) variables are not predetermined but can be adjusted to capture unusual or unexpected features of the data. Nonparametric statistical machine learning models can reduce modeling bias (the difference between estimated and true values) by imposing no specific model structure other than certain smoothness assumptions, and therefore they are particularly useful when we have little information or we want to be flexible about the underlying statistical machine learning model. In general, nonparametric statistical machine learning models are very flexible and are better at fitting the data than parametric statistical machine learning models. However, these models require larger samples than parametric statistical machine learning models because the data must supply the model structure as well as the model estimates.

***Semiparametric Model*** It is a statistical machine learning model in which *part* of the predictors do not take predetermined forms while the other part takes known forms with the response. Some examples are (a) $y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + m(x) + \epsilon$ and (b) $y = \exp(\beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3) + m(x) + \epsilon$. This means that semiparametric models are a mixture of parametric and nonparametric models.

When the relationship between the response and explanatory variables is known, parametric statistical machine learning models should be used. If the relationship is

unknown and nonlinear, nonparametric statistical machine learning models should be used. When we know the relationship between the response and part of the explanatory variables, but do not know the relationship between the response and the other part of the explanatory variables, we should use semiparametric statistical machine learning models. Any application area that uses statistical machine learning analysis could potentially benefit from semi/nonparametric regression.

#### 1.3.3.2 Model Effects

Many statistical machine learning models are expressed as models that incorporate *fixed effects*, which are parameters associated with an entire population or with certain levels of experimental factors of interest. Other models are expressed as *random effects*, where individual experimental units are drawn at random from a population, while a model with *fixed effects* and *random effects* is called a *mixed-effects* model (Pinheiro and Bates 2000).

According to Milliken and Johnson (2009), a factor is a *random effect* if its levels consist of a random sample of levels from a population of possible levels, while a factor is a *fixed effect* if its levels are selected by a nonrandom process or if its levels consist of the entire population of possible levels.

Mixed-effects models, also called multilevel models in the social science community (education, psychology, etc.), are an extension of regression models that allow for the incorporation of random effects; they are better suited to describe relationships between a response variable and some covariates in data that are grouped according to one or more classification factors. Examples of such grouped data include longitudinal data, repeated measures data, multilevel data, and block designs. One example of grouped data are animals that belong to the same herd; for example, assume we have 10 herds with 50 animals (observations) in each. By associating to observations (animals) sharing the same level of a classification factor (herd) a common random effect, mixed-effects models parsimoniously represent the covariance structure induced by the grouping of data (Pinheiro and Bates 2000). Most of the early work on mixed models was motivated by the animal science community driven by the need to incorporate heritabilities and genetic correlations in parsimonious fashion.

Next we provide an example to illustrate how to build these types of models. Assume that five environments were chosen at random from an agroecological area of Mexico. Then in each area, three replicates of a new variety (NV) of maize were tested to measure grain yield (GY) in tons per hectare. The data collected from this experiment are shown in Fig. 1.2.

Since the only factor that changes among the observations measured in this experiment is the environment, they are arranged in a one-way classification because they are classified according to a single characteristic: the environments in which the observations were made (Pinheiro and Bates 2000). The data structure is very simple since each row represents one observation for which the environment and GY were recorded, as can be seen in Table 1.1.

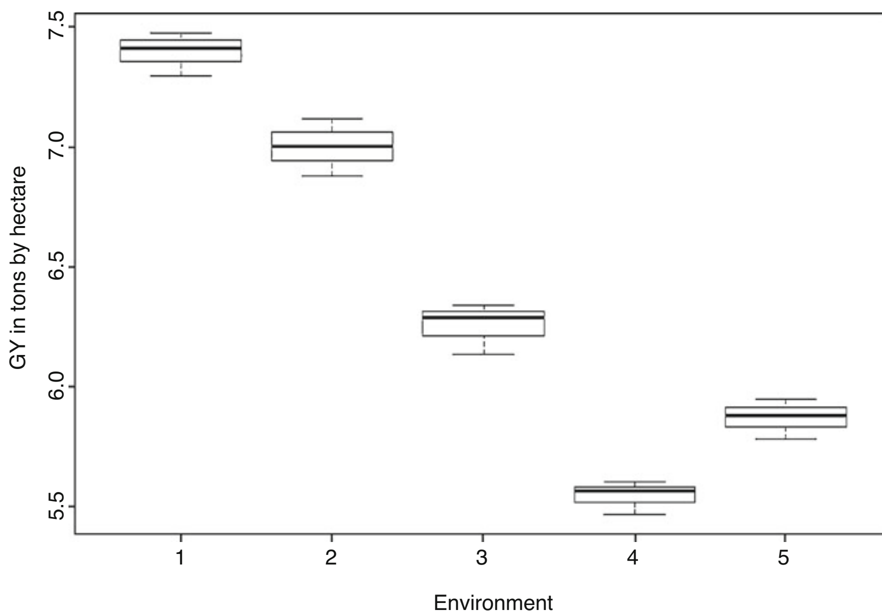



**Fig. 1.2** Grain yield (GY) in tons per hectare by environment

**Table 1.1** Grain yield (GY) measured in five environments (Env) with three repetitions (Rep) in each environment

| Env | Rep | GY |
|---|---|---|
| 1 | 1 | 7.476 |
| 1 | 2 | 7.298 |
| 1 | 3 | 7.414 |
| 2 | 1 | 7.117 |
| 2 | 2 | 6.878 |
| 2 | 3 | 7.004 |
| 3 | 1 | 6.136 |
| 3 | 2 | 6.340 |
| 3 | 3 | 6.288 |
| 4 | 1 | 5.600 |
| 4 | 2 | 5.564 |
| 4 | 3 | 5.466 |
| 5 | 1 | 5.780 |
| 5 | 2 | 5.948 |
| 5 | 3 | 5.881 |

The breeder who conducted this experiment was only interested in the average GY for a typical environment, that is, the expected GY, the variation in average GY among environments (between-environment variability), and the variation in the observed GY for a single environment (within-environment variability). Figure 1.2 shows that there is considerable variability in the mean GY for different

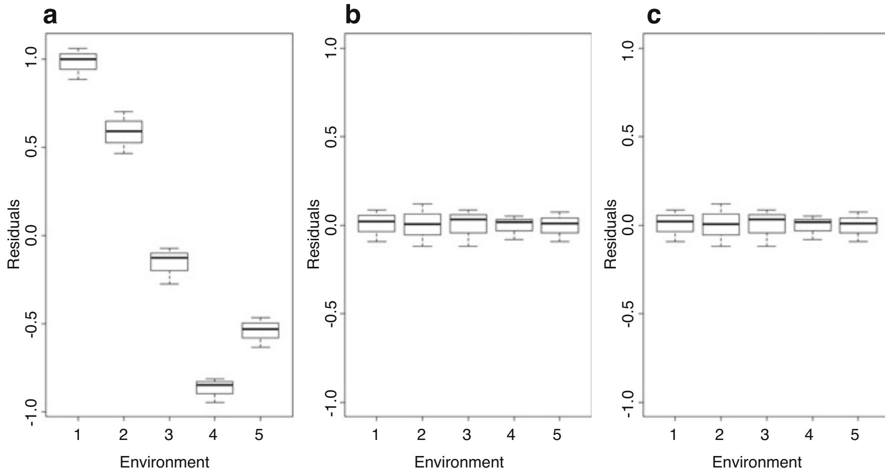



**Fig. 1.3** Box plot of residuals by environments of the fit (**a**) of a single-mean model, (**b**) of a fixed-effects model with environment effect, and (**c**) of a mixed-effects model with environment effect

environments and that the within-environment variability is smaller than the between-environment variability.

Data from a one-way classification like the one given in Table 1.1 can be analyzed under both approaches with fixed effects or random effects. The decision on which approach to use depends basically on the goal of the study, since if the goal is to make inferences about the population for which these environments (levels) were drawn, then the random effect approach is the best option, but if the goal is to make inferences about the particular environments (levels) selected in this experiment, then the fixed-effects approach should be preferred.

Assuming a simple model that ignores the environment

$$GY_{ij} = \beta + e_{ij}, \qquad i = 1, .., 5, j = 1, 2, 3, \tag{1.2}$$

where $GY_{ij}$ denotes the GY of environment $i$ in replication $j$, and $\beta$ is the mean GY across the population of environments sampled with $e_{ij} \sim N(0, \sigma^2)$. Using this model we estimate $\widehat{\beta} = 6.4127$ and the residual standard error is equal to $\widehat{\sigma} = 0.7197$. By fitting this model and observing the boxplot of the residuals (Fig. 1.3a) versus the environments, we can see that these residuals are very large and different for each environment, which can be attributed to the fact that this model ignores the environmental effect and was only fitted as a single-mean model, which implies that the environmental effects are included in the residuals. For this reason, we then incorporated the environmental effect in the model as a separate effect. This fixed-effects model is equal to a one-way classification model as

$$GY_{ij} = \beta_i + e_{ij}, \quad i = 1, .., 5, j = 1, 2, 3, \tag{1.3}$$

where $\beta_i$ represents the fixed effect of environment $i$, while the other terms in the model are the same as those in Eq. (1.2). By fitting this model using least squares, we now have a beta coefficient for each environment equal to: $\widehat{\beta}_1 = 7.396$, $\widehat{\beta}_2 = 6.999$, $\widehat{\beta}_3 = 6.255$, $\widehat{\beta}_4 = 5.543$, $\widehat{\beta}_5 = 5.869$, while the residual standard error is equal to $\widehat{\sigma} = 0.095$. Figure 1.3b shows that the residuals are considerably lower and more centered around zero than in the first fitted model (single-mean model), which can be observed in the residual standard error that is 7.47 times smaller. Therefore, we have evidence that the model given in Eq. (1.3) successfully accounted for the environmental effects. Two drawbacks of the model with fixed effects given in Eq. (1.3) are that it is unable to provide an estimate of the between-environments variability and that the number of parameters in the model increases linearly with the number of environments. Fortunately, the random effects model circumvents these problems by treating the environmental effects as random variations around a population mean. Next we reparameterize model (1.3) as a random effects model. We write

$$GY_{ij} = \overline{\beta} + \left(\beta_i - \overline{\beta}\right) + e_{ij}, \tag{1.4}$$

where $\overline{\beta} = \sum_{i=1}^{5} \beta_i / 5$ represents the average grain yield for the environments in the experiment. The random effects model replaces $\overline{\beta}$ with the mean grain yield across the population of environments and replaces the deviations $\beta_i - \overline{\beta}$ with the random variables whose distribution is to be estimated. If $\beta_i - \overline{\beta}$ is not assumed random the model belongs to fixed effects. Therefore, the random effects version of the model given in Eq. (1.4) is equal to

$$GY_{ij} = \beta + b_i + e_{ij}, \tag{1.5}$$

where $\beta$ is the mean grain yield across the population of environments, $b_i$ is a random variable representing the deviation from the population mean of the grain yield for the $i$th environment, and $e_{ij}$ is defined as before. To complete this statistical machine learning model, we must specify the distribution of the random variables $b_i$, with $i = 1, \ldots, 5$. It is common to assume that $b_i$ is normally distributed with mean zero and variance between environments $\sigma_b^2$, that is, $b_i$ is distributed $N\left(0, \sigma_b^2\right)$. It is also common to assume independence between the two random effects $b_i$ and $e_{ij}$. Models with at least two sources of random variation are also called hierarchical models or multilevel models (Pinheiro and Bates 2000). The covariance between observations in the same environment is $\sigma_b^2$, which corresponds to a correlation of $\sigma_b^2 / \left(\sigma_b^2 + \sigma^2\right)$. The parameters of the mixed model given in Eq. (1.5) are $\beta$, $\sigma_b^2$, and $\sigma^2$, and irrespective of the number of environments in the experiment, the required number of parameters will always be three, although the random effects, $b_i$, behave like parameters. We will, however, require $\widehat{b}_i$ predictions of these random effects, given the observed data at hand. Note that when fitting this model (Eq. 1.5) for the environmental data, the parameter estimates were $\beta = 6.413$, $\sigma_b^2 = 0.594$, and $\widehat{\sigma} = 0.095$. It is evident that there was no improvement in terms of fitting since the plot of

residuals versus environment looks the same as the last fitted model (Fig. 1.3c) and the estimated residual standard error was the same as that under the fixed-effects model, which includes the effects of environment in the predictor, but as mentioned above, many times requires considerably fewer parameter estimates than a fixed-effects model.

## 1.4 Matrix Algebra Review

In this section, we provide the basic elements of linear algebra that are key to understanding the machinery behind the process of building statistical machine learning algorithms.

A ***matrix*** is a rectangular arrangement of numbers whose elements can be identified by the row and column in which they are located. For example, matrix $\boldsymbol{E}$, consisting of three rows and five columns, can be represented as follows:

$$\boldsymbol{E}=\begin{bmatrix} E_{11} & E_{12} & E_{13} & E_{14} & E_{15} \\ E_{21} & E_{22} & E_{23} & E_{24} & E_{25} \\ E_{31} & E_{32} & E_{33} & E_{34} & E_{35} \end{bmatrix}$$

For example, by replacing the matrix with numbers, we have

$$\boldsymbol{E}=\begin{bmatrix} 7 & 9 & 4 & 3 & 6 \\ 9 & 5 & 9 & 8 & 11 \\ 3 & 2 & 11 & 9 & 6 \end{bmatrix}$$

where the element $E_{ij}$ is called the $ij$th element of the matrix; the first subscript refers to the row where the element is located and the second subscript refers to the column, for example, $E_{32}=2$. The order of an array is the number of rows and columns. Therefore, a matrix with $r$ rows and $c$ columns has an order of $r \times c$. Matrix $\boldsymbol{E}$ has an order of $3 \times 5$ and is denoted as $\boldsymbol{E}_{3 \times 5}$.

In R, the way to establish an array is through the command *matrix(...)* with parameters of this function given by *matrix* (*data* = *NA, nrow* = *3, ncol* = *5, byrow* = *FALSE*) where *data* is the data for the matrix, *nrow* the number of rows, *ncol* the number of columns, and *byrow* is the way in which you will accommodate the data in the matrix by row or column. The data entered by default are FALSE, so they will fill the matrix by columns, while if you specified TRUE, they will fill the matrix by rows.

For example, to build matrix $\boldsymbol{E}$ in R, use the following R script:

$$\boldsymbol{E}=\begin{bmatrix} 7 & 9 & 4 & 3 & 6 \\ 9 & 5 & 9 & 8 & 11 \\ 3 & 2 & 11 & 9 & 6 \end{bmatrix}$$

```
E <- matrix(data= c(7,9,4,3,6,9,5,9,8,11,3,2,11,9,6), nrow = 3, ncol = 5, byrow = TRUE)
E
     [,1] [,2] [,3] [,4] [,5]
[1,]   7    9    4    3    6
[2,]   9    5    9    8   11
[3,]   3    2   11    9    6
```

To access all the values of a row, for example, the first row of matrix $\boldsymbol{E}$, you can use:

```
E[1,]
[1] 7 9 4 3 6
```

While, to access all the values of a column, for example, the first column of matrix $\boldsymbol{E}$, you can use:

```
E[,1]
 [1] 7 9 3
```

To access a specific element, for example, row 3, column 2 of matrix $\boldsymbol{E}$, you can specify:

```
E[3,2]
[1] 2
```

A matrix consisting of a single row is called a vector. For example, a vector that has five elements can be represented as

$$\boldsymbol{e} = [8 \;\; 10 \;\; 5 \;\; 4 \;\; 7]$$

Here only a subscript is needed to specify the position of an element within the vector. Therefore, the i*th* element in vector $\boldsymbol{e}$ refers to the element in the ith column. For example, $e_3 = 5$.

To create a vector in R, we use the *c* (...) command that receives the data, separated by a comma. For example, the vector named $\boldsymbol{e}$ can be created using the following command:

```
e <- c(8,10,5,4,7)
e
 [1] 8 10 5 4 7
```

To access a specific index, you specify the value between brackets, for example, index 3 of vector $\boldsymbol{e}$.

```
e[3]
[1] 5
```

Next we provide definitions and examples of some common types of matrices. We start with a ***square matrix***, which is a matrix with the same number of rows and columns. A matrix $\boldsymbol{B}$ of order $3 \times 3$ is shown below.

$$\boldsymbol{B} = \begin{bmatrix} 3 & 5 & 0 \\ 5 & 1 & 5 \\ -2 & -3 & 7 \end{bmatrix}$$

The *ij* elements in the square matrix where *i* equals *j* are called diagonal elements. The rest of the elements are known as elements that are outside the diagonal, so in this example, the elements of the diagonal of matrix $\boldsymbol{B}$ are 3, 1, and 7.

To create this type of matrix in R, simply use the *matrix* (...) command and specify the dimensions in the *ncol* and *nrow* parameters, as in the following command:

```
B <- matrix(data = c(3,5,-2,5,1,-3,0,5,7), nrow = 3, ncol = 3)
B
     [,1] [,2] [,3]
[1,]    3    5    0
[2,]    5    1    5
[3,]   -2   -3    7
```

Next we define a ***diagonal matrix*** as a square matrix that has zeros in all the elements that are outside the diagonal. For example, by extracting the diagonal elements of the above matrix ($\boldsymbol{B}$), we can form the following diagonal matrix:

$$\boldsymbol{D} = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 7 \end{bmatrix}$$

Another special matrix is when the elements of the diagonal are 1; it is called an ***identity matrix***, and is usually denoted as $\boldsymbol{I}_r$, and *r* denotes the order of the matrix.

In R there is the command *diag (x* = *1, ncol, nrow)* that works to create a diagonal matrix, but if you want to extract the diagonal of the $\boldsymbol{B}$ matrix, it can be extracted in the following way:

```
diag(B)
[1] 3 1 7
```

If we want to create an identity matrix, it is enough to specify the size of the matrix. For example, to create an identity matrix of order $5 \times 5$, the command to use would be the following:

```
I <- diag(5)
I
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

A square matrix with all the elements above the diagonal equal to 0 is known as a *lower triangular* matrix; when all the lower elements of the diagonal are 0, it is known as an *upper triangular* matrix. In the example given below, matrix $F$ illustrates a lower triangular matrix and matrix $G$ illustrates an upper triangular matrix.

$$F = \begin{bmatrix} 8 & 0 & 0 \\ 10 & 7 & 0 \\ 4 & 3 & 12 \end{bmatrix}; \qquad G = \begin{bmatrix} 8 & 10 & 5 \\ 0 & 6 & 10 \\ 0 & 0 & 11 \end{bmatrix}$$

The lower triangular matrix can be extracted using the following commands:

```
F <- matrix(data = c(8,10,4,0,7,3,0,0,12), nrow = 3, ncol = 3)
F=F[upper.tri(F)] <- 0
F
     [,1] [,2] [,3]
[1,]    8    0    0
[2,]   10    7    0
[3,]    4    3   12
```

The upper triangular matrix can be extracted using the following commands:

```
G <- matrix(data = c(8,10,5,0,6,10,0,0,11), nrow = 3, ncol = 3, ,byrow=T)
G[lower.tri(G)] <- 0
G
     [,1] [,2] [,3]
[1,]    8   10    5
[2,]    0    6   10
[3,]    0    0   11
```

Next we will illustrate some basic matrix operations. We start by illustrating the ***transpose*** of matrix $\boldsymbol{E}$, commonly indicated as $\boldsymbol{E}'$ or $\boldsymbol{E}^T$; in this type of matrix, the elements *ji* are the *ij* elements of the original matrix. That is, $E'_{ji} = E_{ij}$, where the columns of $\boldsymbol{E}'$ are the rows of $\boldsymbol{E}$, and the rows of $\boldsymbol{E}'$ are the columns of $\boldsymbol{E}$. Below we provide matrix $\boldsymbol{H}$ and its transpose $\boldsymbol{H}'$.

$$\boldsymbol{H} = \begin{bmatrix} 4 & 6 \\ 6 & 2 \\ 0 & -1 \end{bmatrix}; \qquad \boldsymbol{H}' = \begin{bmatrix} 4 & 6 & 0 \\ 6 & 2 & -1 \end{bmatrix}$$

To obtain the transpose of a matrix in $\boldsymbol{H}$, the command *t (...)* is used:

```
H <- matrix(data = c(4,6,0,6,2,-1), nrow = 3,  2)
H
    [,1] [,2]
[1,]    4    6
[2,]    6    2
[3,]    0   -1
```

```
t(H)
 [1,]    4    6    0
[2,]    6    2   -1
```

Two matrices can be added or subtracted only if they have the same number of rows and columns. To demonstrate the adding process, we use the following matrices:

$$\boldsymbol{J} = \begin{bmatrix} 15 & 15 \\ 25 & 35 \end{bmatrix}; \quad \boldsymbol{L} = \begin{bmatrix} 55 & 65 \\ 75 & 85 \end{bmatrix}$$

We form matrix $\boldsymbol{M}$ as the sum of matrices $\boldsymbol{J}$ and $\boldsymbol{L}$, so that $\boldsymbol{M}_{ij} = \boldsymbol{J}_{ij} + \boldsymbol{L}_{ij}$, and their sum is the following:

$$\boldsymbol{M} = \begin{bmatrix} 15+55 & 15+65 \\ 25+75 & 35+85 \end{bmatrix} = \begin{bmatrix} 70 & 80 \\ 100 & 120 \end{bmatrix}$$

Matrix $\boldsymbol{N}$ is reached by subtracting matrices $\boldsymbol{J}$ and $\boldsymbol{L}$, so $\boldsymbol{N}_{ij} = \boldsymbol{J}_{ij} - \boldsymbol{L}_{ij}$, and the subtraction is the following:

$$\boldsymbol{N} = \begin{bmatrix} 15-55 & 15-65 \\ 25-75 & 35-85 \end{bmatrix} = \begin{bmatrix} -40 & -50 \\ -50 & -50 \end{bmatrix}$$

To do the addition and subtraction of matrices in R, it is enough to use the addition or subtraction operator and fulfill the requirement that both matrices have the same dimensions. Next we reproduce the two previous addition and subtraction examples using the commands in R:

```
J <- matrix(data= c(15,15,25,35), ncol = 2, byrow = TRUE)
L <- matrix(data= c(55,65,75,85), ncol = 2, byrow = TRUE)

M<- J+L
M
    [,1] [,2]
[1,]  70   80
[2,]  100  120
```

While the subtraction of matrices is:

```
N<- J-L
N
      [,1]  [,2]
[1,]  -40  -50
[2,]  -50  -50
```

Two matrices can be multiplied only if the number of columns in the first matrix equals the number of rows in the second. The resulting matrix will be equal to the number of rows in the first matrix and the number of columns in the second. Since $\boldsymbol{O} = \boldsymbol{PQ}$, then

$$O = \sum_{j=1}^{m} \sum_{i=1}^{n} \sum_{k=1}^{z} P_{ik} Q_{kj},$$

where $m$ is the number of columns in matrix $\boldsymbol{Q}$, $n$ the number of rows in matrix $\boldsymbol{P}$, and $z$ the number of rows in $\boldsymbol{Q}$ and number of columns in $\boldsymbol{P}$. To demonstrate the above, we have

$$\boldsymbol{P} = \begin{bmatrix} 6 & 8 & 10 \\ 5 & 6 & 8 \\ 4 & 6 & 9 \end{bmatrix}; \boldsymbol{Q} = \begin{bmatrix} 3 & 5 \\ 2 & 2 \\ 9 & 8 \end{bmatrix}$$

Then $\boldsymbol{S}$ is obtained as

$$\begin{aligned} S_{11} &= 6 \times 3 + 8 \times 2 + 10 \times 9 = 124 \\ S_{21} &= 5 \times 3 + 6 \times 2 + 8 \times 9 = 99 \\ S_{31} &= 4 \times 3 + 6 \times 2 + 9 \times 9 = 105 \\ S_{12} &= 6 \times 5 + 8 \times 2 + 10 \times 8 = 126 \\ S_{22} &= 5 \times 5 + 6 \times 2 + 8 \times 8 = 101 \\ S_{32} &= 4 \times 5 + 6 \times 2 + 9 \times 8 = 104 \end{aligned}$$

Therefore,

$$\boldsymbol{S} = \begin{bmatrix} 124 & 126 \\ 99 & 101 \\ 105 & 104 \end{bmatrix}$$

Note that $\boldsymbol{S}$ is of order $3 \times 2$, where 3 represents the number of rows in $\boldsymbol{P}$ and 2 equals the number of columns in $\boldsymbol{Q}$.

In order to multiply matrices in R, it is necessary to use the operator %*% between the two matrices, in addition to meeting the requirements mentioned above.

```
P <- matrix(data=c(6,5,4,8,6,6,10,8,9), ncol=3)
Q <- matrix(data=c(3,2,9,5,2,8), ncol=2)

S <- P%*%Q
S
     [,1]  [,2]
[1,]  124  126
[2,]   99  101
[3,]  105  104
```

The inverse of a matrix usually is denoted as $\boldsymbol{R}^{-1}$, and when it is multiplied by the original matrix, it results in an identity matrix, that is, $\boldsymbol{R}^{-1}\boldsymbol{R} = \boldsymbol{I}$, where $\boldsymbol{I}$ is the identity matrix. Only square matrices are invertible.

If it is a diagonal matrix, its inverse can be calculated simply in the following way:

$$\boldsymbol{R} = \begin{bmatrix} 7 & 0 & 0 \\ 0 & 8 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

Then

$$\boldsymbol{R}^{-1} = \begin{bmatrix} \frac{1}{7} & 0 & 0 \\ 0 & \frac{1}{8} & 0 \\ 0 & 0 & \frac{1}{6} \end{bmatrix}$$

For a square matrix of 2×2, its inverse can be calculated by finding the determinant, which is the difference between the product of the two elements of the diagonals and the product of the two elements outside the diagonal ($R_{11}R_{22} - R_{12}R_{21}$). Then the position of the elements of the diagonals is reversed by multiplying the elements outside the diagonal by $-1$ and dividing all the elements by the determinant.

$$\boldsymbol{R} = \begin{bmatrix} 4 & 2 \\ 1 & 6 \end{bmatrix}$$

Then

$$\boldsymbol{R}^{-1} = \frac{1}{(4 \times 6) - (1 \times 2)} \begin{bmatrix} 6 & -2 \\ -1 & 4 \end{bmatrix} = \begin{bmatrix} 0.2727 & -0.0909 \\ -0.0455 & 0.1818 \end{bmatrix}$$

To obtain the inverse in ***R***, the solve(...) command is the function used to perform this process, as shown in the following example, where the inverse of the matrix ***R*** is obtained.

```
R <- matrix(data = c(4, 1, 2, 6), ncol = 2)
R
     [,1] [,2]
[1,]    4    2
[2,]    1    6
```

```
solve(R)
        [,1]    [,2]
[1,]  0.2727 -0.0909
[2,] -0.0455  0.1818
```

## 1.5 Statistical Data Types

### *1.5.1 Data Types*

To use statistical learning methods correctly, it is very important to understand the classification of the types of data that exist. This is of paramount importance because data are the input to all statistical machine learning methods and because the data type *determines* the appropriate and valid analysis to be implemented; in addition, each statistical machine learning method is specific to a certain type of data. In general, data are most commonly classified as quantitative (numerical) or qualitative (categorical) (Fig. 1.4).

By quantitative (numerical) data, we understand that the result of the observation or the result of a measurement is a number. They are classified as

(a) Discrete. The variable can only have point values and no values in between, that is, the variable can only have a certain set of possible values and represent items that can be counted because they only have isolated numerical values. Examples: number of household members, number of surgical interventions, number of reported cases of a certain pathology, number of accidents per month, etc. Examples in the context of plant breeding are panicle number per plant, seed number per panicle, weed count per plot, number of infected spikelets per spike, etc. Also, discrete values are called as count responses and those models based on Poisson and negative binomial distribution are appropriate for this type of responses.

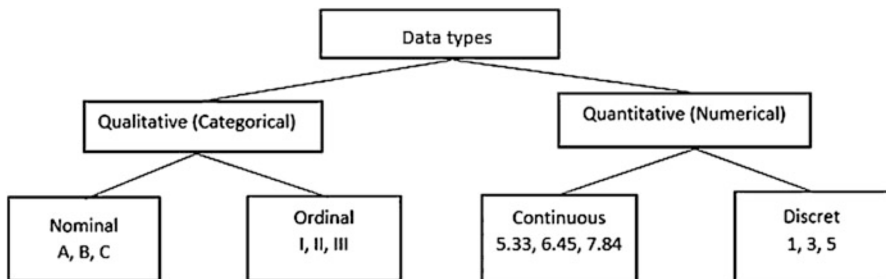



**Fig. 1.4** Types of data

(b) Continuous. They are usually the result of a measurement that is expressed in particular units, and values are measured based on a zero point and are treated as real numbers. There are many types of mathematical operations that can be performed on this type of data. The measurements can theoretically have an infinite set of possible values within a range and they do not need transformation. In practice, the possible values of the variable are limited by the accuracy of the measurement method or by the recording mode. Examples: plant height, age, weight, grain yield, pH, blood cholesterol level, etc. The distinction between discrete and continuous data is important for deciding which statistical learning method to use for the analysis, since there are methods that assume that the data are continuous. Consider, for example, the age variable. Age is continuous, but if it is recorded in years, it turns out to be discrete. In studies with adults, in which the age ranges from 20 to 70 years, for example, there are no problems in treating age as continuous, since the number of possible values is large. But in the case of preschool children, if the age is recorded in years, it should be treated as discrete, while if it is recorded in months, it can be treated as continuous.

Similarly, the variable number of beats per minute is a discrete variable, but it is treated as continuous due to the large number of possible values. Numerical data (discrete or continuous) can be transformed into categorical and be treated as such. Although this is correct, it is not necessarily efficient because information is lost during the categorization process. It is always preferable to record the numerical value of the measurement, since this makes it possible to (a) analyze the variable as numerical because statistical analysis is simpler and more powerful and (b) form new categories using different criteria. Only in special cases it is preferable to record numerical data as categorical, for example, when the measurement is known to be imprecise (number of cigarettes per day, number of cups of coffee per week).

Categorical variables result from registering the presence of an attribute. The categories of a qualitative variable must be clearly defined during the design stage of the research and must be mutually exclusive and exhaustive. This means that each observation unit must be classified unambiguously in one, and only one, of the possible categories and that there is a category to classify each individual. In this sense, it is important to take into account all possibilities when constructing categorical variables, including a category such as "Do not know/No answer," or "Not Registered," or "Other," which ensures that all the observed individuals will be classified based on the criteria that define the variable. Categorical data are also classified as (a) dichotomous, (b) nominal, and (c) ordinal.

(a) Two categories (dichotomous). The individual or observation unit can be assigned to only one of two categories. In general, it is about the presence or absence of the attribute and it is advantageous to assign code 0 to the absence and 1 to the presence.

(b) Examples: (1) resistance—no resistance, (2) disease—no disease, (3) tall—not tall, and (4) red color—no red color. It should be noted that examples 1 and 2 definitely cover all categories, while 3 and 4 are simplifications of more complex categories. In 3 and 4 it was necessary to establish a cutoff criterion to assemble a categorical variable from a numerical variable.

(c) More than two categories. When there are more than two categories, data can be nominal or ordinal. In nominal categories, there is no obvious order between the categories. These types of data values are distinct symbols, and these values serve as labels. The term "nominal" comes from the latin word for "name." Nominal attributes or labels have no relation to one another, nor is any order implied (Patterson and Gibson 2017). Some examples are religion: Catholicism, Islam, Judaism, etc.; race type: African, American, European, Asian, other; type of species; location; plant color; etc. In ordinal data, there is an obvious order between categories. Ordinal values have rank, giving us a notion of order but no concept of distance between the values. We can compare ordinal values with one another, but mathematical operations don't make sense in the context of these values (Patterson and Gibson 2017). Some examples are

    1. Drought resistance: no resistance/low resistance/medium resistance/high resistance/total resistance
    2. Disease severity: absent/mild/moderate/severe
    3. Temperature of a process: hot/mild/cool
    4. Social class: lower/middle/upper

Even when ordinal data can be coded as numbers as in the case of stages of drought resistance from 1 to 5 (1 = no resistance, 2 = low resistance, 3 = medium resistance, 4 = high resistance, 5 = total resistance), we cannot say that a plant in stage 4 has a drought resistance twice as strong as the resistance of a plant in stage 2, nor that the difference between stages 1 and 2 is the same as between stages 3 and 4. In contrast, when considering the age of a person, 40 years is twice 20 and a difference of 1 year is the same across the entire range of values. Therefore, we need to be aware that in ordinal data the difference between categories does not make sense.

Ordinal traits are very common in plant breeding programs for measuring disease incidence and severity and for sensory evaluation, such as the perceived quality of a product (e.g., taste, smell, color, decay) and plant development (e.g., developmental stages, maturity). These types of data are often partially subjective since the scale indicates only relative order and no absolute amounts; therefore, the intervals between successive categories may not be the same (Simko and Piepho 2011).

For this reason, we must be careful when dealing with qualitative variables, especially when they have been coded numerically, since they cannot be analyzed as numbers but must be analyzed as categories. It is incorrect to present, for example, the average stage of drought resistance in a group of plants.

In practice, scales are used to define degrees of a symptom or a disease, such as 1, 2, 3, 4, and 5. For this reason, it is important to operationally define this type of variables and study their reliability in order to ensure that two observers placed in front of the same plant will classify it in the same category.

**Table 1.2** Examples of multivariate data

| Units | Variables | Types of data |
|---|---|---|
| Plant | Several measurements of plant height on a single plant in time | All continuous |
| Animal | Measurement of three animal traits (average daily weight gain, muscularity, and calving) | Mixture of continuous and ordinal |
| Students | Grades in mathematics, physics, chemistry, biology | All continuous |
| People | Income, type of residence, gender, educational level, occupation | Mixture of nominal, ordinal, and continuous |
| Wheat plant | Measurement of four traits: grain yield, panicle number per plant, drought resistance, and type of wheat (common or durum) | Mixture of continuous, discrete, ordinal, and nominal |
| Country | Several measurements of school performance using the programme for international student assessment (PISA) test | Exam scores continuous in mathematics, reading and science |

### *1.5.2 Multivariate Data Types*

Practitioners and researchers in all applied disciplines often measure several variables in each observation, subject, unit, or experimental unit. That is, all variables are simultaneously measured in the same observation. Multivariate data are very common in all disciplines due to the need and facility for data collection in most fields. These variables can consist of only one type of data (for example, plant height measured using a continuous scale on each plant 12 times every 15 days) or a mixture of data types, for example, measuring, on each plant, four different traits: grain yield (on a continuous scale), disease resistance (on an ordinal scale), flower color (nominal scale), and days to flowering (discrete or count). Table 1.2 provides other examples of multivariate data measured using only one scale or a mixture of scales.

It is important to point out that here all measurements are done simultaneously in each observation. For this reason, they are classified as multivariate type of data and include data that will be used as dependent variables or independent variables in the process of training the statistical machine learning algorithms that will be studied here.

## 1.6 Types of Learning

The three most common ways of learning in statistical machine learning are (a) supervised learning, (b) unsupervised learning, and (c) semi-supervised learning. The three methods are explained below.

### 1.6.1 Definition and Examples of Supervised Learning

*Supervised learning* can be defined as the process of learning a function that maps an input to an output based on teaching the statistical machine learning method with input–output pairs. The training data consist of pairs of objects (usually vectors): one component of the pair is the input data (predictors = explanatory variable = input) and the other, the desired results (response variable = dependent variable = output). The output of the function can be a numerical value (as in regression problems) or a class label (as in multinomial regression). The goal of supervised learning is to learn a function that, given a sample of data and desired outputs, best approximates the relationship between input and output observable in the data. This function should be capable of predicting the value corresponding to any valid input object after having seen a series of examples of training data. Under optimal conditions, the algorithm correctly determines the class labels for unseen instances. This implies a learning algorithm that is able to generalize from the training data to unseen situations in a "reasonable" way.

Suppose you're teaching your child to distinguish between corn and tomato (Fig. 1.5). First you show him (her) a picture of an ear of corn and a picture of a tomato. In the learning process, your child must keep in mind that if the color is yellow and the shape is not round, then it is probably an ear of corn, but if the color is red and the shape is round, then it is probably a tomato. This is how your child learns. Then you can show a third picture and ask your child to classify the vegetable as either ear of corn or tomato. When you show the third picture, he (she) will very likely identify if the vegetable is ear of corn or tomato, due to the fact that we have already labeled the two pictures into categories, so your child knows what is an ear of

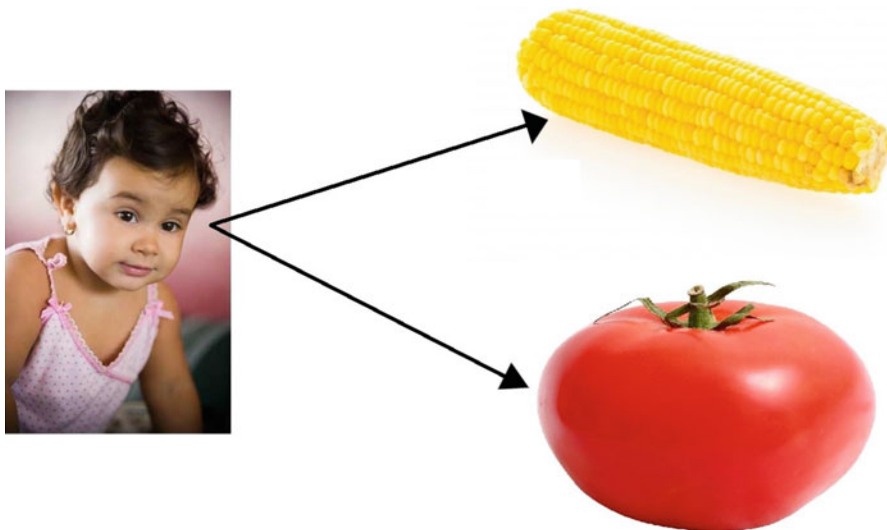


**Fig. 1.5** Supervised learning process for teaching a child to distinguish tomato from corn

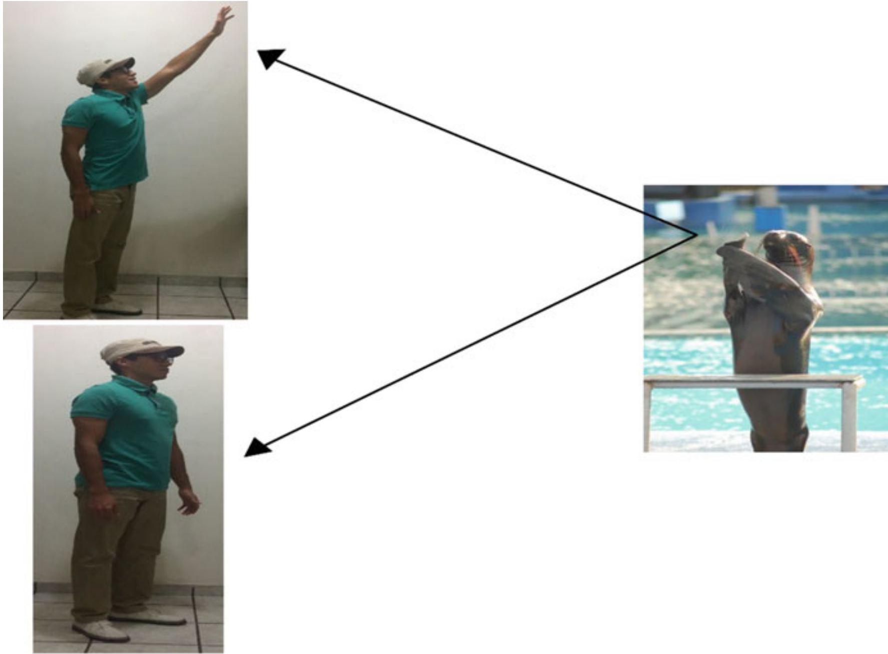


**Fig. 1.6** Supervised learning process for teaching a seal to applaud

corn and what is a tomato. This example illustrates how supervised learning works using ground truth data that consist of having prior knowledge of what the output values of our samples should be.

To give another example, imagine that you're training a seal to applaud (Fig. 1.6). The goal is to make the seal applaud when you raise your right hand. The training process consists of presenting the seal with enough examples by raising your right hand and rewarding it with some great candy whenever it applauds when it sees your right hand is raised. In the same way, the seal may be "punished" if it applauds whenever your right hand is not raised, by doing something unpleasant for the seal but not harmful. Supervision involves stimulating the seal to respond to positive samples by rewarding it, and not to respond to negative samples by "punishing" it. Hopefully, the seal then obtains a built-in feeling (hypothesis) for applauding whenever you raise your hand right. The process is evaluated by presenting the seal with another person raising his/her right hand, someone who did not take part in the training process and who is unknown to the seal. However, based on its built-in feeling for what a person with his/her raised right hand looks like, the seal should be able to transfer this knowledge to the present person. It must then consider and decide whether or not it wants to signal the presence of a right hand raised by applauding.

Next we provide some real examples. In the first example, a scientist has thousands of molecules and information about which ones are drugs and he trains

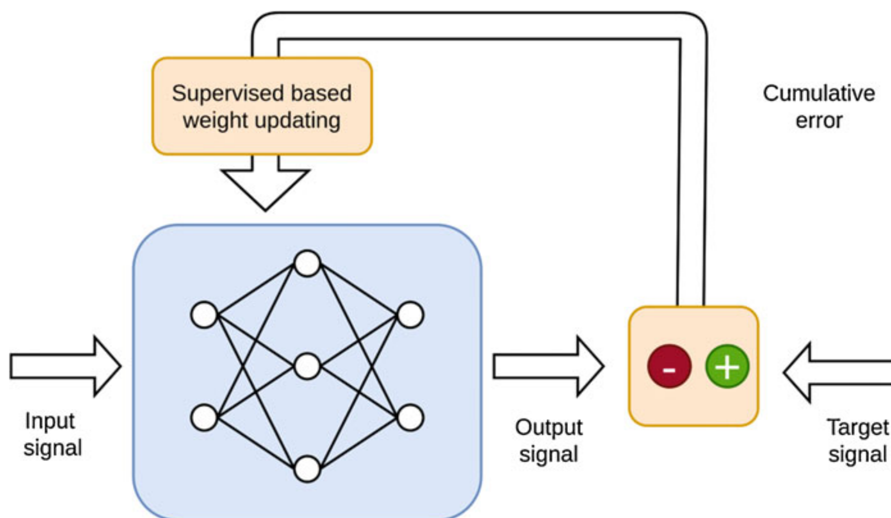



**Fig. 1.7** Diagram illustrating a supervised learning process where there are inputs (X) and response variables (y) for each observation

a statistical machine learning model to determine whether a new molecule is also a drug. In the context of plant breeding, a scientist collects hundreds of markers (genetic information) and thousands of images of hundreds of plants. For this sample of plants, he also measures their phenotype (grain yield) because he wants to implement a statistical machine learning algorithm to estimate (predict) the grain yield of new plants not used in the training process. Another example is in environmental science, where scientists use historical data (as when it's sunny and the temperature is higher, or when it's cloudy and the humidity is higher, etc.) to train a statistical machine learning model to predict the weather for a given future time.

In a more mathematical way, under supervised learning, we usually have access to a set of $p$ predictor (input) variables $X_1, X_2, \ldots, X_p$ measured in $n$ observations, and a response variable (output) $Y$ also measured in those same $n$ observations (Fig. 1.7). The goal is to predict $Y$ using a function of $X_1, X_2, \ldots, X_p$, that is, we use an algorithm to learn the mapping function from the input to the output $Y = f(X_1, X_2, \ldots, X_p)$, and we expect to estimate the mapping function so well that when we have new input data, we can predict the output variables for those data. The term supervised learning was coined because the learning process of any statistical machine learning method from the training dataset can be thought of as a teacher supervising the learning process. We know the correct outputs (response variables), the algorithm iteratively makes predictions on the training data and is corrected by the teacher. Learning stops when the statistical machine learning algorithm achieves an acceptable level of performance.

### 1.6.2 Definitions and Examples of Unsupervised Learning

*Unsupervised learning* is when you only have input (predictors = independent variables) data ($X$) and no previous knowledge of corresponding labeled outputs or response variables (Fig. 1.8). So its goal is to deduce the natural structure present within a set of data points. In other words, to extract the underlying structure or distribution in the data in order to learn more about the data, that is, the network uses training patterns to discover emerging collective properties and organizes the data into clusters. In unsupervised learning (unlike supervised learning), there is no correct answer (output = response variable = dependent variable) and there is no teacher. For this reason, we are not interested in prediction since we do not have an associated response variable $Y$. Statistical machine learning algorithms under unsupervised learning are left to their own devices to discover and present the interesting structure in the data. However, there is no way to determine if our work is correct since we don't know the right answer because the job was done without supervision. Unsupervised learning problems can be divided into clustering and association problems.

Clustering: A clustering problem is when you want to discover the inherent groupings in the data, such as grouping maize hybrids by their genetic architecture. Another example is grouping people according to their consumption behaviors. But in both cases we cannot check if the classifications are correct since we don't know the true grouping of each individual.

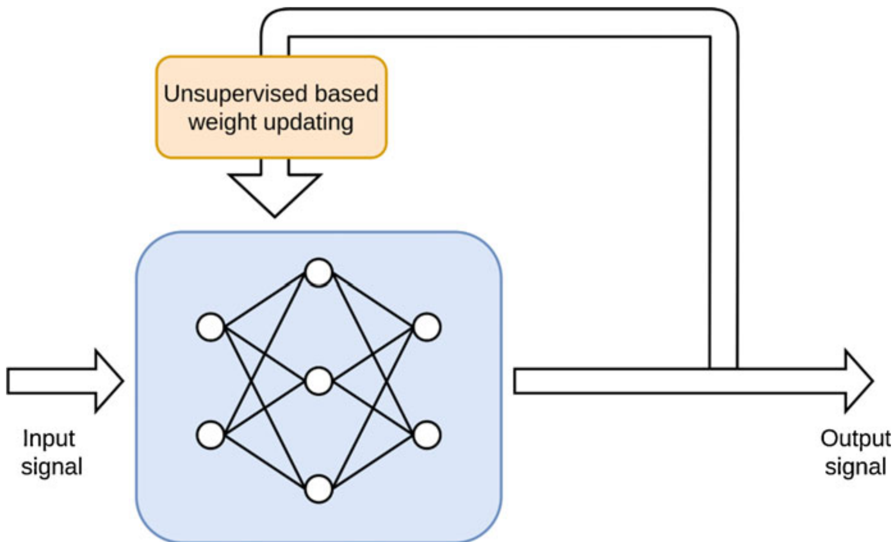



**Fig. 1.8** Diagram illustrating an unsupervised learning process where there are only inputs (X), but no response variables (y) for each observation

Association: An association rule learning problem is when you want to discover rules that describe large portions of your data, such as people who buy X also tend to buy Y.

Some popular examples of unsupervised learning algorithms are

(a) Principal component analysis
(b) Multidimensional scaling for grouping
(c) A k-means algorithm for clustering problems
(d) An a priori algorithm for association rule learning problems

### *1.6.3 Definition and Examples of Semi-Supervised Learning*

*Semi-supervised learning* problems are those that have a large amount of input data (X) available but only some of the data are labeled (Y). For this reason, these problems are positioned between supervised and unsupervised learning. A good example is plant species classification using thousands of images where only some of the images are labeled (e.g., species 1, species 2, species 3, etc.) and the majority are unlabeled. Another example is the classification of exoplanets (exoplanets are planets that are outside our solar system) also using thousands of photos where only a small fraction of the photos is labeled (four types of exoplanets). Many real-world problems in the context of statistical machine learning belong to this type of learning process. This is because it is more expensive and time-consuming to use labeled data than unlabeled data since many times this requires having access to domain experts, whereas it is cheap and easy to collect and store unlabeled data.

## References

Adler I (1912) Primary malignant growths of the lungs and bronchi: a pathological and clinical study. Longmans, Green and Co, New York and London, p 325

Bernardo R (2016) Bandwagons I, too, have known. Theor Appl Genet. https://doi.org/10.1007/s00122-016-2772-5

Box GEP (1976) Science and statistics (PDF). J Am Stat Assoc 71:791–799. https://doi.org/10.1080/01621459.1976.10480949

Breiman L (2001) Statistical modeling: the two cultures. Stat Sci 16:199–215

Crossa J, Pérez-Rodríguez P, Cuevas J, Montesinos-López OA, Jarquín D, de Los Campos G, Burgueño J, González-Camacho JM, Pérez-Elizalde S, Beyene Y, Dreisigacker S, Singh R, Zhang X, Gowda M, Roorkiwal M, Rutkoski J, Varshney RK (2017) Genomic selection in plant breeding: methods, models, and perspectives. Trends Plant Sci 22(11):961–975

Dean J (2018) Big data, data mining, and machine learning. Value creation for business leaders and practitioners. John Wiley & Sons, Inc., Hoboken

FAO (2011) The state of the World's land and water resources for food and agriculture: managing Systems at Risk. Food and agriculture Organization of the United Nations. FAO, Rome

Fischer T, Byerlee D, Edmeades G (2014) Crop yields and global food security. ACIAR, Canberra

James G, Witten D, Hastie T, Tibshirani R (2013) An introduction to statistical learning: with applications in R. Springer, New York

McKinsey Global Institute (2016) The age of analytics: competing in a data-driven world. https://www.mckinsey.com/~/media/mckinsey/business%20functions/mckinsey%20analytics/our%20insights/the%20age%20of%20analytics%20competing%20in%20a%20data%20driven%20world/mgi-the-age-of-analytics-executive-summary.ashx
Meuwissen THE, Hayes BJ, Goddard ME (2001) Prediction of total genetic value using genome-wide dense marker maps. Genetics 157:1819–1829
Milliken GA, Johnson DE (2009) Analysis messy of data, volume 1 designed experiments. CRC Press Taylor & Francis Group, Boca Raton, London, New York
Montesinos-López A, Martín-Vallejo J, Crossa J, Gianola D, Hernández-Suárez CM, Montesinos-López OA, Juliana P, Singh R (2019) A benchmarking between deep learning, support vector machine and Bayesian threshold best linear unbiased prediction for predicting ordinal traits in plant breeding. G3 9(2):601–618
Oury F-X, Godin C, Mailliard A, Chassin A, Gardet O, Giraud A, Heumez E, Morlais J-Y, Rolland B, Rousset M, Trottet M, Charmet G (2012) A study of genetic progress due to selection reveals a negative effect of climate change on bread wheat yield in France. Eur J Agron 40:28–38
Patterson J, Gibson A (2017) Deep learning: a Practitioner's approach. O'Reilly Media, Beijing
Pinheiro JC, Bates DM (2000) Mixed-effects models in S and S-PLUS. Springer Verlag, New York
Proctor RN (2012) The history of the discovery of the cigarette-lung cancer link: evidentiary traditions, corporate denial, global toll. Tob Control 21(2):87–91
Samuel AL (1959) Some studies in machine learning using the game of checkers. IBM J Res Dev 3 (3):210–229
Schuster PM (2014) The scientific life of Victor Franz (Francis) Hess (June 24, 1883–December 17, 1964). Astropart Phys 53:33–49
Sejnowski TJ (2018) The deep learning revolution. The MIT Press, Cambridge, MA, London
Shmueli G (2012) To explain or to predict? Stat Sci 25(3):289–310. https://doi.org/10.1214/10-STS330
Simko I, Piepho H-P (2011) Combining phenotypic data from ordinal rating scales in multiple plant experiments. Trends Plant Sci 16:235–237
Stroup W (2012) Generalized linear mixed models: modern concepts, methods and applications. CRC Press, Boca Raton
Wang X, Xua Y, Hu Z, Hu C (2018) Genomic selection methods for crop improvement: current status and prospects. Crop J 6(4):330–340
Wolpert DH (1996) The lack of a priory distinction between learning algorithms. Neural Comput 8 (7):1341–1390

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 2
# Preprocessing Tools for Data Preparation

## 2.1 Fixed or Random Effects

As mentioned in Chap. 1, fixed effect models in general (to design experiments, regression models, genomic prediction models, etc.) are recommended when the levels under study (collected by the scientist) are the unique levels of interest in the study, and the levels or quantities observed in the explanatory variables are treated as if they were nonrandom. For these reasons, a fixed factor is defined as a categorical or classification variable, chosen to represent specific conditions, for which the researcher has included all levels (or conditions) that are of interest for the study in the model. This means that fixed effects are unknown constant parameters associated with continuous covariates or levels of categorical factors in any fixed or mixed effects (fixed + random effects). The estimation of these fixed parameters in fixed effects models or mixed effects models is generally of intrinsic interest, since they indicate the relationships of the covariates with the response variable. Fixed effects can be associated with continuous covariates such as the weight of an animal in kilograms, maize yield in tons per hectare, qualification of a reference test or socioeconomic level, which will carry a continuous range of values, or they can be associated with factors such as gender, hybrid, or group treatment, which are categorical. This implies that fixed effects are the best option when performing an inference for the whole target population.

A random factor is a classification variable with levels that can be randomly sampled from a population with different levels of study, such as classrooms, regions, cattle herds, or clinics that are randomly sampled from a population. All possible levels of the random factor are not present in the data set, yet it is the intention of the researcher to make an inference about the entire population of levels from the selected sample of these factor levels. Random factors are included in an analysis in order for the modification in the dependent variable through the levels of the random factors to be evaluated and the results of the data analysis generalized to all levels of the population random factor. This means that random effects are

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_2

represented by random variables (not observed) which, we generally assume, have a particular distribution, the normal distribution being the most common. Due to the above, random effects are suggested when we want to perform an inference for all levels of the target population.

## 2.2 BLUEs and BLUPs

This section presents the concepts and terminologies of BLUE and BLUP. Since these two concepts are related to a mixed model, we present the following linear mixed model as

$$\boldsymbol{Y} = \boldsymbol{X\beta} + \boldsymbol{Zu} + \boldsymbol{\varepsilon}, \tag{2.1}$$

where $\boldsymbol{Y}$ is the vector of response variables of order $n \times 1$, $\boldsymbol{X}$ is the design matrix of fixed effects of order $n \times p$, $\boldsymbol{\beta}$ is the vector of order $p \times 1$ of beta coefficients, $\boldsymbol{Z}$ is the design matrix of random effects of order $n \times q$, $\boldsymbol{u}$ is the vector of random effects distributed as $\boldsymbol{N}(\boldsymbol{0}, \boldsymbol{\Sigma})$, where $\boldsymbol{\Sigma}$ is a variance–covariance matrix of random effects of dimension $q \times q$, and $\boldsymbol{\varepsilon}$ is a vector of residuals distributed as $\boldsymbol{N}(\boldsymbol{0}, \boldsymbol{R})$, where $\boldsymbol{R}$ is a variance–covariance matrix of residual effects of dimension $n \times n$. The unconditional mean of $\boldsymbol{Y}$ is equal to $E(\boldsymbol{Y}) = \boldsymbol{X\beta}$, while the conditional mean of $\boldsymbol{Y}$, given the random effects, is equal to $E(\boldsymbol{Y}|\boldsymbol{u}) = \boldsymbol{X\beta} + \boldsymbol{Zu}$. A solution to jointly "estimate" parameters $\boldsymbol{\beta}$ and $\boldsymbol{u}$ was proposed by Henderson (1950, 1963, 1973, 1975, 1984), which consists in solving the mixed model equation (MME)

$$\begin{pmatrix} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{X} & \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{Z} \\ \boldsymbol{Z}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{X} & \boldsymbol{Z}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{Z} + \boldsymbol{\Sigma}^{-1} \end{pmatrix} \begin{pmatrix} \widehat{\boldsymbol{\beta}} \\ \widehat{\boldsymbol{u}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{y} \\ \boldsymbol{Z}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{y} \end{pmatrix} \tag{2.2}$$

The solution obtained for $\boldsymbol{\beta}$ is the BLUE and the solution obtained for $\boldsymbol{u}$ is the BLUP.

While this expression to find the estimates of $\widehat{\boldsymbol{\beta}}$ and $\widehat{\boldsymbol{u}}$ may look quite complex, when the number of observations is larger than the sum of the number of fixed effects and the number of random effects $(p + q)$, it is quite efficient since only needs to calculate the inverse of the small matrices of $\boldsymbol{R}$ and $\boldsymbol{\Sigma}$. Also, the matrix on the left that needs to be inverted to obtain the solution for $\widehat{\boldsymbol{\beta}}$ and $\widehat{\boldsymbol{u}}$ is of order $(p + q) \times (p + q)$, which in some applications is considerably less than a matrix of dimension $n \times n$ as $\boldsymbol{V} = \boldsymbol{Z\Sigma Z}^{\mathrm{T}} + \boldsymbol{R}$, which is also useful to obtain the solution of these parameters by using $\widehat{\boldsymbol{\beta}} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{y}$ and $\widehat{\boldsymbol{u}} = \boldsymbol{\Sigma Z}^{\mathrm{T}}\left(\boldsymbol{y} - \boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)$. Under both solutions, for $\widehat{\boldsymbol{\beta}}$ and $\widehat{\boldsymbol{u}}$ it is assumed that the covariance matrices are known, but in practice these are replaced by estimations and the results are known as empirical BLUE (EBLUE) and empirical BLUP (EBLUP).

The linear combinations of the parameters are called **estimable functions** if they can be constructed from a linear combination of unconditional means (of fixed effects only) of the observations (Littell et al. 1996). Estimable functions do not depend on random effects. Below, we provide a formal definition of an estimable function.

**Definition of an estimable function** $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{\beta}$ is estimable if there is a matrix $\boldsymbol{T}$ such that

$$\boldsymbol{T}^{\mathrm{T}}\boldsymbol{E}(\boldsymbol{Y}) = \boldsymbol{T}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\beta} = \boldsymbol{K}^{\mathrm{T}}\boldsymbol{\beta}\ \forall\ \boldsymbol{\beta}.$$

One way of testing candidate matrices for estimability is to use the following result:

$$\boldsymbol{K}^{\mathrm{T}}\boldsymbol{\beta} \text{ is estimable if, and only if, } \boldsymbol{K}^{\mathrm{T}}\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-}\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right) = \boldsymbol{K}^{\mathrm{T}},$$

where $(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-}$ denotes a generalized inverse of $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$. Quantities such as regression coefficients, treatment means, treatment differences, contrasts, and simple effects in factorial experiments are all common examples of estimable functions and their resulting estimates are examples of BLUEs (Littell et al. 2006). BLUEs correspond to broad inference because they are valid for the whole population under study, and are also called population average inference using the terminology by Zeger et al. (1998). Table 2.1 presents predictors of some regression models and some common experimental designs and also provides some functions of the predictor that are and are not estimable functions.

Table 2.1 indicates that estimable functions used to obtain BLUEs are only linear combinations of fixed effects, and the inference focuses on the average performance throughout the target population. Although there are many possible linear combinations of fixed effects that can be of interest to estimate with BLUEs, the most important ones are treatment means expressed as $\eta_0 + \tau_i$, the difference between treatments $\tau_i - \tau_{i'}$ and simple effects. In general, the most common BLUEs can be obtained from any fitted generalized mixed model using the expression BLUE = $g^{-1}\left(\boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)$, where $g^{-1}(\cdot)$ is the inverse link used to fit the generalized mixed model and the inference is related to population-wide average (broad).

Estimability matters because many models are not full rank, such as analysis of variance (ANOVA) models, where estimating equation solutions for the effects themselves has no intrinsic meaning and solutions depend entirely on the generalized inverse used. Theory says that there is an infinite number of ways to construct a generalized inverse. While estimable functions are invariant of generalized inverse and therefore have an assignable meaning, that is, although the effect estimates per se do not have any legitimate interpretation, estimable functions do (Stroup 2012). Next, an example of how to obtain the BLUEs of genotypes (treatments) under a randomized complete block design is provided.

**Table 2.1** Predictors and some estimable and non-estimable functions of each predictor of some common useful linear models

| Model | Predictor | Estimable functions | Non-estimable functions |
|---|---|---|---|
| Complete randomized design (CRD) | $\eta_i = \eta_0 + \tau_i$ | $\eta_0 + \tau_i$, $\tau_i - \tau_{i'}$, $i \neq i'$ | $\tau_i$, $\eta_0$ |
| Randomized complete blocks (RCBD) | $\eta_{ij} = \eta_0 + \tau_i + b_j$, $b_j \sim N\left(0, \sigma_b^2\right)$ | $\eta_0 + \tau_i$, $\tau_i - \tau_{i'}$, $i \neq i'$ | $\tau_i$, $\eta_0$ |
| Regression | $\eta_i = \eta_0 + \sum_{j=1}^{p} x_{ij}\beta_j$ | $\eta_0 + \sum_{j=1}^{p} x_{ij}\beta_j$, $\eta_0 + x_{ij}\beta_j$ | $\eta_0$ |
| Split plot design in a CRD | $\eta_{ijk} = \eta_0 + \alpha_i + \alpha(r)_{ik} + \beta_j + (\alpha\beta)_{ij}$; $\beta_j \sim N\left(0, \sigma_\beta^2\right)$, $(\alpha\beta)_{ij} \sim N\left(0, \sigma_{\alpha\beta}^2\right)$, $\alpha(r)_{ik} \sim N\left(0, \sigma_{\alpha(r)}^2\right)$ | $\eta_0 + \alpha_i$ | $\eta_0$ |
| Split plot design in RCBD | $\eta_{ijk} = \eta_0 + \alpha_i + r_k + \alpha(r)_{ik} + \beta_j + (\alpha\beta)_{ij}$; $\beta_j \sim N\left(0, \sigma_\beta^2\right)$, $(\alpha\beta)_{ij} \sim N\left(0, \sigma_{\alpha\beta}^2\right)$, $r_k \sim N\left(0, \sigma_r^2\right)$, $\alpha(r)_{ik} \sim N\left(0, \sigma_{\alpha(r)}^2\right)$ | $\eta_0 + \alpha_i$ | $\eta_0$ |

**Example 1** Grain yield of five genotypes evaluated in a randomized complete block design. The data of this experiment are shown in Table 2.2. This example is provided to illustrate the process of estimating the BLUEs of genotypes.

Next, we provide the R code that uses the lme4 library to fit a mixed model for the data given in Table 2.2 to estimate the BLUEs of genotypes:

```
Data_RCBD=read.table("Example_RCBD.csv", header =T, sep = ",")
Data_RCBD

library(lme4)
Data_RCBD$Genotype=as.factor(Data_RCBD$Genotype)
Data_RCBD$Block=as.factor(Data_RCBD$Block)

Fitted=lmer(Yield~ Genotype + (1 | Block), Data_RCBD)
Fitted
####Extracting design matrix of fixed effects (Intercept and Genotype)
X=Fitted@pp$X
X=X[!duplicated(X), ]
X
####Extracting the beta coefficients
Beta=Fitted@beta #fixef(Fitted)
####Obtaining the BLUEs of genotypes
BLUEs_Gen=X%*%Beta
BLUEs_Gen
```

**Table 2.2** Grain yield (Yield) of five genotypes under a randomized complete block design

| Block | Genotype | Yield |
|---|---|---|
| 1 | 1 | 5.25 |
| 1 | 2 | 9 |
| 1 | 3 | 6.25 |
| 1 | 4 | 4.5 |
| 1 | 5 | 5.5 |
| 2 | 1 | 6.5 |
| 2 | 2 | 9.5 |
| 2 | 3 | 6.75 |
| 2 | 4 | 4.25 |
| 2 | 5 | 6.5 |
| 3 | 1 | 4 |
| 3 | 2 | 6.25 |
| 3 | 3 | 5.5 |
| 3 | 4 | 4.5 |
| 3 | 5 | 5.25 |
| 4 | 1 | 7 |
| 4 | 2 | 8.75 |
| 4 | 3 | 6.75 |
| 4 | 4 | 5 |
| 4 | 5 | 6 |

The above code shows that Genotype was specified as a fixed effect, although Block was specified as a random effect. It is important to point out that both effects were converted to factors. The output of this fitted model is given below.

```
> Fitted
Linear mixed model fit by REML ['lmerMod']
Formula: Yield ~ Genotype + (1 | Block)
  Data: Data_RCBD
REML criterion at convergence: 43.1701
Random effects:
 Groups   Name        Std.Dev.
 Block    (Intercept) 0.6922
 Residual             0.6739
Number of obs: 20, groups:  Block, 4
Fixed Effects:
(Intercept)   Genotype2   Genotype3   Genotype4   Genotype5
    5.688       2.688       0.625      -1.125       0.125
> ####Extracting design matrix
> X=Fitted@pp$X
> X=X[!duplicated(X), ]
> X
  (Intercept) Genotype2 Genotype3 Genotype4 Genotype5
1         1         0         0         0         0
5         1         1         0         0         0
9         1         0         1         0         0
13        1         0         0         1         0
17        1         0         0         0         1
```

```
> ####Extracting the fixed effects
> Beta=Fitted@beta
> ####Obtaining the BLUEs of genotypes
> BLUEs_Gen=X%*%Beta
> BLUEs_Gen
     [,1]
1   5.6875
5   8.3750
9   6.3125
13  4.5625
17  5.8125
```

The above code shows that the standard deviation of the random effect of blocks was equal to 0.6922, while the standard deviation of the residual was equal to 0.6739. In the part that reads "Fixed Effects" we find the beta coefficient estimates of the fixed effects with which the BLUEs of each of the genotypes can be obtained. With the Fitted@pp$X, the design matrix of fixed effects is extracted as implemented in lmer, and with X[!duplicated(X),] the rows which are duplicated due to blocks being removed. Then, with Fitted@beta, the beta coefficients for the fixed effects are extracted and, finally, with X%*%Beta, the BLUEs of each genotype are obtained. Since the estimable function (Table 2.1) for genotypes (treatments) are $\eta_0 + \tau_i$, the BLUEs of each genotype are computed as 5.688 + 0 = 5.688 (BLUE of genotype 1), 5.688 + 2.688 = 8.376 (BLUE of genotype 2), 5.688 + 0.625 = 6.313 (BLUE of genotype 3), 5.688 − 1.125 = 4.563 (BLUE of genotype 4), and 5.688 + 0.125 = 5.813 (BLUE of genotype 5). Since for more complex models, obtaining the BLUEs for genotypes can be quite laborious, these can be obtained directly using the following lines of code:

```
library(lsmeans)
Lsmeans_Gen=lsmeans(Fitted,~ Genotype)
#Lsmeans_Gen
BLUEs_Gen=data.frame(GID=Lsmeans_Gen$Genotype,
           BLUEs=Lsmeans_Gen$lsmean,
           SE_BLUEs=Lsmeans_Gen$SE)
BLUEs_Gen
```

In this way we get

```
> BLUEs_Gen
 GID BLUEs SE_BLUEs
1  1 5.6875 0.4830459
2  2 8.3750 0.4830459
3  3 6.3125 0.4830459
4  4 4.5625 0.4830459
5  5 5.8125 0.4830459
```

**Predictable functions** These are linear combinations of the fixed and random effects, $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{\beta} + \boldsymbol{M}^{\mathrm{T}}\boldsymbol{u}$, that is, they can be formed from linear combinations of the conditional means: $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{\beta} + \boldsymbol{M}^{\mathrm{T}}\boldsymbol{u}$ is a predictable function if $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{\beta}$ is estimable. The inference based on these predictable functions is referred to as narrow inference, which, unlike broad inference, has random effects such as additional terms and limits the attention to a group of the sampled random levels (Littell et al. 2006). Then, replacing the estimates obtained from the mixed model equation (2.2) in predictable functions, $\boldsymbol{K}^{\mathrm{T}}\widehat{\boldsymbol{\beta}} + \boldsymbol{M}^{\mathrm{T}}\widehat{\boldsymbol{u}}$ results in the best linear unbiased predictors (BLUPs) of the corresponding predictable function. From a theoretical point of view, BLUP is expected to have a better genotypic predictive accuracy than BLUE, which is important for the selection of new cultivars, or even the genetic values (additive effects) for the selection of progenitors (Piepho et al. 2008). Before BLUP-based selection, selection in crop breeding was based on either simple arithmetic means or BLUEs of genotypes, which can also be calculated in a mixed model context based on fixed genotype effects (Piepho et al. 2008).

BLUP has a long tradition of selection in animal science, but its use in plant breeding is very recent. It is therefore important to provide a clear distinction between the two terms.

Table 2.3 presents the corresponding predictable functions for the same models described in Table 2.1.

It is important to point out that BLUEs and BLUPs are not restricted only to linear mixed models (Eq. 2.1), because they can be obtained in an approximate manner for

**Table 2.3** Some predictable functions for the same models given in Table 2.1

| Model | Predictor | Predictable function |
|---|---|---|
| Complete randomized design (CRD) | $\eta_i = \eta_0 + \tau_i$ | None, since there are no random effects |
| Randomized complete blocks (RCBD) | $\eta_{ij} = \eta_0 + \tau_i + b_j$, $b_j \sim N(0, \sigma_b^2)$ | $\eta_0 + \tau_i + b_j$ |
| Regression | $\eta_i = \eta_0 + \sum_{j=1}^{p} x_{ij}\beta_j$ | None, since there are no random effects |
| Split plot design in a CRD | $\eta_{ijk} = \eta_0 + \alpha_i + \alpha(r)_{ik} + \beta_j + (\alpha\beta)_{ij}$; $\beta_j \sim N\left(0, \sigma_\beta^2\right)$, $(\alpha\beta)_{ij} \sim N\left(0, \sigma_{\alpha\beta}^2\right)$, $\alpha(r)_{ik} \sim N\left(0, \sigma_{\alpha(r)}^2\right)$ | $\eta_0 + \alpha_i + \beta_j + (\alpha\beta)_{ij}$ |
| Split plot design in RCBD | $\eta_{ijk} = \eta_0 + \alpha_i + r_k + \alpha(r)_{ik} + \beta_j + (\alpha\beta)_{ij}$; $\beta_j \sim N\left(0, \sigma_\beta^2\right)$, $(\alpha\beta)_{ij} \sim N\left(0, \sigma_{\alpha\beta}^2\right)$, $r_k \sim N\left(0, \sigma_r^2\right)$ $\alpha(r)_{ik} \sim N\left(0, \sigma_{\alpha(r)}^2\right)$ | $\eta_0 + \alpha_i + \beta_j + (\alpha\beta)_{ij}$ |

**Table 2.4** BLUEs for mean response and BLUPs for conditional mean response for different types of response variables

| Type of response variable | Distribution | Link function | BLUE | BLUP |
|---|---|---|---|---|
| Continuous | Normal | Identity | $\boldsymbol{X}\widehat{\boldsymbol{\beta}}$ | $\boldsymbol{X}\widehat{\boldsymbol{\beta}}+\boldsymbol{Z}\widehat{\boldsymbol{u}}$ |
| Binary | Binomial | Logit | $\frac{1}{1+\exp\left(-\boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)}$ | $\frac{1}{1+\exp\left(-\boldsymbol{X}\widehat{\boldsymbol{\beta}}-\boldsymbol{Z}\widehat{\boldsymbol{u}}\right)}$ |
| Binary | Binomial | Probit | $\Phi\left(\boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)$ | $\Phi\left(\boldsymbol{X}\widehat{\boldsymbol{\beta}}+\boldsymbol{Z}\widehat{\boldsymbol{u}}\right)$ |
| Counts | Poisson | Log | $\exp\left(\boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)$ | $\exp\left(\boldsymbol{X}\widehat{\boldsymbol{\beta}}+\boldsymbol{Z}\widehat{\boldsymbol{u}}\right)$ |
| Counts | Negative binomial | Log | $\exp\left(\boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)$ | $\exp\left(\boldsymbol{X}\widehat{\boldsymbol{\beta}}+\boldsymbol{Z}\widehat{\boldsymbol{u}}\right)$ |
| Continuous proportions | Beta | Logit | $\frac{1}{1+\exp\left(-\boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)}$ | $\frac{1}{1+\exp\left(-\boldsymbol{X}\widehat{\boldsymbol{\beta}}-\boldsymbol{Z}\widehat{\boldsymbol{u}}\right)}$ |
| Continuous positives | Gamma | Inverse | $\frac{1}{\boldsymbol{X}\widehat{\boldsymbol{\beta}}}$ | $\frac{1}{\boldsymbol{X}\widehat{\boldsymbol{\beta}}+\boldsymbol{Z}\widehat{\boldsymbol{u}}}$ |

$\Phi$ is the cumulative distribution function (CDF) of the standard normal distribution

any fitted generalized linear mixed model using the expression BLUP $= g^{-1}\left(\boldsymbol{X}\widehat{\boldsymbol{\beta}} + \boldsymbol{Z}\widehat{\boldsymbol{u}}\right)$ (Stroup 2012). For example, Table 2.4 provides the BLUEs and BLUPs for some of the most popular response variables under a predictor with fixed and random effects.

In sum, the linear combinations of fixed effects only are called estimable functions and give rise to BLUEs. The solution of mixed model equations produces estimates, or BLUEs, for linear combinations of the form $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{\beta}$. Linear combinations of fixed *and* random effects are called predictable functions. Solving the mixed model equations yields *predictors*, or BLUPs, which are used to obtain BLUPs of linear combinations such as $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{\beta} + \boldsymbol{M}^{\mathrm{T}}\boldsymbol{u}$. The best of both BLUEs and BLUPs means that these estimates have minimum mean square errors (see Searle et al. 2006) for the different meanings of the criteria applied to each one.

Both the BLUEs and BLUPs are not possible to be computed in real applications, since true variance–covariance values of $\boldsymbol{R}$ and $\boldsymbol{\Sigma}$ are required, but we only have access to estimates of these variance–covariance matrices. For this reason, only empirical BLUEs and empirical BLUPs are possible, since we use variance–covariance parameter estimates ($\widehat{\boldsymbol{R}}$ and $\widehat{\boldsymbol{\Sigma}}$) to solve the mixed model equations for $\boldsymbol{\beta}$ and $\mathbf{u}$.

Below, we illustrate the calculation of the BLUPs of genotypes of the data set given in Table 2.2 for which the BLUEs of genotypes were obtained. Using the lmer() function again, but now assuming that the genotype is a random effect, instead of fixed effects we obtained the following output of the fitted model. We can see that the standard deviation of genotypes is 1.3575, the standard deviation of blocks is 0.6922, the standard deviation of residuals is 0.6739, and the intercept is equal to 6.15.

```
> #####BLUP of genotypes
> Fitted2=lmer(Yield~ (1|Genotype) + (1 | Block), Data_RCBD)
> Fitted2
```

```
Linear mixed model fit by REML ['lmerMod']
Formula: Yield ~ (1 | Genotype) + (1 | Block)
  Data: Data_RCBD
REML criterion at convergence: 58.8152
Random effects:
 Groups   Name        Std.Dev.
 Genotype (Intercept) 1.3575
 Block    (Intercept) 0.6922
 Residual             0.6739
Number of obs: 20, groups: Genotype, 5; Block, 4
Fixed Effects:
(Intercept)
    6.15
```

From the fitted model (Fitted2), we now extract the intercept with fixef(Fitted2) and the random effects of genotypes with c(ranef(Fitted2)$Genotype); then we sum up these two terms to get the BLUPs of genotypes that we called BLUP_Gen2, and finally, we calculate the correlation between the BLUPs and BLUEs (obtained above) for the same genotypes. This correlation was equal to one, which shows that we should not expect big differences between the use of BLUPs or BLUEs of genotypes, although there are large amounts of empirical evidence showing that the BLUPs should be preferred over the BLUEs and not always are the same results produced by both.

```
> #####Fixed effect=Intercept######
> Intercept=fixef(Fitted2)
> str(Intercept)
Named num 6.15
 - attr(*, "names")= chr "(Intercept)"
> #####Random effects of genoytpes
> U_ref=c(ranef(Fitted2)$Genotype)
> U_ref
$'(Intercept)'
[1] -0.4356567 2.0958619 0.1530686 -1.4953621 -0.3179116
> ########BLUP of Genotypes#####
> BLUP_Gen2=Intercept+U_ref$'(Intercept)'
> BLUP_Gen2
[1] 5.714343 8.245862 6.303069 4.654638 5.832088
> cor(c(BLUEs_Gen),BLUP_Gen2)
[1] 1
```

## 2.3 Marker Depuration

First, we will define markers and their importance. Markers are beneficial in the construction of precise genetic relationships, for parental determination and for the identification and mapping of quantitative trait loci (QTL). Between 1970 and 2001, most of the genetic progress in the livestock industry was reached by using pedigree

and phenotypic information. However, after the first draft of the human genome project was finished in 2001 (The International SNP Map Working Group 2001), the cost of genotyping using single nucleotide polymorphisms (SNPs) started to decrease considerably, and now its cost is at least 1000 times lower. For this reason, Stonecking (2001) points out that SNPs have become the bread and butter of DNA sequence variation and are essential in determining the genetic potential of livestock and plant breeding.

However, it is also important to point out that other types of DNA markers have been discovered, such as restriction fragment length polymorphisms (RFLP), simple sequence repeat (SSR), Diversity Arrays Technology (DArT), simple sequence length polymorphisms (SSLP), amplified fragment length polymorphisms (AFLP), etc. However, SNPs have become the main markers used to detect DNA variation for some of the following reasons: (a) SNPs are abundant and found throughout the entire genome, in intragenic and extragenic regions (Schork et al. 2000), (b) they represent the most common genetic variants, (c) the location in the DNA: they are found in introns, exons, promoters, enhancers, or intergenic regions, (d) they are easily evaluated by automated means, (e) many of them have direct repercussions on traits of interest in plant and animals, (f) they are generally biallelic, and (g) they are now cheap and easy to genotype.

It is important to remember that DNA (deoxyribonucleic acid) is organized in pairs of chromosomes, each inherited from one of the parents. The diversity found among organisms is a result of variations in DNA sequences and of environmental effects. Genetic variation is substantial and each individual of a species, with the exception of monozygotic twins, possesses a unique DNA sequence. DNA variations are mutations resulting from the substitution of single nucleotides (single nucleotide polymorphisms—SNPs), the insertion or deletion of DNA fragments of various lengths (from a single to several thousand nucleotides), or the duplication or inversion of DNA fragments (Marsjan and Oldenbroek 2007). For this reason, the genome is composed of four different nucleotides (A, C, T, and G). Next, we provide two important definitions that are keys to understanding how markers are used in genomic selection.

**Genetic markers** A genetic marker is a gene or DNA sequence with a known location on a chromosome and is generally used to identify individuals, which is why it is a powerful tool to explore genetic diversity. It can be described as a variation that may arise due to a mutation or alteration in the genomic loci that can be observed. A genetic marker may be a short DNA sequence, such as a sequence surrounding a single base-pair change (single nucleotide polymorphism, SNP, see Fig. 2.1), or a long one, such as mini- and microsatellites. Molecular markers can be used in molecular biology and biotechnology to identify a particular DNA sequence in a pool of unknown DNA. For example, DNA is used to search for useful genes, and also for marker-assisted selection, paternity testing, and food traceability. In GS, genetic markers measured across the genome are used to measure genomic similarities between individuals; in theory, this can be more precise than pedigree information.

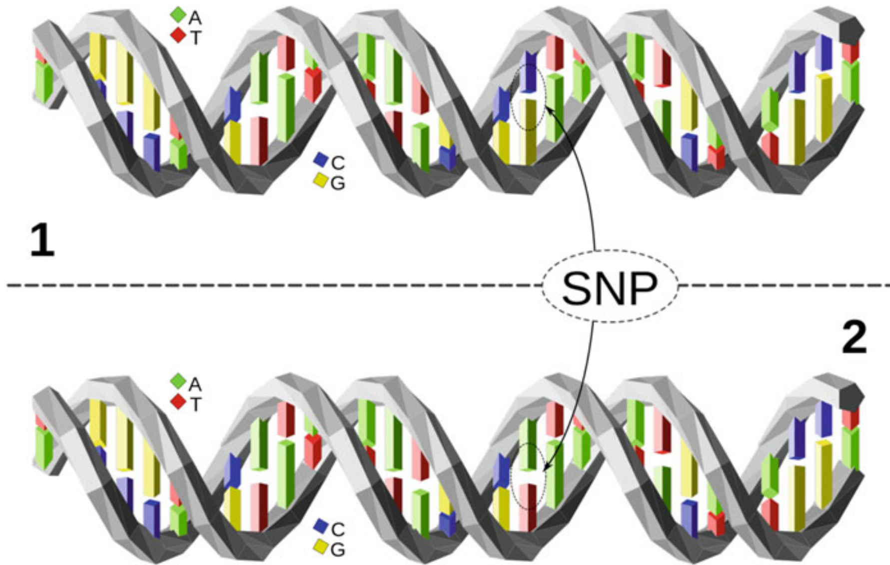



**Fig. 2.1** The upper DNA molecule differs from the lower DNA molecule at a single base-pair location (a C/A polymorphism)

Markers can be used to estimate the proportion of chromosome segments shared by individuals, including the identification of genes that are identical by state (IBS). It is important to point out that probabilities generated from pedigree (A matrix) are discrete between close relatives. For example, full sibs share 0.5 of alleles (genome) that are identical by descent, that is, that are inherited from a common ancestor. A simple nucleotide polymorphism (SNP) is a widely used marker.

As an initial descriptive analysis of this type of data that could help delve into some characteristics of the structure of the genotypes data, we can compute the frequency genotypes in our data, basically the proportion of each genotype, and can help us as description of the gene variation. Rather, a more common description of the genetic variation is the allele frequency, which is the proportion of each allele present in the population, and when the individual is a diploid, its contribution of the proportion is of two alleles for each gene (Griffiths et al. 2005). In a diploid population with two alleles, A and a, if $n_{\mathrm{A}}$, $n_{\mathrm{Aa}}$, and $n_{\mathrm{a}}$ are the frequencies of the three genotypes (AA, Aa, aa) in a sample of $n$ individuals, the frequency of alleles A and a are given by $\frac{2n_{\mathrm{A}}+n_{\mathrm{Aa}}}{2n}=\frac{n_{\mathrm{A}}}{n}+\frac{n_{\mathrm{Aa}}}{2n}$ and $\frac{2n_{\mathrm{a}}+n_{\mathrm{Aa}}}{2n}=\frac{n_{\mathrm{a}}}{n}+\frac{n_{\mathrm{Aa}}}{2n}$, respectively. In this case, the allele with the lower frequency is called the minor allele and the other, the major allele.

To illustrate marker recoding and depuration, consider the following example of eight plants genotyped for seven SNPs.

With the information in Table 2.5, we first proceed to find the minor (less common) and major (most common) alleles of each marker. In marker 1 (SNP1),

**Table 2.5** Marker information for eight plants and seven SNPs denoted as SNP1, . . ., SNP7

| Plant | SNP1 | SNP2 | SNP3 | SNP4 | SNP5 | SNP6 | SNP7 |
|---|---|---|---|---|---|---|---|
| 1 | C_G | C_C | T_T | G_T | G_G | T_C | G_G |
| 2 | C_G | C_G | A_A | G_T | C_C | C_C | G_C |
| 3 | C_C | ?_? | T_A | G_T | G_C | T_C | ?_? |
| 4 | G_G | ?_? | T_T | T_T | G_C | T_C | G_C |
| 5 | C_C | C_G | T_A | G_T | G_C | T_T | C_C |
| 6 | ?_? | C_G | T_A | G_T | C_C | ?_? | G_G |
| 7 | C_G | C_C | T_A | G_G | G_C | C_C | G_C |
| 8 | C_G | C_C | T_A | G_G | G_C | T_C | G_G |

?_? denotes a missing genotype

**Table 2.6** Minor allele (minorAllele) and major allele (majorAllele) for each marker

| Marker | minorAllele | majorAllele |
|---|---|---|
| SNP1 | G | C |
| SNP2 | G | C |
| SNP3 | A | T |
| SNP4 | T | G |
| SNP5 | G | C |
| SNP6 | T | C |
| SNP7 | C | G |

we can see that the minor allele is G, since it only appears in G_G in one out of the eight plants; the major allele is C, since C_C appears in two out of eight plants. In SNP2, the minor allele is G since it does not appear in any of the eight plants, whereas C_C appears in three out of eight plants. Due to this, C is the major allele. In SNP3, the minor and major alleles are A (with A_A observed in 1/8) and T (with T_T observed in 2/8), respectively. Using this logic, Table 2.6 shows the minor and major alleles of each of the markers.

Once we have this information (minor and major alleles), it can be used to fit additive effects models, but it is first necessary to recode the marker information following the rules below:

$$x = \begin{cases} 0 \text{ if the SNP is homozygous for the major allele} \\ 1 \text{ if the SNP is heterozygous} \\ 2 \text{ if the SNP is homozygous for the other allele} \end{cases}$$

The recoded information in terms of additive effects is given in Table 2.7, where the recording is now in terms of 0, 1, and 2, following the above rules. In turn, the missing genotypes are recoded as NA.

Next, we show the minor allele frequency (MAF), the frequency of NAs and the frequency of heterozygotes genotypes (freHetero) for each marker. The frequency allele can be computed from the frequency of genotypes as described earlier and from there, the minor allele and its frequency can be deduced, but once the marker

**Table 2.7** Marker information recoded as 0, 1, and 2, for eight plants and seven SNPs denoted as SNP1, . . ., SNP7

| Plant | SNP1 | SNP2 | SNP3 | SNP4 | SNP5 | SNP6 | SNP7 |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 1 | 2 | 1 | 0 |
| 2 | 1 | 1 | 2 | 1 | 0 | 0 | 1 |
| 3 | 0 | NA | 1 | 1 | 1 | 1 | NA |
| 4 | 2 | NA | 0 | 2 | 1 | 1 | 1 |
| 5 | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
| 6 | NA | 1 | 1 | 1 | 0 | NA | 0 |
| 7 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 8 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

NA denotes a missing genotype

**Table 2.8** Minor allele frequency (MAF), frequency of NAs (freqNA), and frequency of heterogeneous (freqHetero) are reported for each marker

| Marker | minorAllele | majorAllele | MAF | freqNA | freqHetero |
|---|---|---|---|---|---|
| SNP1 | G | C | 0.429 | 0.125 | 0.571 |
| SNP2 | G | C | 0.250 | 0.250 | 0.500 |
| SNP3 | A | T | 0.438 | 0.000 | 0.625 |
| SNP4 | G | T | 0.438 | 0.000 | 0.625 |
| SNP5 | G | C | 0.438 | 0.000 | 0.625 |
| SNP6 | T | C | 0.429 | 0.125 | 0.571 |
| SNP7 | C | G | 0.357 | 0.125 | 0.429 |

information is coded as Table 2.7, in an equivalent way, the MAF also can be calculated as the mean of each column in Table 2.7 divided by 2 without taking into account the missing values, that is, as $\text{MAF} = \left(\sum_{i=1}^{n_c} x_i^*\right)/(2n_c)$, where $x_i^*, i = 1, \ldots, n_c$ are the coded genotyped values for non-missing individual values for a marker. For example, the MAF of marker SNP2 is equal to $\text{MAF}_{\text{SNP2}} = \frac{0+1+1+1+0+0}{2(6)} = \frac{3}{12} = 0.25$. See Table 2.8 for the MAF of the remaining markers, where the frequency of NAs (freqNA) is also reported for each marker, which corresponds to the number of missing values (NAs) in each column divided by the number of individuals (8). Furthermore, the frequency of heterogeneous genotypes (freqHetero) was calculated (ratio of the summation of only the ones divided by the non-missing values).

With the data formatted in this way, we are ready to compute a genomic relationship matrix or matrix of realized genetic similarities among all pairs of individuals.

It is important to point out that the recoding process for values of 0, 1, and 2 can be performed automatically using the library synbreed, but we first need to load the complete information onto R of Table 2.5. Then, for recording and imputing, we used methods available from this library; the code used for these tasks is given in Appendix 1. The output is explained below.

First, we called the library synbreed and then we loaded the marker information contained in a file called MarkersToy.csv, which is saved in an object called snp7. When all the marker information is printed, we can clearly see that it corresponds to the information given in Table 2.5, although without the first column.

```
> library(synbreed)
> ####Loading the marker information
> snp7 <- read.csv("MarkersToy.csv",header=T)
> snp7=snp7[,-1]
> snp7
  SNP1 SNP2 SNP3 SNP4 SNP5 SNP6 SNP7
1 C_G  C_C  T_T  G_T  G_G  T_C  G_G
2 C_G  C_G  A_A  G_T  C_C  C_C  G_C
3 C_C  ?_?  T_A  G_T  G_C  T_C  ?_?
4 G_G  ?_?  T_T  T_T  G_C  T_C  G_C
5 C_C  C_G  T_A  G_T  G_C  T_T  C_C
6 ?_?  C_G  T_A  G_T  C_C  ?_?  G_G
7 C_G  C_C  T_A  G_G  G_C  C_C  G_C
8 C_G  C_C  T_A  G_G  G_C  T_C  G_G
```

Next, we rename the rows of the object snp7 (matrix of marker information) coded with values of ID1 to ID8. We then select the position in this matrix of values equal to ?_?, followed by these values being replaced with NA. Finally, the snp7 object is printed again and we can see that the values of ?_? were replaced by NAs.

```
> ####Set names for individuals
> rownames(snp7) <- paste("ID",1:8,sep="")
> pos.NA=which(snp7=="?_?", arr.ind=TRUE)
> snp7[pos.NA]=NA
> snp7
    SNP1 SNP2 SNP3 SNP4 SNP5 SNP6 SNP7
ID1 C_G  C_C  T_T  G_T  G_G  T_C  G_G
ID2 C_G  C_G  A_A  G_T  C_C  C_C  G_C
ID3 C_C  NA   T_A  G_T  G_C  T_C  NA
ID4 G_G  NA   T_T  T_T  G_C  T_C  G_C
ID5 C_C  C_G  T_A  G_T  G_C  T_T  C_C
ID6 NA   C_G  T_A  G_T  C_C  NA   G_G
ID7 C_G  C_C  T_A  G_G  G_C  C_C  G_C
ID8 C_G  C_C  T_A  G_G  G_C  T_C  G_G
```

Later, the object with the marker information, snp7, is transformed into an object of class gpData.

```
> ####Creating an object of class 'gpData'
> gp <- create.gpData(geno=snp7)
```

Using the function codeGeno() and giving the marker object, gp, as an input, we recode the marker information to values of 0, 1, and 2. The recoded marker information is then extracted and printed, and here we can see that the recoding

performed with this library is exactly equal to what we obtained manually and presented in Table 2.7.

```
> ####Recoding to 0, 1, and 2 values the genotypic data
> gp.coded <-codeGeno(gp)
> Geno_Recoded=gp.coded$geno
> Geno_Recoded
     SNP1 SNP2 SNP3 SNP4 SNP5 SNP6 SNP7
ID1  1     0    0    1    2    1    0
ID2  1     1    2    1    0    0    1
ID3  0     NA   1    1    1    1    NA
ID4  2     NA   0    2    1    1    1
ID5  0     1    1    1    1    2    2
ID6  NA    1    1    1    0    NA   0
ID7  1     0    1    0    1    0    1
ID8  1     0    1    0    1    1    0
```

Finally, we once again use the function codeGeno(), but we also add input=T and inpute.type="random", which will recode the object gp to values 0, 1, and 2, as done previously. However, it will also input the missing cells with NAs using the random method of imputation. Finally, the matrix of markers coded with values 0, 1, and 2 is presented, but with the NAs inputted with values of 0, 1, and 2 using the random method. It is important to point out that this library has other imputation options such as "family," "beagle," "beagleAfterFamily," "beagleNoRand," "beagleAfterFamilyNoRand," and "fix," but the technical details of each imputation method go beyond the scope of this book.

```
> ####Recoding to values of 0, 1, and 2 the genotypic data and inputting
> Imputed_Geno<-codeGeno(gp,impute=T,impute.type="random")
> Imputed_Geno$geno
     SNP1 SNP2 SNP3 SNP4 SNP5 SNP6 SNP7
ID1  1    0    0    1    2    1    0
ID2  1    1    2    1    0    0    1
ID3  0    0    1    1    1    1    2
ID4  2    0    0    2    1    1    1
ID5  0    1    1    1    1    2    2
ID6  2    1    1    1    0    0    0
ID7  1    0    1    0    1    0    1
ID8  1    0    1    0    1    1    0
```

## 2.4 Methods to Compute the Genomic Relationship Matrix

The three methods described here to calculate the genomic relationship matrix (GRM) are based on VanRaden's (2008) paper "Efficient methods to compute genomic predictions" where more theoretical support for each of these methods can be found. We assume that we have a matrix of markers of order $J \times p$, where $J$ denotes the number of lines and $p$ the number of markers, and that this matrix does

not contain missing values and is coded as 0, 1, and 2, or −1, 0, and 1 to refer homozygotes major allele, heterozygous, and homozygous minor allele, respectively. Note that the last codification is related to the first by the relation $\boldsymbol{X}_2 = \boldsymbol{X} + \mathbf{1}_J\mathbf{1}_p^{\mathrm{T}}$, where $\boldsymbol{X}_2$ is a matrix of markers information coded in terms of −1, 0, and 1, while $\boldsymbol{X}$ is the coded marker information in terms of 0, 1, and 2, and $\mathbf{1}_q$ is the column vector of dimension $q$ with ones in all its entries.

Method 1. This method calculates the GRM as

$$\boldsymbol{G} = \frac{1}{p}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}},$$

where $\boldsymbol{X}$ is the matrix of marker genotypes of dimensions $J \times p$. When the marker information is coded as −1, 0, and 1 as described before, the diagonal terms of $p\boldsymbol{G}$ count the number of homozygous loci for each line, and the off-diagonal of $p\boldsymbol{G}$ is a measure of the number of alleles shared by two lines (VanRaden 2008). To illustrate how to calculate the GRM under this method, we will use the matrix of marker genotypes obtained in the previous section with Imputed_Geno$geno, which in the matrix format is equal to

$$\boldsymbol{X} = \begin{bmatrix} 1 & 0 & 0 & 1 & 2 & 1 & 0 \\ 1 & 1 & 2 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 2 \\ 2 & 0 & 0 & 2 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 2 & 2 \\ 2 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Then, using the R code, we calculate the GRM under this first method as

```
##Computing the genomic relationship matrix—Method1
> G_M1=tcrossprod(X)/dim(X)[2]
> G_M1
     ID1   ID2   ID3   ID4   ID5   ID6   ID7   ID8
ID1 0.875 0.250 0.500 0.875 0.625 0.375 0.375 0.500
ID2 0.250 1.000 0.625 0.625 0.750 0.750 0.500 0.375
ID3 0.500 0.625 1.000 0.750 1.125 0.250 0.500 0.375
ID4 0.875 0.625 0.750 1.375 0.875 0.750 0.500 0.500
ID5 0.625 0.750 1.125 0.875 1.500 0.375 0.500 0.500
ID6 0.375 0.750 0.250 0.750 0.375 0.875 0.375 0.375
ID7 0.375 0.500 0.500 0.500 0.500 0.375 0.500 0.375
ID8 0.500 0.375 0.375 0.500 0.500 0.375 0.375 0.500
```

Method 2. In this method the GRM is similar to method 1, but first each marker is centered by twice the minor allele frequency:

$$\boldsymbol{G} = \frac{(\boldsymbol{X}-\boldsymbol{\mu_E})(\boldsymbol{X}-\boldsymbol{\mu_E})^{\mathrm{T}}}{2\sum_{j=1}^{p} p_j\left(1-p_j\right)},$$

where $p_j$ is the minor allele frequency (MAF) of SNP $j = 1, \ldots, p$ and $\boldsymbol{\mu_E}$ is the expected value of matrix $\boldsymbol{X}$ under the Hardy–Weinberg equilibrium (Griffiths et al. 2005) from estimates of allelic frequencies, that is, $\boldsymbol{\mu_E} = \mathbf{1}_{\boldsymbol{J}}[2p_1, \ldots, 2p_p]$. Term $2\sum_{j=1}^{p} p_j\left(1-p_j\right)$ is the sum of the variance estimates of each marker and makes GRM analogous to the numerator relationship matrix (VanRaden 2008).

Now, using the following R code, we calculate the GRM under method 2 as

```
##Computing the genomic relationship matrix—Method2
> phat=colMeans(X)/2#Minor allele frequency
> phat
 SNP1   SNP2   SNP3   SNP4   SNP5   SNP6   SNP7
 0.5000 0.1875 0.4375 0.4375 0.4375 0.3750 0.4375

> X2=scale(X,center=TRUE,scale=FALSE)
> k=2*sum(phat*(1-phat))
> G_M2=tcrossprod(X2)/k

> round(G_M2,3)
       ID1    ID2    ID3    ID4    ID5    ID6    ID7    ID8
ID1  0.930 -0.766 -0.227  0.352 -0.265 -0.227 -0.072  0.275
ID2 -0.766  0.930 -0.072 -0.419 -0.111  0.545  0.082 -0.188
ID3 -0.227 -0.072  0.776 -0.188  0.737 -0.766  0.005 -0.265
ID4  0.352 -0.419 -0.188  1.007 -0.227  0.120 -0.342 -0.304
ID5 -0.265 -0.111  0.737 -0.227  1.316 -0.805 -0.342 -0.304
ID6 -0.227  0.545 -0.766  0.120 -0.805  1.084  0.005  0.043
ID7 -0.072  0.082  0.005 -0.342 -0.342  0.005  0.467  0.198
ID8  0.275 -0.188 -0.265 -0.304 -0.304  0.043  0.198  0.545
```

Method 3. Under this method, the GRM should be calculated as

$$\boldsymbol{G} = \frac{\boldsymbol{Z}\boldsymbol{Z}^{\mathrm{T}}}{p},$$

where $\boldsymbol{Z}$ is the matrix of scaled SNP codes and $p$ is the number of SNPs, that is $z_{ij} = \left(x_{ij} - 2p_j\right)/\sqrt{2p_j\left(1-p_j\right)}$.

Finally, for this third method, the GRM can be calculated as

```
> ##Computing the genomic relationship matrix—Method3
> X3=scale(X,center=TRUE,scale=TRUE)
> G_M3=tcrossprod(X3)/ncol(X3)
> round(G_M3,3)
       ID1    ID2    ID3    ID4    ID5    ID6    ID7    ID8
ID1  0.962 -0.880 -0.093  0.435 -0.221 -0.397 -0.028  0.223
ID2 -0.880  1.084 -0.133 -0.507 -0.014  0.667  0.012 -0.228
ID3 -0.093 -0.133  0.619 -0.112  0.490 -0.658  0.023 -0.136
ID4  0.435 -0.507 -0.112  1.058 -0.241  0.022 -0.350 -0.305
ID5 -0.221 -0.014  0.490 -0.241  1.181 -0.539 -0.391 -0.265
ID6 -0.397  0.667 -0.658  0.022 -0.539  1.053 -0.057 -0.092
ID7 -0.028  0.012  0.023 -0.350 -0.391 -0.057  0.516  0.276
ID8  0.223 -0.228 -0.136 -0.305 -0.265 -0.092  0.276  0.527
```

## 2.5 Genomic Breeding Values and Their Estimation

In plant and animal breeding, it is a common practice to rank and select individuals (plants or animals) based on their true breeding values (TBVs), also called additive genetic values. However, since we cannot see genes and breeding values, this task is not straightforward, and it is therefore estimated indirectly using observed phenotypes. The estimated values are called estimated breeding values (EBVs), which means that TBV is a latent variable that is only approximated using the observable variable (phenotype).

When the TBVs are used, the genetic change is expected to be larger than when the EBVs are used, but this difference is small when the EBVs are accurately estimated. EBVs reflect the true genetic potential or true genetic transmitting ability of individuals (plants or animals). Traditionally, they are estimated based on the performance records of their parents, sibs, progenies, and their own after correcting for various environmental factors such as management, season, age, etc. When parents are selected based on their breeding values with high reliability, a faster genetic progress is expected in the resulting population. For this reason, the process of estimating breeding values is of paramount importance in any breeding program.

There are several methods to estimate genomic estimated breeding values (GEBVs), but first we will describe the best linear unbiased predictor (BLUP) method. When using the BLUP method to estimate the GEBVs, we need to use the mixed model equations (2.2) described above to estimate BLUEs and BLUPs. Using this equation (2.2) but depending on the form taken by the matrices $\boldsymbol{Z}$ and $\boldsymbol{\Sigma}$, we can end up with the GBLUP method or the SNP-BLUP method to estimate the breeding values. First, we explain the GBLUP method, where we substitute $\boldsymbol{Z}$ and $\boldsymbol{\Sigma}$ matrices for the incidence matrix of genotypes and genomic relationship matrix (GRM) derived from allele frequencies calculated with one of the methods of

VanRaden (2008) given in Sect. 2.4. Under this GBLUP method, the GEBV can be obtained as the solution $\widehat{\boldsymbol{u}}$ of the mixed model equation:

$$\begin{pmatrix} \widehat{\boldsymbol{\beta}} \\ \widehat{\boldsymbol{u}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{X} & \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\mathbf{1} \\ \mathbf{1}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{X} & \mathbf{1}^{\mathrm{T}}\boldsymbol{R}^{-1}\mathbf{1} + \sigma_g^{-2}\,\boldsymbol{G}^{-1} \end{pmatrix}^{-1} \begin{pmatrix} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{y} \\ \mathbf{1}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{y} \end{pmatrix}, \tag{2.3}$$

where $\boldsymbol{Z}$ was replaced by $\boldsymbol{Z} = \mathbf{1}$ and $\boldsymbol{\Sigma}$ by $\sigma_g^2\boldsymbol{G}$, the genomic relationship matrix that was calculated with some of the methods described in Sect. 2.4 and the genomic variance component ($\sigma_g^2$) should be estimated. We ended up with the system of equations given in Eq. (2.3), since the model used is $\boldsymbol{Y} = \boldsymbol{X\beta} + \boldsymbol{u} + \boldsymbol{\varepsilon}$, with $\boldsymbol{u} \sim N\left(\mathbf{0}, \sigma_g^2\boldsymbol{G}\right)$, where $\sigma_g^2$ is the genomic variance component, $\boldsymbol{G}$ is a GRM of dimension $q \times q$ calculated using any of the three methods given in Sect. 2.4, and the other terms are exactly as in Eq. (2.2). One of the greatest advantages of using the GBLUP method to obtain GEBV is that the dimensionality of the design matrices is, at most, equal to the number of lines under study. On the other hand, under the SNP-BLUP, we substitute the $\boldsymbol{Z}$ and $\boldsymbol{\Sigma}$ matrices in Eq. (2.2) with $\boldsymbol{M}$ (scaled marker information matrix of order $n \times p$) and $\sigma_M^2\boldsymbol{I}$, respectively. Under this SNP-BLUP method, the mixed model equation is equal to

$$\begin{pmatrix} \widehat{\boldsymbol{\beta}} \\ \widehat{\boldsymbol{u}} \end{pmatrix} = \begin{pmatrix} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{X} & \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{M} \\ \boldsymbol{M}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{X} & \boldsymbol{M}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{M} + \sigma_M^{-2}\boldsymbol{I}^{-1} \end{pmatrix}^{-1} \begin{pmatrix} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{y} \\ \boldsymbol{M}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{y} \end{pmatrix} \tag{2.4}$$

Under this mixed model equation, $\boldsymbol{u}$ is now the random effects of markers, and therefore, to obtain the GEBV, we use the estimates of marker effects ($\widehat{\boldsymbol{u}}$) and GEBV $= \boldsymbol{M}\widehat{\boldsymbol{u}}$, since now the model used is $\boldsymbol{Y} = \boldsymbol{X\beta} + \boldsymbol{Mu} + \boldsymbol{\varepsilon}$, with $\boldsymbol{u} \sim N\left(\mathbf{0}, \sigma_M^2\boldsymbol{I}\right)$.

We then illustrate how to estimate the GEBV under both BLUP methods. For this purpose, we provide, in Table 2.9, a data set with eight lines (evaluated in two environments) for grain yield, for which, in turn, we use the seven markers imputed in Sect. 2.3.

Below are the data of Table 2.9 that were saved in the data.for.GEBV.csv file. Library rrBLUP was used to estimate the GEBV using the mixed model equations.

```
> library(rrBLUP)
> data=read.csv("data.for.GEBV.csv")
> data
  X Env Lines y      SNP1 SNP2 SNP3 SNP4 SNP5 SNP6 SNP7
1 1 E1  L1    5.215  1    0    0    1    2    1    0
2 2 E1  L2    4.998  1    1    2    1    0    0    1
3 3 E1  L3    5.284  0    0    1    1    1    1    2
4 4 E1  L4    5.157  2    0    0    2    1    1    1
5 5 E2  L5    6.601  0    1    1    1    1    2    2
6 6 E2  L6    5.735  2    1    1    1    0    0    0
7 7 E2  L7    5.565  1    0    1    0    1    0    1
8 8 E2  L8    5.829  1    0    1    0    1    1    0
```

**Table 2.9** Grain yield ($y$) of eight lines in two environments

| Env | Lines | $y$ | SNP1 | SNP2 | SNP3 | SNP4 | SNP5 | SNP6 | SNP7 |
|---|---|---|---|---|---|---|---|---|---|
| E1 | L1 | 5.215 | 1 | 0 | 0 | 1 | 2 | 1 | 0 |
| E1 | L2 | 4.998 | 1 | 1 | 2 | 1 | 0 | 0 | 1 |
| E1 | L3 | 5.284 | 0 | 0 | 1 | 1 | 1 | 1 | 2 |
| E1 | L4 | 5.157 | 2 | 0 | 0 | 2 | 1 | 1 | 1 |
| E2 | L5 | 6.601 | 0 | 1 | 1 | 1 | 1 | 2 | 2 |
| E2 | L6 | 5.735 | 2 | 1 | 1 | 1 | 0 | 0 | 0 |
| E2 | L7 | 5.565 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| E2 | L8 | 5.829 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

In matrix M, only the columns corresponding to marker information are selected. This information is scaled by column using the scale command of R, and the scaled markers are saved in the MS matrix; the GRM is calculated with this information using method 3 in Sect. 2.4. Here we obtained the design matrix for environments and lines, and we also added the genomic information to the design matrix of lines by post-multiplying the design matrix of lines by the Cholesky decomposition of the GRM, which also can be used as an alternative way to obtain the breeding values. This is because the GBLUP model (2.1) can be expressed equivalently as follows:

$$\boldsymbol{Y} = \boldsymbol{X\beta} + \boldsymbol{Z}^{*}\boldsymbol{u}^{*} + \boldsymbol{\varepsilon},$$

where now $\boldsymbol{Z}^{*} = \boldsymbol{ZL}^{\mathrm{T}}$, $\boldsymbol{G} = \boldsymbol{L}^{\mathrm{T}}\boldsymbol{L}$ is the Cholesky decomposition of $\boldsymbol{G}$, and $\boldsymbol{u}^{*}$ is a random vector with distribution $N\left(\boldsymbol{0}, \sigma_g^2\boldsymbol{I_q}\right)$.

```
> M=data[,5:11]
> MS=scale(M) #Scales matrix of markers
> G=MS%*%t(MS)/ncol(MS) #Genomic relationship matrix method 3
> X_E=model.matrix(~0+Env,data=data) #Matrix design of environments
> X_L1=model.matrix(~0+Lines,data=data)#Matrix design of lines
> L=chol(G) #Cholesky decomposition of G
> X_L=X_L1%*%t(L)#Modified matrix design of lines, Z*
```

Then, using the mixed.solve() function of the rrBLUP package, providing as input the response variable (y), the X_L1 matrix of lines and the GRM matrix, $\boldsymbol{G}$, we solved the mixed model equation (2.3) and obtained the GEBVs, which are extracted using fm1$u.

```
> #######################Solution GBLUP###########################
> y = data$y
> fm1=mixed.solve(y=y, Z=X_L1, K=G, X=X_E, method="REML",
+       bounds=c(1e-09, 1e+09), SE=FALSE, return.Hinv=FALSE)
> fm1$u
[1] 0.08371533 -0.13384553 0.15177641 0.02540245 0.63649420
-0.22942366 -0.39877555 -0.13534366
###################Alternative solution GBLUP######################
```

```
> fm1a=mixed.solve(y=y, Z=X_L, K=diag(dim(G)[1]), X=X_E,
method="REML",
        bounds=c(1e-09, 1e+09), SE=FALSE, return.Hinv=FALSE)
>#GEBV
>X_L%*%fm1a$u
```

Shown below is the code to obtain the GEBV, but using the SNP-BLUP method and also using the mixed.solve() function. However, instead of giving the GRM as input, this is given by $\boldsymbol{Z}$, which is the MS matrix, containing the matrix of the markers scaled. Now the random effects of markers are extracted with fm2$u, and to obtain the GEBV, these random effects are pre-multiplied by the scaled design matrix of markers:

```
> ######################Solution SNP-BLUP########################
> fm2=mixed.solve(y=y, Z=MS, X=X_E, method="REML",
+       bounds=c(1e-09, 1e+09), SE=FALSE, return.Hinv=FALSE)
> fm2$u
SNP1         SNP2        SNP3         SNP4        SNP5        SNP6        SNP7
-0.121863421 0.115618300 -0.040677892 0.083526359 0.016201822 0.184761140 -0.001827558
> beta_Mar=fm2$u
> GEBV=c(MS%*%beta_Mar)
> GEBV
[1] 0.08373722 -0.13385647 0.15181358 0.02538892 0.63650107 -0.22940375 -0.39883111
-0.13534946
```

This shows that both methods (GBLUP and SNP-BLUP), give exactly the same breeding value estimates. However, although the two methods are specified in almost the same way and give similar estimates, which, based on all the SNPs (SNP-BLUP), required more computational time, this difference will be more notable for larger data sets that containing larger number of markers. Due to this, in these situations we can choose the GBLUP method.

Some advantages of using the Henderson equation to obtain the GEBV are

(a) It fits nicely into existing BLUP software and into existing theory.
(b) It provides measures of accuracy from the inverse of the LHS (Linear Henderson system).
(c) It accommodates all individuals (plants or animals).

However, it has some inconveniences, such as

(a) It can't easily accommodate major genes (unless using weights in the construction of $\boldsymbol{G}$).
(b) Computation of $\boldsymbol{G}$ and inversion might be challenging.

It is important to point out that the GEBV can be obtained using other estimation methods such as Bayesian methods. Bayesian methods for GS are explained in detail in upcoming chapters, but here we will illustrate the use of the BGLR package to estimate the GBLUP Bayesian method and the SNP-BLUP Bayesian method.

We first provide the code for the GBLUP Bayesian method. The predictor ETA is a list and has two sub-list components: the first, for the effects of environments for which a FIXED model (model="FIXED") is used that uses a prior non-informative for each beta coefficient. The second component is an RKHS model to specify the distribution of the random effects of lines that, in general, are of the form $N\left(\mathbf{0}, \sigma_g^2\boldsymbol{K}\right)$, and in this case, using the GRM, $\boldsymbol{K} = \boldsymbol{G}$. Then the Bayesian GBLUP model is fitted using the function BGLR(), and finally, the GEBVs are obtained with fm1$ETA$Gen$u. The response vector value is specified in the first option of the function BGLR, y=y, and with nIter=20000 and burnIn=10000, the number of iterations to run the involved MCMC method are specified, along with the number of these that will be discarded at the beginning of the MCMC, respectively.

```
> library(BGLR)
> ###################GBLUP-BLUP Bayesian########################
> ETA=list(Env=list(X=X_E[,-1], model="FIXED"),Gen=list(K=G,
model="RKHS"))
> fm1=BGLR(y=y,ETA=ETA,nIter=20000,burnIn=10000,verbose=F)
> fm1$ETA$Gen$u
[1] 0.02581315 -0.08430838 0.12806091 0.03949064 0.32406895 -0.17367228
[7] -0.16452623 -0.09492676
```

The SNP-BLUP Bayesian GEBV is then fitted, but now also providing the design matrix of environments as input, and the scaled design matrix of markers in the second term. The model now used is a Bayesian Ridge regression (BRR) that gives a normal distribution with mean zero and a common variance component as a prior for each marker effect.

```
> #######################SNP-BLUP Bayesian######################
> ETA=list(Env=list(X=X_E[,-1], model="FIXED"), Gen=list(X=MS,
model="BRR"))
> fm1=BGLR(y=y,ETA=ETA,nIter=20000,burnIn=10000,verbose=F)
> beta_Mar_Bayes=fm1$ETA$Gen$b
>
> GEBV_Bayes=c(MS%*%beta_Mar_Bayes)
> GEBV_Bayes
[1] 0.03187469 -0.09663427 0.12920479 0.04973765 0.34695500
-0.18172760
[7] -0.18217036 -0.09723989
```

Here we can observe that both methods gave GEBVs that are very similar yet slightly different due to the Monte Carlo sampling. However, these Bayesian GEBVs are different from those obtained above using Henderson's mixed model equations, due to the fact that different machineries are used to estimate the GEBVs. Details of the Bayesian methods for GS will be provided in upcoming chapters.

Finally, it is important to point out that the advantage of the GBLUP over the SNP-BLUP is that the system of equations, when fitting the mixed model equations, is of the size of the individuals (lines or animals), which are, most of the time, fewer

than the number of markers (SNPs). This advantage is also observed under the Bayesian version, because the design matrix of markers is usually larger than the dimension of the genomic relationship matrix.

## 2.6 Normalization Methods

This section describes four types of normalization variables (inputs and outputs). In this case, normalization refers to the process of adjusting the different inputs or outputs that were originally measured in different scales to the same scale. It is very important to carry out the normalization process before giving the inputs and outputs for most statistical machine learning algorithms because it helps improve the numerical stability in the estimation process of some algorithms; it is suggested mostly when the inputs or outputs are in different scales. However, it is important to point out that in some statistical machine learning software, the normalization process is done internally, in which case this process does not need to be carried out manually. The five normalization methods we describe next are centering, scaling, standardization, max normalization, and minimax normalization.

**Centering** This normalization consists of subtracting from each variable (input or output) its mean, $\mu$; this means that the centered values are calculated as

$$X_i^* = X_i - \mu$$

The centered variable $X_i^*$ has a mean of zero.

**Scaling** This normalization consists of dividing each variable (input or output) by its standard deviation, $\sigma$. The scaled values are calculated as

$$X_i^* = \frac{X_i}{\sigma}.$$

The scaled variable $X_i^*$ has unit variance.

**Standardization** This process of normalization consists of calculating its mean, $\mu$, and standard deviation, $\sigma$, for each input or output. The standardized values are then calculated as

$$X_i^* = \frac{X_i - \mu}{\sigma}.$$

This process is carried out for each input or output variable, and this needs to be done with care, since we need to use the corresponding mean and standard deviation of each variable. The output of the standardized score has a mean of zero and a variance of one, which means that most standardized values range between −3.5 and 3.5.

**Max normalization** This normalization consists of dividing the values of the input or output by the maximum (max) value of this variable, meaning that this score is calculated as

$$X_i^* = \frac{X_i}{\max}$$

This normalization can be useful when there are no negative inputs, which guarantees that the normalized variable will be between 0 and 1.

**Minimax normalization** To implement this normalization, we first need to calculate the minimum (min) and maximum (max) value for each input or output; then the minimax score is calculated using the following expression:

$$X_i^* = \frac{X_i - \min}{\max - \min}$$

The resulting score of the minimax normalization is between 0 and 1. An inconvenience of this normalization method is that inputs or outputs with long-tail distributions will be dominated by inputs or outputs with uniform distributions.

It is important to point out that the normalization process is not limited to the independent variables (inputs), but can also be used for the dependent variables (outputs) when dealing with multiple outcomes in different scales. However, the normalization process of the dependent variables is not necessary for univariate prediction models or when developing a machine to predict mixed outcomes, because the original scale of the distributions can be losses, for example, to predict two types of outcomes (binary and continuous), to predict three types of outcomes (ordinal, continuous, and count), or to predict four types of outcomes (binary, ordinal, continuous, and count data). On the other hand, when developing a machine to predict four continuous outcomes in different scales (for example, grain yield in tons by hectare, plant height in centimeters, days to maturity in 0 to 120 days, and vitamin content in milligrams), in these cases, normalizing each of the response variables is suggested to avoid the training process from being dominated by the dependent variable with large variability, which implies that the trained machine would be able to predict only this response variable with the highest accuracy. However, in some statistical machine learning models, it is not necessary to normalize the dependent variables because they allow the user to put different weights on each dependent variable to be able to train the model more fairly.

## 2.7 General Suggestions for Removing or Adding Inputs

The following is a general guide to removing inputs:

(a) Remove an independent variable (input) if it has zero variance, which implies that the input has a single unique value (Kuhn and Johnson 2013).

(b) Remove an independent variable (input) if it has near-zero variance, which implies that the input has very few values.
(c) Remove an independent variable (input) if it is highly correlated with another input variable (nearly perfect correlation), since they are measuring the same underlying information (Kuhn and Johnson 2013). Known as collinearity in statistical machine learning science, this phenomenon is important because in its presence the parameter estimates of some machine learning algorithms (for example, those based on gradient descent) are inflated (not accurately estimated).

These three issues are very common in genomic prediction, since part of the independent variables is marker information and many of them have zero or near-zero variance and other pairs have very high correlations. One of the advantages of removing input information prior to the modeling process is that this reduces the computational resources needed to implement the statistical machine learning algorithm. Also, it is possible to end up with a more parsimonious and interpretable model. Another advantage is that models with less correlated inputs are less prone to unstable parameter estimates, numerical errors, and degraded prediction performance (Kuhn and Johnson 2013).

The following are general rules for the addition of input variables:

(a) Create dummy variables from nominal or categorical inputs.
(b) Manually create a categorical variable from a continuous variable.
(c) Transform the original input variable using a specific transformation.

First, we describe the process of creating dummy variables from categorical (nominal or ordinal) inputs. Transforming categorical inputs into dummy variables is required in most supervised statistical machine learning methods, since providing the original independent variable (not transformed into dummy variables) is incorrect and should be avoided by practitioners of statistical machine learning methods. However, it is important to point out that when the dependent variable is categorical, most statistical machine learning methods do not require it to be transformed into dummy variables. For example, assume that we are studying three genotypes (G1, G2, and G3) in two environments (E1 and E2) and we collected the following grain yield data.

Using the information in Table 2.10, we created the dummy variables for each categorical variable. First, we provide the dummy variables for the environments (Table 2.11).

Next, we provide the dummy variables for the genotypes (Table 2.12).

It is important to point out that in R we can use the model.matrix() to create dummy variables from categorical independent variables. First, we create a data frame called grain.yield with the original data set:

```
grain.yield=data.frame(Environment=c("E1","E1", "E1", "E2", "E2",
"E2"), Genotype=c("G1","G2","G3", "G1","G2","G3"), y=c(5.3,
5.6,5.8,6.5, 6.8,6.9))
```

**Table 2.10** Grain yield was evaluated in two environments, and three genotypes were evaluated in each environment

| Observation | Environment | Genotype | Grain yield ( $y$) |
|---|---|---|---|
| 1 | E1 | G1 | 5.3 |
| 2 | E1 | G2 | 5.6 |
| 3 | E1 | G3 | 5.8 |
| 4 | E2 | G1 | 6.5 |
| 5 | E2 | G2 | 6.8 |
| 6 | E2 | G3 | 6.9 |

**Table 2.11** Resulting dummy variables for the environments

| Observation | Env1 | Env2 |
|---|---|---|
| 1 | 1 | 0 |
| 2 | 1 | 0 |
| 3 | 1 | 0 |
| 4 | 0 | 1 |
| 5 | 0 | 1 |
| 6 | 0 | 1 |

**Table 2.12** Resulting dummy variables for genotypes

| Observation | G1 | G2 | G3 |
|---|---|---|---|
| 1 | 1 | 0 | 0 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 |
| 4 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 |
| 6 | 0 | 0 | 1 |

We then print the data

```
> grain.yield
     Environment Genotype  y
  1  E1          G1        5.3
  2  E1          G2        5.6
  3  E1          G3        5.8
  4  E2          G1        6.5
  5  E2          G2        6.8
  6  E2          G3        6.9
```

Next, we create the dummy variables for the categorical variable environment using the model.matrix() function, as

```
ZE=model.matrix(~0+Environment, data=grain.yield)
```

The resulting matrix with the dummy variables of environments is called design matrix of environments and, in this case, it is

```
> ZE
   EnvironmentE1 EnvironmentE2
1        1             0
2        1             0
3        1             0
4        0             1
5        0             1
6        0             1
```

It is important to point out that, if instead of ~0+Environment, we use ~1-+Environment inside the model.matrix() function, we obtain a different form of the design matrix for environments:

```
> ZE
 (Intercept) EnvironmentE2
1      1           0
2      1           0
3      1           0
4      1           1
5      1           1
6      1           1
```

Strictly speaking, both design matrices contain the same information due to the fact that only $C - 1$ dummy variables are needed to capture all the information of the categorical variable, since once we have the information of $C - 1$ dummy variables, we can infer the information of the missing dummy variable. However, the decision to include all dummy variables depends on the selected statistical machine learning algorithm. Under the second version of the design matrix created with the model. matrix() function, in ZE, the column (dummy variable) corresponding to the first environment was not included, but instead an intercept (a column of ones in all rows) was added. This design matrix with an intercept is very important in some statistical machine learning algorithms such as most generalized regression models, neural networks, and deep learning models. In some cases, when this intercept is not included, numerical problems occur in the estimation of the learnable parameters (beta coefficients or weights, intercepts, etc.). The reason is that for each row (observation), these variables all add up to one, and this would provide the same information as the intercept (Kuhn and Johnson 2013). However, when statistical machine learning is not sensitive to not including the intercept (as the first design matrix), using the complete set of dummy variables can help improve model interpretation.

In the same way, the design matrix (dummy variables) for genotype is created using the following R code:

```
ZG=model.matrix(~0+Genotype, data=grain.yield)
```

which provides the following dummy variables that are accommodated in the ZG matrix:

```
> ZG
  GenotypeG1 GenotypeG2 GenotypeG3
1      1          0          0
2      0          1          0
3      0          0          1
4      1          0          0
5      0          1          0
6      0          0          1
```

This design matrix is composed of three columns because there are three categories for the categorical variable (G1, G2, and G3). Each column represents a genotype and a dummy variable was created for each genotype using an indicator variable of 0 (not present in that row) and 1 (present in that row) for each genotype. Now, using model.matrix(~1+Genotype, data=grain.yield), we obtain the design matrix with an intercept, but without the dummy variable for genotype 1:

```
> ZG
 (Intercept) GenotypeG2 GenotypeG3
1      1          0          0
2      1          1          0
3      1          0          1
4      1          0          0
5      1          1          0
6      1          0          1
```

As mentioned earlier, both design matrices for genotypes are valid since they contain the same information, given that in order to capture all the information of a categorical variable, reporting only $C - 1$ dummy variables in the design matrix is enough. However, the choice of one or another depends mostly on the statistical machine learning model to be used. It is also important to point out that the model. matrix() function, when containing the intercept, deletes the dummy variable corresponding to the first level of the categorical variable, but from the statistical point of view, any other dummy variable can be deleted without a loss of information if, and only if, $C - 1$ dummy variables are maintained.

Regarding the second point (b: manually create a categorical variable from a continuous variable), we refer to the process of manually converting a continuous variable to a categorical variable. For example, let us assume that we measured plant height in centimeters and then decided to categorize this response variable into five groups (group 1 if plant height is less than 100 cm, group 2 if plant height is between 100 and 125 cm, group 3 if it is between 125 and 150 cm, group 4 if it is between 150 and 175 cm, and group 5 when plant height is greater than 175 cm). This type of categorization is sometimes required, although we suggest avoiding categorizing continuous outcomes in this way, since a significant amount of information is lost and will affect the prediction performance of the trained model. In addition, it is

important to point out that the smaller the number of categories created, the greater the loss of information. Also, some researchers like Austin and Brunner (2004) reported that categorizing continuous inputs increases the rate of false positives (Kuhn and Johnson 2013). However, if researchers can justify that categorization is necessary, they should categorize the continuous input or output using a model framework to be able to do this with more precision.

Regarding the third approach to adding or creating inputs by transforming the original input variable using a specific transformation, we refer to the use of kernels, in which the input variables are transformed in such a way that the transformed input is used in the modeling process and the type of transformation depends on the objective of the study. This type of transformation with kernels will be discussed in detail in upcoming chapters. Also, many times transformations are applied to guarantee normality or any other distributional assumption on the variable of interest.

## 2.8 Principal Component Analysis as a Compression Method

Principal component analysis (PCA) is a method often used to compress the input data without losing as much information. The PCA works on a rectangular matrix in which the rows represent the observations ($n$) and the columns, the independent variables ($p$). The PCA creates linear combinations of the columns of matrix information, $\boldsymbol{X}$, and generates, at most, $p$ linear combinations, called principal components. These linear combinations, or principal components, can be obtained as follows:

$$\mathrm{PC}_1 = \boldsymbol{w}_1\boldsymbol{X} = w_{11}X_1 + w_{12}X_2 + \cdots + w_{1p}X_p$$
$$\cdots$$
$$\mathrm{PC}_p = \boldsymbol{w}_p\boldsymbol{X} = w_{p1}X_1 + w_{p2}X_2 + \cdots + w_{pp}X_p$$

These linear combinations are constructed in such a way that the first principal component, $\mathrm{PC}_1$, captures the largest variance, the second principal component, $\mathrm{PC}_2$, captures the second largest variance, and so on. For this reason, it is expected that few principal components ($k < p$) can explain the largest variability contained in the original rectangular matrix ($\boldsymbol{X}$), which means that with a compressed matrix, $\boldsymbol{X}^*$, we contain most of the variability of the original matrix, but with a significant reduction in the number of columns. In matrix notation, the full principal components are obtained with the following expression:

$$\mathbf{PC} = \boldsymbol{X}\boldsymbol{W},$$

where $\boldsymbol{W}$ is a $p$-by-$p$ matrix of weights whose columns are the eigenvectors of $\boldsymbol{Q} = \boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$, that is, we first need to calculate the eigenvalue decomposition of $\boldsymbol{Q}$, which is equal to $\boldsymbol{Q} = \boldsymbol{W}\boldsymbol{\Lambda}\boldsymbol{W}^{\mathrm{T}}$, where $\boldsymbol{W}$ represents the matrix of eigenvectors and $\boldsymbol{\Lambda}$ is a diagonal matrix of order $p$-by-$p$ containing the eigenvalues. For this reason, if we use $k < p$ principal components, the reduced (compressed) matrix is of order $n \times k$ and is calculated as

$$\boldsymbol{X}^{*} = \boldsymbol{X}\boldsymbol{W}^{*},$$

where $\boldsymbol{W}^{*}$ contains the same rows of $\boldsymbol{W}$, but only the first $k$ columns instead of the original $p$ columns. The selection of the number of principal components to maintain is critical, and we therefore provide some classical rules for this process:

(a) Select the required principal components to cover a certain amount of variances, such as 80% or 90%.
(b) Order the eigenvalues from highest to lowest, then make a plot of each of the ordered eigenvalues against its position and select as the number of principal components that number from which little variance is gained by retaining additional eigenvalues. This plot is called a scree plot.
(c) Discard those components associated with eigenvalues below a certain level, which is usually set as the average variance. In particular, when working with a correlation matrix built with the input matrix, $\boldsymbol{X}$, the average value of the components is 1, and this rule leads to selecting eigenvalues greater than the unit.

It is important to point out that the principal components can be obtained from a covariance matrix, $\boldsymbol{Q} = \frac{1}{n-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$, where each column of $\boldsymbol{X}$ is centered, or from the correlation matrix, $\boldsymbol{Q} = \frac{1}{n-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$, where each column of the original matrix of information was standardized. The covariance matrix is used to calculate the principal components when all the independent variables were measured using the same scale, but if they were measured in different scales, we recommend calculating the principal components with the correlation matrix, which agrees with the normalization methods that are suggested when the independent variables were measured in different scales.

Assume that we measured 15 observations (lines) and five independent variables, and the collected data is given in Table 2.13.

Then we place these data (Table 2.13) in a data frame we called Data. Since the data are in different scales, each column is standardized using the function scale in R, and the first six observations of the scaled variables are given below. The complete code that provides the output given below is available in Appendix 2.

```
> Data=read.csv("Simulated_PCA.csv", header = T)
>
> ####We scale each column of the predictors
> Rscaled=scale(Data[,-1])
> head(Rscaled)
```

**Table 2.13** Five independent variables in different scales

| Line | Yield | PlantHeight | DaysFlowering | DaysMaturity | WeightFreshPlant |
|---|---|---|---|---|---|
| L1 | 5.49 | 179.98 | 64.6 | 119.39 | 43.86 |
| L2 | 6.84 | 181.1 | 64.68 | 121.67 | 44.72 |
| L3 | 6.75 | 181 | 64.38 | 120.1 | 45.36 |
| L4 | 4.98 | 180.41 | 64.03 | 120.47 | 44.21 |
| L5 | 8.36 | 180.89 | 66.13 | 122.3 | 45.42 |
| L6 | 4.43 | 179.74 | 63.26 | 119.88 | 43.65 |
| L7 | 6.67 | 180.49 | 64.83 | 120.22 | 44 |
| L8 | 4.44 | 177.94 | 62.31 | 118.46 | 42.6 |
| L9 | 5.62 | 178.91 | 63.41 | 120.64 | 44.07 |
| L10 | 6.96 | 179.93 | 64.03 | 119.99 | 43.53 |
| L11 | 5.91 | 181.21 | 64.03 | 120.86 | 43.82 |
| L12 | 5.59 | 180.32 | 64.89 | 120.98 | 45.21 |
| L13 | 5.27 | 180.97 | 63.75 | 120.14 | 44.29 |
| L14 | 4.32 | 177.79 | 62.47 | 118.72 | 42.28 |
| L15 | 5.48 | 180.04 | 63.82 | 120.08 | 43.86 |

```
          Yield PlantHeight DaysFlowering DaysMaturity WeightFreshPlant
[1,] -0.2814557  -0.06307151   0.57645346  -0.8738660   -0.2214903
[2,]  0.9159136   0.97575340   0.65900049   1.4162657    0.7373101
[3,]  0.8360890   0.88300118   0.34944911  -0.1607110    1.4508360
[4,] -0.7337952   0.33576305  -0.01169416   0.2109332    0.1687191
[5,]  2.2640628   0.78097373   2.15516550   2.0490652    1.5177291
[6,] -1.2216124  -0.28567685  -0.80620937 -0.3816886   -0.4556160
```

We then calculate the scaled variance (correlation) $\boldsymbol{Q} = \frac{1}{n-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ and from this matrix we calculate the eigenvalue decomposition using the function eigen() of R. Next, we extract the eigenvectors and eigenvalues, and using the eigenvalue information, we calculate the variance of each principal component as shown below.

```
> n=nrow(Rscaled)
> Q_scaled=(t(Rscaled)%*%Rscaled)/(n-1) #Q_scaled=var(Rscaled)
> ####Eigenvalue decomposition
> SVD_Rscaled=eigen(Q_scaled)
>
> ####Extracting eigenvectors and eigenvalues
> EVectors=SVD_Rscaled$vectors
> Eigenvalues=SVD_Rscaled$values
> Standar.deviations.PC=sqrt(Eigenvalues)
> Standar.deviations.PC
[1] 2.0090648 0.6469991 0.4964878 0.4356803 0.3297472
```

Afterward, we manually calculate the principal components using the expression PC = $\boldsymbol{XW}$, where $\boldsymbol{W}$ is the matrix of eigenvectors extracted in the previous code and

denoted as EVectors. The calculation used here was the scaled matrix (Rscaled) instead of the original matrix of independent variables.

```
> ####Principal components for all the p=5 variables
> PCM=Rscaled%*%EVectors
> head(PCM)
            [,1]        [,2]       [,3]        [,4]        [,5]
[1,] -0.371645755  0.02096745  0.6294630  0.59140700 -0.58621466
[2,]  2.096405335 -0.02185667 -0.2605723 -0.54289807  0.12914892
[3,]  1.489539155 -0.30245825  0.5022823  0.73805410  0.79018009
[4,] -0.001842872 -0.78507631 -0.2113135 -0.06274286 -0.24361792
[5,]  3.931856138  1.09587334 -0.4321212  0.05736360 -0.17988334
[6,] -1.403180027 -0.72567496 -0.2400658 -0.12744453 -0.08832723
```

We then built the scree plot, which is one of the three tools used to select the number of principal components to maintain.

```
> ####Scree plot
> Ordered_Eigenvalues=sort(Eigenvalues,decreasing =T )
> plot(Ordered_Eigenvalues, type = "l",ylab="Variances",
xlab="Principal components")
```

Figure 2.2 shows that after two principal components, there are no significant gains in variance explained by adding more principal components. Due to this, we can select the first two principal components that explain 89.09% of the total variance of the complete data set.

Finally, the next part of the code selects only the first two principal components that will replace the whole matrix of scaled independent variables denoted here as Rscaled. Therefore, the selection process only consists of extracting the first two columns of the matrix that contain all the principal components, as shown in the following code. This reduced matrix is then replaced by the original matrix of independent variables as input in any statistical machine learning algorithm.

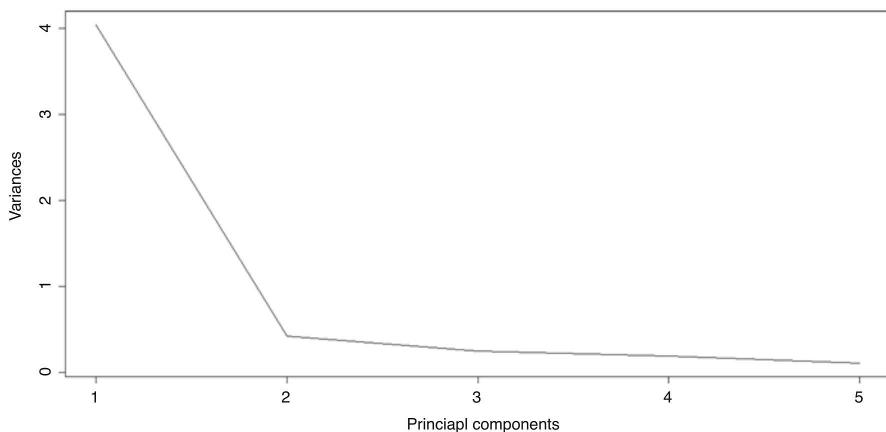



**Fig. 2.2** Scree plot for the five independent variables in different scales

```
> ####Principal components for the first k=2 principal components
> X_star=PCM[,1:2]
> head(X_star)
       [,1]          [,2]
[1,]   -0.371645755   0.02096745
[2,]    2.096405335  -0.02185667
[3,]    1.489539155  -0.30245825
[4,]   -0.001842872  -0.78507631
[5,]    3.931856138   1.09587334
[6,]   -1.403180027  -0.72567496
```

It is important to point out that the code given above was used to manually calculate the new variables, called principal components. However, there are also many libraries of R that can be used to carry out this process automatically. One of these functions is the prcomp(), which performs a principal component analysis when the only input provided is the original scaled (or not scaled) matrix of original inputs. Next, we show how to use this function to perform a principal component analysis. Then, with the function summary(), the standard deviation of each of the five principal components is extracted, as well as the proportion of variance that explains each principal component and the cumulative proportion of variance explained for the principal component. Here, we can see that the standard deviations obtained using this function are the same as those obtained above when the principal components were extracted manually using the eigen() function.

```
> ####PCA analysis using the function prcomp
> PCA=prcomp(Rscaled)
> summary(PCA)
Importance of components:
                          PC1     PC2      PC3     PC4      PC5
Standard deviation      2.0091  0.64700  0.4965  0.43568  0.32975
Proportion of Variance 0.8073   0.08372  0.0493  0.03796  0.02175
Cumulative Proportion  0.8073   0.89099  0.9403  0.97825  1.00000
```

We then show how to extract all the principal components resulting from using the prcomp() function. The principal components extracted are clearly the same, except with negative values, which is not a problem since this does not alter the solution.

```
> ###Extracting the principal components####
> PC_All=PCA$x
> head(PCA$x)
        PC1           PC2           PC3          PC4           PC5
[1,]  -0.371645755  -0.02096745   0.6294630  -0.59140700  -0.58621466
[2,]   2.096405335   0.02185667  -0.2605723   0.54289807   0.12914892
[3,]   1.489539155   0.30245825   0.5022823  -0.73805410   0.79018009
[4,]  -0.001842872   0.78507631  -0.2113135   0.06274286  -0.24361792
[5,]   3.931856138  -1.09587334  -0.4321212  -0.05736360  -0.17988334
[6,]  -1.403180027   0.72567496  -0.2400658   0.12744453  -0.08832723
```

Finally, we make the scree plot using the output of the prcomp() function, and the output of this figure is exactly the same as that given in Fig. 2.2.

```
> #####Variances of each principal component and scree plot####
> Var_PC=PCA$sdev*PCA$sdev
> plot(Var_PC, type = "l",ylab="Variances", xlab="Principal
components")
```

## Appendix 1

```
rm()
library(synbreed)
####Loading the marker information
snp7 <- read.csv("MarkersToy.csv",header=T)
snp7=snp7[,-1]
snp7

####Set names for individuals
rownames(snp7) <- paste("ID",1:8,sep="")
pos.NA=which(snp7=="?_?", arr.ind=TRUE)
snp7[pos.NA]=NA
snp7
####Creating an object of class 'gpData'
gp <- create.gpData(geno=snp7)

####Recoding to 0, 1, and 2 values the genotypic data
gp.coded <-codeGeno(gp)
Geno_Recoded=gp.coded$geno
Geno_Recoded

####Recoding to values of 0, 1, and 2 the genotypic data and inputting
Imputed_Geno<-codeGeno(gp,impute=T,impute.type="random")
Imputed_Geno$geno
```

## Appendix 2

```
rm(list=ls())
Data=read.csv("Simulated_PCA.csv", header = T)

####We scale each column of the predictors
Rscaled=scale(Data[,-1])
head(Rscaled)
n=nrow(Rscaled)
Q_scaled=(t(Rscaled)%*%Rscaled)/(n-1) #Q_scaled=var(Rscaled)
####Eigenvalue decomposition
SVD_Rscaled=eigen(Q_scaled)
```

```
####Extracting eigenvectors and eigenvalues
EVectors=SVD_Rscaled$vectors
Eigenvalues=SVD_Rscaled$values
Standar.deviations.PC=sqrt(Eigenvalues)
Standar.deviations.PC

####Principal components for all the p=5 variables
PCM=Rscaled%*%EVectors
head(PCM)

####Scree plot
Ordered_Eigenvalues=sort(Eigenvalues,decreasing =T )
plot(Ordered_Eigenvalues, type = "l",ylab="Variances",
xlab="Principal components")

####Principal components for the first k=2 principal components
X_star=PCM[,1:2]
head(X_star)

####PCA analysis using the function prcomp
PCA=prcomp(Rscaled)
summary(PCA)

###Extracting the principal components####
PC_All=PCA$x
head(PCA$x)

#####Variances of each principal component and scree plot####
Var_PC=PCA$sdev*PCA$sdev
plot(Var_PC, type = "l",ylab="Variances", xlab="Principal
components")
```

## References

Austin PC, Brunner LJ (2004) Inflation of the type I error rate when a continuous confounding variable is categorized in logistic regression analyses. Stat Med 23(7):1159–1178

Griffiths JF, Griffiths AJ, Wessler SR, Lewontin RC, Gelbart WM, Suzuki DT, Miller JH (2005) An introduction to genetic analysis. Macmillan, New York

Henderson CR (1950) Estimation of genetic parameters. Ann Math Stat 21:309–310

Henderson CR (1963) Selection index and expected genetic advance. In: Hanson WD, Robinson HF (eds) Statistical genetics and plant breeding, Publication 982. Washington, DC, National Academy of Sciences, National Research Council, pp 141–163

Henderson CR (1973) Sire evaluation and genetic trends. In: Proceedings of the animal breeding and genetics symposium in honor of J. L. Lush. Blackburgh, Champaign, IL, American Society for Animal Science, pp 10–41

Henderson CR (1975) Best linear unbiased estimation and prediction under a selection model. Biometrics 31:423–447

Henderson CR (1984) Applications of linear models in animal breeding. University of Guelph, Guelph

Kuhn M, Johnson K (2013) Applied predictive modeling. Springer, New York
Littell RC, Milliken GA, Stroup WW, Wolfinger RD (1996) SAS for mixed models. SAS Institute, Inc., Cary, NC
Littell RC, Milliken GA, Stroup WW, Schabenberger O, Wolfinger RD (2006) SAS for mixed models, 2nd edn. SAS Institute, Cary, NC
Marsjan PA, Oldenbroek JK (2007) Molecular markers, a tool for exploring genetic diversity. In: The state of the world's animal genetic resources for food and agriculture. FAO, Rome. ISBN 9789251057629
Piepho HP, Möhring J, Melchinger AE, Büchse A (2008) BLUP for phenotypic selection in plant breeding and variety testing. Euphytica 161:209–228. https://doi.org/10.1007/s10681-007-9449-8
Schork NJ, Fallin D, Lanchbury S (2000) Single nucleotide polymorphisms and the future of genetic epidemiology. Clin Genet 58:250–264
Searle SR, Casella G, McCulloch CE (2006) Variance components. Wiley, Hoboken, NJ
Stonecking M (2001) From the evolutionary past. Nature 409:821–822
Stroup WW (2012) Generalized linear mixed models: modern concepts, methods and applications. CRC Press, Boca Raton, FL
The International SNP Map Working Group (2001) A map of human genome sequence variation containing 1.42 million single nucleotide polymorphisms. Nature 409:928–933
VanRaden PM (2008) Efficient methods to compute genomic predictions. J Dairy Sci 91:4414–4423
Zeger SL, Liang KY, Albert PS (1998) Models for longitudinal data: a generalized estimating equation approach. Biometrics 44(4):1049–1060

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 3
# Elements for Building Supervised Statistical Machine Learning Models

## 3.1 Definition of a Linear Multiple Regression Model

A linear multiple regression model (LMRM) is a useful tool for investigating linear relationships between two or more explanatory variables (inputs, features in machine learning literature) ($X$) and the conditional expected value of a response $E(Y/X)$. Due to its simplicity, adequate fitting, and easily interpretable results, this has been one of the most popular techniques for studying the association between variables. Specifically, regarding the latter task, this is a useful approach and an ideal (natural) starting point for studying more advanced methods (James et al. 2013) of association and prediction.

In this chapter, we review the main concepts and approaches for fitting a linear regression model.

## 3.2 Fitting a Linear Multiple Regression Model via the Ordinary Least Square (OLS) Method

In a general context, we have a covariate vector $X = (X_1, \ldots, X_p)^{\mathrm{T}}$ and we want to use this information to predict or explain how this variable affects a real-value response $Y$. The linear multiple regression model assumes a relationship given by

$$Y = \beta_0 + \sum_{j=1}^{p} X_j \beta_j + \epsilon, \tag{3.1}$$

where $\epsilon$ is a random error with mean 0, $E(\epsilon) = 0$ and is independent of $X$. This error is included in the model to capture measurement errors and the effects of other unregistered explanatory variables that can help to explain the mean response.

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_3

Then, the conditional mean of this model is $E(Y|X) = \beta_0 + \sum_{j=1}^{p} X_j\beta_j$ and the conditional distribution of $Y$ given $X$ is only affected by the information of $X$.

For estimating the parameters $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)^{\mathrm{T}}$, usually we have a set of data $\left(\boldsymbol{x}_i^{\mathrm{T}}, y_i\right)$, $i = 1, \ldots, n$, often known as training data, where $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})^{\mathrm{T}}$ is a vector of features measurement and $y_i$ is the response measurement corresponding to the $i$th individual drawn. The most common method for estimating $\boldsymbol{\beta}$ is the least squares method (OLS) that consists of taking the $\boldsymbol{\beta}$ value that minimizes the residual sum of squares defined as

$$\mathrm{RSS}(\boldsymbol{\beta}) = \sum_{i=1}^{n} \left(y_i - \beta_0 - \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)^2 = (\boldsymbol{y} - \boldsymbol{X\beta})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X\beta}),$$

where $\boldsymbol{\beta}_0 = (\beta_1, \ldots, \beta_p)^{\mathrm{T}}$, $\boldsymbol{y} = (y_1, \ldots, y_n)^{\mathrm{T}}$ is the vector with the response values of all individuals, and $\boldsymbol{X}$ is an $n \times (p + 1)$ matrix that contains the information of the measured features of all individuals, including the intercept in the first entry:

$$\boldsymbol{X} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{bmatrix}.$$

If the $\boldsymbol{X}$ matrix has full column rank, then by differentiating the residual sum of squares with respect to the $\boldsymbol{\beta}$ coefficients, we can find the set of $\boldsymbol{\beta}$ parameters that minimize the RSS($\boldsymbol{\beta}$),

$$\frac{\mathrm{RSS}(\boldsymbol{\beta})}{\partial\boldsymbol{\beta}} = \frac{(\boldsymbol{y} - \boldsymbol{X\beta})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X\beta})}{\partial\boldsymbol{\beta}} = \frac{\boldsymbol{y}^{\mathrm{T}}\boldsymbol{y} - \mathbf{2}\boldsymbol{y}^{\mathrm{T}}\boldsymbol{X\beta} + \boldsymbol{\beta}^{\mathrm{T}}\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)\boldsymbol{\beta}}{\partial\boldsymbol{\beta}} = \mathbf{2}\left[\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)\boldsymbol{\beta} - \boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}\right]$$

This derivative is also known as the gradient of the residual sum of squares. Then by setting the gradient of the residual sum of squares to zero, we obtain the normal equations

$$\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)\boldsymbol{\beta} = \boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$$

The solution to the normal equations is unique and gives the OLS estimator of $\boldsymbol{\beta}$

$$\widehat{\boldsymbol{\beta}} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y},$$

where super index $-1$ indicates the inversion matrix.

From the above assumptions, we can show that this estimator is unbiased

$$\begin{aligned} E\left(\widehat{\boldsymbol{\beta}}\right) &= E\left[\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}\right] = E\left[\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}(\boldsymbol{X\beta} + \epsilon)\right] \\ &= E\left[\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X\beta}\right] + \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}E(\epsilon) = \boldsymbol{\beta}. \end{aligned}$$

and with the additional assumption that the observation responses $y_i'$s are uncorrelated and have the same variance, $\mathrm{Var}(y_i) = \sigma^2$, we can also show that the variance–covariance matrix of this is

$$\mathrm{Var}\left(\widehat{\boldsymbol{\beta}}\right) = \sigma^2 \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}.$$

When the input features only contain the information of a variable ($p = 1$), the resulting model is known as simple linear regression and can be easily visualized in the Cartesian plane. When $p = 2$, the above multiple linear regression describes a plane in the three-dimensional space $(x_1, x_2, y)$. In general, the conditional expected value of this model defines a hyperplane in the $p$-dimensional space of the input variables (Montgomery et al. 2012).

The fitted values corresponding to all the training individuals are

$$\widehat{\boldsymbol{y}} = \boldsymbol{X}\widehat{\boldsymbol{\beta}} = \boldsymbol{X}\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y} = \boldsymbol{H}\boldsymbol{y},$$

where the matrix $\boldsymbol{H} = \boldsymbol{X}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}$ is commonly called the hat matrix. This is because the vector of the observed response values is mapped by this expression to a vector of fitted values (Montgomery et al. 2012), in this way, puts the hat on $\boldsymbol{y}$ (Hastie et al. 2009). In a similar way, a predicted value of an arbitrary individual with feature $\boldsymbol{x}$ can be obtained by

$$\widehat{y}^* = \boldsymbol{x}^{*\mathrm{T}}\widehat{\boldsymbol{\beta}},$$

where $\boldsymbol{x}^* = (1, \boldsymbol{x}^{\mathrm{T}})^{\mathrm{T}}$.

An unbiased estimator for the common residual variance $\sigma^2$ is obtained by

$$\begin{aligned}
\widehat{\sigma}^2 &= \frac{1}{n-p-1}\sum_{i=1}^{n}(y_i - \widehat{y}_i)^2 \\
&= \frac{1}{n-p-1}\sum_{i=1}^{n} e_i^2 \\
&= \frac{1}{n-p-1}\boldsymbol{e}^{\mathrm{T}}\boldsymbol{e} \\
&= \frac{1}{n-p-1}(\boldsymbol{y} - \widehat{\boldsymbol{y}})^{\mathrm{T}}(\boldsymbol{y} - \widehat{\boldsymbol{y}}) \\
&= \frac{1}{n-p-1}\boldsymbol{y}^{\mathrm{T}}(\boldsymbol{I}_n - \boldsymbol{H})\boldsymbol{y},
\end{aligned}$$

where $e_i = y_i - \widehat{y}_i$ is known as the residual of the model corresponding to the individual $i$, $\boldsymbol{e} = \boldsymbol{y} - \widehat{\boldsymbol{y}}$ is the vector of all residual values, and $\boldsymbol{I}_n$ is the identity matrix of order $n \times n$.

The traditional inferential and prediction analysis for this model assumes that the random error $\epsilon$ is normally distributed with mean zero and variance $\sigma^2$. With this we

can show that the OLS of beta coefficients, $\widehat{\boldsymbol{\beta}}$, is a random vector distributed according to a multivariate normal distribution with vector mean $\boldsymbol{\beta}$ and a variance–covariance matrix, as previously defined (Montgomery et al. 2012; Hastie et al. 2009; Rencher and Schaalje 2008). Another important fact that will be described in more detail in the next section, is that under the Gaussian assumption over errors, the OLS of $\boldsymbol{\beta}$ coincides with the maximum likelihood estimator.

We can also show that $(n-p-1)\widehat{\sigma}^2/\sigma^2$ is independent of $\widehat{\boldsymbol{\beta}}$ and distributed according to a Chi-squared distribution with $n-p-1$ degrees of freedom. Based on this and on the properties of the normal and $t$-student distributions, we show that for each $j=0,\ldots,p$, $T_j=\left(\widehat{\beta}_j-\beta_j\right)/\sqrt{c_{j,j}\widehat{\sigma}^2}$, where $c_{jj}$ is the $(j+1, j+1)$ elements of the matrix $(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}$, are random variables with a $t$-student distribution with $n-p-1$ degrees of freedom ($t_{n-p-1}$). That is, $T_j \sim t_{n-p-1}$ and $\sim$ stands for distributed as. From here, a $100(1-\alpha)\%$ confidence interval for a particular beta coefficient, $\beta_j$, is given by

$$\widehat{\beta}_j \pm t_{1-\alpha/2,\ \ n-p-1}\sqrt{c_{jj}\widehat{\sigma}^2},$$

where $t_{\alpha,\, n-p-1}$ is the $\alpha$ quantile of the $t$-student distribution with $n-p-1$ degrees of freedom. Similarly, a $100(1-\alpha)\%$ joint confidence region for all the beta coefficients, $\boldsymbol{\beta}$, is given if these values satisfy

$$\frac{\left(\widehat{\boldsymbol{\beta}}-\boldsymbol{\beta}\right)^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\left(\widehat{\boldsymbol{\beta}}-\boldsymbol{\beta}\right)}{(p+1)\widehat{\sigma}^2} \leq F_{1-\alpha,\ \ n-p-1}^{p+1},$$

where $F_{\alpha,n-p-1}^{p+1}$ denotes the $\alpha$ quantile of the $F$ distribution with $p+1$ and $n-p-1$ degrees of freedom in the numerator and denominator, respectively (Rencher and Schaalje 2008).

In a similar way, to test a hypothesis over a specific beta coefficient, $H_{0j}=\beta_j=\beta_{j0}$, the following rule can be used: reject $H_{0j}$ if $T_{j0}=\left(\widehat{\beta}_j-\beta_{j0}\right)/\sqrt{c_{j,j}\widehat{\sigma}^2}$ is "large" in magnitude, that is, if $|T_{j0}| > t_{1-\alpha/2,\, n-p-1}$, where $\alpha$ is the desired level test. More generally, the test $H_0=\boldsymbol{W\beta}=\boldsymbol{w}$, where $\boldsymbol{W}$ is a $q \times (p+1)$ matrix of rank $q \leq p+1$, can be performed using the following rule:

$$\begin{aligned}\text{reject } H_0 \text{ if } F &= \frac{n-p-1}{q}\,\frac{(\boldsymbol{W\beta}-\boldsymbol{w})^{\mathrm{T}}\left[\boldsymbol{W}\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{W}^{\mathrm{T}}\right]^{-1}(\boldsymbol{W\beta}-\boldsymbol{w})}{\widehat{\sigma}^2}\\ &\geq F_{1-\alpha,n-p-1}^{p+1}.\end{aligned}$$

## 3.3 Fitting the Linear Multiple Regression Model via the Maximum Likelihood (ML) Method

The maximum likelihood (ML) estimation is a more general and popular method for estimating the parameters of a model (Casella and Berger 2002). It consists of finding the parameter value that maximizes the "probability" of observed values in the sample under the adopted model. Specifically, if $\left(\boldsymbol{x}_i^{\mathrm{T}}, y_i\right)$, $i = 1, \ldots, n$, is a set of observations from a multiple linear regression model (3.1) with homoscedastic and uncorrelated errors, the MLE of $\boldsymbol{\beta}$ and $\sigma^2$, $\widehat{\boldsymbol{\beta}}$ and $\widehat{\sigma}^2$, of this model is defined as

$$\left(\widehat{\boldsymbol{\beta}}^{\mathrm{T}}, \widehat{\sigma}^2\right) = \arg\max_{\boldsymbol{\beta}, \sigma^2} L\left(\boldsymbol{\beta}, \sigma^2; \boldsymbol{y}, \boldsymbol{X}\right),$$

where $L(\boldsymbol{\beta}, \sigma^2; \boldsymbol{y}, \boldsymbol{X})$ is the likelihood function of the parameters, which is the probability of the observed response values but viewed as a function of the parameters

$$L\left(\boldsymbol{\beta}, \sigma^2; \boldsymbol{y}, \boldsymbol{X}\right) = \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^n \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})\right].$$

Then, the $\log(L(\boldsymbol{\beta}, \sigma^2; \boldsymbol{y}, \boldsymbol{X}))$ is equal to

$$\log\left(L\left(\boldsymbol{\beta}, \sigma^2; \boldsymbol{y}, \boldsymbol{X}\right)\right) = -\frac{n}{2}\log(2\pi) - n\log(\sigma) - \frac{1}{2\sigma^2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})$$

To find the maximum of $\sigma^2$ and $\boldsymbol{\beta}$, we get the derivative of $\log\left(L\left(\widehat{\boldsymbol{\beta}}, \sigma^2; \boldsymbol{y}, \boldsymbol{X}\right)\right)$ with regard to these parameters

$$\frac{\log\left(L(\boldsymbol{\beta}, \sigma^2; \boldsymbol{y}, \boldsymbol{X})\right)}{\partial\boldsymbol{\beta}} = \frac{\left[\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)\boldsymbol{\beta} - \boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}\right]}{\sigma^2}$$

$$\frac{\log\left(L(\boldsymbol{\beta}, \sigma^2; \boldsymbol{y}, \boldsymbol{X})\right)}{\partial\sigma^2} = -\frac{n}{2\sigma^2} + \frac{1}{2\sigma^4}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})$$

Now, by setting these derivatives equal to zero and solving the resulting equations for $\boldsymbol{\beta}$ and $\sigma^2$, we found that the estimates of these parameters are

$$\widehat{\boldsymbol{\beta}} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}$$

$$\widehat{\sigma}^2 = \frac{1}{n}\left(\boldsymbol{y} - \boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)^{\mathrm{T}}\left(\boldsymbol{y} - \boldsymbol{X}\widehat{\boldsymbol{\beta}}\right).$$

From this we can see that for each value of $\sigma^2$, the value of $\boldsymbol{\beta}$ that maximizes the likelihood is the same value that maximizes $-\frac{1}{2\sigma^2}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta})$, which in turn

minimizes $(\boldsymbol{y} - \boldsymbol{X\beta})^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X\beta})$, which is precisely the OLS of $\boldsymbol{\beta}, \widehat{\boldsymbol{\beta}}$. But when equating the derivative of $\log\left(L\left(\widehat{\boldsymbol{\beta}}, \sigma^2; \boldsymbol{y}, \boldsymbol{X}\right)\right)$ to zero and solving for $\sigma^2$, the value of $\sigma^2$ that maximizes $L\left(\widehat{\boldsymbol{\beta}}, \sigma^2; \boldsymbol{y}, \boldsymbol{X}\right)$ is $\widehat{\sigma}^2 = \frac{1}{n}\left(\boldsymbol{y} - \boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)^{\mathrm{T}}\left(\boldsymbol{y} - \boldsymbol{X}\widehat{\boldsymbol{\beta}}\right)$.

Finally,

$$L\left(\boldsymbol{\beta}, \sigma^2; \boldsymbol{y}, \boldsymbol{X}\right) \leq L\left(\widehat{\boldsymbol{\beta}}, \sigma^2; \boldsymbol{y}, \boldsymbol{X}\right) \leq L\left(\widehat{\boldsymbol{\beta}}, \widehat{\sigma}^2; \boldsymbol{y}, \boldsymbol{X}\right)$$

and from here, the MLE of $\boldsymbol{\beta}$ and $\sigma^2$ are $\widehat{\boldsymbol{\beta}}$ and $\widehat{\sigma}^2$, because it can be shown that the values of parameters that maximize the likelihood are unique when the design matrix $\boldsymbol{X}$ is of full column rank.

## 3.4 Fitting the Linear Multiple Regression Model via the Gradient Descent (GD) Method

The steepest descent method, also known as the gradient descent (GD) method, is a first-order iterative algorithm for minimizing a function ($f$). It is a central mechanism in statistical learning to training models (to estimate the parameters), for example, in neuronal networks and penalized regression models (Ridge and Lasso). It consists of successively updating the argument of the objective function in the direction of the steepest descent (along the negative of the gradient of the function), that is, in the direction in which $f$ decreases most rapidly (Haykin 2009; Nocedal and Wright 2006). Specifically, each step of this algorithm is described by

$$\eta_{t+1} = \eta_t - \alpha \nabla f(\eta_t),$$

where $\nabla f(\eta_t)$ is the gradient vector of $f$ evaluated in the current value $\eta_t$ and $\alpha$ is a step size or learning rate parameter, which greatly determines the convergence behavior toward an optimal solution (Haykin 2009; Beysolow II 2017) and in neural networks it is popular for setting this at a small, fixed value (Warner and Misra 1996; Goodfellow et al. 2016). The learning rate parameter can be adaptative as well, that is, can be allowed to change at each step. For example, in the library Keras (see Chap. 11) that can be used for implementing and training neuronal networks models, there are several optimizers based on an adaptive gradient descendent algorithm such as Adam Adgrad, Adadelta, RMSprop, among others (Allaire and Chollet 2019). The ideal value of the step size would be the value that gives the larger reduction in each step, that is, the value of $\alpha$ that minimizes $f(\eta_t - \alpha \nabla f(\eta_t))$, which in general is difficult and expensive to obtain (Nocedal and Wright 2006).

Although the use of this algorithm could be avoided in an MLR, especially in small data sets, and also because of its slow convergence in linear systems (Burden and Faires 2011), here we will describe how this works when finding the optimal

beta coefficients in this model. First, the gradient of the residual sum of squares is given by

$$\nabla \mathrm{RSS}(\boldsymbol{\beta}) = 2\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}\right).$$

Then, the next update of beta coefficients in the gradient descent algorithm in this model is given by

$$\begin{aligned} \boldsymbol{\beta}_{t+1} &= \boldsymbol{\beta}_t - 2\alpha\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\beta}_t - \boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}\right) \\ &= \boldsymbol{\beta}_t - 2\alpha\boldsymbol{X}^{\mathrm{T}}(\boldsymbol{X}\boldsymbol{\beta}_t - \boldsymbol{y}) \\ &= \boldsymbol{\beta}_t + 2\alpha\boldsymbol{X}^{\mathrm{T}}\boldsymbol{e}_t, \end{aligned}$$

where $\boldsymbol{e}_t = \boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta}_t$ is the vector of residuals that is obtained in the current iteration. One way to speed up the convergence of the algorithm is by choosing the ideal learning rate in each step, which, as was described before, is given by the value of $\alpha$ that minimizes $f(\eta_t - \alpha \nabla f(\eta_t))$, and in this case for the MLR model is given by (Nocedal and Wright 2006):

$$\alpha_t = \frac{\boldsymbol{e}_t^{\mathrm{T}}\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{e}_t}{\boldsymbol{e}_t^{\mathrm{T}}\left(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\right)^2\boldsymbol{e}_t}.$$

**Example 1** For numerical illustration, we considered a synthetic data set that consists of 100 observations and two covariates. The scatter plots in Fig. 3.1 show how the response variable ($y$) is related to the two covariates ($x_1$, $x_2$). By setting a value of $10^{-2}$ for the learning rate parameter, and as the stopping criterion a tolerance of $10^{-8}$ for the maximum norm of the difference between the current and next vector value, the beta coefficient obtained with the GD method is $\widehat{\boldsymbol{\beta}} = (5.0460764, 0.8551383, 2.1903356)$. For these synthetic examples, 12 iterations were necessary, while by changing the learning rate parameter to $10^{-3}$, the number of iterations increased to 185, but we practically got the same results. Now, by using the "optimal" learning rate parameter described before for MLR with the same tolerance error ($10^{-8}$), the number of required iterations up to convergence is reduced to only 10 iterations. In general, the performance of the gradient descent depends greatly on the objective function and can be affected by the characteristics of the model, the dispersion of the data (explained variance of the predictors), and the dependence between the predictors, among others.

In the data set used in this example, the covariates are independent and the proportion of explained variances by the predictor is about 79% of the total variance of the response. By changing to a pair of moderately correlated covariates with correlation 0.75, while holding the same beta coefficient values, the variance of the residual (1.44), and the same sample size, we generate data where a greater proportion of variance is explained by the covariates (85.6%), but when applying the

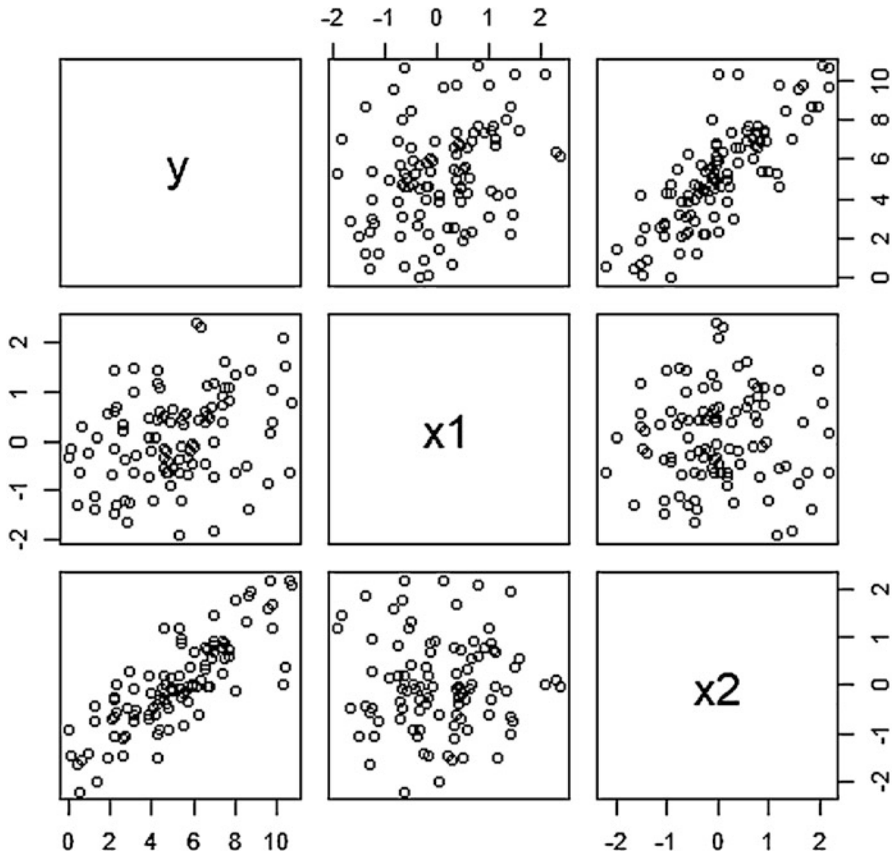



**Fig. 3.1** Scatter plot of synthetic data generated from an MLR with two covariates

gradient descent with the same tolerance error ($10^{-8}$) as before, and learning rate values of $10^{-2}$ and $10^{-3}$, the required number of iterations are about 5.75 times (69) and 3.5 times (649) the number required for the independent covariates case and the example described before, respectively.

Continuing with the last case of dependent variables, when using the optimal learning rate described before for the MLR, the number of iterations is reduced to 60, 9 less than when using the constant learning rate $10^{-2}$.

By multiplying the beta coefficients used before by sqrt(0.1), the proportion of explained variance by the covariates is reduced to 27.30% and 37.2% in the same independent covariate (E3) and the same correlated covariate (E4) scenarios described before, respectively. With a tolerance error of $10^{-8}$ and with a learning rate equal to $10^{-2}$, the required number of iterations are 183 and 66, for scenarios E3 and E4, respectively, while for a learning rate of $10^{-3}$ the required number of iterations are 617 and 1638. When using the "optimal" learning rate parameter, the required number of iterations is reduced to 17 and 56 for scenarios E3 and E4, respectively.

The R code used for implementing the GD method is given next.

```
#################R code for Example 1 ################################
rm(list=ls())
library(mvtnorm)
set.seed(1)
X = cbind(1,rmvnorm(100,c(0,0),diag(2)))
#Uncomment the next three lines code to simulate dependent covariables
#Sigma_X =0.75+0.25*diag(2)
#L = t(chol(Sigma_X))
#X = X%*%t(L)

betav = c(5,1,2.1)
#Uncomment the next line code to reduce the value of the beta cofficients
and reduce the proportion of variance of the response explained by the
features
betav = sqrt(0.1)*beta
y = X%*%betav + rnorm(100,0,1.2)
dat = data.frame(y=y,x1 = X[,2],x2=X[,3])
plot(dat)

alpha = 1e-2
#alpha = 1e-3
tol   = 1e-8
p = 2
betav_0 = c(mean(y),rep(0,p))
tol.e = 1
Iter = 0
tX = t(X)
XtX = X%*%t(X)
while(tol<tol.e)
{
 Iter = Iter + 1
 e = y-X%*%betav_0
 #Uncomment the next line code to use the optimal learning rate
 #alpha = (t(e)%*%XtX%*%e/(t(e)%*%(XtX)%*%XtX%*%e))[1,1]
 betav_t = betav_0 + alpha*tX%*%e
 tol.e = max(abs(betav_t-betav_0))
 betav_0 = betav_t
}
betav_t
tol.e
Iter
```

This code is only for illustrative purposes, that is, to illustrate in a very transparent way how the GD method can be implemented. Of course the existing statistical machine learning software programs implement this method and so there is no need to use this program for real applications, since the existing software programs that implement this method do a lot of work more efficiently and in a more user-friendly way.

## 3.5 Advantages and Disadvantages of Standard Linear Regression Models (OLS and MLR)

The MLR is a simple and computationally appealing class of models, but with many predictors (relative to the sample size) or nearly dependent features, it may result in large prediction error and/or large predictive intervals (Wakefield 2013). To appreciate the latter case (nearly dependent features), consider the spectral decomposition $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} = \boldsymbol{\Gamma\Lambda\Gamma}^{\mathrm{T}}$, where $\boldsymbol{\Lambda} = \mathrm{Diag}(\lambda_0, \ldots, \lambda_p)$ is a diagonal matrix with the eigenvalues of $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$ in decreasing order and $\boldsymbol{\Gamma}$ is an orthogonal matrix with columns corresponding to eigenvectors of $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$. Then the obtained variance–covariance matrix of the OLS estimator of $\widehat{\boldsymbol{\beta}}$ can be expressed as

$$\mathrm{Var}\left(\widehat{\boldsymbol{\beta}}\right) = \sigma^2\left(\boldsymbol{\Gamma\Lambda\Gamma}^{\mathrm{T}}\right)^{-1} = \sigma^2\boldsymbol{\Gamma\Lambda}^{-1}\boldsymbol{\Gamma}^{\mathrm{T}}.$$

When the features are nearly dependent, some $\lambda_j$'s will be "close" to zero and consequently the variance of some $\widehat{\beta}_j'$s will be high; this is even greater when the linear dependence of the features is strong (Wakefield 2013; Christensen 2011). This strong dependence between features is a problem of the OLS in MLR that is also reflected in the quality of the prediction performance, for example, when this is measured by the conditional expected prediction error (EPE) or mean squared error prediction that for an individual with feature $\boldsymbol{x}_o$ is given by

$$\begin{aligned}
\mathrm{EPE}(\boldsymbol{x}_o) &= E_{\boldsymbol{Y},Y_o|\boldsymbol{X},\boldsymbol{x}_o}\left[\left(Y_o - \boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}\right)^2\right] \\
&= E_{\boldsymbol{Y},Y_o|\boldsymbol{X},\boldsymbol{x}_o}\left\{\left[\left(Y_o - E(Y_0|\boldsymbol{x}_0) + E(Y_0|\boldsymbol{x}_0) - \boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}\right)\right]^2\right\} \\
&= E_{\boldsymbol{Y},Y_o|\boldsymbol{X},\boldsymbol{x}_o}\left[\left(Y_o - E\left(Y_0|\boldsymbol{x}_0^{*\mathrm{T}}\right)\right)^2\right] + 2E_{Y_o|\boldsymbol{x}_o}\left[\left(Y_o - E\left(Y_0|\boldsymbol{x}_0^{*\mathrm{T}}\right)\right)\right]\left[E\left(Y_0|\boldsymbol{x}_0^{*\mathrm{T}}\right) - E_{\boldsymbol{Y}|\boldsymbol{X}}\left(\boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}\right)\right] \\
&\quad + E_{\boldsymbol{Y},Y_o|\boldsymbol{X}}\left[\left(E\left(Y_0|\boldsymbol{x}_0^{*\mathrm{T}}\right) - \boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}\right)^2\right] \\
&= \sigma^2 + E_{\boldsymbol{Y},Y_o|\boldsymbol{X},\boldsymbol{x}_o}\left[\left(\boldsymbol{x}_o^{*\mathrm{T}}\boldsymbol{\beta} - \boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}\right)^2\right] = \sigma^2 + Var\left(\boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}|\boldsymbol{x}_o\right) \\
&= \sigma^2 + \sigma^2\boldsymbol{x}_o^{*\mathrm{T}}\boldsymbol{\Gamma\Lambda}^{-1}\boldsymbol{\Gamma}^{\mathrm{T}}\boldsymbol{x}_o^{*} \\
&= \sigma^2\left(1 + \sum_{j=0}^{p}\frac{\left(x_{oj}^{**}\right)^2}{\lambda_j}\right),
\end{aligned}$$

where $\boldsymbol{x}_o^{**} = \boldsymbol{\Gamma}^{\mathrm{T}}\boldsymbol{x}_o^{*} = \left(x_{o0}^{**}, \ldots, x_{op}^{**}\right)^{\mathrm{T}}$. This means that the average loss incurred (squared difference between the value to be predicted and the predicted value) by predicting $Y_0$ with its estimated mean under the MLR, $\boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}$, is composed of intrinsic or irreducible data noise (first term) and the variance of $\boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}$ (second term). The former cannot be avoided no matter how well the mean value of $Y_0 \mid x_0$, $E(Y_0| x_0)$, is

estimated, and the latter increases as the dependence of features is stronger. From this, it is apparent that the EPE is also affected by the strong dependence between features, which is a problem of the OLS in an MLR in a prediction context.

## 3.6 Regularized Linear Multiple Regression Model

### *3.6.1 Ridge Regression*

Ridge regression, originally proposed as a method to combat multicollinearity, is also a common approach for controlling overfitting in an MLR model (Christensen 2011). It translates the OLS problem into the minimization of the penalized residual sum of squares defined as

$$\mathrm{PRSS}_\lambda(\boldsymbol{\beta}) = \sum_{i=1}^{n}\left(y_i - \beta_0 - \sum_{j=1}^{p} x_{ij}\beta_j\right)^2 + \lambda\sum_{j=1}^{p}\beta_j^2,$$

where $\lambda \geq 0$ is known as the regularization or tuning parameter, which determines the level or degree to which the beta coefficients are shrunk toward zero. When $\lambda = 0$, the OLS is the solution to the beta coefficients, but when $\lambda$ is large, the $\mathrm{PRSS}_\lambda(\boldsymbol{\beta})$ is dominated by the penalization term, and the OLS solution has to shrink toward 0 (Christensen 2011). In general, when the number of parameters to be estimated is larger than the number of observations, the estimator can be highly variable. In this situation, the intuition of Ridge regression tries to alleviate this by constraining the sum of squares for the beta coefficients.

Note that $\mathrm{PRSS}_\lambda(\boldsymbol{\beta})$ can be expressed as

$$\mathrm{PRSS}_\lambda(\boldsymbol{\beta}) = \mathrm{RSS}(\boldsymbol{\beta}) + \lambda\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{\beta},$$

where $\boldsymbol{D} =$ diag $(0, 1, \ldots, 1)$ is an identity matrix of dimension $(p + 1) \times (p + 1)$ but with one zero in its first entry. Then, the gradient of $\mathrm{RSS}_\lambda(\boldsymbol{\beta})$, that is, the first derivative with regard to $\boldsymbol{\beta}$ of $\mathrm{RSS}_\lambda(\boldsymbol{\beta})$, is

$$\nabla\mathrm{PRSS}_\lambda(\boldsymbol{\beta}) = 2\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}\right) + 2\lambda\boldsymbol{D}\boldsymbol{\beta}.$$

Solving $\nabla\mathrm{PRSS}_\lambda(\boldsymbol{\beta}) = \mathbf{0}$, the Ridge solution is given by

$$\widehat{\boldsymbol{\beta}}^R(\lambda) = \underset{\boldsymbol{\beta}}{\operatorname{argmin}}\mathrm{PRSS}_\lambda(\boldsymbol{\beta}) = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} + \lambda\boldsymbol{D}\right)^{-1}\mathbf{X}^{\mathrm{T}}\boldsymbol{y}.$$

This is a biased estimator of $\boldsymbol{\beta}$ because the conditional expected value is given by

$$E\left[\widehat{\boldsymbol{\beta}}^{R}(\lambda)\right] = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} + \lambda\boldsymbol{D}\right)^{-1}\mathbf{X}^{\mathrm{T}}\boldsymbol{X}\boldsymbol{\beta}$$

but as will be described later, relative to the OLS estimator, by introducing a "small" bias, the variance or/and the EPE of this method could potentially be reduced (Wakefield 2013).

By using the method of Lagrange multipliers, the Ridge regression estimates of the $\boldsymbol{\beta}$ coefficients can be reformulated in a similar way to the OLS problem, but subject to the condition that the magnitude of the $\boldsymbol{\beta}_0 = (\beta_1, \ldots, \beta_p)^{\mathrm{T}}$ be less or equal to $t(\lambda)^{\frac{1}{2}}$, that is,

$$\widehat{\boldsymbol{\beta}}^{R}(\lambda) = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \mathrm{RSS}(\boldsymbol{\beta})$$
$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 \leq t(\lambda),$$

where $t(\lambda)$ is a one-to-one function that produces an equivalent definition to the penalized OLS presentation of the Ridge regression described before (Wakefield 2013; Hastie et al. 2009, 2015). This constrained reformulation gives a more transparent role than the one played by the tuning parameter, and among other things, suggests a convenient and common way of redefining the Ridge estimator by standardizing the variables when these are of very different scales.

A graphic representation of this constraint problem for $\beta_0 = 0$ and $p = 2$ is given in Fig. 3.2, where the nested ellipsoids correspond to contour plots of RSS($\boldsymbol{\beta}$) and the green region is the restriction with $t(\lambda) = 3^2$, which contains the Ridge solution.

The MLR defined in (3.1) but now defined with the standardized variables is expressed as

$$\begin{aligned} y &= 1_n\mu + \boldsymbol{X}_{1s}\boldsymbol{\beta}_{0s} + \epsilon \\ &= \boldsymbol{X}_s\boldsymbol{\beta}_s + \epsilon, \end{aligned}$$

where $\mathbf{1}_n$ is the column vector with 1's in all its entries, $\boldsymbol{X}_{1s} = \begin{bmatrix} x_{11s} & \cdots & x_{1ps} \\ \vdots & \vdots & \vdots \\ x_{n1s} & \cdots & x_{nps} \end{bmatrix}$, $x_{ijs} = \left(x_{ij} - \overline{x}_j\right)/s_j$, $s_j = \sqrt{\sum_{i=1}^{n}\left(x_{ij} - \overline{x}_j\right)^2/n}$, $j = 1, \ldots, p$; $\boldsymbol{X}_s = [\mathbf{1}_n\ \boldsymbol{X}_{1s}]$; $\boldsymbol{\beta}_s = \left(\mu,\ \boldsymbol{\beta}_{0s}^{\mathrm{T}}\right)^{\mathrm{T}}$; and $\boldsymbol{\beta}_{0s} = (\beta_{1s}, \ldots, \beta_{ps})^{\mathrm{T}}$.

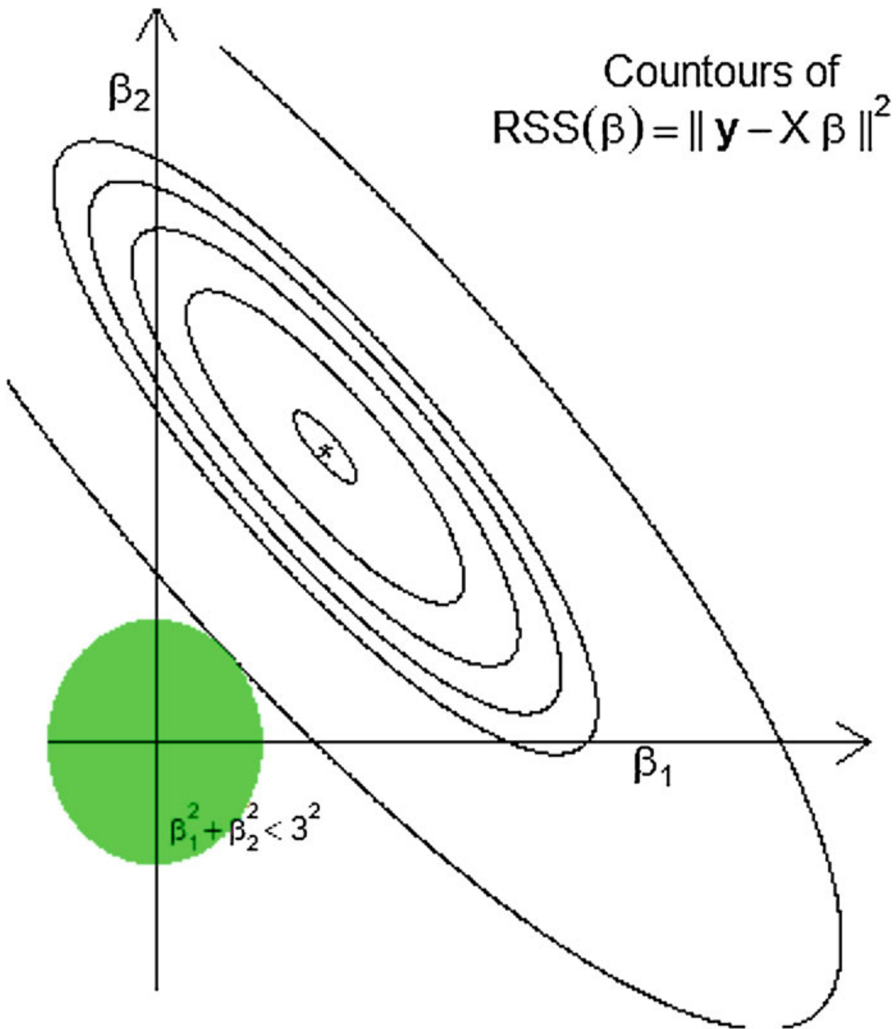



**Fig. 3.2** Graphic representation of the Ridge solution of the OLS with restriction $\sum_{j=1}^{p}\beta_j^2 < 3^2$. The green region contains the Ridge solution for $t(\lambda) = 3^2$

Then, the redefined penalized residual sum squared under this model is

$$\begin{aligned} \mathrm{PRSS}_\lambda(\boldsymbol{\beta}_s) &= \sum_{i=1}^{n}\left(y_i - \mu - \sum_{j=1}^{p} x_{ijs}\beta_{js}\right)^2 + \lambda\sum_{j=1}^{p}\beta_{js}^2 \\ &= \left(\boldsymbol{y} - \boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{\beta}_s\right)^{\mathrm{T}}\left(\boldsymbol{y} - \boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{\beta}_s\right) + \lambda\boldsymbol{\beta}_{0s}{}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{\beta}_{0s}. \end{aligned}$$

The Ridge solution under this redefinition is like the one given before, but now

$$\begin{aligned}
\widehat{\boldsymbol{\beta}}_s^R(\lambda) &= \left(\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{X}_s + \lambda\boldsymbol{D}\right)^{-1}\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{y} \\
&= \left(\begin{bmatrix} 1_n^{\mathrm{T}} \\ \boldsymbol{X}_{1s}^{\mathrm{T}} \end{bmatrix}[1_n\ \boldsymbol{X}_{1s}] + \lambda\boldsymbol{D}\right)^{-1}\begin{bmatrix} 1_n^{\mathrm{T}} \\ \boldsymbol{X}_{1s}^{\mathrm{T}} \end{bmatrix}\boldsymbol{y} \\
&= \begin{bmatrix} n & 0_n^{\mathrm{T}} \\ 0_n & \boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s} + \lambda\boldsymbol{I_p} \end{bmatrix}^{-1}\begin{bmatrix} 1_n^{\mathrm{T}}y \\ \boldsymbol{X}_{1s}^{\mathrm{T}}y \end{bmatrix} \\
&= \begin{bmatrix} \bar{y}_n \\ \widehat{\boldsymbol{\beta}}_{0s}(\lambda) \end{bmatrix},
\end{aligned}$$

where $\bar{y}_n = \sum_{i=1}^{n} y_i/n$ is the sample mean of the responses and $\widehat{\boldsymbol{\beta}}_{0s}(\lambda) = \left(\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s} + \lambda\boldsymbol{I_p}\right)^{-1}\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{y}$ is the Ridge estimator of $\boldsymbol{\beta}_{0s}$. The mean value of this Ridge solution is

$$\mathrm{E}\left[\widehat{\boldsymbol{\beta}}_s^R(\lambda)\right] = \begin{bmatrix} \mu \\ E\left[\widehat{\boldsymbol{\beta}}_{0s}(\lambda)\right] \end{bmatrix},$$

where $E\left[\widehat{\boldsymbol{\beta}}_{0s}(\lambda)\right] = \left(\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s} + \lambda\boldsymbol{I_p}\right)^{-1}\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s}\boldsymbol{\beta}_{0s}$ is the expected value of the Ridge estimator of $\boldsymbol{\beta}_{0s}$. The variance–covariance matrix is

$$\begin{aligned}
\mathrm{Var}\left(\widehat{\boldsymbol{\beta}}_s^R(\lambda)\right) &= \left(\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{X}_s + \lambda\boldsymbol{D}\right)^{-1}\boldsymbol{X}_s^{\mathrm{T}}\mathrm{Var}(\boldsymbol{y})\boldsymbol{X}_s\left(\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{X}_s + \lambda\boldsymbol{D}\right)^{-1} \\
&= \sigma^2\begin{bmatrix} n & 0_n^{\mathrm{T}} \\ 0_n & \boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s} + \lambda\boldsymbol{I_p} \end{bmatrix}^{-1}\begin{bmatrix} 1_n^{\mathrm{T}} \\ \boldsymbol{X}_{1s}^{\mathrm{T}} \end{bmatrix}[1_n\ \ \boldsymbol{X}_{1s}]\begin{bmatrix} n & 0_n^{\mathrm{T}} \\ 0_n & \boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s} + \lambda\boldsymbol{I_p} \end{bmatrix}^{-1} \\
&= \sigma^2\begin{bmatrix} 1/n & 0_n^{\mathrm{T}} \\ 0_n & \mathrm{Var}\left(\widehat{\boldsymbol{\beta}}_{0s}(\lambda)\right) \end{bmatrix},
\end{aligned}$$

where $\mathrm{Var}\left(\widehat{\boldsymbol{\beta}}_{0s}(\lambda)\right) = \left(\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s} + \lambda\boldsymbol{I}_p\right)^{-1}\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s}\left(\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s} + \lambda\boldsymbol{I}_p\right)^{-1}$. So, because in this standardized way, the Ridge solution of the intercept ($\mu$) is the sample mean of the observed responses, and the correlation of this with the rest of the estimated parameters ($\widehat{\boldsymbol{\beta}}_{0s}(\lambda)$) is null, in the literature it is common to handle this parameter separately from all other coefficients ($\boldsymbol{\beta}_{0s}$) (Christensen 2011).

Note that

$$E\left[\widehat{\boldsymbol{\beta}}_{0s}(\lambda)\right] = \boldsymbol{\Gamma}_s\left(\boldsymbol{\Lambda}_s + \lambda\boldsymbol{I_p}\right)^{-1}\boldsymbol{\Lambda}_s\boldsymbol{\beta}_{0s}^*$$

and

$$\mathrm{Var}\left(\widehat{\boldsymbol{\beta}}_{0s}(\lambda)\right) = \boldsymbol{\Gamma}_s\left(\boldsymbol{\Lambda}_p + \lambda \boldsymbol{I}_{\boldsymbol{p}}\right)^{-1}\boldsymbol{\Lambda}_s\left(\boldsymbol{\Lambda}_p + \lambda \boldsymbol{I}_{\boldsymbol{p}}\right)^{-1}\boldsymbol{\Gamma}_s^{\mathrm{T}},$$

where $\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s} = \boldsymbol{\Gamma}_s\boldsymbol{\Lambda}_s\boldsymbol{\Gamma}_s^{\mathrm{T}}$ is the spectral decomposition of $\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s}$ and $\boldsymbol{\beta}_{0s}^{*} = \boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\beta}_{0s}$. So the conditional expected prediction error at $\boldsymbol{x}_o$ when using the Ridge solution is

$$\begin{aligned}
\mathrm{EPE}_\lambda(\boldsymbol{x}_o) &= E_{\boldsymbol{Y},Y_o|\boldsymbol{X},\boldsymbol{x}_o}\left[\left(Y_o - \boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}_s^R(\lambda)\right)^2\right] \\
&= E_{\boldsymbol{Y},Y_o|\boldsymbol{X},\boldsymbol{x}_o}\left[\left(Y_o - E(Y_0|\boldsymbol{x}_0) + E(Y_0|\boldsymbol{x}_0) - \boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}_s^R(\lambda)\right)^2\right] \\
&= \sigma^2 + E_{\boldsymbol{Y},Y_o|\boldsymbol{X},\boldsymbol{x}_o}\left[\left(\boldsymbol{x}_o^{*\mathrm{T}}\boldsymbol{\beta}_s - \boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}_s^R(\lambda)\right)^2\right] \\
&= \sigma^2 + \left[\left(\boldsymbol{x}_o^{*\mathrm{T}}\boldsymbol{\beta}_s - \boldsymbol{x}_o^{*\mathrm{T}}E_{\boldsymbol{Y}|\boldsymbol{X}}\left(\widehat{\boldsymbol{\beta}}_s^R(\lambda)\right)\right)^2\right] + \mathrm{Var}\left(\boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}_s^R(\lambda)|\boldsymbol{x}_o\right) \\
&= \sigma^2 + \left[\left(\mu + \boldsymbol{x}_o^{\mathrm{T}}\boldsymbol{\beta}_{0s} - \mu - \boldsymbol{x}_o^{\mathrm{T}}\boldsymbol{\Gamma}_s\left(\boldsymbol{\Lambda}_s + \lambda\boldsymbol{I}_{\boldsymbol{p}}\right)^{-1}\boldsymbol{\Lambda}_s\boldsymbol{\beta}_{0s}^{*}\right)^2\right] + \sigma^2\left[\frac{1}{n} + \boldsymbol{x}_o^{\mathrm{T}}\boldsymbol{\Gamma}_s\left(\boldsymbol{\Lambda}_p + \lambda\boldsymbol{I}_{\boldsymbol{p}}\right)^{-1}\boldsymbol{\Lambda}_s\left(\boldsymbol{\Lambda}_p + \lambda\boldsymbol{I}_{\boldsymbol{p}}\right)^{-1}\boldsymbol{\Gamma}_s^{T}\boldsymbol{x}_o\right] \\
&= \sigma^2 + \left[\left(\boldsymbol{x}_o^{**\mathrm{T}}\boldsymbol{\beta}_{0s}^{*} - \boldsymbol{x}_o^{**\mathrm{T}}\left(\boldsymbol{\Lambda}_s + \lambda\boldsymbol{I}_{\boldsymbol{p}}\right)^{-1}\boldsymbol{\Lambda}_s\boldsymbol{\beta}_{0s}^{*}\right)^2\right] + \sigma^2\left[\frac{1}{n} + \boldsymbol{x}_o^{**\mathrm{T}}\left(\boldsymbol{\Lambda}_p + \lambda\boldsymbol{I}_{\boldsymbol{p}}\right)^{-1}\boldsymbol{\Lambda}_s\left(\boldsymbol{\Lambda}_p + \lambda\boldsymbol{I}_{\boldsymbol{p}}\right)^{-1}\boldsymbol{x}_o^{**}\right] \\
&= \sigma^2 + \left[\sum_{j=1}^{p}\left(1 - \frac{\lambda_j}{\lambda_j + \lambda}\right)x_{oj}^{**}\beta_{js}^{*}\right]^2 + \sigma^2\left(\frac{1}{n} + \sum_{j=1}^{p}\frac{\lambda_j}{(\lambda_j + \lambda)^2}\left(x_{oj}^{**}\right)^2\right),
\end{aligned}$$

where $\boldsymbol{x}_o^{**} = \boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{x}_o^{*} = \left(x_{o0}^{**}, \ldots, x_{op}^{**}\right)^{\mathrm{T}}$ and $\boldsymbol{\beta}_{0s}^{*} = \boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\beta}_{0s} = \left(\beta_{1s}^{*}, \ldots, \beta_{ps}^{*}\right)^{\mathrm{T}}$. The second and third terms of the last equality correspond to the squared bias and the variance of $\boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}_s^R(\lambda)$ as an estimator of $\boldsymbol{x}_o^{*\mathrm{T}}\boldsymbol{\beta}_s$, respectively. By setting $\lambda = 0$, this EPE corresponds to the EPE of the OLS prediction but with standardized variables, while by letting $\lambda$ be very large, the variance will decrease and the squared bias will increase.

More importantly, because the derivative of $\mathrm{EPE}_\lambda(\boldsymbol{x}_o)$ with respect to $\lambda$, $\frac{d}{d\lambda}\mathrm{PE}_\lambda(\boldsymbol{x}_o)$, is a right continuous function at $\lambda = 0$, and for $\boldsymbol{X}_{1s}$ of full column rank, $\lim_{\lambda\to 0^+}\frac{d}{d\lambda}\mathrm{PE}_\lambda(\boldsymbol{x}_o) = -2\sigma^2\sum_{j=1}^{p}\frac{\left(x_{oj}^{**}\right)^2}{\lambda_j^2} = c$; then for $\epsilon = -\frac{c}{2} > 0$, we have that $\lambda^* > 0$ such that $\left|\frac{d}{d\lambda}\mathrm{PE}_\lambda(\boldsymbol{x}_o) - c\right| < \epsilon$ for $\lambda < \lambda^*$. From this we have that $\frac{d}{d\lambda}\mathrm{PE}_\lambda(\boldsymbol{x}_o) < -\frac{c}{2} + c = \frac{c}{2} < 0$ for all $\lambda < \lambda^*$, for some $\lambda^* > 0$. Then, at least in the interval $[0, \lambda^*]$, the expected prediction error at $\boldsymbol{x}_o$ shows a decreasing behavior, which indicates that there is a value of $\lambda$ such that with the Ridge regression estimation of beta coefficients, we can get a smaller prediction error than with the OLS prediction. Figure 3.3 shows a graphic representation of this behavior of Ridge prediction, where the lower EPE is reached at about $\lambda = \exp(2.22)$. Figure 3.3 also shows the increasing and decreasing behavior of the bias-squared and the variance involved.

When $\boldsymbol{X}_{1s}$ is not full column rank, the previous argument regarding the behavior of the EPE of the Ridge solution is already not valid directly, but it could be used for

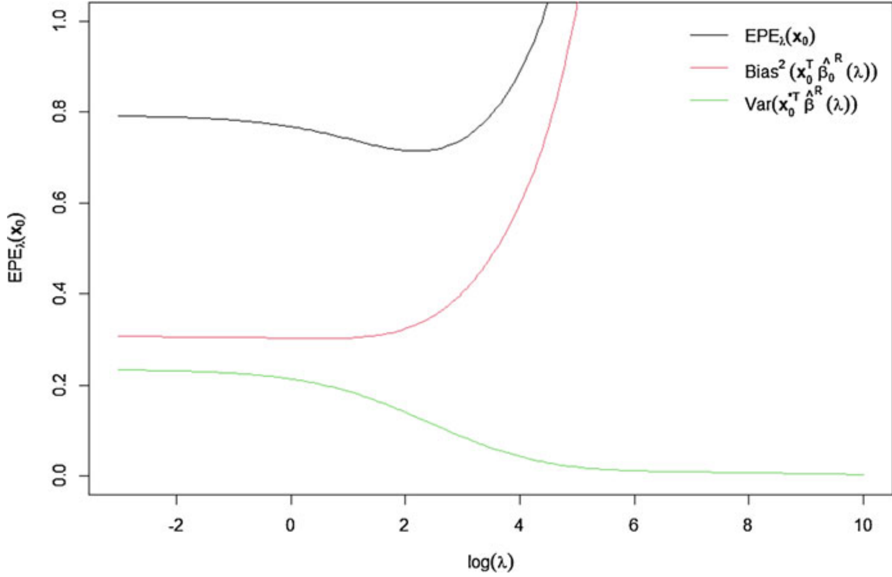



**Fig. 3.3** Behavior of the expected prediction error at $\boldsymbol{x}_o$ of the Ridge solution

validating part of the more general case. To see this, first note that the spectral decomposition of $\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s}$ can be reduced to

$$\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s} = [\boldsymbol{\Gamma}_{1s}\ \boldsymbol{\Gamma}_{2s}]\begin{bmatrix} \boldsymbol{\Lambda}_{1s} & \mathbf{0} \\ \mathbf{0}^{\mathrm{T}} & \boldsymbol{\Lambda}_{2s} \end{bmatrix}\begin{bmatrix} \boldsymbol{\Gamma}_{1s}^{\mathrm{T}} \\ \boldsymbol{\Gamma}_{2s}^{\mathrm{T}} \end{bmatrix} = \boldsymbol{\Gamma}_{1s}^{\mathrm{T}}\boldsymbol{\Lambda}_{1s}\boldsymbol{\Gamma}_{1s},$$

where $\boldsymbol{\Lambda}_{1s} = \mathrm{Diag}\left(\lambda_1, \ldots, \lambda_{p^*}\right)$, $p^* =$ rank $(\mathbf{X}_{1c})$ is the rank of design matrix and $\boldsymbol{\Lambda}_{2s}$ is the null matrix of order $(p - p^*) \times (p - p^*)$. Furthermore, because $\boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\Gamma}_s = \boldsymbol{I}_p$ implies that $\boldsymbol{\Gamma}_{2s}^{\mathrm{T}}\boldsymbol{X}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1s}\boldsymbol{\Gamma}_{2s} = \mathbf{0}$, which in turn implies that $\boldsymbol{X}_{1s}\boldsymbol{\Gamma}_{2s} = \mathbf{0}$, then the MLR can be conveniently expressed by

$$\begin{aligned}
\boldsymbol{y} &= 1_n\mu + \boldsymbol{X}_{1s}\boldsymbol{\beta}_{0s} + \epsilon \\
&= 1_n\mu + \boldsymbol{X}_{1s}\boldsymbol{\Gamma}_s\boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\beta}_{0s} + \epsilon \\
&= 1_n\mu + \boldsymbol{X}_{1s}[\boldsymbol{\Gamma}_{1s}\ \boldsymbol{\Gamma}_{2s}]\boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\beta}_{0s} + \epsilon \\
&= 1_n\mu + [\boldsymbol{X}_{1s}\boldsymbol{\Gamma}_{1s}\ \boldsymbol{X}_{1s}\boldsymbol{\Gamma}_{2s}]\boldsymbol{\beta}_{0s}^* + \epsilon \\
&= 1_n\mu + \boldsymbol{X}_{1s}^*\boldsymbol{\beta}_{01s}^* + \epsilon,
\end{aligned}$$

where $\boldsymbol{\beta}_{0s}^* = \boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\beta}_{0s}$ , $\boldsymbol{X}_{1s}^* = \boldsymbol{X}_{1s}\boldsymbol{\Gamma}_{1s}$, and $\boldsymbol{\beta}_{0s}^* = \boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\beta}_{0s} = \left[\boldsymbol{\beta}_{0s}^T\boldsymbol{\Gamma}_{1s}, \boldsymbol{\beta}_{0s}^{\mathrm{T}}\ \boldsymbol{\Gamma}_{2s}\ \right]^{\mathrm{T}} = \left[\boldsymbol{\beta}_{01s}^{*}{}^{\mathrm{T}}\ \boldsymbol{\beta}_{02s}^{*\mathrm{T}}\right]^{\mathrm{T}}$.

Also, from similar arguments, note that the penalized residual sum of squares of the Ridge solution can be expressed by

$$\begin{aligned}
&(\boldsymbol{y} - 1_n\mu - \boldsymbol{X}_{1s}\boldsymbol{\beta}_{0s})^{\mathrm{T}}(\boldsymbol{y} - 1_n\mu - \boldsymbol{X}_{1s}\boldsymbol{\beta}_{0s}) + \lambda\boldsymbol{\beta}_{0s}^{\mathrm{T}}\boldsymbol{\beta}_{0s} \\
&= \left(\boldsymbol{y} - 1_n\mu - \boldsymbol{X}_{1s}^{*}\boldsymbol{\beta}_{01s}^{*}\right)^{\mathrm{T}}\left(\boldsymbol{y} - 1_n\mu - \boldsymbol{X}_{1s}^{*}\boldsymbol{\beta}_{01s}^{*}\right) + \lambda\boldsymbol{\beta}_{0s}^{\mathrm{T}}\left(\boldsymbol{\Gamma}_{1s}^{\mathrm{T}}\boldsymbol{\Gamma}_{1s} + \boldsymbol{\Gamma}_{2s}^{\mathrm{T}}\boldsymbol{\Gamma}_{2s}\right)\boldsymbol{\beta}_{0s} \\
&= \left(\boldsymbol{y} - 1_n\mu - \boldsymbol{X}_{1s}^{*}\boldsymbol{\beta}_{01s}^{*}\right)^{\mathrm{T}}\left(\boldsymbol{y} - 1_n\mu - \boldsymbol{X}_{1s}^{*}\boldsymbol{\beta}_{01s}^{*}\right) + \lambda\boldsymbol{\beta}_{01s}^{*\mathrm{T}}\boldsymbol{\beta}_{01s}^{*} + \lambda\boldsymbol{\beta}_{02s}^{*\mathrm{T}}\boldsymbol{\beta}_{02s}^{*}
\end{aligned}$$

This function of $\boldsymbol{\beta}_{0s}^{*}$ is minimized at $\widetilde{\boldsymbol{\beta}}_{0s}^{*}(\lambda) = \left(\widetilde{\boldsymbol{\beta}}_{01s}^{*\mathrm{T}}(\lambda), \mathbf{0}_{p-p^{*}}^{\mathrm{T}}\right)^{\mathrm{T}}$, where $\widetilde{\boldsymbol{\beta}}_{01s}^{*\mathrm{T}}(\lambda) = \left(\boldsymbol{\Lambda}_{1s} + \lambda\boldsymbol{I}_{p^{*}}\right)^{-1}\boldsymbol{\Gamma}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1S}^{\mathrm{T}}\boldsymbol{y}$ is the Ridge solution of the MLR expressed in terms of $\boldsymbol{X}_{1s}^{*}\boldsymbol{\beta}_{0s}^{*}$. Furthermore, because $\boldsymbol{\beta}_{0s}^{*} = \boldsymbol{\Gamma}_{s}^{\mathrm{T}}\boldsymbol{\beta}_{0s}$ is a non-singular transformation, the original Ridge solution of $\boldsymbol{\beta}_{s}$ can be expressed in terms of $\widetilde{\boldsymbol{\beta}}_{0s}^{*}(\lambda)$ as $\widehat{\boldsymbol{\beta}}_{s}(\lambda) = \left(\bar{y}_n, \widehat{\boldsymbol{\beta}}_{0s}^{*\mathrm{T}}(\lambda)\right)^{\mathrm{T}}$, where $\widehat{\boldsymbol{\beta}}_{0s}(\lambda) = \boldsymbol{\Gamma}_{s}\widehat{\boldsymbol{\beta}}_{0s}^{*\mathrm{T}}(\lambda) = \boldsymbol{\Gamma}_{1s}\widetilde{\boldsymbol{\beta}}_{01s}^{*}(\lambda)$. Then, in a similar fashion as before, the conditional expected prediction error at $\boldsymbol{x}_o$ by using the Ridge solution in this case can be computed as

$$\begin{aligned}
&\mathrm{EPE}_{\lambda}(\boldsymbol{x}_o) \\
&= \sigma^2 + E_{\boldsymbol{Y},Y_o|\boldsymbol{X},\boldsymbol{x}_o}\left[\left(\boldsymbol{x}_o^{*\mathrm{T}}\boldsymbol{\beta}_s - \boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}_s^{R}(\lambda)\right)^2\right] \\
&= \sigma^2 + \left[\left(\boldsymbol{x}_o^{*\mathrm{T}}\boldsymbol{\beta}_s - \boldsymbol{x}_o^{*\mathrm{T}}E_{\boldsymbol{Y}|\boldsymbol{X}}\left(\widehat{\boldsymbol{\beta}}_s^{R}(\lambda)\right)\right)^2\right] + \mathrm{Var}\left(\boldsymbol{x}_o^{*\mathrm{T}}\widehat{\boldsymbol{\beta}}_s^{R}(\lambda)|\boldsymbol{x}_o\right) \\
&= \sigma^2 + \left[\left(\mu + \boldsymbol{x}_o^{\mathrm{T}}\boldsymbol{\Gamma}_s\boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\beta}_{0s} - \mu - \boldsymbol{x}_o^{\mathrm{T}}\boldsymbol{\Gamma}_{1s}E_{\boldsymbol{Y}|\boldsymbol{X}}\left(\widehat{\boldsymbol{\beta}}_{0s}^{*\mathrm{T}}(\lambda)\right)\right)^2\right] + \left[\frac{\sigma^2}{n} + \mathrm{Var}\left(\boldsymbol{x}_o^{\mathrm{T}}\boldsymbol{\Gamma}_s\widehat{\boldsymbol{\beta}}_{0s}^{*\mathrm{T}}(\lambda)|\boldsymbol{x}_o\right)\right] \\
&= \sigma^2 + \left[\left(\mu + \boldsymbol{x}_o^{\mathrm{T}}\boldsymbol{\Gamma}_s\boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\beta}_{0s} - \mu - \boldsymbol{x}_o^{\mathrm{T}}\boldsymbol{\Gamma}_{1s}\left(\boldsymbol{\Lambda}_{1s} + \lambda\boldsymbol{I}_{p^{*}}\right)^{-1}\boldsymbol{\Gamma}_{1s}^{\mathrm{T}}\boldsymbol{X}_{1c}^{\mathrm{T}}\boldsymbol{X}_{1c}\boldsymbol{\beta}_{0S}\right)^2\right] \\
&\quad + \sigma^2\left[\frac{1}{n} + \boldsymbol{x}_o^{\mathrm{T}}\boldsymbol{\Gamma}_{1s}\left(\boldsymbol{\Lambda}_{1s} + \lambda\boldsymbol{I}_{p^{*}}\right)^{-1}\boldsymbol{\Lambda}_{1s}\left(\boldsymbol{\Lambda}_{1s} + \lambda\boldsymbol{I}_{p^{*}}\right)^{-1}\boldsymbol{\Gamma}_{1s}^{T}\boldsymbol{x}_o\right] \\
&= \begin{cases}
\sigma^2 + \left[\left(\sum_{j=1}^{p^{*}}\left(1 - \frac{\lambda_j}{\lambda_j + \lambda}\right)x_{oj}^{*}\beta_{oj}^{*}\right)^2\right] + \sigma^2\left[\frac{1}{n} + \sum_{j=1}^{p^{*}}\frac{\lambda_j}{\left(\lambda_j + \lambda\right)^2}\left(x_{oj}^{*}\right)^2\right] & \text{if } x_o = \boldsymbol{\Gamma}_{1s}\boldsymbol{a}_1 \\
\sigma^2 + \left[\boldsymbol{x}_o^{T}\boldsymbol{\beta}_{0s}\right]^2 + \frac{\sigma^2}{n} & \text{if } x_o = \boldsymbol{\Gamma}_{2s}\boldsymbol{a}_2,
\end{cases}
\end{aligned}$$

where $\boldsymbol{x}_o^{*} = \boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{x}_o = \left[x_{o1}^{*}, \ldots, x_{op}^{*}\right]^{\mathrm{T}}$, $\boldsymbol{\beta}_o^{*} = \boldsymbol{\Gamma}_s^{\mathrm{T}}\boldsymbol{\beta}_{os} = \left[\beta_{o1}^{*}, \ldots, \beta_{op}^{*}\right]^{\mathrm{T}}$, and $\boldsymbol{a}_1 \in \mathbb{R}^{p^{*}}$ and $\boldsymbol{a}_2 \in \mathbb{R}^{p-p^{*}}$. So, using a similar argument as before, in the first case ($x_o = \boldsymbol{\Gamma}_{1s}\boldsymbol{a}_1$), the value of $\lambda > 0$ is such that the expected prediction error at $\boldsymbol{x}_o$ is better than that obtained with the OLS approach, $\widehat{\boldsymbol{\beta}}_s(0) = \lim_{\lambda\to 0}\widehat{\boldsymbol{\beta}}_s(\lambda) = \left[\bar{y}_n, \left(\boldsymbol{\Gamma}_{1s}\Lambda_{1s}^{-1}\boldsymbol{\Gamma}_{1s}^{\mathrm{T}}X_{1c}^{\mathrm{T}}\boldsymbol{y}\right)^{\mathrm{T}}\right]^{\mathrm{T}}$. In the second case, $x_o = \boldsymbol{\Gamma}_{2s}\boldsymbol{a}_2$, the EPE($\boldsymbol{x}_o$) in both approaches is the same and doesn't depend on $\lambda$, so in such cases, no improved gain with regard to the Ridge solution is achieved. A third case was included, that is, when the target feature is of the form $\boldsymbol{x}_o = \boldsymbol{\Gamma}_{1s}\boldsymbol{a}_1 + \boldsymbol{\Gamma}_{2s}\boldsymbol{a}_2$. In this case, under the described argument, the advantage of Ridge regression over the OLS approach in a prediction context is not clear.

However, in practice, we don't know the true value of the parameters, and we need to evaluate the test error in all possible values of the training sample, which we also don't have. So a common way to choose the $\lambda$ value is by cross-validation. For more details about validation strategies, see Chap. 4. For example, with a $k$-fold CV, the complete data set is divided into $K$ balanced disjoint subsets, $S_k$, $k = 1, \ldots, K$. One subset is used as validation and the rest are used to fit the model in each value of a chosen grid of values of $\lambda$. This procedure is repeated $K$ times, where each time a subset in the partition is taken as the validation set. A more detailed $k$-fold CV procedure is described below:

1. First, choose a grid of values of $\lambda$, $\lambda = (\lambda_1, \ldots, \lambda_L)$ .
2. Remove the subset $S_k$ and for each value $\lambda_l$ in the grid, fit the model with the remaining $K - 1$ elements of the partition denoted by $\widehat{\boldsymbol{\beta}}^R_{-k}(\lambda_l)$, the corresponding Ridge estimation of $\boldsymbol{\beta}$, and compute the average prediction error across all observations in the validation set $S_k$ as

$$\widehat{\mathrm{APE}}_{-k}(\lambda_l) = \frac{1}{|S_k|} \sum_{y_i \in S_k} \left( y_i - \boldsymbol{x}_i^{*\mathrm{T}} \widehat{\boldsymbol{\beta}}^R_{-k}(\lambda_l) \right)^2,$$

   where $|S_k|$ denotes the total observations in partition $k$.

3. Choose as the best value of $\lambda$ in the grid $\left( \widetilde{\lambda}^* \right)$, the one with the lower average prediction error across all partitions, that is

$$\widetilde{\lambda}^* = \arg\min_{\lambda_l} \widehat{\mathrm{APE}}(\lambda_l),$$

   where $\widehat{\mathrm{APE}}(\lambda_l) = \frac{1}{K} \sum_{k=1}^{K} \widehat{\mathrm{APE}}_{-k}(\lambda_l)$.

4. Once $\widetilde{\lambda}^*$ is chosen, we fit the model with the complete data set and the prediction of new individuals with feature $\boldsymbol{x}_o$ can be made with $\widehat{y}_i = \boldsymbol{x}_o^{*T} \widehat{\boldsymbol{\beta}}^R \left( \widetilde{\lambda}^* \right)$, where $\widehat{\boldsymbol{\beta}}^R \left( \widetilde{\lambda}^* \right)$ is the Ridge estimation of $\boldsymbol{\beta}$ at $\lambda = \widetilde{\lambda}^*$.

It is important to point out that very often the performance of the model needs to be evaluated for comparison purposes with other competing models. A common way to do this is to split the data set several times into two subsets, one for training the model ($D_{\mathrm{tr}}$) (to fit the model) and the other for testing ($D_{\mathrm{tst}}$) it, in which the predictive ability of a model is tested. In each splitting, only the training data set ($D_{\mathrm{tr}}$) is used to train the model (by steps 1–3 before fitting the whole training data set), and the prediction evaluation of the fitted model is made with the testing data set, as explained before in point 4. The prediction evaluation of the testing data set is done by an empirical "estimate" of the EPE, $\mathrm{MSE} = \frac{1}{|D_{\mathrm{tst}}|} \sum_{i \in D_{\mathrm{tst}}} \left( y_i - \boldsymbol{x}_o^{*\mathrm{T}} \widehat{\boldsymbol{\beta}}^R \left( \widetilde{\lambda}^* \right), \right)^2$, and

finally, an average evaluation of the performance of the model is obtained across all chosen splittings. See Chap. 4 for more explicit details.

The Ridge solution can also be obtained from a Bayesian formulation. To do this, consider the MLR model described before with standardized features and the vector of residuals distributed as $N_n(\mathbf{0}, \sigma^2\boldsymbol{I}_n)$. With this assumption, the vector of responses $\boldsymbol{y}$ is distributed in a multivariate normal distribution with vector mean $\mathbf{1}_n\mu + \boldsymbol{X}_{1s}\boldsymbol{\beta}_{0s}$ and variance–covariance matrix $\sigma^2\boldsymbol{I}_n$. Then, to complete the Bayesian formulation, assume $\boldsymbol{\beta}_{os} \sim N_p\left(\mathbf{0}, \sigma_\beta^2\boldsymbol{I}_p\right)$ as the prior distribution of the beta coefficients in $\boldsymbol{\beta}_{0s}$ and a "flat" prior for the intercept $\mu$, where $\sigma^2$ and $\sigma_\beta^2$ are known. Under this Bayesian specification, the posterior distribution of $\boldsymbol{\beta}_s$ is

$$\begin{aligned} & f(\boldsymbol{\beta}_s|\boldsymbol{y}, \boldsymbol{X}_{1s}) \\ & \propto \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X}_s\boldsymbol{\beta}_s)^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{X}_s\boldsymbol{\beta}_s)\right]\exp\left(-\frac{1}{2\sigma_\beta^2}\boldsymbol{\beta}_{0s}^{\mathrm{T}}\boldsymbol{\beta}_{0s}\right) \\ & \propto \exp\left\{-\frac{1}{2}\left[\boldsymbol{\beta}_s^{\mathrm{T}}\left(\sigma_\beta^{-2}\boldsymbol{D}+\sigma^{-2}\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{X}_s\right)\boldsymbol{\beta}_s - 2\sigma^{-2}\boldsymbol{y}^{\mathrm{T}}\boldsymbol{X}_s\boldsymbol{\beta}_s\right]\right\} \\ & \propto \exp\left\{-\frac{1}{2}\left(\boldsymbol{\beta}_s-\widetilde{\boldsymbol{\beta}}_s\right)^{\mathrm{T}}\widetilde{\boldsymbol{\Sigma}}_\beta^{-1}\left(\boldsymbol{\beta}_s-\widetilde{\boldsymbol{\beta}}_s\right)\right\}, \end{aligned}$$

where $\widetilde{\boldsymbol{\Sigma}}_\beta = \left(\sigma_\beta^{-2}\boldsymbol{D}+\sigma^{-2}\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{X}_s\right)^{-1} = \sigma^2\left(\sigma^2/\sigma_\beta^2\boldsymbol{D}+\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{X}_s\right)^{-1}$, $\widetilde{\boldsymbol{\beta}}_s = \sigma^{-2}\widetilde{\boldsymbol{\Sigma}}_\beta\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{y}$, and $\boldsymbol{D}$ is the diagonal penalty matrix. That is, the posterior distribution of $\boldsymbol{\beta}_s$ is a multivariate normal distribution with vector mean $\widetilde{\boldsymbol{\beta}}_s = \sigma^{-2}\widetilde{\boldsymbol{\Sigma}}_\beta\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{y} = \left(\sigma^2/\sigma_\beta^2\boldsymbol{D}+\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{X}_s\right)^{-1}\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{y}$ and variance–covariance matrix $\widetilde{\boldsymbol{\Sigma}}_\beta = \sigma^2\left(\sigma^2/\sigma_\beta^2\boldsymbol{D}+\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{X}_s\right)^{-1}$. Then, by taking $\lambda = \sigma^2/\sigma_\beta^2$, we have that the mean/mode of the posterior distribution of $\boldsymbol{\beta}_s$ coincides with the Ridge estimation described before, $\widetilde{\boldsymbol{\beta}}^R(\lambda)$.

**Example 2** We considered a genomic example to illustrate the Ridge regression approach and the CV process to choose the learning parameter $\lambda$ (WheatMadaToy, PH the response). This data set consists of 50 observations corresponding to 50 lines and a relationship genomic matrix computed from marker information. Table 3.1 shows the prediction behavior of the Ridge and the OLS approaches in terms of the MSEP, across five different splittings obtained by partitioning the complete data set into five subsets: the data of a subset are used as a testing set and the rest to train the model. For training the model, a five-fold cross-validation (5FCV) was used along the lines following steps 1–3 described before, and the prediction performance was done following step 4.

**Table 3.1** Prediction behavior of the Ridge and OLS regression models across different partitions of the complete data set: one subset of the partition (20%) is used for evaluating the performance of the model and the rest (80%) for training the model. RR denotes Ridge regression method

| Partition | MSE RR | MSE OLS |
|---|---|---|
| 1 | 325.40 | 379.33 |
| 2 | 433.19 | 454.37 |
| 3 | 803.76 | 1341.35 |
| 4 | 319.53 | 312.81 |
| 5 | 403.62 | 555.10 |
| Average | 457.10 | 608.59 |

Table 3.1 indicates that in four out of five partitions, the Ridge regression shows less MSE than the corresponding OLS approach. In all these cases, the MSE of the OLS was, on average, 31.46% greater than that of the Ridge regression approach, and in general, on average, by 31.14% (MSE = 421.8834 for Ridge and MSE = 655.8596 for OLS). From this, we have that the Ridge regression approach shows a better prediction performance than the OLS. The large variation of the MSE between folds observed in this example could indicate that for obtaining a more precise comparison between models, a larger number of partitions need to be used. Often, the use of more partitions is avoided when larger data sets are used in applications.

The R code used for obtaining this result is the following:

```
######################R code for Example 2 #########################
rm(list=ls())
library(BMTME)
data("WheatMadaToy")
dat_F = phenoMada
dim(dat_F)
dat_F$GID = as.character(dat_F$GID)
G = genoMada
eig_G = eigen(G)
G_0.5 = eig_G$vectors%*%diag(sqrt(eig_G$values))%*%t(eig_G$vectors)
X = G_0.5
y = dat_F$PH
n = length(y)
source('TR_RR.R')
#5FCV
set.seed(3)
K = 5
Tab = data.frame()
Grpv = findInterval(cut(sample(1:n,n),breaks=K),1:n)
for(i in 1:K)
{
 Pos_tr = which(Grpv!=i)
 y_tr = y[Pos_tr]
 X_tr = X[Pos_tr,]
 TR_RR = Tr_RR_f(y_tr,X_tr,K=5,KG=100,KR=1)
 lambv = TR_RR$lambv
```

```
 #Tst
 y_tst = y[-Pos_tr]; X_tst = X[-Pos_tr,]
 #RR
 Pred_RR = Pred_RR_f(y_tst,X_tst,TR_RR)
 #OLS
 Pred_ols = Pred_ols_f(y_tst,X_tst,y_tr,X_tr)
 Tab = rbind(Tab,data.frame(Sim=i,MSEP_RR = Pred_RR$MSEP,
              MSEP_ols = Pred_ols$MSEP))
 cat('i = ', i,'\n')
}
Tab
```

Tr_RR_f, Pred_RR_f, and Pred_ols_f are R functions accessed by the command source('TR_RR.R'), where TR_RR.R is the file R script defined in Appendix 1.

From the last code, three things are important to point out:

1. The Grpv contains the information of the K=5 folds for the outer CV implemented and each time the model is trained with $K - 1$ and tested with the remaining fold.
2. The function that trains the model under Ridge regression is called Tr_RR_f, while the function that obtains the predictions of the testing set of this trained model is Pred_RR_f; both functions are fully described in Appendix 1.
3. The predictions under the OLS method are obtained with the function Pred_ols_f, which is also fully detailed in Appendix 1. It is important to point out that the function Tr_RR_f internally implements an inner $k$-fold CV to tune the hyperparameter $\lambda$ required in Ridge regression.

**Example 3 (Simulation)** To get a better idea about the behavior of the Ridge solution, here we report the results of a small simulation study in a scenario where the number of observations ($n$=100) is less than the number of features ($p = 500$) and these are moderately correlated. Specifically, we generated 100 data sets, each of size 100, from the following model:

$$y_i = 5 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta}_0 + \epsilon_i,$$

where the vector of beta coefficients ($\boldsymbol{\beta}_0$) was set to the values shown in Fig. 3.4, and the features of all the individuals in each data set were generated from a multivariate normal distribution centered on the null vector and variance–covariance matrix $\boldsymbol{\Sigma} = 0.25\boldsymbol{I}_p + 0.75\boldsymbol{J}_p$, where $\boldsymbol{I}_p$ and $\boldsymbol{J}_p$ are the identity matrix and matrix of ones of dimension $p \times p$. The random errors ($\epsilon_i$) were simulated from a normal distribution with mean 0 and variance 0.025.

The behavior of the Ridge and OLS solutions across the 100 simulated data sets is shown in Fig. 3.5. The MSE of Ridge regression is located on the $x$-axis and the corresponding MSE of the OLS is located on the $y$-axis. On average, the OLS resulted in an MSE equal to 808.81, which is 30.59% larger than the average MSE

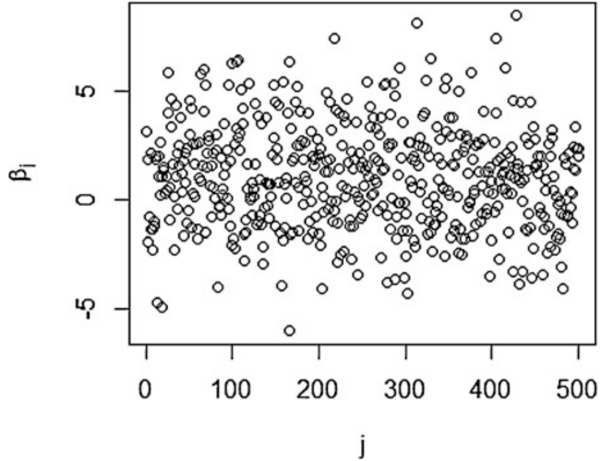



**Fig. 3.4** Beta coefficients used in simulation: $\beta_j, j = 1, \ldots, p$

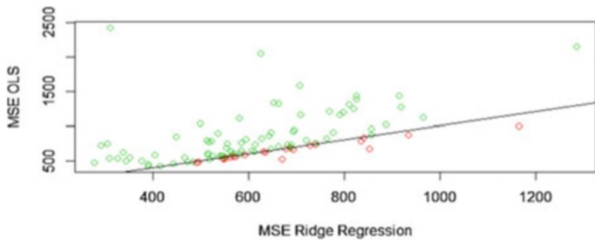



**Fig. 3.5** MSE of Ridge regression (MSE RR) versus MSE OLS regression (MSE OLS)

(619.32) of the Ridge approach. In terms of the percentage of simulations in favor of each method, Ridge regression was better in 78 out of 100 simulations, while the OLS was better only in 22 out of 100 simulations. In general, from this small simulation study we obtained more evidence in favor of the Ridge regression method.

The R code used for obtaining this result is the following:

```
###########################R code for Example 3######################
rm(list=ls(all=TRUE))
library(mvtnorm)
library(MASS)
source('TR_RR.R')
set.seed(10)
n = 100
p = 500
Var = 0.25*diag(p)+0.75
Tab = data.frame()
betav = rnorm(p,rpois(p,1),2)
plot(betav,xlab=expression(j),ylab=expression(beta[j]))
for(i in 1:100)
{
 X = rmvnorm(n,rep(0,p),Var)
```

```
 dim(X)
 y = 5+X%*%betav + rnorm(n,0,0.5)
 Pos_tr = sample(1:n,n*0.80)
 y_tr = y[Pos_tr]; X_tr = X[Pos_tr,]
 y_tst = y[-Pos_tr]; X_tst = X[-Pos_tr,]
 #Training RR
 TR_RR = Tr_RR_f(y_tr,X_tr,K=5,KG=100,KR=1)
 TR_RR$lamb_o
 lambv = TR_RR$lambv
 plot(log(TR_RR$lambv),TR_RR$iPEv_mean)
 #Prediction RR in testing data
 Pred_RR = Pred_RR_f(y_tst,X_tst,TR_RR)
 Pred_RR$MSEP

 #OLS
 Pred_ols = Pred_ols_f(y_tst,X_tst,y_tr,X_tr)
 Pred_ols

 Tab = rbind(Tab,data.frame(Sim=i,MSEP_RR = Pred_RR$MSEP,
              MSEP_ols = Pred_ols$MSEP))
 cat('i = ', i,'\n')}

Mean_v = colMeans(Tab)
(Mean_v[3]-Mean_v[2])/Mean_v[2]*100

mean(Tab$MSEP_RR<Tab$MSEP_ols)
Pos = which(Tab$MSEP_RR<Tab$MSEP_ols)
mean((Tab$MSEP_ols[Pos]-Tab$MSEP_RR[Pos])/Tab$MSEP_RR[Pos])*100
mean((Tab$MSEP_RR[-Pos]-Tab$MSEP_ols[-Pos])/Tab$MSEP_ols[-Pos])
*100

plot(Tab$MSEP_RR, Tab$MSEP_ols,
  col=ifelse(Tab$MSEP_RR<Tab$MSEP_ols,3,2),
  xlab='MSEP RR', ylab='MSEP OLS')
abline(a=0,b=1)
```

The TR_RR.R script file is the same as the one defined in Example 2 in Appendix 1.

### 3.6.2 *Lasso Regression*

Like Ridge regression, the Lasso regression solves the OLS problem but penalizes the residual sum squared in a slightly different way. With the standardized variables, the Lasso estimator of $\boldsymbol{\beta}_s$ is defined as

$$\widetilde{\boldsymbol{\beta}}_s^L(\lambda) = \arg\min_{\mu, \boldsymbol{\beta}_{0s}} \mathrm{PRSS}_\lambda(\boldsymbol{\beta}_s),$$

where now $\mathrm{PRSS}_\lambda(\boldsymbol{\beta}_s) = \sum_{i=1}^{n}\left(y_i - \mu - \sum_{j=1}^{p} x_{ijs}\beta_{js}\right)^2 + \lambda\sum_{j=1}^{p}\left|\beta_{js}\right|$ is the $\mathrm{RSS}(\boldsymbol{\beta})$ but penalized by the sum of the absolute regression coefficients. For $\lambda = 0$, the solution is the OLS, while when $\lambda$ is large, the OLS solutions are shrunken toward 0 (Tibshirani 1996).

Note that for any given values of $\boldsymbol{\beta}_{0s}$, the value of $\mu$ that minimizes $\mathrm{PRSS}_\lambda(\boldsymbol{\beta}_s)$ is the sample mean of the responses, $\widetilde{\mu} = \frac{1}{n}\sum_{i=1}^{n} y_i$, the same as the Ridge estimator. However, the rest of the Lasso estimator of $\boldsymbol{\beta}_s$, $\boldsymbol{\beta}_{0s}$, cannot be obtained analytically, so numerical methods are often used.

Although there are efficient algorithms for computing the entire regularization path for the Lasso regression coefficients (Efron et al. 2004; Friedman et al. 2008), here we will describe the coordinate-wise descent given in Friedman et al. (2007). The idea of this method is to successively optimize the $\mathrm{PRSS}_\lambda(\boldsymbol{\beta}_s)$ one parameter at a time (beta coefficient). Holding $\beta_{ks}$, $j \neq k$, fixed at their current values $\widetilde{\beta}_{js}(\lambda)$, the value of $\beta_k$ that minimizes $\mathrm{PRSS}_\lambda(\boldsymbol{\beta}_s)$ is given by

$$\begin{aligned}\widetilde{\beta}^{*}_{ks}(\lambda) &= S\left(\sum_{i=1}^{n} x_{ijs}\left(y_i - \widetilde{y}_i^{(k)}\right), \lambda\right) \\ &= S\left(n\widetilde{\beta}_{ks}(\lambda) + \sum_{i=1}^{n} x_{ijs}(y_i - \widetilde{y}_i), \lambda\right),\end{aligned}$$

where $\widetilde{y}_i^{(k)} = \overline{y} + \sum_{\substack{j=1 \\ j\neq k}}^{p} x_{ijs}\widetilde{\beta}_{js}(\lambda)$ and $S(\beta, \lambda) = \begin{cases} \beta - \lambda & \text{if } \beta > 0 \text{ and } \lambda < |\beta| \\ \beta + \lambda & \text{if } \beta < 0 \text{ and } \lambda < |\beta| \\ 0 & \text{if } \lambda \geq |\beta| \end{cases}$. To obtain the Lasso estimate of $\boldsymbol{\beta}_{0s}$, this process is repeated across all the coefficients until a convergence threshold criterion is reached.

This algorithm can be implemented with the glmnet R package (Friedman et al. 2010) as part of a more general penalty regression (elastic net), which is defined as a combination of the Ridge and Lasso penalties. Due to the structure of the algorithm, this can be used on very large data sets and can benefit from sparsity in the explanatory variables (Friedman et al. 2008).

Equivalently, the Lasso estimator of beta coefficients $\boldsymbol{\beta}_{0s}$ can be defined as

$$\begin{aligned}&\widetilde{\boldsymbol{\beta}}^{L}_{0s}(\lambda) = \underset{\boldsymbol{\beta}_{0s}}{\operatorname{argmin}} \sum_{i=1}^{n}\left(y_i - \overline{y} - \sum_{j=1}^{p} x_{ijs}\beta_{js}\right)^2 \\ &\text{subject to } \sum_{j=1}^{p}\left|\beta_{js}\right| \leq t\end{aligned}$$

With this, a graphic representation of the Lasso estimator is like the Ridge (see Fig. 3.6). The nested ellipsoids correspond to contour plots of $\mathrm{RSS}(\boldsymbol{\beta})$ and the green

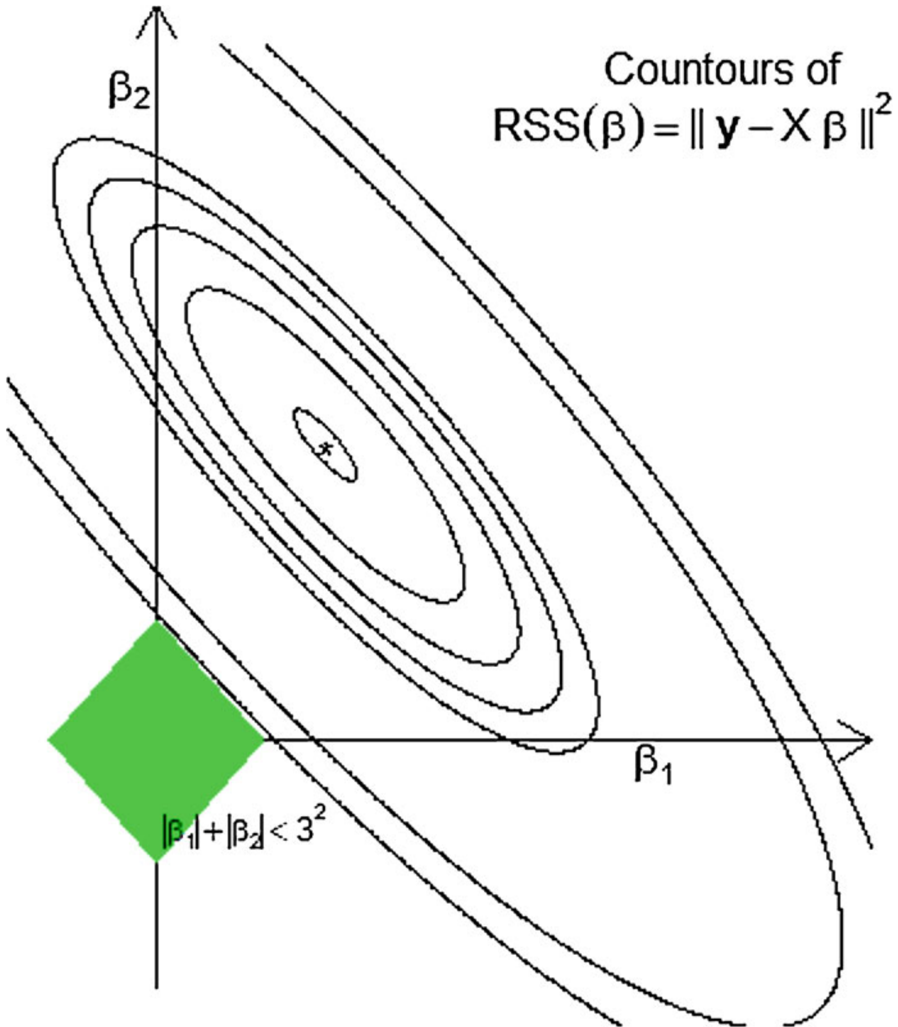



**Fig. 3.6** Graphic representation of the Lasso solution of the OLS with restriction $\sum_{j=1}^{p}\left|\beta_j\right| < 3^2$. The green region contains the Lasso solution

region is the restriction with $t = 3^2$, which contains the Lasso solution. Indeed, the Lasso solution is the first point of the contours that touches the square, and this will sometimes be in a corner that makes some coefficients zero. Because there are no corners in Ridge regression, this will rarely happen (Tibshirani 1996).

The Lasso estimator can also be derived from a Bayesian perspective. Supposing that the vector of residuals is distributed as $N_n(\mathbf{0}, \sigma^2 \boldsymbol{I}_n)$, like in the Ridge regression case, and assuming that the priors of $\boldsymbol{\beta}_{0s}$ are independent and identically distributed according to Laplace distribution with mean 0 and variance $\sigma_\beta^2$, and adopting a "flat" prior for $\mu$, with known $\sigma^2$ and $\sigma_\beta^2$, the posterior distribution of $\boldsymbol{\beta}$ is

$$
\begin{aligned}
& f(\boldsymbol{\beta}_s | \boldsymbol{y}, \boldsymbol{X}_{1s}) \\
& \propto \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{y} - \boldsymbol{X}_s\boldsymbol{\beta}_s)^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X}_s\boldsymbol{\beta}_s)\right] \prod_{i=1}^{n} \exp\left(-\frac{\sqrt{2}}{\sigma_\beta^2}|\beta_{js}|\right) \\
& \propto \exp\left\{-\frac{1}{2\sigma^2}\left[\sum_{i=1}^{n}\left(y_i - \mu - \sum_{j=1}^{p} x_{ijs}\beta_{js}\right) + \lambda\sum_{j=1}^{p}|\beta_{js}|\right]\right\},
\end{aligned}
$$

where $\lambda = \sqrt{8}\sigma^2/\sigma_\beta^2$. Then, the model of the posterior distribution of $\boldsymbol{\beta}_s$ corresponds to the Lasso estimator described before, $\widetilde{\boldsymbol{\beta}}^L(\lambda)$.

The performance of Lasso regression in terms of prediction error is sometimes comparable to Ridge regression (Hastie et al. 2009). However, as we pointed out before, and based on the nature of the restriction term, for any given value of $t$, only a subset of the coefficients $\beta_{js}$ is nonzero, so this gives a sparse solution (Efron et al. 2004).

**Example 4** To illustrate Lasso regression, here we considered the data used in Example 2, but instead of using a five-fold cross-validation (5FCV) to explore the behavior of this, we built 100 random splittings of the complete data set: 80% for training and 20% for testing. Figure 3.7 presents a representation of the MSE of the Lasso regression ($y$-axis) and the MSE corresponding to Ridge regression ($x$-axis). In 81 out of 100 random splittings, the Ridge regression approach gives a better performance, and in this case, on average, the Lasso regression shows an MSE that is 92.13% greater than the Ridge solution. In the other cases, the Ridge was worse, on average, by 30.91%.

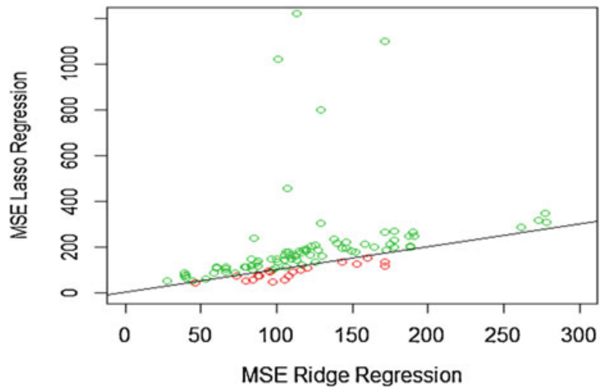



**Fig. 3.7** MSE of Ridge regression versus MSE of Lasso regression in 100 random splittings of data: 20% for testing and 80% for training

On average across all the splittings, the performance of the Ridge regression (average = 118.9726 and standard deviation = 50.7193 of MSE) was superior to the Lasso (200.6021 and standard deviation = 222.5494 of MSE) by 68.61%, but this was better than the OLS solution (average = 1609.4635 and standard deviation = 1105.4434 of MSE) by 802.32%, while the Ridge was 1352.80% better than the OLS estimate.

```
#########################R code for Example 4#########################
rm(list=ls())
library(BMTME)
data("WheatMadaToy")
dat_F = phenoMada
dim(dat_F)
dat_F$GID = as.character(dat_F$GID)
G = genoMada
eig_G = eigen(G)
G_0.5 = eig_G$vectors%*%diag(sqrt(eig_G$values))%*%t(eig_G$vectors)
X = G_0.5
y = dat_F$PH
n = length(y)
library(glmnet)
#5FCV
set.seed(3)
K = 5
Tab = data.frame()
set.seed(1)
for(k in 1:100)
{
 Pos_tr = sample(1:n,n*0.8)
 y_tr = y[Pos_tr]; X_tr = X[Pos_tr,]; n_tr = dim(X_tr)[1]
 y_tst = y[-Pos_tr]; X_tst = X[-Pos_tr,]
 #Partition for internal training the model
 Grpv_k = findInterval(cut(sample(1:n_tr,n_tr),breaks=5),1:n_tr)
 #RR
 A_RR = cv.glmnet(X_tr,y_tr,alpha=0,foldid=Grpv_k,type.
measure='mse')
 yp_RR = predict(A_RR,newx=X_tst,s='lambda.min')
 #LR
 A_LR = cv.glmnet(X_tr,y_tr,alpha=1,foldid=Grpv_k,type.
measure='mse')
 yp_LR = predict(A_LR,newx=X_tst,s='lambda.min')
 #OLS
 A_OLS = glmnet(X_tr,y_tr,alpha=1,lambda=0)
 yp_OLS = predict(A_OLS,newx=X_tst)

 Tab = rbind(Tab,data.frame(PT=k,MSEP_RR = mean((y_tst-yp_RR)^2),
              MSEP_LR = mean((y_tst-yp_LR)^2),
              MSEP_OLS = mean((y_tst-yp_OLS)^2)))
 cat('k = ', k,'\n')
}
Tab
```

Now the key components of the just given R code are

1. Hundred random partitions were implemented where each partition is obtained with Pos_tr = sample(1:n,n*0.8), which means that 80% of the data is used for training and 20% for testing, and for each training set, an inner K=5 fold CV is performed to tune the $\lambda$ hyperparameter.
2. The Grpv_k contains the information of the K=5 fold inner CV implemented to tune the hyperparameter $\lambda$.
3. Now we use the cv.glmnet function that is useful for implementing supervised learning methods with cross-validation. This function belongs to the R package glmnet and the input we give to this function is the training set (X_tr, y_tr), alpha=0, that tells glmnet to implement a Ridge regression method, while alpha=1 orders glmnet to implement a Lasso regression. In foldid=Grpv_k we are given training and testing sets to tune the hyperparameter $\lambda$, and in type. measure='mse', we are specifying the metric with which we will evaluate the prediction performance of the inner testing sets to be able to choose the best hyperparameter.
4. The function glmnet with lambda=0 implements the OLS estimator.

It is important to point out that Lasso regression performs particularly well when there is a subset of true coefficients that are small or even zero. It doesn't do as well when all of the true coefficients are moderately large; however, in this case, it can still outperform linear regression over a pretty narrow range of (small) $\lambda$ values.

## 3.7 Logistic Regression

The logistic regression is a useful and traditional tool used to explain or predict a binary response based on information of explanatory variables. It models the conditional distribution of the response variable as a Bernoulli distribution with the probability of success given by

$$P(Y_i = 1|\boldsymbol{x}_i) = p(\boldsymbol{x}_i;\boldsymbol{\beta}) = \frac{\exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)}{1 + \exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)}.$$

To estimate parameters under logistic regression, suppose that we have a set of data $\left(\boldsymbol{x}_i^{\mathrm{T}}, y_i\right)$, $i = 1, \ldots, n$ (training data), where $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})^{\mathrm{T}}$ is a vector of features measurement and $y_i$ is the response measurement corresponding to the $i$th drawn individual. To obtain the MLE of $\boldsymbol{\beta}$, first we need to build the likelihood function of the parameters of $\boldsymbol{\beta}$. This is given by

$$L(\boldsymbol{\beta};\boldsymbol{y}) = \prod_i^n p(\boldsymbol{x}_i;\boldsymbol{\beta})^{y_i}[1 - p(\boldsymbol{x}_i;\boldsymbol{\beta})]^{1-y_i} = \prod_i^n \left(\frac{p(\boldsymbol{x}_i;\boldsymbol{\beta})}{1 - p(\boldsymbol{x}_i;\boldsymbol{\beta})}\right)^{y_i}[1 - p(\boldsymbol{x}_i;\boldsymbol{\beta})]^1$$
$$= \exp\left(\sum_{i=1}^n y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)\right)\prod_{i=1}^n \frac{1}{1 + \exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)}.$$

and from here the log-likelihood is

$$\ell(\boldsymbol{\beta};\boldsymbol{y}) = \log\left[L(\boldsymbol{\beta};\boldsymbol{y})\right] = \sum_{i=1}^{n} y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right) - \sum_{i=1}^{n} \log\left[1 + \exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)\right].$$

Then, because the gradient of the likelihood is given by

$$\begin{aligned}\frac{\partial \ell(\boldsymbol{\beta};\boldsymbol{y})}{\partial \boldsymbol{\beta}} &= \begin{bmatrix} \sum_{i=1}^{n} y_i \\ \sum_{i=1}^{n} y_i x_{i1} \\ \vdots \\ \sum_{i=1}^{n} y_i x_{ip} \end{bmatrix} - \begin{bmatrix} \sum_{i=1}^{n} \frac{\exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)}{1 + \exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)} \\ \sum_{i=1}^{n} \frac{\exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)}{1 + \exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)} x_{i1} \\ \vdots \\ \sum_{i=1}^{n} \frac{\exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)}{1 + \exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)} x_{ip} \end{bmatrix} \\ &= \boldsymbol{X}^{\mathrm{T}}\boldsymbol{y} - \boldsymbol{X}^{\mathrm{T}}\boldsymbol{p}(\boldsymbol{X};\boldsymbol{\beta}) \\ &= \boldsymbol{X}^{\mathrm{T}}[\boldsymbol{y} - \boldsymbol{p}(\boldsymbol{X};\boldsymbol{\beta})],\end{aligned}$$

where $\boldsymbol{p}(\boldsymbol{X};\boldsymbol{\beta}) = [p(\boldsymbol{x}_1;\boldsymbol{\beta}), \ldots, p(\boldsymbol{x}_n;\boldsymbol{\beta})]^{\mathrm{T}}$, the MLE of $\boldsymbol{\beta}$, $\widehat{\boldsymbol{\beta}}$, can be iteratively approximated by using the gradient descent method:

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t + \alpha \boldsymbol{X}^{\mathrm{T}}[\boldsymbol{y} - \boldsymbol{p}(\boldsymbol{X};\boldsymbol{\beta}_t)].$$

For inferential purposes, we have that the asymptotic distribution of $\widehat{\boldsymbol{\beta}}$ is a multivariate normal distribution with vector mean $\boldsymbol{\beta}$ and variance–covariance matrix, the inverse of the negative of the expected value of the Hessian of the log-likelihood, $E\left\{\left[-\frac{\partial \ell(\boldsymbol{\beta};\boldsymbol{y})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^{\mathrm{T}}}\right]^{-1}\right\} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{X}\right)^{-1}$ (McCullagh and Nelder 1989). This is because

$$\begin{aligned}\frac{\partial^2 \ell(\boldsymbol{\beta};\boldsymbol{y})}{\partial \beta_j \partial \beta_k} = \frac{\partial \ell(\boldsymbol{\beta};\boldsymbol{y})}{\partial \beta_j} &= -\sum_{i=1}^{n} \frac{\exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)}{\left[1 + \exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)\right]^2} x_{ij} x_{ik} \\ &= -\sum_{i=1}^{n} p(\boldsymbol{x}_i;\boldsymbol{\beta})[1 - p(\boldsymbol{x}_i;\boldsymbol{\beta})] x_{ij} x_{ik}\end{aligned}$$

The Hessian of the log-likelihood is given by

$$\frac{\partial \ell(\boldsymbol{\beta};\boldsymbol{y})}{\partial \boldsymbol{\beta} \partial \boldsymbol{\beta}^{\mathrm{T}}} = \boldsymbol{H} = -\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{X},$$

where $\boldsymbol{W} = \mathrm{Diag}\{p(\boldsymbol{x}_1;\boldsymbol{\beta})[1 - p(\boldsymbol{x}_1;\boldsymbol{\beta})], \ldots, p(\boldsymbol{x}_n;\boldsymbol{\beta})[1 - p(\boldsymbol{x}_n;\boldsymbol{\beta})]\}$.

Once the parameters have been estimated, the prediction response is obtained from the estimated probabilities: $\widehat{y}_o = 1$ if $p\left(\boldsymbol{x}_o;\widehat{\boldsymbol{\beta}}\right) > 0.5$ and $\widehat{y}_o = 0$ if

$p\left(\boldsymbol{x}_o;\widehat{\boldsymbol{\beta}}\right) \leq 0.5$. Of course, a different threshold to 0.5 can be used and this could be considered as a hyperparameter that needs to be tuned in a similar fashion as the penalty parameter in the Ridge procedure.

It is important to point out that the minimization process of the log-likelihood can be performed using a more efficient iterative technique called the Newton–Raphson technique, which is an iterative optimization technique that uses a local quadratic approximation to the log-likelihood function. The following is the Newton–Raphson iterative equation used to search for the beta coefficients:

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t + \boldsymbol{H}^{-1}\boldsymbol{X}^{\mathrm{T}}[\boldsymbol{y} - \boldsymbol{p}(\boldsymbol{X};\boldsymbol{\beta}_t)],$$

where $\boldsymbol{H} = -\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{X}$ is the Hessian matrix whose elements comprise the second derivative of the log-likelihood with regard to the beta coefficients. Therefore, the inverse of the Hessian is $\boldsymbol{H}^{-1} = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{X})^{-1}$. So, the previous equation can be expressed as

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t + \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}[\boldsymbol{y} - \boldsymbol{p}(\boldsymbol{X};\boldsymbol{\beta}_t)].$$

It is important to recall that the Hessian is no longer constant since it depends on $\boldsymbol{\beta}$ through the weighting matrix $\boldsymbol{W}$. Also, it is clear that the logistic regression does not have a closed solution due to the nonlinearity of the logistic sigmoid function. It is important to point out that if instead of maximizing the likelihood we minimize the negative of the log-likelihood, the Newton–Raphson equation for updating the beta coefficients is equal to

$$\boldsymbol{\beta}_{t+1} = \boldsymbol{\beta}_t - \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{W}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}[\boldsymbol{y} - \boldsymbol{p}(\boldsymbol{X};\boldsymbol{\beta}_t)].$$

This Newton–Raphson algorithm for logistic regression is known as the iterative reweighted least squares since the diagonal weighting matrix $\boldsymbol{W}$ is interpreted as variances. Another alternative method for estimating the beta coefficients in logistic regression is the Fisher scoring method that is very similar to the Newton–Raphson method just described, but with the difference that instead of using the Hessian ($\boldsymbol{H}$), it uses the expected value of the Hessian matrix, $E(\boldsymbol{H})$.

### 3.7.1 Logistic Ridge Regression

Like the MLR, when there is strong collinearity, the variance of the MLE is severely affected and the true effects of the explanatory variables could be falsely identified (Lee and Silvapulle 1988). In a similar fashion as for the MLR, this could be judged directly from the asymptotic covariance matrix of $\widehat{\boldsymbol{\beta}}$. Moreover, in a common prediction context, when the number of features is larger than the number of

observations ($p \gg n$), the matrix design is not of full column rank and can cause overfitting, affecting the expected classification error (generalization error) when using the "MLE." One way to avoid overfitting is by replacing the MLE with a regularized MLE as the Ridge MLE estimator of MLR. This is defined as

$$\widetilde{\boldsymbol{\beta}}_s^R(\lambda) = \underset{\boldsymbol{\beta}_s}{\operatorname{argmax}}\left[\ell(\boldsymbol{\beta}_s;\boldsymbol{y}) - \lambda\sum_{j=1}^{p}\beta_{js}^2\right],$$

where $\lambda$ is a hyperparameter that has a similar interpretation as in the MLR.

In the literature, there are some algorithms that approximate the Ridge estimation. For example, Genkin et al. (2007) used a cyclic coordinate descent optimization algorithm to approximate this. The one-dimensional optimization problem involved is solved by a modified Newton–Raphson method. Another method was proposed by Friedman et al. (2008) in a more general context. Given the current values of $\widetilde{\boldsymbol{\beta}}_s(\lambda)$, the next update of coordinate $\beta_k$ is given by

$$\beta_{ks} = \frac{\sum_{i=1}^{n} w_i y_{ij}^* x_{ij}}{\sum_{i=1}^{n} w_i x_{ij}^2 + \lambda}$$

with $y_{ij}^* = y_i^* - \widetilde{\mu}(\lambda) - \sum_{\substack{j=1 \\ j\neq k}}^{p} x_{ijs}\widetilde{\beta}_{js}(\lambda)$ for $k = 1, \ldots, p$, and of $\mu$ is given by

$$\mu = \frac{\sum_{i=1}^{n} w_i e_i^*}{\sum_{i=1}^{n} w_i}$$

with $e_i^* = y_i^* - \sum_{j=1}^{p} x_{ijs}\widetilde{\beta}_{js}(\lambda)$, where $y_i^* = \widetilde{\beta}_0(\lambda) + \boldsymbol{x}_i^{\mathrm{T}}\widetilde{\boldsymbol{\beta}}_{0s}(\lambda) + \frac{y_i - p\left(\boldsymbol{x}_i;\widetilde{\boldsymbol{\beta}}_s(\lambda)\right)}{w_i}$ and $w_i = p\left(\boldsymbol{x}_i;\widetilde{\boldsymbol{\beta}}_s(\lambda)\right)\left[1 - p\left(\boldsymbol{x}_i;\widetilde{\boldsymbol{\beta}}_s(\lambda)\right)\right]$, $i = 1, \ldots, n$, are pseudo responses and weights that change across the updates. This can be obtained by maximizing, with respect to $\beta_{ks}$, the next quadratic approximation of the penalized likelihood at the current values of $\boldsymbol{\beta}_s$ ($\widetilde{\boldsymbol{\beta}}_s(\lambda)$)

$$\ell(\boldsymbol{\beta}_s;\boldsymbol{y}) - \lambda\sum_{j=1}^{p}|\beta_{js}| \approx \ell^*(\boldsymbol{\beta}_s;\boldsymbol{y}) - \frac{\lambda}{2}\sum_{j=1}^{p}\beta_{js}^2 + c,$$

where $\ell^*(\boldsymbol{\beta}_s;\boldsymbol{y}) = -\frac{1}{2}\sum_{i=1}^{n} w_i\left(y_i^* - \mu - \sum_{j=1}^{p} x_{ijs}\beta_{js}\right)^2$ is the quadratic approximation of $\ell(\boldsymbol{\beta}_s;\boldsymbol{y})$ at current values of $\boldsymbol{\beta}_s$, $\widetilde{\boldsymbol{\beta}}_s(\lambda)$, and $c$ is a constant that does not depend on $\boldsymbol{\beta}_s$. More details of this implementation (glmnet R package) can be found in Friedman et al. (2008). Note that under this approximation, the beta coefficients can be updated by

$$\widehat{\boldsymbol{\beta}}_s^R(\lambda) = \left(\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{W}\boldsymbol{X}_s + \lambda\boldsymbol{D}\right)^{-1}\boldsymbol{X}_s^{\mathrm{T}}\boldsymbol{W}\boldsymbol{y},$$

where $\boldsymbol{W} = \mathrm{Diag}(w_1, \ldots, w_n)$ and $\boldsymbol{D}$ is the diagonal matrix as defined before.

### *3.7.2 Lasso Logistic Regression*

The Lasso penalization can be applied to other models (Tibshirani 1996). In particular, for logistic regression, the Lasso estimator of $\boldsymbol{\beta}_s$ is defined as

$$\widetilde{\boldsymbol{\beta}}_s^L(\lambda) = \underset{\boldsymbol{\beta}_s}{\operatorname{argmax}} \ell_L(\boldsymbol{\beta}_s; \boldsymbol{y}),$$

where $\ell_L(\boldsymbol{\beta}_s; \boldsymbol{y}) = \ell(\boldsymbol{\beta}_s; \boldsymbol{y}) - \lambda\sum_{j=1}^{p}\left|\beta_{js}\right|$ and is often known as the regularized Lasso likelihood. Numerical methods are also required to obtain this Lasso estimate. There are several possibilities (Genkin et al. 2007), such as non-quadratic programming and iteratively reweighted least squares (Tibshirani 1996), but here we will briefly describe the one proposed by Friedman et al. (2008) and implemented in the glmnent R package. This method consists of applying the coordinate descent procedure to a penalized reweighted least square, which is formed by making a Taylor approximation to the likelihood around the current values of the coefficients. That is, this procedure consists of successively updating the parameters by

$$\widetilde{\boldsymbol{\beta}}_s = \underset{\boldsymbol{\beta}_s}{\operatorname{argmin}}\left(-\ell^*(\boldsymbol{\beta}_s; \boldsymbol{y}) + \frac{\lambda}{2}\sum_{j=1}^{p}\left|\beta_{js}\right|\right),$$

where $\ell^*(\boldsymbol{\beta}_s; \boldsymbol{y}) = -\frac{1}{2}\sum_{i=1}^{n} w_i\left(y_i^* - \mu - \sum_{j=1}^{p} x_{ijs}\beta_{js}\right)^2 + c,$ $y_i^* = \widetilde{\beta}_0(\lambda) + \boldsymbol{x}_i^{\mathrm{T}}\widetilde{\boldsymbol{\beta}}_{0s}(\lambda) + \frac{y_i - p\left(\boldsymbol{x}_i; \widetilde{\boldsymbol{\beta}}_s(\lambda)\right)}{w_i}$, and $w_i = p\left(\boldsymbol{x}_i; \widetilde{\boldsymbol{\beta}}_s(\lambda)\right)\left[1 - p\left(\boldsymbol{x}_i; \widetilde{\boldsymbol{\beta}}_s(\lambda)\right)\right]$, $i = 1, \ldots, n$, as defined in the Ridge regression case. More details of this implementation can be consulted in Friedman et al. (2008).

**Example 5** In this example, we used data corresponding to 40 lines planted with four repetitions. For illustrative purposes, we will use as response a binary variable based on Plant Height. The matrix of features used here was obtained from the genomic relationship ($\boldsymbol{G}$), $\boldsymbol{X} = \boldsymbol{Z}_L\boldsymbol{G}^{1/2}$, where $\boldsymbol{G}^{1/2}$ is the square root matrix of $\boldsymbol{G}$.

The performance of logistic regression, logistic Ridge regression, and Lasso logistic regression for this data set was evaluated across 100 random splittings of the complete data set: 20% for testing (evaluation performance) and 80% for training. The performance was measured by the proportion of cases correctly classified (PCCC) in the testing data. These results are summarized in Table 3.2,

**Table 3.2** Performance of the standard, Ridge, and Lasso logistic regression models

| Method | PCCC | SD |
|---|---|---|
| SLR | 0.7284 | 0.0765 |
| Ridge | 0.7240 | 0.0763 |
| Lasso | 0.7206 | 0.0705 |

PCCC denotes the proportion of cases correctly classified

where for each method the mean (PCCC) and standard deviation (SD) of the PCCC across the 100 splittings are reported. The table indicates that, on average, the standard logistic (SLR) approach shows slightly better performance than the other two approaches, even better than the Lasso solution. Out of the 100 random partitions, 72, 24, and 4, the SLR, the logistic Ridge regression (LRR), and the logistic lasso regression (LLR) resulted in the higher PCCC value, respectively. However, the difference in the performance of the three methods is not significant because of the large deviation obtained across the different partitions.

The computations were done with the help of the glmnet R package using the following R code:

```
#########################R code for Example 5#######################
load(file ='dat-E3.5.RData')
dat_F = dat$dat_F
dat_F = dat_F[order(dat_F$Rep,dat_F$GID),]
head(dat_F)
G = dat$G
dat_F$y = dat_F$Height
ZL = model.matrix(~0+GID,data=dat_F)
colnames(ZL)
Pos = match(colnames(ZL),paste('GID',colnames(G),sep=''))
max(abs(diff(Pos)))
y = dat_F$y
ei = eigen(G)
X = ZL%*%ei$vectors%*%diag(sqrt(ei$values))%*%t(ei$vectors)
n = length(y)
library(glmnet)
#5FCV
set.seed(1)
Tab = data.frame()
set.seed(1)
for(k in 1:100)
{
 Pos_tr = sample(1:n,n*0.8)
 y_tr = y[Pos_tr] ; X_tr = X[Pos_tr,]; n_tr = dim(X_tr)[1]
 y_tst = y[-Pos_tr]; X_tst = X[-Pos_tr,]
 #Partition for internal training the model
 Grpv_k = findInterval(cut(sample(1:n_tr,n_tr),breaks=5),1:n_tr)
 #RR
 A_RR = cv.glmnet(X_tr,y_tr,family='binomial',
         alpha=0,foldid=Grpv_k,type.measure='class')
 yp_RR = as.numeric(predict(A_RR,newx=X_tst,s='lambda.min',
type='class'))
```

```
 #LR
 A_LR = cv.glmnet(X_tr,y_tr,family='binomial',
         alpha=1,foldid=Grpv_k,type.measure='class')
 yp_LR = as.numeric(predict(A_LR,newx=X_tst,s='lambda.min',
type='class'))
 #SLR
 A_SLR = glmnet(X_tr,y_tr,family='binomial',alpha=0,lambda=0)
 yp_SLR = as.numeric(predict(A_SLR,newx=X_tst,type='class'))

 Tab = rbind(Tab,data.frame(PT=k,PCCC_RR = 1-mean(y_tst!=yp_RR),
               PCCC_LR = 1-mean(y_tst!=yp_LR),
               PCCC_SLR = 1-mean(y_tst!=yp_SLR)))
 cat('k = ', k,'\n')
}
Tab
```

Also, in this R code there are four relevant points:

1. Hundred random partitions were implemented, Pos_tr = sample(1:n,n*0.8), with 80% for training and 20% for testing, and for each training set, an inner K=5 fold CV is performed to tune the $\lambda$ hyperparameter.
2. Also, here Grpv_k contains the information of the K=5 inner folds to tune the hyperparameter $\lambda$.
3. The cv.glmnet function with the following input is used: (a) training set (X_tr, y_tr); (b) with alpha=0 to implement a Ridge regression and alpha=1 to implement a Lasso regression; (c) with foldid=Grpv_k containing training and testing sets to tune the hyperparameter $\lambda$; (d) with family='binomial' to implement a logistic regression; and (e) with type.measure='class' as a metric for categorical data to measure the prediction performance of the inner testing set to choose the best value of the hyperparameter $\lambda$.
4. The function glmnet with family='binomial' and lambda=0 implements the logistic regression but without penalization.

## Appendix 1: R Code for Ridge Regression Used in Example 2

The TR_RR.R script file must contain the following:

```
library(epiR)
source("RR.R")
Tr_RR_f<-function(y_tr,X_tr,K=5,KG=100,KR=1)
{
 n_tr = dim(X_tr)[1]
 X_tr_s = scale2_f(X_tr)
 mu    = mean(y_tr)
 y_tr  = y_tr-mu
 #Inner CV
 lambv = lamb_f(X_tr,K=KG,li=1e-7,ls=1-1e-7)
```

```
 iPEv_mean = 0
 for(ir in 1:KR)
 {
  iPE_mat = matrix(0,nr=length(lambv),nc=K)
  MSEP_mat=iPE_mat
  Grpv = findInterval(cut(sample(1:n_tr,n_tr),breaks=K),1:n_tr)
  for(i in 1:K)
  {
   Pos_itr = which(Grpv!=i)
   X_itr  = X_tr_s[Pos_itr,]
   y_itr  = y_tr[Pos_itr];
   y_itst  = y_tr[-Pos_itr];
   dat_itr = data.frame(y=y_itr,X=X_itr)
   n_itr = dim(X_itr)[1]
   betav_itr = A_RR_f_a_V(y_itr,X_itr,lambv)
   yp_mat   = 0 + (X_tr_s[-Pos_itr,])%*%betav_itr
   iPEv    = colMeans((matrix(y_itst,nr=length(y_itst),
                  nc=length(lambv))-yp_mat)**2)
   iPE_mat[,i] =iPEv
  }
  iPEv_mean = (ir-1)/ir*iPEv_mean + rowMeans(iPE_mat)/ir

 }

 #
 plot(log(lambv),iPEv_mean)
 Pos_o = which.min(iPEv_mean)
 lamb_o = lambv[Pos_o]
 A_RR  = RR_f(y=y_tr+mu,X=X_tr,lamb=lamb_o,Intercept = TRUE,Std=TRUE)
 betav_s = A_RR$betav_s
 betav_o = A_RR$betav_o
 list(lamb_o = lamb_o, betav_s = betav_s,
    betav_o = betav_o,
    lambv=lambv,iPEv_mean=iPEv_mean,X_tr=X_tr,y_tr =y_tr,Grpv=Grpv)
}

Pred_RR_f<-function(y_tst,X_tst,TR_RR)
{
 #betava_RR = TR_RR$betava_RR
 y_tr = TR_RR$y_tr; X_tr = TR_RR$X_tr
 X  = rbind(X_tr,X_tst)
 betav_s = TR_RR$betav_s
 betav_o = TR_RR$betav_o
 yp_tst  = c(cbind(1,X_tst)%*%betav_o)
 plot(y_tst,yp_tst); abline(a=0,b=1)
 MSEP_RR = mean((y_tst-yp_tst)**2)
 list(MSEP=MSEP_RR, betav_s = betav_s,betav_o = betav_o)
}
Pred_ols_f<-function(y_tst,X_tst,y_tr,X_tr)
{
 #OLS
 p =  dim(X_tr)[2]
 A = lm_f(y_tr,X_tr)
```

```
 sdv  = apply(X_tr,2,sd2_f)
 A_inv = diag(c(1,1/sdv))
 A_inv[1,1] = 1
 A_inv[1,2:(p+1)] = -apply(X_tr,2,mean)/sdv
 betav_s = A$betav_s
 betav_o = A_inv%*%betav_s
 yp_tst_ols = c(cbind(1,X_tst)%*%betav_o)
 MSEP_ols = mean((y_tst-yp_tst_ols)**2)
 list(MSEP=MSEP_ols,betav_s = betav_s)
}
```

The script file accessed by source("RR.R") must contain the following R code:

```
sd2_f<-function(x)
{
 (mean((x-mean(x))**2))^0.5
}
scale2_f<-function(X)
{
 scale(X,center=TRUE,scale = apply(X,2,sd2_f))
}

#RR internal standarized: zij = (xij-\bar{x}_j)/ss_j
#ss_j = mean (xij-\bar{x}_j)^2
RR_f<-function(y,X,lamb = 0, Intercept=TRUE,Std=TRUE)
{
 p   = dim(X)[2]; n = dim(X)[1]
 if(Std == TRUE)
 {
  sdv  = apply(X,2,sd2_f)
  A_inv = diag(c(1,1/sdv))
  A_inv[1,1] = 1
  A_inv[1,2:(p+1)] = -apply(X,2,mean)/sdv
  mu = mean(y)
  X_s  = scale2_f(X)
  Xa = X_s
  svd_X = svd(Xa); d1  = svd_X$d
  Gama1 = svd_X$v
  U = svd_X$u
  pa   = dim(Gama1)[2]
  betav_s = c(mu,Gama1%*%(((d1/(d1^2+lamb)))*(t(U)%*%(y-mu))))
  betav_o = A_inv%*%betav_s# betav in orginal scale
  list(betav_s = betav_s,betav_o=betav_o)
 }
 else
 {
  Xa = X
  sdv  = apply(X,2,sd2_f)
  p = dim(X)[2]
  svd_X = svd(Xa)
  d1  = svd_X$d
  Gama1 = svd_X$v; U = svd_X$u
  betav = Gama1%*%((d1/(d1^2+lamb))*(t(U)%*%(y)))#-mean(y)
```

```
    betav
    #tX = t(X)
    #betav = ginv(tX%*%X+lamb*diag(p))%*%tX%*%y
    #betav
  }
}

RR_f_V = Vectorize(RR_f,'lamb')

A_RR_f_a <-function(y_itr,X_itr,lamb)
{
 A = RR_f(y=y_itr,X=X_itr,lamb=lamb,Std=FALSE,Intercept = FALSE)
 A
}
A_RR_f_a_V<-Vectorize(A_RR_f_a,'lamb')

lamb_f<-function(X,K=100,li=0.001,ls=0.999)
{
 Xac = scale2_f(X)
 n = dim(Xac)[1]
 R2v = seq(li,ls,length=K)
 lambv = (1-R2v)/R2v*sum(diag(Xac%*%t(Xac)))/n
 lambv = exp(seq(min(log(lambv)),max(log(lambv)),length=K))
 sort(lambv,decreasing = TRUE)
}

library(MASS)
lm_f<-function(y,X)
{
 p = dim(X)[2]
 sdv  = apply(X,2,sd2_f)
 A_inv = diag(c(1,1/sdv))
 A_inv[1,1] = 1
 A_inv[1,2:(p+1)] = -apply(X,2,mean)/sdv
 X = scale2_f(X); mu = mean(y);
 svd_X = svd(X); d = svd_X$d
 d1  = svd_X$d; Gama1 = svd_X$v; U = svd_X$u
 betav_s = c(mu,Gama1%*%((1/d1)*(t(U)%*%(y-mu))))
 betav_o = A_inv%*%betav_s# betav in orginal scale
 list(betav_s = betav_s,betav_o=betav_o)
}
```

## References


Allaire JJ, Chollet F (2019) keras: R Interface to ‘Keras’. R package version, 2(4)

Beysolow T II (2017) Introduction to deep learning using R: a step-by-step guide to learning and implementing deep learning models using R. Apress, San Francisco, CA

Burden RL, Faires JD (2011) Numerical analysis. Cengage Learning, Boston, MA

Casella G, Berger RL (2002) Statistical inference. Duxbury, Thomson Learning, Pacific Grove, CA

Christensen P (2011) Plane answers to complex questions: the theory of linear models. Springer Science+Business Media, New York
Efron B, Hastie T, Johnstone I, Tibshirani R (2004) Least angle regression. Ann Stat 32(2):407–499
Friedman J, Hastie T, Hoeing H, Tibshirani R (2007) Pathwise coordinate optimization. Ann Appl Stat 2(1):302–332
Friedman J, Hastie T, Tibshirani R (2008) Sparse inverse covariance estimation with the graphical lasso. Biostatistics 9(3):432–441
Friedman J, Hastie T, Tibshirani R (2010) Regularization paths for generalized linear models via coordinate descent. J Stat Softw 33(1):1–22
Genkin A, Lewis D, Madigan D (2007) Large-scale Bayesian logistic regression for text categorization. Technometrics 49(3):291–304
Goodfellow I, Bengio Y, Courville A (2016) Deep learning. The MIT Press, Cambridge, MA
Hastie T, Tibshirani R, Friedman J (2009) The elements of statistical learning: data mining, inference, and prediction. Springer, New York
Hastie T, Tibshirani R, Wainwright M (2015) Statistical learning with sparsity: the lasso and generalizations. Chapman and Hall/CRC, New York
Haykin S (2009) Neural networks and learning machines. Pearson Prentice Hall, New Jersey, p 909
James G, Witten D, Hastie T, Tibshirani R (2013) An introduction to statistical learning: with applications in R. Springer Science+Business Media, New York
Lee AH, Silvapulle MJ (1988) Ridge estimation in logistic regression. Commun Stat Simul Comput 178(4):1231–1257. https://doi.org/10.1080/03610918808812723
McCullagh P, Nelder JA (1989) Generalized linear models. Chapman and Hall, London, England
Montgomery DC, Peck EA, Vining GG (2012) Introduction to linear regression. Wiley, Hoboken, NJ
Nocedal J, Wright SJ (2006) Numerical optimization. Springer, New York
Rencher AC, Schaalje GB (2008) Linear models in statistics. Wiley, Hoboken, NJ
Tibshirani R (1996) Regression shrinkage and selection via the lasso. J R Stat Soc B 58(1):267–288
Wakefield J (2013) Bayesian and frequentist regression methods. Springer Science+Business Media, New York
Warner B, Misra M (1996) Understanding neural networks as statistical tools. Am Stat 50 (4):284–293

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 4
# Overfitting, Model Tuning, and Evaluation of Prediction Performance

## 4.1 The Problem of Overfitting and Underfitting

The *overfitting* phenomenon occurs when the statistical machine learning model learns the training data set so well that it performs poorly on unseen data sets. In other words, this means that the predicted values match the true observed values in the training data set too well, causing what is known as overfitting. Overfitting happens when a statistical machine learning model learns the systematic and noise (random fluctuations) parts in the training data to the extent that it negatively impacts the performance of the statistical machine learning model on new data. This means that the statistical machine learning model adapts very well to the noise as well as to the signal that is present in the training data. The problem is that these concepts do not apply to independent (new) data and negatively affect the model's ability to generalize. Overfitting is more probable when learning a loss function from a complex statistical machine learning model (with more flexibility). For this reason, many nonparametric statistical machine learning models also include constraints in the loss function to improve the learning process of the statistical machine learning models. For example, artificial neural networks (ANN), mentioned later, are a nonparametric statistical machine learning model that is very flexible and is subject to overfitting training data. This problem can be addressed by dropping out (setting to zero) the weights of a certain percentage of hidden units in order to avoid overfitting.

On the other hand, an *underfitted* phenomenon occurs when few predictors are included in the statistical machine learning model, i.e., it is a very simple model that poorly represents the complete picture of the predominant data pattern. This problem also arises when the training data set is too small or not representative of the population data. An *underfitted* model does a poor job of fitting the training data and for this reason it is not expected to satisfactorily predict new data points. This implies that the predictions using unseen data are weak, since individuals are perceived as strangers unfamiliar with the training data set.

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_4

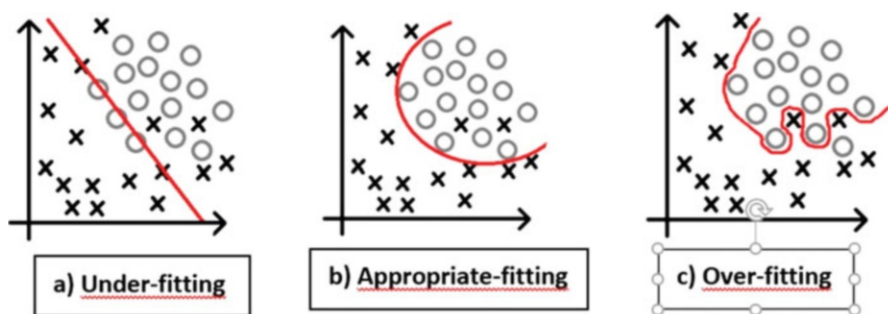



**Fig. 4.1** Schematic illustration of three models for classification: (**a**) M1 with underfitting, (**b**) M2 with appropriate fitting, and (**c**) M3 with overfitting

Consider a scattered series of points $(y_1, x_1), \ldots, (y_n, x_n)$, on a plane, to which we want to adjust a statistical machine learning method. This means that we are looking for the best $f(\boldsymbol{x}_i)$ that explains the existing relationship between the response variable $(y_i)$ and the predictors $(x_1, \ldots, x_n)$. We assume that we have three options for $f(\boldsymbol{x}_i)$: M1, the simple model plotted in Fig. 4.1a; M2, an intermediate model shown in Fig. 4.1b; and M3, a complex model shown in Fig. 4.1c.

Under the classification framework, the first panel in Fig. 4.1 (left side, panel a) shows an unsatisfactory fit (underfitted) since the line does not cover most of the points (has high bias) in the plot. As such, we expect that the prediction of unseen data of this model, M1, will perform badly. In contrast, panel c of Fig. 4.1 shows an almost perfect fit, since the predicted line covers all the data points. While at first glance, you may think that model M3 will perform well when predicting unseen data, this is actually untrue since the predicted line covers all points that are noise and those that are signal (overfit); for this reason, this type of model also performs poorly in the prediction of future data due to its complexity and high variance. Therefore, the best option for predicting unseen data is model M2 (Fig. 4.1, panel b) since it represents the predominant (smooth) pattern enough to represent the apparent data pattern while maintaining a balance between bias and variance. For this reason, a well-fitted model is one that faithfully represents the sought-after predominant pattern in the data, while ignoring the idiosyncrasies in the training data. As such, a well-fitted model in the testing set should be in the neighborhood of the model's accuracy based on the training data set, that is, the model's accuracy in the testing set should be approximately equal to that of the model's accuracy in the training set. In contrast, an overfitted model in the testing data set will be far from the neighborhood of the model's accuracy based on the training data set, and usually its prediction performance is very high (good) in the training set and consequently low (bad) in the testing set (Ratner 2017).

The paradox of overfitting is defined as complex models that contain more information about the training data, but less information about the testing data (future data we want to predict). In statistical machine learning, overfitting is a major issue and leads to some serious problems in research: (a) some relationships

that seem statistically significant are only noise, (b) the complexity of the statistical machine learning model is very large for the amount of data provided, and (c) the model in general is not replicable and predicts poorly.

Since the main goal of developing and implementing statistical machine learning methods is to predict unseen data not used for training the statistical machine learning algorithm, researchers are mainly interested in minimizing the testing error (generalization error applicable to future samples) instead of minimizing the training error that is applicable to the observed data used for training the statistical machine learning algorithm.

According to Shalev-Shwartz and Ben-David (2014), if the learning fails, these are some approaches to follow:

1. Increase the sample size of the training set.
2. Modify the hypothesis by (a) enlarging it, (b) reducing it, (c) completely changing it, and (d) changing the parameters being used. We understand a hypothesis as the models and their parameters under evaluation. This point is very important to reach a reasonable model for your data.
3. Change the feature representation of the data.
4. Change the statistical machine learning algorithm used.

## 4.2 The Trade-Off Between Prediction Accuracy and Model Interpretability

Accuracy is the ability of a statistical machine learning model to make correct predictions and those models with more complexity (called flexible models) are better in terms of accuracy, while the simple, less complex models (called inflexible models) are less accurate but more interpretable. Interpretability indicates to what degree the model allows for human understanding of natural phenomena. For these reasons, when the goal of the study is prediction, flexible models should be used; however, when the goal of the study is inference, inflexible models are more appropriate because they more easily interpret the relationship between the response variables and the predictor variables. As the complexity of the statistical machine learning model increases, the bias is reduced and the variance increases. For this reason, when more parameters are included in the statistical machine learning model, the complexity of the model increases and the variance becomes the main concern while the bias steadily falls. For example, James et al. (2013) state that the linear regression model is a relatively inflexible method because it only generates linear functions, while the support vector machine method is one of the most flexible statistical machine learning methods.

Before providing an analytical interpretation of the trade-off between the bias and variance, we must understand the meaning of both concepts. Bias is the difference between the expected prediction of our statistical machine learning model and the true observed values. For example, assume that you poll a specific city where half of

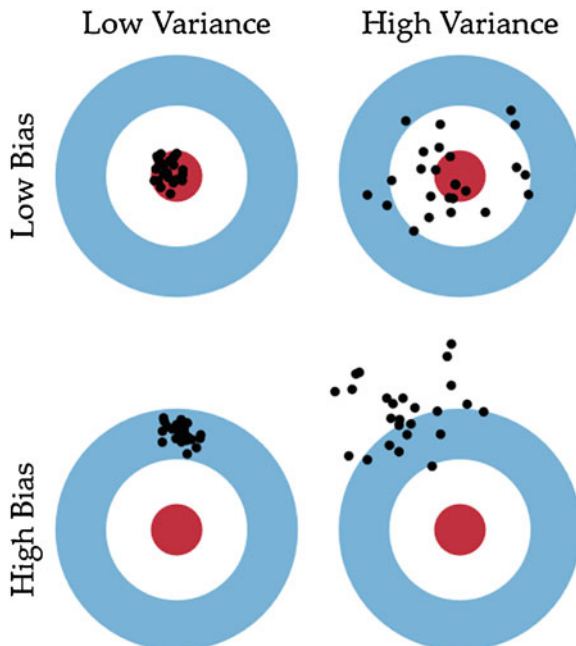



**Fig. 4.2** Graphical representations of different levels of bias and variance

the population is high-income and the other half is low-income. If you collected a sample of high-income people, you would conclude that the entire city has high income. This means that your conclusion is heavily biased since you only sampled people with high income. On the other hand, error variance refers to the amount that the estimate of the objective function will change using a different training data set. In other words, the error variance accounts for the deviation of predictions from one repetition to another using the same training set. Ideally, when a statistical machine learning model with low error variance predicts a value, the predicted value should remain almost the same, even when changing from one training data set to another; however, if the model has high variance, then the predicted values of the statistical machine learning method are affected by the values of the data set. We provide a graphical visualization of bias and variance with a bull's eye diagram (see Fig. 4.2). We assume that the center of the target is a statistical machine learning model that perfectly predicts the correct answers. As we move away from the bull's eye, our predictions get worse. Let us assume that we can repeat our entire statistical machine learning model building process to get a number of separate hits on the target. Each hit represents an individual realization of our statistical machine learning model, given the chance variability in the training data we gathered. Sometimes we accurately predict the observations of interest since we captured a representative sample in our training data, while other times we obtain unreliable predictions since our training data may be full of outliers or nonrepresentative values. The four combinations of cases resulting from both high and low bias and variance are shown in Fig. 4.2.

Burger (2018) concludes that the best scenario is the one with *low bias* and *low variance*, since samples are acceptable representatives of the population (Fig. 4.2), while in the case of *high bias* and *low variance*, the samples are fairly consistent, but not particularly representative of the population (Fig. 4.2). However, when there is *low bias* and *high variance*, the samples vary widely in their consistency, and only some may be representative of the population (Fig. 4.2). Finally, with *high bias* and *high variance*, the samples are somewhat consistent, but unlikely to be representative of the population (Fig. 4.2).

If we denote the variable we are trying to predict as $y$ and our covariates as $\boldsymbol{x}_i$, we may assume that there is a relationship between $y$ and $\boldsymbol{x}_i$, as that given in Eq. (1.1) from Chap. 1, where the error term is normally distributed with a mean of zero and variance $\sigma^2$. The expected prediction error for a new observation with value $x$, using a quadratic loss function, is given by Hastie et al. (2008, page 223):

$$
\begin{aligned}
E\left(y-\widehat{f}\{\boldsymbol{x}_i\}\right)^2 &= E(y-f\{\boldsymbol{x}_i\})^2+\left(E\left(\widehat{f}\{\boldsymbol{x}_i\}\right)-f\{\boldsymbol{x}_i\}\right)^2+E\left\{\widehat{f}(\boldsymbol{x}_i)-E\left(\widehat{f}\{\boldsymbol{x}_i\}\right)\right\}^2 \\
&= \mathrm{Var}(y)+\left[\mathrm{Bias}\left(\widehat{f}\{\boldsymbol{x}_i\}\right)\right]^2+\mathrm{Var}\left(\widehat{f}(\boldsymbol{x}_i)\right) \\
&= \mathrm{Var}(\epsilon)+\left[\mathrm{Bias}\left(\widehat{f}\{\boldsymbol{x}_i\}\right)\right]^2+\mathrm{Var}\left(\widehat{f}(\boldsymbol{x}_i)\right),
\end{aligned}
$$

where Bias is the result of misspecifying the statistical model $f$. Estimation variance (the third term) is the result of using a sample to estimate $f$. The first term is the error (irreducible error) that results even if the model is correctly specified and accurately estimated. This irreducible error is the noise term in the true relationship that cannot fundamentally be reduced by any model. Given the true model and infinite data to train (calibrate) it, we should be able to reduce both the bias and variance terms to 0. However, in a world with imperfect models and finite data, there is a trade-off between minimizing the bias and minimizing the variance. The above decomposition reveals a source of the difference between explanatory and predictive modeling: In explanatory modeling, the focus is on minimizing bias to obtain the most accurate representation of the underlying theory. In contrast, predictive modeling seeks to minimize the combination of bias and estimation variance, occasionally sacrificing theoretical accuracy for improved empirical precision (Shmueli 2010).

These four aspects impact every step of the modeling process, such that the resulting $f$ is markedly different in the explanatory and predictive contexts.

Let us assume that $f$ is a reasonable operationalization of the true function ($F$) relating constructs $X$ and $Y$. Choosing a function $f^*$ that is intentionally biased in place of $f$ is very undesirable from a theoretical–explanatory standpoint. However, the election of $f^*$ is desirable to $f$ under the prediction approach. We show this using the statistical model $y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \epsilon$, which is assumed to be correctly specified with respect to $F$. Using data, we obtain the estimated model $\widehat{f}$, which has the following properties:

$$\text{Bias} = 0$$

$$\text{Var}\left(\widehat{f}(\boldsymbol{x}_i)\right) = \text{Var}\left(\widehat{\beta}_1 x_1 + \widehat{\beta}_2 x_2 + \widehat{\beta}_3 x_3\right) = \sigma^2 \boldsymbol{x}^{\mathrm{T}} \left(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\right)^{-1} \boldsymbol{x},$$

where $\boldsymbol{x}$ is the vector $\boldsymbol{x} = [x_1, x_2, x_3]^{\mathrm{T}}$ and $\boldsymbol{X}$ is the design matrix based on all predictors. Combining the squared bias with the variance gives, as expected, the prediction error (EPE).

$$E\left(y - \widehat{f}\{\boldsymbol{x}_i\}\right)^2 = \sigma^2 + 0 + \sigma^2 \boldsymbol{x}^{\mathrm{T}} \left(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\right)^{-1} \boldsymbol{x} = \sigma^2 \left[1 + \boldsymbol{x}^{\mathrm{T}} \left(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\right)^{-1} \boldsymbol{x}\right].$$

In comparison, consider the estimated underspecified form $\widehat{f}^*(\boldsymbol{x}_i) = x_1 \widehat{\gamma}$. The bias and variance here are

$$\begin{aligned}\text{Bias} = E\left(\widehat{f}\{\boldsymbol{x}_i\}\right) - f\{\boldsymbol{x}_i\} &= x_1 \left(x_1 x_1^{\mathrm{T}}\right)^{-1} x_1^{\mathrm{T}} (\beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3) \\ &\quad - (\beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3)\end{aligned}$$

$$\text{Var}\left(\widehat{f}(\boldsymbol{x}_i)\right) = \sigma^2 x_1 \left(x_1 x_1^{\mathrm{T}}\right)^{-1} x_1^{\mathrm{T}}$$

Combining the squared bias with the variance EPE is equal to

$$\begin{aligned}E\left(y - \widehat{f}\{\boldsymbol{x}_i\}\right)^2 &= \left[x_1 \left(x_1 x_1^{\mathrm{T}}\right)^{-1} x_1^{\mathrm{T}} (x_2 \beta_2 + x_3 \beta_3) - (x_2 \beta_2 + x_3 \beta_3)\right]^2 \\ &\quad + \sigma^2 \left[1 + x_1 \left(x_1 x_1^{\mathrm{T}}\right)^{-1} x_1^{\mathrm{T}}\right].\end{aligned}$$

Although the bias of the underspecified model $f^*(\boldsymbol{x}_i)$ is larger than that of $f\{\boldsymbol{x}_i\}$, its variance can be smaller, and in some cases, so small that the overall EPE will be lower for the underspecified model. Wu et al. (2007) showed the general result for an underspecified linear regression model with multiple predictors. In particular, they showed that the underspecified model that leaves out $q$ predictors has a lower EPE when the following inequality holds:

$$q\sigma^2 > \beta_2^{\mathrm{T}} X_2^{\mathrm{T}} (I - H_1) X_2 \beta_2$$

This means that the underspecified model produces more accurate predictions, in terms of lower EPE, in the following situations: (a) when the data are very noisy (large $\sigma^2$); (b) when the true absolute values of the excluded parameters (in our example, $\beta_2$ and $\beta_3$) are small; (c) when the predictors are highly correlated; and (d) when the sample size is small or the range of left-out variables is small (Shmueli 2010).

Hagerty and Srinivasan (1991) nicely summarize this situation: "We note that the practice in applied research of concluding that a model with a higher predictive validity is "truer," is not a valid inference. This paper shows that a parsimonious but less true model can have a higher predictive validity than a truer but less parsimonious model."

## 4.3 Cross-validation

Cross-validation (CV) is a strategy for model selection or algorithm selection. CV consists of splitting the data (at least once) for estimating the error of each algorithm. Part of the data (the training set) is used for training each algorithm, and the remaining part (the testing set) is used for estimating the error of the algorithm. Then, CV selects the algorithm with the smallest estimated error. For this reason, CV is used to evaluate the prediction performance of a statistical machine learning model in out-of-sample data. This technique ensures that the data used for training the statistical machine learning model are independent of the testing data set in which the prediction performance is evaluated. It consists of repeating and recording the arithmetic average obtained from the evaluation measures on different partitions. Under $k$-fold CV, which is explained in greater detail later, this process is repeated a total of $k$ times, with each of the $k$ groups getting the chance to play the role of the test data, and the remaining $k - 1$ groups used as training data. In this way, we obtain k different estimates of the prediction error. As prediction performance is reported the average of these estimates of prediction error. CV is used in data analysis to validate the implemented models where the main objective is prediction and to estimate the prediction performance of a statistical learning model that will be carried out in practice. In other words, CV evaluates how well the statistical machine learning model generalized new data not used for training the model. The results of the CV largely depend on how the division between the training and testing sets is carried out. For this reason, in the following sections, we provide the more popular types of CV used in the implementation of statistical learning models.

### *4.3.1 The Single Hold-Out Set Approach*

The single hold-out set or validation set approach consists of randomly dividing the available data set into a training set and a validation or hold-out set (Fig. 4.3). The statistical machine learning model is trained with the training set while the hold-out

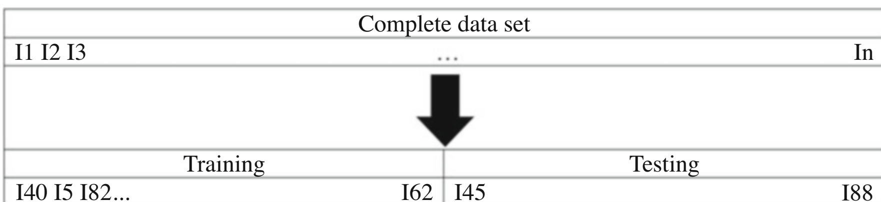



**Fig. 4.3** Schematic representation of the hold-out set approach. A set of observations are randomly split into a training set with individuals I40, I5, I82, among others, and into a testing set with observations I45, I88, among others. The statistical machine learning model is fitted on the training set and its performance is evaluated on the validation set (James et al. 2013)

set (testing set) is used to study how well that statistical machine learning model performs on unseen data. For example, 80% of the data can be used for training the model and the remaining 20% of the data for testing it. One weakness of the hold-out (validation) set approach is that it depends on just one training-testing split and its performance depends on how the data are split into the training and testing sets.

### 4.3.2 *The* k*-Fold Cross-validation*

In $k$-fold CV, the data set is randomly divided into $k$ complementary folds (groups) of approximately equal size. One of the subsets is used as testing data and the rest $(k-1)$ as training data. Then $k-1$ folds are used for training the statistical machine learning model and the remaining fold for evaluating the out-of-sample prediction performance. For these reasons, the statistical machine learning model is fitted $k$ times using a different partition (fold) as the testing set and the remaining $k-1$ as the training set. Finally, the arithmetic mean of the $k$ folds is obtained and reported as the prediction performance of the statistical machine learning model (see Fig. 4.4). This method is very accurate because it combines $k$ measures of fitness resulting from the $k$ training and testing data sets into which the original data set was divided, but at the cost of more computational resources. In practice, the choice of the number of folds depends on the measurement of the data set, although 5 or 10 folds are the most common choices.

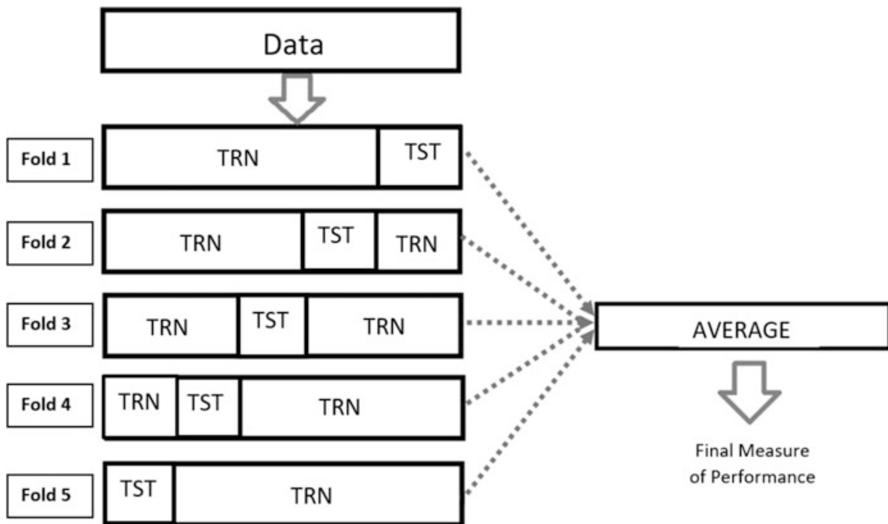



**Fig. 4.4** Schematic representation of the $k$-fold cross-validation with complementary subsets with $k = 5$

It is important to point out that to reduce variability, we recommend implementing the $k$-fold CV multiple times, each time using different complementary subsets to form the folds; the validation results are combined (e.g., averaged) over the rounds (times) to give a better estimate of the statistical machine learning model predictive performance.

### 4.3.3 The Leave-One-Out Cross-validation

The Leave-One-Out (or LOO) CV is very simple since each training data set is created by including all the individuals except one, while the testing set only includes the excluded individual. Thus, for $n$ individuals in the full data set, we have $n$ different training and testing sets. This CV scheme wastes minimal data, as only one individual is removed from the training set.

Regarding the $k$-fold cross-validation that was just explained, $n$ models are built from $n$ individuals (samples) instead of $k$ models, where $n > k$. Moreover, each model is trained on $n - 1$ samples rather than $(k - 1)n/k$. In both cases, since $k$ is normally not too large and $k < n$, LOO is more computationally expensive than $k$-fold cross-validation. In terms of prediction performance, LOO normally produces high variance for the estimation of the test error. Because $n - 1$ of the n samples is used to build each statistical machine learning model, those constructed from folds are virtually identical to each other and to the model built from the entire training set. However, when the learning curve is steep for the evaluated training size, then five- or ten-fold cross-validation usually overestimates the generalization error.

Learning curves (LC) are considered effective tools to monitor the performance of the employee exposed to a new task. LCs provide a mathematical representation of the learning process that takes place as the task is repeated. In statistical machine learning the LC is a line plot of learning ($y$-axis) over experience ($x$-axis). Learning curves are extensively used in statistical machine learning for algorithms that learn (their parameters) incrementally over time, such as deep neural networks. In general, there is considerable empirical evidence suggesting that five- or ten-fold cross-validation should be preferred to LOO.

### 4.3.4 The Leave-m-Out Cross-validation

The Leave-m-Out (LmO) CV is very similar to LOO as it creates all the possible training/test sets by removing $m$ samples from the complete set. For $n$ samples, this produces $\binom{n}{m}$ train-test pairs. Unlike LOO and $k$-fold, the test sets will overlap for $m > 1$. For example, in a Leave-2-Out CV with a data set with four samples (I1, I2,

I3, and I4), the total number of training-testing sets is equal to $\binom{4}{2} = 6$; this means that the six testing sets are: [I3, I4] [I1, I2], [I2, I4] [I1, I3], [I2, I3] [I1, I4], [I1, I4] [I2, I3],[I1, I3] [I2, I4], and [I1,I2] [I3, I4], while the training sets are the complementary elements of each testing set.

### 4.3.5 Random Cross-validation

In this type of CV, the number of partitions (independent training-testing data set splits) is defined by the user, and more partitions are better. Each partition is generated by randomly dividing the whole data set into two subsets: the training (TRN) data set and the testing (TST) data set. The percentage of the whole data set assigned to the TRN and TST data sets is also fixed by the user. For example, for each random partition, the user can decide that 80% of the whole data set can be assigned to the TRN data set and the remaining 20% to the TST data set. Random cross-validation is different from *k*-fold cross-validation because the partitions are not mutually exclusive; this means that in the random cross-validation approach, one observation can appear in more than one partition. Consequently, some samples cannot be evaluated, whereas others can be evaluated more than once, meaning that the testing and training subsets can be superimposed (Montesinos-López et al. 2018a, b). To control the randomness for reproducibility, we recommend using a specific seed in the random number generator. We recommend using at least ten random partitions to obtain enough accuracy in the estimate of prediction performance.

### 4.3.6 The Leave-One-Group-Out Cross-validation

The Leave-One-Group-Out (LOGO) CV is useful when individuals are grouped (in environments or years, or even another criterion), where the number of groups ($g$) is at least two and the information of $g - 1$ groups are used as the training set while all individuals of the remaining group are used as the testing set. For example, in the context of genomic selection, when the plant breeder is interested in predicting these lines in another environment, the same (or different) lines were frequently evaluated in $g$ environments or years, that represent the groups. Jarquín et al. (2017) denotes this type of CV strategy as CV1 in the context of plant breeding. Under this approach, the predictions are reported for each of the $g$ groups because the scientist is interested in the prediction performance of each environment. Many times, the groups are the years under study and the aim is to predict the information of a complete year. However, when the groups are years and if we suspect that there is a considerable correlation between observations that are near in time. Therefore, it is

| Fold 1 | TRN | TST | | | |
|---|---|---|---|---|---|
| Fold 2 | TRN | | TST | | |
| Fold 3 | TRN | | | TST | |
| Fold 4 | TRN | | | | TST |
| | Year 1 | Year 2 | Year 3 | Year 4 | Year 5 |

**Fig. 4.5** Schematic representation of time series data with 5 years, using the previous year for predicting the next year. However, in some practical applications, we are not interested in going too far back in time since the training and testing sets will be less related the farther you go

imperative to evaluate our statistical machine learning model for time series data on "future" observations. In this sense, the training sets are composed of the previous years to predict the subsequent year. This method can also be seen as a variation of $k$-fold CV, where the first folds are used for training the statistical machine learning model and the fold ($k + 1$) is the corresponding testing set. The main difference in this CV method is that successive training sets are supersets of those that come before them. Also, it adds all surplus data to the first training partition, which is always used to train the model (see Fig. 4.5).

Figure 4.5 shows that under this type of CV, for time series data, the predictions are for $g - 1$ years; individuals from the first year are not predicted since a training set is not available.

### *4.3.7 Bootstrap Cross-validation*

First, we will define the bootstrapping method to understand how it is used in the CV approach, which should then be straightforward. Bootstrapping is a type of resampling method where, for example, $B = 10$ samples of the same size are repeatedly drawn, with replacement, from a single original sample. Afterward, each of these $B$ samples is used to estimate statistics (for example, the mean, variance, median, minimum, etc.) of a population, and the average of all the $B$ sample estimates of the target statistic is reported as the final estimate. In the context of statistical machine learning, these samples are used to evaluate the prediction performance of the algorithm under study for unseen data. One important difference between this CV approach and all the procedures explained above is that now the training set has the same size (number of observations) as the original sample because the bootstrap method replaced some individuals more than once. According to Kuhn and Johnson (2013), as a result, some observations will be represented multiple times in the bootstrap sample, while others will not be selected at all; those observations not selected are referred to as the testing set, however, this CV strategy is quite different than the previously explained. Efron (1983) pointed out that the prediction performance of the bootstrap samples tends to have less uncertainty than the $k$-fold cross-validation since on average, 63.2% of the data points are represented (for training) at least once in any sample size. For this reason,

| Fold | Training | Testing |
|---|---|---|
| 1 | I7, I4, I6, I9, I2, I3, I4, I4, I8, I6, I8, I7 | I1, I5, I10, I11, I12 |
| 2 | I2, I8, I5, I6, I1, I4, I5, I11, I1 1,I8,I10, I5 | I3, I7, I9, I12 |
| 3 | I5, I9, I11, I3, I10, I5, I7, I2, I 3, I11, I6, I9 | I1, I4, I8, I12 |
| 4 | I10, I12, I9, I7, I4, I3, I1, I9, I3, I2, I1, I6 | I5, I8, I11 |
| 5 | I2, I10, I5, I12, I3, I6, I3, I7, I6, I6, I5, I7 | I1, I4, I8, I9 |

**Fig. 4.6** Schematic representation of bootstrap cross-validation

this CV approach has a bias similar to implementing a $k =$ two fold cross-validation, and as the training set becomes smaller, the bias becomes more problematic. To understand this CV method, we provide a simple example of how the training and testing samples are constructed. If we have a sample with 12 individuals denoted as I1, I2, . . ., I12, we will select $B = 5$ bootstrap samples. Each bootstrap sample is obtained with replacement and the individuals that appear in each one correspond to the training sample; those that are not present will correspond to the testing set. Figure 4.6 provides the five bootstrap samples; each training sample has the same size as the original, however, only some individuals appear in each bootstrap sample, while those individuals that do not appear are included in the testing set. For example, in the first fold, the training bootstrap sample contains seven different individuals (I2, I3, I4, I6, I7, I8, and I9), while the testing set contains five individuals (I1, I5, I10, I11, and I12). It is important to point out that since the training sample has the same size as the original sample, some individuals in the training sample are repeated at least twice; in the first fold, the individual I4 are repeated three times, whereas I6, I7, and I8 are repeated twice. Finally, similar to other methods, the statistical machine learning model is trained with each training set; likewise, the prediction performance of the model is evaluated in each testing set. The average of these sample predictions is reported as the estimated testing error.

### *4.3.8 Incomplete Block Cross-validation*

Incomplete block (IB) CV should be used when there are $J$ treatments evaluated in $I$ blocks and the same treatments are evaluated in all the blocks. The idea behind this

**Table 4.1** TRN data set for $I = 3$ blocks and $J = 10$ treatments with 70% of data for training

| Block | Treatments per block | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 5 | 7 | 8 | 9 | 10 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
| 3 | 1 | 2 | 4 | 6 | 7 | 8 | 10 |

CV method is that some treatments should be present in some blocks but absent in others, whereas the same treatment should be present in at least one environment (block). The theory of incomplete block designs developed in the experimental designs statistical area can be used to construct the training set. For example, under a balanced incomplete block (BIB) design, the term incomplete means that all treatments in each block cannot be evaluated, whereas balanced means that each pair of treatments occur together $\lambda$ times. The training set is constructed by first defining the % of individuals in the TRN set using the equation $sI = Jr = N_{\mathrm{TRN}}$, where $J$ represents the number of treatments under study, $I$ represents the number of blocks under study, $r$ denotes the number of repetitions of each treatment, and s denotes the treatments per block. For example, suppose that we had $J = 10$ treatments and $I = 3$ blocks (that is, 30 individuals), and we decided to use $N_{\mathrm{TRN}} = 21$ (70%) of the total individuals in the TRN set. Therefore, the number of treatments by block can be obtained by solving ($sI = N_{\mathrm{TRN}}$) for $s$, which results in $s = N_{\mathrm{TRN}}/I$. This means that $s = 21/3 = 7$ treatments per block. Then, the corresponding elements for the training set can be obtained with the function find.BIB(10, 3, 7) of the package crossdes of the R statistical software. The numbers used in the function find.BIB() denote the treatments, the blocks, and the treatments per block, respectively. Finally, the treatments that make up the TRN set are shown in Table 4.1.

According to Table 4.1, it is clear that each treatment is present in two blocks and missing in one block. For example, in Block 1 the testing set includes treatments 2, 4, and 6; in Block 2, the testing set is composed of treatments 1, 8, and 10; and in Block 3, the testing set is composed of treatments 3, 5, and 9.

### *4.3.9 Random Cross-validation with Blocks*

Random cross-validation with blocks was proposed by Lopez-Cruz et al. (2015) and belongs to the so-called replicated TRN-TST cross-validation that appears in the publication of Daetwyler et al. (2012), since some individuals can never be part of the training set. This algorithm, like the incomplete block cross-validation, is appropriate when we are interested in evaluating $J$ lines in $I$ blocks or environments and tries to mimic a prediction problem faced by breeders in incomplete field trials where lines are evaluated in some, but not all, target environments. The algorithm for constructing the TRN-TST sets is described by the following steps: Step 1. Calculate the total number of observations under study as $N = J \times I$; Step 2. Define the proportion of observations used for training and testing, that is, $P_{\mathrm{TRN}}$ and $P_{\mathrm{TST}}$; Step 3. Calculate the size of the testing set $N_{\mathrm{TST}} = N \times P_{\mathrm{TST}}$; Step 4. Choose $N_{\mathrm{TST}}$ lines at

**Table 4.2** TRN-TST data sets for $J = 10$ lines and $I = 3$ environments

| Environment | Lines | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | L1 | L2 | L3 | L4 | L5 | L6 | L7 | L8 | L9 | L10 |
| 1 | TRN | TST | TRN | TRN | TST | TRN | TRN | TST | TRN | TRN |
| 2 | TST | TRN | TRN | TRN | TRN | TRN | TST | TRN | TST | TRN |
| 3 | TRN | TRN | TST | TST | TRN | TRN | TRN | TRN | TRN | TST |

random without replacement if $J \geq N_{\mathrm{TST}}$, and with replacement otherwise; Step 5. Each chosen line will then be assigned to one of the $I$ environments chosen at random without replacement; Step 5. All the selected lines and environments will form the training set, while the lines and environments that were not chosen will form the corresponding testing set; and Step 6. Steps 1–5 are repeated depending on the number of TRN-TST partitions required (Lopez-Cruz et al. 2015). This CV is called CV2 in Jarquín et al. (2017). Next, we assume that we have ten lines or treatments and three environments or blocks that will form the corresponding training testing sets for only one partition: Step 1. The total number of observations under study is $N = 10 \times 3 = 30$; Step 2. We define $P_{\mathrm{TRN}} = 0.7$ and $P_{\mathrm{TST}} = 0.3$; Step 3. The size of the testing set is $N_{\mathrm{TST}} = 30 \times 0.3 = 9$; Step 4. Since $J = 10 \geq N_{\mathrm{TST}} = 9$, we selected the following lines at random without replacement: L1, L2, L3, L4, L5, L6, L7, L8, L9, and L10; and Step 5. Each chosen line was assigned to one of the $I = 3$ environments randomly chosen without replacement, as shown in Table 4.2. It is important to point out that this CV strategy only differs from the incomplete block cross-validation in the way the lines are allocated to blocks.

### 4.3.10 Other Options and General Comments on Cross-validation

It is important to highlight that when the data set is considerably large, it is better to randomly split it into three parts: a training set, a validation set (or tuning set), and a testing set. The training set and testing set are used as explained before, while the validation (tuning set) set is used to estimate the prediction error for model selection, which is the process of estimating the performance of different models in order to choose the best one, or to evaluate the chosen statistical machine learning model with a range of values of tuning hyperparameters to select the combination of hyperparameters with the best prediction performance and then use these hyperparameters (or best model) to evaluate the prediction performance in the testing set (Fig. 4.7). It is important to point out that Fig. 4.7 shows only one random split of the data in terms of the training, testing, and validation sets.

For example, assume that our data set has 50,000 rows (observations) and that we have decided to use 5000 of them for the testing set and another 5000 for the validation set. This means that 40,000 rows are left for the training set. At this point, we train our statistical machine learning model with each component of the

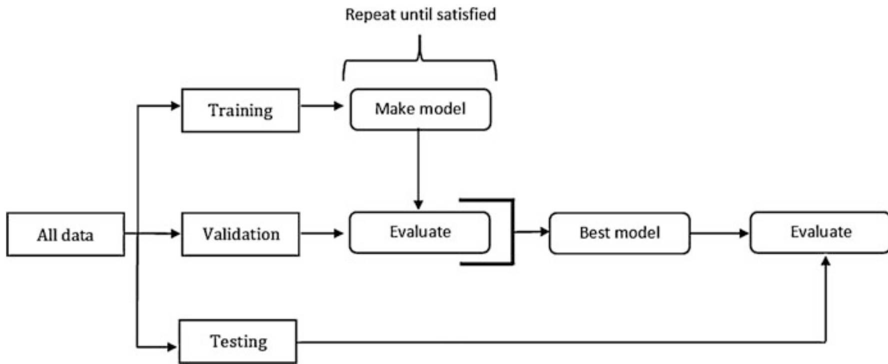



**Fig. 4.7** Schematic representation of the training, validation (tuning), and testing sets proposed by Cook (2017)

grid of hyperparameters of the training set and evaluate the prediction performance on the validation set, as shown in the middle of Fig. 4.7. Finally, we will pick the best model (best subset of hyperparameters) in terms of prediction performance in the validation set and we are ready to evaluate the prediction performance in the testing set. Then, we will report the testing error on the testing set, as can be observed at the bottom of Fig. 4.7 (Cook 2017). Under this approach, the testing and validation sets have approximately the same size to guarantee a similar out-of-sample prediction performance. Since it is difficult to give general rules on how to choose the number of observations in each of the three parts, a typical number might be 50% for training, and 25% each for validation and testing (Hastie et al. 2008) or 70%, 15%, and 15% for training, validation, and testing, respectively. Another way to find the optimal setting of hyperparameters using the grid search, which is very common in deep learning, consists of picking values for each parameter from a finite set of options [e.g., number of epochs (100, 150, 200, 250, 300); batch sizes (25, 50, 75, 100, 125); number of layers (1, 2, 3, 4, 5); and types of activation functions (RELU, Sigmoid), . . .] and training the statistical machine learning model with every permutation of hyperparameter choices using the training set. Then, the combination of hyperparameters with the best prediction performance on the validation set is chosen, and we report the prediction performance of the best selected model (set of hyperparameters) in the testing set (Buduma 2017). The aforementioned examples use the validation data set as a proxy measure of the accuracy during the hyperparameter optimization process. However, it can also be used as a proxy measure of the accuracy for model selection, and instead of using a grid of hyperparameters, we can use a set of different statistical machine learning models; here, the best model is chosen instead of the best combination of hyperparameters. It is important to understand that when you have more than one random partition (training-testing-validation), as shown in Fig. 4.8, the same process provided in Fig. 4.7 is followed, however, the average of all the partitions is reported as a measure of prediction performance. Also, if more precision is required in the estimated prediction performance, you can repeat the process given in Fig. 4.8

<table>
<tr><td>Partition 1</td><td colspan="3">TRN</td><td>TST</td><td>VAL</td></tr>
<tr><td>Partition 2</td><td colspan="2">TRN</td><td>TST</td><td>VAL</td><td>TRN</td></tr>
<tr><td>Partition 3</td><td>TRN</td><td>TST</td><td>VAL</td><td colspan="2">TRN</td></tr>
<tr><td>Partition 4</td><td>TST</td><td>VAL</td><td colspan="3">TRN</td></tr>
<tr><td>Partition 5</td><td>VAL</td><td colspan="3">TRN</td><td>TST</td></tr>
</table>

**Fig. 4.8** Five partitions of training-validation-testing

multiple times and report the average of all repetitions as a measure of prediction performance.

As mentioned above, this approach (Figs. 4.7 and 4.8) is used for a large data set however, in the case of a smaller data set, we suggest modifying this approach to avoid wasting too much training data in validation sets. This modification consists of performing an inner cross-validation approach since the training set is split into complementary subsets, and each model is trained against a different combination of these subsets, which is subsequently validated against the remaining parts. Once the model hyperparameters have been selected, a final model is trained with the whole training set (refitted) and the generalized prediction performance is measured on the testing set. This approach was applied by Montesinos-López et al. (2018a, b), who also split the original data into training and testing sets. Subsequently, each training set was split again and 80% of the data was used for training a grid of hyperparameters while the remaining 20% was used for validating (tuning) the prediction performance and selecting the best combination of hyperparameters with the best prediction performance. At this point, the deep learning algorithm was refitted with the whole training set, and with this they evaluated the out-of-sample prediction. They called the conventional training-testing partition outer cross-validation, while the split performed in each training set used for hyperparameter tuning was called inner cross-validation. It should be highlighted that there are no differences between outer and inner CV and training-validation-test. Finally, it should also be mentioned that any type of the cross-validation strategies mentioned in this section (random CV, *k*-fold CV, Bootstrap CV, IB CV, etc.,) can be used in both outer and inner CV, such as is the case with the five-fold CV.

## 4.4 Model Tuning

A hyperparameter is a parameter whose value is set before the learning process begins. Hyperparameters govern many aspects of the behavior of statistical machine learning models, such as their ability to learn features from data, the models' exhibited degree of generalizability in performance when presented with new data, as well as the time and memory cost of training the model, since different hyperparameters often result in models with significantly different performance. This means that tuning hyperparameter values is a critical aspect of the statistical machine learning training process and a key element for the quality of the resulting

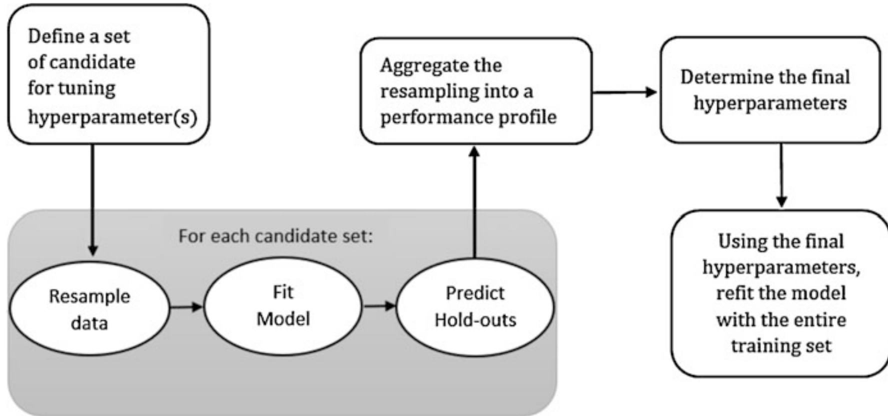



**Fig. 4.9** Schematic representation of the tuning process proposed by Kuhn and Johnson (2013)

prediction accuracies. However, choosing appropriate hyperparameters is challenging (Montesinos-López et al. 2018a). Hyperparameter tuning finds the best version of a statistical machine learning model by running many training sets on the original data set using the algorithm and ranges of values of hyperparameters as specified. The hyperparameter values that provide the best performance in out-of-sample prediction evaluated by the chosen metric are then selected.

There are many ways of searching for the best hyperparameters. However, a general approach defines a set of candidate values for each hyperparameter. Each value of this set of candidate values is then applied with a resample of the training set of the chosen statistical machine learning method, where we aggregate all the hold-out predictions from which the best hyperparameters are chosen and refit the model with the entire set (Kuhn and Johnson 2013). A schematic representation of the tuning process proposed by Kuhn and Johnson (2013) is given in Fig. 4.9. It is important to highlight that this process should be performed correctly because when the same data are used for training and evaluating the prediction performance, the prediction performance obtained is extremely optimistic.

For example, suppose a breeder is interested in developing an algorithm to classify unseen plants as diseased or not diseased with an available training data set. The goal is to minimize the rate of misclassification or to maximize the percentage of cases correctly classified (PCCC). Also, assume that you are new to the world of statistical machine learning and that you only understand the $k$-nearest neighbor method. Since this algorithm depends only on the hyperparameter called the number of neighbors ($k$), the question is which value of $k$ to choose in such a way that the prediction performance of this algorithm will be the best in the sample prediction of plants. To find the best value of the $k$ hyperparameter, you must specify a range of values for $k$ (for example, from 1 to 60 with increments of 1), then with a part of your training data set, called the training-inner (or tuning that corresponds to the training data in the inner loop) set, which is randomly selected. You proceed to evaluate the 60 values of $k$ with the $k$-nearest neighbor method and evaluate the

prediction performance in the remaining part of the training set (validation set). Next, you select the value of $k$ from this range of values that best predicts (according, for example, to the PCCC) out-of-sample data (validation set) and use this value to perform the prediction of the unseen plants not used for training the model (testing set). This is a widely adopted practice that consists of searching for the parameter (usually through brute force loops) that yields the best performance over a validation set. However, the process illustrated here is very simple because the $k$-nearest neighbor model only depends on a unique hyperparameter; however, there are other statistical machine learning algorithms (for example, deep learning methods) where the tuning process is required for a considerable amount of hyperparameters. For this reason, we encourage caution when choosing the statistical machine learning algorithm, since the amount of work required for performing the tuning process depends on the chosen method.

### *4.4.1 Why Is Model Tuning Important?*

Tuning the hyperparameters of the models is a key element to optimize your statistical machine learning model to perform well in out-of-sample predictions. The tuning process is more an art than a science because there is no unique formal scientific procedure available in the literature. Nowadays, the tuning process is trial and error that consists of implementing the statistical machine learning model many times with different values of the hyperparameters and then comparing its performance on the validation set in order to determine which set of hyperparameters results in the most accurate model; for the final implementation, the set of hyperparameters of the best model is used. As mentioned above, for the $k$-nearest neighbor classifier, we need to choose the number of neighbors ($k$) using the tuning process to obtain the optimal prediction performance of this algorithm, while for conventional Ridge regression, the parameter lambda ($\lambda$) is obtained by tuning to improve the out-of-sample predictions. These two statistical machine learning algorithms that we just mentioned need only one hyperparameter; however, other statistical machine learning methods may require more hyperparameters, as exemplified by deep learning models that require at least three hyperparameters (number of neurons, number of hidden layers, type of activation function, batch size, etc.). After tuning the required hyperparameters, the statistical machine learning model learns the parameters from the data to be used for the final prediction of the testing set. The choice of hyperparameters significantly influences the time required to train and test a statistical machine learning model. Hyperparameters can be continuous or of the integer type; for this reason, there are mixed-type hyperparameter optimization methods.

### 4.4.2 *Methods for Hyperparameter Tuning (Grid Search, Random Search, etc.)*

Manual tuning of statistical machine learning models is of course possible, but relies heavily on the user's expertise and understanding of the underlying problem. Additionally, due to factors such as time-consuming model evaluations, nonlinear hyperparameter interactions in the case of large models that consist of tens or even hundreds of hyperparameters, manual tuning may not be feasible since it is equivalent to brute force. For this reason, the four most common approaches for hyperparameter tuning reported in the literature are (a) grid search, (b) random search, (c) Latin hypercube sampling, and (d) optimization (Koch et al. 2017).

In the *grid search* method, each hyperparameter of interest is discretized into a desired set of values to be studied where the models are trained and assessed for all combinations of the values across all hyperparameters (that is, a "grid"). Although fairly simple and straightforward to carry out, a grid search is appropriate when there are only a few values for a limited number of hyperparameters. However, although this is a comprehensive way of assessing different hyperparameter values, when there are many values for some or many hyperparameters, it quickly becomes quite costly due to the number of hyperparameters and the number of discrete levels of each. For example, in Ridge regression, this approach is implemented as follows: since $\lambda$ is the hyperparameter to be tuned, we first propose, for example, a grid of 100 values for this hyperparameter from $\lambda = 10^{10}$ to $\lambda = 10^{-2}$; then we divide the training set into five inner training sets and five inner testing (tuning) sets, where each of the 100 values of the grid is fitted using the inner training sets and the testing error is evaluated with the inner testing sets. Then we get the average predicted test error and pick one value out of the 100 values of the grid that produces the best prediction performance. Next, we refit the statistical machine learning method to the whole training set using the picked value of $\lambda$, and finally perform the predictions for the testing set using the learned parameters of the training set with the best picked value of $\lambda$. In all the models with one hyperparameter, it is practical to implement the *grid search* method, but for example, in deep learning models, which many times require six hyperparameters to be tuned, if only three values are used for each hyperparameter, there are $3^6 = 729$ combinations that need to be evaluated, quickly becoming computationally impracticable.

A *random search* differs from a *grid search* in that rather than providing a discrete set of values to explore each hyperparameter, we determine a statistical distribution for each hyperparameter from which values may be randomly sampled. This affords a much greater chance of finding effective values for each hyperparameter. While Latin hypercube sampling is similar to the previous method, it is a more structured approach (McKay 1992) since it is an experimental design in which samples are exactly uniform across each hyperparameter but random in combinations. These so-called low-discrepancy point sets attempt to ensure that points are approximately equidistant from one another in order to fill the space

efficiently. This sampling supports coverage across the entire range of each hyperparameter and is more likely to find good values of each hyperparameter.

The previous two methods for hyperparameter tuning are used to perform individual experiments by building models with various hyperparameter values and recording the model performance for each. Because each experiment is performed in isolation, this process is parallelized, but is unable to use the information from one experiment to improve the next experiment. Optimization methods, on the other hand, consist of sequential model-based optimization where the results of previous experiments are used to improve the sampling method of the next experiment. These methods are designed to make intelligent use of fewer evaluations and thus save on the overall computation time (Koch et al. 2017). Optimization algorithms that have been used in statistical machine learning generally for hyperparameter tuning include Broyden–Fletcher–Goldfarb–Shanno (BFGS) (Konen et al. 2011), covariance matrix adaptation evolution strategy (CMA-ES) (Konen et al. 2011), particle swarm (PS) (Renukadevi and Thangaraj 2014), tabu search (TS), genetic algorithms (GA) (Lorena and de Carvalho 2008), and more recently, surrogate-based Bayesian optimization (Dewancker et al. 2016). Also, recently the use of the response surface methodology has been explored for tuning hyperparameters in random forest models (Lujan-Moreno et al. 2018). However, the implementation of these optimization methods is not straightforward because it requires expensive computation; also, software development is required for implementing these algorithms automatically. There have been advances in this direction for some machine learning algorithms in the statistical analysis system (SAS), R and Python software (Koch et al. 2017). An additional challenge is the potential unpredictable computation expense of training and validating predictive models using different hyperparameter values. Finally, although it is challenging, the tuning process often leads to hyperparameter settings that are better than the default values, as it provides a heuristic validation of these settings, giving greater assurance that a model configuration with a higher accuracy has not been overlooked.

## 4.5 Metrics for the Evaluation of Prediction Performance

The quality of prediction performance of any statistical machine learning method in a given data set consists of evaluating how close the predicted values are to the true observed ones. In other words, the prediction performance quantifies the matching degree between the predicted response value for a given observation and the true response value for that observation (James et al. 2013). However, the metrics used for quantifying the prediction performance depend on the type of response variable under study; for this reason, we subsequently give the most popular metrics used for this goal for four types of response variables.

### 4.5.1 Quantitative Measures of Prediction Performance

Before implementing a statistical machine learning model, we assume that we have our training observation $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ and we estimate $f$, as $\widehat{f}$, with the chosen statistical machine learning model. Then we can make predictions for each of the response values ($y_i$) with $\widehat{f}(x_i)$ and compute the predicted values for each of the $n$ observations in the training set; with these values we can calculate the ***mean square error*** (MSE) for the training data set as $E = \frac{1}{n}\left(\sum_{i=1}^{n}\left(y_i - \widehat{f}(x_i)\right)^2\right)$; however, what we really want to predict are the values for unseen test observations that were not used to train the statistical machine learning model. Assuming that the unseen testing set is equal to $\{(x_{n+1}, y_{n+1}), (x_{n+2}, y_{n+2}), \ldots, (x_{n+T}, y_{n+T})\}$, the MSE for the testing data set should be calculated as

$$\mathrm{MSE}_{\mathrm{TST}} = \frac{1}{T}\sum_{i=n+1}^{n+T}\left(y_i - \widehat{f}(x_i)\right)^2, \tag{4.1}$$

where $\widehat{f}(x_i)$ is the prediction that $\widehat{f}$ gives to the $i$th observation. The $\mathrm{MSE}_{\mathrm{TST}}$ with a lower value will have better predictions, which means that the predicted values are very close to the true observed values. Also, the square root of $\mathrm{MSE}_{\mathrm{TST}}$ can be used as a measure of prediction performance and is called root mean square error (RMSE).

***Pearson's correlation coefficient*** is a very popular measure of prediction performance in plant breeding and can be calculated as

$$r_{\mathrm{TST}} = \frac{\sum_{i=n+1}^{n+T}\left(\widehat{f}(x_i) - \overline{\widehat{f}(x_i)}\right)(y_i - \overline{y}_i)}{\sqrt{\sum_{i=n+1}^{n+T}\left(\widehat{f}(x_i) - \overline{\widehat{f}(x_i)}\right)^2}\sqrt{\sum_{i=n+1}^{n+T}(y_i - \overline{y}_i)^2}}, \tag{4.2}$$

where $\overline{\widehat{f}(x_i)}$ is the average of the $T$ predictions that conform to the testing set, and $\overline{y}_i$ is the average of the $T$ true observed values. In this case, the closer that the predictions are to 1, the better the implemented statistical machine learning model will perform. It is important to point out that Pearson's correlation is defined between $-1$ and 1. However, to be convinced that the observed and predicted values match, it is a common practice to perform a scatter plot of predicted versus observed (or vice versa) values and when the observed and predicted values follow a straight line (45° diagonal line) from the bottom left corner to the top right corner, this indicates a perfect match between the observed and predicted values. For this reason, Pearson's correlation as a metric should be complemented with the intercept and slope, since the slope and intercept describe the consistency and the model bias, respectively (Smith and Rose 1995; Mesple et al. 1996). To obtain the slope and intercept, the observed values (as $y$) versus the predicted values (as $x$) are regressed and in addition

to Pearson's correlation, these values should also be reported (slope and intercept). The expected intercept should be zero and the slope 1, if the correlation obtained between the observed and predicted values is high. It is important to avoid carrying out the regression in the opposite way, i.e., using predicted values as $y$'s, and observed values as $x$'s, since this leads to incorrect estimates of the slope and the $y$-intercept. This denotes that a spurious effect is added to the regression parameters when regressing predicted versus observed values and comparing them against the 1: 1 line. The user should also remember that underestimation of the slope and overestimation of the $y$-intercept increase as Pearson's correlation values decrease. We strongly recommend that scientists evaluate their models by regressing observed versus predicted values and test the significance of slope = 1 and intercept = 0 (Piñeiro et al. 2008). Finally, it is important to recall that the square of Pearson's correlation can also be used as a metric for measuring prediction performance since it represents the proportion of the total variance explained by the regression model and is called the coefficient of determination denoted as $R^2$.

Next, we present the mean absolute error (MAE) metric that measures the difference between two continuous variables (observed and predicted). The MAE can be calculated with the following expression:

$$\mathrm{MAE}_{\mathrm{TST}} = \frac{1}{T}\sum_{i=n+1}^{n+T}\left|y_i - \widehat{f}(x_i)\right| \tag{4.3}$$

Below we present another metric used to evaluate the prediction performance of any statistical machine learning model; it was proposed by Kim and Kim (2016) and is called ***mean arctangent absolute percentage error*** (MAAPE) that is calculated as follows:

$$\mathrm{MAAPE}_{\mathrm{TST}} = \frac{\sum_{i=n+1}^{n+T} \arctan\left(\left|\frac{y_i - \widehat{f}(x_i)}{y_i}\right|\right)}{T} \tag{4.4}$$

Although MAAPE is finite when the response variable (i.e., $y_i = 0$) equals zero, since it has a satisfactory trigonometric representation. However, because MAAPE's value is expressed in radians, it is less intuitive, in addition to being scale-free. This metric is a modification of the mean absolute percentage error (MAPE) which is problematic because it is undefined when the response variable is equal to zero ($y_i = 0$). MAAPE is also asymmetric since division by zero is defined and is not a problem. It is important to point out that there are other metrics for measuring prediction accuracy for continuous data, but we only presented the most popular ones.

The distinction between the training and test MSE is important since we are not interested in how well the statistical machine learning method performs in the training data set, due to the fact that our main goal is to perform accurate predictions in the unseen test data. For example, a plant breeder may be interested in developing an algorithm to predict disease resistance of a plant in new environments based on

records that were collected in a set of environments. We can train the statistical machine learning method with the information collected in the set of environments, but the interest is not in how well the statistical machine learning method predicts in those previously collected environments. An environmental scientist can also be interested in predicting the average annual rainfall in a municipality in Mexico, using data from the last 20 years to train the model. Such measures could include sea surface temperature, time, the yearly rotation of the earth, among others. In this case, the scientist is really interested in predicting the average rainfall of the next 1 or 2 years, not in accurately predicting the years measured in the training set.

### 4.5.2 Binary and Ordinal Measures of Prediction Performance

The binary and ordinal response variables are very common in classification problems, where the goal is to predict which category something falls into. An example of a classification problem is analyzing financial data to determine if a client will be granted credit or not. Another example is analyzing breeding data to predict if an animal is at high risk for a certain disease or not. Below, we provide some popular metrics to evaluate the prediction performance of this type of data.

The first metric is called a ***confusion matrix***, which is a tool to visualize the performance of a statistical machine learning algorithm that is used in supervised learning for classifying categorical and binary data. Each column of the matrix represents the number of predictions in each class, while each row represents the instances in the real class. One of the benefits of confusion matrices is that they make it easy to determine whether the system is confusing classes. Table 4.3 shows a sample format of a confusion matrix of C classes.

With Eqs. (4.5–4.8) we can calculate the total number of false negatives (TFN), false positives (TFP), true negatives (TTN) for each class $i$, and the total true positives in the system, respectively:

$$\mathrm{TFN}_i = \sum_{\substack{j=1 \\ j \neq i}}^{C} n_{ij} \tag{4.5}$$

**Table 4.3** Confusion matrix with more than two classes

| | | Predicted values | | | |
|---|---|---|---|---|---|
| | | Class 1 | Class 2 | … | Class C |
| Observed values | Class 1 | $n_{11}$ | $n_{12}$ | … | $n_{1C}$ |
| | Class 2 | $n_{21}$ | $n_{22}$ | … | $n_{2C}$ |
| | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| | Class C | $n_{C1}$ | $n_{C2}$ | … | $n_{CC}$ |

$$\text{TFP}_i = \sum_{\substack{j=1 \\ j \neq i}}^{C} n_{ji} \tag{4.6}$$

$$\text{TTN}_i = \sum_{\substack{j=1 \\ j \neq i}}^{C} \sum_{\substack{k=1 \\ k \neq i}}^{C} n_{ji} \tag{4.7}$$

$$\text{TTP}_{\text{all}} = \sum_{j=1}^{C} n_{jj} \tag{4.8}$$

Below, we define the sensitivity ($S_e$), precision ($P$), and specificity ($S_p$). The sensitivity indicates the ability of our statistical learning algorithm to determine the proportion of true positives that are correctly identified by the test. The precision is the proportion of correct classification of our statistical machine learning model and represents the proportion of cases correctly classified, while the specificity is the ability of our statistical machine learning model to classify the true negative cases, that is, the specificity is the proportion of true negatives that are correctly identified by the test. Under the "one-versus-all basis," where each category is compared with the composed information of the remaining categories, we provide the expressions for computing the generalized precision, sensitivity, and specificity for each class $i$:

$$P_i = \frac{\text{TTP}_{\text{all}}}{\text{TTP}_{\text{all}} + \text{TFP}_i} \tag{4.9}$$

$$S_{ei} = \frac{\text{TTP}_{\text{all}}}{\text{TTP}_{\text{all}} + \text{TFN}_i} \tag{4.10}$$

$$S_{pi} = \frac{\text{TTN}_{\text{all}}}{\text{TTN}_{\text{all}} + \text{TFP}_i} \tag{4.11}$$

$$\text{pCCC} = \frac{\text{TTN}_{\text{all}}}{\sum_{i=1}^{C} \sum_{j=1}^{C} n_{ij}} \tag{4.12}$$

The term pCCC denotes the ***proportion of cases correctly classified***, which is a measure of the overall accuracy, and when multiplied by 100, denotes the percentage of cases correctly classified. Many times, this is the only metric reported for measuring prediction performance in multi-class problems. However, PCCC alone is sometimes quite misleading as there may be a model with relatively "high" accuracy, but it predicts the "unimportant" class labels fairly accurately (e.g., "unknown bucket"). However, the model may be making all sorts of mistakes on the classes that are actually critical to the application. This problem is serious when in the input data the number of samples of different classes is very unbalanced. For

**Table 4.4** Confusion matrix with two classes

| | | Predicted values | | |
|---|---|---|---|---|
| | | True | False | Sum |
| Observed values | True | tp | fn | tp + fn |
| | False | fp | tn | fp + tn |
| | Sum | tp + fp | fn + tn | $n$ |

tp denotes true positives, fp denotes false positives, fn denotes false negatives, tn denotes true negatives, and $n$ denotes total number of individuals

example, if there are 990 samples of class 1 and only 10 of class 2, the classifier can easily have a bias toward class 1. If the classifier classifies all samples as class 1, its accuracy will be 99%. This does not mean that it is an appropriate classifier, as it had a 100% error when classifying the samples of class 2. For this reason, reporting this metric with those reported in Eqs. (4.9–4.11) is recommended in order to have a better picture of the prediction performance of any statistical machine learning method (Ratner 2017). Also, it is important to highlight that when the problem only has two classes, the confusion matrix is reduced to Table 4.4.

From Table 4.4 the PCCC is calculated as $\frac{\text{tp+tn}}{n}$, while the $S_e = \frac{\text{tp}}{\text{tp+fn}}$, $S_p = \frac{\text{tn}}{\text{tn+fp}}$, and $P = \frac{\text{tp}}{\text{tp+fp}}$. Also, when there are only two classes, González-Camacho et al. (2018) suggest calculating the ***Kappa coefficient*** ($\kappa$) or Cohen's Kappa, which is defined as

$$\kappa = \frac{P_0 - P_e}{1 - P_e},$$

where $P_0$ is the agreement between observed and predicted values and is computed by the PCCC described above for two classes; $P_e$ is the probability of agreement calculated as $P_e = \frac{\text{tp+fn}}{n} \times \frac{\text{tp+fp}}{n} + \frac{\text{fp+tn}}{n} \times \frac{\text{fn+tn}}{n}$, where fp is the number of false positives, and fn is the number of false negatives (Table 4.4). This statistic can take on values between $-1$ and 1; a value of 0 means there is no agreement between the observed and predicted classes, while a value of 1 indicates perfect agreement between the model prediction and the observed classes. Negative values indicate that a prediction may be incorrect; however large negative values seldom occur when working with predictive models. Depending on the context, a Kappa value from 0.30 to 0.50 indicates reasonable agreement (Kuhn and Johnson 2013). The Kappa coefficient is appropriate when data are unbalanced, because it estimates the proportion of cases that were correctly identified by taking into account coincidences expected from chance alone (Fielding and Bell 1997). It is important to point out that this statistic was originally designed to assess the agreement between two raters (Cohen 1960).

Another popular metric for binary data is the ***Area Under the receiver operating characteristic*** Curve (AUC–ROC) and it ranks the positive predictions higher than the negative. The ROC curve is defined as a plot of 1 − specificity or false positive rate (FPR) as the $x$-axis versus its model sensitivity as the $y$-axis. For a given set of

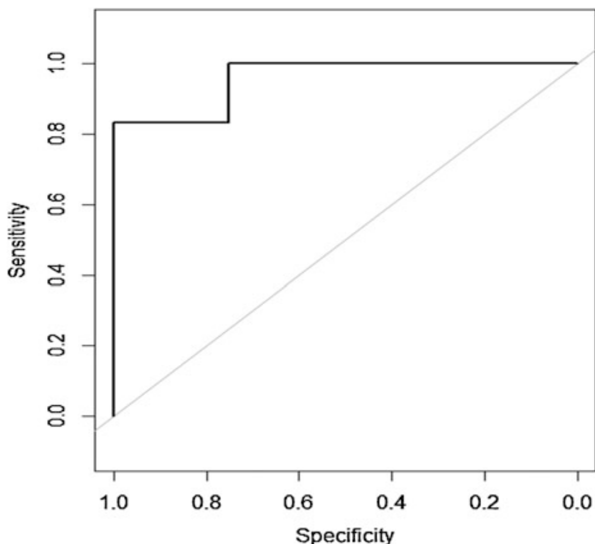



**Fig. 4.10** ROC curve for the synthetic data

thresholds $\tau$, it is an effective method for evaluating the quality or performance of diagnostic tests, and is widely used in statistical machine learning to evaluate the prediction performance of learning algorithms. Since the mathematical construction of this metric is not required in this book, we illustrate its calculation with one simple example. Assume that the observed (y), predicted probabilities (*pi*) and predicted values ($\hat{y}$) obtained after implementing a statistical machine learning model are $y = \{1, 0, 1, 1, 0, 0, 1, 1, 0, 1\}$, $pi = \{0.6, 0.55, 0.8, 0.78, 0.3, 0.42, 0.9, 0.45, 0.3, 0.88\}$, and $\widehat{y} = \{1, 1, 1, 1, 0, 0, 1, 0, 0, 1\}$; then, using the following R code, we can obtain many metrics for binary data (Fig. 4.10):

```
########################Libraries required########################
  library(caret)
  library(pROC)
  ##Observed (y), predicted probability (pi) and predicted values of
synthetic data##
  y=c(1, 0,1, 1, 0, 0, 1, 1, 0, 1)
  pi=c(0.6, 0.55, 0.8, 0.78, 0.3, 0.42, 0.9, 0.45, 0.3, 0.88)
  yhat=c(1, 1, 1, 1, 0, 0, 1, 0, 0, 1)
  xtab <- table(y,yhat)
  confusionMatrix(xtab)

  plot.roc(y,pi) #####This make the ROC curve plot

  Confusion Matrix and Statistics

   yhat
  y  0 1
   0 3 1
   1 1 5
```

```
              Accuracy : 0.8
                95% CI : (0.4439, 0.9748)
   No Information Rate : 0.6
   P-Value [Acc > NIR] : 0.1673

                 Kappa : 0.5833
Mcnemar's Test P-Value : 1.0000

           Sensitivity : 0.7500
           Specificity : 0.8333
        Pos Pred Value : 0.7500
        Neg Pred Value : 0.8333
            Prevalence : 0.4000
        Detection Rate : 0.3000
  Detection Prevalence : 0.4000
     Balanced Accuracy : 0.7917

      'Positive' Class : 0
```

It is important to point out that when there are more than two classes, these metrics (accuracy, sensitivity, specificity, etc.) can be calculated on a "one-versus-all" basis that consists of using each class versus the pool of the remaining classes, as was illustrated for the confusion matrix with more than two classes (James et al. 2013).

When the statistical machine learning model discriminates correctly between the two groups, it produces a curve that coincides with the left and top sides of the plot. Under this scenario, the perfect model would have 100% sensitivity and specificity, and the ROC curve would be a single step between (0,0) and (0,1) and would remain constant from (0,1) to (1,1); this implies that the area under the ROC curve of the model would be 1. In general, the larger the area under the ROC curve, the better the model in terms of prediction performance. On the other hand, a completely useless statistical machine learning algorithm would give a straight line (45° diagonal line) from the bottom left corner to the top right corner of the plot. Different statistical machine learning models or the same model with different training sets or hyperparameters can be compared by superimposing their ROC curves in the same graph. In practice, most of the time the values in the two groups overlap, so the curve often lies between these extremes.

From this ROC curve, we can obtain a global assessment of the prediction performance of the statistical machine learning method by measuring the area under the receiver operating characteristic curve. This area is equal to the probability that a random individual (person) of the sample with the presence of the target has a higher value of the measurement than a random individual without the target.

When a statistical machine learning model is unable to discriminate between the positive and negative classes, this means that it has low discriminatory power. Therefore, only in statistical machine learning models that had good discriminatory power we can be confident of the predictions they provide and furthermore those models that provide a curve that lies considerably above the curve will be better.

***Matthews correlation coefficient*** (MCC). Introduced in 1975 by Brian Matthews (1975) and regarded by many scientists as the most informative score that connects all four measures in a confusion matrix, the Matthews Correlation Coefficient is typically used in statistical machine learning to measure the quality of binary classifications and it is particularly useful when there is a significant imbalance in class sizes (data). MCC is calculated according to the following expression:

$$\mathrm{MCC} = \frac{\mathrm{tp} \times \mathrm{tn} - \mathrm{fp} \times \mathrm{fn}}{\sqrt{(\mathrm{tp} + \mathrm{fp}) \times (\mathrm{tp} + \mathrm{fn}) \times (\mathrm{tn} + \mathrm{fp}) \times (\mathrm{tn} + \mathrm{fn})}} \tag{4.13}$$

If any of the denominator terms equals zero in (4.13) it will be set to 1 and MCC becomes zero, which has been shown to be the correct limiting value. It returns a value between $-1$ and 1, where 1 means a perfect prediction, 0 means no better than random and $-1$ means a total disagreement between predicted and observed values.

Next, we present the ***Brier score*** (Brier 1950) for categorical or binary data that can be computed as

$$\mathrm{BS} = T^{-1} \sum_{i=n+1}^{n+T} \sum_{c=1}^{C} (\widehat{\pi}_{ic} - d_{ic})^2, \tag{4.14}$$

where $\widehat{\pi}_i$ denotes the estimated probabilities (predictive distribution) derived from the estimated model for observation $i$ and $d_{ic}$ takes a value of 1 if the categorical response observed for individual $i$ falls into category $c$; otherwise, $d_{ic} = 0$. The range of BS in Eq. (4.14) for categorical data is between 0 and 2. For this reason, we suggest dividing by 2, that is, BS/2, to obtain the Brier score bound between 0 and 1; lower scores imply better predictions (Montesinos-López et al. 2015a, b).

Finally, we describe the use of negative log-likelihood (MLL) to evaluate the prediction performance. This metric has the characteristic that better forecasts have lower values and for this reason, it is analogous to the MSE. For categorical data,

$$\mathrm{MLL} = -\frac{1}{T}\left[\sum_{i=n+1}^{n+T} \sum_{c=1}^{C} 1\{y_i = k\} \log\left(\widehat{\pi}_{ic}\right)\right],$$

where $1\{y^{(i)} = k\}$ is an indicator variable taken the value of 1 when the $i$th observation is assigned to category $c$, for $c = 1, 2, \ldots, C$ takes place in the $i$th observation. When the data are binary, the MLL is reduced to $\mathrm{MLL} = -\frac{1}{T}\left[\sum_{i=n+1}^{n+T} [y_i \log\left(\widehat{\pi}_i\right) + (1 - y_i) \log\left(1 - \widehat{\pi}_i\right)]\right]$. Following, we provide some advantages of using the MML as a measure of prediction performance: (a) it has a simple definition that, from a purely intuitive point of view, seems to be a reasonable basis on which to compare forecasts; (b) it is mathematically optimal in the sense that estimates of parameters of calibration models fitted by maximizing the likelihood are usually the most accurate possible estimates (see Cassella and Berger 2002); (c) it is a generalization to probabilistic forecasts of the most commonly used skill score for single forecasts; (d) the properties of the likelihood have been studied

at great length over the last 90 years, and as such is well understood; (e) it is both a measure of resolution and reliability; (f) likelihood can be used for both calibration and assessment: this creates consistency between these two operations; (g) use of the likelihood also creates consistency with other statistical modeling activities, since most other statistical modeling uses the likelihood, which is important in cases where the use of forecasts is simply a small part of a larger statistical modeling effort, as is the case of our particular business; (h) likelihood can be used for all types of response variables; and (i) likelihood can be used to compare multiple leads, multiple variables, and multiple locations at the same time in a sensible way with a single score even when these leads, variables, and locations are cross-correlated.

### 4.5.3 Count Measures of Prediction Performance

Spearman's correlation and the MML are recommended to measure the prediction performance for count data.

For the application of the ***Spearman's correlation***, the formula given in Eq. (4.2) for Pearson's correlation can be used; however, instead of using the observed and predicted values directly, these are replaced by their corresponding ranks. For example, assuming that the observed and predicted values are $y = \{15, 9, 12, 27, 6, 3, 36, 15, 21, 30\}$ and $\widehat{y} = \{20, 17, 24, 25, 3, 3, 34, 22, 21, 33\}$, we can thus show how to get the rank for the observed values: $\text{rango}_y = \{5, 3, 4, 8, 2, 1, 10, 6, 7, 9\}$. However, in this vector, observations 1 and 8 are the same, and as such, their positions are added and divided by two, that is, $\frac{5+6}{2} = 5.5$. Therefore, the final rank for the observed values is $\text{rango}_y = \{5.5, 3, 4, 8, 2, 1, 10, 5.5, 7, 9\}$. Now the range for the predicted values is $\text{rango}_{\widehat{y}} = \{4, 3, 7, 8, 1, 2, 10, 6, 5, 9\}$, but again, since values 5 and 6 are the same, we add their ranges, and as this is repeated twice, it is divided by two and we get $\frac{1+2}{2} = 1.5$. Therefore, the final range of the predicted values is $\text{rango}_{\widehat{y}} = \{4, 3, 7, 8, 1.5, 1.5, 10, 6, 5, 9\}$. Finally, to obtain the Spearman correlation, we used the expression given in Eq. (4.2) for Pearson's correlation, and instead of using the original observed and predicted values, we used $\text{rango}_y$ and $\text{rango}_{\widehat{y}}$. The interpretation of this metric is equal to that of the Pearson correlation, that is, when it is closer to 1, the prediction performance of the implemented statistical learning method is better. It should be noted that when the number of repeated values in the observed and predicted values is greater than two, the adjusted range is the sum of the repeated ranges divided by the number of repeated values; this new range is then given to the repeated values. In this case, it is also important to regress the observed versus the predicted values to obtain the intercept and slope using the ranges of the observed and predicted values.

It is also possible to use the MLL criteria to assess the prediction performance for count data; however, the new expression is now based on minus the log-likelihood of a Poisson distribution, which is equal to

$$\mathrm{MLL}=\frac{1}{T}\sum_{i=n+1}^{n+T}\left[-\widehat{f}(x_i)+y_i\log\left(\widehat{f}(x_i)\right)\right]$$

Again, when the values of MLL are lower, the observed and predicted values are closer to one another.

## References

Brier GW (1950) Verification of forecasts expressed in terms of probability. Mon Weather Rev 78: 1–3

Buduma M (2017) Fundamentals of deep learning, 1st edn. O'Reilly, Sabastopol, CA

Burger SV (2018) Introduction to machine learning with R. Rigorous mathematical analysis, 1st edn. O'Reilly, Sabastopol, CA

Cassella G, Berger RL (2002) Statistical inference. Duxbury, Belmont, CA

Cohen J (1960) A coefficient of agreement for national data. Educ Psychol Meas 20:37–46

Cook D (2017) Practical machine learning with $H_2O$. O'Reilly Media, Inc, Sabastopol, CA

Daetwyler HD, Calus MPL, Pong-Wong R, de los Campos G, Hickey JM (2012) Genomic prediction in animals and plants: simulation of data, validation, reporting and benchmarking. Genetics 193:347–365

Dewancker I, McCourt M, Clark S, Hayes P, Johnson A, Ke G (2016) A stratified analysis of Bayesian optimization methods. arXiv:1603.09441v1

Efron B (1983) Estimating the error rate of a prediction rule: improvement on cross-validation. J Am Stat Assoc 78(382):316–331

Fielding AH, Bell JF (1997) A review of methods for the assessment of prediction errors in conservation presence/absence models. Environ Conserv 24:38–49. https://doi.org/10.1017/S0376892997000088

González-Camacho JM, Ornella L, Pérez-Rodríguez P, Gianola D, Dreisigacker S, Crossa J (2018) Applications of machine learning methods to genomic selection in breeding wheat for rust resistance. Plant Genome 11(2):1–15. https://doi.org/10.3835/plantgenome2017.11.0104

Hagerty MR, Srinivasan S (1991) Comparing the predictive powers of alternative multiple regression models. Psychometrika 56:77–85. MR1115296

Hastie T, Tibshirani R, Friedman J (2008) The elements of statistical learning: data mining, inference, and prediction, Springer series in statistics, 2nd edn. Springer, New York

James G, Witten D, Hastie T, Tibshirani R (2013) An introduction to statistical learning: with applications in R. Springer, New York

Jarquín D, Lemes da Silva C, Gaynor RC, Poland J, Fritz AR et al (2017) Increasing genomic-enabled prediction accuracy by modeling genotype × environment interactions in Kansas wheat. Plant Genome 10(2):1–15. https://doi.org/10.3835/plantgenome2016.12.0130

Kim S, Kim H (2016) A new metric of absolute percentage error for intermittent demand forecasts. Int J Forecast 32(3):669–679

Koch P, Wujek B, Golovidov O, Gardner S (2017) Automated hyperparameter tuning for effective machine learning. In: Proceedings of the SAS global forum 2017 conference. SAS Institute Inc, Cary, NC. http://support.sas.com/resources/papers/proceedings17/SAS514-2017.pdf

Konen W, Koch P, Flasch O, Bartz-Beielstein T, Friese M, Naujoks B (2011) Tuned data mining: a benchmark study on different tuners. In: Proceedings of the 13th annual conference on genetic and evolutionary computation (GECCO-2011). SIGEVO/ACM, New York

Kuhn M, Johnson K (2013) Applied predictive modeling. Springer, New York

Lopez-Cruz M, Crossa J, Bonnett D, Dreisigacker S, Poland J, Jannink J-L, Singh RP, Autrique E, de los Campos, G. (2015) Increased prediction accuracy in wheat breeding trials using a marker × environment interaction genomic selection method. G3 5(4):569–582

Lorena AC, de Carvalho ACPLF (2008) Evolutionary tuning of SVM parameter values in multiclass problems. Neurocomputing 71:3326–3334

Lujan-Moreno GA, Howard PR, Rojas OG, Montgomery DC (2018) Design of experiments and response surface methodology to tune machine learning hyperparameters, with a random forest case-study. Expert Syst Appl 109:195–205

Matthews BW (1975) Comparison of the predicted and observed secondary structure of T4 phage lysozyme. Biochim Biophys Acta Protein Struct 405:442–451

McKay MD (1992) Latin hypercube sampling as a tool in uncertainty analysis of computer models. In: Swain JJ, Goldsman D, Crain RC, Wilson JR (eds) Proceedings of the 24th conference on winter simulation (WSC 1992). ACM, New York, pp 557–564

Mesple F, Troussellier M, Casellas C, Legendre P (1996) Evaluation of simple statistical criteria to qualify a simulation. Ecol Model 88:9–18

Montesinos-López OA, Montesinos-López A, Pérez-Rodríguez P, de los Campos G, Eskridge KM, Crossa J (2015a) Threshold models for genome-enabled prediction of ordinal categorical traits in plant breeding. G3 5(1):291–300

Montesinos-López OA, Montesinos-López A, Crossa J, Burgueño J, Eskridge K (2015b) Genomic-enabled prediction of ordinal data with Bayesian logistic ordinal regression. G3 5(10):2113–2126. https://doi.org/10.1534/g3.115.021154

Montesinos-López A, Montesinos-López OA, Gianola D, Crossa J, Hernández-Suárez CM (2018a) Multi-environment genomic prediction of plant traits using deep learners with a dense architecture. G3 8(12):3813–3828. https://doi.org/10.1534/g3.118.200740

Montesinos-López OA, Montesinos-López A, Crossa J, Gianola D, Hernández-Suárez CM et al (2018b) Multi-trait, multi-environment deep learning modeling for genomic-enabled prediction of plant traits. G3 8(12):3829–3840. https://doi.org/10.1534/g3.118.200728

Piñeiro G, Perelman S, Guerschman JP, Paruelo JM (2008) How to evaluate models: observed vs. predicted or predicted vs. observed? Ecol Model 216:316–322

Ratner B (2017) Statistical and machine-learning data mining. Techniques for better predictive modelling and analysis of big data, 3rd edn. CRC Press Taylor & Francis Group, Boca Raton, FL

Renukadevi NT, Thangaraj P (2014) Performance analysis of optimization techniques for medical image retrieval. J Theor Appl Inf Technol 59:390–399

Shalev-Shwartz S, Ben-David S (2014) Understanding machine learning from theory to algorithms. Cambridge University press, New York

Shmueli G (2010) To explain or to predict? Stat Sci 25(3):289–310

Smith EP, Rose KA (1995) Model goodness-of-fit analysis using regression and related techniques. Ecol Model 77:49–64

Wu S, Harris T, Mcauley K (2007) The use of simplified or misspecified models: linear case. Canad J Chem Eng 85:386–398

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 5
# Linear Mixed Models

## 5.1 General of Linear Mixed Models

Linear mixed models (LMM) are flexible extensions of linear models in which fixed and random effects enter linearly into the model. This is useful in many disciplines to model repeated, longitudinal, or clustered observations, in which random effects are introduced to help capture correlation or/and random variation among observations in the same group of individuals. Random effects are random values associated with levels of a random factor, and often represent random deviations from the population mean and linear relationships described by fixed effects (Pinheiro and Bates 2000; West et al. 2014).

The first formulation of a linear mixed model was applied in the field of astronomy to analyze repeated telescopic observations made at various hourly intervals over a range of nights (West et al. 2014). The mixed model approach is often called by various names, depending on the discipline in which it is applied. For example, in the social sciences, this approach is known as a multilevel or hierarchical model that is often used to flexibly measure the different levels of grouping present in the data structure (e.g., an impact evaluation of a new teaching method, survey of job satisfaction, education applications, etc.) (Goldstein 2011; Speelman et al. 2018; Finch et al. 2019).

Other application areas can be found in medicine (health care research, Leyland and Goldstein 2001; Brown and Prescott 2014), agriculture, ecology, industry, and animal science, where this model is often referred to as the random effects or mixed-effects model (Pinheiro and Bates 2000; Raudenbush and Bryk 2002; Meeker et al. 2011; Zuur et al. 2009). Specifically, now there is an increasing number of applications of this model in genomic selection for plant and animal breeding, where molecular markers obtained by genotyping-by-sequencing or other technologies are used to predict breeding values for non-phenotyped lines to select candidate lines prior to phenotypic evaluation (Meuwissen et al. 2001; Poland et al. 2012; Cabrera-Bosquet et al. 2012; Araus and Cairns 2014; Crossa et al. 2017;

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_5

Covarrubias-Pazaran et al. 2018; Wang et al. 2018; Cappa et al. 2019). However, the use of this model in animal science can be traced back to Henderson (1950).

The general univariate linear mixed model (Harville 1977) is provided by the formula

$$\boldsymbol{Y} = \boldsymbol{X\beta} + \boldsymbol{Zb} + \boldsymbol{\epsilon}, \tag{5.1}$$

where $\boldsymbol{Y}$ is the $n \times 1$ random response vector, $\boldsymbol{X}$ is the $n \times (p + 1)$ design matrix for the fixed effects, $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)^{\mathrm{T}}$ is the $(p + 1) \times 1$ coefficient vector of fixed effects, $\boldsymbol{b}$ is a $q \times 1$ vector of random effects, $\boldsymbol{Z}$ is the associate matrix design for the random effects, and $\boldsymbol{\epsilon}$ is the $n \times 1$ vector of random errors. It assumes that $\boldsymbol{\epsilon}$ is a random vector with a mean vector of $\boldsymbol{0}$ and a variance–covariance matrix $\boldsymbol{R}$, $\boldsymbol{b}$ is a random vector with a mean of $\boldsymbol{0}$ and variance–covariance matrix $\boldsymbol{D}$, and a null variance–covariance matrix between $\boldsymbol{\epsilon}$ and $\boldsymbol{b}$, $\mathrm{Cov}(\boldsymbol{\epsilon}, \boldsymbol{b}) = \boldsymbol{0}_{n \times q}$. In genomic applications, $\boldsymbol{b}$ often includes the genotypic effects and genotype $\times$ environment interaction effects, while $\boldsymbol{X}$ may contain information about environment covariates and other related information.

Note that under this model, $E(\boldsymbol{Y}) = \boldsymbol{X\beta}$ and the variance–covariance matrix of the response vector is $\mathrm{Var}(\boldsymbol{Y}) = \boldsymbol{ZDZ}^{\mathrm{T}} + \boldsymbol{R}$.

## 5.2 Estimation of the Linear Mixed Model

### *5.2.1 Maximum Likelihood Estimation*

One method typically used for the estimation of the parameters of the LMM is the maximum likelihood approach. For the estimation under an LMM, the random errors and the random effects components are needed. Assuming that $\boldsymbol{\epsilon} \sim N_n(\boldsymbol{0}, \boldsymbol{R})$, and $\boldsymbol{b} \sim N_q(\boldsymbol{0}, \boldsymbol{D})$, with $\boldsymbol{R}$ and $\boldsymbol{D}$ positive semi-defined matrices, the marginal distribution of the response vector $\boldsymbol{Y}$ is $N_n(\boldsymbol{X\beta}, \boldsymbol{ZDZ}^{\mathrm{T}} + \boldsymbol{R})$, and so the likelihood of the parameters is given by

$$L(\boldsymbol{\beta}, \boldsymbol{D}, \boldsymbol{R}; \boldsymbol{y}) = \frac{|\boldsymbol{V}|^{-\frac{1}{2}}}{(2\pi)^{\frac{n}{2}}} \exp\left[-\frac{1}{2}(\boldsymbol{y} - \boldsymbol{X\beta})^{\mathrm{T}} \boldsymbol{V}^{-1} (\boldsymbol{y} - \boldsymbol{X\beta})\right], \tag{5.2}$$

where $\boldsymbol{V} = \boldsymbol{Z}^{\mathrm{T}}\boldsymbol{DZ} + \boldsymbol{R}$ is the marginal variance of $\boldsymbol{Y}$.

The maximum likelihood estimators (MLE) of the parameters, $\boldsymbol{\beta}$, $\boldsymbol{D}$, and $\boldsymbol{R}$, are the values that maximize the likelihood function (5.2) (Searle et al. 2006; Stroup 2012), but due to the fact that no explicit formulas to estimate these parameters exist, numerical methods such as Newton–Raphson and Fisher Scoring are used. See details for implementing these methods in Jennrich and Sampson (1976) for the case where: $\boldsymbol{D}$ is a block diagonal with a submatrix in each diagonal of the form $\sigma_j^2 \boldsymbol{A}_j$, where $\boldsymbol{A}_j$ is a known matrix and $\sigma_j^2$ is the variance component parameter to be

estimated in this case; while for $\boldsymbol{R} = \sigma^2\boldsymbol{C}$, where $\boldsymbol{C}$ is a known matrix, only $\sigma^2$ should be estimated. For a more general explanation, see Jennrich and Schluchter (1986), and for an improvement of the algorithms proposed by these authors, consult Lindstrom and Bates (1988).

Another numerical method that can be used to obtain the MLE is the expected maximization (EM) algorithm, which is conceptually a simple algorithm for parameter estimation in this model (Laird and Ware 1982). This algorithm is an iterative numerical method to obtain maximum likelihood in the context of missing or hidden data (Borman 2004). The EM algorithm is described for the case where $\boldsymbol{R} = \sigma^2\boldsymbol{I}_n$, and where $\boldsymbol{I}_n$ is the identity matrix of dimension $n$. This algorithm, for some specific variance–covariance matrices of random effects, as described and used below, can be implemented using the sommer R package (Covarrubias-Pazaran 2016, 2018), which provides two additional algorithms available to obtain the MLE of the parameters in this same model.

#### 5.2.1.1 EM Algorithm

The likelihood for complete data, $\boldsymbol{y}$ and $\boldsymbol{b}$, is given by

$$\begin{aligned} f_{\boldsymbol{Y},\boldsymbol{b}}(\boldsymbol{y},\boldsymbol{b}) &= f_{\boldsymbol{Y}|\boldsymbol{b}}(\boldsymbol{y}|\boldsymbol{b})f_{\boldsymbol{b}}(\boldsymbol{b}) \\ &= \frac{|\boldsymbol{D}|^{-\frac{1}{2}}}{(2\pi\sigma^2)^{\frac{n}{2}}}\exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}-\boldsymbol{Z}\boldsymbol{b})^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}-\boldsymbol{Z}\boldsymbol{b})-\frac{1}{2}\boldsymbol{b}^{\mathrm{T}}\boldsymbol{D}^{-1}\boldsymbol{b}\right] \end{aligned}$$

As such, the log-likelihood for the complete data, $\boldsymbol{y}$ and $\boldsymbol{b}$, is given by

$$\begin{aligned} \ell_c(\boldsymbol{\beta},\boldsymbol{\theta};\boldsymbol{y},\boldsymbol{b}) &= \log\left[f_{\boldsymbol{Y},\boldsymbol{b}}(\boldsymbol{y},\boldsymbol{b})\right] \\ &= -\frac{n}{2}\log\left(2\pi\sigma^2\right)-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}-\boldsymbol{Z}\boldsymbol{b})^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}-\boldsymbol{Z}\boldsymbol{b}) \\ &\quad -\frac{1}{2}\boldsymbol{b}^{\mathrm{T}}\boldsymbol{D}^{-1}\boldsymbol{b}-\frac{1}{2}\log(|\boldsymbol{D}|), \end{aligned}$$

where $\boldsymbol{\theta}$ is the vector parameter that defines the variance–covariance matrix of the random effects ($\boldsymbol{D}$) and the random vector ($\boldsymbol{R}$). Some specific examples are given below.

E Step

Because $E(\boldsymbol{u}^{\mathrm{T}}\boldsymbol{A}\boldsymbol{u}) = \mathrm{tr}[\boldsymbol{A}\mathrm{Var}(\boldsymbol{u})] + E(\boldsymbol{u})^{\mathrm{T}}\boldsymbol{A}E(\boldsymbol{u})$ and $\boldsymbol{b} \mid \boldsymbol{Y} = \boldsymbol{y} \sim N_q\left(\widetilde{\boldsymbol{b}},\widetilde{\boldsymbol{D}}\right)$ (see Appendix 1), given the current values of the parameters $\boldsymbol{\beta}_{(t)}$ and $\boldsymbol{\theta}_{(t)}$, the conditional expected value of the complete likelihood, $\ell_c(\boldsymbol{\beta},\boldsymbol{\theta};\boldsymbol{y},\boldsymbol{b})$, is given by [Step (E)]:

$$
\begin{aligned}
Q\left(\boldsymbol{\beta},\boldsymbol{\theta}|\boldsymbol{\beta}_{(t)},\boldsymbol{\theta}_{(t)},\right) &= E_{\boldsymbol{b}|\boldsymbol{y}}[\ell_c(\boldsymbol{\beta},\boldsymbol{b},\boldsymbol{\theta};\boldsymbol{y})] \\
&= -\frac{n}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\mathrm{tr}\left(\boldsymbol{Z}\widetilde{\boldsymbol{D}}_{(t)}\boldsymbol{Z}^{\mathrm{T}}\right) - \frac{1}{2\sigma^2}\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}-\boldsymbol{Z}\widetilde{\boldsymbol{b}}_{(t)}\right)^{\mathrm{T}}\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}-\boldsymbol{Z}\widetilde{\boldsymbol{b}}_{(t)}\right) \\
&\quad -\frac{1}{2}\mathrm{tr}\left(\boldsymbol{D}^{-1}\widetilde{\boldsymbol{D}}_{(t)}\right) - \frac{1}{2}\widetilde{\boldsymbol{b}}_{(t)}^{\mathrm{T}}\boldsymbol{D}^{-1}\widetilde{\boldsymbol{b}}_{(t)} - \frac{1}{2}\log\left(|\boldsymbol{D}|\right) \\
&= -\frac{n}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\left[\mathrm{tr}\left(\boldsymbol{Z}\widetilde{\boldsymbol{D}}_{(t)}\boldsymbol{Z}^{\mathrm{T}}\right) + \left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}-\boldsymbol{Z}\widetilde{\boldsymbol{b}}_{(t)}\right)^{\mathrm{T}}\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}-\boldsymbol{Z}\widetilde{\boldsymbol{b}}_{(t)}\right)\right] \\
&\quad -\frac{1}{2}\left\{\mathrm{tr}\left[\left(\widetilde{\boldsymbol{D}}_{(t)}+\widetilde{\boldsymbol{b}}_{(t)}\widetilde{\boldsymbol{b}}_{(t)}^{\mathrm{T}}\right)\boldsymbol{D}^{-1}\right] + \log\left(|\boldsymbol{D}|\right)\right\}
\end{aligned}
$$

where $\widetilde{\boldsymbol{D}}_{(t)} = \left(\boldsymbol{D}_{(t)}^{-1} + \sigma_{(t)}^{-2}\boldsymbol{Z}^{\mathrm{T}}\boldsymbol{Z}\right)^{-1}$, $\widetilde{\boldsymbol{b}}_{(t)} = \sigma_{(t)}^{-2}\widetilde{\boldsymbol{D}}_{(t)}\boldsymbol{Z}^{\mathrm{T}}\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}_{(t)}\right)$, and $\boldsymbol{\beta}_{(t)}$, $\sigma_{(t)}^2$, and $\boldsymbol{D}_{(t+1)}$ are the current values of $\boldsymbol{\beta}$, $\sigma^2$, and $\boldsymbol{D}$, respectively.

M Step

The second step of the EM algorithm is the M step, which consists of updating the parameters by maximizing the conditional expected value of the complete likelihood. First, we can observe that for any value of $\boldsymbol{\theta}$, the value of $\boldsymbol{\beta}$ that maximizes $Q(\boldsymbol{\beta},\boldsymbol{\theta}|\boldsymbol{\beta}_{(t)},\boldsymbol{\theta}_{(t)})$ is given by

$$
\boldsymbol{\beta}_{(t+1)} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\left(\boldsymbol{y}-\boldsymbol{Z}\widetilde{\boldsymbol{b}}_{(t)}\right)
$$

which does not depend on the chosen values of $\boldsymbol{\theta}$, but rather specifically on the chosen values of $\sigma^2$ and $\boldsymbol{D}$. Then by equating to zero, the derivative of $Q(\boldsymbol{\beta}_{(t+1)},\boldsymbol{\theta}|\boldsymbol{\beta}_{(t)},\boldsymbol{\theta}_{(t)})$ with respect to $\sigma^2$, and solving for $\sigma^2$, we can obtain that the value of $\sigma^2$ that maximizes $Q(\boldsymbol{\beta}_{(t+1)},\boldsymbol{\theta}|\boldsymbol{\beta}_{(t)},\boldsymbol{\theta}_{(t)})$, for $\boldsymbol{D}$ fixed, is given by

$$
\sigma_{(t+1)}^2 = \frac{1}{n}\left[\mathrm{tr}\left(\boldsymbol{Z}\widetilde{\boldsymbol{D}}_{(t)}\boldsymbol{Z}^{\mathrm{T}}\right) + \left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}_{(t+1)}-\boldsymbol{Z}\widetilde{\boldsymbol{b}}_{(t)}\right)^{\mathrm{T}}\left(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}_{(t+1)}-\boldsymbol{Z}\widetilde{\boldsymbol{b}}_{(t)}\right)\right]
$$

which is independent of the value of $\boldsymbol{D}$. Now, according to result 4.10 in Johnson and Wichern (2002), the value of $\boldsymbol{D}$ that maximizes $Q(\boldsymbol{\beta},\boldsymbol{\theta}|\boldsymbol{\beta}_{(t)},\boldsymbol{\theta}_{(t)})$ is given by

$$
\boldsymbol{D}_{(t+1)} = \widetilde{\boldsymbol{D}}_{(t)} + \widetilde{\boldsymbol{b}}_{(t)}\widetilde{\boldsymbol{b}}_{(t)}^{\mathrm{T}}
$$

and does not depend on $\boldsymbol{\beta}$ and $\sigma^2$. So, by joining the above optimization, we have the M step that consists of updating the parameters $\boldsymbol{\beta}$, $\sigma^2$, and $\mathbf{D}$, with $\boldsymbol{\beta}_{(t+1)}$, $\sigma_{(t+1)}^2$, and $\boldsymbol{D}_{(t+1)}$, respectively. At this point, we can observe that the current value of the parameters $\boldsymbol{\beta}_{(t)}$, $\sigma_{(t)}^2$, and $\boldsymbol{D}_{(t)}$ are used in the computation of $\widetilde{\boldsymbol{b}}$ and $\widetilde{\boldsymbol{D}}$.

In the case of $\boldsymbol{D} = \sigma_g^2\boldsymbol{A}$, where $\boldsymbol{A}$ is a known matrix (the case in some genomic prediction models, where $\boldsymbol{A}$ corresponds to the genomic relationship matrix, the pedigree matrix, or the environmental matrix), the unique variance parameters to

estimate are $\sigma_g^2$ and $\sigma^2$, that is, $\left(\boldsymbol{\theta} = \left(\sigma_g^2, \sigma^2\right)\right)$. In the same fashion, very often in genomic applications but also in a more general setting, $\boldsymbol{D} = \text{Diag}\left(\sigma_1^2\boldsymbol{A}_1, \ldots, \sigma_K^2\boldsymbol{A}_K\right)$, where $\boldsymbol{A}_k$ represents a known variance–covariance matrix (or correlation) structure (genomic relationship matrix, pedigree relationship matrix, etc., Burgueño et al. 2012) between the different random effects included. In this context, the variance component parameters to estimate are $\sigma_k^2, k = 1, \ldots, K,$ and $\sigma^2$ $\left(\boldsymbol{\theta} = \left(\sigma_1^2, \ldots, \sigma_K^2, \sigma^2\right)\right)$, where $\boldsymbol{A}_k$, $k = 1, \ldots, K$, are positive defined known matrices of dimensions $q_k \times q_k$, $k = 1, \ldots, K$, such that $\sum_{k=1}^{K} q_k = q$, the E step can be reduced (see Appendix 2) to

$$Q\left(\boldsymbol{\beta},\boldsymbol{\theta}|\boldsymbol{\beta}_{(t)},\boldsymbol{\theta}_{(t)}\right) = -\frac{n}{2}\log\left(2\pi\sigma^2\right) - \frac{1}{2\sigma^2}\left[\text{tr}\left(\boldsymbol{Z}\widetilde{\boldsymbol{D}}_{(t)}\boldsymbol{Z}^{\text{T}}\right) + \left(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{Z}\widetilde{\boldsymbol{b}}_{(t)}\right)^{\text{T}}\left(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{Z}\widetilde{\boldsymbol{b}}_{(t)}\right)\right]$$
$$-\frac{1}{2}\sum_{k=1}^{K}\left\{\frac{\sigma_{k(t)}^2}{\sigma_k^2}\left[q_k - \sigma_{k(t)}^2\text{tr}\left(\boldsymbol{A}_k\boldsymbol{Z}_k^{\text{T}}\boldsymbol{V}_{(t)}^{-1}\boldsymbol{Z}_k\right) + \sigma_{k(t)}^{-2}\widetilde{\boldsymbol{b}}_{k(t)}^{\text{T}}\boldsymbol{A}_k^{-1}\widetilde{\boldsymbol{b}}_{k(t)}\right] + q_k\log\left(\sigma_k^2\right) + \log\left(|\boldsymbol{A}_k|\right)\right\},$$

where $\boldsymbol{V}_{(t)}$ is the marginal matrix of variance–covariance of the response vector in the current value of the parameters, $\boldsymbol{Z} = [\boldsymbol{Z}_1\boldsymbol{Z}_2\cdots\boldsymbol{Z}_K]$ is the partitioned design matrix of random effects, with $\boldsymbol{Z}_k$ $n \times q_k$, the corresponding matrix design for the random effects $k$, $\boldsymbol{b}_k$ ($\boldsymbol{b}^{\text{T}} = \left(\boldsymbol{b}_1^{\text{T}}, \ldots, \boldsymbol{b}_K^{\text{T}}\right)$), $\widetilde{\boldsymbol{b}}_{(t)}^{\text{T}} = \left(\widetilde{\boldsymbol{b}}_{1(t)}^{\text{T}}, \ldots, \widetilde{\boldsymbol{b}}_{K(t)}^{\text{T}}\right)$, and $\sigma_{k(t)}^2, k = 1, \ldots, K,$ are the current values of the variance parameters. Finally, for this specific model, the maximization updates for the beta coefficients and variance components are the same as before, where the variance components are

$$\sigma_{k(t+1)}^2 = \frac{1}{q_k}\sigma_{k(t)}^2\left[q_k - \sigma_{k(t)}^2\text{tr}\left(\boldsymbol{A}_k\boldsymbol{Z}_k^{\text{T}}\boldsymbol{V}_{(t)}^{-1}\boldsymbol{Z}_k\right) + \sigma_{k(t)}^{-2}\widetilde{\boldsymbol{b}}_{k(t)}^{\text{T}}\boldsymbol{A}_k^{-1}\widetilde{\boldsymbol{b}}_{k(t)}\right], k = 1, \ldots, K.$$

These are obtained by maximizing $Q(\boldsymbol{\beta}, \boldsymbol{\theta}|\boldsymbol{\beta}_{(t)}, \boldsymbol{\theta}_{(t)})$ (defined above) with respect to $\sigma_k^2, k = 1, \ldots, K$.

#### 5.2.1.2 REML

An alternative to the ML estimation of the variance components of model (5.1) and to avoid the underestimation of the maximum likelihood method is the restricted maximum likelihood estimation method (REML) proposed by Patterson and Thompson (1971). Among the several ways to define this, one is discussed by Laird and Ware (1982), which under a Bayesian paradigm, consists of estimating the parameters of the variance components by maximizing the marginal posterior distribution of the variance components by assuming a "locally" uniform prior to the distribution for $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$, that is, $f(\boldsymbol{\beta}, \boldsymbol{\theta}) \propto 1$ (Pinheiro and Bates 2000). The marginal posterior of the variance components is given in Appendix 3

$$f(\boldsymbol{\theta}|\boldsymbol{y}) \propto \int f(\boldsymbol{\beta}, \boldsymbol{\theta}|\boldsymbol{y})d\boldsymbol{\beta} \propto \int f(\boldsymbol{y}|\boldsymbol{\beta}, \boldsymbol{\theta})d\boldsymbol{\beta}$$
$$\propto |\boldsymbol{V}|^{-\frac{1}{2}}|\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}|^{\frac{1}{2}}\exp\left\{-\frac{1}{2}\left(\boldsymbol{y}-\boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right)^{\mathrm{T}}\boldsymbol{V}^{-1}\left(\boldsymbol{y}-\boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right)\right\},$$

where $\widetilde{\boldsymbol{\beta}} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{y}$, which corresponds to the maximum likelihood of the fixed effects when the variance components are assumed known as generalized least squares estimates of $\boldsymbol{\beta}$ (GLS). Then, the restricted maximum likelihood estimators (REML) $\boldsymbol{\theta}$ are those that maximize $f(\boldsymbol{\theta}|\boldsymbol{y})$, or equivalently, the REML $\boldsymbol{\theta}$ can be defined as maximizing the

$$\ell_R(\boldsymbol{\theta};\boldsymbol{y}) = -\frac{1}{2}\log\left(\left|\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\right|\right) - \frac{1}{2}\log\left(|\boldsymbol{V}|\right) - \frac{1}{2}\left(\boldsymbol{y}-\boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right)^{\mathrm{T}}\boldsymbol{V}^{-1}\left(\boldsymbol{y}-\boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right)$$

This function is known as the restricted likelihood because it can be shown that this also corresponds to the likelihood associated with the maximum number $(n-p-1)$ of linearly independent error contrasts $\boldsymbol{FY}$, where $\boldsymbol{F}$ is a full row rank $(n-p-1)\times n$ known matrix such that $\boldsymbol{FX}=\boldsymbol{0}$. It is important to point out that the associated likelihood based on the transformed data, $\boldsymbol{FY}$, gives the same result for any chosen contrast error matrix $\boldsymbol{F}$ and, consequently, this is invariant to fixed effect parameters (Harville 1974). Equivalently, the REML of $\boldsymbol{\beta}$ and $\boldsymbol{\theta}$ can be defined as those that maximize the

$$\ell_R(\boldsymbol{\beta}, \boldsymbol{\theta};\boldsymbol{y}) = -\frac{1}{2}\log\left(\left|\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\right|\right) - \frac{1}{2}\log\left(|\boldsymbol{V}|\right) - \frac{1}{2}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})^{\mathrm{T}}\boldsymbol{V}^{-1}(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta})$$

This objective function is like the natural logarithm of likelihood function given in Eq. (5.2) (log-likelihood) except for the first term. To obtain the REML solutions or the maximum a posteriory (when adopting a locally uniform prior for the parameter, as described before) of the variance components parameters, (like for the MLE) numerical methods are required. See Jennrich and Schluchter (1986) and Lindstrom and Bates (1988) for details on the Newton–Raphson and Fisher Scoring algorithms; consult the lme4 R package (Bates et al. 2015) which uses a generic nonlinear optimizer and implements a large variety of different models (MLE and REML) that arise from the LMM when different structures of the variance of random effects and errors are adopted. For a derivation of the EM algorithm to obtain the REML, see Laird and Ware (1982) under model (5.1) for longitudinal data; this same approach can be used for the genomic model previously described, where $\boldsymbol{D} = \mathrm{Diag}\left(\sigma_1^2\boldsymbol{A}_1, \ldots, , \sigma_K^2\boldsymbol{A}_K\right)$ and $\boldsymbol{R} = \sigma^2\boldsymbol{I}_n$. Consult Searle (1993) and Covarrubias-Pazaran (2016, 2018) for an implementation of this algorithm with the EM function in the sommer R package.

#### 5.2.1.3 BLUPs

In many situations, in addition to the estimation of the fixed effects, the prediction of the random effects is also of interest. A standard method for "estimating" the random effects is the best linear unbiased predictor (BLUP; Robinson 1991), which originally was developed by Henderson (1975) in animal breeding for estimating merit in dairy cattle and is now commonly employed in many research areas (Piepho et al. 2008). If the variance components $\boldsymbol{D}$ and $\boldsymbol{R}$ are known, the best linear unbiased predictor (BLUP) of the random effects $\boldsymbol{b}$ is given by

$$\widetilde{\boldsymbol{b}}^{*} = \boldsymbol{D}\boldsymbol{Z}^{\mathrm{T}}\boldsymbol{V}^{-1}\left(\boldsymbol{y} - \boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right),$$

where $\widetilde{\boldsymbol{\beta}} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{y}$ is the generalized least squared (GLS) estimator of $\boldsymbol{\beta}$. This can be obtained by maximizing with respect to $\boldsymbol{\beta}$ and $\boldsymbol{b}$, the joint density of $\boldsymbol{y}$ and $\boldsymbol{b}$, $f_{\boldsymbol{Y},\boldsymbol{b}}(\boldsymbol{y}, \boldsymbol{b})$, and is the reason why Harville (1985) called these estimates of realized values $\boldsymbol{b}$ (McLean et al. 1991), or likewise by solving the mixed model equations (MME) (Henderson 1975):

$$\begin{bmatrix} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{X} & \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{Z} \\ \boldsymbol{Z}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{X} & \boldsymbol{Z}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{Z} + \boldsymbol{D}^{-1} \end{bmatrix} \begin{bmatrix} \widetilde{\boldsymbol{\beta}} \\ \widetilde{\boldsymbol{b}}^{*} \end{bmatrix} = \begin{bmatrix} \boldsymbol{X}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{y} \\ \boldsymbol{Z}^{\mathrm{T}}\boldsymbol{R}^{-1}\boldsymbol{y} \end{bmatrix}$$

from which the inversion of the variance–covariance matrix of $\boldsymbol{Y}$ is avoided, which can be helpful in some situations to save considerable computational resources. Note that $\widetilde{\boldsymbol{b}}^{*}$ corresponds to the posterior mean of the random effects where the fixed effects are replaced by its GLS (Searle et al. 2006).

When the variance components are unknown, which is most often the case, they are frequently estimated using restricted maximum likelihood estimators, which replace them in the corresponding equations. Then, the approximate best linear unbiased predictor is obtained and is referred to as the estimated or empirical best linear unbiased predictor (EBLUP) (Rencher 2008).

To solve the mixed model equations, there are several software packages that can be useful, but one in particular in the genomic context is the sommer package (Covarrubias-Pazaran 2016, 2018) that internally solves the MME after the variance components are estimated. The github version of the sommer R package can be accessed at https://github.com/cran/sommer and can be installed with the following commands:

```
install.packages('devtools');
 library(devtools);
 install_github('covaruber/sommer')
```

## 5.3 Linear Mixed Models in Genomic Prediction

In a simple genomic prediction context where $\boldsymbol{b}$ includes the genotype effects, and the genomic relationship matrix (VanRaden 2008), that is, $\boldsymbol{G}$ is available, very often the assumed variance–covariance matrix of the random effects is $\sigma_g^2\boldsymbol{G}$ and the errors are assumed as independently and identically distributed, $\boldsymbol{R} = \sigma^2\boldsymbol{I}_n$, where $n$ is the total number of observations. In this case, the resultant model is known as the GBLUP model and when the pedigree is used, it is referred to as PBLUP. Another kind of information between lines can also be used, such as the relationship matrices derived from hyperspectral reflectance information (Krause et al. 2019). Other extensions of this model can be developed by taking into account other factors, for example, genotype× environment interaction, as will be illustrated later in the genomic prediction context.

In this case, where only the genotypic effects are taken into account, in the linear mixed model (5.1), the fixed effects design matrix is $\boldsymbol{X} = \boldsymbol{1}_n$, where the vector of length $n$ corresponds to the general mean $\boldsymbol{\beta} = \beta_0$, $\boldsymbol{b} = (b_1, b_2, \ldots, b_J)^{\mathrm{T}}$ contains the genotypic effects of $J$ lines, and $\boldsymbol{Z}$ is the incidence matrix design for the random line effects ($\boldsymbol{Z}_L$):

$$\boldsymbol{Y} = \boldsymbol{1}_n\mu + \boldsymbol{Z}_L\boldsymbol{b} + \boldsymbol{\epsilon}, \tag{5.3}$$

where $\boldsymbol{b} \sim N_J\left(\boldsymbol{0}, \sigma_g^2\boldsymbol{G}\right)$ and $\boldsymbol{R} = \sigma^2\boldsymbol{I}_n$.

The basic code to implement the GBLUP model (5.3) with the sommer package is the following:

```
A = mmer(y ~ 1,random= ~ vs(GID,Gu=G) , rcov= ~ vs(units),
         data=dat_F, verbose=FALSE)
```

where **y** and **GID** are the column names that contain the response variable and genotypes in data set **dat_F**. **G** is the genomic relationship matrix for lines which is specified in the Gu argument, that in general serves to provide a known variance–covariance matrix between the levels of the random effects (GID). In the “rcov” option, the argument “units” is always used to specify the error term.

## 5.4 Illustrative Examples of the Univariate LMM

**Example 1** To illustrate the performance of the LMM in a genomic prediction context doing the fitting process with the sommer package, we considered a wheat data set that consisted of 500 markers measured for each line as the genomic information, and with 229 observations in total that registered grain yield (tons/ha): 30 lines in four environments with one or two repetitions.

**Table 5.1** Prediction performance of the GBLUP model (5.3, M1) and the model (5.3) that results from ignoring the genomic information (M10): mean squared error of prediction (MSE) and Pearson's correlation (PC), and its standard deviation for each criterion is reported for each partition

| | M1 | | M10 | |
|---|---|---|---|---|
| PT | MSE | PC | MSE | PC |
| 1 | 0.7025 | 0.6917 | 0.6924 | 0.6969 |
| 2 | 0.4583 | 0.6681 | 0.4407 | 0.7044 |
| 3 | 0.5075 | 0.5251 | 0.4923 | 0.5506 |
| 4 | 0.4719 | 0.6024 | 0.4468 | 0.6328 |
| 5 | 0.6433 | 0.5479 | 0.6433 | 0.5307 |
| 6 | 0.3657 | 0.5088 | 0.3569 | 0.5255 |
| 7 | 0.6923 | 0.5081 | 0.6777 | 0.5243 |
| 8 | 0.3285 | 0.5054 | 0.2952 | 0.57 |
| 9 | 0.4364 | 0.6855 | 0.429 | 0.6962 |
| 10 | 0.6807 | 0.6374 | 0.6513 | 0.6811 |
| Average (SD) | 0.5287 (0.14) | 0.5881 (0.078) | 0.5126 (0.143) | 0.6112 (0.078) |

The prediction performance of the model given in Eq. (5.3) (**M1**) was evaluated with 10 random partitions, where each partition was made up of two subsets, one containing 80% of the data and used for training the model, and the other containing the remaining 20% of the data and used to evaluate the prediction performance of the model in terms of the mean squared error of prediction (MSE).

Furthermore, this model assumes that the errors were independently and identically distributed as $\boldsymbol{\epsilon} \sim N_n(\mathbf{0}, \sigma^2 \boldsymbol{I}_n)$; independently of the genotypic effects, $\boldsymbol{b}$ was assumed multivariate normal with a null mean vector and a variance–covariance matrix equal to $\sigma_g^2 \boldsymbol{G}$, where the genomic relationship matrix was computed with the information of the 500 markers.

The variance components parameters, in this case $\sigma_g^2$ and $\sigma^2$, were estimated by restricted maximum likelihood estimation with the mmer function in the sommer package, using the default algorithm optimization, the Newton–Raphson method. For univariate response variables, the EM algorithm through the EM function in this R package can also be used.

The results are shown in Table 5.1, where we also present the Pearson's correlation (PC) and MSE of the same model but without taking into account the information of the genomic relationship between lines ($\boldsymbol{G}$), that is, the variance–covariance matrix for the genotypic effects is assumed to be $\text{Var}(\boldsymbol{b}) = \sigma_g^2 \boldsymbol{I}_J$. This model is referred to as **M10**. From this table, we can observe that model **M10** shows a slightly better performance in terms of both MSE and PC criteria than the **M1** model: the MSE of model **M1** is 3.15% greater than the MSE of **M10**, while the PC of the **M10** is 3.94% greater than the corresponding **M1** model. The better average performance was observed with model **M10**, which did not consider genomic information, suggests that the marker information in this particular case did not provide useful information; however, in general, this is not expected when using

marker information for prediction, although this could change with larger data sets (more lines and more markers) or by improving the quality of the available data.

The R code to reproduce this result is given in Appendix 4. This can be adapted easily to another CV strategy of interest where the objective, for example, can be the prediction of non-observed lines in some environments or the prediction of lines in a future year.

An extension of the GBLUP model is the G×E BLUP model that takes into account the main environmental effects, the genotypic effects, and the genotype ×environment interaction effects:

$$Y = \mathbf{1}_n\mu + \boldsymbol{X}_E\boldsymbol{\beta}_E + \boldsymbol{Z}_L\boldsymbol{b}_1 + \boldsymbol{Z}_{EL}\boldsymbol{b}_2 + \boldsymbol{\epsilon} \tag{5.4}$$

where now the fixed effects are part of the linear mixed model (5.1) that was explicitly split into the general mean part ($\mathbf{1}_n\mu$) and the environment effects term ($\boldsymbol{X}_E\boldsymbol{\beta}_E$), $\boldsymbol{X} = [\mathbf{1}_n\,\boldsymbol{X}_E]$ and $\boldsymbol{\beta} = \left(\mu, \boldsymbol{\beta}_E^{\mathrm{T}}\right)^{\mathrm{T}}$. Similarly, for the random effects, $\boldsymbol{Z} = [\boldsymbol{Z}_L\,\boldsymbol{Z}_{EL}]$ and $\boldsymbol{b} = \left[\boldsymbol{b}_1^{\mathrm{T}}, \boldsymbol{b}_2^{\mathrm{T}}\right]^{\mathrm{T}}$, where $\boldsymbol{b}_1$ and $\boldsymbol{b}_2$ were the vectors with the random genotypic effects and the vector with the random genotype×environment interaction effects, with incidence matrix $\boldsymbol{Z}_L$ and $\boldsymbol{Z}_{EL}$, respectively. For $\boldsymbol{b}_1$, the same distribution as the GBLUP model was assumed, $\boldsymbol{b}_1 \sim N_J\left(\mathbf{0}, \sigma_g^2\boldsymbol{G}\right)$, and for the second random effect, $\boldsymbol{b}_2 \sim N_J(\mathbf{0}, \boldsymbol{\Sigma}_E \otimes \boldsymbol{G})$, where $\boldsymbol{\Sigma}_E \otimes \boldsymbol{G}$ is the relationship matrix of the genotype×environment interaction term, with $\boldsymbol{\Sigma}_E$ the genetic variance–covariance matrix between $I$ environments; the $i$th element of the diagonal of $\boldsymbol{\Sigma}_E$, $\sigma_{Ei}^2$, is the genetic variance in environment $i$, $i = 1, \ldots, I$, and $\sigma_{Eik}\boldsymbol{G}$ is the genetic variance–covariance matrix for lines in environments $i$ and $k$, where $\sigma_{Eik}$ is the element $(i, k)$ of $\boldsymbol{\Sigma}_E$.

When $\boldsymbol{\Sigma}_E$ has a non-diagonal structure, the information from the genomic relationship matrix and the correlated environments can be helpful for improving the prediction performance of the model by borrowing information between lines inside an environment and between lines across and among environments (Burgueño et al. 2012).

**Example 2** To illustrate how model (5.4) can be implemented using the sommer package, the same data used in Example 1 are considered, where the same 30 genotypes are in the four environments. Besides the line indicator (GID), environment information (Env) was also available in the data set, which was needed for implementing model (5.4). The adopted structure for the variance–covariance matrix between environments is $\boldsymbol{\Sigma}_E = \sigma_{EG}^2\boldsymbol{I}_I$ and the resulting model is referred to as M2. Another explored model (M20) was obtained under the same specification, with the difference that $\boldsymbol{G}$ was set equal to the identity matrix.

Using the same validation scheme that was used in Example 1, the results for each of the 10 random partitions are shown in Table 5.2, in which, for illustrative purposes, model (5.3) plus environment as a fixed effect (M11) is also included, that is,

**Table 5.2** Prediction performance of two sub-models of (5.4): model M2 in which $\text{Var}(\boldsymbol{b}_1) = \sigma_g^2\boldsymbol{G}$, $\boldsymbol{b}_2 \sim N_J(\boldsymbol{0}, \boldsymbol{\Sigma}_E \otimes \boldsymbol{G})$ and $\boldsymbol{\Sigma}_E = \sigma_{EG}^2\boldsymbol{I}_I$ (M2); and model M20 that is the same as model M2 but the genomic information is not taken into account, that is, $\boldsymbol{G} = \boldsymbol{I}_J$

| | M2 | | M20 | | M11 | |
|---|---|---|---|---|---|---|
| PT | MSE | PC | MSE | PC | MSE | PC |
| 1 | 0.5641 | 0.8198 | 0.5364 | 0.8576 | 0.5658 | 0.8168 |
| 2 | 0.4292 | 0.7645 | 0.4014 | 0.8393 | 0.4447 | 0.6424 |
| 3 | 0.3865 | 0.7873 | 0.3621 | 0.8497 | 0.4246 | 0.6422 |
| 4 | 0.45 | 0.6997 | 0.401 | 0.7897 | 0.3853 | 0.6992 |
| 5 | 0.6879 | 0.5307 | 0.6636 | 0.5902 | 0.595 | 0.5933 |
| 6 | 0.3048 | 0.6802 | 0.2819 | 0.7326 | 0.3267 | 0.5777 |
| 7 | 0.6405 | 0.6823 | 0.6369 | 0.7066 | 0.558 | 0.6914 |
| 8 | 0.4132 | 0.5261 | 0.4053 | 0.5562 | 0.2999 | 0.5817 |
| 9 | 0.3217 | 0.8384 | 0.2978 | 0.8869 | 0.3457 | 0.7797 |
| 10 | 0.5397 | 0.8368 | 0.5206 | 0.8572 | 0.5647 | 0.7307 |
| Average (SD) | 0.4738 (0.13) | 0.7166 (0.116) | 0.4507 (0.133) | 0.7666 (0.117) | 0.451 (0.112) | 0.6755 (0.083) |

M11 is referred to as model (5.3) plus environment effects (Env). The mean squared error of prediction (MSE) and Pearson's correlation (PC) for each partition are reported. SD is the standard deviation

$$\boldsymbol{Y} = \boldsymbol{1}_n\mu + \boldsymbol{X}_E\boldsymbol{\beta}_E + \boldsymbol{Z}_L\boldsymbol{b} + \boldsymbol{\epsilon},$$

where $\mu$, $\boldsymbol{\beta}_E$, and $\boldsymbol{b}$ are as before (5.3), and $\boldsymbol{X}_E\boldsymbol{\beta}_E$ is the predictor term corresponding to the environment fixed effects.

From Table 5.2 we can observe yet again a moderately better performance of model M20 that does not take into account the genomic information. Model M2 also confirms the lack of usefulness of the marker information in this case, but again, this in general is expected to change for other data sets with a greater number of lines, markers, or more data quality. The MSE of model M2 is 5.11% greater than the MSE of model M20, while the PC value of this last model is 6.98% greater than the corresponding PC value obtained with the M20 model. When comparing the M20 model and M11, the MSE of this last model is just 0.075% greater than the corresponding MSE of M20, but when considering the PC value, the first model resulted in 13.98% greater than model (5.1) plus the environment effect. Indeed, because of the high variation observed across partitions (SD values of PC and MSE), there is no significant difference between models in Table 5.2.

Furthermore, in terms of the average MSE, the model with the best performance between those presented in Table 5.1 (M10) is 13.73% greater than the average MSE of the best performance model between those compared in Table 5.2 (M11), while in terms of the average Pearson's correlation, the best model in Table 5.2 (M20) is 25.42% greater than the average Pearson's correlation of the best model in Table 5.1. The worse average MSE performance of those in Table 5.1 (M1) is 17.31% greater than the best average MSE performance in Table 5.2 (M20), and the best average PC

performance in Table 5.2 (M20) is 30.37% greater than the worse average PC performance in Table 5.1 (M1). Actually, the best average MSE model in Table 5.1 (M10) is 8.19 greater than the worse average MSE model in Table 5.2 (M2), while the worse average PC model in Table 5.2 (M11) is 10.51% greater than the average PC model in Table 5.1 (M10).

The R code to reproduce these results is given in Appendix 5.

Other versions of model (5.4) can be obtained by adopting other variance–covariance structures. For example, another version of model (5.4) can be obtained when environment covariates are available ($\boldsymbol{W}$) and they are used to model the $G\times E$ predictor term, specifically when the genetic variance–covariance matrix between environments, $\boldsymbol{\Sigma}_E$, is modeled by $\boldsymbol{\Sigma}_E = \sigma^2_{EG}\boldsymbol{O}$, where $\boldsymbol{O} = \frac{1}{p_w}\boldsymbol{W}\boldsymbol{W}^{\mathrm{T}}$ and the similarity between environments is computed like the genomic relationship matrix (**G**), using the information of $p_w$ environment covariates (Jarquín et al. 2014; Martini et al. 2020), or where $\boldsymbol{O}$ is obtained from phenotypic correlations across environments from related historical data (Martini et al. 2020). In the sommer package, this can be implemented using the following basic R code:

```
O = diag(I) #Specified the O matrix for the I environments
dat_F$Env_GID = paste(dat_F$Env,dat_F$GID,sep='_')
GE = kronecker(O,G)
rnGWE = expand.grid(row.names(G),unique(dat_F$Env))
row.names(GE) = paste(rnGWE[,2],rnGWE[,1],sep='_')
colnames(GE) = row.names(GE)
A = mmer(y ~ Env, random= ~ vs(GID,Gu=G) + vs(Env_GID,Gu = GE),
      rcov= ~ vs(units), data=dat_F)
```

Other more complex models can be explored with the sommer package when more data information is available, such as specifying an unstructured variance–covariance matrix for $\boldsymbol{\Sigma}_E$. A simpler model is the non-correlated heterogeneous variance components (for environments) which arises by assuming a diagonal structure, $\boldsymbol{\Sigma}_E = \mathrm{Diag}\left(\sigma^2_1, \ldots, \sigma^2_I\right)$. This can be implemented by replacing the interaction term in the predictor in sommer *vs(**Env**,Gu = **G**)* by *vs(ds(**Env**), **GID**,Gu = **G**)*. Similarly, for a specific environment residual variance, *vs(units)* need to be replaced by *vs(ds(Env),units)* or *vs(at(Env),units)*. See Appendix 7 for a basic code to implement all these models and see Covarrubias-Pazaran (2018) for more variance structures that can be exploited in this model.

## 5.5 Multi-trait Genomic Linear Mixed-Effects Models

In some genomic applications, there are several traits of interest and all of them are measured in some lines but in other lines only subsets of those traits are measured. Although separate univariate genomic linear mixed models can be performed to analyze all measured traits, sometimes single univariate genomic models do not

work well, especially in traits with low heritability. When low heritability traits have at least moderate correlation with high heritability traits, the prediction performance ability for these low heritability traits could strongly increase by using a multi-trait model (Jia and Jannink 2012; Montesinos-López et al. 2016; Budhlakoti et al. 2019).

If for each line ($j = 1, \ldots J$), $n_T$ traits are measured, $Y_{jt}$, $t = 1, \ldots n_T$, the multi-trait genomic linear mixed-effects model adopts an unstructured covariance matrix for the residuals between traits and for the random genotypic effects between traits, and similar to the univariate trait models (5.3), this can be expressed as

$$\begin{bmatrix} Y_{j1} \\ Y_{j2} \\ \vdots \\ Y_{jn_T} \end{bmatrix} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{n_T} \end{bmatrix} + \begin{bmatrix} g_{j1} \\ g_{j2} \\ \vdots \\ g_{jn_T} \end{bmatrix} + \begin{bmatrix} \epsilon_{j1} \\ \epsilon_{j2} \\ \vdots \\ \epsilon_{jn_T} \end{bmatrix}, j = 1, \ldots, J, \tag{5.5}$$

where $\mu_t$, $t = 1, \ldots, n_T$, are the specific trait means, $g_{jt}$, $t = 1, \ldots, n_T$, are the specific trait genotypic effects, and $\epsilon_{jt}$, $t = 1, \ldots, n_T$, are the random error terms corresponding to each trait. Furthermore, $\boldsymbol{b} = \left[\boldsymbol{g}_1^{\mathrm{T}}, \ldots, \boldsymbol{g}_J^{\mathrm{T}}\right]^{\mathrm{T}} \sim N(\boldsymbol{0}, \boldsymbol{G} \otimes \boldsymbol{\Sigma}_{\mathrm{T}})$, $\boldsymbol{g}_j = \left[g_{j1}, \ldots, g_{jn_T}\right]^{\mathrm{T}}$, $j = 1, \ldots, J$, and $\boldsymbol{\epsilon}_j = \left[\epsilon_{j1}, \ldots, \epsilon_{jn_T}\right]^{\mathrm{T}}$, $j = 1, \ldots, J$, are independent multivariate normal random vectors with null mean and variance $\mathbf{R}_{n_T}$, $\boldsymbol{\Sigma}_T$ is $n_T \times n_T$ matrix that represents the genetic covariance between traits, and $\otimes$ is the Kronecker product.

In matrix notation, it is the linear mixed model (5.1) where $\boldsymbol{Y} = \left[\boldsymbol{Y}_1^{\mathrm{T}}, \ldots, \boldsymbol{Y}_J^{\mathrm{T}}\right]^{\mathrm{T}}$, $\boldsymbol{Y}_j = \left[Y_{j1}, \ldots, Y_{jn_T}\right]^{\mathrm{T}}$, $\boldsymbol{X} = \boldsymbol{1}_J \otimes \boldsymbol{I}_{\mathrm{n_T}}$, $\boldsymbol{\beta} = \boldsymbol{\mu} = \left(\mu_1, \ldots, \mu_{n_T}\right)^{\mathrm{T}}$, $\boldsymbol{Z} = \boldsymbol{I}_{n_T J}$, $\boldsymbol{b} = \left[\boldsymbol{g}_1^{\mathrm{T}}, \ldots, \boldsymbol{g}_J^{\mathrm{T}}\right]^{\mathrm{T}}, \boldsymbol{\epsilon} = \left[\boldsymbol{\epsilon}_1^{\mathrm{T}} \ldots \boldsymbol{\epsilon}_J^{\mathrm{T}}\right]^{\mathrm{T}} \sim N(\boldsymbol{0}, \boldsymbol{I}_J \otimes \boldsymbol{R}_{\mathrm{n}_T})$, and $\boldsymbol{b} = \left[\boldsymbol{g}_1^{\mathrm{T}}, \ldots, \boldsymbol{g}_J^{\mathrm{T}}\right]^{\mathrm{T}} \sim N(\boldsymbol{0}, \boldsymbol{G} \otimes \boldsymbol{\Sigma}_{\mathrm{T}})$. Similarly, the extended model that arises by adding more fixed effects ($\boldsymbol{X}$) can be specified by adding a term $\boldsymbol{X\beta}$ to the predictor:

$$\boldsymbol{Y} = (\boldsymbol{1}_{IJ} \otimes \boldsymbol{I}_{\mathrm{n_T}})\boldsymbol{\mu} + \boldsymbol{X\beta} + \boldsymbol{Zb} + \boldsymbol{\epsilon} \tag{5.5a}$$

When $\boldsymbol{\Sigma}_T$ and $\boldsymbol{R}$ are diagonal matrices, model (5.5) is equivalent to separately fitting a univariate GBLUP model to each trait.

The R code to fit this multivariate model with the sommer package is

```
A = mmer(cbind(T1,...,TnT) ~x1+x2+...+xp ,
         random= ~ vs(GID,Gu=G), rcov= ~ vs(units), data=dat_F)
```

where **y** and **GID** are again the column names corresponding to the response variables and genotypes in data set **dat_F**, while *T1*, …, *TnT* are the column names of the matrix of response variables (**y**) in **dat_F** corresponding to the traits to be used, and similarly, *x1*, *x2*,…, *xp* are the column names of *p* covariates to be included in the fitting process (see below the R code for Example 3). The rest of the arguments are the same as those described in the R code of model 5.3.

**Example 3** To illustrate the fitting of the multi-trait genomic model (5.5) (M3), we considered the same data set used in Examples 1 and 2, but with the addition of trait (y2) to be able to explore the implementation of a bivariate trait genomic model. The same CV strategy implemented in Example 1 was used. In addition to model (5.5), we also evaluated a sub-model that was obtained by considering a diagonal structure for $\boldsymbol{\Sigma}_T$, that is, $\boldsymbol{\Sigma}_T = \text{Diag}\left(\sigma^2_{T_1}, \sigma^2_{T_2}\right)$. This model will be referred to as M32.

The results are in Table 5.3. On average, the two evaluated models (M3 and M32) showed a similar performance in terms of the two criteria used, MSE and PC, for both traits, but in all partitions a slightly better performance was observed in favor of model M3. For trait T1, the simpler model (M32) gave an MSE 0.785% greater than model M3, while the more complex model (M3) gave a PC only 1.066% greater than that of model M32. The difference was less for the second trait (T2), where the average MSE of M32 was only 0.165% greater than the one corresponding to model M3, while the PC of M3 was only 0.046% greater than the PC of M32.

Furthermore, note that the difference between the univariate models presented in Tables 5.1 and 5.2 and the multivariate models of Table 5.3 is not significant (only on average the models in Table 5.2 result better than models in Table 5.1) because the large standard deviation observed across partitions in MSE and PC, which in this case indicate that the multivariate model does not help improve the prediction accuracy in the trait of interest (first trait). But as commented before, this benefit could be obtained with more related auxiliary secondary traits and larger data sets of good quality.

**Table 5.3** Prediction performance of a bivariate trait model (5.5)

| Models | M3 | | | | M32 | | | |
|---|---|---|---|---|---|---|---|---|
| Trait | T1 | | T2 | | T1 | | T2 | |
| PT | MSE | PC | MSE | PC | MSE | PC | MSE | PC |
| 1 | 0.6959 | 0.6954 | 0.0978 | 0.984 | 0.6973 | 0.6939 | 0.0982 | 0.9835 |
| 2 | 0.4476 | 0.6851 | 0.1411 | 0.9256 | 0.4526 | 0.6759 | 0.1409 | 0.9253 |
| 3 | 0.494 | 0.5448 | 0.0446 | 0.9846 | 0.503 | 0.5315 | 0.0436 | 0.985 |
| 4 | 0.4636 | 0.6125 | 0.1258 | 0.9665 | 0.4674 | 0.6079 | 0.1266 | 0.9663 |
| 5 | 0.6398 | 0.5496 | 0.1588 | 0.9306 | 0.6413 | 0.5489 | 0.1596 | 0.93 |
| 6 | 0.3598 | 0.5192 | 0.0691 | 0.9775 | 0.3632 | 0.5134 | 0.0693 | 0.9772 |
| 7 | 0.6864 | 0.514 | 0.0273 | 0.9953 | 0.691 | 0.5094 | 0.0272 | 0.9951 |
| 8 | 0.3104 | 0.5421 | 0.2011 | 0.9492 | 0.3187 | 0.5267 | 0.2 | 0.9482 |
| 9 | 0.4394 | 0.678 | 0.1983 | 0.9468 | 0.4359 | 0.6833 | 0.2005 | 0.9452 |
| 10 | 0.6685 | 0.6575 | 0.0838 | 0.974 | 0.6759 | 0.644 | 0.0837 | 0.9738 |
| Average (SD) | 0.5205 (0.142) | 0.5998 (0.074) | 0.1148 (0.061) | 0.9634 (0.024) | 0.5246 (0.141) | 0.5935 (0.076) | 0.115 (0.061) | 0.963 (0.024) |

This model is referred to as M3 and when assuming a diagonal structure for $\boldsymbol{\Sigma}_T$, $\boldsymbol{\Sigma}_T = \text{Diag}\left(\sigma^2_{1T}, \sigma^2_{2T}\right)$, it is referred to as M32. The mean squared error of prediction (MSE) and Pearson's correlation (PC) for each trait in each partition are reported. SD is the standard deviation

The R code to obtain the results given in Table 5.3 is provided in Appendix 6. At the end of this Appendix, in the comment lines, the code is also available for a CV strategy, when we are interested in evaluating the performance of a bivariate model where only trait y2 is missing in testing data set and all the information of the other trait (y1) is available. This could be useful in real applications where the interest lies in predicting traits that are difficult or expensive to measure, with the phenotypic information of correlated traits that are easy or inexpensive to measure (Calus and Veerkamp 2011; Jiang et al. 2015). Of course, the code could be adapted for any other relevant strategy.

In a similar fashion, just as univariate genomic linear mixed model (5.4), model (5.5) can be directly extended to a model that considers the genotype ×environment interaction term. Next, we do this for the balanced case, and for this we assume that for each environment $i = 1, \ldots, I$, $J$ lines were phenotyped for $n_T$ traits, $Y_{ijt}$, $t = 1, \ldots, n_T$. In matrix notation, the extended $G\times E$ model (5.4) plus fixed effects ($\boldsymbol{X\beta}$) is given by

$$\boldsymbol{Y} = (\mathbf{1}_{IJ}\otimes \boldsymbol{I}_{\mathrm{n_T}})\boldsymbol{\mu} + \boldsymbol{X\beta} + \boldsymbol{Z}_L\boldsymbol{b}_1 + \boldsymbol{Z}_{EL}\boldsymbol{b}_2 + \boldsymbol{\epsilon}, \tag{5.6}$$

where $\boldsymbol{Y} = \left[\boldsymbol{Y}_1^{\mathrm{T}}\ldots\boldsymbol{Y}_I^{\mathrm{T}}\right]^{\mathrm{T}}$, $\boldsymbol{Y}_i = \left[\boldsymbol{Y}_{i1}^{\mathrm{T}},\ldots,\boldsymbol{Y}_{iJ}^{\mathrm{T}}\right]^{\mathrm{T}}$, $\boldsymbol{Y}_{ij} = \left[\boldsymbol{Y}_{ij1},\ldots\boldsymbol{Y}_{ijn_T}\right]^{\mathrm{T}}$, $i = 1, \ldots, I$, $j = 1, \ldots, J$, $\mathbf{1}_{IJ}$ is the vector of ones of order $IJ$, $\boldsymbol{I}_{n_T}$ is the identity matrix of dimension $n_T$, $\boldsymbol{\mu} = \left(\mu_1,\ldots,\mu_{n_T}\right)^{\mathrm{T}}$ is the vector with the general specific trait means, $\boldsymbol{Z}_L = \mathbf{1}_I\otimes \boldsymbol{I}_{\mathrm{n_T J}}$ and $\boldsymbol{Z}_{EL} = \boldsymbol{I}_{IJn_T}$ are the incidence matrices of genotype random effects ($\boldsymbol{b}_1$) and the incidence matrices of the genotype×environment interactions random effects ($\boldsymbol{b}_2$), respectively, with $\boldsymbol{b}_1 = \left[\boldsymbol{g}_1^{\mathrm{T}},\ldots,\boldsymbol{g}_J^{\mathrm{T}}\right]^{\mathrm{T}}$ and $\boldsymbol{b_2} = \left[\boldsymbol{g}_{21}^{\mathrm{T}},\ldots,\boldsymbol{g}_{2I}^{\mathrm{T}}\right]^{\mathrm{T}}$, $\boldsymbol{g}_j = \left[g_{j1},\ldots,g_{jn_T}\right]^{\mathrm{T}}$, $\boldsymbol{g}_{2i} = \left[\boldsymbol{g}_{2i1}^{\mathrm{T}},\ldots,\boldsymbol{g}_{2iJ}^{\mathrm{T}}\right]^{\mathrm{T}}$, and $\boldsymbol{g}_{2ij} = \left[\boldsymbol{g}_{2ij1},\ldots\boldsymbol{g}_{2ijn_T}\right]^{\mathrm{T}}$, $i = 1, \ldots, I$, $j = 1, \ldots, J$. In addition, it is assumed that $\boldsymbol{\epsilon} = \left[\boldsymbol{\epsilon}_1^{\mathrm{T}}\ldots\boldsymbol{\epsilon}_I^{\mathrm{T}}\right]^{\mathrm{T}} \sim N(\mathbf{0}, \boldsymbol{I}_{IJ}\otimes \boldsymbol{R}_{\mathrm{n}_T})$, $\boldsymbol{b}_1 \sim N(\mathbf{0}, \boldsymbol{G}\otimes \boldsymbol{\Sigma}_T)$, and $\boldsymbol{b}_2 \sim N(\mathbf{0}, \boldsymbol{\Sigma}_E\otimes \boldsymbol{G}\otimes \boldsymbol{\Sigma}_{2T})$.

This shows that when $\boldsymbol{\Sigma}_T$, $\boldsymbol{\Sigma}_{2T}$, $\boldsymbol{\Sigma}_E$, and $\boldsymbol{R}$ are diagonal matrices, model (5.6) is equivalent to separately fitting a univariate GBLUP model for each trait.

**Example 4** To illustrate the fitting and evaluation process of model (5.6), we considered a data set that contains the information of two traits, for which 150 lines were phenotyped each in two environments, and given a total of 300 bivariate phenotypic data points. Also, a genomic relationship matrix for the lines is available that was computed with marker information.

The first explored model is referred to as M4 and assumes an unstructured variance–covariance matrix for all the components in model (5.6), except for the assumption that $\boldsymbol{\Sigma}_E = \boldsymbol{I}_I$, i.e., the model assumes the same variance–covariance among environments. In addition to this model (M4), three sub-models were also explored: M42, which considers a diagonal structure for the genetic variance–covariance between traits, $\boldsymbol{\Sigma}_{2T} = \mathrm{Diag}\left(\sigma_{1T_1}^2, \sigma_{1T_2}^2\right)$, model M43 in which $\boldsymbol{\Sigma}_{1T} = \mathrm{Diag}\left(\sigma_{1T_1}^2, \sigma_{1T_2}^2\right)$ and $\boldsymbol{\Sigma}_{2T} = \mathrm{Diag}\left(\sigma_{2T_1}^2, \sigma_{2T_2}^2\right)$, and model M44 which is the same as

M43 but with an assumed diagonal variance–covariance matrix structure for the error, $\boldsymbol{R} = \mathrm{Diag}\left(\sigma_{e_1}^2, \sigma_{e_2}^2\right)$.

The results are shown in Table 5.4, from which we can observe that for trait 1 (GY), the best performance under both criteria (MSE and PC) was obtained with the more complex model: M4. For this trait (GY), the MSE of models M42, M43, and M44 were 7.53%, 8.21%, and 8.17%, respectively, greater than the MSE of

**Table 5.4** Prediction performance of some sub-models of model (5.6)

| Model | M4 | | | | M42 | | | |
|---|---|---|---|---|---|---|---|---|
| Trait | T1 (GY) | | T2 (Testwt) | | T1 (GY) | | T2 (Testwt) | |
| PT | MSE | PC | MSE | PC | MSE | PC | MSE | PC |
| 1 | 0.1895 | 0.5962 | 1.0291 | 0.6416 | 0.1717 | 0.5691 | 0.9357 | 0.6942 |
| 2 | 0.2414 | 0.5091 | 0.8139 | 0.6466 | 0.2444 | 0.5127 | 0.7199 | 0.6965 |
| 3 | 0.2638 | 0.8089 | 1.2914 | 0.5875 | 0.2324 | 0.5661 | 1.0839 | 0.7705 |
| 4 | 0.475 | 0.1266 | 0.9975 | 0.7118 | 0.4856 | 0.101 | 1.0271 | 0.6846 |
| 5 | 0.2728 | 0.2237 | 0.8053 | 0.6897 | 0.272 | 0.2402 | 0.8 | 0.683 |
| 6 | 0.2919 | 0.7391 | 1.031 | 0.5288 | 0.29 | 0.373 | 0.6211 | 0.7809 |
| 7 | 0.2413 | 0.4145 | 1.0778 | 0.7836 | 0.3155 | 0.2617 | 1.1276 | 0.7808 |
| 8 | 0.1696 | 0.7954 | 1.0456 | 0.538 | 0.1785 | 0.4238 | 1.1182 | 0.726 |
| 9 | 0.2412 | 0.8813 | 1.0937 | 0.4014 | 0.3293 | 0.3998 | 1.9201 | 0.6492 |
| 10 | 0.2078 | 0.9006 | 1.4155 | 0.3746 | 0.2704 | 0.6108 | 1.7571 | 0.6505 |
| Average (SD) | 0.2594 (0.085) | 0.5995 (0.275) | 1.0601 (0.186) | 0.5904 (0.132) | 0.279 (0.089) | 0.4058 (0.166) | 1.1111 (0.422) | 0.7116 (0.051) |
| Model | M43 | | | | M44 | | | |
| Trait | T1 (GY) | | T2 (Testwt) | | T1 (GY) | | T2 (Testwt) | |
| PT | MSE | PC | MSE | PC | MSE | PC | MSE | PC |
| 1 | 0.1717 | 0.5602 | 0.9354 | 0.7015 | 0.1716 | 0.5438 | 0.9623 | 0.7053 |
| 2 | 0.25 | 0.4885 | 0.7258 | 0.6987 | 0.2515 | 0.4383 | 0.7536 | 0.6907 |
| 3 | 0.2344 | 0.5482 | 1.1496 | 0.7698 | 0.2324 | 0.5182 | 1.2674 | 0.7631 |
| 4 | 0.4956 | 0.0226 | 1.0861 | 0.6277 | 0.4852 | −0.055 | 1.1752 | 0.5615 |
| 5 | 0.274 | 0.2154 | 0.806 | 0.6778 | 0.2738 | 0.1416 | 0.8357 | 0.66 |
| 6 | 0.2895 | 0.3758 | 0.6211 | 0.7803 | 0.2909 | 0.3468 | 0.6096 | 0.7864 |
| 7 | 0.2925 | 0.319 | 1.0604 | 0.7936 | 0.3001 | 0.2746 | 1.0997 | 0.7856 |
| 8 | 0.1782 | 0.4247 | 1.1177 | 0.7261 | 0.1828 | 0.3739 | 1.1289 | 0.7139 |
| 9 | 0.324 | 0.4071 | 1.9091 | 0.6488 | 0.3306 | 0.3438 | 1.9212 | 0.6499 |
| 10 | 0.2974 | 0.5764 | 1.8732 | 0.6506 | 0.2874 | 0.5483 | 1.9158 | 0.6434 |
| Average (SD) | 0.2807 (0.091) | 0.3938 (0.173) | 1.1284 (0.439) | 0.7075 (0.059) | 0.2806 (0.088) | 0.3474 (0.191) | 1.1669 (0.445) | 0.696 (0.071) |

Model (5.6) with $\boldsymbol{\Sigma}_E = \boldsymbol{I}_I$ is referred to as M4, and if additionally, $\boldsymbol{\Sigma}_{2T} = \mathrm{Diag}\left(\sigma_{1T_1}^2, \sigma_{1T_2}^2\right)$, the model is referred to as M42. Model M42 but with $\boldsymbol{\Sigma}_{1T} = \mathrm{Diag}\left(\sigma_{1T_1}^2, \sigma_{1T_2}^2\right)$ is referred to as M43, and model M44 is the same as M43 but with $\boldsymbol{R} = \mathrm{Diag}\left(\sigma_{e_1}^2, \sigma_{e_2}^2\right)$. The mean squared error of prediction (MSE) and Pearson's correlation (PC) for each trait in each partition are reported. SD is the standard deviation

model M4. For the same trait (GY), the PC of model M4 was 47.73%, 52.24%, and 72.57%, greater than the PC of models M42, M43, and M44, respectively.

For the second trait (Testwt), model M4 also showed the best performance, but only under the MSE criteria: the MSE of models M42, M43, and M44 were 4.81%, 6.44%, and 10.08%, respectively, greater than the MSE corresponding to model M4, also suggesting an increasing degradation pattern in the MSE performance as the model became simpler with fewer parameters to estimate in relation to M4. In terms of PC, the best performance was achieved with model M42, which gave 20.54%, 0.583%, and 2.247% greater performances than models M4, M43, and M44, respectively.

Appendix 7 shows the R code used to reproduce the results in Table 5.4 with the sommer package. At the end of this code, we also included the basic code to explore other variance–covariance structures. Specifically, this is the code used to explore the model with heterogeneous genetic variance–covariance matrix, $\boldsymbol{\Sigma}_{2T}$, across environments, that is, $\boldsymbol{g}_{2i} \sim N(\boldsymbol{0}, \boldsymbol{G} \otimes \boldsymbol{\Sigma}_{2iT})$, $i = 1, \ldots, I$, which are assumed independent across environments.

## 5.6 Final Comments

The multi-trait linear model proposed by Henderson and Quaas (1976) in animal breeding can bring benefits in comparison to single-trait modeling for the improvement of prediction accuracy when incorporating correlated traits, as well as for obtaining an optimal and simplified total merit selection index (Okeke et al. 2017).

When the goal is to predict difficult or expensive traits that are correlated with inexpensive secondary traits, the use of multi-trait models could be helpful in developing better genomic selection strategies. Similarly, improvement of the accuracy of prediction for low-heritability key traits can follow from the use of high-heritability secondary traits (Jia and Jannink 2012; Muranty et al. 2015). Furthermore, this can be combined with the information of traits obtained using the speed breeding methodology to shorten the breeding cycles and accelerate breeding programs (Ghosh et al. 2018; Watson et al. 2019).

While the advantage of the multi-trait model is clearly documented, larger data sets and more computing resources are required, as there are additional parameters that need to be estimated (genetic and error covariances), which may affect the accuracy of genomic prediction. Additionally, convergence problems often arise when implementing complex mixed linear models and especially when small data sets are used.

Although the application of multi-trait models can be easily adapted with regard to genetic correlation, heritability, training population composition, and size of data sets, some other factors need to be carefully considered when using these methods for the improvement of genomic accuracy predictions (Lorenz and Smith 2015; Covarrubias-Pazaran et al. 2018). For example, biased and suboptimal choices between univariate and multi-trait models can result from using auxiliary traits that

are measured on individuals to be tested, but appropriate cross-validations strategies could be helpful in determining the usefulness of combining the multi-trait information with multi-trait models (Runcie and Cheng 2019).

# Appendix 1

$$\begin{aligned} f_{\boldsymbol{b}|\boldsymbol{Y}}(\boldsymbol{b}|\boldsymbol{y}) &\propto f_{\boldsymbol{Y}|\boldsymbol{b}}(\boldsymbol{y}|\boldsymbol{b})f_{\boldsymbol{b}}(\boldsymbol{b}) \\ &\propto \frac{1}{(2\pi\sigma^2)^{-\frac{n}{2}}}\exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X\beta}-\boldsymbol{Zb})^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{X\beta}-\boldsymbol{Zb})-\frac{1}{2}\boldsymbol{b}^{\mathrm{T}}\boldsymbol{D}^{-1}\boldsymbol{b}\right] \\ &\propto \exp\left[-\frac{1}{2}\boldsymbol{b}^{\mathrm{T}}\left(\boldsymbol{D}^{-1}+\sigma^{-2}\boldsymbol{Z}^{\mathrm{T}}\boldsymbol{Z}\right)\boldsymbol{b}-\sigma^{-2}(\boldsymbol{y}-\boldsymbol{X\beta})^{\mathrm{T}}\boldsymbol{Zb}\right] \\ &\propto \exp\left[-\frac{1}{2}\left(\boldsymbol{b}-\widetilde{\boldsymbol{b}}\right)^{\mathrm{T}}\widetilde{\boldsymbol{D}}^{-1}\left(\boldsymbol{b}-\widetilde{\boldsymbol{b}}\right)\right] \end{aligned}$$

$\widetilde{\boldsymbol{D}}=\left(\boldsymbol{D}^{-1}+\sigma^{-2}\boldsymbol{Z}^{\mathrm{T}}\boldsymbol{Z}\right)^{-1}$ and $\widetilde{\boldsymbol{b}}=\sigma^{-2}\widetilde{\boldsymbol{D}}\boldsymbol{Z}^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{X\beta})$. So $\boldsymbol{b}\mid\boldsymbol{Y}=\boldsymbol{y}\sim N_q\left(\widetilde{\boldsymbol{b}},\widetilde{\boldsymbol{D}}\right)$.

# Appendix 2

Because $(\boldsymbol{ABA}^{\mathrm{T}}+\boldsymbol{C})^{-1}=\boldsymbol{C}^{-1}-\boldsymbol{C}^{-1}\boldsymbol{A}(\boldsymbol{A}^{\mathrm{T}}\boldsymbol{C}^{-1}\boldsymbol{A}+\boldsymbol{B}^{-1})^{-1}\boldsymbol{A}^{\mathrm{T}}\boldsymbol{C}^{-1}$ (Johnson and Wichern (2002)), then

$$\widetilde{\boldsymbol{D}}=\left(\boldsymbol{D}+\sigma^{-2}\boldsymbol{Z}^{\mathrm{T}}\boldsymbol{Z}\right)^{-1}=\boldsymbol{D}-\boldsymbol{DZ}^{\mathrm{T}}\left(\boldsymbol{ZDZ}^{\mathrm{T}}+\sigma^2\boldsymbol{I}_n\right)^{-1}\boldsymbol{ZD}$$

and thus

$$\begin{aligned} \mathrm{tr}\left(\widetilde{\boldsymbol{D}}^{(t)}\boldsymbol{D}^{-1}\right) &= \mathrm{tr}\left[\left(\boldsymbol{D}^{(t)}-\boldsymbol{D}^{(t)}\boldsymbol{Z}^{\mathrm{T}}\left(\boldsymbol{ZD}^{(t)}\boldsymbol{Z}^{\mathrm{T}}+\sigma^{2(t)}\boldsymbol{I}_n\right)^{-1}\boldsymbol{ZD}^{(t)}\right)\boldsymbol{D}^{-1}\right] \\ &= \sum_{k=1}^{K}\frac{\sigma_k^{2(t)}}{\sigma_k^2}\left[q_k-\sigma_k^{2(t)}\mathrm{tr}\left(\boldsymbol{A}_k\boldsymbol{Z}_k^T\boldsymbol{V}^{-(t)}\boldsymbol{Z}_k\right)\right], \end{aligned}$$

where $\boldsymbol{V}^{-(t)}=(\boldsymbol{ZD}^{(t)}\boldsymbol{Z}^{\mathrm{T}}+\sigma^{2(t)}\boldsymbol{I}_n)^{-1}$.

# Appendix 3

Because

$$\exp\left[-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{X\beta})^{\mathrm{T}}\boldsymbol{V}^{-1}(\boldsymbol{y}-\boldsymbol{X\beta})\right]=\exp\left\{-\frac{1}{2}\left[\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X\beta}-2\boldsymbol{y}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X\beta}+\boldsymbol{y}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{y}\right]\right\}$$
$$=\exp\left\{-\frac{1}{2}\left(\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}\right)^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\left(\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}\right)+\frac{1}{2}\widetilde{\boldsymbol{\beta}}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\widetilde{\boldsymbol{\beta}}-\frac{1}{2}\boldsymbol{y}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{y}\right\},$$

where $\widetilde{\boldsymbol{\beta}}=\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{y}$,

$$\begin{aligned}f(\boldsymbol{\theta}|\boldsymbol{y})&\propto\int\frac{|\boldsymbol{V}|^{-\frac{1}{2}}}{(2\pi)^{\frac{n}{2}}}\exp\left[-\frac{1}{2}(\boldsymbol{y}-\boldsymbol{X\beta})^{\mathrm{T}}\boldsymbol{V}^{-1}(\boldsymbol{y}-\boldsymbol{X\beta})\right]d\boldsymbol{\beta}\\&\propto|\boldsymbol{V}|^{-\frac{1}{2}}\left|\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\right|^{-1/2}\exp\left\{-\frac{1}{2}\left[\boldsymbol{y}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{y}-\widetilde{\boldsymbol{\beta}}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right]\right\}\\&\propto|\boldsymbol{V}|^{-\frac{1}{2}}\left|\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\right|^{1/2}\exp\left\{-\frac{1}{2}\left[\boldsymbol{y}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{y}-2\widetilde{\boldsymbol{\beta}}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\widetilde{\boldsymbol{\beta}}+\widetilde{\boldsymbol{\beta}}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right]\right\}\\&\propto|\boldsymbol{V}|^{-\frac{1}{2}}\left|\boldsymbol{X}^{\mathrm{T}}\boldsymbol{V}^{-1}\boldsymbol{X}\right|^{-1/2}\exp\left\{-\frac{1}{2}\left(\boldsymbol{y}-\boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right)^{\mathrm{T}}\boldsymbol{V}^{-1}\left(\boldsymbol{y}-\boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right)\right\}\end{aligned}$$

# Appendix 4

R code for Example 1:

```
rm(list=ls())
library(sommer)
load('dat_ls_E1.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
#Marker data
dat_M = dat_ls$dat_M
dim(dat_M)
dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F,5)
#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
XM = scale(XM)
#Genomic relationship matrix derived from markers
dat_M = scale(dat_M)
G = tcrossprod(dat_M)/dim(dat_M)[2]
```

```
dat_F$GID = factor(dat_F$GID,levels=row.names(G))
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]

#10 random partitions
K = 10
n = dim(dat_F)[1]
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

#Example 1
#GBLUP model
Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 dat_F$y_NA = dat_F$y
 dat_F$y_NA[Pos_tst] = NA
 #M1
 A = mmer(y_NA ~ 1,na.method.Y='include', random= ~ vs(GID,Gu=G),
          rcov= ~ vs(units), data=dat_F,verbose=FALSE)
 #BLUPs
 #bv = A$U$`u:GID`$y_NA
 yp = fitted(A)$dataWithFitted$y_NA.fitted
 #Prediction of testing
 yp_ts = yp[Pos_tst]
 #MSEP and Cor
 Tab$MSEP[k] = mean((dat_F$y[Pos_tst]-yp_ts)^2)
 Tab$Cor[k] = cor(dat_F$y[Pos_tst],yp_ts)

 #M10
 A2 = mmer(y_NA ~ 1,na.method.Y='include', random= ~ vs(GID),
           rcov= ~ vs(units), data=dat_F,verbose=FALSE)
 yp2 = fitted(A2)$dataWithFitted$y_NA.fitted
 #Prediction of testing
 yp2_ts = yp2[Pos_tst]
 #MSEP
 Tab$MSEP10[k] = mean((dat_F$y[Pos_tst]-yp2_ts)^2)
 Tab$Cor10[k] = cor(dat_F$y[Pos_tst],yp2_ts)

}
```

## Appendix 5

```
#Example 2
rm(list=ls())
library(sommer)
load('dat_ls_E1.RData',verbose=TRUE)
```

```
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
#Marker data
dat_M  = dat_ls$dat_M
dim(dat_M)
dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F,5)
#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
XM = scale(XM)
#Genomic relationship matrix derived from markers
dat_M = scale(dat_M)
G = tcrossprod(dat_M)/dim(dat_M)[2]

dat_F$GID = factor(dat_F$GID,levels=row.names(G))
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]

#10 random partitions
K = 10
n = dim(dat_F)[1]
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

#Model (5.4)
#Y = mu + Env + GID + GID:Env
Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
dat_F$Env_GID = paste(dat_F$Env,dat_F$GID,sep='_')
GE = kronecker(diag(length(unique(dat_F$Env))),G)
rnGWE = expand.grid(row.names(G),unique(dat_F$Env))
row.names(GE) = paste(rnGWE[,2],rnGWE[,1],sep='_')
colnames(GE) = row.names(GE)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 dat_F$y_NA = dat_F$y
 dat_F$y_NA[Pos_tst] = NA
 #M2
 A = mmer(y_NA ~ Env,na.method.Y='include',
       random= ~ vs(GID,Gu=G) + vs(Env_GID,Gu = GE),
       rcov= ~ vs(units),
       data=dat_F,verbose=FALSE)
 yp = fitted(A)$dataWithFitted$y_NA.fitted
 #Prediction of testing
 yp_ts = yp[Pos_tst]
 #MSEP
 Tab$MSEP[k] = mean((dat_F$y[Pos_tst]-yp_ts)^2)
 Tab$Cor[k] = cor(dat_F$y[Pos_tst],yp_ts)

 #M20
 A20 = mmer(y_NA ~ Env,na.method.Y='include',
```

```
      random= ~ vs(GID)+vs(Env_GID),
      rcov= ~ vs(units),
      data=dat_F,verbose=FALSE)
 yp20 = fitted(A20)$dataWithFitted$y_NA.fitted
 #Prediction of testing
 yp20_ts = yp20[Pos_tst]
 Tab$MSEP20[k] = mean((dat_F$y[Pos_tst]-yp20_ts)^2)
 Tab$Cor20[k] = cor(dat_F$y[Pos_tst],yp20_ts)

 #M11
 A = mmer(y_NA ~ Env,na.method.Y='include',
       random= ~ vs(GID,Gu=G),
       rcov= ~ vs(units),
       data=dat_F,verbose=FALSE)
 yp = fitted(A)$dataWithFitted$y_NA.fitted
 #Prediction of testing
 yp_ts = yp[Pos_tst]
 Tab$MSEP11[k] = mean((dat_F$y[Pos_tst]-yp_ts)^2)
 Tab$Cor11[k] = cor(dat_F$y[Pos_tst],yp_ts)
 #M10a: Model1 (5.4) plus environment effects but with G = I_J
 A = mmer(y_NA ~ Env,na.method.Y='include',
      random= ~ vs(GID),
      rcov= ~ vs(units),
      data=dat_F,verbose=FALSE)
 yp = fitted(A)$dataWithFitted$y_NA.fitted
 #Prediction of testing
 yp_ts = yp[Pos_tst]
 Tab$MSEP10a[k] = mean((dat_F$y[Pos_tst]-yp_ts)^2)
 Tab$Cor10a[k] = cor(dat_F$y[Pos_tst],yp_ts)

}

#Basic code to implement model (5.4) with an unstructured form for
Sigma_E
A = mmer(y~ Env, na.method.Y='include',
         random= ~ vs(GID,Gu=G)+vs(us(Env),GID,Gu=G),
         rcov= ~ vs(units), data=dat_F)
#Basic code to implement model (5.4) with heterogeneous environment
variances
A = mmer(y ~ Env, ,
         random= ~ vs(GID,Gu=G)+
                   vs(ds(Env),GID,Gu=G),#or vs(at(Env),GID,Gu=G)
         rcov= ~ vs(units), data=dat_F)
#Y = mu + Env + GID + GID:ENV
#Y = mu + Env + GID + GID:ENV
#Basic code to implement model (5.4) with heterogeneous environment and
residuals variances
A = mmer(y ~ Env,
     random= ~ vs(GID,Gu=G) + vs(ds(Env),GID,Gu=G),
     rcov= ~ vs(ds(Env),units),
     data=dat_F)
```

# Appendix 6

```
rm(list=ls())
library(sommer)
load('dat_ls_E1.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
#Marker data
dat_M  = dat_ls$dat_M
dim(dat_M)
dat_F =  transform(dat_F, GID = as.character(GID))
head(dat_F,5)
#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
XM =  scale(XM)
dim(XM)
n =  dim(dat_F)[1]
#Genomic relationship matrix derived from markers
dat_M =  scale(dat_M)
G =  tcrossprod(dat_M)/dim(dat_M)[2]

dat_F$GID = factor(dat_F$GID,levels=row.names(G))
dat_F =  dat_F[order(dat_F$Env,dat_F$GID),]

#10 random partitions
K =  10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))


#Example 3
Tab = data.frame(PT = 1:K)
set.seed(1)
for(k in 1:K)
{
 #M3
 Pos_tst = PT[,k]
 dat_F$y_NA = dat_F$y
 dat_F$y_NA[Pos_tst] = NA
 dat_F$y2_NA = dat_F$y2
 dat_F$y2_NA[Pos_tst] = NA
 A =  mmer(cbind(y_NA,y2_NA) ~ 1,na.method.Y='include',
       random= ~ vs(GID,Gu=G) ,
       rcov= ~ vs(units),
       data=dat_F,verbose=FALSE)
 #BLUPs
 b_ls = A$U$`u:GID`
 b_T1 = b_ls$y_NA# Trait 1
 b_T2 = b_ls$y2_NA# Trait 2
```

```
b_mat = cbind(b_T1,b_T2)
Pos = match(dat_F$GID,names(b_ls$y_NA))
#Y "fitted"
yp = A$fitted + b_mat[Pos,]
#Prediction of testing for both traits
yp_ts = yp[Pos_tst,1]# Trait 1
y2p_ts = yp[Pos_tst,2]# Trait 2
#MSEP and Cor
#Trait 1
Tab$MSEP_T1[k] = mean((dat_F$y[Pos_tst]-yp_ts)^2)
Tab$Cor_T1[k] = cor(dat_F$y[Pos_tst],yp_ts)
#Trait 2
Tab$MSEP_T2[k] = mean((dat_F$y2[Pos_tst]-y2p_ts)^2)
Tab$Cor_T2[k] = cor(dat_F$y2[Pos_tst],y2p_ts)
#M32: Sigma_T diagonal
A = mmer(cbind(y_NA,y2_NA) ~ 1,na.method.Y='include',
      random= ~ vs(GID,Gu=G,Gtc=diag(2)),
      rcov= ~ vs(units),
      data=dat_F,verbose=FALSE)
#BLUPs
b_ls = A$U$`u:GID`
b_T1 = b_ls$y_NA# Trait 1
b_T2 = b_ls$y2_NA# Trait 2
b_mat = cbind(b_T1,b_T2)
Pos = match(dat_F$GID,names(b_ls$y_NA))
#Y "fitted"
yp = A$fitted + b_mat[Pos,]
#Prediction of testing
yp_ts = yp[Pos_tst,1]# Trait 1
y2p_ts = yp[Pos_tst,2]# Trait 2
#MSEP and Cor
#Trait 1
Tab$MSEP_T1_32[k] = mean((dat_F$y[Pos_tst]-yp_ts)^2)
Tab$Cor_T1_32[k] = cor(dat_F$y[Pos_tst],yp_ts)
#Trait 2
Tab$MSEP_T2_32[k] = mean((dat_F$y2[Pos_tst]-y2p_ts)^2)
Tab$Cor_T2_32[k] = cor(dat_F$y2[Pos_tst],y2p_ts)

# #M3
# #dat_F$y2_NA = dat_F$y2
# dat_F$y_NA = dat_F$y# Trait T1 is not NA
# A = mmer(cbind(y_NA,y2_NA) ~ 1,na.method.Y='include',
#       random= ~ vs(GID,Gu=G),
#       rcov= ~ vs(units),#tolparinv = 1e-2,
#       data=dat_F,verbose=FALSE)
# #BLUPs
# b_ls = A$U$`u:GID`
# b_T1 = b_ls$y_NA# Trait 1
# b_T2 = b_ls$y2_NA# Trait 2
# b_mat = cbind(b_T1,b_T2)
# Pos = match(dat_F$GID,names(b_ls$y_NA))
# #Y "fitted"
# yp = A$fitted + b_mat[Pos,]
```

```
 # #Prediction of testing
 # yp_ts = yp[Pos_tst,1]# Trait 1
 # y2p_ts = yp[Pos_tst,2]# Trait 2
 # #MSEP
 # Tab$MSEP_T1_31[k] = mean((dat_F$y[Pos_tst]-yp_ts)^2)
 # Tab$Cor_T1_31[k] = cor(dat_F$y[Pos_tst],yp_ts)
 # Tab$MSEP_T2_31[k] = mean((dat_F$y2[Pos_tst]-y2p_ts)^2)
 # Tab$Cor_T2_31[k] = cor(dat_F$y2[Pos_tst],y2p_ts)
}
```

# Appendix 7

```
#Example 4
rm(list=ls())
library(sommer)
load('dat_ls_E4.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
dat_F = transform(dat_F, GID = as.character(GID))
#Genomic relationship matrix derived from markers
G = dat_ls$G
dat_F$GID = factor(dat_F$GID,levels=row.names(G))
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]
#Fitting model M4 with data set
#A = mmer(cbind(GY,TESTWT) ~ Env,na.method.Y='include',
#       random= ~ vs(GID,Gu=G) + vs(Env_GID,Gu=GE) ,
#       rcov= ~ vs(units),
#       data=dat_F,verbose=FALSE)
#Example 4
#10 random partitions
n = dim(dat_F)[1] ;K = 10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))
#Model 5.6
#Y = mu + Env + GID + GID:Env + e
Tab = data.frame(PT = 1:K)
set.seed(1)
dat_F$Env_GID = paste(dat_F$Env,dat_F$GID,sep='_')
GE = kronecker(diag(length(unique(dat_F$Env))),G)
rnGWE = expand.grid(row.names(G),unique(dat_F$Env))
row.names(GE) = paste(rnGWE[,2],rnGWE[,1],sep='_')
colnames(GE) = row.names(GE)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 #M4
 Pos_tst = PT[,k]
 dat_F$GY_NA = dat_F$GY
 dat_F$GY_NA[Pos_tst] = NA
```

```
dat_F$TESTWT_NA = dat_F$TESTWT
dat_F$TESTWT_NA[Pos_tst] = NA
A = mmer(cbind(GY_NA,TESTWT_NA ) ~ Env,na.method.Y='include',
     random= ~ vs(GID,Gu=G) + vs(Env_GID,Gu=GE) ,
     rcov= ~ vs(units),tolparinv = 1,
     data=dat_F,verbose=FALSE)
#BLUPs
b_ls = A$U$`u:GID`
b_T1 = b_ls$GY_NA# Trait 1
b_T2 = b_ls$TESTWT_NA# Trait 2
b_mat = cbind(b_T1,b_T2)
Pos = match(dat_F$GID,names(b_ls$TESTWT))
#Y "fitted"
yp = data.frame(A$fitted + b_mat[Pos,] )
colnames(yp) = names(b_ls)
plot(dat_F$GY,yp$GY_NA);abline(a=0,b=1)
#Prediction of testing of both traits
yp_tst = yp[Pos_tst,]
#plot(dat_F$y2,yp[,2]); abline(a=0,b=1)
#MSEP and Cor
#Trait 1
Tab$MSEPT1[k] = mean((dat_F$GY[Pos_tst]-yp_tst$GY_NA)^2)
Tab$CorT1[k] = cor(dat_F$GY[Pos_tst],yp_tst$TESTWT_NA)
#Trait 2
Tab$MSEPT2[k] = mean((dat_F$TESTWT[Pos_tst]-yp_tst$TESTWT_NA)^2)
Tab$CorT2[k] = cor(dat_F$TESTWT[Pos_tst],yp_tst$TESTWT_NA)

#M42
A42 = mmer(cbind(GY_NA,TESTWT_NA) ~ Env,na.method.Y='include',
      random= ~ vs(GID,Gu=G) + vs(Env_GID,Gu=GE,Gtc=diag(2)) ,
      rcov= ~ vs(units),
      data=dat_F,verbose=FALSE)
#BLUPs
b_ls = A42$U$`u:GID`
b_T1 = b_ls$GY_NA# Trait 1
b_T2 = b_ls$TESTWT_NA# Trait 2
b_mat = cbind(b_T1,b_T2)
Pos = match(dat_F$GID,names(b_ls$TESTWT))
#Y "fitted"
yp = data.frame(A$fitted + b_mat[Pos,] )
colnames(yp) = names(b_ls)
#plot(dat_F$GY,yp$GY_NA);abline(a=0,b=1)
#Prediction of testing of both traits
yp_tst = yp[Pos_tst,]
#MSEP and Cor
#Trait 1
Tab$MSEPT1_42[k] = mean((dat_F$GY[Pos_tst]-yp_tst$GY_NA)^2)
Tab$CorT1_42[k] = cor(dat_F$GY[Pos_tst],yp_tst$TESTWT_NA)
#Trait 2
Tab$MSEPT2_42[k] = mean((dat_F$TESTWT[Pos_tst]-yp_tst$TESTWT_NA)
^2)
Tab$CorT2_42[k] = cor(dat_F$TESTWT[Pos_tst],yp_tst$TESTWT_NA)
```

```
 #M43
 A43 = mmer(cbind(GY_NA,TESTWT_NA) ~ Env,na.method.Y='include',
       random= ~ vs(GID,Gu=G,Gtc=diag(2)) +
            vs(Env_GID,Gu=GE,Gtc=diag(2)) ,
       rcov= ~ vs(units),
       data=dat_F,verbose=FALSE)
 #BLUPs
 b_ls = A43$U$`u:GID`
 b_T1 = b_ls$GY_NA# Trait 1
 b_T2 = b_ls$TESTWT_NA# Trait 2
 b_mat = cbind(b_T1,b_T2)
 Pos = match(dat_F$GID,names(b_ls$TESTWT))
 #Y "fitted"
 yp = data.frame(A$fitted + b_mat[Pos,] )
 colnames(yp) = names(b_ls)
 #plot(dat_F$GY,yp$GY_NA);abline(a=0,b=1)
 #Prediction of testing of both traits
 yp_tst = yp[Pos_tst,]
 #MSEP and Cor
 #Trait 1
 Tab$MSEPT1_43[k] = mean((dat_F$GY[Pos_tst]-yp_tst$GY_NA)^2)
 Tab$CorT1_43[k] = cor(dat_F$GY[Pos_tst],yp_tst$TESTWT_NA)
 #Trait 2
 Tab$MSEPT2_43[k] = mean((dat_F$TESTWT[Pos_tst]-yp_tst$TESTWT_NA)
^2)
 Tab$CorT2_43[k] = cor(dat_F$TESTWT[Pos_tst],yp_tst$TESTWT_NA)
  A44 = mmer(cbind(GY_NA,TESTWT_NA) ~ Env,na.method.Y='include',
       random= ~ vs(GID,Gu=G,Gtc=diag(2)) +
        vs(Env_GID,Gu=GE,Gtc=diag(2)) ,
       rcov= ~ vs(units,Gtc=diag(2)),
       data=dat_F,verbose=FALSE)
 #A44$sigma
 #BLUPs
 b_ls = A44$U$`u:GID`
 b_T1 = b_ls$GY_NA# Trait 1
 b_T2 = b_ls$TESTWT_NA# Trait 2
 b_mat = cbind(b_T1,b_T2)
 Pos = match(dat_F$GID,names(b_ls$TESTWT))
 #Y "fitted"
 yp = data.frame(A$fitted + b_mat[Pos,] )
 colnames(yp) = names(b_ls)
 #plot(dat_F$GY,yp$GY_NA);abline(a=0,b=1)
 #Prediction of testing of both traits
 yp_tst = yp[Pos_tst,]
 #MSEP and Cor
 #Trait 1
 Tab$MSEPT1_44[k] = mean((dat_F$GY[Pos_tst]-yp_tst$GY_NA)^2)
 Tab$CorT1_44[k] = cor(dat_F$GY[Pos_tst],yp_tst$TESTWT_NA)
 #Trait 2
 Tab$MSEPT2_44[k] = mean((dat_F$TESTWT[Pos_tst]-yp_tst$TESTWT_NA)
^2)
 Tab$CorT2_44[k] = cor(dat_F$TESTWT[Pos_tst],yp_tst$TESTWT_NA)
 cat('k=',k,'\n')

}
```

```
#Model 5.6 with Sigma_2T different for each Env
A4 = mmer(cbind(GY,TESTWT)~Env,
      random = ~vs(GID,Gu=G)+vs(ds(Env),GID,Gu=G),data=dat_F,
      rcov= ~ vs(units))
```

## References


Araus JL, Cairns JE (2014) Field high-throughput phenotyping: the new crop breeding frontier. Trends Plant Sci 19(1):52–61

Bates D, Maechler M, Bolker B, Walker S (2015) Fitting linear mixed-effects models using lme4. J Stat Softw 67(1):1–48

Borman S (2004) The expectation maximization algorithm: a short tutorial. https://www.lri.fr/~sebag/COURS/EM_algorithm.pdf

Brown H, Prescott R (2014) Applied mixed models in medicine. John Wiley & Sons, Hoboken, NJ

Budhlakoti N, Mishra DC, Rai A, Lal SB, Chaturvedi KK, Kumar RR (2019) A comparative study of single-trait and multi-trait genomic selection. J Comput Biol 26(10):1100–1112

Burgueño J, de los Campos G, Weigel K, Crossa J (2012) Genomic prediction of breeding values when modeling genotype × environment interaction using pedigree and dense molecular markers. Crop Sci 52(2):707–719

Cabrera-Bosquet L, Crossa J, von Zitzewitz J, Serret MD, Luis Araus J (2012) High-throughput phenotyping and genomic selection: the frontiers of crop breeding converge F. J Integr Plant Biol 54(5):312–320

Calus MP, Veerkamp RF (2011) Accuracy of multi-trait genomic selection using different methods. Genet Select Evol 43(1):26. https://doi.org/10.1186/1297-9686-43-26

Cappa EP, de Lima BM, da Silva-Junior OB, Garcia CC, Mansfield SD, Grattapaglia D (2019) Improving genomic prediction of growth and wood traits in Eucalyptus using phenotypes from non-genotyped trees by single-step GBLUP. Plant Sci 284:9–15

Covarrubias-Pazaran G (2016) Genome-assisted prediction of quantitative traits using the R package sommer. PLoS One 11(6):e0156744

Covarrubias-Pazaran G (2018) Software update: moving the R package sommer to multivariate mixed models for genome-assisted prediction. https://doi.org/10.1101/354639

Covarrubias-Pazaran G, Schlautman B, Diaz-Garcia L, Grygleski E, Polashock J, Johnson-Cicalese J et al (2018) Multivariate GBLUP improves accuracy of genomic selection for yield and fruit weight in biparental populations of Vaccinium macrocarpon Ait. Front Plant Sci 9:1310

Crossa J, Pérez-Rodríguez P, Cuevas J, Montesinos-López O, Jarquín D, de los Campos G et al (2017) Genomic selection in plant breeding: methods, models, and perspectives. Trends Plant Sci 22(11):961–975

Finch WH, Bolin JE, Kelley K (2019) Multilevel modeling using R. CRC Press, Boca Raton, FL

Ghosh S, Watson A, Gonzalez-Navarro OE, Ramirez-Gonzalez RH, Yanes L, Mendoza-Suárez M et al (2018) Speed breeding in growth chambers and glasshouses for crop breeding and model plant research. Nat Protoc 13(12):2944–2963

Goldstein H (2011) Multilevel statistical models. Wiley, Hoboken, NJ

Harville DA (1974) Bayesian inference for variance components using only error contrasts. Biometrika 61(2):383–385

Harville DA (1977) Maximum likelihood approaches to variance component estimation and to related problems. J Am Stat Assoc 72(358):320–338

Harville DA (1985) Decomposition of prediction error. J Am Stat Assoc 80(389):132–138

Henderson CR (1950) Estimation of genetic parameters. Ann Math Stat 21:309–310

Henderson CR (1975) Best linear unbiased estimation and prediction under a selection model. Biometrics 31:423–447

Henderson CR, Quaas RL (1976) Multiple trait evaluation using relatives' records. J Anim Sci 43 (6):1188–1197

Jarquín D, Crossa J, Lacaze X, Du Cheyron P, Daucourt J, Lorgeou J et al (2014) A reaction norm model for genomic selection using high-dimensional genomic and environmental data. Theor Appl Genet 127(3):595–607
Jennrich RI, Sampson PF (1976) Newton-Raphson and related algorithms for maximum likelihood variance component estimation. Technometrics 18(1):11–17
Jennrich RI, Schluchter MD (1986) Unbalanced repeated-measures models with structured covariance matrices. Biometrics 42:805–820
Jia Y, Jannink JL (2012) Multiple-trait genomic selection methods increase genetic value prediction accuracy. Genetics 192(4):1513–1522
Jiang J, Zhang Q, Ma L, Li J, Wang Z, Liu JF (2015) Joint prediction of multiple quantitative traits using a Bayesian multivariate antedependence model. Heredity 115(1):29–36
Johnson RA, Wichern DW (2002) Applied multivariate statistical analysis. Prentice Hall, Upper Saddle River, NJ
Krause MR, González-Pérez L, Crossa J, Pérez-Rodríguez P, Montesinos-López O, Singh RP et al (2019) Hyperspectral reflectance-derived relationship matrices for genomic prediction of grain yield in wheat. G3 9(4):1231–1247
Laird NM, Ware JH (1982) Random-effects models for longitudinal data. Biometrics 38:963–974
Leyland AH, Goldstein H (2001) Multilevel modelling of health statistics. Wiley, Hoboken, NJ
Lindstrom MJ, Bates DM (1988) Newton–Raphson and EM algorithms for linear mixed-effects models for repeated-measures data. J Am Stat Assoc 83(404):1014–1022
Lorenz AJ, Smith KP (2015) Adding genetically distant individuals to training populations reduces genomic prediction accuracy in barley. Crop Sci 55(6):2657–2667
Martini JW, Crossa J, Toledo FH, Cuevas J (2020) On Hadamard and Kronecker products in covariance structures for genotype × environment interaction. Plant Genome 13:e20033
McLean RA, Sanders WL, Stroup WW (1991) A unified approach to mixed linear models. Am Stat 45(1):54–64
Meeker W, Hong Y, Escobar L (2011) Degradation models and analyses. In: Encyclopedia of statistical sciences. Wiley, Hoboken, NJ. https://doi.org/10.1002/0471667196.ess7148
Meuwissen THE, Hayes BJ, Goddard ME (2001) Prediction of total genetic values using genome-wide dense marker maps. Genetics 157:1819–1829
Montesinos-López OA, Montesinos-López A, Crossa J, Toledo FH, Pérez-Hernández O, Eskridge KM, Rutkoski J (2016) A genomic Bayesian multi-trait and multi-environment model. G3 6 (9):2725–2744
Muranty H, Troggio M, Sadok IB, Al Rifaï M, Auwerkerken A, Banchi E et al (2015) Accuracy and responses of genomic selection on key traits in apple breeding. Horticult Res 2(1):1–12
Okeke UG, Akdemir D, Rabbi I, Kulakow P, Jannink JL (2017) Accuracies of univariate and multivariate genomic prediction models in African cassava. Genet Sel Evol 49(1):88
Patterson HD, Thompson R (1971) Recovery of inter-block information when block sizes are unequal. Biometrika 58:545–554
Piepho HP, Möhring J, Melchinger AE, Büchse A (2008) BLUP for phenotypic selection in plant breeding and variety testing. Euphytica 161(1–2):209–228
Pinheiro JC, Bates DM (2000) Mixed-effects models in S and S-PLUS. Springer, New York
Poland J, Endelman J, Dawson J, Rutkoski J, Wu S, Manes Y et al (2012) Genomic selection in wheat breeding using genotyping-by-sequencing. Plant Genome 5(3):103–113
Raudenbush SW, Bryk AS (2002) Hierarchical linear models: applications and data analysis methods. Sage Publications, Inc, Thousand Oaks, CA
Rencher AC (2008) Linear models in statistics. Wiley, Hoboken, NJ
Robinson GK (1991) That BLUP is a good thing: the estimation of random effects. Stat Sci 6 (1):15–32
Runcie D, Cheng H (2019) Pitfalls and remedies for cross validation with multi-trait genomic prediction methods. G3 9(11):3727–3741

Searle SR (1993) Applying the EM algorithm to calculating ML and REML estimates of variance components. In: Paper invited for the 1993 American Statistical Association Meeting, San Francisco

Searle SR, Casella G, McCulloch CE (2006) Variance components. Wiley, Hoboken, NJ

Speelman D, Heylen K, Geeraerts D (eds) (2018) Mixed-effects regression models in linguistics. Springer, New York

Stroup WW (2012) Generalized linear mixed models: modern concepts, methods and applications. CRC Press, Boca Raton, FL

VanRaden PM (2008) Efficient methods to compute genomic predictions. J Dairy Sci 91:4414–4423

Wang X, Xu Y, Hu Z, Xu C (2018) Genomic selection methods for crop improvement: current status and prospects. Crop J 6(4):330–340

Watson A, Hickey LT, Christopher J, Rutkoski J, Poland J, Hayes BJ (2019) Multivariate genomic selection and potential of rapid indirect selection with speed breeding in spring wheat. Crop Sci 59(5):1945–1959

West BT, Welch KB, Galecki AT (2014) Linear mixed models: a practical guide using statistical software. CRC Press, Boca Raton, FL

Zuur A, Ieno EN, Walker N, Saveliev AA, Smith GM (2009) Mixed effects models and extensions in ecology with R. Springer Science & Business Media, New York

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 6
# Bayesian Genomic Linear Regression

## 6.1 Bayes Theorem and Bayesian Linear Regression

Unlike classic statistical inference, which assumes that the parameter $\boldsymbol{\theta}$ that defines the model is a fixed unknown quantity, in Bayesian inference it is considered as a random variable whose variation tries to represent the knowledge or ignorance about this, before the data points are collected (Box and Tiao 1992). The probability density function which describes such variation is known as the prior distribution and is an additional component in the specification of the complete model in a Bayesian framework.

Given a data set $\boldsymbol{y}_n = (y_1, \ldots, y_n)$ whose distribution is assumed to be $f(\boldsymbol{y}|\boldsymbol{\theta})$, and a prior distribution for the parameter $\boldsymbol{\theta}$, $f(\boldsymbol{\theta})$, the Bayesian analysis uses the Bayes theorem to combine these two pieces of information to obtain the posterior distribution of the parameters, on which the inference is fully based (Christensen et al. 2011):

$$f(\boldsymbol{\theta}|\boldsymbol{y}) = \frac{f(\boldsymbol{y},\boldsymbol{\theta})}{f(\boldsymbol{y})} = \frac{f(\boldsymbol{\theta})f(\boldsymbol{y}|\boldsymbol{\theta})}{f(\boldsymbol{y})} \propto f(\boldsymbol{\theta})L(\boldsymbol{\theta};\boldsymbol{y}),$$

where $f(\boldsymbol{y}) = \int f(\boldsymbol{y}|\boldsymbol{\theta})f(\boldsymbol{\theta})d\boldsymbol{\theta} = E_{\boldsymbol{\theta}}[f(\boldsymbol{y}|\boldsymbol{\theta})]$ is the marginal distribution of $\boldsymbol{\theta}$. This conditional distribution describes what is known about $\boldsymbol{\theta}$ after data is collected and can be thought of as the updated prior knowledge about $\boldsymbol{\theta}$ with the information contained in the data, which is done through the likelihood function $L(\boldsymbol{\theta};\boldsymbol{y})$ (Box and Tiao 1992).

In general, because the posterior distribution doesn't always have a recognizable form and it is often not easy to simulate from this, numerical approximation methods are employed. Once a sample of the posterior distribution is obtained, estimation of a parameter is often found by averaging the sample values or averaging a function of the sample values when another quantity is of interest. For example, in genomic prediction with dense molecular markers, the main interest is to predict the trait of

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_6

interest of the non-phenotyped individuals that have only genotypic information, environment variables, or other information (covariates). In this situation, a convenient practice is to include the individuals to be predicted ($\boldsymbol{y}_p$) in the posterior distribution to be sampled.

Specifically, a standard Bayesian framework for a normal linear regression model (see Chap. 3)

$$Y = \beta_0 + \sum_{j=1}^{p} X_j \beta_j + \epsilon \tag{6.1}$$

with $\epsilon$ a random error with normal distribution with mean 0 and variance $\sigma^2$, is fully specified by assuming the next non-informative prior distribution: $\boldsymbol{\beta}$ and $\log(\sigma)$ approximately independent and locally uniform.

$$f\left(\boldsymbol{\beta}, \sigma^2\right) \propto \sigma^{-2} \tag{6.2}$$

which is not a proper distribution because it does not integrate to 1 (Box and Tiao 1992; Gelman et al. 2013). However, when $\boldsymbol{X}$ is of full column rank, the posterior distribution is a proper distribution and is given by

$$\begin{aligned}
f(\boldsymbol{\beta}, \sigma^2 | \boldsymbol{y}, \boldsymbol{X}) &\propto (\sigma^2)^{-\frac{n}{2}} \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{y}-\boldsymbol{X\beta})^{\mathrm{T}}(\boldsymbol{y}-\boldsymbol{X\beta})\right](\sigma^2)^{-1} \\
&\propto (\sigma^2)^{-\frac{n}{2}} \exp\left[-\frac{1}{2\sigma^2}(\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X\beta} - 2\boldsymbol{y}^{\mathrm{T}}\boldsymbol{X\beta} + \boldsymbol{y}^{\mathrm{T}}\boldsymbol{y})\right](\sigma^2)^{-1} \\
&\propto (\sigma^2)^{-\frac{p+1}{2}} \exp\left[-\frac{1}{2\sigma^2}\left(\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}\right)^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\left(\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}\right) - \frac{1}{2\sigma^2}\left(\boldsymbol{y}^{\mathrm{T}}\boldsymbol{y} - \widetilde{\boldsymbol{\beta}}^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\widetilde{\boldsymbol{\beta}}\right)\right](\sigma^2)^{-1-(n-p-1)/2} \\
&\propto (\sigma^2)^{-\frac{p+1}{2}} \exp\left[-\frac{1}{2\sigma^2}\left(\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}\right)^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\left(\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}\right)\right](\sigma^2)^{-1-(n-p-1)/2} \exp\left[-\frac{1}{2\sigma^2}\boldsymbol{y}^{\mathrm{T}}(\boldsymbol{I}-\boldsymbol{H})\boldsymbol{y}\right] \\
&\propto (\sigma^2)^{-\frac{p+1}{2}} \exp\left[-\frac{1}{2\sigma^2}\left(\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}\right)^{\mathrm{T}}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\left(\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}\right)\right](\sigma^2)^{-1-\frac{n-p-1}{2}} \exp\left(-\frac{(n-p-1)\widetilde{\sigma}^2}{2\sigma^2}\right),
\end{aligned}$$

where $\widetilde{\boldsymbol{\beta}} = \left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{y}$, $\widetilde{\sigma}^2 = \frac{1}{n-p-1}\boldsymbol{y}^{\mathrm{T}}(\boldsymbol{I}-\boldsymbol{H})\boldsymbol{y}$, and $\boldsymbol{H} = \boldsymbol{X}(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}$. From here the marginal posterior distribution of $\sigma^2$ is $\sigma^2 \mid \boldsymbol{y}, \boldsymbol{X} \sim IG\left((n-p-1)/2, \frac{(n-p-1)\widetilde{\sigma}^2}{2}\right)$ with mean $\frac{(n-p-1)\widetilde{\sigma}^2}{2}/[(n-p-1)/2] = \widetilde{\sigma}^2$, and given $\sigma^2$, the posterior conditional distribution of $\boldsymbol{\beta}$ is given by $\boldsymbol{\beta} \mid \sigma^2, \boldsymbol{y}, \boldsymbol{X} \sim N\left(\widetilde{\boldsymbol{\beta}}, \sigma^2\left(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)^{-1}\right)$.

## 6.2 Bayesian Genome-Based Ridge Regression

When $p > n$, $\boldsymbol{X}$ is not of full column rank and the posterior of model (6.1) may not be proper (Gelman et al. 2013), so a solution is instead to consider independently proper prior distributions, $\boldsymbol{\beta} \sim N(0, \boldsymbol{I}\sigma^2)$ and $\sigma^2 \sim IG(\alpha_0, \alpha_0)$, which for large values of $\sigma^2$

($10^6$) and small values of $\alpha_0$ ($10^{-3}$) is an approximation to the standard non-informative prior given in (6.1) (Christensen et al. 2011). A similar prior specification is taken in genomic prediction where different models are obtained by adopting different prior distributions of the parameters. For example, the Bayesian Linear Ridge Regression (Pérez and de los Campos 2014) with standardized covariates ($X_j's$) is given by

$$Y = \mu + \sum_{j=1}^{p} X_j\beta_j + \epsilon \tag{6.3}$$

with a flat prior for mean parameter ($\mu$), $f(\mu) \propto 1$, which can be approximately specified by $\mu \sim N(0, \sigma_0^2)$, with a large value of $\sigma_0^2$ ($10^{10}$), a multivariate normal distribution with mean vector $\mathbf{0}$ and covariance matrix $\sigma_\beta^2 \boldsymbol{I}_p$ on the beta coefficients, $\boldsymbol{\beta}_0 = (\beta_1, \ldots, \beta_p)^{\mathrm{T}} \mid \sigma_\beta^2 \sim N_p\left(\mathbf{0}, \boldsymbol{I}_p\sigma_\beta^2\right)$, and scaled inverse Chi-square distributions as priors for the variance component: $\sigma_\beta^2 \sim \chi^{-2}_{v_\beta,\ S_\beta}$ (prior for the variance of the regression coefficients $\beta_j$) and $\sigma^2 \sim \chi^{-2}_{v,S}$ (prior for the variance of random errors, $\epsilon$), where $\chi^{-2}_{v,S}$ denotes the scaled inverse Chi-squared distribution with shape parameter $v$ and scale parameter $S$. The joint posterior distribution of the parameters in this model, $\boldsymbol{\theta} = \left(\mu, \boldsymbol{\beta}_0^{\mathrm{T}}, \sigma^2, \sigma_\beta^2\right)^{\mathrm{T}}$, is given by

$$\begin{aligned}
f\left(\mu, \boldsymbol{\beta}_0^{\mathrm{T}}, \sigma_\beta^2, \sigma^2 | \boldsymbol{y}, \boldsymbol{X}\right) &\propto L\left(\mu, \boldsymbol{\beta}_0^{\mathrm{T}}, \sigma^2; \boldsymbol{y}\right) f(\boldsymbol{\theta}) \\
&\propto L\left(\mu, \boldsymbol{\beta}_0^{\mathrm{T}}, \sigma^2; \boldsymbol{y}\right) f(\mu) f\left(\boldsymbol{\beta}_0 | \sigma_\beta^2\right) f\left(\sigma_\beta^2\right) f\left(\sigma^2\right) \\
&\propto \frac{1}{(2\pi\sigma^2)^{\frac{n}{2}}} \exp\left[-\frac{1}{2\sigma^2}\|\boldsymbol{y} - 1_n\mu - \boldsymbol{X}_1\boldsymbol{\beta}_0\|^2\right] \times \exp\left(-\frac{1}{2\sigma_0^2}\mu^2\right) \\
&\quad \times \frac{1}{\left(\sigma_\beta^2\right)^{\frac{p}{2}}} \exp\left(\left[-\frac{1}{2\sigma_\beta^2}\boldsymbol{\beta}_0^{\mathrm{T}}\boldsymbol{\beta}_0\right]\right) \times \frac{\left(\frac{S_\beta}{2}\right)^{\frac{v_\beta}{2}}}{\Gamma\left(\frac{v_\beta}{2}\right)\left(\sigma_\beta^2\right)^{1+\frac{v_\beta}{2}}} \exp\left(-\frac{S_\beta}{2\sigma_\beta^2}\right) \\
&\quad \times \frac{\left(\frac{S}{2}\right)^{\frac{v}{2}}}{\Gamma\left(\frac{v}{2}\right)\left(\sigma^2\right)^{1+\frac{v}{2}}} \exp\left(-\frac{S}{2\sigma^2}\right).
\end{aligned}$$

This has no known form and it is not easy to simulate values of it, so numerical methods are required to explore it. One way to simulate values of this distribution is by means of the Gibbs sampler method, which consists of alternatingly generating samples of the full conditional distributions of each variable (or block of variables) given the rest of the parameters (Casella and George 1992).

The full conditional posteriors to implement the Gibbs sampler are obtained in the lines below.

The conditional posterior distribution of $\boldsymbol{\beta}_0$ is given by

$$
\begin{aligned}
f(\boldsymbol{\beta}_0|-) &\propto L\left(\mu, \boldsymbol{\beta}_0^{\mathrm{T}}, \sigma^2; \boldsymbol{y}\right) f\left(\boldsymbol{\beta}_0 | \sigma_\beta^2\right) \\
&\propto \exp\left[-\frac{1}{2\sigma^2}\|\boldsymbol{y} - 1_n\mu - \boldsymbol{X}_1\boldsymbol{\beta}_0\|^2 - \frac{1}{2\sigma_\beta^2}\boldsymbol{\beta}_0^{\mathrm{T}}\boldsymbol{\beta}_0\right] \\
&\propto \exp\left\{-\frac{1}{2}\left[\boldsymbol{\beta}_0^{\mathrm{T}}\left(\sigma_\beta^{-2}\boldsymbol{I}_n + \sigma^{-2}\boldsymbol{X}_1^{\mathrm{T}}\boldsymbol{X}_1\right)\boldsymbol{\beta}_0 - 2\sigma^{-2}(\boldsymbol{y} - 1_n\mu)^{\mathrm{T}}\boldsymbol{X}_1\boldsymbol{\beta}_0\right]\right\} \\
&\propto \exp\left\{-\frac{1}{2}\left[\left(\boldsymbol{\beta}_0 - \widetilde{\boldsymbol{\beta}}_0\right)^{\mathrm{T}}\widetilde{\boldsymbol{\Sigma}}_0^{-1}\left(\boldsymbol{\beta}_0 - \widetilde{\boldsymbol{\beta}}_0\right)\right]\right\}
\end{aligned}
$$

$\widetilde{\boldsymbol{\Sigma}}_0 = \left(\sigma_\beta^{-2}\boldsymbol{I}_p + \sigma^{-2}\boldsymbol{X}_1^{\mathrm{T}}\boldsymbol{X}_1\right)^{-1}$ and $\widetilde{\boldsymbol{\beta}}_0 = \sigma^{-2}\widetilde{\boldsymbol{\Sigma}}_0\boldsymbol{X}_1^{\mathrm{T}}(\boldsymbol{y} - \mathbf{1}_n\mu)$. That is, $\boldsymbol{\beta}_0 \mid - \sim \boldsymbol{N}_p\left(\widetilde{\boldsymbol{\beta}}_0, \widetilde{\boldsymbol{\Sigma}}_0\right)$. Similarly, the conditional distribution of $\mu$ is $\mu \mid - \sim N\left(\widetilde{\mu}, \widetilde{\sigma}_0^2\right)$, where $\widetilde{\sigma}_0^2 = \frac{\sigma^2}{n}$ and $\widetilde{\mu} = \frac{1}{n}\mathbf{1}_n^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X}_1\boldsymbol{\beta}_0)$.

The conditional distribution of $\sigma^2$ is

$$
\begin{aligned}
f\left(\sigma^2|-\right) &\propto L\left(\mu, \boldsymbol{\beta}_0^{\mathrm{T}}, \sigma^2; \boldsymbol{y}\right) f\left(\sigma^2\right) \\
&\propto \frac{1}{\left(\sigma^2\right)^{\frac{n}{2}}} \exp\left[-\frac{1}{2\sigma^2}\|\boldsymbol{y} - 1_n\mu - \boldsymbol{X}_1\boldsymbol{\beta}_0\|^2\right] \frac{\left(\frac{S}{2}\right)^{\frac{v}{2}}}{\Gamma\left(\frac{v}{2}\right)\left(\sigma^2\right)^{1+\frac{v}{2}}} \exp\left(-\frac{S}{2\sigma^2}\right) \\
&\propto \frac{\left(\frac{\widetilde{S}}{2}\right)^{\frac{\widetilde{v}}{2}}}{\left(\sigma^2\right)^{1+\frac{\widetilde{v}}{2}}} \exp\left(-\frac{\widetilde{S}}{2\sigma^2}\right),
\end{aligned}
$$

where $\widetilde{v} = v + n$ and $\widetilde{S} = S + \|\boldsymbol{y} - \mathbf{1}_n\mu - \boldsymbol{X}_1\boldsymbol{\beta}_0\|^2$. So $\sigma^2 \mid - \sim \chi^{-2}_{\widetilde{v},\widetilde{S}}$, where $\chi^{-2}_{v,s}$ denotes a scaled inverse Chi-squared distribution with parameters $v$ and $S$. Similarly, $\sigma_\beta^2 \mid - \sim \chi^{-2}_{\widetilde{v}_\beta,\widetilde{S}_\beta}$, where $\widetilde{v}_\beta = v_\beta + p$ and $\widetilde{S}_\beta = S_\beta + \boldsymbol{\beta}_0^{\mathrm{T}}\boldsymbol{\beta}_0$.

In summary, for the Ridge regression model, a Gibbs sampler consists of the following steps:

1. Choose initial values for $\mu$, $\boldsymbol{\beta}_0$, and $\sigma^2$.
2. Simulate a value of the full conditional distribution of $\sigma_\beta^2$:

$$\sigma_\beta^2 \mid \mu, \boldsymbol{\beta}_0, \sigma^2 \sim \chi^{-2}_{\widetilde{v},\widetilde{S}_\beta},$$

where $\chi^{-2}_{\widetilde{v},\widetilde{S}_\beta}$ denotes a scaled inverse Chi-square distribution with shape parameter $\widetilde{v}_\beta = v_\beta + p$ and scale parameter $\widetilde{S}_\beta = S_\beta + \boldsymbol{\beta}_0^{\mathrm{T}}\boldsymbol{\beta}_0$.

3. Simulate the full conditional posterior distribution of $\beta_0$:

$$\boldsymbol{\beta}_0 \mid \mu, \sigma_\beta^2, \sigma^2 \sim \boldsymbol{N}_p\left(\widetilde{\boldsymbol{\beta}}_0, \widetilde{\boldsymbol{\Sigma}}_0\right),$$

where $\widetilde{\boldsymbol{\Sigma}}_0 = \left(\sigma_\beta^{-2}\boldsymbol{I}_p + \sigma^{-2}\boldsymbol{X}_1^{\mathrm{T}}\boldsymbol{X}_1\right)^{-1}$ and $\widetilde{\boldsymbol{\beta}}_0 = \sigma^{-2}\widetilde{\boldsymbol{\Sigma}}_0\boldsymbol{X}_1^{\mathrm{T}}(\boldsymbol{y} - \mathbf{1}_n\mu)$

4. Simulate the full conditional distribution of $\mu$:

$$\mu \mid \boldsymbol{\beta}_0, \sigma_\beta^2, \sigma^2 \sim \boldsymbol{N}\left(\widetilde{\mu}, \widetilde{\sigma}_\mu^2\right),$$

where $\widetilde{\sigma}_\mu^2 = \frac{\sigma^2}{n}$ and $\widetilde{\mu} = \mathbf{1}_n^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X}_1\boldsymbol{\beta}_0)$.

5. Simulate the full conditional distribution of $\sigma^2$:

$$\sigma^2 \mid \mu, \boldsymbol{\beta}_0, \sigma_\beta^2 \sim \chi^{-2}_{\widetilde{v}, \widetilde{S}},$$

where $\widetilde{v} = v + n$ and $\widetilde{S} = S + \|\boldsymbol{y} - \mathbf{1}_n\mu - \boldsymbol{X}_1\boldsymbol{\beta}_0\|^2$.

6. Repeat steps 2–5 depending on how many values of the parameter vector $(\boldsymbol{\beta}^{\mathrm{T}}, \sigma_\beta^2, \sigma^2)$ you wish to simulate. Usually a large number of iterations are needed and an early part of them are discarded, to finally average the rest of each parameter to obtain estimates of them.

The Gibbs sampler described above can be implemented easily with the BGLR R package: if the hyperparameters $S$-$v$ and $S_\beta$-$v_\beta$ are not specified, by default the BGLR function assigns $v = v_\beta = 5$, and to $S$ and $S_\beta$ assigns values such that the mode of the priors of $\sigma^2$ and $\sigma_\beta^2$ (inverse scaled Chi-square) matches a certain proportion of the total variance ($1 - R^2$ and $R^2$): $S = \mathrm{Var}(Y) \times (1 - R^2) \times (v + 2)$ and $S_\beta = \mathrm{Var}(Y) \times R^2 \times (v_\beta + 2)$ (see Appendix 2 for more details). Explicitly, in BGLR this model can be implemented by running the following R code:

```
ETA = list( list( model = 'BRR', X = X1, df0 = vβ, S0 = Sβ,  R2 = 1-R^2) )
A = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 = R^2)
```

where nIter = 1e4 and burnIn = 1e3 are the desired number of iterations and the number of them to be discarded when computing the estimates of the parameters. Remember that when the hyperparameter values are not given, they are set up in the default values, as previously described.

A sub-model of the BRR that does not induce shrinkage of the beta coefficients is obtained by assuming that $\left(\beta_1, \ldots, \beta_p\right)^{\mathrm{T}} \mid \sigma_\beta^2 \sim N_p\left(\mathbf{0}, \boldsymbol{I}_p\sigma_\beta^2\right)$, ignoring the prior distribution of $\sigma_\beta^2$ and setting this at a very high value ($10^{10}$). Note that this model is very similar to the Bayesian model obtained by adopting the prior (6.2), under which the beta coefficients are estimated solely with the information contained in the likelihood function (Pérez and de los Campos 2014). This prior model can also be implemented in the BGLR package and is called **FIXED**. Certainly, the Gibbs

sampler steps for its implementation are the same as the steps described before for the BRR, except that step 2 is removed (no simulations are obtained from $\sigma_\beta^2$) and $\sigma_\beta^{-2}$ is set equal to zero in the full conditional of $\boldsymbol{\beta}_0$ (step 3).

## 6.3 Bayesian GBLUP Genomic Model

In genomic-enabled prediction, the number of markers used to predict the performance of a trait of interest is often very large compared to the number of individuals phenotyped in the sample ($p \gg n$); for this reason, some computational difficulties may arise when exploring the posterior distribution of the beta coefficients. When the main objective is to use this model for predictive purposes, a solution consists of reducing the dimension of the problem by directly simulating values of $\boldsymbol{g} = \boldsymbol{X}_1\boldsymbol{\beta}_0$ (breeding values or genomic effects, Lehermeier et al. 2013) instead of only from $\boldsymbol{\beta}_0$. To do this, first note that because $\boldsymbol{\beta}_0 \mid \sigma_\beta^2 \sim N_p\left(\boldsymbol{0}, \boldsymbol{I}_p\sigma_\beta^2\right)$, to induce a prior for $\boldsymbol{g}$, this is defined as $\boldsymbol{g} = \boldsymbol{X}_1\boldsymbol{\beta}_0 \mid \sigma_\beta^2 \sim N_n\left(\boldsymbol{0}, \sigma_\beta^2\boldsymbol{X}_1\boldsymbol{X}_1^{\mathrm{T}}\right) = N_n\left(\boldsymbol{0}, \sigma_g^2\boldsymbol{G}\right)$, where $\sigma_g^2 = p\sigma_\beta^2$ and $\boldsymbol{G} = \frac{1}{p}\boldsymbol{X}_1\boldsymbol{X}_1^{\mathrm{T}}$, which is known as the genomic relationship matrix (VanRaden 2007). Then, under this parameterization ($\boldsymbol{g} = \boldsymbol{X}_1\boldsymbol{\beta}_0$ and $\sigma_g^2 = p\sigma_\beta^2$), the model specified in (6.3), in matrix notation takes the following form:

$$\boldsymbol{Y} = 1_n\mu + \boldsymbol{g} + \epsilon \tag{6.4}$$

with a flat prior to mean parameter ($\mu$), $\sigma^2 \sim \chi_{v,S}^{-2}$, and the induced priors: $\boldsymbol{g} = \boldsymbol{X}_1\boldsymbol{\beta}_0 \mid \sigma_g^2 \sim N_n\left(\boldsymbol{0}, \sigma_g^2\boldsymbol{G}\right)$ and $\sigma_g^2 \sim \chi_{v_g,\ S_g}^{-2}$ ($v_g = v_\beta$, $S_g = pS_\beta$).

Similarly to what was done for model (6.3), the full conditional posterior distribution of $\boldsymbol{g}$ in model (6.4) is given by

$$\begin{aligned}
f(\boldsymbol{g}|-) &\propto L\left(\mu, \boldsymbol{g}, \sigma^2; \boldsymbol{y}\right) f\left(\boldsymbol{g}|\sigma_g^2\right) \\
&\propto \frac{1}{(2\pi\sigma^2)^{\frac{n}{2}}} \exp\left[-\frac{1}{2\sigma^2}\|\boldsymbol{y} - 1_n\mu - \boldsymbol{g}\|^2\right] \frac{1}{\left(\sigma_g^2\right)^{\frac{n}{2}}} \exp\left(\left[-\frac{1}{2\sigma_g^2}\boldsymbol{g}^{\mathrm{T}}\boldsymbol{G}^{-1}\boldsymbol{g}\right]\right) \\
&\propto \exp\left\{-\frac{1}{2}\left[(\boldsymbol{g} - \widetilde{\boldsymbol{g}})^{\mathrm{T}}\widetilde{\boldsymbol{G}}^{-1}(\boldsymbol{g} - \widetilde{\boldsymbol{g}})\right]\right\},
\end{aligned}$$

where $\widetilde{\boldsymbol{G}} = \left(\sigma_g^{-2}\boldsymbol{G}^{-1} + \sigma^{-2}\boldsymbol{I}_n\right)^{-1}$ and $\widetilde{\boldsymbol{g}} = \sigma^{-2}\widetilde{\boldsymbol{G}}(\boldsymbol{y} - \boldsymbol{1}_n\mu)$, and from here $\boldsymbol{g} \mid - \sim N_n\left(\widetilde{\boldsymbol{g}}, \widetilde{\boldsymbol{G}}\right)$. Then, the mean/mode of $\boldsymbol{g} \mid -$ is $\widetilde{\boldsymbol{g}} = \sigma^{-2}\widetilde{\boldsymbol{G}}(\boldsymbol{y} - \boldsymbol{1}_n\mu)$, which is also the best linear unbiased predictor (BLUP) of $\boldsymbol{g}$ under the mixed model equation of Henderson (1975) using the machinery of a classic linear mixed model described in the previous chapter for model (6.4), after recognizing the

prior distribution of $\boldsymbol{g}$ as the distribution of the random effects in a model, ignoring the priors specification of the rest of the parameters and assuming that they are known (Henderson 1975). For this reason, model (6.4) is often referred to as GBLUP. If $\boldsymbol{G}$ is replaced by the pedigree matrix $\boldsymbol{A}$, the resulting model is known as PBLUP or ABLUP.

The full conditional posterior of the rest of parameters is similar to the BRR model: $\mu \mid - \sim N(\widetilde{\mu}, \widetilde{\sigma}_0^2)$, where $\widetilde{\sigma}_0^2 = \frac{\sigma^2}{n}$ and $\widetilde{\mu} = \frac{1}{n}\mathbf{1}_n^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{g})$; $\sigma^2 \mid - \sim \chi^{-2}_{\widetilde{v},\widetilde{S}}$, where $\widetilde{v} = v + n$ and $\widetilde{S} = S + \|\boldsymbol{y} - \mathbf{1}_n\mu - \boldsymbol{g}\|^2$; and $\sigma_g^2 \mid - \sim \chi^{-2}_{\widetilde{v}_g,\widetilde{S}_g}$, where $\widetilde{v}_g = v_g + n$ and $\widetilde{S}_g = S_g + \boldsymbol{g}^{\mathrm{T}}\boldsymbol{G}^{-1}\boldsymbol{g}$.

Note that when $p \gg n$, then the dimension of the parameter space of the posterior of GBLUP model is lower than the BRR.

The GBLUP model (6.4) also can be implemented easily with the BGLR R package, and when the hyperparameters $S$-$v$ and $S_g$-$v_g$ are not specified, $v = v_g = 5$ is used by default and the scale parameters are settled similarly as in the BRR.

The BGLR code to fit this model:

```
ETA = list(list(model = 'RHKS', K = G, df0 = v_g, S0 = S_g, R2 = 1-R^2)))
A = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 = R^2)
```

The GBLUP can be equivalently expressed and consequently fitted with the BRR model by making the design matrix equal to the lower triangular factor of the Cholesky decomposition of the genomic relationship matrix, i.e., $\boldsymbol{X} = \boldsymbol{L}$, where $\boldsymbol{G} = \boldsymbol{L}\boldsymbol{L}'$. So, with the BGLR package, the BRR implementation of a GBLUP model is.

```
L = t(chol(G))
ETA = list(list(model = 'BRR', X = L, df0 = v_β, S0 = S_β, R2 = 1-R^2))
A = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 = R^2)
```

When there is more than one repetition of an individual in the data at hand, or a more sophisticated design is used in the data collection, model (6.4) can be specified in a more general way to take into account this structure, as follows:

$$\boldsymbol{Y} = \mathbf{1}_n\mu + \boldsymbol{Z}\boldsymbol{g} + \boldsymbol{\epsilon} \tag{6.5}$$

with $\boldsymbol{Z}$ the incident matrix of the genotypes. This model cannot be fitted directly in the BGLR and some precalculus is needed first to compute the "covariance" matrix of the predictor $\boldsymbol{Zg}$ in model (6.5): $\mathbf{K}_L = \mathrm{Var}(\boldsymbol{Zg}) = \boldsymbol{ZGZ}^{\mathrm{T}}$. The BGLR code for implementing this model is the following:

```
Z = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
K_L = Z%*%G%*%t(Z)
ETA = list(list(model = 'RHKS', K = K_L, df0 = v_g, S0 = S_g, R2 = 1-R^2)))
A = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 = R^2)
```

where **dat_F** is the data set that contains the necessary phenotypic information (GID: Lines or individuals; y: response variable of the trait of interest).

## 6.4 Genomic-Enabled Prediction BayesA Model

Another variant to the standard Bayesian model (6.1) is the BayesA model proposed by Meuwissen et al. (2001), which is a slight modification of the BRR model obtained with the same prior distributions, except that now a specific variance $\sigma^2_{\beta_j}$ is assumed for each of the covariate (marker) effects, that is, $\beta_j \mid \sigma^2_{\beta_j} \sim N\left(0, \sigma^2_{\beta j}\right)$, and these variance parameters are supposed to be independent random variables with a scaled inverse Chi-square distribution with parameters $v_\beta$ and $S_\beta$, $\sigma^2_{\beta_j} \sim \chi^{-2}\left(v_\beta, S_\beta\right)$. These specific variances for each marker effect provide covariate heterogeneous shrinkage estimation. Furthermore, a gamma distribution is assigned to $S_\beta$, $S_\beta \sim G(r, s)$, where $G(r, s)$ denotes a gamma distribution where $r$ and $s$ are the rate and shape parameters, respectively. By providing a different prior variance for each $\beta_j$, this model has the potential of inducing covariate-specific shrinkage of estimated effects (Pérez and de los Campos 2013).

Note that choosing $r = r^*/v_\beta$ and taking very large values of $v_\beta$, the prior of $\sigma^2_{\beta j}$ collapses to a degenerate distribution at $S_\beta$, and the BRR model is obtained, but with a gamma distribution with parameters $r^*$ and $s$ as priors to the common variance of the effects $\sigma^2_\beta = \text{Var}\left(\beta_j\right) = S_\beta$, instead of $\chi^{-2}(v_\beta, S_\beta)$. Furthermore, the marginal distribution of each beta coefficient $\beta_j$, that is, the unconditional distribution of $\beta_j \mid S_\beta$, is a scaled-$t$-student distribution (scaled by $\sqrt{S_\beta/v_\beta}$). These distributions, compared to the normal, have heavier tails and put higher mass around 0, which compared to the BRR, induce fewer shrinkage estimates of covariates with sizable effects, and induce strong shrinkage toward zero estimates of covariates with smaller effects, respectively (de los Campos et al. 2013).

A Gibbs sampler implementation for estimating the parameters of this model can be done following steps 1–6 of the BRR model, where the second step is replaced by the next 2.1 and 2.2 steps, the third and the last step are replaced and modified by the next steps 3 and 6:

2.1 Simulate from the full conditional of each $\sigma_{\beta}{}^2_j$

$$\sigma^2_{\beta_j} \mid \mu, \boldsymbol{\beta}_0, \boldsymbol{\sigma}^2_{-j}, S_\beta, \sigma^2 \sim \chi^{-2}_{\widetilde{v}_j, \widetilde{S}_{\beta j}},$$

where $\widetilde{v}_{\beta_j} = v_\beta + 1$ and scale parameter $\widetilde{S}_{\beta_j} = S_\beta + \beta^2_j$, where $\boldsymbol{\sigma}^2_{-j}$ is the vector $\boldsymbol{\sigma}^2_\beta = \left(\sigma^2_{\beta_1}, \ldots, \sigma^2_{\beta_p}\right)$ but without the $j$th entry.

2.2 Simulate from the full conditional of $S_\beta$

$$f(S_\beta|-) \propto \left[\prod_{j=1}^{p} f\left(\sigma^2_{\beta_j}|S_\beta\right)\right] f(S_\beta)$$

$$\propto \prod_{j=1}^{p}\left[\frac{\left(\frac{S_\beta}{2}\right)^{\frac{v_\beta}{2}}}{\Gamma\left(\frac{v_\beta}{2}\right)\left(\sigma^2_{\beta_j}\right)^{1+\frac{v_\beta}{2}}}\exp\left(-\frac{S_\beta}{2\sigma^2_{\beta_j}}\right)\right] S_\beta^{s-1}\exp\left(-rS_\beta\right)$$

$$\propto S_\beta^{s+\frac{pv_\beta}{2}-1}\exp\left[-\left(r+\frac{1}{2}\sum_{j=1}^{p}\frac{1}{\sigma^2_{\beta j}}\right)S_\beta\right]$$

which corresponds to the kernel of the gamma distribution with rate parameter $\widetilde{r} = r + \frac{1}{2}\sum_{j=1}^{p}\frac{1}{\sigma^2_{\beta j}}$ and shape parameter $\widetilde{s} = s + \frac{pv_\beta}{2}$, and so $S_\beta \mid - \sim \text{Gamma}(\widetilde{r}, \widetilde{s})$.

3. Simulate the full conditional posterior distribution of $\boldsymbol{\beta}_0$:

$$\boldsymbol{\beta}_0 \mid \mu, \boldsymbol{\sigma}^2_\beta, \sigma^2 \sim N_p\left(\widetilde{\boldsymbol{\beta}}_0, \widetilde{\boldsymbol{\Sigma}}_0\right),$$

where $\widetilde{\boldsymbol{\Sigma}}_0 = \left(\boldsymbol{D}_p^{-1} + \sigma^{-2}\boldsymbol{X}_1^T\boldsymbol{X}_1\right)^{-1}$ and $\widetilde{\boldsymbol{\beta}}_0 = \sigma^{-2}\widetilde{\boldsymbol{\Sigma}}_0\boldsymbol{X}_1^{\mathrm{T}}(\boldsymbol{y} - \mathbf{1}_n\mu)$ , $\boldsymbol{D}_p = \text{Diag}\left(\sigma^2_{\beta_1}, \ldots, \sigma^2_{\beta_p}\right)$.

6. Repeat steps 2–5 (given in the BRR method) depending on how many values of the parameter vector $(\boldsymbol{\beta}^{\mathrm{T}}, \boldsymbol{\sigma}^2_\beta, \sigma^2, S_\beta)$ we wish to simulate.

When implementing this model in the BGLR package, by default $v = v_\beta = 5$ are used and $S = \text{Var}(Y) \times (1 - R^2) \times (v + 2)$, which makes the mode of the priors of $\sigma^2$ $(\chi^{-2}(v, S))$ match a certain proportion of the total variance $(1 - R^2)$. If the hyperparameters of the a priori for $S_\beta$, $s$, and $r$, are not specified, by default BGLR takes $s = 1.1$ to have an a priori $(S_\beta \sim G(s, r))$ that is relatively non-informative with a coefficient of variation $(1/\sqrt{s})$ of approximately 95%. Then, to the rate parameter it assigns the value $r = (s - 1)/S_\beta$, with $S_\beta = \text{Var}(y) \times R^2 \times \left(v_\beta + 2\right)/S_x^2$, where $S_x^2$ is the sum of the variances of the columns of $\boldsymbol{X}$ and $R^2$ is the percentage of the total variation that a priori is required to attribute to the covariates in $\boldsymbol{X}$. The BGLR code for implementing this model is

```
ETA = list(list(model = 'BayesA', X=X1, df0 = vβ, rate0 = r, shape0 = s,
R2 = 1-R2))
A   = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 =
R2)
```

## 6.5 Genomic-Enabled Prediction BayesB and BayesC Models

Other variants of the model (6.1) are the BayesC and the BayesB models, which in turn can be considered as direct extensions of the BRR and BayesA models, respectively, by adding a parameter $\pi$ that represents the prior proportion of covariates with nonzero effect (Pérez and de los Campos 2014).

The **BayesC** is the same as the BRR, but instead of assuming a priori that the beta coefficients are independent normal random variables with mean 0 and variance $\sigma_\beta^2$, it assumes that with probability $\pi$ each $\beta_j$ comes from a $N\left(0,\sigma_\beta^2\right)$, and with probability $1-\pi$ comes from a degenerate distribution (DG) at zero, that is, $\beta_1,\ldots,\beta_p \mid \sigma_\beta^2,\pi \overset{iid}{\sim} \pi_p \mathrm{N}\left(0,\sigma_\beta^2\right)+\left(1-\pi_p\right)\mathrm{DG}(0)$ (mix of a normal distribution with mean 0 and variance $\sigma_\beta^2$, and degenerate distribution at zero). In addition, for the parameter $\pi_p$, a beta distribution is assigned as prior, that is, $\pi_p$~Beta ($\pi_{p0}$, $\phi_0$), where $\pi_{p0}=E(\pi_p)$ represents the mean and $\phi_0^{-1}$ is the "dispersion" parameter ($\mathrm{var}\left(\pi_p\right)=\frac{\pi_{p0}\left(1-\pi_{p0}\right)}{\phi_0+1}$). If $\phi_0=2$ and $\pi_{p0}=0.5$, the prior for $\pi_p$ is a uniform distribution in (0,1). For large values of $\phi_0$, the distribution for $\pi_p$ is highly concentrated around $\pi_{p0}$, and so the BayesC is reduced to BRR when $\pi_{p0}=1$ for large values of $\phi_0$.

For this model, the full conditional distributions of $\mu$ and $\sigma_\beta^2$ are the same as the model described before, that is, $\mu \mid - \sim N\left(\widetilde{\mu},\widetilde{\sigma}_\mu^2\right)$ and $\sigma^2 \mid - \sim \chi^{-2}_{\widetilde{v},\widetilde{S}}$. However, for the rest of the parameters, this does not have a known form and is not easy to simulate from them. A solution is to introduce a latent variable to represent the prior distribution of each $\beta_j$, and compute all the conditional distributions in this augmented scheme, including the distribution corresponding to the latent variable. To do this, note that this prior can be specified by assuming that conditional to a binary latent variable $Z_j$,

$$\beta_j \mid \sigma_\beta^2, Z_j=z \sim \begin{cases} N\left(0,\sigma_\beta^2\right), \\ \mathrm{DG}(0), \end{cases}$$

where $Z_j$ is a Bernoulli random variable with parameter $\pi_p$ ($Z_j \sim \mathrm{Ber}(\pi_p)$). With this introduced latent variable, all the full conditionals can be derived, as is described next.

If the current value of $z_j$ is 1, the full conditional posterior of $\beta_j$ is

$$
\begin{aligned}
f\left(\beta_j|-\right) &\propto L\left(\mu,\boldsymbol{\beta}_0,\sigma^2;\boldsymbol{y}\right)f\left(\beta_j|\sigma_\beta^2,z_j\right)\\
&\propto \exp\left(-\frac{1}{2\sigma^2}\|\boldsymbol{y}-1_n\mu-\boldsymbol{X}_1\boldsymbol{\beta}_0\|\right)\frac{1}{\sqrt{2\pi\sigma_\beta^2}}\exp\left(-\frac{\beta_j^2}{2\sigma_\beta^2}\right)\\
&\propto \exp\left(-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y_{ij}-x_{ij}\beta_j\right)^2-\frac{1}{2\sigma_\beta^2}\beta_j^2\right)\\
&\propto \exp\left\{-\frac{1}{2}\left[\left(\sigma_\beta^{-2}+\sigma^{-2}\sum_{i=1}^{n}x_{ij}^2\right)\beta_j^2-2\sigma^{-2}\sum_{i=1}^{n}x_{ij}y_{ij}\beta_j+\frac{1}{\sigma^2}\sum_{i=1}^{n}y_{ij}^2\right]\right\}\\
&\propto \exp\left[-\frac{\left(\beta_j-\widetilde{\beta}_j\right)^2}{2\widetilde{\sigma}_j^2}\right],
\end{aligned}
$$

where $y_{ij}=y_i-\sum_{k=1, k\neq j}^{p}x_{ik}\beta_k$, $\widetilde{\sigma}_j^2=\left(\sigma_\beta^{-2}+\sigma^{-2}\sum_{i=1}^{n}x_{ij}^2\right)^{-1}$, and $\widetilde{\beta}_j=\sigma^{-2}\widetilde{\sigma}_j^2\sum_{i=1}^{n}x_{ij}y_{ij}$. That is, when the current value of $z_j$ is 1, $\beta_j\mid-\sim N\left(\widetilde{\beta}_j,\widetilde{\sigma}_j^2\right)$. However, if $z_j=0$, the full conditional posterior of $\beta_j$ is a degenerate random variable at 0, that is, $\beta_j\mid-\sim \mathrm{DG}(0)$.

The full conditional distribution of $Z_j$ is

$$
\begin{aligned}
f\left(z_j|-\right) &\propto f\left(\beta_j|\sigma_\beta^2,z_j\right)f\left(z_j\right)\\
&\propto f\left(\beta_j|\sigma_\beta^2,z_j\right)\pi_p^{z_j}\left(1-\pi_j\right)^{1-z_j}
\end{aligned}
$$

from which, conditional on the rest of the parameters, $Z_j$ is a Bernoulli random variable with parameter $\widetilde{\pi}_{pj}=\frac{\frac{\pi_p}{\sqrt{2\pi\sigma_\beta^2}}\exp\left(-\frac{\beta_j^2}{2\sigma_\beta^2}\right)}{\frac{\pi_p}{\sqrt{2\pi\sigma_\beta^2}}\exp\left(-\frac{\beta_j^2}{2\sigma_\beta^2}\right)+\left(1-\pi_p\right)\delta_0\left(\beta_j\right)}$. Note however that, if $\beta_j\neq 0$, $\widetilde{\pi}_{pj}=1$, and then $Z_j=1$ with probability 1, when simulating from the full conditional posterior of $\beta_j$, we will always obtain values different from zero, and this cyclic behavior will remain permanent. On the other hand, note that if $\beta_j=0$, $\widetilde{\pi}_{pj}=\frac{\frac{\pi_p}{\sqrt{2\pi\sigma_\beta^2}}}{\frac{\pi_p}{\sqrt{2\pi\sigma_\beta^2}}+\left(1-\pi_p\right)}$ is not 0, then the next simulated value of $\beta_j$ will be different from 0, and in this scenario, in the next steps of the "Gibbs," $Z_j$ will at all times be 1, and so the chain has absorbing states and will not explore the entire sampling space. A solution to this problem consists of trying to simulate from the joint conditional distribution of $\beta_j$ and $Z_j$, that is, from $\beta_j, Z_j\mid-$. This full joint conditional distribution can be computed as

$$f(\beta_j, z_j|-) \propto f_*(z_j) f(\beta_j|z_j, -) \propto f_*(z_j) L(\mu, \boldsymbol{\beta}_0, \sigma^2; \boldsymbol{y}) f\left(\beta_j|\sigma_\beta^2, z_j\right),$$

where $f_*(z_j)$ is the marginal conditional distribution of $Z_j$ conditioned to all parameters except $\beta_j$ ($Z_j \mid -j \sim f_*(\cdot)$). Specifically, this is given by "integrating" ($f(\beta_j, z_j|-)$ with respect to $\beta_j$)

$$\begin{aligned} f_*(z_j) &\propto \int_{-\infty}^{\infty} f(\beta_j, z_j|-) d\beta_j \propto \int_{-\infty}^{\infty} L(\mu, \boldsymbol{\beta}_0, \sigma^2; \boldsymbol{y}) f\left(\beta_j|\sigma_\beta^2, z_j\right) f(z_j|\pi_p) d\beta_j \\ &\propto \begin{cases} \int_{-\infty}^{\infty} \exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y_{ij} - x_{ij}\beta_j\right)^2\right] \frac{1}{\sqrt{2\pi\sigma_\beta^2}} \exp\left(-\frac{\beta_j^2}{2\sigma_\beta^2}\right)\pi_p d\beta_j \\ \int_{-\infty}^{\infty} \exp\left[-\frac{1}{2\sigma^2}\sum_{i=1}^{n}\left(y_{ij} - x_{ij}\beta_j\right)^2\right] \delta_0(\beta_j)(1-\pi_p) d\beta_j \end{cases} \\ &\propto \begin{cases} \pi_p\sqrt{\frac{\widetilde{\sigma}_j^2}{\sigma_\beta^2}} \\ 1 - \pi_p. \end{cases} \end{aligned}$$

From here, $Z_j \mid -$ is a Bernoulli random distribution with parameter $\widetilde{\pi}_p = \left(\pi_p\sqrt{\frac{\widetilde{\sigma}_j^2}{\sigma_\beta^2}}\right) / \left(\pi_p\sqrt{\frac{\widetilde{\sigma}_j^2}{\sigma_\beta^2}} + 1 - \pi_p\right)$. With this and the full conditional distribution derived above for $\beta_j$, an easy way to simulate values from $\beta_j, Z_j \mid -$ consists of first simulating a $z_j$ value from $Z_j \mid -j \sim \mathrm{Ber}(\widetilde{\pi}_p)$, and then, if $z_j = 1$, simulating a value of $\beta_j$ from $\beta_j \mid - \sim N\left(\widetilde{\beta}_j, \widetilde{\sigma}_j^2\right)$, otherwise take $\beta_j = 0$.

Now, note that the full conditional distribution of $\sigma_\beta^2$ is

$$\begin{aligned} f\left(\sigma_\beta^2|-\right) &\propto \left\{\prod_{j}^{p}\left[f\left(\beta_j|\sigma_\beta^2, z_j\right)\right]^{z_j}\right\} f\left(\sigma_\beta^2\right) \\ &\propto \frac{1}{\left(\sigma_\beta^2\right)^{p\bar{z}_p}} \exp\left(-\frac{1}{2\sigma_\beta^2}\sum_{j=1}^{p} z_j\beta_j^2\right) \frac{\left(\frac{S_\beta}{2}\right)^{\frac{v_\beta}{2}}}{\Gamma\left(\frac{v_\beta}{2}\right)\left(\sigma_\beta^2\right)^{1+\frac{v_\beta}{2}}} \exp\left(-\frac{S_\beta}{2\sigma_\beta^2}\right) \\ &\propto \frac{1}{\left(\sigma_\beta^2\right)^{p\bar{z}_p}} \exp\left(-\frac{1}{2\sigma_\beta^2}\sum_{j=1}^{p} z_j\beta_j^2\right) \frac{\left(\frac{S_\beta}{2}\right)^{\frac{v_\beta}{2}}}{\Gamma\left(\frac{v_\beta}{2}\right)\left(\sigma_\beta^2\right)^{1+\frac{v_\beta}{2}}} \exp\left(-\frac{S_\beta}{2\sigma_\beta^2}\right) \\ &\propto \frac{1}{\left(\sigma_\beta^2\right)^{1+\frac{\widetilde{v}_\beta}{2}}} \exp\left(-\frac{\widetilde{S}_\beta}{2\sigma_\beta^2}\right), \end{aligned}$$

where $\overline{z}_p = 1/p\sum_{j=1}^{p} z_j$, $\widetilde{S}_\beta = S_\beta + \sum_{j=1}^{p} z_j\beta_j^2$, and $\widetilde{v}_\beta = v_\beta + p\overline{z}_p$. That is, $\sigma_\beta^2 \mid \mu, \boldsymbol{\beta}_0, \sigma^2, \mathbf{z} \sim \chi^{-2}_{\widetilde{v}, \widetilde{S}_\beta}$. The full conditional distributions of $\mu$ and $\sigma^2$ are the same as BRR, that is, $\mu \mid - \sim N\left(\widetilde{\mu}, \widetilde{\sigma}_\mu^2\right)$ and $\sigma^2 \mid - \sim \chi^{-2}_{\widetilde{v}, \widetilde{S}}$, with $\widetilde{\sigma}_\mu^2 = \frac{\sigma^2}{n}$, $\widetilde{\mu} = \mathbf{1}_n^{\mathrm{T}}(\boldsymbol{y} - \boldsymbol{X}_1\boldsymbol{\beta}_0)$, $\widetilde{v} = v + n$, and $\widetilde{S} = S + \|\boldsymbol{y} - \mathbf{1}_n\mu - \boldsymbol{X}_1\boldsymbol{\beta}_0\|^2$.

The full conditional distribution of $\pi_p$ is

$$
\begin{aligned}
f(\pi_p|-) &\propto \left[\prod_{j=1}^{p} f(z_j|\pi_p)\right] f(\pi_p) \\
&\propto \pi_p^{p\overline{z}_p}(1-\pi_p)^{p(1-\overline{z}_p)}\pi_p^{\phi_0\pi_{p0}+1-1}(1-\pi_p)^{\phi_0(1-\pi_{p0})-1} \\
&\propto \pi_p^{\phi_0\pi_{p0}+p\overline{z}_p-1}(1-\pi_p)^{\phi_0(1-\pi_{p0})+p(1-\overline{z}_p)-1}
\end{aligned}
$$

which means that $\pi_p \mid - \sim$ Beta $\left(\widetilde{\pi}_{p0}, \widetilde{\phi}_0\right)$, with $\widetilde{\phi}_0 = \phi_0 + p$ and $\widetilde{\pi}_{p0} = \frac{\phi_0\pi_{p0} + p\overline{z}_p}{\phi_0 + p}$.

The **BayesB** model is a variant of BayesA that assumes almost the same prior models to the parameters, except that instead of assuming independent normal random variables with common mean 0 and common variance $\sigma_\beta^2$ for the beta coefficients, this model adopts a mixture distribution, that is, $\beta_j \mid \sigma_{\beta_j}^2, \pi \overset{iid}{\sim} \pi N\left(0, \sigma_{\beta_j}^2\right) + (1-\pi)\mathrm{DG}(0)$, with $\pi \sim \mathrm{Beta}(\pi_0, p_0)$. This model has the potential to perform variable selection and produce covariate-specific shrinkage estimates (Pérez et al. 2010).

This model can also be considered an extension of the BayesC model with a gamma distribution as prior to the scale parameter of the a priori distribution of the variance of the beta coefficients, that is, $S_\beta \sim G(s, r)$. It is interesting to point out that if $\pi = 1$, this model is reduced to BayesA, which is obtained by taking $\pi_0 = 1$ and letting $\phi_0$ go to $\infty$. Also, this is reduced to the BayesC by setting $s/r = S_\beta^0$ and choosing a very large value for $r$.

To explore the posterior distribution of this model, the same Gibbs sampler given for BayesC can be used, but adding to the process the full conditional posterior distribution of $S_\beta$: $S_\beta \mid - \sim \mathrm{Gamma}(\widetilde{r}, \widetilde{s})$, where $\widetilde{r} = r + \frac{1}{2\sigma_\beta^2}$ and shape parameter $\widetilde{s} = s + \frac{v_\beta}{2}$.

When implementing both these models in the BGLR R package, by default this assigns $\pi_{p0} = 0.5$ and $\phi_0 = 10$, for the hyperparameters of the prior model of $\pi_p$, Beta $(\pi_{p0}, \phi_0)$, which results in a weakly informative prior. For the remaining hyperparameters of the **BayesC** model, by default BGLR assigns values like those assigned to the BRR model, but with some modifications to consider because a priori now only a proportion $\pi_0$ of the covariates (columns of $\boldsymbol{X}$) has nonzero effects:

$$v = v_\beta = 5,$$
$$S = \text{Var}(Y) \times \left(1 - R^2\right) \times (v + 2),$$
$$S_\beta = \text{Var}(y) \times R^2 \times \frac{\left(v_\beta + 2\right)}{S_x^2 \pi_0}.$$

While for the remaining hyperparameters of **BayesB**, by default BGLR also assigns values similar to **BayesA**: $v = v_\beta = 5$, $S = \text{Var}(Y) \times (1 - R^2) \times (v + 2)$, $r = (s - 1)/S_\beta$, with $S_\beta = \text{Var}(y) \times R^2 \times \frac{(v_\beta+2)}{S_x^2 \pi_0}$, where $S_x^2$ is the sum of the variances of the columns of $\boldsymbol{X}$.

The BGLR codes to implement these models are, respectively:

```
ETA = list( list( model = 'BayesC', X=X1 ) )
A   = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, df0 = v, S0 = S, probIn
= πp0, counts = ϕ0, R2 = R2)
```

and

```
ETA = list( list( model = 'BayesB', X=X1 ) )
A   = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, df0 = v, rate0 = r,
shape0 = s, probIn = πp0, counts = ϕ0, R2 = R2)
```

## 6.6 Genomic-Enabled Prediction Bayesian Lasso Model

Another variant of the model (6.1) is the Bayesian Lasso linear regression model (**BL**). This model assumes independent Laplace or double-exponential distributions with location and scale parameters 0 and $\frac{\sqrt{\sigma^2}}{\lambda}$, respectively, for the beta coefficients, that is, $\beta_1, \ldots, \beta_p \mid \sigma^2, \lambda \overset{iid}{\sim} L\left(0, \frac{\sqrt{\sigma^2}}{\lambda}\right)$. Furthermore, the priors for parameters $\mu$ and $\sigma^2$ are the same as in the models described before, while for $\lambda^2$, a gamma distribution with parameters $s_\lambda$ and $r_\lambda$ is often adopted.

Because compared to the normal distribution, the Laplace distribution has fatter tails and puts higher density around 0, this prior induces stronger shrinkage estimates for covariates with relatively small effects and reduced shrinkage estimates for covariates with larger effects (Pérez et al. 2010).

A more convenient specification of the prior for the beta coefficients in this model is obtained with the representation proposed by Park and Casella (2008), which is a continuous scale mixture of a normal distribution: $\beta_j \mid \tau_j \sim N(0, \tau_j \sigma^2)$ and $\tau_j \sim \text{Exp}(2/\lambda^2)$, $j = 1, \ldots, p$, where $\text{Exp}(\theta)$ denotes an exponential distribution with scale parameter $\theta$. So, unlike the prior used by the BRR model, this prior distribution also puts a higher mass at zero and has heavier tails, which induce stronger shrinkage estimates for covariates with relatively small effect and less shrinkage estimates for markers with sizable effect (Pérez et al. 2010).

Note that the prior distribution for the beta coefficients and the prior variance of this distribution in BayesB and BayesC can be equivalently expressed as a mixture of a scaled inverse Chi-squared distribution with parameters $v_\beta$ and $S_\beta$, and a degenerate distribution at zero, that is, $\beta_j \sim N\left(0, \sigma_\beta^2\right)$ and $\sigma_\beta^2 \sim \pi_p \chi^{-2}\left(v_\beta, S_\beta\right) + \left(1 - \pi_p\right)\mathrm{DG}(0)$. So, based on this result and the connections between the models described before, the main difference between all these models is the manner in which the prior variance of the predictor variable is modelled.

**Example 1** To illustrate how to use the models described before, here we consider the prediction of grain yield (tons/ha) based on marker information. The data set used consists of 30 lines in four environments with one and two repetitions and the genotyped information contains 500 markers for each line. The numbers of lines with one (two) repetition are 6 (24), 2 (28), 0 (30), and 3 (27) in Environments 1, 2, 3, and 4, respectively, resulting in 229 observations. The performance prediction of all these models was evaluated with 10 random partitions in a cross-validation strategy, where 80% of the complete data set was used to fit the model and the rest to evaluate the model in terms of the mean squared error of prediction (MSE).

The results for all models (shown in Table 6.1) were obtained by iterating 10,000 times the corresponding Gibbs sampler and discarding the first 1000 of them, using the default hyperparameter values implemented in BGLR. This indicates that the behavior of all the models is similar, except the BayesC, where the MSE is slightly greater than the rest.

The R code to obtain the results in Table 6.1 is given in Appendix 3.

What happens when using other hyperparameter values? Although the ones used here (proposed by Pérez et al. 2010) did not always produce the best prediction performance (Lehermeier et al. 2013) and there are other ways to propose the hyperparameter values in these models (Habier et al. 2010, 2011), it is important to point out that the values used by default in BGLR work reasonably well and that it

**Table 6.1** Mean squared error (MSE) of prediction across 10 random partitions, with 80% for training and the rest for testing, in five Bayesian linear models

| PT | BRR | GBLUP | BayesA | BayesB | BayesC |
|---|---|---|---|---|---|
| 1 | 0.5395 | 0.5391 | 0.5399 | 0.5415 | 0.5470 |
| 2 | 0.3351 | 0.3345 | 0.3345 | 0.3411 | 0.3511 |
| 3 | 0.4044 | 0.4065 | 0.4056 | 0.4050 | 0.4106 |
| 4 | 0.3540 | 0.3583 | 0.3571 | 0.3561 | 0.3559 |
| 5 | 0.3564 | 0.3555 | 0.3556 | 0.3549 | 0.3604 |
| 6 | 0.7781 | 0.7772 | 0.7702 | 0.7777 | 0.7776 |
| 7 | 0.7430 | 0.7357 | 0.7380 | 0.7384 | 0.7431 |
| 8 | 0.3662 | 0.3717 | 0.3695 | 0.3669 | 0.3661 |
| 9 | 0.3065 | 0.3026 | 0.3056 | 0.3021 | 0.3030 |
| 10 | 0.5842 | 0.5822 | 0.5846 | 0.5860 | 0.6079 |
| Mean (SD) | 0.4767 (0.1742) | 0.4763 (0.1724) | 0.476 (0.1716) | 0.4769 (0.1734) | 0.4822 (0.1744) |

is not easy to find other combinations that work better in all applications, and when you want to use other combinations of hyperparameters you need to be very careful because you can dramatically affect the predictive performance of the model that uses the default hyperparameters.

Indeed, by means of simulated and experimental data, Lehermeier et al. (2013) observed a strong influence on the predictive performance of the hyperparameters given to the prior distributions in BayesA, BayesB, and the Bayes Lasso with fixed $\lambda$. Specifically, in the first two models, they observed that the scale parameter $S_\beta$ of the prior distribution of variance of $\beta_j$ had a strong effect on the predictive ability because overfitting in the data occurred when a too large value of this value was chosen, whereas underfitting was observed when too small values of this parameter were used. Note that this is expected approximately by seeing that in both models (BayesA and BayesB), $\mathrm{Var}\left(\beta_j\right) = E\left(\sigma^2_{\beta_j}\right) = S_\beta/\left(v_\beta - 1\right)$, which is almost the inverse of the regularization parameter in any type of Ridge regression model.

## 6.7 Extended Predictor in Bayesian Genomic Regression Models

All the Bayesian formulations of the model (6.1) described before can be extended, in terms of the predictor, to easily take into account the effects of other factors. For example, effects of environments and environment–marker interaction can be added as

$$\boldsymbol{y} = \mathbf{1}_n\mu + \boldsymbol{X}_E\boldsymbol{\beta}_E + \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{X}_{EM}\boldsymbol{\beta}_{EM} + \boldsymbol{\epsilon}, \tag{6.6}$$

where $\boldsymbol{X}_E$ and $\boldsymbol{X}_{EM}$ are the design matrices of the environments and environment–marker interactions, respectively, while $\boldsymbol{\beta}_E$ and $\boldsymbol{\beta}_{EM}$ are the vectors of the environment effects and the interaction effects, respectively, with a prior distribution that can be specified as was done for $\boldsymbol{\beta}$. Indeed, with the BGLR function all these things are possible, and all the options described before can also be adopted for the rest of effects added in the model: FIXED, BRR, BayesA, BayesB, BayesC, and BL.

Under the RKHS model with genotypic and environment–genotypic interaction effects, in the predictor, the modified model (6.6) is expressed as

$$\boldsymbol{Y} = \mathbf{1}_n\mu + \boldsymbol{X}_E\boldsymbol{\beta}_E + \boldsymbol{Z}_L\boldsymbol{g} + \boldsymbol{Z}_{EL}\boldsymbol{gE} + \boldsymbol{\epsilon}, \tag{6.7}$$

where $\boldsymbol{Z}_L$ and $\boldsymbol{Z}_{LE}$ are the incident matrices of the genomic and environment–genotypic interaction effects, respectively. Similarly to model (6.5), this model cannot be fitted directly in the BGLR and some precalculations are needed first to compute the "covariance" matrix of the predictors $\boldsymbol{Z}_L\boldsymbol{g}$ and $\boldsymbol{Z}_{EL}\boldsymbol{gE}$, which are $\boldsymbol{K}_L = \sigma_g^{-2}\ \mathrm{Var}(\boldsymbol{Z}_L\boldsymbol{g}) = \boldsymbol{Z}_L\boldsymbol{G}\boldsymbol{Z}_L^{\mathrm{T}}$ and $\boldsymbol{K}_{LE} = \sigma_{gE}^{-2}\mathrm{Var}(\boldsymbol{Z}_{LE}\boldsymbol{gE}) = \boldsymbol{Z}_{LE}(\boldsymbol{I}_I \otimes \boldsymbol{G})\boldsymbol{Z}_{LE}^{\mathrm{T}}$ ,

respectively, where $I$ is the number of environments. The BGLR code for implementing this model is the following:

```
 I = length(unique(dat_F$Env))
 XE = model.matrix(~0+Env,data=dat_F)[,-1]
 Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
 K_L = Z_L %*%G%*%t(Z_L)
 Z_LE = model.matrix(~0+GID:Env,data=dat_F,
        xlev = list(GID=unique(dat_F$GID),Env = unique(dat_F$Env)))
 K_LE = Z_LE%*%kronecker(diag(I),G)%*%t(Z_LE)
     ETA = list( list(model='FIXED',X=XE),
             list( model = 'RHKS', K = K_L, df0 = v_g, S0 = S_g, R2 = 1-R^2)),
               list(model='RKHS',K=K_LE )
A   = BGLR(y,ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 = R^2)
```

where **dat_F** is the data set that contains the necessary phenotypic information (GID: Lines or individuals; Env: Environment; y: response variable of trait under study).

**Example 2 (Includes Models with Only Env Effects and Models with Env and LinexEnv Effects)** To illustrate how to fit the extended genomic regression models described before, here we consider the prediction of grain yield (tons/ha) based on marker information and the genomic relationship derived from it. The data set used consists of 30 lines in four environments, and the genotyped information of 500 markers for each line. The performance prediction of 18 models was evaluated with a five-fold cross-validation, where 80% of the complete data set was used to fit the model and the rest to evaluate the model in terms of the mean squared error of prediction (MSE). These models were obtained by considering different predictors (marker, environment, or/and environment–marker interaction) and different prior models to the parameters of each predictor included.

The model M1 only considered in the predictor the marker effects, from which six sub-models were obtained by adopting one of the six options (BRR, RKHS, BayesA, BayesB, BayesC, or BL) to the prior model of $\boldsymbol{\beta}$ (or $\boldsymbol{g}$). Model M2 is model M1 plus the environment effects with a FIXED prior model, for all prior sub-models in the marker predictor. Model M3 is model M2 plus the environment–marker interaction, with a prior model of the same family as those chosen for the marker predictor (see in Table 6.2 all the models we compared).

The performance prediction of the models presented in Table 6.2 is shown in Table 6.3. The first column represents the kind of prior model used in both marker effects and env:marker interaction terms, when the latter is included in the model. In each of the first five prior models, model M2 resulted in better MSE performance, while when the BL prior model was used, model M3, the model with the interaction term, was better. The greater difference is between M1 and M2, where the average MSE across all priors of the first model is approximately 21% greater than the corresponding average of the M2 model. Similar behavior was observed with Pearson's correlation, with the average of this criterion across all priors about 32%

**Table 6.2** Fitted models: $M_{md}$, $m = 1, 2, 3, d = 1, \ldots, 6$

| Model | Sub-model | Predictor | | |
|---|---|---|---|---|
| | | Marker or G: $\boldsymbol{X\beta}$ | Environment: $\boldsymbol{X_E\beta_E}$ | Interaction Env-Markers: $\boldsymbol{X_{EM}\beta_{EM}}$ |
| M1 | M11 | BRR | | |
| | M12 | RKHS | | |
| | M12 | BayesA | | |
| | M14 | BayesB | | |
| | M15 | BayesC | | |
| | M16 | BL | | |
| M2 | M21 | BRR | FIXED | |
| | M22 | RKHS | FIXED | |
| | M23 | BayesA | FIXED | |
| | M24 | BayesB | FIXED | |
| | M25 | BayesC | FIXED | |
| | M26 | BL | FIXED | |
| M3 | M31 | BRR | FIXED | BRR |
| | M32 | RKHS | FIXED | RKHS |
| | M33 | BayesA | FIXED | BayesA |
| | M34 | BayesB | FIXED | BayesB |
| | M35 | BayesC | FIXED | BayesC |
| | M36 | BL | FIXED | BL |

The third to fifth columns are the considered predictors corresponding to marker (genetic effect), environment, and environment–marker (genetic effect) effects, respectively. The cells in row 2 and column 3 indicate the prior distribution model specified for the beta parameters in each predictor when that parameter is present. M1 $: \boldsymbol{Y} = \boldsymbol{1\mu} + \boldsymbol{X\beta} + \boldsymbol{\epsilon}$, M2 $: \boldsymbol{Y} = \boldsymbol{1}\mu + \boldsymbol{X_E\beta_E} + \boldsymbol{X\beta} + \boldsymbol{\epsilon}$, and M3 $: \boldsymbol{Y} = \boldsymbol{1}\mu + \boldsymbol{X_E\beta_E} + \boldsymbol{X\beta} + \boldsymbol{X_{EM}\beta_{EM}} + \boldsymbol{\epsilon}$. For the RKHS prior model, predictors $\boldsymbol{X\beta}$ and $\boldsymbol{X_{EM}\beta_{EM}}$ are replaced by $\boldsymbol{g}$ and $\boldsymbol{Eg}$, respectively

greater in model M2 than in M1. So the inclusion of the environment effect was important, but not the environment:marker interaction.

The best prediction performance in terms of MSE was obtained with sub-model M25 (M2 with a BayesC prior) followed by M21 (M2 with a BRR prior). However, the difference between those and sub-models M22, M23, and M24, also derived from M2, is only slight and a little more than with M26, which as commented before, among the models that assume a BL prior, showed a worse performance than M36 (M3 plus a BL prior for marker effects and environment–marker interaction).

## 6.8 Bayesian Genomic Multi-trait Linear Regression Model

The univariate models described for continuous outcomes do not exploit the possible correlation between traits, when the selection of better individuals is based on several traits and these univariate models are fitted separately to each trait. Relative

**Table 6.3** Performance prediction of models in Table 6.2: Mean squared error of prediction (MSE) and average Pearson's correlation (PC), each with its standard deviation across the five partitions

| | Markers | | Env + Markers | | Env + Markers + Env:Markers | |
|---|---|---|---|---|---|---|
| | M1 | | M2 | | M3 | |
| Prior | MSE (SD) | PC (SD) | MSE (SD) | PC (SD) | MSE (SD) | PC (SD) |
| BRR | 0.5384 (0.08) | 0.4678 (0.11) | 0.4446 (0.11) | 0.6112 (0.1) | 0.4782 (0.12) | 0.579 (0.11) |
| RKHS | 0.5509 (0.09) | 0.4505 (0.13) | 0.4542 (0.12) | 0.6012 (0.1) | 0.51 (0.13) | 0.5271 (0.12) |
| BayesA | 0.5441 (0.08) | 0.4622 (0.12) | 0.4452 (0.11) | 0.616 (0.1) | 0.4806 (0.15) | 0.5964 (0.11) |
| BayesB | 0.5462 (0.08) | 0.4595 (0.12) | 0.4455 (0.11) | 0.6141 (0.1) | 0.4669 (0.13) | 0.5981 (0.11) |
| BayesC | 0.5449 (0.08) | 0.4615 (0.12) | 0.4416 (0.11) | 0.614 (0.1) | 0.474 (0.12) | 0.5864 (0.11) |
| BL | 0.5621 (0.1) | 0.4529 (0.12) | 0.4719 (0.14) | 0.5871 (0.09) | 0.4537 (0.12) | 0.606 (0.1) |

The first column represents the prior used in the model in both marker main effects and env:marker interaction term (when included in the model)

to this, some advantages of jointly modelling the multi-traits is that in this way the correlation among the traits is appropriately accounted for and can help to improve statistical power and the precision of parameter estimation, which are very important in genomic selection, because they can help to improve prediction accuracy and reduce trait selection bias (Schaeffer 1984; Pollak et al. 1984; Montesinos-López et al. 2018).

An example of this is when crop breeders collect phenotypic data for multiple traits such as grain yield and its components (grain type, grain weight, biomass, etc.), tolerance to biotic and abiotic stresses, and grain quality (taste, shape, color, nutrient, and/or content) (Montesinos-López et al. 2016). In this and many other cases, sometimes the interest is to predict traits that are difficult or expensive to measure with those that are easy to measure or the aim can be to improve all these correlated traits simultaneously (Henderson and Quaas 1976; Calus and Veerkamp 2011; Jiang et al. 2015). In these lines, there is evidence of the usefulness of multi-trait modelling. For example, Jia and Jannink (2012) showed that, compared to single-trait modelling, the prediction accuracy of low-heritability traits could be increased by using a multi-trait model when the degree of correlation between traits is at least moderate. They also found that multi-trait models had better prediction accuracy when phenotypes were not available on all individuals and traits. Joint modelling also has been found useful for increasing prediction accuracy when the traits of interest are not measured in the individuals of the testing set, but this and other traits were observed in individuals in the training set (Pszczola et al. 2013).

### 6.8.1 Genomic Multi-trait Linear Model

The genomic multi-trait linear model adopts a univariate genomic linear model structure for each trait but with correlated residuals and genotypic effects for traits in the same individual. Assuming that for individual $j$, $n_T$ traits ($Y_{jt}$, $t = 1, \ldots, n_T$) are measured, this model assumes that

$$\begin{bmatrix} Y_{j1} \\ Y_{j2} \\ \vdots \\ Y_{jn_T} \end{bmatrix} = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{n_T} \end{bmatrix} + \begin{bmatrix} \boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{\beta}_1 \\ \boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{\beta}_2 \\ \vdots \\ \boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{\beta}_{n_T} \end{bmatrix} + \begin{bmatrix} g_{j1} \\ g_{j2} \\ \vdots \\ g_{jn_T} \end{bmatrix} + \begin{bmatrix} \epsilon_{j1} \\ \epsilon_{j2} \\ \vdots \\ \epsilon_{jn_T} \end{bmatrix},$$

where $\mu_t$, $t = 1, \ldots, n_T$, are the specific trait intercepts, $\boldsymbol{x}_j$ is a vector of covariates equal for all traits, $g_{jt}$, $t = 1, \ldots, n_T$, are the specific trait genotype effects, and $\epsilon_{jt}$, $t = 1, \ldots, n_T$ are the random error terms corresponding to each trait. In matrix notation, it can be expressed as

$$\boldsymbol{Y}_j = \boldsymbol{\mu} + \boldsymbol{B}^{\mathrm{T}} \boldsymbol{x}_j + \boldsymbol{g}_j + \boldsymbol{\epsilon}_j, \tag{6.8}$$

where $\boldsymbol{Y}_j = \left[Y_{j1}, \ldots, Y_{jn_T}\right]^{\mathrm{T}}$, $\boldsymbol{\mu} = \left[\mu_1, \ldots, \mu_{n_T}\right]^{\mathrm{T}}$, $\boldsymbol{B} = \left[\boldsymbol{\beta}_1, \quad \ldots, \quad \boldsymbol{\beta}_{n_T}\right]$, $\boldsymbol{g}_j = \left[g_{j1}, \ldots, g_{jn_T}\right]^{\mathrm{T}}$, and $\boldsymbol{\epsilon}_j = \left[\epsilon, \ldots, \epsilon_{n_T}\right]^{\mathrm{T}}$. The residual vectors are assumed to be independent with multivariate normal distribution, that is $\boldsymbol{\epsilon}_j \sim N_{n_T}(\boldsymbol{0}, \boldsymbol{R})$, and all the random genotype effects are assumed to be $\boldsymbol{g} = \left[\boldsymbol{g}_1^{\mathrm{T}}, \ldots, \boldsymbol{g}_J^{\mathrm{T}}\right]^{\mathrm{T}} \sim N(\boldsymbol{0}, \boldsymbol{G} \otimes \boldsymbol{\Sigma}_{\mathrm{T}})$, $\otimes$ being the Kronecker product. For a full Bayesian specification of this model, we suppose that $\boldsymbol{\beta} = \mathrm{vec}(\boldsymbol{B}) \sim N(\boldsymbol{\beta}_0, \boldsymbol{\Sigma}_\beta)$, that is, marginally, for the fixed effect of each trait, a prior multivariate normal distribution is adopted, $\boldsymbol{\beta}_t \sim N_p\left(\boldsymbol{\beta}_{t0}, \boldsymbol{\Sigma}_{\beta_t}\right), t = 1, \ldots, n_T$; a flat prior for the intercepts, $f(\boldsymbol{\mu}) \propto 1$; and independent inverse Wishart distributions for the covariance matrix of residuals $\boldsymbol{R}$ and for $\boldsymbol{\Sigma}_T$, that is, $\boldsymbol{\Sigma}_T \sim \mathrm{IW}(v_t, \boldsymbol{S}_t)$ and $\boldsymbol{R} \sim \mathrm{IW}(v_R, \boldsymbol{S}_R)$.

Putting all the information together where the measured traits of each individual ($\boldsymbol{Y}_j$) are accommodated in the rows of a matrix response ($\boldsymbol{Y}$), model (6.8) can be expressed as

$$\boldsymbol{Y} = \boldsymbol{1}_J \boldsymbol{\mu}^{\mathrm{T}} + \boldsymbol{X}\boldsymbol{B} + \boldsymbol{Z}_1 \boldsymbol{b}_1 + \boldsymbol{E}, \tag{6.9}$$

where $=[\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_J]^{\mathrm{T}}$, $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_J]^{\mathrm{T}}$, $\boldsymbol{b}_1 = [\boldsymbol{g}_1, \ldots, \boldsymbol{g}_J]^{\mathrm{T}}$, and $\boldsymbol{E} = [\boldsymbol{\epsilon}_1, \ldots, \boldsymbol{\epsilon}_J]^{\mathrm{T}}$. Note that under this notation, $\boldsymbol{E}^{\mathrm{T}} \sim MN_{n_T \times J}(\boldsymbol{0}, \boldsymbol{R}, \boldsymbol{I}_J)$ or equivalently $\boldsymbol{E} \sim MN_{J \times n_T}(\boldsymbol{0}, \boldsymbol{I}_J, \boldsymbol{R})$, and $\boldsymbol{b}_1^{\mathrm{T}} \sim MN_{n_T \times J}(\boldsymbol{0}, \boldsymbol{\Sigma}_T, \boldsymbol{G})$ or $\boldsymbol{b}_1 \sim MN_{J \times n_T}(\boldsymbol{0}, \boldsymbol{G}, \boldsymbol{\Sigma}_T)$. Here $\boldsymbol{Z} \sim MN_{J \times n_T}(\boldsymbol{M}, \boldsymbol{U}, \boldsymbol{V})$ means that the random matrix $\boldsymbol{Z}$ follows the matrix variate normal distribution with parameters $\boldsymbol{M}$, $\boldsymbol{U}$, and $\boldsymbol{V}$, or equivalently, that the $Jn_T$ random vector vec($\boldsymbol{Z}$) is distributed as $N_{Jn_T}(\mathrm{vec}(\boldsymbol{M}), \boldsymbol{V} \otimes \boldsymbol{U})$, with vec($\cdot$) denoting the vectorization of a matrix that stacks the columns of this in a single column. Note that when $\boldsymbol{\Sigma}_T$ and $\boldsymbol{R}$ are diagonal matrices, model (6.9) is equivalent to separately fitting a univariate GBLUP model to each trait.

The conditional distribution of all traits is given by

$$\begin{aligned} f(\boldsymbol{Y}|\boldsymbol{\mu}, \boldsymbol{\beta}, \boldsymbol{b}_1, \boldsymbol{\Sigma}_T, \mathbf{R}) &= \frac{|\boldsymbol{R}|^{-\frac{J}{2}}}{(2\pi)^{Jn_T}} \exp\left\{-\frac{1}{2}\mathrm{tr}\left[\boldsymbol{R}^{-1}\left(\boldsymbol{Y} - 1_J\boldsymbol{\mu}^{\mathrm{T}} - \boldsymbol{X}\boldsymbol{B} - \boldsymbol{Z}_1\boldsymbol{b}_1\right)^{\mathrm{T}} \boldsymbol{I}_J \left(\boldsymbol{Y} - 1_J\boldsymbol{\mu}^{\mathrm{T}} - \boldsymbol{X}\boldsymbol{\beta} - \boldsymbol{Z}_1\boldsymbol{b}_1\right)\right]\right\} \\ &= \frac{|\boldsymbol{R}|^{-\frac{J}{2}}}{(2\pi)^{Jn_T}} \exp\left\{-\frac{1}{2}\left[\mathrm{vec}\left(\boldsymbol{Y} - 1_J\boldsymbol{\mu}^{\mathrm{T}} - \boldsymbol{X}\boldsymbol{B} - \boldsymbol{Z}_1\boldsymbol{b}_1\right)\right]^{\mathrm{T}} \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{I}_J\right) \left[\mathrm{vec}\left(\boldsymbol{Y} - 1_J\boldsymbol{\mu}^{\mathrm{T}} - \boldsymbol{X}\boldsymbol{B} - \boldsymbol{Z}_1\boldsymbol{b}_1\right)\right]\right\} \end{aligned}$$

and the joint posterior of parameters $\boldsymbol{\mu}$, $\boldsymbol{B}$, $\boldsymbol{b}_1$, $\boldsymbol{\Sigma}_T$, and $\boldsymbol{R}$ is given by

$$f(\boldsymbol{\mu}, \boldsymbol{B}, \boldsymbol{b}_1, \boldsymbol{\Sigma}_T, \boldsymbol{R}|\boldsymbol{Y}) \propto f(\boldsymbol{Y}|\boldsymbol{\mu}, \boldsymbol{B}, \boldsymbol{b}_1, \boldsymbol{\Sigma}_T, \mathbf{R}) f(\boldsymbol{b}_1|\boldsymbol{\Sigma}_T) f(\boldsymbol{\Sigma}_T) f(\boldsymbol{\beta}) f(\boldsymbol{R}),$$

where $f(\boldsymbol{b}_1|\boldsymbol{\Sigma}_T)$ denotes the conditional distribution of the genotype effects, and $f(\boldsymbol{\Sigma}_T)$, $f(\boldsymbol{\beta})$, and $f(\boldsymbol{R})$ denote the prior density distribution of $\boldsymbol{\Sigma}_T$, $\boldsymbol{B}$, and $\boldsymbol{R}$, respectively. This joint posterior distribution of the parameters doesn't have closed form; for this reason, next are derived the full conditional distributions for Gibbs sampler implementation.

Let $\boldsymbol{\beta}_0$ and $\boldsymbol{\Sigma}_\beta$ be the prior mean and variance of $\boldsymbol{\beta} = \text{vec}(\boldsymbol{B})$. Because tr $(\boldsymbol{AB}) = \text{vec}(\boldsymbol{A}^{\text{T}})^{\text{T}}\text{vec}(\boldsymbol{B}) = \text{vec}(\boldsymbol{B})^{\text{T}}\text{vec}(\boldsymbol{A}^{\text{T}})$ and vec $(\boldsymbol{AXB}) = (\boldsymbol{B}^{\text{T}} \otimes \boldsymbol{A})\text{vec}(\boldsymbol{X})$**,** we have that

$$\begin{aligned}
P(\boldsymbol{\beta}|-) &\propto f(\boldsymbol{Y}|\boldsymbol{\mu},\boldsymbol{B},\boldsymbol{b}_1,\boldsymbol{\Sigma}_T,\mathbf{R})f(\boldsymbol{\beta}) \\
&\propto \exp\left\{\begin{array}{l} -\frac{1}{2}\left[\text{vec}\left(\boldsymbol{Y}-1_J\boldsymbol{\mu}^{\text{T}}-\boldsymbol{Z}_1\boldsymbol{b}_1\right)-(\boldsymbol{I}_{n_T}\otimes\boldsymbol{X})\text{vec}(\boldsymbol{B})\right]^{\text{T}}\left(\boldsymbol{R}^{-1}\otimes\boldsymbol{I}_J\right) \\ \times[\text{vec}(\boldsymbol{Y}-1_J\boldsymbol{\mu}^{\text{T}}-\boldsymbol{Z}_1\boldsymbol{b}_1)-(\boldsymbol{I}_{n_T}\otimes\boldsymbol{X})\text{vec}(\boldsymbol{B})]-\frac{1}{2}(\boldsymbol{\beta}-\boldsymbol{\beta}_0)^{\text{T}}\boldsymbol{\Sigma}_\beta^{-1}(\boldsymbol{\beta}-\boldsymbol{\beta}_0) \end{array}\right\} \\
&\propto \exp\left\{\begin{array}{l} -\frac{1}{2}\left[\text{vec}\left(\boldsymbol{Y}-1_J\boldsymbol{\mu}^{\text{T}}-\boldsymbol{Z}_1\boldsymbol{b}_1\right)-(\boldsymbol{I}_{n_T}\otimes\boldsymbol{X})\boldsymbol{\beta}\right]^{\text{T}}\left(\boldsymbol{R}^{-1}\otimes\boldsymbol{I}_J\right) \\ \times[\text{vec}(\boldsymbol{Y}-1_J\boldsymbol{\mu}^{\text{T}}-\boldsymbol{Z}_1\boldsymbol{b}_1)-(\boldsymbol{I}_{n_T}\otimes\boldsymbol{X})\boldsymbol{\beta}]-\frac{1}{2}(\boldsymbol{\beta}-\boldsymbol{\beta}_0)^{\text{T}}\boldsymbol{\Sigma}_\beta^{-1}(\boldsymbol{\beta}-\boldsymbol{\beta}_0) \end{array}\right\} \\
&\propto \exp\left\{-\frac{1}{2}\left[\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}_0\right]^{\text{T}}\widetilde{\boldsymbol{\Sigma}}_\beta^{-1}\left[\boldsymbol{\beta}-\widetilde{\boldsymbol{\beta}}_0\right]\right\},
\end{aligned}$$

where $\widetilde{\boldsymbol{\Sigma}}_\beta = \left[\boldsymbol{\Sigma}_\beta^{-1} + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{X}^{\text{T}}\boldsymbol{X}\right)\right]^{-1}$ and $\widetilde{\boldsymbol{\beta}}_0 = \widetilde{\boldsymbol{\Sigma}}_\beta\left[\boldsymbol{\Sigma}_\beta^{-1}\boldsymbol{\beta}_0 + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{X}^{\text{T}}\right)\text{vec}\right.$ $\left.(\boldsymbol{Y} - \mathbf{1}_J\boldsymbol{\mu}^{\text{T}} - \boldsymbol{Z}_1\boldsymbol{b}_1)\right]$. So, the full conditional distribution of $\boldsymbol{\beta}$ is $N_p\left(\widetilde{\boldsymbol{\beta}}_0, \widetilde{\boldsymbol{\Sigma}}_\beta\right)$. Similarly, the full conditional distribution of $\boldsymbol{g} = \text{vec}(\boldsymbol{b}_1)$ is $N_J\left(\widetilde{\boldsymbol{g}}, \widetilde{\boldsymbol{G}}\right)$, with $\widetilde{\boldsymbol{G}} = \left[\left(\boldsymbol{\Sigma}_T^{-1} \otimes \boldsymbol{G}^{-1}\right) + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{Z}_1^{\text{T}}\boldsymbol{Z}_1\right)\right]^{-1}$ and $\widetilde{\boldsymbol{g}} = \widetilde{\boldsymbol{G}}\left(\boldsymbol{R}^{-1} \otimes \boldsymbol{Z}_1^{\text{T}}\right)\text{vec}(\boldsymbol{Y} - \mathbf{1}_J\boldsymbol{\mu}^{\text{T}} - \boldsymbol{XB})$. Now, because $\text{vec}(\mathbf{1}_J\boldsymbol{\mu}^{\text{T}}) = (\boldsymbol{I}_{n_T} \otimes \mathbf{1}_J)\boldsymbol{\mu}$, similarly as before, the full conditional of $\boldsymbol{\mu}$ is $N_{n_T}\left(\widetilde{\boldsymbol{\mu}}, \widetilde{\boldsymbol{\Sigma}}_\mu\right)$, where $\widetilde{\boldsymbol{\Sigma}}_\mu = J^{-1}\boldsymbol{R}$ and $\widetilde{\boldsymbol{\mu}} = \widetilde{\boldsymbol{\Sigma}}_\mu\left(\boldsymbol{R}^{-1} \otimes \mathbf{1}_J\right)\text{vec}(\boldsymbol{Y} - \boldsymbol{XB} - \boldsymbol{Z}_1\boldsymbol{B})$.

The full conditional distribution of $\boldsymbol{\Sigma}_T$

$$\begin{aligned}
P(\boldsymbol{\Sigma}_T|-) &\propto P(\boldsymbol{b}_1|\boldsymbol{\Sigma}_T)P(\boldsymbol{\Sigma}_T) \\
&\propto |\boldsymbol{\Sigma}_T|^{-\frac{J}{2}}|\boldsymbol{G}|^{-\frac{n_T}{2}}\exp\left\{-\frac{1}{2}\text{tr}\left[\boldsymbol{b}_1^{\text{T}}\boldsymbol{G}^{-1}\boldsymbol{b}_1\boldsymbol{\Sigma}_T^{-1}\right]\right\}P(\boldsymbol{\Sigma}_T) \\
&\propto \boldsymbol{\Sigma}_T^{-\frac{v_T+J+n_T+1}{2}}\exp\left\{-\frac{1}{2}\text{tr}\left(\boldsymbol{b}_1^{\text{T}}\boldsymbol{G}^{-1}\boldsymbol{b}_1 + \boldsymbol{S}_T\right)\boldsymbol{\Sigma}_T^{-1}\right\}.
\end{aligned}$$

From here we have that $\boldsymbol{\Sigma}_T \mid - \sim \text{IW}\left(v_T + J, \boldsymbol{b}_1^{\text{T}}\boldsymbol{G}^{-1}\boldsymbol{b}_1 + \boldsymbol{S}_T\right)$. Now, because

$$\begin{aligned}
P(\boldsymbol{R}|-) &\propto f(\boldsymbol{Y}|\boldsymbol{\mu},\boldsymbol{B},\boldsymbol{b}_1,\boldsymbol{\Sigma}_T,\mathbf{R})f(\boldsymbol{R}) \\
&\propto |\boldsymbol{R}|^{-\frac{J}{2}}\exp\left\{-\frac{1}{2}\text{tr}\left[\boldsymbol{R}^{-1}\left(\boldsymbol{Y}-1_J\boldsymbol{\mu}^{\text{T}}-\boldsymbol{XB}-\boldsymbol{Z}_1\boldsymbol{b}_1\right)^{\text{T}}\boldsymbol{I}_J\left(\boldsymbol{Y}-1_J\boldsymbol{\mu}^{\text{T}}-\boldsymbol{X\beta}-\boldsymbol{Z}_1\boldsymbol{b}_1\right)\right]\right\} \\
&\quad |\boldsymbol{R}|^{-\frac{v_R+n_T+1}{2}}\exp\left\{-\frac{1}{2}\text{tr}\left(\boldsymbol{S}_T\boldsymbol{R}^{-1}\right)\right\} \\
&\propto |\boldsymbol{R}|^{-\frac{v_R+J+n_T+1}{2}}\exp\left\{-\frac{1}{2}\text{tr}\left[\boldsymbol{S}_T+\left(\boldsymbol{Y}-1_J\boldsymbol{\mu}^{\text{T}}-\boldsymbol{XB}-\boldsymbol{Z}_1\boldsymbol{b}_1\right)^{\text{T}}\left(\boldsymbol{Y}-1_J\boldsymbol{\mu}^{\text{T}}-\boldsymbol{X\beta}-\boldsymbol{Z}_1\boldsymbol{b}_1\right)\right]\boldsymbol{R}^{-1}\right\}
\end{aligned}$$

the full conditional distribution of $\boldsymbol{R}$ is $\text{IW}\left(\widetilde{v}_R, \widetilde{\boldsymbol{S}}_R\right)$, where $\widetilde{v}_R = v_R + J$ and $\widetilde{\boldsymbol{S}}_R = \boldsymbol{S}_T + (\boldsymbol{Y} - \mathbf{1}_J\boldsymbol{\mu}^{\text{T}} - \boldsymbol{XB} - \boldsymbol{Z}_1\boldsymbol{b}_1)^{\text{T}}(\boldsymbol{Y} - \mathbf{1}_J\boldsymbol{\mu}^{\text{T}} - \boldsymbol{X\beta} - \boldsymbol{Z}_1\boldsymbol{b}_1)$.

In summary, a Gibbs sampler exploration of the joint posterior distribution of $\boldsymbol{\mu}$, $\boldsymbol{\beta}$, $\boldsymbol{g}$, $\boldsymbol{\Sigma}_T$, and $\boldsymbol{R}$ can be done with the following steps:

1. Simulate $\boldsymbol{\beta}$ from a multivariate normal distribution $N_p\left(\widetilde{\boldsymbol{\beta}}_0, \widetilde{\boldsymbol{\Sigma}}_{\boldsymbol{\beta}}\right)$, where $\widetilde{\boldsymbol{\Sigma}}_{\boldsymbol{\beta}} = \left[\boldsymbol{\Sigma}_{\beta}^{-1} + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}\right)\right]^{-1}$ and $\widetilde{\boldsymbol{\beta}}_0 = \widetilde{\boldsymbol{\Sigma}}_{\boldsymbol{\beta}}\left[\boldsymbol{\Sigma}_{\beta}^{-1}\boldsymbol{\beta}_0 + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{X}^{\mathrm{T}}\right)\mathrm{vec}\left(\boldsymbol{Y} - \mathbf{1}_J\boldsymbol{\mu}^{\mathrm{T}} - \boldsymbol{Z}_1\boldsymbol{b}_1\right)\right]$.
2. Simulate $\boldsymbol{\mu}$ from $N_{n_T}\left(\widetilde{\boldsymbol{\mu}}, \widetilde{\boldsymbol{\Sigma}}_{\mu}\right)$, where $\widetilde{\boldsymbol{\Sigma}}_{\mu} = J^{-1}\boldsymbol{R}$ and $\widetilde{\boldsymbol{\mu}} = \widetilde{\boldsymbol{\Sigma}}_{\mu}\left(\boldsymbol{R}^{-1} \otimes \mathbf{1}_J\right)$ $\mathrm{vec}\left(\boldsymbol{Y} - \boldsymbol{XB} - \boldsymbol{Z}_1\boldsymbol{B}\right)$.
3. Simulate $\boldsymbol{g} = \mathrm{vec}(\boldsymbol{b}_1)$ from $N_J\left(\widetilde{\boldsymbol{g}}, \widetilde{\boldsymbol{G}}\right)$, where $\widetilde{\boldsymbol{G}} = \left[\left(\boldsymbol{\Sigma}_T^{-1} \otimes \boldsymbol{G}^{-1}\right) + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{Z}_1^{\mathrm{T}}\boldsymbol{Z}_1\right)\right]^{-1}$ and $\widetilde{\boldsymbol{g}} = \widetilde{\boldsymbol{G}}\left(\boldsymbol{R}^{-1} \otimes \boldsymbol{Z}_1^{\mathrm{T}}\right)\mathrm{vec}\left(\boldsymbol{Y} - \mathbf{1}_J\boldsymbol{\mu}^{\mathrm{T}} - \boldsymbol{XB}\right)$.
4. Simulate $\boldsymbol{\Sigma}_T$ from $\mathrm{IW}\left(v_T + J, \boldsymbol{b}_1^{\mathrm{T}}\boldsymbol{G}^{-1}\boldsymbol{b}_1 + \boldsymbol{S}_T\right)$.
5. Simulate $\boldsymbol{R}$ from $\mathrm{IW}\left(\widetilde{v}_R, \widetilde{\boldsymbol{S}}_R\right)$, where $\widetilde{v}_R = v_R + J$ and $\widetilde{\boldsymbol{S}}_R = \boldsymbol{S}_T + \left(\boldsymbol{Y} - \mathbf{1}_J\boldsymbol{\mu}^{\mathrm{T}} - \boldsymbol{XB} - \boldsymbol{Z}_1\boldsymbol{b}_1\right)^{\mathrm{T}}\left(\boldsymbol{Y} - \mathbf{1}_J\boldsymbol{\mu}^{\mathrm{T}} - \boldsymbol{X\beta} - \boldsymbol{Z}_1\boldsymbol{b}_1\right)$.
6. Return to step 1 or terminate when chain length is adequate to meet convergence diagnostics and the required sample size is reached.

An implementation of this model can be done using the github version of the BGLR R library, which can be accessed from https://github.com/gdlc/BGLR-R and can be installed directly in the R console by running the following commands: *install.packages('devtools'); library(devtools); install_git('https://github.com/gdlc/BGLR-R')*. This implementation also uses a flat prior for the fixed effect regression coefficients $\boldsymbol{\beta}$, and in such a case, the corresponding full conditional of this parameter is the same as step 1 of the Gibbs sampler given before, but removing $\boldsymbol{\Sigma}_{\beta}^{-1}$ and $\boldsymbol{\Sigma}_{\beta}^{-1}\boldsymbol{\beta}_0$ from $\widetilde{\boldsymbol{\Sigma}}_{\boldsymbol{\beta}}$ and $\widetilde{\boldsymbol{\beta}}_0$, respectively. Specifically, model (6.8) can be implemented with this version of the BGLR package as follows:

```
ETA = list( list( X=X, model='FIXED' ), list( K=Z1 G Z1^T, model=' RKHS' ) )
A   =Multitrait(y = Y, ETA=ETA, resCov = list( type = 'UN', S0 = S_R, df0 =
v_R ), nIter = nI, burnIn = nb)
```

The first argument in the Multitrait function is the response variable which many times is a phenotype matrix where each row corresponds to the measurement of $n_T$ traits in each individual. The second argument is a list predictor in which the first sub-list specifies the design matrix and prior model to the fixed effects part of the predictor in model (6.9), and in the second sub-list are specified the parameters of the distribution of random genetic effects of $\boldsymbol{b}_1$, where it is specified the $K = \boldsymbol{G}$ genomic relationship matrix, that accounts for the similarity between individuals based on marker information, df0 $= \boldsymbol{v}_T$ and $S0 = \boldsymbol{S}_T$ are the degrees of freedom parameter ($v_T$) and the scale matrix parameter ($\boldsymbol{S}_T$) of the inverse Wishart prior distribution for $\boldsymbol{\Sigma}_T$, respectively. In the third argument (resCOV), $S0$ and df0 are the scale matrix parameter ($\boldsymbol{S}_R$) and the degree of freedom parameter ($v_R$) of the inverse Wishart

prior distribution for $\boldsymbol{R}$. The last two arguments are the required number of iterations (**nI**) and the burn-in period (**nb**) for running the Gibbs sampler.

Similarly to the univariate case, model (6.9) can be equivalently described and implemented as a multivariate Ridge regression model, as follows:

$$\boldsymbol{Y} = \mathbf{1}_J\boldsymbol{\mu}^{\mathrm{T}} + \boldsymbol{X}\boldsymbol{B} + \boldsymbol{X}_1\boldsymbol{B}_1 + \boldsymbol{E}, \tag{6.10}$$

where $\boldsymbol{X}_1 = \boldsymbol{Z}_1\boldsymbol{L}_G$, $\boldsymbol{G} = \boldsymbol{L}_G\boldsymbol{L}_G^{\mathrm{T}}$ is the Cholesky factorization of $\boldsymbol{G}$, $\boldsymbol{B}_1 = \boldsymbol{L}_G^{-1}\boldsymbol{b}_1 \sim MN_{J\times n_T}(\mathbf{0}, \boldsymbol{I}_J, \boldsymbol{\Sigma}_T)$, and the specifications for the rest of parameters and prior distribution are the same as given in model (6.8). A Gibbs sampler implementation of this model is very similar to the one described before, with little modification. Indeed, a Gibbs implementation with the same multi-trait function is as follows:

```
LG= t(chol(G))
X1= Z1LG
ETA = list( list( X=X, model='FIXED' ), list( X=X1, model='BRR' ) )
A   =Multitrait(y = Y, ETA=ETA, resCov = list( type = 'UN', S0 = SR, df0 =
vR ), nIter = nI, burnIn = nb)
```

with the only change in the second sub-list predictor, where now the design matrix $\boldsymbol{X}_1$ and the Ridge regression model (**BRR**) are specified.

**Example 3** To illustrate the performance in terms of the prediction power of these models and how to implement this in R software, we considered a reduced data set that consisted of 50 wheat lines grown in two environments. In each individual, two traits were measured: FLRSDS and MIXTIM. The evaluation was done with a five-fold cross-validation, where lines were evaluated in some environments with all traits but are missing for all traits in other environments. Model (6.9) was fitted and the environment effect was assumed a fixed effect.

The results are shown in Table 6.4, where the average (standard deviation) of two performance criteria is shown for each trait in each environment: average Pearson's correlation (PC) and the mean squared error of prediction (MSE). Table 6.4 shows good correlation performance in both traits and in both environments, and better predictions were obtained in environment 2 with both criteria. The magnitude of the

**Table 6.4** Average Pearson's correlation (PC) and mean squared error of prediction (MSE) between predicted and observed values across five random partitions where lines were evaluated in some environments with all traits but are missing for all traits in other environments, SD represent standard deviation across partitions

| Env | Trait | PC (SD) | MSE (SD) |
|---|---|---|---|
| 1 | FLRSDS | 0.6252 (0.276) | 4.3009 (2.343) |
| | MIXTIM | 0.6709 (0.367) | 0.457 (0.343) |
| 2 | FLRSDS | 0.6895 (0.219) | 3.9553 (2.171) |
| | MIXTIM | 0.7523 (0.157) | 0.3764 (0.249) |

MSE in the first trait is mainly because the measurement scale is greater than in the second trait.

The R codes to reproduce these results (Table 6.4) are shown in Appendix 5.

## 6.9 Bayesian Genomic Multi-trait and Multi-environment Model (BMTME)

Model (6.9) does not take into account the possible trait–genotype–environment interaction ($T \times G \times E$), when environment information is available. An extension of this model is the one proposed by Montesinos-López et al. (2016), who added this interaction term to vary the specific trait genetic effects ($\boldsymbol{g}_j$) across environments. If the information of $n_{\mathrm{T}}$ traits of $J$ lines is collected in $I$ environments, this model is given by

$$\boldsymbol{Y} = \mathbf{1}_{IJ}\boldsymbol{\mu}^{\mathrm{T}} + \boldsymbol{XB} + \boldsymbol{Z}_1\boldsymbol{b}_1 + \boldsymbol{Z}_2\boldsymbol{b}_2 + \boldsymbol{E}, \tag{6.11}$$

where $=[\boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_{IJ}]^{\mathrm{T}}$, $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{IJ}]^{\mathrm{T}}$, $\boldsymbol{Z}_1$ and $\boldsymbol{Z}_2$ are the incident lines and the incident environment–line interaction matrices, $\boldsymbol{b}_1 = [\boldsymbol{g}_1, \ldots, \boldsymbol{g}_J]^{\mathrm{T}}$, $\boldsymbol{b}_2 = [\boldsymbol{g}_{21}, \ldots, \boldsymbol{g}_{2IJ}]^{\mathrm{T}}$, and $\boldsymbol{E} = [\boldsymbol{\epsilon}_1, \ldots, \boldsymbol{\epsilon}_{IJ}]^{\mathrm{T}}$. Here, $\boldsymbol{b}_2 \mid \boldsymbol{\Sigma}_T, \boldsymbol{\Sigma}_E \sim MN_{IJ\times n_T}(\boldsymbol{0}, \boldsymbol{\Sigma}_E \otimes \boldsymbol{G}, \boldsymbol{\Sigma}_T)$, and similar to model (6.2), $\boldsymbol{b}_1 \mid \boldsymbol{\Sigma}_T \sim MN_{J\times n_T}(\boldsymbol{0}, \boldsymbol{G}, \boldsymbol{\Sigma}_T)$ and $\boldsymbol{E} \sim MN_{IJ\times n_T}(\boldsymbol{0}, \boldsymbol{I}_{IJ}, \boldsymbol{R})$. The complete Bayesian specification of this model assumes independent multivariate normal distributions for the columns of $\boldsymbol{B}$, that is, for the fixed effect of each trait a prior multivariate normal distribution is adopted, $\boldsymbol{\beta}_t \sim N_p\left(\boldsymbol{\beta}_{t0}, \boldsymbol{\Sigma}_{\beta_t}\right), t = 1, \ldots, n_T$; a flat prior for the intercepts, $f(\boldsymbol{\mu}) \propto 1$; and independent inverse Wishart distributions for the covariance matrices of residuals $\boldsymbol{R}$ and for $\boldsymbol{\Sigma}_T$, $\boldsymbol{\Sigma}_T \sim \mathrm{IW}(v_T, \boldsymbol{S}_T)$ and $\boldsymbol{R} \sim \mathrm{IW}(v_R, \boldsymbol{S}_R)$, and also an inverse Wishart distribution for $\boldsymbol{\Sigma}_E$, $\boldsymbol{\Sigma}_E \sim \mathrm{IW}(v_E, \boldsymbol{S}_E)$.

The full conditional distributions of $\boldsymbol{\mu}, \boldsymbol{B}, \boldsymbol{b}_1, \boldsymbol{b}_2$, and $\boldsymbol{R}$ can be derived as in model (6.9). The full conditional distribution of $\boldsymbol{\Sigma}_T$ is

$$\begin{aligned}
f(\boldsymbol{\Sigma}_T|-) &\propto f(\boldsymbol{b}_1|\boldsymbol{\Sigma}_T)P(\boldsymbol{b}_2|\boldsymbol{\Sigma}_T, \boldsymbol{\Sigma}_E)P(\boldsymbol{\Sigma}_T) \\
&\propto |\boldsymbol{\Sigma}_T|^{-\frac{J}{2}}|\boldsymbol{G}|^{-\frac{L}{2}}\exp\left\{-\frac{1}{2}\mathrm{tr}\left[\boldsymbol{b}_1^{\mathrm{T}}\boldsymbol{G}^{-1}\boldsymbol{b}_1\boldsymbol{\Sigma}_T^{-1}\right]\right\} \\
&\quad \times |\boldsymbol{\Sigma}_T|^{-\frac{IJ}{2}}|\boldsymbol{\Sigma}_E \otimes \boldsymbol{G}|^{-\frac{n_T}{2}}\exp\left\{-\frac{1}{2}\mathrm{tr}\left[\boldsymbol{b}_2^{\mathrm{T}}\left(\boldsymbol{\Sigma}_E^{-1} \otimes \boldsymbol{G}^{-1}\right)\boldsymbol{b}_2\boldsymbol{\Sigma}_T^{-1}\right]\right\}P(\boldsymbol{\Sigma}_T) \\
&\propto \boldsymbol{\Sigma}_T^{-\frac{v_T+J+IJ+n_T+1}{2}}\exp\left\{-\frac{1}{2}\mathrm{tr}\left(\boldsymbol{b}_1^{\mathrm{T}}\boldsymbol{G}^{-1}\boldsymbol{b}_1 + \boldsymbol{b}_2^{\mathrm{T}}\left(\boldsymbol{\Sigma}_E^{-1} \otimes \boldsymbol{G}^{-1}\right)\boldsymbol{b}_2 + \boldsymbol{S}_t\right)\boldsymbol{\Sigma}_T^{-1}\right\},
\end{aligned}$$

that is, $\boldsymbol{\Sigma}_T \mid - \sim \mathrm{IW}\left(v_T + J + IJ, \boldsymbol{b}_1^{\mathrm{T}}\boldsymbol{G}^{-1}\boldsymbol{b}_1 + \boldsymbol{b}_2^{\mathrm{T}}\left(\boldsymbol{\Sigma}_E^{-1} \otimes \boldsymbol{G}^{-1}\right)\boldsymbol{b}_2 + \boldsymbol{S}_T\right)$.

Now, let be $\boldsymbol{b}_2^*$ a $Jn_T \times I$ matrix such that $\text{vec}\left(\boldsymbol{b}_2^{\text{T}}\right) = \text{vec}\left(\boldsymbol{b}_2^*\right)$. Then because $\boldsymbol{b}_2^{\text{T}} \mid \boldsymbol{\Sigma}_T, \boldsymbol{\Sigma}_E \sim MN_{n_T \times IJ}(\boldsymbol{0}, \boldsymbol{\Sigma}_T, \boldsymbol{\Sigma}_E \otimes \boldsymbol{G})$ , $\text{vec}\left(\boldsymbol{b}_2^{\text{T}}\right) \mid \boldsymbol{\Sigma}_T, \boldsymbol{\Sigma}_E \sim N(\boldsymbol{0}, \boldsymbol{\Sigma}_E \otimes (\boldsymbol{G} \otimes \boldsymbol{\Sigma}_{\text{T}}))$, and $\boldsymbol{b}_2^* \mid \boldsymbol{\Sigma}_T, \boldsymbol{\Sigma}_E \sim MN_{n_T \times IJ}(\boldsymbol{0}, \boldsymbol{G} \otimes \boldsymbol{\Sigma}_{\text{T}}, \boldsymbol{\Sigma}_E)$, the full conditional posterior distribution of $\boldsymbol{\Sigma}_E$ is

$$
\begin{aligned}
P(\boldsymbol{\Sigma}_E|\text{ELSE}) &\propto P(\boldsymbol{b}_2|\boldsymbol{\Sigma}_E)\ P(\boldsymbol{\Sigma}_E) \\
&\propto |\boldsymbol{\Sigma}_E|^{-\frac{JL}{2}}|\boldsymbol{G} \otimes \boldsymbol{\Sigma}_T|^{-\frac{I}{2}} \exp\left\{-\frac{1}{2}\text{tr}\left[\boldsymbol{\Sigma}_E^{-1}\boldsymbol{b}_2^{*\text{T}}\left(\boldsymbol{G}^{-1} \otimes \boldsymbol{\Sigma}_T^{-1}\right)\boldsymbol{b}_2^*\right]\right\} \\
&\quad |\boldsymbol{S}_E|^{\frac{v_E+I-1}{2}} \times |\boldsymbol{\Sigma}_E|^{-\frac{v_E}{2}} \exp\left\{-\frac{1}{2}\text{tr}\left(\boldsymbol{S}_E\boldsymbol{\Sigma}_E^{-1}\right)\right\} \\
&\propto |\boldsymbol{\Sigma}_E|^{-\frac{v_E+JL+I+1}{2}} \exp\left\{-\frac{1}{2}\text{tr}\left[\left(\boldsymbol{b}_2^{*\text{T}}\left(\boldsymbol{G}^{-1} \otimes \boldsymbol{\Sigma}_T^{-1}\right)\boldsymbol{b}_2^* + \boldsymbol{S}_E\right)\right]\boldsymbol{\Sigma}_E^{-1}\right\}
\end{aligned}
$$

which means that $\boldsymbol{\Sigma}_E \mid - \sim \text{IW}\left(v_E + JL, \boldsymbol{b}_2^{*\text{T}}\left(\boldsymbol{G}^{-1} \otimes \boldsymbol{\Sigma}_T^{-1}\right)\boldsymbol{b}_2^* + \boldsymbol{S}_E\right)$.

A Gibbs sampler to explore the joint posterior distribution of parameters of model (6.11), $\boldsymbol{\mu}$, $\boldsymbol{\beta}$, $\boldsymbol{b}_1$,$\boldsymbol{b}_2$, $\boldsymbol{\Sigma}_T$, $\boldsymbol{\Sigma}_E$, and $\boldsymbol{R}$, can be implemented with the following steps:

1. Simulate $\boldsymbol{\beta}$ from a multivariate normal distribution $N_p\left(\widetilde{\boldsymbol{\beta}}_0, \widetilde{\boldsymbol{\Sigma}}_{\boldsymbol{\beta}}\right)$, where $\widetilde{\boldsymbol{\Sigma}}_{\boldsymbol{\beta}} = \left[\boldsymbol{\Sigma}_{\beta}^{-1} + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{X}^{\text{T}}\boldsymbol{X}\right)\right]^{-1}$ and $\widetilde{\boldsymbol{\beta}}_0 = \widetilde{\boldsymbol{\Sigma}}_{\boldsymbol{\beta}}\left[\boldsymbol{\Sigma}_{\beta}^{-1}\boldsymbol{\beta}_0 + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{X}^{\text{T}}\right)\right.$ $\left.\text{vec}(\boldsymbol{Y} - \boldsymbol{1}_{IJ}\boldsymbol{\mu}^{\text{T}} - \boldsymbol{Z}_1\boldsymbol{b}_1 - \boldsymbol{Z}_2\boldsymbol{b}_2)\right]$.
2. Simulate $\boldsymbol{\mu}$ from $N_{n_T}\left(\widetilde{\boldsymbol{\mu}}, \widetilde{\boldsymbol{\Sigma}}_{\mu}\right)$, where $\widetilde{\boldsymbol{\Sigma}}_{\mu} = (IJ)^{-1}\boldsymbol{R}$ and $\widetilde{\boldsymbol{\mu}} = \widetilde{\boldsymbol{\Sigma}}_{\mu}\left(\boldsymbol{R}^{-1} \otimes \boldsymbol{1}_{IJ}\right)$ $\text{vec}(\boldsymbol{Y} - \boldsymbol{XB} - \boldsymbol{Z}_1\boldsymbol{b}_1 - \boldsymbol{Z}_2\boldsymbol{b}_2)$.
3. Simulate $\boldsymbol{g}_1 = \text{vec}(\boldsymbol{b}_1)$ from $N_J\left(\widetilde{\boldsymbol{g}}_1, \widetilde{\boldsymbol{G}}\right)$, where $\widetilde{\boldsymbol{G}} = \left[\left(\boldsymbol{\Sigma}_T^{-1} \otimes \boldsymbol{G}^{-1}\right) + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{Z}_1^{\text{T}}\boldsymbol{Z}_1\right)\right]^{-1}$ and $\widetilde{\boldsymbol{g}} = \widetilde{\boldsymbol{G}}\left(\boldsymbol{R}^{-1} \otimes \boldsymbol{Z}_1^{\text{T}}\right)\text{vec}\left(\boldsymbol{Y} - \boldsymbol{1}_{IJ}\boldsymbol{\mu}^{\text{T}} - \boldsymbol{XB} - \boldsymbol{Z}_2\boldsymbol{b}_2\right)$.
4. Simulate $\boldsymbol{g}_2 = \text{vec}(\boldsymbol{b}_2)$ from $N_J\left(\widetilde{\boldsymbol{g}}_2, \widetilde{\boldsymbol{G}}_2\right)$, where $\widetilde{\boldsymbol{G}}_2 = \left[\left(\boldsymbol{\Sigma}_T^{-1} \otimes \boldsymbol{\Sigma}_E^{-1} \otimes \boldsymbol{G}^{-1}\right) + \left(\boldsymbol{R}^{-1} \otimes \boldsymbol{Z}_2^{\text{T}}\boldsymbol{Z}_2\right)\right]^{-1}$ and $\widetilde{\boldsymbol{g}}_2 = \widetilde{\boldsymbol{G}}_2\left(\boldsymbol{R}^{-1} \otimes \boldsymbol{Z}_2^{\text{T}}\right)\text{vec}\left(\boldsymbol{Y} - \boldsymbol{1}_{IJ}\boldsymbol{\mu}^{\text{T}} - \boldsymbol{XB} - \boldsymbol{Z}_1\boldsymbol{b}_1\right)$.
5. Simulate $\boldsymbol{\Sigma}_T$ from $\text{IW}\left(v_T + J + IJ, \boldsymbol{b}_1^{\text{T}}\boldsymbol{G}^{-1}\boldsymbol{b}_1 + \boldsymbol{b}_2^{\text{T}}\left(\boldsymbol{\Sigma}_E^{-1} \otimes \boldsymbol{G}^{-1}\right)\boldsymbol{b}_2 + \boldsymbol{S}_T\right)$.
6. Simulate $\boldsymbol{\Sigma}_E$ from $\text{IW}\left(v_E + JL, \boldsymbol{b}_2^{*\text{T}}\left(\boldsymbol{G}^{-1} \otimes \boldsymbol{\Sigma}_T^{-1}\right)\boldsymbol{b}_2^* + \boldsymbol{S}_E\right)$.
7. Simulate $\boldsymbol{R}$ from $\text{IW}\left(\widetilde{v}_R, \widetilde{\boldsymbol{S}}_R\right)$, where $\widetilde{v}_R = v_R + IJ$ and $\widetilde{\boldsymbol{S}}_R = \boldsymbol{S}_T + \left(\boldsymbol{Y} - \boldsymbol{1}_{IJ}\boldsymbol{\mu}^{\text{T}} - \boldsymbol{XB} - \boldsymbol{Z}_1\boldsymbol{b}_1 - \boldsymbol{Z}_2\boldsymbol{b}_2\right)^{\text{T}}\left(\boldsymbol{Y} - \boldsymbol{1}_{IJ}\boldsymbol{\mu}^{\text{T}} - \boldsymbol{X\beta} - \boldsymbol{Z}_1\boldsymbol{b}_1 - \boldsymbol{Z}_2\boldsymbol{b}_2\right)$.
8. Return to step 1 or terminate when chain length is adequate to meet convergence diagnostics and the required sample size is reached.

A similar Gibbs sampler is implemented in the BMTME R package, with the main difference, that this package does not allow specifying a general fixed effect design matrix $\boldsymbol{X}$, only the corresponding to the design matrix for the environment effects, and also the intercept vector $\boldsymbol{\mu}$ is ignored because it is included in the fixed

environment effects. Specifically, to fit model (6.11) where the only fixed effects to be taken into account are the environment's effects, the R code to implement this with the BMTME package is as follows:

```
XE   = model.matrix(~Env,data=dat_F)
Z1   = model.matrix(~0+GID,data=dat_F)
Lg   = t(chol(G))
Z1_a = Z1%*%Lg
Z2   = model.matrix(~0+GID:Env,data=dat_F)
G2   = kronecker(diag(dim(XE)[2])),Lg)
Z2 _a = Z2%*%G2
A = BMTME(Y = Y, X = XE, Z1 = Z1_a, Z2 = Z2_a, nIter = nI, burnIn = nb, thin =
2, bs = 50)
```

where **Y** is the matrix of response variables where each row corresponds to the measurement of $n_T$ traits in each individual, **XE** is a design matrix for the environment effects, **Z1** is the incidence matrix of the genetics effects, **Z2** is the design matrix of the genetic–environment interaction effects, **nI** and **nb** are the required number of interactions and the burn-in period, and *bs* is the number of blocks to use internally to sample from vec($\boldsymbol{b}_2$).

**Example 4** To illustrate how to implement this model with the BMTME R package, we considered the data in Example 2, but now the explored model includes the trait–genotype–environment interaction.

The average results of the prediction performance in terms of PC and MSE for implementing the same five-fold cross-validation used in Example 3 are shown in Table 6.5. These results show an improvement in terms of prediction performance with this model in all trait environments combinations and in both criteria (PC and MSE) to measure the prediction performance, except in trait MIXTIM and Env 2, where the MSE is slightly greater than the one obtained with model (6.9), which does not take into account the triple interaction (trait–genotype–environment).

The R code used to obtained these results is given in Appendix 5.

**Table 6.5** Average Pearson's correlation (PC) and mean squared error of prediction (MSE) between predicted and observed values across five random partitions where lines were evaluated in some environments with all traits but are missing for all traits in other environments

| Env | Trait | PC (SD) | MSE (SD) |
|---|---|---|---|
| 1 | FLRSDS | 0.668 (0.243) | 3.4483 (1.9) |
| | MIXTIM | 0.6913 (0.35) | 0.4255 (0.325) |
| 2 | FLRSDS | 0.7218 (0.219) | 3.2304 (2.034) |
| | MIXTIM | 0.7667 (0.188) | 0.3806 (0.287) |

## Appendix 1

- The probability density function (pdf) of the scaled inverse Chi-square distribution with $v$ degrees of freedom and scale parameter $S$, $\chi^{-2}(v,S)$, is given by

$$f\left(\sigma^2;v,S\right)=\frac{\left(\frac{S}{2}\right)^{\frac{v}{2}}}{\Gamma\left(\frac{v}{2}\right)\left(\sigma^2\right)^{1+\frac{v}{2}}}\exp\left(-\frac{S}{2\sigma^2}\right).$$

and the mean, mode, and variance of this distribution are given by $\frac{S}{v-2}$, $\frac{S}{v+2}$, and $\frac{2S^2}{(v-2)^2(v-4)}$, respectively.

- The pdf of the gamma distribution with shape parameter $s$ and rate parameter $r$:

$$f_{S_\beta}(x;s,r)=\frac{r^s x^{s-1}}{\Gamma(s)}\exp\left(-rx\right).$$

The mean, mode, and variance of this distribution are $s/r$, $(s-1)/r$, and $s/r^2$, respectively.

- The pdf of a beta distribution with mean $\mu$ and precision parameter $\phi$ ("dispersion" parameter $\phi^{-1}$) is given by

$$f(x;\mu,\phi)=\frac{1}{B[\mu\phi,(1-\mu)\phi]}x^{\mu\phi-1}(1-x)^{(1-\mu)\phi-1},$$

where the relation with the standard parameterization of this distribution, Beta $(\alpha,\beta)$, is

$$\mu=\frac{\alpha}{\alpha+\beta},\phi=\alpha+\beta.$$

- The pdf of a Laplace distribution is

$$f(x;b)=\frac{1}{2b}\exp\left(-\frac{|x|}{b}\right),b>0,\,-\infty\leq x\leq\infty.$$

The mean and variance of this distribution are 0 and $2b^2$.

- A random matrix $\boldsymbol{\Sigma}$ of dimension $p\times p$ is distributed as inverse Wishart distribution with parameter $v$ and $\boldsymbol{S}$, $\boldsymbol{\Sigma}\sim\mathrm{IW}(v,\boldsymbol{S})$, if it has a density function

$$f(\boldsymbol{\Sigma})=\frac{1}{2^{\frac{vp}{2}}\Gamma_p(v/2)}|\boldsymbol{S}|^{\frac{v}{2}}|\boldsymbol{\Sigma}|^{-\frac{v+p+1}{2}}\exp\left[-\frac{1}{2}\mathrm{tr}\left(\boldsymbol{S}\boldsymbol{\Sigma}^{-1}\right)\right],$$

where $\Gamma_p(v/2)$ is the multivariate gamma function, $v>0$, and $\boldsymbol{\Sigma}$ and $\boldsymbol{S}$ are positive defined matrices. The mean matrix of this distribution is $\frac{\boldsymbol{S}}{v-p-1}$.

- A $(p \times q)$ random matrix $\boldsymbol{Z}$ follows the matrix normal distribution with matrix parameters $\boldsymbol{M}$ $(p \times q)$, $\boldsymbol{U}$ $(p \times p)$, and $\boldsymbol{V}(q \times q)$, $\boldsymbol{Z} \sim MN_{p,\,q}(\boldsymbol{M}, \boldsymbol{U}, \boldsymbol{V})$, if it has density

$$f(\boldsymbol{Z}|\boldsymbol{M}, \boldsymbol{U}, \boldsymbol{V}) = \frac{\exp\left\{-\frac{1}{2}\mathrm{tr}\left[\boldsymbol{V}^{-1}(\boldsymbol{Z}-\boldsymbol{M})^{\mathrm{T}}\boldsymbol{U}^{-1}(\boldsymbol{Z}-\boldsymbol{M})\right]\right\}}{(2\pi)^{pq/2}|\boldsymbol{V}|^{p/2}|\boldsymbol{U}|^{q/2}}.$$

## Appendix 2: Setting Hyperparameters for the Prior Distributions of the BRR Model

The following rules are those used in Pérez and de los Campos (2014), and provide proper but weakly informative prior distributions. In general, this consists of assigning a certain proportion of the total variance of the phenotypes, to the different components of the model.

Specifically, for model (6.3), first the total variance of $y$ is partitioned into two components: (1) the error and (2) the linear predictor:

$$\mathrm{Var}\left(y_j\right) = \mathrm{Var}\left(\boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{\beta}_0\right) + \sigma^2$$

Therefore, the average of the variance of the individuals, called total variance, is equal to

$$\frac{1}{n}\sum_{j=1}^{n}\mathrm{Var}\left(y_j\right) = \frac{1}{n}\sum_{j=1}^{n}\mathrm{Var}\left(\boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{\beta}_0\right) + \sigma^2 = \frac{1}{n}\mathrm{tr}\left(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\right)\sigma_\beta^2 + \sigma^2 = V_M + V_\epsilon.$$

Then, by setting $R_1^2$ as the proportion of the total variance ($\mathbf{V}_y$), that is explained by markers a priori, $V_M = R_1^2\mathbf{V}_y$, and replacing $\sigma_\beta^2$ in $V_M$ by its prior mode, $\frac{S_\beta}{v_\beta+2}$, we have that

$$\frac{1}{n}\mathrm{tr}\left(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\right)\left(\frac{S_\beta}{v_\beta+2}\right) = R_1^2\mathbf{V}_y.$$

From here, once we have set a value for $v_\beta$, the scale parameter is given by

$$S_\beta = \frac{R_1^2\mathbf{V}_y}{\frac{1}{n}\mathrm{tr}\left(\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}}\right)}\left(v_\beta + 2\right).$$

A commonly used value of the shape parameter is $v_\beta = 5$ and the value for the proportion of explained variances is $R_1^2 = 0.5$.

Because the model only has two predictors and $R_1^2$ was set as the proportion of the total variance that is explained by markers a priori, the corresponding proportion that is explained by error a priori is $R_2^2 = 1 - R_1^2$. Then, similar to what was done before, once there is a value for the shape parameter of the prior distribution of $\sigma^2$, $v$, the value of the scale parameter is given by

$$S = \left(1 - R_1^2\right)\mathbf{V}_y(v + 2).$$

By default, $v = 5$ is often used.

## Appendix 3: R Code Example 1

```
rm(list=ls())
library(BGLR)
load('dat_ls_E1.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
#Marker data
dat_M = dat_ls$dat_M
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F,5)

#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
XM = scale(XM)
dim(XM)

n = dim(dat_F)[1]
y = dat_F$y

#10 random partitions
K = 10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

#BRR
ETA_BRR = list(list(model='BRR',X=XM))
Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
for(k in 1:K)
{
```

```
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_BRR,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP[k] = mean((y[Pos_tst]-yp_ts)^2)
}

#GBLUP
dat_M = scale(dat_M)
G = tcrossprod(XM)/dim(XM)[2]
dim(G)
#Matrix design of GIDs
Z = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
K_L = Z%*%G%*%t(Z)
ETA_GB = list(list(model='RKHS',K = K_L))
#Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_GB,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_GB[k] = mean((y[Pos_tst]-yp_ts)^2)
}

#BA
ETA_BA = list(list(model='BayesA',X=XM))
#Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_BA,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_BA[k] = mean((y[Pos_tst]-yp_ts)^2)
}

#BB
ETA_BB = list(list(model='BayesB',X=XM))
#Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_BB,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
```

```
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_BB[k] = mean((y[Pos_tst]-yp_ts)^2)
}

#BC
ETA_BC = list(list(model='BayesC',X=XM))
#Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_BC,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_BC[k] = mean((y[Pos_tst]-yp_ts)^2)
}

#BL
ETA_BL = list(list(model='BL',X=XM))
#Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_BL,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_BL[k] = mean((y[Pos_tst]-yp_ts)^2)
}
Tab

#Mean and SD across the five partitions
apply(Tab[,-1],2,function(x)c(mean(x),sd(x)))
```

## Appendix 4: R Code Example 2

```
rm(list=ls())
library(BGLR)
library(BMTME)
load('dat_ls_E2.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
dim(dat_F)
#Marker data
dat_M = dat_ls$dat_M
dim(dat_M)
```

```
dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F,5)

#Matrix design for markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
dim(XM)
XM = scale(XM)
#Environment design matrix
XE = model.matrix(~0+Env,data=dat_F)[,-1]
head(XE)
#Environment-marker design matrix
XEM = model.matrix(~0+XM:XE)
#GID design matrix and Environment-GID design matrix
#for RKHS models
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
Z_LE = model.matrix(~0+GID:Env,data=dat_F,
         xlev = list(GID=unique(dat_F$GID),Env = unique(dat_F$Env)))
#Genomic relationship matrix derived from markers
dat_M = scale(dat_M)
G = tcrossprod(dat_M)/dim(dat_M)[2]
dim(G)
#Covariance matrix for Zg
K_L = Z_L%*%G%*%t(Z_L)
#Covariance matrix for random effects ZEg
K_LE = Z_LE%*%kronecker(diag(4),G)%*%t(Z_LE)
n = dim(dat_F)[1]
y = dat_F$y

#Number of random partitions
K = 5
PT = CV.KFold(dat_F,DataSetID = 'GID', K=5, set_seed = 1)
Models = c('BRR','RKHS','BayesA','BayesB','BayesC','BL')
Tab = data.frame()
for(m in 1:6)
{
 ETA1 = list(list(model=Models[m],X=XM))
 ETA2 = list(list(model='FIXED',X=XE),list(model=Models[m],X=XM))
 ETA3 = list(list(model='FIXED',X=XE),list(model=Models[m],X=XM),
       list(model=Models[m],X=XEM))
 if(Models[m]=='RKHS')
 {
  ETA1 = list(list(model='RKHS',K=K_L))
  ETA2 = list(list(model='FIXED',X=XE),list(model='RKHS',K=K_L))
  ETA3 = list(list(model='FIXED',X=XE),list(model='RKHS',K=K_L),
        list(model='RKHS',K=K_LE))
 }
 Tab1_m = data.frame(PT = 1:K,MSEP = NA)
 set.seed(1)
 Tab2_m = Tab1_m
 Tab3_m = Tab2_m
 for(k in 1:K)
```

```
 {
  Pos_tst = PT$CrossValidation_list[[k]]
  y_NA = y
  y_NA[Pos_tst] = NA
  A = BGLR(y=y_NA,ETA=ETA1,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_m$MSEP[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_m$Cor[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA2,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab2_m$MSEP[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab2_m$Cor[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA3,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab3_m$MSEP[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab3_m$Cor[k]  = cor(y[Pos_tst],yp_ts)
 }
 Tab = rbind(Tab,data.frame(Model=Models[m],Tab1_m,Tab2_m,Tab3_m))
}
Tab
```

# Appendix 5

## *R Code Example 3*

```
rm(list=ls(all=TRUE))
library(BGLR)
library(BMTME)
library(dplyr)

load('dat_ls.RData',verbose=TRUE)
dat_F = dat_ls$dat_F
head(dat_F)
Y = as.matrix(dat_F[,-(1:2)])
dat_F$Env = as.character(dat_F$DS)
G = dat_ls$G
J = dim(G)[1]
XE = matrix(model.matrix(~0+Env,data=dat_F)[,-1],nc=1)
Z  = model.matrix(~0+GID,data=dat_F)
K = Z%*%G%*%t(Z)
#Partitions for a 5-FCV
PT_ls = CV.KFold(dat_FF, DataSetID='GID',K=5,set_seed = 123)
PT_ls = PT_ls$CrossValidation_list
#Predictor BGLR
ETA = list(list(X=XE,model='FIXED'),list(K=K,model='RKHS'))
#Function to summarize the performance prediction: PC_MM_f
source('PC_MM.R')#See below
Tab = data.frame()
```

```
set.seed(1)
for(p in 1:5)
{
 Y_NA = Y
 Pos_NA = PT_ls[[p]]
 Y_NA[Pos_NA,] = NA
 A = Multitrait(y = Y_NA, ETA=ETA,
          resCov = list( type = 'UN', S0 = diag(2), df0 = 5 ),
          nIter = 5e3, burnIn = 1e3)
 PC = PC_MM_f(Y[Pos_NA,],A$yHat[Pos_NA,],Env=dat_F$Env[Pos_NA])
 Tab = rbind(Tab,data.frame(PT=p,PC))
 cat('PT=',p,'\n')
}
Tab_R = Tab%>%group_by(Env,Trait)%>%select(Cor,MSEP)%>%summarise
(Cor_mean = mean(Cor),
                              Cor_sd  = sd(Cor),
                              MSEP_mean = mean(MSEP),
                              MSEP_sd = sd(MSEP))
Tab_R = as.data.frame(Tab_R)
Tab_R
#Save in the same folder the following in a file with name "PC_MM.R"
#Performance criteria
PC_MM_f<-function(y,yp,Env=NULL)
{
 if(is.null(Env))
 {
  Cor  = diag(cor(as.matrix(y),as.matrix(yp)))
  MSEP  = colMeans((y-yp)^2)
  PC = data.frame(Trait = colnames(y),Cor=Cor, MSEP=MSEP)
 }
 else
 {
  PC = data.frame()
  Envs = unique(Env)
  nE = length(Envs)
  for(e in 1:nE)
  {
   y_e = y[Env==Envs[e],]
   yp_e = yp[Env==Envs[e],]
   Cor  = diag(cor(as.matrix(y_e),as.matrix(yp_e)))
   MSEP  = colMeans((y_e-yp_e)^2)
   PC = rbind(PC,data.frame(Trait = colnames(y),Env=Envs[e],Cor=Cor,
MSEP=MSEP))
  }
 }
 PC
}
```

## *R Code for Example 4*

```
rm(list=ls(all=TRUE))
library(BMTME)
library(dplyr)
load('dat_ls.RData',verbose=TRUE)
dat_F = dat_ls$dat_F
head(dat_F)
Y = as.matrix(dat_F[,-(1:2)])
dat_F$Env = as.character(dat_F$DS)
G = dat_ls$G
Lg = t(chol(G))
XE = model.matrix(~Env,data=dat_F)
Z1 = model.matrix(~0+GID,data=dat_F)
Z1_a = Z1%*%Lg
Z2 = model.matrix(~0+GID:Env,data=dat_F)
L2 = kronecker(diag(dim(XE)[2]),Lg)
Z2_a = Z2%*%L2
#Partitions for a 5-FCV
PT_ls = CV.KFold(dat_FF, DataSetID='GID',K=5,set_seed = 123)
PT_ls = PT_ls$CrossValidation_list
source('PC_MM.R')#See file R "PC_MM.R" defined in example 6.1
Tab = data.frame()
set.seed(1)
for(p in 1:5)
{
 Y_NA = Y
 Pos_NA = PT_ls[[p]]
 Y_NA[Pos_NA,] = NA
 A = BMTME(Y = Y_NA, X = XE, Z1 = Z1_a, Z2 = Z2_a,
          nIter = 3e3, burnIn = 5e2, thin = 2, bs = 50)
 PC = PC_MM_f(Y[Pos_NA,],A$yHat[Pos_NA,],Env=dat_F$Env[Pos_NA])
 Tab = rbind(Tab,data.frame(PT=p,PC))
 cat('PT=',p,'\n')
}
Tab_R = Tab%>%group_by(Env,Trait)%>%select(Cor,MSEP)%>%summarise
(Cor_mean = mean(Cor),
                              Cor_sd  = sd(Cor),
                              MSEP_mean = mean(MSEP),
                              MSEP_sd = sd(MSEP))
Tab_R = as.data.frame(Tab_R)
Tab_R
```

## References

Box GEP, Tiao GC (1992) Bayesian inference in statistical analysis. Wiley, New York
Calus MP, Veerkamp RF (2011) Accuracy of multi-trait genomic selection using different methods. Genet Sel Evol 43(1):26
Casella G, George EI (1992) Explaining the Gibbs sampler. Am Stat 46(3):167–174
Christensen R, Johnson W, Branscum A, Hanson TE (2011) Bayesian ideas and data analysis: an introduction for scientists and statisticians. Chapman & Hall/CRC, Stanford, CA
de los Campos G, Hickey JM, Pong-Wong R, Daetwyler HD, Calus MP (2013) Whole-genome regression and prediction methods applied to plant and animal breeding. Genetics 193 (2):327–345
Gelman A, Carlin JB, Stern HS, Dunson DB, Vehtari A, Rubin DB (2013) Bayesian data analysis. Chapman and Hall/CRC, Stanford, CA
Habier D, Tetens J, Seefried FR, Lichtner P, Thaller G (2010) The impact of genetic relationship information on genomic breeding values in German Holstein cattle. Genet Sel Evol 42(1):5
Habier D, Fernando RL, Kizilkaya K, Garrick DJ (2011) Extension of the Bayesian alphabet for genomic selection. BMC Bioinformatics 12(1):186
Henderson C (1975) Best linear unbiased estimation and prediction under a selection model. Biometrics 31(2):423–447. https://doi.org/10.2307/2529430
Henderson CR, Quaas RL (1976) Multiple trait evaluation using relative's records. J Anim Sci 43: 11–88
Jia Y, Jannink JL (2012) Multiple-trait genomic selection methods increase genetic value prediction accuracy. Genetics 192(4):1513–1522
Jiang J, Zhang Q, Ma L, Li J, Wang Z, Liu JF (2015) Joint prediction of multiple quantitative traits using a Bayesian multivariate antedependence model. Heredity 115(1):29–36
Lehermeier C, Wimmer V, Albrecht T, Auinger HJ, Gianola D, Schmid VJ, Schön CC (2013) Sensitivity to prior specification in Bayesian genome-based prediction models. Stat Appl Genet Mol Biol 12(3):375–391
Meuwissen TH, Hayes BJ, Goddard ME (2001) Prediction of total genetic value using genome-wide dense marker maps. Genetics 157(4):1819–1829
Montesinos-López OA, Montesinos-López A, Crossa J, Toledo FH, Pérez-Hernández O, Eskridge KM, Rutkoski J (2016) A genomic Bayesian multi-trait and multi-environment model. G3 6 (9):2725–2744
Montesinos-López OA, Montesinos-López A, Montesinos-López JC, Crossa J, Luna-Vázquez FJ, Salinas-Ruiz J (2018) A Bayesian multiple-trait and multiple-environment model using the matrix normal distribution. In: Physical methods for stimulation of plant and mushroom development. IntechOpen, Croatia, p 19
Park T, Casella G (2008) The Bayesian lasso. J Am Stat Assoc 103(482):681–686
Pérez P, de los Campos G (2013) BGLR: a statistical package for whole genome regression and prediction. R package version 1(0.2)
Pérez P, de los Campos G (2014) Genome-wide regression and prediction with the BGLR statistical package. Genetics 198(2):483–495
Pérez P, de los Campos G, Crossa J, Gianola D (2010) Genomic-enabled prediction based on molecular markers and pedigree using the Bayesian linear regression package in R. Plant Genome 3(2):106–116
Pollak EJ, Van der Werf J, Quaas RL (1984) Selection bias and multiple trait evaluation. J Dairy Sci 67(7):1590–1595
Pszczola M, Veerkamp RF, De Haas Y, Wall E, Strabel T, Calus MPL (2013) Effect of predictor traits on accuracy of genomic breeding values for feed intake based on a limited cow reference population. Animal 7(11):1759–1768
Schaeffer LR (1984) Sire and cow evaluation under multiple trait models. J Dairy Sci 67 (7):1567–1580
VanRaden PM (2007) Genomic measures of relationship and inbreeding. Interbull Bull 7:33–36

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 7
# Bayesian and Classical Prediction Models for Categorical and Count Data

## 7.1 Introduction

Categorical and count response variables are common in many plant breeding programs. For this reason, in this chapter, the fundamental and practical issues for implementing genomic prediction models with these types of response variables are provided. Also, current open source software is used for its implementation. The prediction models proposed here are Bayesian and classical models. For each type of these models, we provide the fundamental principles and concepts, whereas their practical implementation is illustrated with examples in the genomic selection context. Taken into account are the most common terms in the predictor, such as the main effects of environments and genotypes, and interaction terms like genotype $\times$ environment interaction and marker $\times$ environment interaction. First, the models under the Bayesian framework are presented and then those under a classical framework. Next, the fundamentals and applications of the Bayesian ordinal regression model are provided.

## 7.2 Bayesian Ordinal Regression Model

In many applications, the response ($Y$) is not continuous, but rather multi-categorical in nature: ordinal (the categories of the response can be ordered, 1, . . ., $C$) or nominal (the categories have no order). For example, in plant breeding, categorical scores for resistance and susceptibility are often collected [disease: 1 (no disease), 2 (low infection), 3 (moderate infection), 4 (high infection), and 5 (complete infection)]. Another example is in animal breeding, where calving ease is scored as 1, 2, or 3 to indicate normal birth, slight difficult birth, or extreme difficult birth, respectively. The categorical traits are assigned to one of a set of mutually exclusive and exhaustive response values.

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_7

It is well documented that linear regression models in this situation are difficult to justify (Gianola 1980, 1982; Gianola and Foulley 1983) and inadequate results can be obtained, unless there is a large number of categories and the data seem to show a distribution that is approximately normal (Montesinos-López et al. 2015b).

For the ordinal model, we appeal to the existence of a latent continuous random variable and the categories are conceived as contiguous intervals on the continuous scale, as presented in McCullagh (1980), where the points of division (thresholds) are denoted as $\gamma_0, \gamma_1, \ldots, \gamma_C$. In this way, the ordinal model assumes that conditioned to $\boldsymbol{x}_i$ (covariates of dimension $p$), $Y_i$ is a random variable that takes values $1, \ldots, C$, with the following probabilities:

$$\begin{aligned} p_{ic} &= P(Y_i = c) = P(\gamma_{c-1} \leq L_i \leq \gamma_c) \\ &= F\left(\gamma_c + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) - F\left(\gamma_{c-1} + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right), \ \ c = 1, \ldots, C, \end{aligned} \tag{7.1}$$

where $L_i = -\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} + \epsilon_i$ is a latent variable, $\epsilon_i$ is a random error term with cumulative distribution function $F$, $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_p)^{\mathrm{T}}$ denotes the parameter vector associated with the effects of the covariates, and $-\infty = \gamma_0 < \gamma_1 < \ldots < \gamma_C = \infty$ are threshold parameters and also need to be estimated, that is, the parameter vector to be estimated in this model is $\boldsymbol{\theta} = \left(\boldsymbol{\beta}^{\mathrm{T}}, \boldsymbol{\gamma}^{\mathrm{T}}, \sigma_\beta^2\right)^{\mathrm{T}}$, where $\boldsymbol{\gamma} = (\gamma_1, \ldots, \gamma_{C-1})^{\mathrm{T}}$.

All the ordinal models presented in this chapter share the property that the categories can be considered as contiguous intervals on some continuous scale, but they differ in their assumptions regarding the distributions of the latent variable. When F is the standard logistic distribution function, the resulting model is known as the **logistic** ordinal model:

$$p_{ic} = F\left(\gamma_c + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) - F\left(\gamma_{c-1} + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right), \ \ c = 1, \ldots, C,$$

where $F(z) = \frac{1}{1+\exp(-z)}$. When F is the standard normal distribution function, the resulting model is the ordinal **probit** model:

$$p_{ic} = F\left(\gamma_c + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) - F\left(\gamma_{c-1} + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right), \ \ c = 1, \ldots, C,$$

where $F(z) = \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{u^2}{2}\right) du$. When the response value only takes two values, model (7.1) is reduced to the binary regression model, which in the first case is better known as logistic regression, while in the second case, it is known as the probit regression model. For this reason, logistic regression or probit regression is a particular case of ordinal regression even under a Bayesian framework; for this reason, the Bayesian frameworks that will be described for ordinal regression in this chapter are reduced to logistic or probit regression when only one threshold parameter needs to be estimated ($\gamma_1$), or equivalently this is set to 0 and an intercept parameter ($\beta_0$) is added to the vector coefficients that also need to be estimated,

$\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)^{\mathrm{T}}$. In this model, the following development will be concentrated in the probit ordinal regression model.

A Bayesian specification for the ordinal regression is very similar to the linear model described in Chap. 6, and assumes the following prior distribution for the parameter vector $\boldsymbol{\theta} = \left(\boldsymbol{\beta}^T, \boldsymbol{\gamma}^T, \sigma_\beta^2\right)^{\mathrm{T}}$ :

$$f(\boldsymbol{\theta}) \propto \frac{1}{\left(\sigma_\beta^2\right)^{\frac{p}{2}}} \exp\left(\left[-\frac{1}{2\sigma_\beta^2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{\beta}\right]\right) \frac{\left(\frac{S_\beta}{2}\right)^{\frac{v_\beta}{2}}}{\Gamma\left(\frac{v_\beta}{2}\right)\left(\sigma_\beta^2\right)^{1+\frac{v_\beta}{2}}} \exp\left(-\frac{S_\beta}{2\sigma_\beta^2}\right)$$

That is, a flat prior is assumed for the threshold parameter $\boldsymbol{\gamma}$, a multivariate normal distribution for the vector of beta coefficients, $\boldsymbol{\beta} \mid \sigma_\beta^2 \sim N_p\left(\mathbf{0}, \sigma_\beta^2 \boldsymbol{I}_p\right)$, and a scaled inverse chi-squared for $\sigma_\beta^2$. From now on, this model will be referred to as BRR in this chapter in analogy to the Bayesian linear regression and the way that this is implemented in the BGLR R package. Similar to the genomic linear regression model, the posterior distribution of the parameter vector does not have a known form and a Gibbs sampler is used to explore this; for this reason, in the coming lines, the Gibbs sampling method proposed by Albert and Chib (1993) is described. To make it possible to derive the full conditional distributions, the parameter vector is augmented with a latent variable in the representation of model (7.1).

The joint probability density function of the vector of observations, $\boldsymbol{Y} = (Y_1, \ldots, Y_n)^{\mathrm{T}}$, and the latent variables $\boldsymbol{L} = (L_1, \ldots, L_n)^{\mathrm{T}}$, evaluated in the vector values $\boldsymbol{y} = (y_1, \ldots, y_n)^{\mathrm{T}}$ and $\boldsymbol{l} = (l_1, \ldots, l_n)^{\mathrm{T}}$, is given by

$$\begin{aligned} f(\boldsymbol{y}, \boldsymbol{l} | \boldsymbol{\beta}, \boldsymbol{\gamma}) &= \prod_{i=1}^{n} f_{L_i}(l_i) I_{\left(\gamma_{y_i-1}, \gamma_{y_i}\right)}(l_i) \\ &= \prod_{i=1}^{n} \exp\left[-\frac{1}{2}\left(l_i + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right)^2\right] I_{\left(\gamma_{y_i-1}, \gamma_{y_i}\right)}(l_i) \\ &= \exp\left[-\frac{1}{2}\sum_{i=1}^{n}\left(l_i + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right)^2\right] \prod_{i=1}^{n} I_{\left(\gamma_{y_i-1}, \gamma_{y_i}\right)}(l_i), \end{aligned}$$

where $I_{\mathrm{A}}$ is the indicator function of a set A, and $f_{L_i}(l_i)$ is the density function of the latent variable that in the ordinal probit regression model corresponds to the normal distribution with mean $-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}$ and unit variance. Then the full conditional of the $j$th component of $\boldsymbol{\beta}$, $\beta_j$, given the rest of the parameters and the latent variables, is given by

$$\begin{aligned}f\left(\beta_j|-\right) &\propto f(\boldsymbol{y},\boldsymbol{l}|\boldsymbol{\beta},\boldsymbol{\gamma})f\left(\boldsymbol{\beta}|\sigma_\beta^2\right)\\&\propto \prod_{i=1}^{n} f_{L_i}(l_i)I_{\left(\gamma_{y_i-1},\gamma_{y_i}\right)}(l_i)f\left(\boldsymbol{\beta}|\sigma_\beta^2\right)\\&\propto \exp\left[-\frac{1}{2}\sum_{i=1}^{n}\left(e_{ij}+x_{ij}\beta_j\right)^2-\frac{1}{2\sigma_\beta^2}\beta_j^2\right]\\&\propto \exp\left\{-\frac{1}{2\widetilde{\sigma}_{\beta_j}^2}\left(\beta_j-\widetilde{\beta}_j\right)^2\right\},\end{aligned}$$

where $e_{ij}=l_i+\sum_{\substack{k=1\\k\neq j}}^{p}x_{1k}\beta_k$, $\boldsymbol{l}=(l_1,\ldots,l_n)^{\mathrm{T}}$, $\boldsymbol{x}_j=[x_{1j},\ldots,x_{nj}]^{\mathrm{T}}$, $\boldsymbol{e}_j=[e_{1j},\ldots,e_{nj}]^{\mathrm{T}}$, $\widetilde{\sigma}_{\beta_j}^2=\left(\sigma_\beta^{-2}+\boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{x}_j\right)^{-1}$, and $\widetilde{\beta}_j=-\widetilde{\sigma}_{\beta_j}^2\left(\boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{e}_j\right)$. So, the full conditional distribution of $\beta_j$ is $N\left(\widetilde{\beta}_j,\widetilde{\sigma}_{\beta_j}^2\right)$.

Now, the full conditional distribution of the threshold parameter $\gamma_c$ is

$$\begin{aligned}f(\gamma_c|-) &\propto f(\boldsymbol{y},\boldsymbol{l}|\boldsymbol{\beta},\boldsymbol{\gamma})\\&\propto \prod_{i=1}^{n} I_{\left(\gamma_{y_i-1},\gamma_{y_i}\right)}(l_i)\\&\propto \prod_{i\in\{i:y_i=c\}} I_{(\gamma_{c-1},\gamma_c)}(l_i)\prod_{i\in\{i:y_i=c+1\}} I_{\left(\gamma_c,\gamma_{c+1}\right)}(l_i)\\&\propto I_{(a_c,b_c)}(\gamma_c),\end{aligned}$$

where $a_c=\max\{l_i:y_i=c\}$ and $b_c=\min\{l_i:y_i=c+1\}$. So $\gamma_c\mid-\sim U(a_c,b_c)$, $c=1,2,\ldots,C-1$. Now, the full conditional distribution of the latent variables is

$$f(\boldsymbol{l}|-)\propto f(\boldsymbol{y},\boldsymbol{l}|\boldsymbol{\beta},\boldsymbol{\gamma})\propto\prod_{i=1}^{n} f_{L_i}(l_i)I_{\left(\gamma_{y_i-1},\gamma_{y_i}\right)}(l_i)$$

from which we have that conditional on the rest of parameters, the latent variables are independent truncated normal random variables, that is, $L_i\mid-\sim N\left(-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta},1\right)$ truncated in the interval $\left(\gamma_{y_i-1},\gamma_{y_i}\right)$, $i=1,\ldots,n$.

The full conditional distribution of $\sigma_\beta^2$ is the same as in the linear regression model described in Chap. 6, so a Gibbs sampler for exploration of the posterior distribution of the parameters of the models can be implemented by iterating the following steps:

1. Initialize the parameters: $\beta_j=\beta_{j0}, j=1,\ldots,p$, $\gamma_c=\gamma_{c0}$, $c=1,\ldots,C-1$, $l_i=l_{i0}$, $i=1,\ldots,n$, and $\sigma_\beta^2=\sigma_{\beta 0}^2$.
2. For each $i=1,\ldots,n$, simulate $l_i$ from the normal distribution $N\left(-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta},1\right)$ truncated in $\left(\gamma_{y_i-1},\gamma_{y_i}\right)$.

3. For each $j = 1, \ldots, p$, simulate from the full conditional distribution of $\beta_j$, that is, from a $N\left(\widetilde{\beta}_j, \widetilde{\sigma}^2_{\beta_j}\right)$, where $\widetilde{\sigma}^2_{\beta_j} = \left(\sigma_\beta^{-2} + \boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{x}_j\right)^{-1}$ and $\widetilde{\beta}_j = -\widetilde{\sigma}^2_{\beta_j}\left(\boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{e}_j\right)$.
4. For each $c = 1, 2, \ldots, C - 1$, simulate from the full conditional distribution of $\gamma_c$, $\gamma_c \mid - \sim U(a_c, b_c)$, where $a_c = \max\{l_i : y_i = c\}$ and $b_c = \min\{l_i : y_i = c + 1\}$.
5. Simulate $\sigma^2_\beta$ from a scaled inverse chi-squared distribution with parameters $\widetilde{v}_\beta = v_\beta + p$ and $\widetilde{S}_\beta = S_\beta + \boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{\beta}$, that is, from a $\chi^{-2}\left(\widetilde{v}_\beta, \widetilde{S}_\beta\right)$.
6. Repeat 1–5 until a burning period and the desired number of samples are reached.

A similar Gibbs sampler is implemented in the BGLR R package, and the hyperparameters are established in a similar way as the linear regression models described in Chap. 6. When the hyperparameters $S_\beta$ and $v_\beta$ are not specified, by default BGLR is assigned $v_\beta = 5$ and to $S_\beta$ a value such that the prior mode of $\sigma^2_\beta$ ($\chi^{-2}(v_\beta, S_\beta)$) is equal to half of the "total variance" of the latent variables (Pérez and de los Campos 2014). An implementation of this model can be done with a fixed value of the variance component parameter $\sigma^2_{0\beta}$ by choosing a very large value of the degree of freedom parameter ($v_\beta = 10^{10}$) and taking the scale value parameter $S_\beta = \sigma^2_{0\beta}\left(v_\beta + 2\right)$.

The basic R code with the BGLR implementation of this model is as follows:

```
ETA = list( list( X=X, model='BRR' ) )
A = BGLR(y=y, response_type='ordinal', ETA=ETA, nIter = 1e4, burnIn =
1e3)
Probs = A$Probs
```

where the predictor component is specified in **ETA**: argument **X** specifies the design matrix, whereas the argument model describes the priors of the model's parameters. The response variable vector is given in **y**, and the ordinal model is specified in the response_type argument. In the last line, the posterior means of the probabilities of each category are extracted, and can be used to give an estimation of a particular metric when comparing models or to evaluate the performance of this model, which will be explained later. An application of this ordinal model is given by Montesinos-López et al. (2015a).

Similar to the linear regression model in Chap. 6, when the number of markers to be used in genomic prediction is very large relative to the number of observations, $p \gg n$, the dimension of the posterior distribution (number of parameters in $\boldsymbol{\theta}$, $p + c - 1 + 1$) to be explored can be reduced by simulating the genetic vector effect $\boldsymbol{b} = (b_1, \ldots, b_n)^{\mathrm{T}}$, where $b_i = \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}$, $i = 1, \ldots, n$, instead of $\boldsymbol{\beta}$. Because $\boldsymbol{\beta} \mid \sigma^2_\beta \sim N_p\left(\boldsymbol{0}, \sigma^2_\beta \boldsymbol{I}_p\right)$, the induced distribution of $\boldsymbol{b} = (b_1, \ldots, b_n)^{\mathrm{T}}$ is $\boldsymbol{b} \mid \sigma^2_g \sim N_n\left(\boldsymbol{0}, \sigma^2_g \boldsymbol{G}\right)$, where $\sigma^2_g = p\sigma^2_\beta$ and $\boldsymbol{G} = \frac{1}{p}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$. Then, with this and assuming a scaled inverse chi-squared distribution as prior for $\sigma^2_g$, $\sigma^2_g \sim \chi^{-2}\left(v_g, S_g\right)$, and a flat prior for the threshold parameters ($\boldsymbol{\gamma}$), an ordinal probit GBLUP Bayesian regression model specification of model (7.1) is given by

$$p_{ic} = P(Y_i = c) = \Phi(\gamma_c + b_i) - \Phi(\gamma_{c-1} + b_i), \ \ c = 1, \ldots, C, \tag{7.2}$$

where now $L_i = -b_i + \epsilon_i$ is the latent variable, and $\Phi$ is the cumulative normal standard distribution. In matrix form the model for the latent variable can be specified as $\boldsymbol{L} = \boldsymbol{b} + \boldsymbol{\epsilon}$, where $\boldsymbol{L} = (L_1, \ldots, L_n)^{\mathrm{T}}$ and $\boldsymbol{\epsilon} \sim N_n(\boldsymbol{0}, \boldsymbol{I}_n)$. A Gibbs sampler of the posterior of the parameters of this model can be obtained similarly as was done for model (7.1). Indeed, the full conditional density of $\boldsymbol{b}$ is given by

$$\begin{aligned} f(\boldsymbol{b}|-) &\propto f(\boldsymbol{y}, \boldsymbol{l}|\boldsymbol{b}, \boldsymbol{\gamma}) f\left(\boldsymbol{b}|\sigma_g^2\right) \\ &\propto \prod_{i=1}^{n} f_{L_i}(l_i) I_{\left(\gamma_{y_i-1}, \gamma_{y_i}\right)}(l_i) f\left(\boldsymbol{b}|\sigma_g^2\right) \\ &\propto \exp\left[-\frac{1}{2}(\boldsymbol{l}+\boldsymbol{b})^{\mathrm{T}}(\boldsymbol{l}+\boldsymbol{b}) - \frac{1}{2\sigma_g^2}\boldsymbol{b}^{\mathrm{T}}\boldsymbol{G}^{-1}\boldsymbol{b}\right] \\ &\propto \exp\left[-\frac{1}{2}\left(\boldsymbol{b}-\widetilde{\boldsymbol{b}}\right)^{\mathrm{T}}\widetilde{\boldsymbol{\Sigma}}_b\left(\boldsymbol{b}-\widetilde{\boldsymbol{b}}\right)\right], \end{aligned}$$

where $\widetilde{\boldsymbol{\Sigma}}_b = \left(\sigma_g^{-2}\boldsymbol{G}^{-1} + \boldsymbol{I}_n\right)^{-1}$ and $\widetilde{\boldsymbol{b}} = -\widetilde{\boldsymbol{\Sigma}}_b \boldsymbol{l}$ . So the full conditional distribution of $\boldsymbol{b}$ is $N_n\left(\widetilde{\boldsymbol{b}}, \widetilde{\boldsymbol{\Sigma}}_b\right)$. Then, a Gibbs sampler exploration of this model can be done by iterating the following steps:

1. Initialize the parameters: $b_i = b_{i0}$, $i = 1, \ldots, n$, $\gamma_c = \gamma_{c0}$, $c = 1, \ldots, C-1$, $l_i = l_{i0}$, $i = 1, \ldots, n$, and $\sigma_\beta^2 = \sigma_{\beta 0}^2$.
2. For each $i = 1, \ldots, n$, simulate $l_i$ from the normal distribution $N\left(-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}, 1\right)$ truncated in $\left(\gamma_{y_i-1}, \gamma_{y_i}\right)$.
3. Simulate $\boldsymbol{b}$ from $N_n\left(\widetilde{\boldsymbol{b}}, \widetilde{\boldsymbol{\Sigma}}_b\right)$, with $\widetilde{\boldsymbol{\Sigma}}_b = \left(\sigma_g^{-2}\boldsymbol{G}^{-1} + \boldsymbol{I}_n\right)^{-1}$ and $\widetilde{\boldsymbol{b}} = -\widetilde{\boldsymbol{\Sigma}}_b \boldsymbol{l}$.
4. For each $c = 1, 2\ldots, C-1$, simulate from the full conditional distribution of $\gamma_c$ $\gamma_c \mid - \sim U(a_c, b_c)$, where $a_c = \max\{l_i : y_i = c\}$ and $b_c = \min\{l_i : y_i = c+1\}$.
5. Simulate $\sigma_g^2$ from a scaled inverse chi-squared distribution with parameters $\widetilde{v}_g = v_g + n$ and $\widetilde{S}_g = S_g + \boldsymbol{b}^{\mathrm{T}}\boldsymbol{b}$, that is, from a $\chi^{-2}\left(\widetilde{v}_g, \widetilde{S}_g\right)$.
6. Repeat 1–5 until a burning period and the desired number of samples are reached.

This model can also be implemented with the BGLR package:

```
ETA = list( list( K=G, model='RKHS' ) )
A = BGLR(y=y, response_type='ordinal', ETA=ETA, nIter = 1e4, burnIn =
1e3)
Probs = A$Probs
```

where instead of specifying the design matrix and the prior model to the associated regression coefficients, now the genomic relationship matrix $\boldsymbol{G}$ is given, and the corresponding model in this case corresponds to RKHS, which can also be used

when another type of information between the lines is available, for example, the pedigree relationship matrix.

Like the Bayesian linear regression model, other variants of model BRR can be obtained by adopting different prior models for the beta coefficients ($\boldsymbol{\beta}$): FIXED, BayesA, BayesB, BayesC, or BL, for example. See Chap. 6 for details of each of these prior models for the regression coefficients. Indeed, the full conditional distributions used to implement a Gibbs sampler in each of these ordinal models are the same as the corresponding Bayesian linear regression models, except that the full conditional distribution for the variance component of random errors is not needed and two full conditional distributions (for the threshold parameters and the latent variables) are added, where now the latent variable will play the role of the response value in the Bayesian linear regression. These models can also be implemented in the BGLR package.

For example, a Gibbs sampler implementation for ordinal model (7.1) with a BayesA prior (see Chap. 6) can be done by following steps:

1. Initialize the parameters: $\beta_j = \beta_{j0}, j = 1, \ldots, p, \gamma_c = \gamma_{c0}, c = 1, \ldots, C-1, l_i = l_{i0}$, $i = 1, \ldots, n$, and $\sigma_\beta^2 = \sigma_{\beta 0}^2$.
2. For each $i = 1, \ldots, n$, simulate $l_i$ from the normal distribution $N\left(-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}, 1\right)$ truncated in $\left(\gamma_{y_i-1}, \gamma_{y_i}\right)$.
3. For each $j = 1, \ldots, p$, simulate from the full conditional distribution of $\beta_j$, that is, from a $N\left(\widetilde{\beta}_j, \widetilde{\sigma}_{\beta_j}^2\right)$, where $\widetilde{\sigma}_{\beta_j}^2 = \left(\sigma_{\beta_j}^{-2} + \boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{x}_j\right)^{-1}$ and $\widetilde{\beta}_j = \widetilde{\sigma}_{\beta_j}^2\left(\boldsymbol{x}_j^{\mathrm{T}}\boldsymbol{e}_j\right)$.
4. For each $c = 1, 2, \ldots, C-1$, simulate from the full conditional distribution of $\gamma_c$, $\gamma_c \mid - \sim U(a_c, b_c)$, where $a_c = \max\{l_i : y_i = c\}$ and $b_c = \min\{l_i : y_i = c + 1\}$.
5. Simulate from the full conditional of each $\sigma_{\beta_j^2}$

$$\sigma_{\beta_j}^2 \mid \boldsymbol{\gamma}, \boldsymbol{l}, \boldsymbol{\beta}_0, \boldsymbol{\sigma}_{-j}^2, S_\beta, \sim \chi^{-2}_{\widetilde{v}_j, \widetilde{S}_{\beta_j}},$$

   where $\widetilde{v}_{\beta_j} = v_\beta + 1$, scale parameter $\widetilde{S}_{\beta_j} = S_\beta + \beta_j^2$, and $\boldsymbol{\sigma}_{-j}^2$ is the vector $\boldsymbol{\sigma}_\beta^2 = \left(\sigma_{\beta_1}^2, \ldots, \sigma_{\beta_p}^2\right)$ but without the $j$th entry.
6. Simulate from the full conditional of $S_\beta$

$$\begin{aligned}
f(S_\beta|-) &\propto \left[\prod_{j=1}^{p} f\left(\sigma_{\beta_j}^2 | S_\beta\right)\right] f(S_\beta) \\
&\propto \prod_{j=1}^{p}\left[\frac{\left(\frac{S_\beta}{2}\right)^{\frac{v_\beta}{2}}}{\Gamma\left(\frac{v_\beta}{2}\right)\left(\sigma_{\beta_j}^2\right)^{1+\frac{v_\beta}{2}}} \exp\left(-\frac{S_\beta}{2\sigma_{\beta_j}^2}\right)\right] S_\beta^{s-1} \exp\left(-rS_\beta\right) \\
&\propto S_\beta^{s+\frac{pv_\beta}{2}-1} \exp\left[-\left(r + \frac{1}{2}\sum_{j=1}^{p}\frac{1}{\sigma_{\beta j}^2}\right) S_\beta\right]
\end{aligned}$$

which corresponds to the kernel of the gamma distribution with rate parameter $\widetilde{r} = r + \frac{1}{2}\sum_{j=1}^{p}\frac{1}{\sigma_{\beta j}^2}$ and shape parameter $\widetilde{s} = s + \frac{pv_\beta}{2}$, and so, $S_\beta \mid - \sim \text{Gamma}(\widetilde{r}, \widetilde{s})$.

7. Repeat steps 2–6 depending on how many values you wish to simulate of the parameter vector $(\boldsymbol{\beta}^{\mathrm{T}}, \boldsymbol{\sigma}_\beta^2, \boldsymbol{\gamma}^{\mathrm{T}})$

The implementation of BayesA for an ordinal response variable in BGLR is as follows:

```
ETA = list( list( X=X, model='BayesA' ) )
A = BGLR(y=y, response_type='ordinal', ETA=ETA, nIter = 1e4, burnIn =
1e3)
Probs = A$Probs
```

The implementation of the other Bayesian ordinal regression models, BayesB, BayesC, and BL, in BGLR is done by only replacing in ETA the corresponding model, as was commented before, see Chap. 6 for details and the difference between all these prior models for the regression coefficients.

### 7.2.1 Illustrative Examples

**Example 1 BRR and GBLUP** To illustrate how these models work and how they can be implemented, here we used a toy data set with an ordinal response with five levels and consisting of 50 lines that were planted in three environments (a total of 150 observations) and the genotyped information of 1000 markers for each line. To evaluate the performance of the described models (BRR, BayesA, GBLUP, BayesB, BayesC, and BL), 10 random partitions were used in a cross-validation strategy, where 80% of the complete data set was used to train the model and the rest to evaluate the model in terms of the Brier score (BS) and the proportion of cases correctly classified (PCCC).

Six models were implemented with the BGLR R package, using a burn-in period equal to 1000 and 10,000 iterations, and the default hyperparameter for prior distribution. The results are shown in Table 7.1, where we can appreciate similar performance in terms of the Brier score metric in all models, and a similar but poor performance of all models when the proportion of cases correctly classified (PCCC) was used. With the BRR model only in 4 out of 10 partitions, the values of the PCCC were slightly greater or equal to what can be obtained by random assignation (1/5), while with the GBLUP model this happened in 3 out of 10 partitions. A similar behavior is obtained with the rest of the models. On average, the PCCC values were 0.1566, 0.1466, 0.15, 0.1466, 0.1533, and 0.1433 for the BRR, GBLUP, BayesA, BayesB, BayesC, and Bayes Lasso (BL) models, respectively.

**Table 7.1** Brier score (BS) and proportion of cases correctly classified (PCCC) across 10 random partitions, with 80% of the total data set used for training and the rest for testing, under model (7.1) with different priors models to the marker coefficients: ordinal Bayesian Ridge regression model (BRR), BayesA, BayesB, BayesC, and BL (model (7.1)); and under the ordinal GBLUP regression model (7.2)

| Model | BRR | | GBLUP | | BayesA | |
|---|---|---|---|---|---|---|
| PT | BS | PCCC | BS | PCCC | BS | PCCC |
| 1 | 0.3957 | 0.1333 | 0.3937 | 0.1333 | 0.3923 | 0.1333 |
| 2 | 0.3891 | 0.2000 | 0.3884 | 0.2000 | 0.3896 | 0.2000 |
| 3 | 0.3977 | 0.2333 | 0.3944 | 0.2333 | 0.3935 | 0.2333 |
| 4 | 0.4168 | 0.1333 | 0.4154 | 0.1333 | 0.4131 | 0.2000 |
| 5 | 0.4079 | 0.0667 | 0.3991 | 0.0667 | 0.4013 | 0.0667 |
| 6 | 0.4068 | 0.1000 | 0.4038 | 0.1000 | 0.4074 | 0.1000 |
| 7 | 0.4342 | 0.1000 | 0.4258 | 0.1000 | 0.4298 | 0.0667 |
| 8 | 0.3867 | 0.2000 | 0.3845 | 0.1667 | 0.3858 | 0.1667 |
| 9 | 0.4145 | 0.1667 | 0.4171 | 0.1000 | 0.4183 | 0.1000 |
| 10 | 0.3875 | 0.2333 | 0.3862 | 0.2333 | 0.3866 | 0.2333 |
| Average (SD) | 0.4036 (0.01) | 0.1566 (0.05) | 0.4008 (0.01) | 0.1466 (0.05) | 0.4017 (0.01) | 0.15 (0.06) |
| Model | BayesB | | BayesC | | BL | |
| PT | BS | PCCC | BS | PCCC | BS | PCCC |
| 1 | 0.3924 | 0.1333 | 0.3943 | 0.1333 | 0.3918 | 0.1333 |
| 2 | 0.3881 | 0.2000 | 0.3885 | 0.2000 | 0.3878 | 0.2000 |
| 3 | 0.3941 | 0.2667 | 0.3951 | 0.2333 | 0.3935 | 0.2333 |
| 4 | 0.4141 | 0.1333 | 0.4167 | 0.1333 | 0.4142 | 0.1333 |
| 5 | 0.4037 | 0.0667 | 0.4044 | 0.0667 | 0.4047 | 0.0667 |
| 6 | 0.3978 | 0.1000 | 0.4022 | 0.1000 | 0.3984 | 0.1000 |
| 7 | 0.4268 | 0.0667 | 0.4286 | 0.1000 | 0.4281 | 0.0667 |
| 8 | 0.3849 | 0.1667 | 0.3862 | 0.2000 | 0.3852 | 0.1667 |
| 9 | 0.4184 | 0.1000 | 0.4155 | 0.1000 | 0.4169 | 0.1000 |
| 10 | 0.3869 | 0.2333 | 0.3862 | 0.2667 | 0.3861 | 0.2333 |
| Average (SD) | 0.4007 (0.01) | 0.1466 (0.06) | 0.4017 (0.01) | 0.1533 (0.06) | 0.4006 (0.01) | 0.1433 (0.06) |

The R code to reproduce the results in Table 7.1 is given in Appendix 1.

In some applications, additional information is available, such as the sites (locations) where the experiments were conducted, environmental covariates, etc., which can be taken into account to improve the prediction performance.

One extension of model (7.1) that takes into account environment effects and environment–marker interaction is given by

$$p_{ic} = P(Y_i = c) = F(\gamma_c - \eta_i) - F(\gamma_{c-1} - \eta_i), c = 1, \dots, C, \tag{7.3}$$

where now the predictor $\eta_i = \boldsymbol{x}_{Ei}^{\mathrm{T}}\boldsymbol{\beta}_E + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} + \boldsymbol{x}_{EMi}^{T}\boldsymbol{\beta}_{EM}$, in addition to the marker effects ($\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}$ ) in the second summand, is included the environment and the

environment–marker interaction effects, respectively. In the latent variable representation, this model is equivalently expressed as

$$P(Y_i = c) = P(\gamma_{c-1} \leq L_i \leq \gamma_c), c = 1, \ldots, C,$$

where $\boldsymbol{L} = (L_1, \ldots, L_n)^{\mathrm{T}} = \mathbf{X}_E\boldsymbol{\beta}_E + \boldsymbol{X\beta} + \boldsymbol{X}_{EM}\boldsymbol{\beta}_{EM} + \boldsymbol{\epsilon}$ is the vector with latent random variables of all observations, $\boldsymbol{\epsilon} \sim N_n(\mathbf{0}, \boldsymbol{I}_n)$ is a random error vector, $\boldsymbol{X}_E$, $\boldsymbol{X}$, and $\boldsymbol{X}_{EM}$ are the design matrices of the environments, markers, and environment–marker interactions, respectively, while $\boldsymbol{\beta}_E$ and $\boldsymbol{\beta}_{EM}$ are the vectors of the environment effects and the interaction effects, respectively, with a prior distribution that can be specified as was done for $\boldsymbol{\beta}$. In fact, with the BGLR function, it is also possible to implement all these extensions, since it allows using any of the several priors included here: FIXED, BRR, BayesA, BayesB, BayesC, and BL. For example, the basic BGLR code to implement model (7.3) with a flat prior ("FIXED") for the environment effects, a BRR prior for marker effects and for the environment–marker interaction effects, is as follows:

```
X = scale(X)
#Environment matrix design
XE = model.matrix(~Env,data=dat_F)[,-1]
#Environment-marker interaction matrix design
XEM = model.matrix(~0+X:Env,data=dat_F)
ETA = list(list(X=XE, model='FIXED'), list(X=X, model='BRR'),
          list(X=XEM, model='BRR'))#Predictor
A = BGLR(y=y,response_type='ordinal',ETA=ETA, nIter = 1e4,burnIn =
1e3,verbose = FALSE)
Probs = A$probs
```

where **dat_F** is the data file that contains all the information of how the data was collected (GID: Lines or individuals; Env: Environment; y: response variable of the trait). Other desired prior models to beta coefficients of each predictor component are obtained only by replacing the "model" argument of each of the three components of the predictor. For example, for a BayesA prior model for the marker effects, in the second sub-list we must use model=**'BayesA'**.

The latent random vector of model (7.1) under the GBLUP specification, plus genotypic and environment×genotypic interaction effects, takes the form

$$\boldsymbol{L} = \boldsymbol{X}_E\boldsymbol{\beta}_E + \boldsymbol{Z}_L\boldsymbol{g} + \boldsymbol{Z}_{LE}\boldsymbol{gE} + \boldsymbol{\epsilon} \tag{7.4}$$

which is like model (6.7) in Chap. 6, where $\boldsymbol{Z}_L$ and $\boldsymbol{Z}_{LE}$ are the incident matrices of the genotypes and environment×genotype interaction effects, respectively, and $\boldsymbol{g}$ and $\boldsymbol{gE}$ are the corresponding random effects which have distributions $N_J\left(\boldsymbol{O}, \sigma_g^2\boldsymbol{G}\right)$ and $N_J\left(\boldsymbol{O}, \sigma_{gE}^2(\boldsymbol{I}_I \otimes \boldsymbol{G})\right)$, respectively, where $J$ is the number of different lines in the data set. This model can be trained with the BGLR package as follows:

```
I = length(unique(dat_F$Env))
XE = model.matrix(~0+Env,data=dat_F)[,-1]
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
K_L = Z_L %*%G%*%t(Z_L)
Z_LE = model.matrix(~0+GID:Env,data=dat_F,
         xlev = list(GID=unique(dat_F$GID),Env = unique(dat_F$Env)))
K_LE = Z_LE%*%kronecker(diag(I),G)%*%t(Z_LE)
ETA = list( list(model='FIXED',X=XE),
                    list( model = 'RHKS', K = K_L )),
                    list(model='RKHS',K=K_LE )
A   = BGLR(y, response_type = "ordinal", ETA = ETA, nIter = 1e4, burnIn =
1e3)
Probs = A$Probs
```

where **dat_F** is as before, that is, the data file that contains all the information of how the data were collected. Similar specification can be done when other kinds of covariates are available, for example, environmental covariates.

**Example 2 Environments + Markers + Marker×Environment Interaction** This example illustrates how models (7.3) and (7.4) can be fitted and used for prediction. Here we used the same data set as in Example 1, but now the environment and environment×marker effects are included in the predictor. For model (7.3), to the environment effect a flat prior is assigned [FIXED, a normal distribution with mean 0 and very large variance ($10^{10}$)], and one of BRR, BayesA, BayesB, BayesC, or BL prior model is assigned to the marker and marker×environment interaction effects. For model (7.4), a flat prior is assigned for environment effects. The performance evaluation was done using the same 10 random partitions used in Example 1, where 80% of the complete data set was used for training the model and the rest for testing, in which the Brier score (BS) and the proportion of correctly classified (PCCC) metrics were computed.

The results are shown in Table 7.2. They indicate an improved prediction performance of these models compared to the models fitted in Example 1 which only takes into account the marker effects (see Table 7.1). However, this improvement is only slightly better under the Brier score, because the reduction in the average BS across all 10 partitions was 1.55, 3.74, 0.96, 0.83, 1.41, and 1.32% for the BRR, GBLUP, BayesA, BayesB, BayesC, and BL, respectively. A more notable improvement was obtained with the PCCC criteria, where now for all models in about 8 out of the 10 partitions, the value of this metric was greater than the one obtained by random chance only. Indeed, the average values across the 10 partitions were 0.27, 0.28, 0.28, 0.27, 0.28, and 0.27, for the BRR, GBLUP, BayesA, BayesB, BayesC, and BL, respectively. This indicates 74.47, 90.90, 86.67, 86.36, 80.43, and 86.05% improvement for these models with respect to their counterpart models when only marker effects or genomic effects were considered. So, greater improvement was observed with the GBLUP model with both metrics, but finally the performance of all models is almost undistinguishable but with an advantage in time execution of the GBLUP model with respect to the rest. These issues were

**Table 7.2** Brier score (BS) and proportion of cases correctly classified (PCCC) across 10 random partitions, with 80% of the total data set used for training and the rest for testing, under model (7.3) with different prior models for the marker effects and the environment×marker interaction: BRR, BayesA, BayesB, BayesC, and BL; and under the ordinal GBLUP regression model (7.4)

| Model | BRR | | GBLUP | | BayesA | |
|---|---|---|---|---|---|---|
| PT | BS | PCC | BS | PCC | BS | PCC |
| 1 | 0.3942 | 0.3333 | 0.3721 | 0.3667 | 0.3925 | 0.2333 |
| 2 | 0.3872 | 0.3000 | 0.3881 | 0.2667 | 0.3871 | 0.3000 |
| 3 | 0.4081 | 0.2667 | 0.4019 | 0.2667 | 0.4054 | 0.2667 |
| 4 | 0.4428 | 0.1333 | 0.4375 | 0.0667 | 0.4448 | 0.1667 |
| 5 | 0.3853 | 0.2667 | 0.3602 | 0.3000 | 0.3837 | 0.2333 |
| 6 | 0.3905 | 0.3667 | 0.3681 | 0.3000 | 0.3899 | 0.3667 |
| 7 | 0.3995 | 0.1333 | 0.3849 | 0.2000 | 0.4007 | 0.1667 |
| 8 | 0.3737 | 0.3000 | 0.3690 | 0.3333 | 0.3816 | 0.3333 |
| 9 | 0.3999 | 0.3000 | 0.3893 | 0.3333 | 0.3989 | 0.3333 |
| 10 | 0.3937 | 0.3333 | 0.3925 | 0.3667 | 0.3953 | 0.4000 |
| Average (SD) | 0.3975 (0.01) | 0.2733 (0.07) | 0.3863 (0.02) | 0.28 (0.09) | 0.3979 (0.01) | 0.28 (0.08) |

| Model | BayesB | | BayesC | | BL | |
|---|---|---|---|---|---|---|
| PT | BS | PCC | BS | PCC | BS | PCC |
| 1 | 0.3913 | 0.3000 | 0.3894 | 0.2667 | 0.3890 | 0.2333 |
| 2 | 0.3868 | 0.3333 | 0.3868 | 0.3000 | 0.3851 | 0.3333 |
| 3 | 0.4077 | 0.2667 | 0.4081 | 0.2667 | 0.4063 | 0.2667 |
| 4 | 0.4456 | 0.1333 | 0.4420 | 0.1667 | 0.4403 | 0.1667 |
| 5 | 0.3809 | 0.2667 | 0.3805 | 0.2667 | 0.3814 | 0.2667 |
| 6 | 0.3883 | 0.3667 | 0.3862 | 0.3667 | 0.3846 | 0.3667 |
| 7 | 0.4026 | 0.1333 | 0.3992 | 0.1333 | 0.3964 | 0.1333 |
| 8 | 0.3735 | 0.3000 | 0.3735 | 0.3000 | 0.3739 | 0.3000 |
| 9 | 0.4026 | 0.3333 | 0.4015 | 0.3333 | 0.4037 | 0.3000 |
| 10 | 0.3948 | 0.3000 | 0.3948 | 0.3667 | 0.3939 | 0.3000 |
| Average (SD) | 0.3974 (0.01) | 0.2733 (0.07) | 0.3961 (0.01) | 0.2766 (0.07) | 0.3954 (0.01) | 0.2666 (0.07) |

pointed out in Chaps. 5 and 6. This difference is even greater when the number of markers is larger than the number of observations.

**Example 3 Binary Traits** For this example, we used the EYT Toy data set consisting of 40 lines, four environments (Bed5IR, EHT, Flat5IR, and LHT), and a response binary variable based on plant Height (0 = low, 1 = high). For this example, marker information is not available, only the genomic relationship matrix for the 40 lines. So, only the models (M3 and M4) in (7.3) and (7.4) are fitted, and the performance prediction for these models was evaluated using cross-validation. Also, in this comparison model, (M5) (7.5) is added but without the line×environment interaction effects, that is, only the environment effect and the genetic effects are taken into account in the linear predictor:

**Table 7.3** Brier score (BS) and proportion of cases correctly classified (PCCC) across 10 random partitions, with 80% of the total data set used for training and the rest for testing, under models (7.3), (7.4), and (7.5) with a flat prior for environment effects

| Model | M3 | | M4 | | M5 | |
|---|---|---|---|---|---|---|
| PT | BS | PCCC | BS | PCCC | BS | PCCC |
| 1 | 0.1841 | 0.8438 | 0.2250 | 0.7500 | 0.2166 | 0.7813 |
| 2 | 0.1580 | 0.8438 | 0.2307 | 0.5938 | 0.2221 | 0.7188 |
| 3 | 0.1960 | 0.7813 | 0.2357 | 0.6563 | 0.2308 | 0.6563 |
| 4 | 0.2261 | 0.7188 | 0.2119 | 0.6875 | 0.2102 | 0.6563 |
| 5 | 0.2108 | 0.6250 | 0.3275 | 0.4063 | 0.3174 | 0.4063 |
| 6 | 0.2070 | 0.7188 | 0.2280 | 0.6563 | 0.2242 | 0.6250 |
| 7 | 0.2094 | 0.6563 | 0.2261 | 0.6250 | 0.2248 | 0.5938 |
| 8 | 0.1864 | 0.7500 | 0.2322 | 0.7188 | 0.2293 | 0.6875 |
| 9 | 0.1813 | 0.7813 | 0.2643 | 0.5000 | 0.2561 | 0.5313 |
| 10 | 0.2310 | 0.6563 | 0.2865 | 0.5625 | 0.2878 | 0.5313 |
| Average (SD) | 0.199 (0.02) | 0.7375 (0.07) | 0.2468 (0.03) | 0.6156 (0.1) | 0.2419 (0.03) | 0.6187 (0.1) |

$$\boldsymbol{L} = \boldsymbol{X}_E\boldsymbol{\beta}_E + \boldsymbol{Z}_L\boldsymbol{g} + \boldsymbol{\epsilon} \tag{7.5}$$

The results are presented in Table 7.3 with the BS and PCCC metrics obtained in each partition of the random CV strategy. From this we can appreciate that the best performance with both metrics was obtained with the model that considered only the genetic effects (M3; (7.3)). On average, models M4 and M5 gave a performance in terms of BS, that was 24.02 and 19.79% greater than the corresponding performance of model M3, while with model M3, on average across the 10 partitions, an improvement of 21.57 and 19.19% in terms of PCCC was obtained with regard to models M4 and M5. The difference between these last two models is only slight; the average BS value of the first model was 2.01% greater than that of the second, and the PCCC of the second model was 0.50% greater than that of the first. The R code to reproduce the results is in Appendix 3.

## 7.3 Ordinal Logistic Regression

As described at the beginning of this chapter, the ordinal logistic model is given in model (7.1) but with $F$ the cumulative logistic distribution. Again, as in the ordinal probit regression model, the posterior distribution of the parameter is not easy to simulate and numerical methods are required. Here we describe the Gibbs sampler proposed by Montesinos-López et al. (2015b), which in addition to the latent variable $L_i$ in the representation of model (7.1), the parameter vectors are also augmented with a Pólya-Gamma latent random variable.

By using the following identity proposed by Polson et al. (2013):

$$\frac{[\exp(\eta)]^a}{[1+\exp(\eta)]^b} = 2^{-b}\exp(k\eta)\int_0^{\infty}\exp\left(-\frac{\eta^2}{2}\omega\right)f_{\omega}(\omega;b,0)d\omega,$$

where $k = a - b/2$ and $f_{\omega}(\omega; b, 0)$ is the density of a Pólya-Gamma random variable $\omega$ with parameters $b$ and $d = 0$ ($\omega \sim \mathrm{PG}(b, d)$, (see Polson et al. (2013) for details), the joint distribution of the vector of observations, $\boldsymbol{Y} = (Y_1, \ldots, Y_n)^{\mathrm{T}}$, and the latent variables $\boldsymbol{L} = (L_1, \ldots, L_n)^{\mathrm{T}}$ can be expressed as

$$\begin{aligned} f(\boldsymbol{y},\boldsymbol{l}|\boldsymbol{\beta},\boldsymbol{\gamma}) &= \prod_{i=1}^{n} f_{L_i}(l_i) I_{\left(\gamma_{y_i-1},\gamma_{y_i}\right)}(l_i) \\ &= \prod_{i=1}^{n} \frac{\exp(-l_i-\eta_i)}{[1+\exp(-l_i-\eta_i)]^2} I_{\left(\gamma_{y_i-1},\gamma_{y_i}\right)}(l_i) \\ &\propto \prod_{i=1}^{n} \left[\int_0^{\infty}\exp\left(-\frac{(l_i+\eta_i)^2}{2}\omega_i\right)f_{\omega_i}(\omega_i;2,0)d\omega_i\right] I_{\left(\gamma_{y_i-1},\gamma_{y_i}\right)}(l_i), \end{aligned}$$

where $\eta_i = \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}$ and $\omega_1, \ldots, \omega_n$ are independent random variables with the same Pólya-Gamma distribution with parameters $b = 2$ and $d = 0$.

Now, the Bayesian specification developed in Montesinos-López et al. (2015b) assumes the same priors as for the ordinal probit model (BRR prior model), except that now for the threshold parameter vector $\boldsymbol{\gamma}$, the prior distribution proposed by Sorensen et al. (1995) is adopted, which is the distribution of the order statistics of a random sample of size $C - 1$ of the uniform distribution in $(\gamma_L, \gamma_U)$, that is,

$$f(\boldsymbol{\gamma}) = (C-1)!\left(\frac{1}{\gamma_U-\gamma_L}\right)^{C-1} I_{\{\boldsymbol{\gamma}\in \boldsymbol{S}_{\gamma}\}},$$

where $\boldsymbol{S}_{\gamma} = \{(\gamma_1, \ldots, \gamma_{C-1}) : \gamma_L < \gamma_1 < \gamma_2 < \cdots < \gamma_{C-1} < \gamma_U\}$.

Then, by conditioning on the rest of parameters, including the latent Pólya-Gamma random variables $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_n)^{\mathrm{T}}$, the full conditional of $\gamma_c$ is given by

$$\begin{aligned} f(\gamma_c|-) &\propto f(\boldsymbol{y},\boldsymbol{l},\boldsymbol{\omega}|\boldsymbol{\beta},\boldsymbol{\gamma})f(\boldsymbol{\gamma}) \\ &\propto \left[\prod_{i=1}^{n} I_{\left(\gamma_{y_i-1},\gamma_{y_i}\right)}(l_i)\right](C-1)!\left(\frac{1}{\gamma_U-\gamma_L}\right)^{C-1} I_{\boldsymbol{S}_{\gamma}}(\boldsymbol{\gamma}) \\ &\propto I_{(a_c,b_c)}(\gamma_c) \end{aligned}$$

where $f(\boldsymbol{y}, \boldsymbol{l}, \boldsymbol{\omega}|\boldsymbol{\beta}, \boldsymbol{\gamma})$ is the joint density of the observations and the vector of the latent random variables, $\boldsymbol{L}$ and $\boldsymbol{\omega}$, $a_c = \max\{\gamma_{c-1}, \max\{l_i : y_i = c\}\}$, and $b_c = \min\{\gamma_{c+1}, \min\{l_i : y_i = c+1\}\}$. So the full conditional for the threshold parameters is also uniform, $\gamma_c \mid - \sim U(a_c, b_c)$, $c = 1, 2\ldots, C-1$.

Now, the conditional distribution of $\beta_j$ ($j$th element of $\boldsymbol{\beta}$) is given by

$$\begin{aligned} f\left(\beta_j|\sigma_\beta^2, \boldsymbol{\gamma}, \boldsymbol{L}, \boldsymbol{\omega}\right) &\propto f(\boldsymbol{y}, \boldsymbol{l}, \boldsymbol{\omega}|\boldsymbol{\beta}, \boldsymbol{\gamma}) f\left(\boldsymbol{\beta}|\sigma_\beta^2\right) \\ &\propto \exp\left[-\frac{1}{2}\sum_i^n \omega_i \frac{(l_i+\eta_i)^2}{2} - \frac{1}{2\sigma_\beta^2}\beta_j^2\right] \\ &\propto \exp\left[-\frac{1}{2}\sum_i^n \omega_i \frac{\left(e_{ij}+x_{ij}\beta_j\right)^2}{2} - \frac{1}{2\sigma_\beta^2}\beta_j^2\right] \\ &\propto \exp\left\{-\frac{1}{2\widetilde{\sigma}_{\beta_j}^2}\left(\beta_j - \widetilde{\beta}_j\right)^2\right\}, \end{aligned}$$

where $e_{ij} = l_i + \sum_{\substack{k=1 \\ k\neq j}}^p x_{1k}\beta_k$, $\boldsymbol{e}_j = [e_{1j}, \ldots, e_{nj}]^{\mathrm{T}}$, $\boldsymbol{x}_j = [x_{1j}, \ldots, x_{nj}]^{\mathrm{T}}$, $\widetilde{\sigma}_{\beta_j}^2 = \left(\sigma_\beta^{-2} + \sum_{i=1}^n \omega_i x_{ij}^2\right)^{-1}$, and $\widetilde{\beta}_j = -\widetilde{\sigma}_{\beta_j}^2\left(\sum_{i=1}^n w_i x_{ij} e_{ij}\right)$. From this, the full conditional distribution of $\beta_j$ is a normal distribution with mean $\widetilde{\beta}_j$ and variance $\widetilde{\sigma}_{\beta_j}^2$.

The full conditional distribution of $\sigma_\beta^2$ is the same as the one obtained for the ordinal probit model, $\sigma_\beta^2 \mid - \sim \chi^{-2}\left(\widetilde{v}_\beta, \widetilde{S}_\beta\right)$ with $\widetilde{v}_\beta = v_\beta + p$ and $\widetilde{S}_\beta = S_\beta + \boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{\beta}$. In a similar fashion, for the latent variable $L_i$, $i = 1, \ldots, n$, it can be seen that its full conditional distributions are also truncated normal in $\left(\gamma_{y_i-1}, \gamma_{y_i}\right)$ but with mean parameter $-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}$ and variance $1/\omega_i$, i.e., $L_i \mid - \sim N\left(-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}, \omega_i^{-1}\right)$ truncated in $\left(\gamma_{y_i-1}, \gamma_{y_i}\right)$, for each $i = 1, \ldots, n$. Finally, by following Eq. (5) in Polson et al. (2013), note that the full joint conditional distribution of the Pólya random variables $\boldsymbol{\omega}$ can be written as

$$\begin{aligned} f(\boldsymbol{\omega}|-) &\propto f(\boldsymbol{y}, \boldsymbol{l}, \boldsymbol{\omega}|\boldsymbol{\beta}, \boldsymbol{\gamma}) \\ &\propto \prod_{i=1}^n \exp\left(-\frac{(l_i+\eta_i)^2}{2}\omega_i\right) f_{\omega_i}(\omega_i; 2, 0) \\ &\propto \prod_{i=1}^n f_{\omega_i}(\omega_i; 2, l_i+\eta_i). \end{aligned}$$

From here, conditionally to $\boldsymbol{\beta}, \sigma_\beta^2, \boldsymbol{\gamma},$ and $\boldsymbol{L}$, $\omega_1, \ldots, \omega_n$ are independently Pólya-Gamma random variables with parameters $b = 2$ and $d = l_i + \eta_i$, $i = 1, \ldots, n$, respectively, that is, $\omega_i \mid - \sim \mathrm{PG}(2, l_i + \eta_i)$, $i = 1, \ldots, n$.

With the above derived full conditionals, a Gibbs sampler exploration of this ordinal logistic regression model can be done with the following steps:

1. Initialize the parameters: $\beta_j = \beta_{j0}, j = 1, \ldots, p, \gamma_c = \gamma_{c0}, c = 1, \ldots, C-1, l_i = l_{i0}$, $i = 1, \ldots, n$, and $\sigma_\beta^2 = \sigma_{\beta 0}^2$.
2. Simulate $\omega_1, \ldots, \omega_n$ independently Pólya-Gamma random variables with parameters $b = 2$ and $d = l_i + \eta_i$, $i = 1, \ldots, n$.
3. For each $i = 1, \ldots, n$, simulate $l_i$ from the normal distribution $N\left(-\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}, \omega_i^{-1}\right)$ truncated in $\left(\gamma_{y_i-1}, \gamma_{y_i}\right)$.
4. For each $j = 1, \ldots, p$, simulate from the full conditional distribution of $\beta_j$, that is, from a $N\left(\widetilde{\beta}_j, \widetilde{\sigma}_{\beta_j}^2\right)$, where $\widetilde{\sigma}_{\beta_j}^2 = \left(\sigma_\beta^{-2} + \sum_{i=1}^n \omega_i x_{ij}^2\right)^{-1}$ and $\widetilde{\beta}_j = -\widetilde{\sigma}_{\beta_j}^2 \left(\sum_{i=1}^n \omega_i x_{ij} e_{ij}\right)$.
5. For each $c = 1, 2\ldots, C-1$, simulate from the full conditional distribution of $\gamma_c$, $\gamma_c \mid - \sim U(a_c, b_c)$, where $a_c = \max\{\gamma_{c-1}, \max\{l_i : y_i = c\}\}$ and $b_c = \min\{\gamma_{c+1}, \min\{l_i : y_i = c+1\}\}$.
6. Simulate $\sigma_\beta^2$ from a scaled inverse chi-squared distribution with parameters $\widetilde{v}_\beta = v_\beta + p$ and $\widetilde{S}_\beta = S_\beta + \boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{\beta}$, that is, from a $\chi^{-2}\left(\widetilde{v}_\beta, \widetilde{S}_\beta\right)$.
7. Repeat 1–6 until a burning period and a desired number of samples are reached.

Similar modifications can be done to obtain Gibbs samplers corresponding to other prior adopted models for the beta coefficients (FIXED, BayesA, BayesB, BayesC, or BL; see Chap. 6 for details of these priors). Also, for the ordinal logistic GBLUP Bayesian regression model specification as done in (7.2) for the ordinal probit model, a Gibbs sampler implementation can be done following the steps below, which can be obtained directly from the ones described above:

1. Initialize the parameters: $b_i = b_{i0}, j = 1, \ldots, n, \gamma_c = \gamma_{c0}, c = 1, \ldots, C-1, l_i = l_{i0}$, $i = 1, \ldots, n$, and $\sigma_g^2 = \sigma_{g0}^2$.
2. Simulate $\omega_1, \ldots, \omega_n$ independently Pólya-Gamma random variables with parameters $b = 2$ and $d = l_i + b_i$, $i = 1, \ldots, n$.
3. For each $i = 1, \ldots, n$, simulate $l_i$ from the normal distribution $N\left(-b_i, \omega_i^{-1}\right)$ truncated in $\left(\gamma_{y_i-1}, \gamma_{y_i}\right)$.
4. Simulate $\boldsymbol{b}$ from $N_n\left(\widetilde{\boldsymbol{b}}, \widetilde{\boldsymbol{\Sigma}}_b\right)$, with $\widetilde{\boldsymbol{\Sigma}}_b = \left(\sigma_g^{-2}\boldsymbol{G}^{-1} + \boldsymbol{D}_\omega\right)^{-1}$ and $\widetilde{\boldsymbol{b}} = -\widetilde{\boldsymbol{\Sigma}}_b \boldsymbol{D}_\omega \boldsymbol{l}$, where $\boldsymbol{D}_\omega = \mathrm{Diag}(\omega_1, \ldots, \omega_n)$.
5. For each $c = 1, 2\ldots, C-1$, simulate from the full conditional distribution of $\gamma_c$, $\gamma_c \mid - \sim U(a_c, b_c)$, where $a_c = \max\{\gamma_{c-1}, \max\{l_i : y_i = c\}\}$ and $b_c = \min\{\gamma_{c+1}, \min\{l_i : y_i = c+1\}\}$.
6. Simulate $\sigma_g^2$ from a scaled inverse chi-squared distribution with parameters $\widetilde{v}_g = v_g + n$ and $\widetilde{S}_g = S_g + \boldsymbol{b}^{\mathrm{T}}\boldsymbol{b}$, that is, from a $\chi^{-2}\left(\widetilde{v}_g, \widetilde{S}_g\right)$.
7. Repeat 1–6 until a burning period and the desired number of samples are reached.

There is a lot of empirical evidence that there are no large differences between the prediction performance of the probit or logistic regression models. For this reason, here we only explained the Gibbs sampler for ordinal data under a logistic framework and did not provide illustrated examples. Also, we did not provide illustrative examples because it is not possible to implement this logistic ordinal model in BGLR.

## 7.4 Penalized Multinomial Logistic Regression

An extension of the logistic regression model described in Chap. 3 is the multinomial regression model that is also used to explain or predict a categorical nominal response variable (that does not have any natural order) with more than two categories. For example, a study could investigate the association of markers with diabesity (diabetes and obesity, diabetes but no obesity, obesity but no diabetes, and no diabetes and no obesity); another example could be to study the effects of age group on the histological subtypes of woman cancer (adenocarcinoma, adenosquamous, others), predict the preference in an election among four candidates (C1, C2, C3, and C4) using socioeconomic and demographic variables, etc.

The multinomial logistic regression model assumes that, conditionally to a covariate $\boldsymbol{x}_i$, a multinomial response random variable $Y_i$ takes one of the categories 1, 2, . . ., $C$, with the following probabilities:

$$P(Y_i = c|\boldsymbol{x}_i) = \frac{\exp\left(\beta_{0c} + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_c\right)}{\sum_{l=1}^{C}\exp\left(\beta_{0l} + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_l\right)}, c = 1, \ldots, C, \tag{7.6}$$

where $\boldsymbol{\beta}_c$, $c = 1, \ldots, C$, is a vector of coefficients of the same dimension as $\boldsymbol{x}$.

Model (7.6) is not identifiable because by evaluating these probabilities with the parameter values $\left(\beta_{0c}^{*}, \boldsymbol{\beta}_c^{*\mathrm{T}}\right) = \left(\beta_{0c} + \beta_{0c}^{**}, \boldsymbol{\beta}_{0c}^{\mathrm{T}} + \boldsymbol{\beta}_c^{**\mathrm{T}}\right)$, $c = 1, \ldots, C$, where $\beta_{0c}^{**}$ and $\boldsymbol{\beta}_c^{**}$ are arbitrary constants and vectors, give equal probabilities than when computing this with the original parameter values $\left(\beta_{0c}, \boldsymbol{\beta}_c^{\mathrm{T}}\right)$, $c = 1, \ldots, C$. A common constraint that avoids this lack of identifiability is to set $\left(\beta_{0C}, \boldsymbol{\beta}_C^{\mathrm{T}}\right) = \left(0, \boldsymbol{0}^{\mathrm{T}}\right)$, although any one of the other $C - 1$ of the vectors could be chosen.

With such a constraint ($\left(\beta_{0C}, \boldsymbol{\beta}_C^{\mathrm{T}}\right) = \left(0, \boldsymbol{0}^{\mathrm{T}}\right)$), we can identify a model by assuming that the effects of the covariate vector $\boldsymbol{x}_i$ over the log "odds" of each of the categories $c = 1, \ldots, C - 1$ with respect to category $C$ (baseline category) are given (Agresti 2012) by

$$\log\left(\frac{P(Y_i = c|x)}{P(Y_i = C|x)}\right) = \beta_{0c} + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_c, \quad c = 1, \ldots, C - 1,$$

where the effects of $\boldsymbol{x}_i$ depend on the chosen response baseline category. Similar expressions can be obtained when using the unconstrained model in (7.6). From

these relationships, the log odds corresponding to the probabilities of any two categories, $c$ and $l$, can be derived:

$$\log\left(\frac{P(Y_i=c|x)}{P(Y_i=l|x)}\right)=\beta_{0c}-\beta_{0l}+\boldsymbol{x}_i^{\mathrm{T}}(\boldsymbol{\beta}_c-\boldsymbol{\beta}_l),\quad c=1,\ldots,C-1.$$

Sometimes the number of covariates is larger than the number of observations (for example, in expression arrays and genomic prediction where the number of markers is often larger than the number of phenotyped individuals), so the conventional constraint described above to force identifiability in the model is not enough and some identifiability problems remain. One way to avoid this problem is to use similar quadratic regularization maximum likelihood estimation methods (Zhu and Hastie 2004) as done for some models in Chap. 3, but here only on the slopes ($\boldsymbol{\beta}_c$) for the covariates, under which setting the constraints mentioned before are no longer necessary.

For a given value of the regularization parameter, $\lambda > 0$, the regularized maximum likelihood estimation of the beta coefficients, $\boldsymbol{\beta}=\left(\beta_{0c},\boldsymbol{\beta}_c^{\mathrm{T}},\beta_{02},\boldsymbol{\beta}_2^{\mathrm{T}},\ldots,\beta_{0C},\boldsymbol{\beta}_C^{\mathrm{T}}\right)^{\mathrm{T}}$, is the value of this that maximizes the penalized log-likelihood:

$$\ell_p(\boldsymbol{\beta};\boldsymbol{y})=\ell(\boldsymbol{\beta};\boldsymbol{y})-\lambda\sum_{c=1}^{C}\boldsymbol{\beta}_c^{\mathrm{T}}\boldsymbol{\beta}_c, \tag{7.7}$$

where $\ell(\boldsymbol{\beta};\boldsymbol{y})=\log\left[L(\boldsymbol{\beta};\boldsymbol{y})\right]$ is the logarithm of the likelihood and takes the form:

$$\begin{aligned}\ell(\boldsymbol{\beta};\boldsymbol{y})&=\sum_{i=1}^{n}\log\left[P(Y_i=y_i|\boldsymbol{x}_i)\right]=\sum_{i=1}^{n}\sum_{c=1}^{C}I_{\{y_i=c\}}\log\left[\frac{\exp\left(\beta_{0c}+\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_c\right)}{\sum_{l=1}^{C}\exp\left(\beta_{0l}+\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_l\right)}\right]\\&=\sum_{i=1}^{n}\sum_{c=1}^{C}I_{\{y_i=c\}}\left(\beta_{0c}+\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_c\right)-\sum_{i=1}^{n}\log\left[\sum_{l=1}^{C}\exp\left(\beta_{0l}+\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_l\right)\right]\end{aligned} \tag{7.8}$$

When $p$ is large ($p \gg n$), direct optimization of $\ell_p(\boldsymbol{\beta};\boldsymbol{y})$ is almost impossible. An alternative is to use the sequential minimization optimization algorithm proposed by Zhu and Hastie (2004), which is applied after a transformation trick is used to make the involved computations feasible, because the number of parameters in the optimization is reduced to only $(n + 1)C$ instead of $(p + 1)C$.

Another alternative available in the glmnet package is the one proposed by Friedman et al. (2010) that is similar to that of the logistic Ridge regression in Chap. 3. This consists of maximizing (7.7) by using a block-coordinate descent strategy, where each block is formed by the beta coefficients corresponding to each class, $\boldsymbol{\beta}_c^{*\mathrm{T}}=\left(\beta_{0c},\boldsymbol{\beta}_c^{\mathrm{T}}\right)$, but with $\ell(\boldsymbol{\beta};\boldsymbol{y})$ replaced by a quadratic approximation with respect to beta coefficients of the chosen block, $\left(\beta_{0c},\boldsymbol{\beta}_c^{\mathrm{T}}\right)$, at the current values

of $\boldsymbol{\beta}$ ($\widetilde{\boldsymbol{\beta}}$). That is, the update of block $c$ is achieved by maximizing the following function with respect to $\beta_{0c}$ and $\boldsymbol{\beta}_c$:

$$f_c(\beta_{0c},\boldsymbol{\beta}_c) = \ell_c^*(\boldsymbol{\beta};\boldsymbol{y}) - \lambda\boldsymbol{\beta}_c^{\mathrm{T}}\boldsymbol{\beta}_c, \tag{7.9}$$

where $\ell_c^*(\boldsymbol{\beta};\boldsymbol{y}) = -\frac{1}{2}\sum_{i=1}^{n} w_{ic}\left(y_{ic}^* - \beta_{0c} - \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_c\right)^2 + \widetilde{c}$ is the second-order Taylor approximation of $\ell(\boldsymbol{\beta};\boldsymbol{y})$ with respect to the beta coefficients that conform block $c$, $\left(\beta_{0c},\boldsymbol{\beta}_c^{\mathrm{T}}\right)$, about the current estimates $\widetilde{\boldsymbol{\beta}} = \left(\widetilde{\beta}_{0c},\widetilde{\boldsymbol{\beta}}_c^{\mathrm{T}},\widetilde{\beta}_{02},\widetilde{\boldsymbol{\beta}}_2^{\mathrm{T}},\ldots,\widetilde{\beta}_{0C},\widetilde{\boldsymbol{\beta}}_C^{\mathrm{T}}\right)^{\mathrm{T}}$; $y_{ic}^* = \widetilde{\beta}_{0c} + \boldsymbol{x}_i^{\mathrm{T}}\widetilde{\boldsymbol{\beta}}_c + w_{ic}^{-1}\left(I_{\{y_i=c\}} - \widetilde{p}_c(\boldsymbol{x}_i)\right)$ is the "working response," $w_{ic} = \widetilde{p}_c(\boldsymbol{x}_i)\times(1-\widetilde{p}_c(\boldsymbol{x}_i))$, and $\widetilde{p}_c(\boldsymbol{x}_i)$ is $P(Y_i = c|\boldsymbol{x}_i)$ given in (7.6) but evaluated at the current parameter values.

When $\boldsymbol{X}$ is the design matrix with standardized independent variables, the updated parameters of block $c$, $\widehat{\boldsymbol{\beta}}_c^{*\mathrm{T}}$, can be obtained by the following formula:

$$\widehat{\boldsymbol{\beta}}_c^{*\mathrm{T}} = \left(\boldsymbol{X}^{*\mathrm{T}}\boldsymbol{W}_c\boldsymbol{X}^* + \lambda\boldsymbol{D}\right)^{-1}\boldsymbol{X}^{*\mathrm{T}}\boldsymbol{W}_c\boldsymbol{y}^*,$$

where $\boldsymbol{X}^* = [\boldsymbol{1}_n\,\boldsymbol{X}]$, $\boldsymbol{W}_c = \mathrm{Diag}(w_{1c},\ldots,w_{nc})$, $\boldsymbol{y}^* = \left(y_1^*,\ldots,y_n^*\right)^{\mathrm{T}}$, and $\boldsymbol{D}$ is an identity matrix of dimension $(p+1)\times(p+1)$ except that in the first entry we have the value of 0 instead of 1. However, in the context of $p \gg n$, a non-prohibited optimization of (7.9) is achieved by using coordinate descent methods as done in the glmnet package and commented in Chap. 3.6.2.

For other penalization terms, a very similar algorithm to the one described before can be used. For example, for Lasso penalty, the penalized likelihood (7.7) is modified as

$$\ell_p(\boldsymbol{\beta};\boldsymbol{y}) = \ell(\boldsymbol{\beta};\boldsymbol{y}) - \lambda\sum_{c=1}^{C}\sum_{j=1}^{p}\left|\beta_{cj}\right| \tag{7.10}$$

and block updating can be done as in (7.9), except that now $\boldsymbol{\beta}_c^{\mathrm{T}}\boldsymbol{\beta}_c$ is replaced by $\sum_{j=1}^{p}\left|\beta_{cj}\right|$. Like the penalized logistic regression studied in Chap. 3, the more common approach for choosing the "optimal" regularization parameter $\lambda$ in the penalized multinomial regression model in (7.7) is by using a $k$-fold cross-validation strategy with misclassification error as metrics. This will be used here. For more details, see Friedman et al. (2010). It is important to point out that the tuning parameter $\lambda$ used in glmnet is equal to the one used in the penalized log-likelihood (7.7) and (7.10) but divided by the number of observations.

### 7.4.1 Illustrative Examples for Multinomial Penalized Logistic Regression

**Example 4** To illustrate the penalized multinomial regression model, here we considered the ordinal data of Example 1, but considering the data as nominal, which under the prediction paradigm may not be so important. To evaluate the performance of this model with Ridge (7.9) and Lasso penalization (7.10), the same 10 random partitions as used in Example 1 were used in the cross-validation strategy. By combining these two penalizations and including different covariates information, the performance of six resulting models were evaluated: penalized Ridge multinomial logistic regression model (PRMLRM) with markers as predictors (PRMLRM-1); PRMLRM with environment and markers as predictors (PRMLRM-2); PRMLRM with environment, markers, and environment×marker interaction as predictors (PRMLRM-3); and the penalized Lasso multinomial logistic regression models, PLMLRM-1, PLMLRM-2, and PLMLRM-3, which have the same predictors as PRMLRM-1, PRMLRM-2, and PRMLRM-3, respectively.

The results are shown in Table 7.4, where again metrics BS and PCCC were computed for each partition. We can observe a similar performance of all models with both metrics. The greater difference in the average BS values across the 10 partitions was found between PRMLRM-3 (worse) model and the PLMLRM-1-PLMRM-2 (best) models, given a 1.34% greater average BS value of the first one compared to the other two. With respect to the other metric, the best average PCCC value (0.29) was obtained with model PRMLRM-2, which gave a 2.36% better performance than the average value of the worse PCCC performance obtained with models PRMLRM-1-2 (0.28).

All these models gave a better performance than all models fitted in Example 1, but only a slightly better performance than the models considered in Example 2, which contain additional information besides the marker information [the only inputs included in the models in Example 1 and PRMLRM-1 and PLMLRG-1], also included environment and environment×marker interaction as predictors. Specifically, even the simpler models in Table 7.4 that use only marker information in the predictor (PRMLRM-1 and PLMLRM-1) gave results comparable to the more complex and competitive models fitted in Example 2. The relative difference in the average BS value between the better models in each of these two sets of compared models (models in Example 2 vs. models in Table 7.4) was 1.25%, while the relative difference in the average PCCC was 3.57%. The R code to reproduce the results of models under Ridge penalization is in Appendix 4. The codes corresponding to the Lasso penalization models are the same as those for Ridge penalization but with the alpha value set to 1 instead of 0, as needs to be done in the basic code of the glmnet to fit the penalized Ridge multinomial logistic regression models:

```
    A = cv.glmnet(X, y, family='multinomial', type.measure = "class",
nfolds = 10, alpha = 0)
```

**Table 7.4** Brier score (BS) and proportion of cases correctly classified (PCCC) across 10 random partitions, with 80% of the total data set used for training and the rest for testing, under model (7.6), with Ridge and Lasso penalization and different covariable information: the penalized Ridge multinomial logistic regression (PRMLR) model with markers as covariates (PRMLRM-1); the PRMLR model with environment and markers as covariates (PRMLRM-2); the PRMLR model with environment, markers, and environment×marker interaction as covariates (PRMLRM-3); PLMLRM-1, PLRMLRM-2, and PLMLRM-3 denote the penalized Lasso multinomial logistic regression model with the same covariates as in PRMLRM-1, PRMLRM-2, and PRMLRM-3, respectively

| Model | PRMLRM-1 | | PRMLRM-2 | | PRMLRM-3 | |
|---|---|---|---|---|---|---|
| PT | BS | PCCC | BS | PCCC | BS | PCCC |
| 1 | 0.3755 | 0.3000 | 0.3755 | 0.3000 | 0.4632 | 0.3000 |
| 2 | 0.3831 | 0.3000 | 0.3831 | 0.3000 | 0.3957 | 0.2667 |
| 3 | 0.3953 | 0.2667 | 0.3954 | 0.2667 | 0.4052 | 0.4000 |
| 4 | 0.4093 | 0.2333 | 0.4099 | 0.2333 | 0.4271 | 0.0667 |
| 5 | 0.3578 | 0.3333 | 0.3578 | 0.3333 | 0.3578 | 0.3333 |
| 6 | 0.3639 | 0.3333 | 0.3639 | 0.3333 | 0.3639 | 0.3333 |
| 7 | 0.3885 | 0.2000 | 0.3885 | 0.2000 | 0.3885 | 0.2000 |
| 8 | 0.3696 | 0.3333 | 0.3700 | 0.3667 | 0.3746 | 0.3333 |
| 9 | 0.3926 | 0.2333 | 0.3926 | 0.2333 | 0.3926 | 0.2333 |
| 10 | 0.3955 | 0.3333 | 0.3956 | 0.3333 | 0.3913 | 0.2667 |
| Average (SD) | 0.3831 (0.01) | 0.2866 (0.05) | 0.3832 (0.01) | 0.29 (0.05) | 0.3959 (0.03) | 0.2733 (0.09) |
| Model | PLMLRM-1 | | PLMLRM-2 | | PLMLRM-3 | |
| PT | BS | PCCC | BS | PCCC | BS | PCCC |
| 1 | 0.3644 | 0.3333 | 0.3644 | 0.3333 | 0.3516 | 0.4333 |
| 2 | 0.3831 | 0.3000 | 0.3831 | 0.3000 | 0.3813 | 0.3000 |
| 3 | 0.3968 | 0.3000 | 0.3968 | 0.3000 | 0.3996 | 0.3000 |
| 4 | 0.4147 | 0.2333 | 0.4147 | 0.2333 | 0.4305 | 0.1667 |
| 5 | 0.3586 | 0.3000 | 0.3586 | 0.3000 | 0.3586 | 0.3000 |
| 6 | 0.3623 | 0.3000 | 0.3623 | 0.3000 | 0.3667 | 0.3000 |
| 7 | 0.3885 | 0.2000 | 0.3885 | 0.2000 | 0.3885 | 0.2000 |
| 8 | 0.3636 | 0.3667 | 0.3636 | 0.3667 | 0.3710 | 0.4000 |
| 9 | 0.3926 | 0.2333 | 0.3926 | 0.2333 | 0.3926 | 0.2333 |
| 10 | 0.3913 | 0.2667 | 0.3913 | 0.2667 | 0.3919 | 0.2667 |
| Average (SD) | 0.3815 (0.01) | 0.2833 (0.05) | 0.3815 (0.01) | 0.2833 (0.05) | 0.3832 (0.02) | 0.29 (0.08) |

where **X** and **y** are the design matrix and vector of response variable for training the multinomial model, and the multinomial model is specified as family='**multinomial**'; with "**class**" we specified the metric that will be used in the inner cross-validation (CV) strategy to internally select the "optimal" value of the regularization parameter $\lambda$ (over a grid of 100 values of $\lambda$, by default). The misclassification error is equal to $1 - \text{PCCC}$; nfolds is the number of folds that are used internally with the inner CV strategy and when this value is not specified, by default it is set to 10; alpha is a mixing parameter for a more general penalty function

(Elastic Net penalty) that when it is set to 0, the Ridge penalty is obtained, while the Lasso penalty is used when alpha is set to 1; however, when alpha takes values between 0 and 1, the Elastic Net penalization is implemented.

A kind of GBLUP implementation of the simpler models (PRMLRM-1 and PLMLRM-1) considered in Example 4 can be done by considering the genomic relationship matrix derived from the marker information and using as predictor the lower triangular factor of the Cholesky decomposition or its root squared matrix. Something similar can be done to include the Env×Line interaction term. The kinds of "GBLUP" implementation of PRMLRM-1 and PLMLRM-1 are referred to as GMLRM-R1 and GMLRM-L1, respectively. In both cases, the input matrix is $\boldsymbol{X} = \boldsymbol{Z}_L\boldsymbol{L}_g$, where $\boldsymbol{Z}_L$ is the incident matrix of the lines and $\boldsymbol{L}_g$ is the lower triangular part of the Cholesky decomposition of $\boldsymbol{G}$, $\boldsymbol{G} = \boldsymbol{L}_g\boldsymbol{L}_g^{\mathrm{T}}$. For the rest of the models, this kind of "GBLUP" implementation will be referred to as GMLRM-R2 and GMLRM-R3 for Ridge penalty, and as GMLRM-L2 and GMLRM-L3 for Lasso penalty, and in both cases the input matrix $\boldsymbol{X}$ will be $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g]$ when an environment and genetic effect is present in the predictor (GMLRM-R2, GMLRM-L3), or $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g, \boldsymbol{Z}_{EL}(\boldsymbol{I}_I \otimes \boldsymbol{L}_g)]$ when the genotype×environment interaction effect is also taken into account (GMLRM-R2, GMLRM-L3), and where $I$ is the number of environments.

The basic code for glmnet implementation of the GMLRM-L3 model is given below, from which the other five kinds of "GBLUP" implementations described above can be performed by only removing the corresponding predictors and changing to 0 the alpha value in the Ridge penalty:

```
Lg = t(chol(G))
#Environment matrix design
XE = model.matrix(~Env,data=dat_F)
#Matrix design of the genetic effect
ZL = model.matrix(~0+GID,data=dat_F)
ZLa = ZL%*%Lg
#Environment-genetic interaction
ZEL = model.matrix(~0+GID:Env,data=dat_F)
ZELa = ZEL%*%kronecker(diag(dim(XE)[2]),Lg)
X = cbind(XE[,-1],ZLa, ZELa)#Input X matrix
 A = cv.glmnet(X, y, family='multinomial', type.measure = "class",
              alpha = 1)#alpha=0 for Ridge penalty
```

**Example 5** Here we considered the data in Example 4 to illustrate the implementation of the following six kinds of "GBLUP" models: GMLRM-R1, GMLRM-R2, GMLRM-R3, GMLRM-L1, GMLRM-L2, and GMLRM-L3. To evaluate the performance of these models, the same CV strategy was used, where for each of the 10 random partitions, 80% of the full data set was taken to train the models and the rest to evaluate their performance.

**Table 7.5** Brier score (BS) and proportion of cases correctly classified (PCCC) across 10 random partitions, with 80% of the data used for training and the rest for testing, for multinomial model (7.6) under the six types of "GBLUP" implementation of the Ridge and Lasso penalization in Example 4 that were obtained by varying the input information: GMLRM-R1 ($\boldsymbol{X} = \boldsymbol{Z}_L\boldsymbol{L}_g$), GMLRM-R2 ($\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g]$), GMLRM-R3 ($\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g, \boldsymbol{Z}_{EL}(\boldsymbol{I}_I \otimes \boldsymbol{L}_g)]$), GMLRM-L1 ($\boldsymbol{X} = \boldsymbol{Z}_L\boldsymbol{L}_g$,), GMLRM-L2 ($\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g]$), and GMLRM-L3 ($\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g, \boldsymbol{Z}_{EL}(\boldsymbol{I}_I \otimes \boldsymbol{L}_g)]$), where the first three use Ridge penalization and the last three Lasso penalty

| Model | GMLRM-R1 | | GMLRM-R2 | | GMLRM-R3 | |
|---|---|---|---|---|---|---|
| PT | BS | PCCC | BS | PCCC | BS | PCCC |
| 1 | 0.3639 | 0.3000 | 0.3625 | 0.3333 | 0.3641 | 0.3333 |
| 2 | 0.3831 | 0.3000 | 0.3831 | 0.3000 | 0.4187 | 0.3000 |
| 3 | 0.3705 | 0.3333 | 0.3773 | 0.3000 | 0.4078 | 0.2333 |
| 4 | 0.4083 | 0.2667 | 0.4083 | 0.2333 | 0.4172 | 0.2667 |
| 5 | 0.3578 | 0.3333 | 0.3578 | 0.3333 | 0.3578 | 0.3333 |
| 6 | 0.3639 | 0.3333 | 0.3639 | 0.3333 | 0.3634 | 0.3667 |
| 7 | 0.3885 | 0.2000 | 0.3885 | 0.2000 | 0.3889 | 0.2000 |
| 8 | 0.3697 | 0.3667 | 0.3697 | 0.3667 | 0.3875 | 0.3333 |
| 9 | 0.3926 | 0.2333 | 0.3926 | 0.2333 | 0.3970 | 0.2667 |
| 10 | 0.3913 | 0.2667 | 0.3913 | 0.2667 | 0.3913 | 0.2667 |
| Average (SD) | 0.3789 (0.01) | 0.2933 (0.05) | 0.3795 (0.01) | 0.29 (0.05) | 0.3893 (0.02) | 0.29 (0.05) |
| Model | GMLRM-L1 | | GMLRM-L2 | | GMLRM-L3 | |
| PT | BS | PCCC | BS | PCCC | BS | PCCC |
| 1 | 0.3865 | 0.3667 | 0.3505 | 0.4000 | 0.3598 | 0.3667 |
| 2 | 0.3831 | 0.3000 | 0.3831 | 0.3000 | 0.3806 | 0.3000 |
| 3 | 0.3946 | 0.2667 | 0.3946 | 0.2667 | 0.3948 | 0.2667 |
| 4 | 0.4147 | 0.2333 | 0.4147 | 0.2333 | 0.4147 | 0.2333 |
| 5 | 0.3564 | 0.3333 | 0.3564 | 0.3333 | 0.3823 | 0.2667 |
| 6 | 0.3578 | 0.3667 | 0.3578 | 0.3667 | 0.3639 | 0.3333 |
| 7 | 0.4078 | 0.1333 | 0.4078 | 0.1333 | 0.4291 | 0.1333 |
| 8 | 0.3670 | 0.3667 | 0.3670 | 0.3667 | 0.3684 | 0.3667 |
| 9 | 0.3926 | 0.2333 | 0.3926 | 0.2333 | 0.3926 | 0.2333 |
| 10 | 0.3854 | 0.3333 | 0.3854 | 0.3333 | 0.3913 | 0.2667 |
| Average (SD) | 0.3845 (0.01) | 0.2933 (0.07) | 0.3809 (0.02) | 0.2966 (0.08) | 0.3877 (0.02) | 0.2766 (0.07) |

The results presented in Table 7.5 are comparable to those given in Table 7.4 for Example 4. The relative difference between the models in Table 7.4 with better average BS value (PLMLRM-1) and the model in Table 7.5 with better average BS value (GMLRM-R1) is 0.6990%, in favor of the latter model. Regarding the average PCCC, the "kind" of GBLUP model with better value, GMLRM-L2, was 2.2989% greater than the average PCCC value of the better models in Table 7.4, PMLR-R2 and PMLR-L3. Furthermore, for all models in Table 7.5, the greatest difference between the average BS values was observed in models GMLRM-1 (better) and GMLRM-R3 (worse), a relative difference of 2.748%. The best average PCCC performance was observed with model GMLRM-L2, while the worst one was with model GMLRM-L3, a relative difference of 7.23% between the best and the worst.

The R code to reproduce the results in Table 7.5 for the first three models, see Appendix 5. By only changing the value of alpha to 1, the corresponding R code allows implementing the last three Lasso penalty models.

## 7.5 Penalized Poisson Regression

In contrast to ordinal data, sometimes the response value is not known to be upper bound, rather it is often a count or frequency data, for example, the number of customers per minute in a specific supermarket, number of infected spikelets per plant, number of defects per unit, number of seeds per plant, etc. The most often used count model to relate a set of explanatory variables with a count response variable is Poisson regression.

Given vector covariates $\boldsymbol{x}_i = (x_{i1}, \ldots, x_{ip})^{\mathrm{T}}$, the Poisson log-linear regression modeled the number of events $Y_i$, as a Poisson random variable with mass density

$$P(Y_i = y|\boldsymbol{x}_i) = \frac{\lambda_i^y \exp(-\lambda_i)}{y!}, y = 0, 1, 2, \ldots, \tag{7.11}$$

where the expected value of $Y_i$ is given by $\lambda_i = \exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right)$ and $\boldsymbol{\beta}_0 = (\beta_1, \ldots, \beta_p)^{\mathrm{T}}$ is the vector of beta coefficients effect of the covariates. For example, $Y_i$ may be the number of defects of a product under specific production conditions (temperature, raw material, schedule, operator, etc.) or the number of spikelets per plant which can be predicted from environment and marker information, etc.

For the parameter estimation of this model, the more usual method is maximum likelihood estimation, but this is not suitable in the context of a large number of covariates, so here, as in the logistic and multinomial models, we describe the penalized likelihood method. For a Ridge penalty, the penalized log-likelihood function is given by

$$\begin{aligned}\ell_p(\beta_0, \boldsymbol{\beta}_0; \boldsymbol{y}) &= \sum_{i=1}^{n} \log\left[f_{Yi}(y_i; \beta_0, \boldsymbol{\beta}_0)\right] - \lambda \sum_{j=1}^{p} \beta_j^2 \\ &= \sum_{i=1}^{n} y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right) - \sum_{i=1}^{n} \exp\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}_0\right) - \sum_{i=1}^{n} \log(y_i!) - \frac{\lambda}{2} \sum_{j=1}^{p} \beta_j^2\end{aligned}.$$

This can be optimized directly by using traditional numerical methods, but in a general genomic prediction context where $p \gg n$, other alternatives could be more useful. One is the same procedure described before for multinomial and logistic regressions, in which the log-likelihood is replaced by a second-order approximation at the current beta coefficients and then the resulting weighted least-squares problem is solved by coordinate descent methods. This method for Poisson regression is also

implemented in the glmnet package and the basic code for implementing this is as follows:

```
A = cv.glmnet(X, y, family='poisson', type.measure = "mse", nfolds =
10,
      alpha = 0)
```

where again **X** and **y** are the design matrix and vector of response variable for training the Poisson model which is specified by family='**poisson**'; the metric that will be used internally in the inner cross-validation (CV) strategy to choose the "optimal" regularization parameter $\lambda$ (over a grid of 100 values of $\lambda$, by default) is specified in "type.measure". Additionally, to the mean square error (**mse**), the deviance (**deviance**) and the mean absolute error (**mae**) can also be adopted. The other parameters are the same as described for logistic and multinomial regression models. For implementing the Lasso Poison regression (Poisson regression with Lasso penalty), this same code can be used by only making the argument alpha = 1, instead of 0. Further, in the genomic context, as accomplished before for other kinds of responses (continuous, binary, nominal), with this function more complex models involving marker (genetic), environment, or/and marker × environment interaction effects can also be implemented.

Like the predictors described for the multinomial model, all these predictors are shown in Table 7.6, where the first three (PRPRM-1–3) correspond to Poisson Ridge

**Table 7.6** Penalized Poisson regression models with Ridge or Lasso penalties and different covariate information: penalized Ridge Poisson regression (PRPRM) model with markers as covariates (PRPRM-1); PRPRM model with environment and markers as covariates (PRPRM-2); PRPRM model with environment, markers, and environment ×marker interaction as covariates (PRPRM-3); PLPRM-1, PLPRM-2, and PLPRM-3 denote the corresponding penalized Lasso Poisson regression models. GMLRM-R1–R3 are the corresponding GBLUP implementation of the PRPRM models, while GMLRM-L1–L3 are the respective GBLUP of the PLPRM models. $\boldsymbol{X}_E$, $\boldsymbol{X}_M$, and $\boldsymbol{X}_{EM}$ are the design matrices of environment, markers and environment ×marker interaction, $\boldsymbol{Z}_L$ and $\boldsymbol{Z}_{EL}$ are the incident matrices of lines and environment ×line interaction effects, and $\boldsymbol{L}_g$ is the lower triangular part of the Cholesky decomposition of $\boldsymbol{G}$, $\boldsymbol{G} = \boldsymbol{L}_g\boldsymbol{L}_g^T$

| Penalization | Model | Predictor effects | Matrix design |
|---|---|---|---|
| Ridge | PRPRM-1 | Markers | $\boldsymbol{X} = \boldsymbol{X}_M$ |
| | PRPRM-2 | Env + Markers | $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{X}_M]$ |
| | PRPRM-3 | Env + Markers + Env ×Marker | $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{X}_M, \boldsymbol{X}_{EM}]$ |
| Lasso | PLPRM-1 | Markers | $\boldsymbol{X} = \boldsymbol{X}_M$ |
| | PLPRM-2 | Env + Markers | $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{X}_M]$ |
| | PLPRM-3 | Env + Markers + Env ×Marker | $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{X}_M, \boldsymbol{X}_{EM}]$ |
| | GPRM-R1 | Genetic | $\boldsymbol{X} = \boldsymbol{Z}_L\boldsymbol{L}_g$ |
| Ridge | GPRM-R2 | Env + Genetic | $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g]$ |
| | GPRM-R3 | Env + Genetic + Env ×Genetic | $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g, \boldsymbol{Z}_{EL}(\boldsymbol{I}_I \otimes \boldsymbol{L}_g)]$ |
| | GPRM-L1 | Genetic | $\boldsymbol{X} = \boldsymbol{Z}_L\boldsymbol{L}_g$ |
| Lasso | GPRM-L2 | Env + Genetic | $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g]$ |
| | GPRM-L3 | Env + Genetic + Env× Genetic | $\boldsymbol{X} = [\boldsymbol{X}_E, \boldsymbol{Z}_L\boldsymbol{L}_g, \boldsymbol{Z}_{EL}(\boldsymbol{I}_I \otimes \boldsymbol{L}_g)]$ |

regression with marker, environment plus marker, and environment plus markers plus environment×marker interaction effects, respectively. The second group of three models (PLPRM-1–3) are, respectively, the same as before but with the Lasso penalization. The third group—three models (GRPRM-R1–R3)—are the corresponding GBLUP implementation of the first three predictors, while the last group—three models (GLPRM-L1–L3)—corresponds to the GBLUP implementation of the second group—three Lasso penalized models.

**Example 6** To illustrate how to fit the Poisson model with the predictors given in Table 7.6 and compare these in terms of the mean squared error of prediction, we considered a small data set that is part of the data set used in Montesinos-López et al. (2016). It consists of 50 lines in three environments and a total of 1635 markers. The performance evaluation was done using 10 random partitions where 80% of the data was used to train the model and the rest to test the model. The glmnet R package was used to train all these regression models.

The results are shown in Table 7.7, where the first six models are the first six penalized Poisson regression models given in Table 7.6. With respect to the Ridge penalized models (PRPRM-1–3), we can observe a slightly better performance of the more complex models that take into account environment, marker, and environment ×marker interaction effects (PRPRM-3). As for the penalized Lasso Poisson regression models (PLPRM-1–3), but with a more notable difference, the more complex models also resulted in better performance, and this was also better than the penalized Ridge model (PRPRM-3).

Furthermore, in its GBLUP implementation of the Poisson regression models with Ridge penalty, better MSE performance was obtained with the model that includes environment and genetic effects (GPRM-R2) and this was the best among all Ridge penalized models (PRPRM-1–3 and GPRM-R1–R3); the average MSE of the best PRPRM (PRPRM-3) is 7.5668 greater than the average MSE of the best GPRM with Ridge penalty (GPRM-R2). Under the GBLUP implementations of the Poisson regression models with Lasso penalty (GPRM-L1–L3), the best performance prediction was also obtained when using the environment and genetic effects (GPRM-R2), but its average MSE was slightly different (by only 2.71%) than that obtained with the average MSE of the best Lasso penalized Poisson regression model (PLPRM-3). Additionally, this GBLUP implementation (GPRM-L2) also showed the best average MSE performance among all 12 implemented models, and a notable difference with the second best (GPRM-R2) that gave a relative MSE that was 12.01% greater than that for GPRM-L2 model.

The worse average MSE performance of model **GPRM-L3** is because of the high value of the MSE obtained in partition 2. However, this high variability of the MSEP observed across partitions can be unexpected for other larger data sets.

The R code to reproduce these results is shown in Appendix 6.

**Table 7.7** Mean square error of prediction (MSE) of six penalized Poisson regression models with Ridge or Lasso penalties and different covariate information (first six models) and MSE of the GBLUP implementation of the corresponding models (second six models), across 10 random partitions, with 80% of the total data set used for training and the rest for testing. See Table 7.6 for details of each model

| Model | PRPRM-1 | PRPRM-2 | PRPRM-3 | PLPRM-1 | PLPRM-2 | PLPRM-3 |
|---|---|---|---|---|---|---|
| PT | MSE | MSE | MSE | MSE | MSE | MSE |
| 1 | 3.7346 | 3.7380 | 3.5678 | 3.7573 | 3.0309 | 3.0019 |
| 2 | 2.0492 | 2.1012 | 2.1012 | 2.1194 | 2.0408 | 2.0511 |
| 3 | 2.6610 | 2.6757 | 2.7178 | 2.6642 | 2.1885 | 2.1930 |
| 4 | 4.6873 | 4.7079 | 4.4640 | 4.5714 | 4.3558 | 3.8514 |
| 5 | 3.4277 | 3.4252 | 3.3573 | 3.3317 | 2.5760 | 2.6570 |
| 6 | 2.6719 | 2.6589 | 2.6592 | 2.9291 | 2.1381 | 2.1104 |
| 7 | 2.7093 | 2.6601 | 2.6601 | 2.6601 | 3.2845 | 2.4267 |
| 8 | 3.8690 | 3.8791 | 3.7156 | 3.7156 | 3.1182 | 3.1605 |
| 9 | 1.9520 | 1.9699 | 1.8081 | 1.8081 | 1.5642 | 1.7106 |
| 10 | 4.6182 | 4.6086 | 4.5979 | 4.6416 | 3.6876 | 3.8178 |
| Average (SD) | 3.238 (0.98) | 3.2424 (0.97) | 3.1648 (0.93) | 3.2198 (0.96) | 2.7984 (0.85) | 2.698 (0.74) |
| Model | GPRM-R1 | GPRM-R2 | GPRM-R3 | GPRM-L1 | GPRM-L2 | GPRM-L3 |
| PT | MSE | MSE | MSE | MSE | MSE | MSE |
| 1 | 3.7110 | 3.4279 | 3.8221 | 3.8243 | 2.8399 | 2.9275 |
| 2 | 2.1012 | 1.7678 | 2.1050 | 2.1012 | 1.8690 | 69.3985 |
| 3 | 2.6606 | 2.2621 | 2.7178 | 2.7178 | 2.0578 | 1.9326 |
| 4 | 4.7780 | 4.8127 | 4.2797 | 4.2820 | 3.9353 | 3.3327 |
| 5 | 3.5584 | 3.0863 | 3.5038 | 3.3910 | 2.6122 | 2.6139 |
| 6 | 2.6117 | 2.0641 | 2.8482 | 2.7042 | 2.0844 | 1.9349 |
| 7 | 2.7123 | 2.5304 | 2.9432 | 2.6601 | 2.7140 | 2.3736 |
| 8 | 3.7156 | 3.5213 | 3.7107 | 3.6467 | 2.9335 | 3.0768 |
| 9 | 1.8100 | 1.6769 | 1.8005 | 1.8081 | 1.4778 | 1.6548 |
| 10 | 4.5861 | 4.2733 | 4.5821 | 4.8303 | 3.7439 | 4.1025 |
| Average (SD) | 3.2244 (1.0) | 2.9422 (1.06) | 3.2312 (0.9) | 3.1965 (0.96) | 2.6267 (0.79) | 9.3347 (21.11) |

## 7.6 Final Comments

In this chapter, we give the fundamentals of some popular Bayesian and classical regression models for categorical and count data. We also provide many practical examples for implementing these models for genomic prediction. The examples used take into account in the predictor not only the effect of markers or genotypes but also illustrate how to take into account the effects of environment, genotype × environment interaction, and marker × environment interaction. Also, for each type of response variables, we calculated the prediction performance using appropriate

metrics, which is very important since the metrics for evaluating the prediction performance are dependent upon the type of response variable. It is important to point out that the components to include in the predictor are not restricted to those illustrated here, since the user can include other components as main effects or interaction effects in similar fashion as illustrated in the examples provided.

## Appendix 1

```
rm(list=ls(all=TRUE))
library(BGLR)
load('dat_ls.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
#Marker information
dat_M = dat_ls$dat_M
#Trait response values
y = dat_F$y
summary(dat_F)
#Sorting data phenotypic data set first by Environment, and then by GID
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]
#10 random partitions
K = 10; n = length(y)
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

#Brier score function
BS_f<-function(y,p)
{
 n = length(y)
 CM = matrix(0,nr=n,nc=dim(p)[2])
 CM[cbind(1:n,y)]<-1
 mean(rowSums((p-CM)^2))/2
}
#Matrix design of markers for all observations
Pos = match(dat_F$GID,row.names(dat_M))
X = dat_M[Pos,]
X = scale(X)
#Ordinal BRR model
ETA_BRR = list(list(X=X,model='BRR'))
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_BRR,
      nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst],A$probs[Pos_tst,])
```

```
 Tab$PCC[k] = mean(y[Pos_tst]==apply(A$probs,1,which.max)[Pos_tst])
}
Tab
#Ordinal Bayesian GBLUP model
#Genomic relationship matrix
X_M = scale(dat_M)
G = (X_M%*%t(X_M))/dim(X_M)[2]
dat_F$GID = factor(dat_F$GID,levels=colnames(G))
Z_L = model.matrix(~0+GID,data=dat_F)
Ga = Z_L%*%G%*%t(Z_L)
ETA_GBLUP = list(list(K=Ga,model='RKHS'))
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_GBLUP,
       nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst],A$probs[Pos_tst,])
 Tab$PCC[k] = mean(y[Pos_tst]==apply(A$probs,1,which.max)[Pos_tst])
}
Tab

#Ordinal BayesA model
ETA_BA = list(list(X=X,model='BayesA'))
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_BA,
       nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst],A$probs[Pos_tst,])
 Tab$PCC[k] = mean(y[Pos_tst]==apply(A$probs,1,which.max)[Pos_tst])
}
Tab
#Ordinal BayesB model
ETA_BB = list(list(X=X,model='BayesB'))
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_BB,
       nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst],A$probs[Pos_tst,])
 Tab$PCC[k] = mean(y[Pos_tst]==apply(A$probs,1,which.max)[Pos_tst])
```

```
}
Tab
#Ordinal BayesC model
ETA_BC = list(list(X=X,model='BayesC'))
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_BC,
      nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst],A$probs[Pos_tst,])
 Tab$PCC[k] = mean(y[Pos_tst]==apply(A$probs,1,which.max)[Pos_tst])
}
Tab

#Ordinal BL model
ETA_BL = list(list(X=X,model='BL'))
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_BL,
      nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst],A$probs[Pos_tst,])
 Tab$PCC[k] = mean(y[Pos_tst]==apply(A$probs,1,which.max)[Pos_tst])
}
Tab
```

# Appendix 2

```
rm(list=ls(all=TRUE))
library(BGLR)
load('dat_ls.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
#Marker information
dat_M = dat_ls$dat_M
y = dat_F$y
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]
#PT
#10 random partitions
K = 10
n = length(y)
set.seed(1)
```

```
PT = replicate(K,sample(n,0.20*n))
#Brier score
BS_f<-function(y,p)
{
 n = length(y)
 CM = matrix(0,nr=n,nc=dim(p)[2])
 CM[cbind(1:n,y)]<-1
 mean(rowSums((p-CM)^2))/2
}
#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
X = dat_M[Pos,]
X = scale(X)

#Environment matrix design
XE = model.matrix(~Env,data=dat_F)[,-1]
#Environment-marker interaction
Envs = unique(dat_F$Env)
Markers = colnames(X)
XEM = model.matrix(~0+X:Env,data=dat_F)
Pos = !(colnames(XEM)%in%paste0('X',Markers,':Env',Envs[length
(Envs)]))
XEM = XEM[,Pos]

#Models to evaluate
Models = c('BRR','BayesA','BayesB','BayesC','BL')
for(m in 1:5)
{
 ETA = list(list(X=XE,model='FIXED'),list(X=X,model=Models[m]),
       list(X=XEM,model=Models[m]))
 Tab = data.frame(PT = 1:K,BS=NA)
 set.seed(1)
 for(k in 1:K)
 {
  Pos_tst = PT[,k]
  y_NA = y
  y_NA[Pos_tst] = NA
  A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA,
       nIter = 1e4,burnIn = 1e3,verbose = FALSE)
  Tab$BS[k]  = BS_f(y[Pos_tst],A$probs[Pos_tst,])
 Tab$PCC[k] = mean(y[Pos_tst]==apply(A$probs,1,which.max)[Pos_tst])
  if(dim(A$probs)[2]<5) stop
 }
#Saving the output (Tab) for each model
 write.csv(Tab,file=paste0('Tab_M3',Models[m],'.csv'),row.names =
FALSE)
}

#GBLUP regression model (4)
#Genomic relationship matrix
G = (dat_M%*%t(dat_M))/dim(dat_M)[2]
dat_F$GID = factor(dat_F$GID,levels=colnames(G))
Z_L = model.matrix(~0+GID,data=dat_F)
```

```
Ga = Z_L%*%G%*%t(Z_L)
#"Genomic" relationship matrix for interaction
Ga2 = kronecker(diag(3),G)
ETA_GBLUP = list(list(K=Ga,model='RKHS'),
          list(X=XE,model='FIXED'),
          list(K=Ga2,model='RKHS'))
Tab = data.frame(PT = 1:K,BS=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_GBLUP,
      nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst],A$probs[Pos_tst,])
 Tab$PCC[k] = mean(y[Pos_tst]==apply(A$probs,1,which.max)[Pos_tst])
}
Tab
```

# Appendix 3

```
rm(list=ls(all=TRUE))
library(BGLR)
load('Data_Toy_EYT.RData',verbose=TRUE)
#Phenotypic data
dat_F = Pheno_Toy_EYT
head(dat_F)
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]
y = dat_F$Height
#Genomic relationship matrix
G = G_Toy_EYT
#PT
#10 random partitions
K = 10
n = length(y)
set.seed(1)
PT = replicate(K,sample(n,0.20*n))
#Brier score
#The min value of y must be 1
BS_f<-function(y,p)
{
 n = length(y)
 CM = matrix(0,nr=n,nc=dim(p)[2])
 CM[cbind(1:n,y)]<-1
 mean(rowSums((p-CM)^2))/2
}
#M3
dat_F$GID = factor(dat_F$GID,levels=colnames(G))
Z_L = model.matrix(~0+GID,data=dat_F)
```

```
Ga = Z_L%*%G%*%t(Z_L)
ETA_GBLUP = list(list(K=Ga,model='RKHS'))
#GBLUP
Tab = data.frame(PT = 1:K,BS=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_GBLUP,
      nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst]+1,A$probs[Pos_tst,])
 lbs = as.numeric(colnames(A$probs))
 yp = apply(A$probs,1,function(x)lbs[which.max(x)])
 Tab$PCC[k] = mean(y[Pos_tst]==yp[Pos_tst])
}
Tab

#M4
#Environment matrix design
XE = model.matrix(~Env,data=dat_F)[,-1]
#"Genomic" relationship matrix for interaction
Ga2 = kronecker(diag(4),G)
ETA_GBLUP = list(list(X=XE,model='FIXED'),
          list(K=Ga,model='RKHS'),
          list(K=Ga2,model='RKHS'))
#GBLUP
Tab = data.frame(PT = 1:K,BS=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_GBLUP,
      nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst]+1,A$probs[Pos_tst,])
 lbs = as.numeric(colnames(A$probs))
 yp = apply(A$probs,1,function(x)lbs[which.max(x)])
 Tab$PCC[k] = mean(y[Pos_tst]==yp[Pos_tst])
}
Tab

#M5
#Environment matrix design
XE = model.matrix(~Env,data=dat_F)[,-1]
#mean(G!=Ga2)
ETA_GBLUP = list(list(K=Ga,model='RKHS'),
          list(X=XE,model='FIXED'))
Tab = data.frame(PT = 1:K,BS=NA)
set.seed(1)
for(k in 1:K)
```

```
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,response_type='ordinal',ETA=ETA_GBLUP,
      nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 Tab$BS[k]  = BS_f(y[Pos_tst]+1,A$probs[Pos_tst,])
 lbs = as.numeric(colnames(A$probs))
 yp = apply(A$probs,1,function(x)lbs[which.max(x)])
 Tab$PCC[k] = mean(y[Pos_tst]==yp[Pos_tst])
}
Tab
```

## Appendix 4 (Example 4)

```
rm(list=ls(all=TRUE))
library(glmnet)
load('dat_ls.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
#Marker information
dat_M = dat_ls$dat_M
#Response variable
y = dat_F$y
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]
head(dat_F)
#PT
#10 random partitions
K = 10
n = length(y)
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

#Brier score function
BS_f<-function(y,p)
{
 n = length(y)
 CM = matrix(0,nr=n,nc=dim(p)[2])
 CM[cbind(1:n,y)]<-1
 mean(rowSums((p-CM)^2))/2
}
#PRMLRM-1
#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
X = dat_M[Pos,]
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
```

```
 y_tr = as.character(y[-Pos_tst])
 X_tr = X[-Pos_tst,]
 A = cv.glmnet(X_tr, y_tr, family='multinomial',
        type.measure = "class",# Misclassification error
        nfolds = 10, parallel = FALSE,thresh=1e-10,
        alpha = 0) # alpha=1 for lasso penalization
 p_tst = predict(A,newx = X[Pos_tst,],type='response',s=A$lambda.
min)[,,1]
 y_tst = y[Pos_tst]
 Tab$BS[k]  = BS_f(y_tst,p_tst)
 yp_tst = predict(A,newx = X[Pos_tst,],type='class',s=A$lambda.min)
 yp_tst = as.numeric(yp_tst)
 Tab$PCC[k] = mean(y_tst==yp_tst)
}
Tab
#PRMLRM-2
# Marker + Env effect
#Environment_marker interaction
XE = model.matrix(~Env,data=dat_F)[,-1]
#Environment matrix design
XEM = model.matrix(~0+X:Env,data=dat_F)
dim(XEM)
head(XEM)[,1:5]
#XEMa = kronecker(diag(3),unique(X))
#sum((XEM-XEMa)^2)
#Matrix desing for PMR
X = cbind(XE,X)
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_tr = as.character(y[-Pos_tst])
 X_tr = X[-Pos_tst,]
 A = cv.glmnet(X_tr, y_tr, family='multinomial',
        type.measure = "class",
        nfolds = 10, parallel = FALSE,thresh=1e-10,
        alpha = 0) # alpha=1 for lasso penalization
 p_tst = predict(A,newx = X[Pos_tst,],type='response',s=A$lambda.
min)[,,1]
 y_tst = y[Pos_tst]
 Tab$BS[k]  = BS_f(y_tst,p_tst)
 yp_tst = predict(A,newx = X[Pos_tst,],type='class',s=A$lambda.min)
 yp_tst = as.numeric(yp_tst)
 Tab$PCC[k] = mean(y_tst==yp_tst)
}
Tab
#PRMLRM-3
#Marker + Env + Env:Marker effect
#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
X = dat_M[Pos,]
#Environment_marker interaction
```

```
XEM = model.matrix(~0+X:Env,data=dat_F)
X = cbind(XE,X,XEM)
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_tr = as.character(y[-Pos_tst])
 X_tr = X[-Pos_tst,]
 A = cv.glmnet(X_tr, y_tr, family='multinomial',
        type.measure = "class",
        nfolds = 10, parallel = FALSE,thresh=1e-10,
        alpha = 0) # alpha=1 for lasso penalization
 #plot(A)
 p_tst = predict(A,newx = X[Pos_tst,],type='response',s=A$lambda.
min)[,,1]
 y_tst = y[Pos_tst]
 Tab$BS[k]  = BS_f(y_tst,p_tst)
 yp_tst = predict(A,newx = X[Pos_tst,],type='class',s=A$lambda.min)
 yp_tst = as.numeric(yp_tst)
 Tab$PCC[k] = mean(y_tst==yp_tst)
}
Tab
```

## Appendix 5

```
rm(list=ls(all=TRUE))
library(glmnet)
load('dat_ls.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
#Marker information
dat_M = dat_ls$dat_M
y = dat_F$y
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]
#PT
#10 random partitions
K = 10
n = length(y)
set.seed(1)
PT = replicate(K,sample(n,0.20*n))
#Brier score function
BS_f<-function(y,p)
{
 n = length(y)
 CM = matrix(0,nr=n,nc=dim(p)[2])
 CM[cbind(1:n,y)]<-1
 mean(rowSums((p-CM)^2))/2
}
#GMLRM-R1: Genomic effects
```

```
#Matrix design of markers
X  = dat_M
#GIDs = sort(row.names(X))
#X  = X[match(GIDs,row.names(X)),]
G  = tcrossprod(X)/dim(X)[2]
dat_F$GID = factor(dat_F$GID,levels=row.names(G))
ZL = model.matrix(~0+GID,data=dat_F)
Lg = t(chol(G))#Lower triangular part of the Cholesky descomposition of
G
#Design matrix to use with glmnet
ZLa  = ZL%*%Lg
X =  ZLa
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_tr = as.character(y[-Pos_tst])
 X_tr = X[-Pos_tst,]
 A = cv.glmnet(X_tr, y_tr, family='multinomial',
        type.measure = "class",# Misclassification error
        nfolds = 10, parallel = FALSE,thresh=1e-10,
        alpha = 0)#take alpha=1 for lasso penalty, GMLRM-L1
 #Prediction of testing set
 p_tst = predict(A,newx =  X[Pos_tst,],type='response',s=A$lambda.
min)[,,1]
 y_tst = y[Pos_tst]
 Tab$BS[k]  = BS_f(y_tst,p_tst)
 yp_tst = predict(A,newx =  X[Pos_tst,],type='class',s=A$lambda.min)
 yp_tst =  as.numeric(yp_tst)
 Tab$PCC[k] = mean(y_tst==yp_tst)
}
Tab
#GMLRM-R2: Genomic + Env effect
#Matrix design of markers
ZLa  = ZL%*%Lg
#Environment_marker interaction
XE = model.matrix(~Env,data=dat_F)
#Matrix design to use in glmnet
X = cbind(XE[,-1],ZLa)
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_tr = as.character(y[-Pos_tst])
 X_tr = X[-Pos_tst,]
 A = cv.glmnet(X_tr, y_tr, family='multinomial',
        type.measure = "class",
        nfolds = 10, parallel = FALSE,thresh=1e-10,
        alpha = 0)# take alpha=1 for lasso penalty, GMLRM-L2
 p_tst = predict(A,newx =  X[Pos_tst,],type='response',s=A$lambda.
min)[,,1]
```

```
 y_tst = y[Pos_tst]
 Tab$BS[k]  = BS_f(y_tst,p_tst)
 #Prediction of testing set
 yp_tst = predict(A,newx = X[Pos_tst,],type='class',s=A$lambda.min)
 yp_tst = as.numeric(yp_tst)
 Tab$PCC[k] = mean(y_tst==yp_tst)
}
Tab

#GMLRM-R3: Genomic + Env effect + Env:Marker effect
#Matrix design of markers
ZLa = ZL%*%Lg
#Environment-genetic interaction
ZEL = model.matrix(~0+GID:Env,data=dat_F)
ZELa = ZEL%*%kronecker(diag(dim(XE)[2]),Lg)
#Input matrix to use in glmnet
X = cbind(XE,ZLa,ZELa)
Tab = data.frame(PT = 1:K,BS=NA,PCC=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_tr = as.character(y[-Pos_tst])
 X_tr = X[-Pos_tst,]
 A = cv.glmnet(X_tr, y_tr, family='multinomial',
        type.measure = "class",
        nfolds = 10, parallel = FALSE,thresh=1e-10,
        alpha = 0)# alpha=1 for lasso penalty, GMLRM-L3
 p_tst = predict(A,newx = X[Pos_tst,],type='response',s=A$lambda.
min)[,,1]
 y_tst = y[Pos_tst]
 Tab$BS[k]  = BS_f(y_tst,p_tst)
  #Prediction of testing set
 yp_tst = predict(A,newx = X[Pos_tst,],type='class',s=A$lambda.min)
 yp_tst = as.numeric(yp_tst)
 Tab$PCC[k] = mean(y_tst==yp_tst)
}
Tab
```

# Appendix 6

```
rm(list=ls(all=TRUE))
library(glmnet)
load('dat_ls.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
#Marker information
dat_M = dat_ls$dat_M
y = dat_F$y
dat_F$Env= as.character(dat_F$Loc)
```

```
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]
head(dat_F)
#PT
#10 random partitions
K = 10
n = length(y)
set.seed(1)
PT = replicate(K,sample(n,0.20*n))
#Penalized Ridge (lasso) Poisson regression models
#PRPRM-1-3 (PLPRM-1-3)
#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
X = dat_M[Pos,]
#Environment matrix design
XE = model.matrix(~Env,data=dat_F)[,-1]
#Environment-marker interaction
Envs = unique(dat_F$Env)
Markers = colnames(X)
XEM = model.matrix(~0+X:Env,data=dat_F)
Pos = !(colnames(XEM)%in%paste0('X',Markers,':Env',Envs[length
(Envs)]))
XEM = XEM[,Pos]
X = X# Design matrix for PRPRM-1 and PLPRM-1
#X = cbind(XE,X) # Uncomment this line to implement PRPRM-2 or PLPRM-2
#X = cbind(XE,X,XEM) # Uncomment this line to implement PRPRM-3 or PLPRM-
3
Tab = data.frame(PT = 1:K,MSEP=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_tr = y[-Pos_tst]
 X_tr = X[-Pos_tst,]
 A = cv.glmnet(X_tr, y_tr, family='poisson',
        type.measure = "mse",
        nfolds = 10, parallel = FALSE,thresh=1e-10,
        alpha = 0)# Take alpha=1 for lasso penalty
 yp_tst = predict(A,newx = X[Pos_tst,],type='response',s=A$lambda.
min)
 y_tst = y[Pos_tst]
 Tab$MSEP[k]  = mean((y_tst-yp_tst)^2)

}
Tab

#GBLUP implementation of penalized Ridge (lasso) Poisson regression
models
#GPRM-R1-R3 (GPRM-L1-L3)
#Matrix design of Environments
XE = model.matrix(~Env,data=dat_F)
#Matrix design of markers
X  = dat_M
G  = tcrossprod(X)/dim(X)[2] # Genomic relationship matrix
```

```
dat_F$GID = factor(dat_F$GID,levels=row.names(G))
dat_F = dat_F[order(dat_F$Env,dat_F$GID),]
ZL = model.matrix(~0+GID,data=dat_F)
mean(substring(colnames(ZL),4)!=row.names(G))
Lg = t(chol(G))
#Matrix design of genetic effects
ZLa = ZL%*%Lg
#Environment-genetic interaction matrix design
ZEL = model.matrix(~0+GID:Env,data=dat_F)
ZELa = ZEL%*%kronecker(diag(dim(XE)[2]),Lg)
X = ZLa # Matrix design for GPRM-R1 and GPRM-L1
#X = cbind(XE[,-1],ZLa)# Uncomment this line to implement GPRM-R2 or
GPRM-L2
#X = cbind(XE[,-1],ZLa,ZELa) )# Uncomment this line to implement GPRM-
R3 or GPRM-L3
Tab = data.frame(PT = 1:K,MSEP=NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_tr = y[-Pos_tst]
 X_tr = X[-Pos_tst,]
 A = cv.glmnet(X_tr, y_tr, family='poisson',
        type.measure = "mse",
        nfolds = 10, parallel = FALSE,thresh=1e-10,
        alpha = 0)# Take alpha=1 for lasso penalty
 yp_tst = predict(A,newx = X[Pos_tst,],type='response',s=A$lambda.
min)
 y_tst = y[Pos_tst]
 Tab$MSEP[k]  = mean((y_tst-yp_tst)^2)
}
Tab
```

## References


Agresti A (2012) Categorical data analysis, 3rd edn. Wiley, Hoboken, NJ

Albert JH, Chib S (1993) Bayesian analysis of binary and polychotomous response data. J Am Stat Assoc 88(422):669–679

Friedman J, Hastie T, Tibshirani R (2010) Regularization paths for generalized linear models via coordinate descent. J Stat Softw 33(1):1–22. http://www.jstatsoft.org/v33/i01/

Gianola D (1980) A method of sire evaluation for dichotomies. J Anim Sci 51(6):1266–1271

Gianola D (1982) Theory and analysis of threshold characters. J Anim Sci 54(5):1079–1096

Gianola D, Foulley JL (1983) Sire evaluation for ordered categorical data with a threshold model. Genet Select Evol 15(2):201–224

McCullagh P (1980) Regression models for ordinal data. J R Stat Soc B Methodol 42(2):109–142

Montesinos-López OA, Montesinos-López A, Pérez-Rodríguez P, de los Campos G, Eskridge K, Crossa J (2015a) Threshold models for genome-enabled prediction of ordinal categorical traits in plant breeding. G3 5(2):291–300

Montesinos-López OA, Montesinos-López A, Crossa J, Burgueño J, Eskridge K (2015b) Genomic-enabled prediction of ordinal data with Bayesian logistic ordinal regression. G3 5(10):2113–2126

Montesinos-López A, Montesinos-López OA, Crossa J, Burgueño J, Eskridge KM, Falconi-Castillo E et al (2016) Genomic Bayesian prediction model for count data with genotype × environment interaction. G3 6(5):1165–1177

Pérez P, de los Campos, G. (2014) BGLR: a statistical package for whole genome regression and prediction. Genetics 198(2):483–495

Polson NG, Scott JG, Windle J (2013) Bayesian inference for logistic models using Pólya–Gamma latent variables. J Am Stat Assoc 108(504):1339–1349

Sorensen DA, Andersen S, Gianola D, Korsgaard I (1995) Bayesian inference in threshold models using Gibbs sampling. Genet Sel Evol 27(3):1–21

Zhu J, Hastie T (2004) Classification of gene microarrays by penalized logistic regression. Biostatistics 5(3):427–443

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 8
# Reproducing Kernel Hilbert Spaces Regression and Classification Methods

## 8.1 The Reproducing Kernel Hilbert Spaces (RKHS)

One of the main goals of genetic research is accurate phenotype prediction. This goal has largely been achieved for Mendelian diseases with a small number of risk variants (Schrodi et al. 2014). However, many traits (like grain yield) have a complex genetic architecture that is not well understood (Golan and Rosset 2014). Phenotype prediction for such traits remains a major challenge. A key challenge in complex phenotype prediction is accurate modeling of genetic interactions, commonly known as epistatic effects (Cordell 2002). In recent years, there has been mounting evidence that epistatic interactions are widespread throughout biology (Moore and Williams 2009; Lehner 2011; Hemani et al. 2014; Buil et al. 2015). It is well accepted that epistatic interactions are biologically plausible, on the one hand (Zuk et al. 2012), and are difficult to detect, on the other hand (Cordell 2009), suggesting that they may be highly influential in our limited success in modeling complex heritable traits.

Reproducing Kernel Hilbert Spaces (RKHS) regression was one of the earliest statistical machine learning methods suggested for use in plant and animal breeding (Gianola et al. 2006; Gianola and van Kaam 2008) for the prediction of complex traits. An RKHS is a Hilbert space of functions in which all the evaluation functionals are bounded linear functionals. The fundamental idea of RKHS methods is to project the given original input data contained in a finite-dimensional vector space onto an infinite-dimensional Hilbert space. The kernel method consists of transforming the data using a kernel function and then applying conventional statistical machine learning techniques to the transformed data, hoping for better results. Methods based on implicit transformations (RKHS methods) have become very popular for analyzing nonlinear patterns in data sets from various fields of study. Furthermore, the introduction of kernel functions has become an efficient alternative to obtain measures of similarity between objects that do not have a natural vector representation. Although the best known application of kernel methods is

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_8

Support Vector Machines (SVM), which is studied in the next chapter, lately it has been shown that any learning algorithm based on distances between objects can be formulated in terms of kernel functions, applying the so-called "kernel trick." However, RKHS methods are not limited to regression; they are also really powerful for classification and data compression problems and theoretically sound for dealing with nonlinear phenomena in general. For these reasons, they have found a wide range of practical applications ranging from bioinformatics to text categorization, from image analysis to web retrieval, from 3D reconstruction to handwriting recognition, and from geostatistics to chemoinformatics. The increase in popularity of kernel-based methods is also due in part to the fact that they provide a rich way to capture nonlinear patterns in data that cannot be captured with conventional linear statistical learning methods. In genomic selection, the application of RKHS methods continues to increase, for example, Long et al. (2010) found better performance of RKHS methods over linear models in body weight of broiler chickens. Crossa et al. (2010) compared RKHS versus Bayesian Lasso and found that RKHS was better than Bayesian Lasso in the wheat data set, but a similar performance of both methods was observed in the maize data set. Cuevas et al. (2016, 2017, 2018) found superior performance of RKHS methods over linear models using Gaussian kernels on data of maize and wheat. Cuevas et al. (2019) also found that when using pedigree, markers, and near-infrared spectroscopy (NIR) data (which is an inexpensive and nondestructive high-throughput phenotyping technology for predicting unobserved line performance in plant breeding trials), kernel methods (Gaussian kernel and arc-cosine kernel) outperformed linear models in terms of prediction performance. However, other authors found minimal differences between RKHS methods and linear models, for example, Tusell et al. (2013) in litter size in swine, Long et al. (2010) and Morota et al. (2013) in progeny tests of dairy sires, and Morota et al. (2014) in phenotypes of dairy cows. These publications have empirically shown equal or better prediction ability of RKHS methods over linear models. For this reason, the applications of kernel methods in GS are expected to continue increasing since they can be implemented in current software of genomic prediction and because they are (a) very flexible, (b) easy to interpret, (c) theoretically appealing for accommodating cryptic forms of gene action (Gianola et al. 2006; Gianola and van Kaam 2008), (d) these methods can be used with almost any type of information (e.g., covariates, strings, images, and graphs) (de los Campos et al. 2010), (e) computation is performed in an $n$-dimensional space even when the original input information has more columns ($p$) than observations ($n$) thus avoiding the $p \gg n$ problem (de los Campos et al. 2010), (f) they provide a new viewpoint whose full potential is still far from our understanding, and (g) they are very attractive due to their computational efficiency, robustness, and stability.

The goal of this chapter is to give the user (student or scientist) a friendly introduction to regression and classification methods based on kernels. We also cover the essentials of kernels methods, and with examples, we show the user how to handcraft an algorithm of a kernel for applications in the context of genomic selection.

## 8.2 Generalized Kernel Model

Like any regression problem, a generalized kernel model assumes that we have pairs $(y_i, \boldsymbol{x}_i)$ for $i = 1, \ldots, n$, where $y_i$ and $\boldsymbol{x}_i$ are the response variable and the vector of independent variables (pedigree of marker data) measured in individual $i$, and the relationship between $y_i$ and $\boldsymbol{x}_i$ is given by

$$\text{Distribution}: y_i \sim p(y_i|\mu_i)$$

$$\text{Linear predictor}: \eta_i = f(\boldsymbol{x}_i) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}$$

$$\text{Link function}: \eta_i = g(\mu_i)$$

where $g(.)$ is a known link function, $\mu_i = h(\eta_i)$, $h(.)$ denotes the inverse link function, $f(\boldsymbol{x}_i) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}, \eta_0$ is an intercept term, $\boldsymbol{k}_i = [K(\boldsymbol{x}_i, \boldsymbol{x}_1), \ldots, K(\boldsymbol{x}_i, \boldsymbol{x}_n)]^{\mathrm{T}}$, $K(.,.)$ is the kernel function, and $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_n)^{\mathrm{T}}$ is an $n \times 1$ vector of coefficients. This generalized kernel model provides a unifying framework for kernel-based analyses for dealing with continuous, binary, categorical, and count data, since with different $p(y_i|\mu_i)$ and $g(.)$, we have different models. It is very interesting to point out that under the kernel framework, the problem is reduced to finding $n$ regression coefficients instead of $p$, as in conventional regression models, thus avoiding the problem of having to solve a regression problem with $p \gg n$. Also, kernel methods are very useful when genotypes and phenotypes are connected in ways that are not well addressed by the linear additive models that are standard in quantitative genetics.

### *8.2.1 Parameter Estimation Under the Frequentist Paradigm*

Inferring $f$ requires defining a collection (or space) of functions from which an element, $\widehat{f}$, will be chosen via a criterion. Specifically, in RKHS, estimates are obtained by solving the following optimization problem:

$$\underbrace{\min}_{f \in H} \left\{ \frac{1}{n} \sum_{i=1}^{n} L(y_i, f(\boldsymbol{x}_i)) + \lambda \| f \|_H^2 \right\}, \tag{8.1}$$

which mean that the optimization problem is performed within the space of functions $H$, a RKHS, $f \in H$ and $\|f\|_H$ denotes the norm of $f$ in Hilbert space $H$; $L(y_i, f(\boldsymbol{x}_i))$ is some measure of goodness of fit, that is, a loss function viewed as the negative conditional log-likelihood, which should be chosen in agreement with the type of response variable. For example, for continuous outcomes, this should be constructed in terms of Gaussian distributions, when the response variable is binary in terms of Bernoulli distributions, when the response is count in terms of Poisson or negative

binomial distribution, and when it is categorical in terms of multinomial distributions; $\lambda$ is a smoothing or regularization parameter that should be positive and should control the trade-off between model goodness of fit and complexity; and $\| f\|_H^2$ is the square of the norm of $f(\boldsymbol{x}_i)$ on $H$, a measure of model complexity (de los Campos et al. 2010). Hilbert spaces are complete linear spaces endowed with a norm that is the square root of the inner product in the space. The Hilbert spaces that are relevant for our discussion are RKHS of real-valued functions, here denoted as $H$. Those interested in more technical details of RKHS of real functions should read Wahba (1990). By the representer theorem (Wahba 1990), which tells us that the solutions to some regularization functionals in high or infinite-dimensional spaces fall in a finite-dimensional space, the solution for (8.1) admits a linear representation

$$f(\boldsymbol{x}_i) = \eta_0 + \sum_{j=1}^{n} \beta_j K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}} \boldsymbol{\beta}, \tag{8.2}$$

where $\eta_0$ is an intercept term, $K(\cdot, \cdot)$ is the kernel function, $k_{\mathrm{i}} = [K(x_i,x_1), \ldots, K(x_i,x_{\mathrm{n}})]^{\mathrm{T}}$ as defined before, and $\beta_j$ are beta coefficients. Notice that $\| f\|_H^2 = \sum_{l,j=1}^{n} \beta_l \beta_j K\left(\boldsymbol{x}_l, \boldsymbol{x}_j\right)$, and by substituting (8.2) into (8.1), we obtain the minimization problem under a frequentist framework with respect to $\eta_0$ and $\boldsymbol{\beta}$, as did Gianola et al. (2006) and Zhang et al. (2011):

$$\underbrace{\min}_{\eta_0, \boldsymbol{\beta}} \left\{ \frac{1}{n} \sum_{i=1}^{n} L\left(y_i, \eta_0 + \boldsymbol{k}_i^{\mathrm{T}} \boldsymbol{\beta}\right) + \frac{\lambda}{2} \boldsymbol{\beta}^{\mathrm{T}} \boldsymbol{K} \boldsymbol{\beta} \right\}, \tag{8.3}$$

where $\boldsymbol{K} = [\boldsymbol{k}_1, \ldots, \boldsymbol{k}_n]$ is the $n \times n$ kernel matrix with $\boldsymbol{k}_i$ as defined above. Since $\boldsymbol{K}$ needs to be symmetric and positive semi-definite, the term $\boldsymbol{\beta}^{\mathrm{T}} \boldsymbol{K} \boldsymbol{\beta}$ is an empirical RKHS norm with regard to the training data, $\lambda$ is a smoothing or regularization parameter that should be positive and should control the trade-off between model goodness of fit and complexity, and the factor $\frac{1}{2}$ is introduced for convenience. The second term of (8.3) acts as a penalization term that is added to the minus log-likelihood. The goal is to find $\eta_0$ and $\boldsymbol{\beta}$, which is equivalent to finding $f(\boldsymbol{x}_i) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}} \boldsymbol{\beta}$ that minimizes (8.3). $f(\boldsymbol{x}_i)$ is based on a basis expansion of kernel functions and this relationship $f(\boldsymbol{x}) = \eta_0 + \boldsymbol{k}_i^{\mathrm{T}} \boldsymbol{\beta}$ is due to the representer theorem (Wahba 1990). Therefore, model specification under the generalized RKHS methods depends on the choice of loss function $L(..)$, the Hilbert space $H$ to build $\boldsymbol{K}$, and the smoothing parameter $\lambda$. The smoothing parameter $\lambda$ can be chosen by cross-validation or generalized cross-validation under the frequentist framework or by specifying a prior distribution for the $\boldsymbol{\beta}$ coefficients under the Bayesian framework (Gianola and van Kaam 2008). It is important to point out that when the response variable is coded as $y_i \in \{-1, 1\}$ and the hinge function is used as the loss function, the problem to solve is the standard support vector machine (Vapnik 1998), which is studied in the next chapter.

### 8.2.2 Kernels

A kernel function converts information on a pair of subjects into a quantitative measure representing their similarity with the requirement that the function must create a symmetric positive semi-definite (psd) matrix when applied to any subset of subjects. The psd requirement ensures a statistical foundation for using the kernel in penalized regression models. From a statistical perspective, the kernel matrix can be viewed as a covariance matrix, and we later show how this aids in the construction of kernels. Kernels are used to nonlinearly transform the input data $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \boldsymbol{X}$ into a high-dimensional feature space. Next, we provide a definition of kernel function.

Kernel function. Kernel function $K$ is a "similarity" function that corresponds to an inner product in some expanded feature space that for all $\boldsymbol{x}_i, \boldsymbol{x}_j \in X$ satisfies

$$K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \varphi(\boldsymbol{x}_i)^{\mathrm{T}} \varphi\left(\boldsymbol{x}_j\right),$$

where $\varphi$ is a mapping (transformation) from $\boldsymbol{X}$ to an (inner product) feature space $F$, $\varphi : \mathbf{x} \rightarrow \varphi(\mathbf{x})$. From this definition, we can see that the kernel has the following properties:

1. It is a symmetric function of its argument so that $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = K(\boldsymbol{x}_j, \boldsymbol{x}_i)$.
2. A necessary and sufficient condition for a function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$ to be a valid kernel (Shawe-Taylor and Cristianini 2004) is that the Gram matrix, also called kernel matrix $\mathbf{K}$, whose elements are given by $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, should be positive semi-definite for all possible choices of $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \boldsymbol{X}$.
3. Kernels are all those functions $K(\boldsymbol{u}, \boldsymbol{v})$ that verify Mercer's theorem, that is, for which

$$\int_{\boldsymbol{u}, \boldsymbol{v}} K(\boldsymbol{u}, \boldsymbol{v}) g(\boldsymbol{u}) g(\boldsymbol{v}) d\boldsymbol{u} d\boldsymbol{v} > 0$$

for all $g()$ square-integrable functions.

Mercer's theorem is an equivalent formulation of the finitely positive semi-definite property for vector spaces. The finitely positive semi-definite property suggests that kernel matrices form the core data structure for kernel methods technology. By manipulating kernel matrices, one can tune the corresponding embedding of the data in the kernel-defined feature space.

Next, we give an example of the utility of a kernel function and how this works. We assumed that we measured a sample of $n$ plants with two independent variables $(x_1, x_2)$ and one binary dependent variable $(y)$, that is, $(x_{11}, x_{21}, y_1), \ldots, (x_{1n}, x_{2n}, y_n)$. Then we plotted the observed data in Fig. 8.1 (left panel), and since the response variable is binary, we used triangles for denoting diseased plants and crosses for non-diseased plants. The goal is to build a classifier for unseen data using the data given in Fig. 8.1 as the training set. It is not possible to create a linear decision

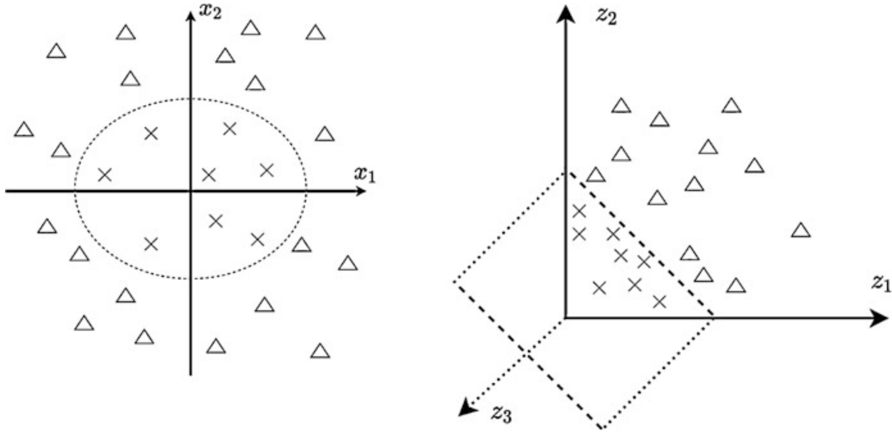



**Fig. 8.1** Mapping of the two predictor ($x_1$, $x_2$) problems with binary dependent variables ($y$; crosses = 1 and triangles = 0) where the true decision boundary is an ellipse in predictor space (left panel) to a feature map via nonlinear mapping, $\varphi(\mathbf{x})$. Input space (left panel) and feature space (right panel)

boundary to separate both types of plants (diseased vs. non-diseased) since the true decision boundary is an ellipse in predictor space (Fig. 8.1, left panel). The job of a kernel consists of estimating this boundary by first transforming (mapping) the input information (predictors) via a nonlinear mapping function into a feature map, where the problem can be reduced to estimating a hyperplane (linear boundary) between the diseased and non-diseased plants. We mapped the input information (Fig. 8.1, left panel) to the feature space using the following nonlinear map $\varphi(\mathbf{x}) = \left(z_1 = x_1^2, z_2 = x_2^2, z_3 = \sqrt{2}x_1x_2\right)$ (Fig. 8.1, right panel) and the ellipse became a hyperplane that is parallel to the $z_3$ axis, which means that all points are plotted on the ($z_1$, $z_2$) plane. Therefore, in the feature space, the problem reduces to estimating a hyperplane from the mapped data points.

For this reason, in generalized kernel models, the choice of the kernel function ($H$) is of paramount importance since it defines the space of functions over which the search for $f$ is performed, and because Hilbert spaces are normed spaces (Akhiezer and Glazman 1963). As mentioned above, by choosing $H$ one automatically defines the reproducing kernel ($\boldsymbol{K}$) which should be at least a psd matrix (de los Campos et al. 2010). There are two main properties that are required for the successful implementation of a kernel function. First, it should capture as precisely as possible the measure of similarity to the particular task and domain, and second, its construction should require significantly less computational resources than would be needed for an explicit evaluation of the corresponding feature mapping, $\varphi$.

We will call the original input information ($\boldsymbol{X}$) the input space. Then, with the kernel approach, we define a function for each pair of elements (columns) in this space $\boldsymbol{X}$ that corresponds to a real value. The transformed feature information with the kernel function is called mapped feature space.

### 8.2.3 Kernel Trick

By kernel trick we mean the use of kernel functions to operate in a high-dimensional space, *implicit* feature space, without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between the images of all pairs of data in the feature space. This operation is often computationally cheaper than the explicit computation of the coordinates. This means that the kernel trick allows you to perform algebraic operations in the transformed data space efficiently and without knowing the transformation $\varphi$. For this reason, the kernel trick is a computational trick used to compute inner products in higher dimensional spaces at a low cost. Thus, in principle, any statistical machine learning technique for data in $\boldsymbol{X} \subset \mathbb{R}^n$ that can be formulated in a computational algorithm in terms of dot products can be generalized to the transformed data using the kernel trick. Kernel functions have been introduced for sequence data, graphs, text, images, as well as vectors.

To better understand the kernel trick, we provide an example. Assume that we measure two independent variables ($x_1$, $x_2$) in four individuals. In matrix notation, the information of the independent variables (input information) is equal to

$$\boldsymbol{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \\ x_{41} & x_{42} \end{bmatrix}$$

Also, assume we will build a polynomial kernel of degree 2, with

$$\varphi(\mathbf{x}_i)^{\mathrm{T}} = \left(z_1 = x_1^2, z_2 = x_2^2, z_3 = \sqrt{2}x_1x_2\right).$$

Therefore, for building the Gram matrix (kernel matrix), we need to compute

$$\boldsymbol{K} = \begin{bmatrix} \varphi(\mathbf{x}_1)^{\mathrm{T}}\varphi(\mathbf{x}_1) & \varphi(\mathbf{x}_1)^{\mathrm{T}}\varphi(\mathbf{x}_2) & \varphi(\mathbf{x}_1)^{\mathrm{T}}\varphi(\mathbf{x}_3) & \varphi(\mathbf{x}_1)^{\mathrm{T}}\varphi(\mathbf{x}_4) \\ \varphi(\mathbf{x}_2)^{\mathrm{T}}\varphi(\mathbf{x}_1) & \varphi(\mathbf{x}_2)^{\mathrm{T}}\varphi(\mathbf{x}_2) & \varphi(\mathbf{x}_2)^{\mathrm{T}}\varphi(\mathbf{x}_3) & \varphi(\mathbf{x}_2)^{\mathrm{T}}\varphi(\mathbf{x}_4) \\ \varphi(\mathbf{x}_3)^{\mathrm{T}}\varphi(\mathbf{x}_1) & \varphi(\mathbf{x}_3)^{\mathrm{T}}\varphi(\mathbf{x}_2) & \varphi(\mathbf{x}_3)^{\mathrm{T}}\varphi(\mathbf{x}_3) & \varphi(\mathbf{x}_3)^{\mathrm{T}}\varphi(\mathbf{x}_4) \\ \varphi(\mathbf{x}_4)^{\mathrm{T}}\varphi(\mathbf{x}_1) & \varphi(\mathbf{x}_4)^{\mathrm{T}}\varphi(\mathbf{x}_2) & \varphi(\mathbf{x}_4)^{\mathrm{T}}\varphi(\mathbf{x}_3) & \varphi(\mathbf{x}_4)^{\mathrm{T}}\varphi(\mathbf{x}_4) \end{bmatrix}$$

This means that we need to compute each coordinate (cell of $\boldsymbol{K}$) with $\varphi(\mathbf{x}_i)^{\mathrm{T}}\varphi(\mathbf{x}_j)$, with $i, j = 1, 2, 3, 4$. Note that

$$\varphi(\mathbf{x}_i)^{\mathrm{T}}\varphi(\mathbf{x}_j) = \left(x_{i1}^2, x_{i2}^2, \sqrt{2}x_{i1}x_{i2}\right)\begin{bmatrix} x_{j1}^2 \\ x_{j2}^2 \\ \sqrt{2}x_{j1}x_{j2} \end{bmatrix} = x_{i1}^2x_{j1}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} + x_{i2}^2x_{j2}^2$$

$$= \left(x_{i1}x_{j1} + x_{i2}x_{j2}\right)^2.$$

Therefore,

$$\boldsymbol{K} = \begin{bmatrix} \left(x_{11}^2 + x_{12}^2\right)^2 & \left(x_{11}x_{21} + x_{12}x_{22}\right)^2 & \left(x_{11}x_{31} + x_{12}x_{32}\right)^2 & \left(x_{11}x_{41} + x_{12}x_{42}\right)^2 \\ \left(x_{21}x_{11} + x_{22}x_{12}\right)^2 & \left(x_{21}^2 + x_{22}^2\right)^2 & \left(x_{21}x_{31} + x_{22}x_{32}\right)^2 & \left(x_{21}x_{41} + x_{22}x_{42}\right)^2 \\ \left(x_{31}x_{11} + x_{32}x_{12}\right)^2 & \left(x_{31}x_{21} + x_{32}x_{22}\right)^2 & \left(x_{31}^2 + x_{32}^2\right)^2 & \left(x_{31}x_{41} + x_{32}x_{42}\right)^2 \\ \left(x_{41}x_{11} + x_{42}x_{12}\right)^2 & \left(x_{41}x_{21} + x_{42}x_{22}\right)^2 & \left(x_{41}x_{31} + x_{42}x_{32}\right)^2 & \left(x_{41}^2 + x_{42}^2\right)^2 \end{bmatrix}$$

To compute $\boldsymbol{K}$ we calculated each coordinate using $\varphi(\mathbf{x}_i)^{\mathrm{T}}\varphi(\mathbf{x}_j)$. However, note that

$$\varphi(\mathbf{x}_i)^{\mathrm{T}}\varphi(\mathbf{x}_j) = \left(x_{i1}x_{j1} + x_{i2}x_{j2}\right)^2 = \left(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + 0\right)^2,$$

where $\boldsymbol{x}_i^{\mathrm{T}} = [x_{i1}, x_{i2}]$ and $\boldsymbol{x}_j = \begin{bmatrix} x_{j1} \\ x_{j2} \end{bmatrix}$.

Hence, the function

$$K\left(\mathbf{x}_j, \mathbf{x}_j\right) = \left(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + 0\right)^2$$

corresponds to a polynomial kernel of degree $d = 2$, and constant $a = 0$, with $F$, its corresponding feature space. This means that we can compute the inner product between the projections of two points into the feature space without explicitly evaluating the coordinates. In other words, the kernel trick means that we can compute each element (coordinate) of the kernel matrix **K**, without any knowledge of the true nature of $\varphi(\mathbf{x}_i)$; we only need to know the kernel function $K(\mathbf{x}_j, \mathbf{x}_j)$. This means that the kernel function is a key ingredient for implementing kernel methods in statistical machine learning.

Next, we provide another simple example also using the polynomial kernel of degree 2, with the same two independent variables ($x_1$, $x_2$) but with a constant value $a = 1$, that is, $K(\mathbf{x}_j, \mathbf{x}_j) = \left(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + 1\right)^2$. According to the kernel trick, this means that we do not need knowledge of $\varphi(\mathbf{x}_i)$ to compute all coordinates of the matrix of kernel **K**, since each coordinate will take values of

$$K(\mathbf{x}_j, \mathbf{x}_j) = (\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + 1)^2 = (x_{i1}x_{j1} + x_{i2}x_{j2} + 1)^2$$
$$= (x_{i1}x_{j1} + x_{i2}x_{j2})^2 + 2(x_{i1}x_{j1} + x_{i2}x_{j2}) + 1$$
$$= x_{i1}^2x_{j1}^2 + 2x_{i1}x_{i2}x_{j1}x_{j2} + x_{i2}^2x_{j2}^2 + 2(x_{i1}x_{j1} + x_{i2}x_{j2}) + 1.$$

Therefore, the **K** matrix is

$$= \begin{bmatrix} (x_{11}^2+x_{12}^2+1)^2 & (x_{11}x_{21}+x_{12}x_{22}+1)^2 & (x_{11}x_{31}+x_{12}x_{32}+1)^2 & (x_{11}x_{41}+x_{12}x_{42}+1)^2 \\ (x_{21}x_{11}+x_{22}x_{12}+1)^2 & (x_{21}^2+x_{22}^2+1)^2 & (x_{21}x_{31}+x_{22}x_{32}+1)^2 & (x_{21}x_{41}+x_{22}x_{42}+1)^2 \\ (x_{31}x_{11}+x_{32}x_{12}+1)^2 & (x_{31}x_{21}+x_{32}x_{22}+1)^2 & (x_{31}^2+x_{32}^2+1)^2 & (x_{31}x_{41}+x_{32}x_{42}+1)^2 \\ (x_{41}x_{11}+x_{42}x_{12}+1)^2 & (x_{41}x_{21}+x_{42}x_{22}+1)^2 & (x_{41}x_{31}+x_{42}x_{32}+1)^2 & (x_{41}^2+x_{42}^2+1)^2 \end{bmatrix}$$

This implies that we computed each coordinate of **K** without first computing $\varphi(\mathbf{x}_i)$. This trick is really useful since for computing each coordinate of **K** in this example, we only performed dot products with vectors of size two, in the original dimension of the input information, and not dot products of vectors of dimension $\binom{2+2}{2} = 6$, which is the dimension, in this example, of $\varphi(\mathbf{x}_i)^{\mathrm{T}} = [x_{i1}^2, \sqrt{2}x_{i1}, \sqrt{2}x_{i1}x_{i2}, \sqrt{2}x_{i2}, x_{i2}^2, 1]$. Therefore, this trick facilitates the computation of **K** since it requires less computation resources. The utility of the trick is better appreciated in a large dimensional setting. For example, assume that the input information of each individual ($\mathbf{x}_i$) contains 784 independent variables; this means that to compute matrix **K** we need to compute, for each coordinate, only dot products of vectors of dimension 784 and not of dimension $\binom{784+2}{2} = 308,505$, which is the dimension of $\varphi(\mathbf{x}_i)^{\mathrm{T}}$ for the same polynomial kernel with degree 2. For this reason, kernel methods are well suited for handling a massive amount of information, because the computational burden can be proportional to the number of data points rather than to the number of predictor variables (e.g., markers in the context of genomic prediction). This is particularly true if a common weight is assigned to each marker (Morota et al. 2013).

In simple terms, the *kernel trick* makes it possible to perform a transformation from the input data space to a higher dimensional feature space, where the transformed data can be analyzed with conventional linear models and the problem becomes tractable. However, the result highly depends on the considered transformation. If the kernel function is not appropriate for the problem, or the kernel parameters are badly set, the fitted model can be of poor quality. Due to this, special care must be taken when selecting both the kernel function and the kernel parameters to obtain good results.

The kernel trick allows an efficient search in a higher dimensional space, while the related estimation problems are often cast as convex optimization problems that can be solved by many established algorithms and packages. Kernel methods can be

applied to all data analysis algorithms whose inputs can be expressed in terms of dot products. If the data in the original space cannot be analyzed satisfactorily with conventional statistical machine learning techniques, the strategy to extend it to nonlinear models using kernel methods is based on the apparently paradoxical idea of transforming the data, by means of a nonlinear function, toward a space with a greater dimension than the space where the data are located and applying any statistical machine learning algorithm to the transformed data.

Therefore, in general terms, the kernelization of an algorithm consists of its reformulation, so that the determination of a pattern or linear regularity in the data can be carried out exclusively from the information collected in the scalar products calculated for all the pairs of elements in the space. Kernel functions are characterized by the property that all finite kernel matrices are positive semi-definite.

### 8.2.4 Popular Kernel Functions

Next, we provide the most popular kernel methods in statistical machine learning.

***Linear Kernel*** This kernel is defined as $K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j$. For example,

$$K(\boldsymbol{x}, \boldsymbol{z}) = (x_1, x_2)\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = x_1 z_1 + x_2 z_2 = \varphi(\boldsymbol{x})^{\mathrm{T}} \varphi(\boldsymbol{z}).$$

Next, we provide an *R* function for calculating this kernel that can be used for both single-attribute value vectors and for the whole data set:

```
K.linear=function(x1, x2=x1) {as.matrix(x1)%*%t(as.matrix(x2)) }
```

Next, we simulate a matrix data set:

```
set.seed(3)
X=matrix(round(rnorm(16,2,0.2),2),ncol=8)
X
```

that gives as output:

```
> set.seed(3)
> X=matrix(round(rnorm(16,2,0.2),2),ncol=8)
> X
   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,] 1.81 2.05 2.04 2.02 1.76 1.85 1.86 2.03
[2,] 1.94 1.77 2.01 2.22 2.25 1.77 2.05 1.94
```

For individual features in pairs of individuals, this function is used as

```
> K.linear(X[1,1:4],X[2,1:4])
       [,1]   [,2]   [,3]   [,4]
[1,] 3.5114 3.2037 3.6381 4.0182
[2,] 3.9770 3.6285 4.1205 4.5510
[3,] 3.9576 3.6108 4.1004 4.5288
[4,] 3.9188 3.5754 4.0602 4.4844
```

while for the full set of features, it can be used as

```
> K.linear(X)
        [,1]    [,2]
[1,] 29.8212 30.7104
[2,] 30.7104 32.0265
```

This kernel does not overcome the linearity limitation of linear classification and linear regression models in any way since it leaves the original representation unchanged. It is important to point out that linear kernels (such as linear regression, linear support vector machines, and linear support vector regression algorithms) are special cases of more sophisticated kernel-based algorithms.

***Polynomial Kernel*** As mentioned above, this kernel is defined as $K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \left(\gamma \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j + a\right)^d$, where $a$ is a real scalar and $d$ is a positive integer, and where $\gamma > 0$, $a \geq 0$, and $d > 0$ are parameters. This kernel family makes it possible to easily control the enhanced representation size and degree of nonlinearity by adjusting the $d$ parameter. Positive $a$ can be used to adjust the relative impact of higher order and lower order terms in the resulting polynomial representation. For example, when $\gamma = 1$, $a = 0$, and $d = 2$, we have

$$K(\boldsymbol{x}, \boldsymbol{z}) = \left((x_1, x_2)\begin{pmatrix} z_1 \\ z_2 \end{pmatrix}\right)^2 = (x_1 z_1 + x_2 z_2)^2 =$$

$$x_1^2 z_1^2 + 2x_1 z_1 x_2 z_2 + x_2^2 z_2^2 = \left[x_1^2, \sqrt{2}x_1 x_2, x_2^2\right]\begin{bmatrix} z_1^2 \\ \sqrt{2} z_1 z_2 \\ z_2^2 \end{bmatrix} = \varphi(\boldsymbol{x})^{\mathrm{T}} \varphi(\boldsymbol{z}).$$

However, when $\gamma = 1$, $a = 1$, and $d = 2$, we have

$$K(\boldsymbol{x}, \boldsymbol{z}) = \left((x_1, x_2)\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} + 1\right)^2 = (x_1 z_1 + x_2 z_2 + 1)^2 =$$

$$1 + 2x_1z_1 + 2x_2z_2 + x_1^2z_1^2 + x_2^2z_2^2 + 2x_1z_1x_2z_2$$

$$= \left[1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2\right] \begin{bmatrix} 1 \\ \sqrt{2}z_1 \\ \sqrt{2}z_2 \\ z_1^2 \\ \sqrt{2}z_1z_2 \\ z_2^2 \end{bmatrix} = \varphi(\boldsymbol{x})^{\mathrm{T}}\varphi(\boldsymbol{z}).$$

This demonstrates that increasing $a$ increases the coefficients of lower order terms. The dimension of the feature space for the polynomial kernel is equal to $\binom{p+d}{d}$. For example, for an input vector of dimension $p = 10$ and polynomial with degree $d = 3$, the dimension for this polynomial kernel is equal to $\binom{10+3}{3}$ =286, while if $p = 1000$ and $d = 3$, the dimension for this polynomial kernel is equal to $\binom{1000+3}{3}$ =167,668,501. Although convenient to control and easy to understand, the polynomial kernel family may be insufficient to adequately represent more complex relationships.

The R code for calculating this kernel is given next:

```
K.polynomial=function(x1, x2=x1, gamma=1, b=0, d=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^d}
```

Now this function can be used as

```
> K.polynomial(X[1,1:4],X[2,1:4])
      [,1]     [,2]     [,3]     [,4]
[1,] 43.29532 32.88180 48.15306 64.87758
[2,] 62.90234 47.77288 69.95999 94.25850
[3,] 61.98630 47.07716 68.94117 92.88582
[4,] 60.18099 45.70607 66.93331 90.18058
```

But for the full set of features, it can be used as

```
> K.polynomial(X)
      [,1]     [,2]
[1,] 26520.11 28963.86
[2,] 28963.86 32849.48
```

***Sigmoidal Kernel*** This kernel is defined as $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \tan h(\gamma \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j + b)$, where tan$h$ is the hyperbolic tangent defined as $\tan h(z) = \sin h(z) / \cos h(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$. This function is widely used as the activation function for artificial neural networks and deep learning models, and hence has also become popular for kernel methods. If used with properly adjusted parameters, it can represent complex nonlinear relationships. In some parameter settings, it actually becomes similar to the radial kernel (Lin and Lin 2003) described below. However, the sigmoid function may not be positive definite for some parameters, and therefore may not actually represent a valid kernel (Lin and Lin 2003).

Next, we provide an R code for calculating this kernel:

```
K.sigmoid=function(x1,x2=x1, gamma=0.1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
```

This function is used as

```
> K.sigmoid(X[1,1:4],X[2,1:4])
          [,1]      [,2]      [,3]      [,4]
[1,] 0.3373862 0.3098414 0.3485656 0.3815051
[2,] 0.3779793 0.3477219 0.3902119 0.4260822
[3,] 0.3763152 0.3461650 0.3885066 0.4242635
[4,] 0.3729798 0.3430454 0.3850881 0.4206158
```

For the full set of features, it is used as

```
> K.sigmoid(X)
          [,1]      [,2]
[1,] 0.9948752 0.9957083
[2,] 0.9957083 0.9966999
```

***Gaussian Kernel*** This kernel, also known as the radial basis function kernel, depends on the Euclidean distance between the original attribute value vectors (i.e., the Euclidean norm of their difference) rather than on their dot product, $K(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{-\gamma \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2} = e^{-\gamma[\boldsymbol{x}_i^T \boldsymbol{x}_i - 2\boldsymbol{x}_i^T \boldsymbol{x}_i + \boldsymbol{x}_j^T \boldsymbol{x}_j]}$, where $\gamma$ is a positive real scalar. It is known that the feature vector $\varphi$ that corresponds to the Gaussian kernel is actually infinitely dimensional (Lin and Lin 2003). Therefore, without the kernel trick, the solution cannot be computed explicitly. This type of kernel tends to be particularly popular, but it is sensitive to the choice of the γ parameter and may be prone to overfitting.

The R code for calculating this kernel is given next:

```
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]) ^2)))}
```

This function is used as

```
> K.radial(X[1,1:4],X[2,1:4])
          [,1]      [,2]      [,3]      [,4]
[1,] 0.9832420 0.9984013 0.9607894 0.8452693
[2,] 0.9879729 0.9245945 0.9984013 0.9715136
[3,] 0.9900498 0.9296938 0.9991004 0.9681193
[4,] 0.9936204 0.9394131 0.9999000 0.9607894
```

while for the full set of features, it can be used as

```
> K.radial(X)
          [,1]      [,2]
[1,] 1.0000000 0.6525288
[2,] 0.6525288 1.0000000
```

The parameter $\gamma$ controls the flexibility of the Gaussian kernel in a similar way as the degree $d$ in the polynomial kernel. Large values of $\gamma$ correspond to large values of $d$ since, for example, they allow classifiers to fit any labels, hence risking overfitting. In such cases, the kernel matrix becomes close to the identity matrix. On the other hand, small values of $\gamma$ gradually reduce the kernel to a constant function, making it impossible to learn any nontrivial classifier. The feature space has infinite dimensions for every value of $\gamma$, but for large values, the weight decays very fast on the higher order features. In other words, although the rank of the kernel matrix is full, for all practical purposes, the points lie in a low-dimensional subspace of the feature space.

***Exponential Kernel*** This kernel is defined as $K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = e^{-\gamma\left\|\boldsymbol{x}_i - \boldsymbol{x}_j\right\|}$, which is quite similar to the Gaussian kernel function.

The R code is given below:

```
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}
```

For individual features in pairs of individuals, it can be used as

```
> K.exponential(X[1,1:4],X[2,1:4])
          [,1]      [,2]      [,3]      [,4]
[1,] 0.8780954 0.9607894 0.8187308 0.6636503
[2,] 0.8958341 0.7557837 0.9607894 0.8436648
[3,] 0.9048374 0.7633795 0.9704455 0.8352702
[4,] 0.9231163 0.7788008 0.9900498 0.8187308
```

while for full set of features, it can be used as

```
> K.exponential(X)
          [,1]      [,2]
[1,] 1.0000000 0.5202864
[2,] 0.5202864 1.0000000
```

***Arc-Cosine Kernel (AK)*** For AK, an important component is the angle between two vectors computed from inputs $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ as

$$\theta_{i,j} = \cos^{-1}\left(\frac{\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j}{\|\boldsymbol{x}_i\|\|\boldsymbol{x}_j\|}\right),$$

where $\|\boldsymbol{x}_i\|$ is the norm of observation $i$. The following kernel is positive semi-definite and related to an ANN with a single hidden layer and the ramp activation function (Cho and Saul 2009).

$$\mathrm{AK}^1\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = \frac{1}{\pi}\|\boldsymbol{x}_i\|\|\boldsymbol{x}_j\| \, J\left(\theta_{i,j}\right), \tag{8.4}$$

where $\pi$ is the pi constant and $J(\theta_{i,j}) = [\sin(\theta_{i,j}) + (\pi - \theta_{i,j}) \cos(\theta_{i,j})]$. Equation (8.4) gives a symmetric positive semi-definite matrix ($\mathrm{AK}^1$) preserving the norm of the entries such that $\mathrm{AK}(\boldsymbol{x}_i, \boldsymbol{x}_i) = \|\boldsymbol{x}_i\|^2$ and $\mathrm{AK}(\boldsymbol{x}_i, -\boldsymbol{x}_i) = 0$ and models nonlinear relationships.

Note that the diagonals of the AK matrix are not homogeneous and express heterogeneous variances of the genetic value $\boldsymbol{u}$; this is different from the Gaussian kernel matrix, with a diagonal that expresses homogeneous variances. This property could be a theoretical advantage of AK when modeling interrelationships between individuals.

In order to emulate the performance of an ANN with more than one hidden layer ($l$), Cho and Saul (2009) proposed a recursive relationship of repeating $l$ times the interior product:

$$\mathrm{AK}^{(l+1)}\left(\boldsymbol{x}_i,\boldsymbol{x}_j\right)=\frac{1}{\pi}\left[\mathrm{AK}^{(l)}(\boldsymbol{x}_i,\boldsymbol{x}_i)\mathrm{AK}^{(l)}\left(\boldsymbol{x}_j,\boldsymbol{x}_j\right)\right]^{\frac{1}{2}}J\left(\theta_{i,j}^{(l)}\right), \tag{8.5}$$

where $\theta_{i,j}^{(l)}=\cos^{-1}\left\{\mathrm{AK}^{(l)}\left(\boldsymbol{x}_i,\boldsymbol{x}_j\right)\left[\mathrm{AK}^{(l)}(\boldsymbol{x}_i,\boldsymbol{x}_i)\mathrm{AK}^{(l)}\left(\boldsymbol{x}_j,\boldsymbol{x}_j\right)\right]^{-\frac{1}{2}}\right\}$.

Thus, computing $\mathrm{AK}^{(l+1)}$ at level (layer) $l + 1$ is done from the previous layer $\mathrm{AK}^{(l)}$. Computing a bandwidth (the smoothing parameter that controls variance and bias in the output, e.g., the $\gamma$ parameter in the Gaussian kernel) is not necessary, and the only computational effort required is to compute the number of hidden layers. Cuevas et al. (2019) described a maximum marginal likelihood method used to select the number of hidden layers ($l$) for the AK kernel. It is important to point out that this kernel method is like a deep neural network since it allows using more than one hidden layer.

The R code for the AK kernel with one hidden layer is given below:

```
K.AK1_Final<-function(x1,x2){
 n1<-nrow(x1)
 n2<-nrow(x2)
 x1tx2<-x1%*%t(x2)
 norm1<-sqrt(apply(x1,1,function(x) crossprod(x)))
 norm2<-sqrt(apply(x2,1,function(x) crossprod(x)))
 costheta = diag(1/norm1)%*%x1tx2%*%diag(1/norm2)
 costheta[which(abs(costheta)>1,arr.ind = TRUE)] = 1
 theta<-acos(costheta)
 normx1x2<-norm1%*%t(norm2)
 J = (sin(theta)+(pi-theta)*cos(theta))
 AK1 = 1/pi*normx1x2*J
 AK1<-AK1/median(AK1)
 colnames(AK1)<-rownames(x2)
 rownames(AK1)<-rownames(x1)
 return(AK1)
}
```

For the full set of features, it can be used as

```
> K.AK1_Final(x1=X,x2=X)
      [,1]    [,2]
[1,] 0.9709  1.000000
[2,] 1.0000  1.042699
```

Since the K.AK1_Final() kernel function is only useful for one hidden layer, for this reason, the next part of the code extends this to more than one hidden layer.

```
####Kernel Arc-Cosine with deep=4#####
diagAK_f<-function(dAK1)
{
 AKAK = dAK1^2
 costheta = dAK1*AKAK^(-1/2)
 costheta[which(costheta>1,arr.ind = TRUE)] = 1
```

```
 theta = acos(costheta)
 AKl  = (1/pi)*(AKAK^(1/2))*(sin(theta)+(pi-theta)*cos(theta))
 AKl
 AKl<-AKl/median(AKl)
}
AK_L_Final<-function(AK1,dAK1,nl){
 n1<-nrow(AK1)
 n2<-ncol(AK1)
 AKl1 = AK1
 for ( l in 1:nl){

  AKAK<-tcrossprod(dAK1,diag(AKl1))

  costheta<-AKl1*(AKAK^(-1/2))
  costheta[which(costheta>1,arr.ind = TRUE)] = 1
  theta <-acos(costheta)

  AKl<-(1/pi)*(AKAK^(1/2))*(sin(theta)+(pi-theta)*cos(theta))
  dAKl = diagAK_f(dAK1)
  AKl1 = AKl
  dAK1 = dAKl
 }
 AKl<-AKl/median(AKl)
 rownames(AKl)<-rownames(AK1)
 colnames(AKl)<-colnames(AK1)
 return(AKl)
}
```

Next, we illustrate how to use this kernel function for an AR kernel with four hidden layers:

```
> AK1=K.AK1_Final(x1=X,x2=X)
> AK_L_Final(AK1= AK1,dAK1=diag(AK1),nl=4)
      [,1]      [,2]
[1,] 0.9649746 1.000000
[2,] 1.0000000 1.036335
```

***Hybrid Kernel*** We understand by hybrid kernels when two or more kernels are combined, since complex kernels can be created by simple operations (multiplication, addition, etc.) that combine simpler kernels. An example of a hybrid kernel can be obtained by multiplying the polynomial kernel and the Gaussian kernel. This kernel is defined as $\left(\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{x}_j + a\right)^d e^{-\gamma\left\|\boldsymbol{x}_i-\boldsymbol{x}_j\right\|}$. However, other types of kernels can also be combined in the same fashion or with other basic operations, like kernel averaging, which is explained next.

***Kernel Averaging*** Averaging is another way to create hybrid kernels, since kernel methods do not preclude the use of several kernels together (de los Campos et al. 2010). To illustrate the construction of these kernels, we assume that we have three

kernels $\boldsymbol{K}_1$, $\boldsymbol{K}_2$, and $\boldsymbol{K}_3$ that are distinct from each other. In this approach, the three kernels are "averaged" to form a new kernel $\boldsymbol{K} = \boldsymbol{K}_1 \frac{\sigma^2_{K_1}}{\sigma^2_K} + \boldsymbol{K}_2 \frac{\sigma^2_{K_2}}{\sigma^2_K} + \boldsymbol{K}_3 \frac{\sigma^2_{K_3}}{\sigma^2_K}$, where $\sigma^2_{K_1}, \sigma^2_{K_2}, \sigma^2_{K_3}$ are variance components attached to kernels $\boldsymbol{K}_1$, $\boldsymbol{K}_2$, and $\boldsymbol{K}_3$, respectively, and $\sigma^2_K$ is the sum of the three variances. The ratios of the three variance components are tantamount to the relative contributions of the kernels. For instance, the kernels used can be three Gaussian kernels with different bandwidth parameter values, as employed in Tusell et al. (2013), or one can fit several parametric kernels jointly, e.g., the additive (**G**), dominance (**D**), and additive by dominance (**G#D**) kernels, as in Morota et al. (2014). While there are many possible choices of kernels, the kernel function can be estimated via maximum likelihood by recourse to the Matérn family of covariance functions (e.g., Ober et al. 2011) or by fitting several candidate kernels simultaneously through multiple kernel learning.

Hybrid kernels illustrate a general principle of how more complex kernels can be created from simpler ones in a number of different ways. Kernels can even be constructed that correspond to infinite-dimensional feature spaces at the cost of only a few extra operations in the kernel evaluations, like the Gaussian kernel which most often is good enough (Ober et al. 2011). There are many other kernels, however, the above-mentioned kernels are the most popular. For example, Morota et al. (2013) evaluated diffusion kernels for discrete inputs with animal and plant data, and compared these to the Gaussian kernel. Differences in predictive ability were minimal; this is fortunate because computing diffusion kernels is time-consuming.

The bandwidth parameter can be selected based on (1) a cross-validation procedure, (2) restricted maximum likelihood (Endelman 2011), and (3) an empirical Bayesian method such as the one proposed by Pérez-Elizalde et al. (2015). The optimal value of the bandwidth parameter is expected to change with many factors such as (a) distance function, (b) number of markers, allelic frequency, and coding of markers, all markers affecting the distribution of observed distances, and (c) genetic architecture of the trait, a factor affecting the expected prior correlation of genetic values (de los Campos et al. 2010).

As pointed out above, kernel methods only need information of the kernel function $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, assuming that this has been defined. For this reason, nonvectorial patterns $\mathbf{x}$ such as sequences, trees, and graphs can be handled. That is, kernel functions are not restricted to vectorial inputs: kernels can be designed for objects and structures as diverse as strings, graphs, text documents, sets, and graph nodes. It is important to point out that the kernel trick can be applied in unsupervised methods like cluster analysis, and dimensionality reduction methods like principal component analysis, independent component analysis, etc.

### 8.2.5 A Two Separate Step Process for Building Kernel Machines

The goal of this section is to emphasize that the building process of kernel machines consists of two general independent steps. The first one consists of calculating the Gram matrix (kernel matrix $\boldsymbol{K}$) using only the information of the independent variables (input). This means that in this process the user needs to define the type of kernel function that he (she) will use in such a way as to capture the hypothesized nonlinear patterns in the input data. Then in the second step, after the kernel is ready, we select the statistical machine learning algorithm that will be used for training the model using the dependent variable, the kernel built in the first step and other available covariates. These two separate steps for building kernel methods for prediction imply that we can use conventional linear statistical machine learning algorithms to accommodate a particular type of kernel function. The only important consideration when choosing the kernel is that it should be suitable for the data at hand. But, if you built the kernel, you can evaluate the performance of this kernel with many other statistical machine learning methods. This illustrates the two separate steps required for training predictive machines using kernel methods where any statistical machine learning method can be combined with any kernel function. It is important to point out that since many machine learning methods are only able to work with linear patterns, using the kernel trick allows you to build nonlinear versions of the linear algorithms, without the need to modify the original machine learning algorithm. The following sections show how the kernel trick works in some standard statistical machine learning models.

## 8.3 Kernel Methods for Gaussian Response Variables

When the response variable is Gaussian, the negative log-likelihood that needs to be used to minimize expression (8.3) belongs to a normal distribution and the expression (8.3) is reduced to

$$\underbrace{\min}_{\eta_0,\boldsymbol{\beta}}\left\{\frac{1}{n}\sum_{i=1}^{n}\left(y_i-\eta_0-\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}\right)^2+\frac{\lambda}{2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\beta}\right\}.$$

In matrix notation, the latter expression can be expressed as

$$\underbrace{\min}_{\eta_0,\boldsymbol{\beta}}\left\{\frac{1}{2}(\boldsymbol{y}^*-\boldsymbol{K}\boldsymbol{\beta})^{\mathrm{T}}(\boldsymbol{y}^*-\boldsymbol{K}\boldsymbol{\beta})+\frac{\lambda}{2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\beta}\right\},$$

where $\boldsymbol{y}^*=\boldsymbol{y}-\mathbf{1}\overline{y}$, using $\overline{y}$ as an estimator of the intercept ($\eta_0$). The first-order conditions to this problem are familiar to us (see Chap. 3) and are

$$\left[\boldsymbol{K}^{\mathrm{T}}\boldsymbol{K}+\lambda\boldsymbol{K}\right]\boldsymbol{\beta}=\boldsymbol{K}^{\mathrm{T}}\boldsymbol{y}^{*}$$

Further, since $\boldsymbol{K}=\boldsymbol{K}^{\mathrm{T}}$ and $\boldsymbol{K}^{-1}$ exist, pre-multiplication by $\boldsymbol{K}^{-1}$ yields

$$\boldsymbol{\beta}=[\boldsymbol{K}+\lambda\boldsymbol{I}]^{-\mathbf{1}}\boldsymbol{y}^{*},$$

where $\boldsymbol{I}$ is an identity matrix of dimension $n \times n$, and to estimate $\boldsymbol{\beta}$, $\lambda$ must be known. It is important to point out that even in the context of large $p$ and small $n$, the number of beta coefficients ($\boldsymbol{\beta}$) that need to be estimated is equal to $n$, which considerably reduces the computation resources in the estimation process. This solution to the beta coefficients obtained under Gaussian response variables is known as kernel Ridge regression in statistical machine learning, and was first obtained by Gianola et al. (2006) and Gianola and van Kaam (2008) in the context of a mixed effects model under a Bayesian treatment. The predicted values in the original scale of the response variables can be obtained as

$$\widehat{\boldsymbol{y}}=\mathbf{1}\overline{y}+\boldsymbol{K}\widehat{\boldsymbol{\beta}}.$$

For a new observation with vector of inputs ($\boldsymbol{x}_{\mathrm{new}}$), the predictions are made using the following expression:

$$\widehat{\boldsymbol{y}}_{\mathrm{new}}=\overline{y}+\sum_{i=1}^{n}\widehat{\beta}_{i}K(\boldsymbol{x}_{i},\boldsymbol{x}_{\mathrm{new}})$$

Next, we provide some examples of Gaussian response variables using different kernel methods.

**Example 1 for continuous response variables**. The data comprise family, marker, and phenotypic information of 599 lines that were evaluated for grain yield (GY) in four environments. Marker information consisted of 1447 Diversity Array Technology (DArT) markers, generated by Triticarte Pty. Ltd. (Canberra, Australia). Also, this data set contains the pedigree relationship matrix and is preloaded in the BGLR package with the name wheat. We named this data set the wheat599 data set. The GY measured in the four environments was used for single environment analysis using various kernel methods.

The first six observations for trait GY in the four environments (labeled 1, 2, 4, and 5) are given next.

```
> head(y)
        1          2           4          5
775   1.6716295 -1.72746986 -1.89028479  0.0509159
2166 -0.2527028  0.40952243  0.30938553 -1.7387588
2167  0.3418151 -0.64862633 -0.79955921 -1.0535691
2465  0.7854395  0.09394919  0.57046773  0.5517574
```

```
3881 0.9983176 -0.28248062 1.61868192 -0.1142848
3889 2.3360969 0.62647587 0.07353311 0.7195856
```

Also, next are given the first six observations for five standardized markers

```
> head(XF[,1:5])
     wPt.0538   wPt.8463  wPt.6348 wPt.9992  wPt.2838
[1,] -1.3598855 0.2672768 0.772228 0.4419075 0.439209
[2,]  0.7341284 0.2672768 0.772228 0.4419075 0.439209
[3,]  0.7341284 0.2672768 0.772228 0.4419075 0.439209
[4,] -1.3598855 0.2672768 0.772228 0.4419075 0.439209
[5,] -1.3598855 0.2672768 0.772228 0.4419075 0.439209
[6,]  0.7341284 0.2672768 0.772228 0.4419075 0.439209
```

Then with the code given in Appendix 1, that uses the wheat599 data set, the nine kernels explained above were illustrated. We implemented the kernel Ridge regression method using the library glmnet. The results of the nine kernels for GY in each of the four environments are given next.

Table 8.1 indicates that the best predictions were observed in the four environments under the Sigmoid kernel and the worst under the polynomial kernel.

## 8.4 Kernel Methods for Binary Response Variables

When the response variable is binary, instead of using the sum of squares loss function that was used before for continuous response variables, we now use the negative log-likelihood of the product of Bernoulli distributions, and the expression that needs to be minimized is given next:

$$\underbrace{\min}_{\eta_0,\boldsymbol{\beta}}\left\{\frac{1}{n}\sum_{i=1}^{n}\left(y_i\left(\eta_0+\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}\right)+\log\left[1+\exp\left(\eta_0+\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}\right)\right]\right)^2+\frac{\lambda}{2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\beta}\right\}$$

Estimation of the parameters $\eta_0$ and $\boldsymbol{\beta}$ requires an iterative procedure, and gradient descent methods are used for their estimation, like those explained for logistic regression in Chaps. 3 and 7. Here, for the examples, we will use the glmnet package.

In Table 8.2, we can observe the prediction performance using the binary trait Height of the Data_Toy_EYT.R with nine kernels (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4). The Data_Toy_EYT data set contains 160 observations with 40 in each of the four environments that are present. The phenotypic information consists of a column for lines, another for environments and four corresponding to traits, two measured on a categorical scale, one continuous, and the last one binary. The data set also contains a genomic relationship matrix of the 40 lines that were evaluated in each of the four environments. Ten fold cross-

**Table 8.1** Prediction performance in terms of the mean square error for GY in the four environments (Env) under nine kernel methods

| Env | Linear | Polynomial | Sigmoid | Gaussian | Exponential | AK1 | AK2 | AK3 | AK4 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1.009 | 1.001 | **0.762** | 0.893 | 0.892 | 0.882 | 0.881 | 0.886 | 0.891 |
| 2 | 0.916 | 1.069 | **0.772** | 0.922 | 0.905 | 0.841 | 0.899 | 0.901 | 0.909 |
| 4 | 0.984 | 1.004 | **0.859** | 0.940 | 0.950 | 0.941 | 0.936 | 0.943 | 0.945 |
| 5 | 1.019 | 1.002 | **0.814** | 0.901 | 0.949 | 0.868 | 0.876 | 0.885 | 0.894 |

**Table 8.2** Prediction performance in terms of the proportion of cases correctly classified (PCCC) with ten fold cross-validation for the binary trait Height of the Data_Toy_EYT.R data set with nine kernels

| Fold | Linear | Polynomial | Sigmoid | Gaussian | Exponential | AK1 | AK2 | AK3 | AK4 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.813 | 0.813 | 0.688 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 |
| 2 | 0.688 | 0.688 | 0.563 | 0.688 | 0.688 | 0.688 | 0.688 | 0.688 | 0.688 |
| 3 | 0.688 | 0.750 | 0.563 | 0.625 | 0.625 | 0.625 | 0.625 | 0.625 | 0.750 |
| 4 | 0.625 | 0.750 | 0.375 | 0.750 | 0.750 | 0.625 | 0.750 | 0.625 | 0.750 |
| 5 | 0.750 | 0.750 | 0.625 | 0.750 | 0.750 | 0.750 | 0.688 | 0.750 | 0.688 |
| 6 | 0.625 | 0.688 | 0.625 | 0.625 | 0.625 | 0.625 | 0.625 | 0.625 | 0.688 |
| 7 | 0.750 | 0.750 | 0.500 | 0.750 | 0.750 | 0.750 | 0.750 | 0.750 | 0.750 |
| 8 | 0.813 | 0.813 | 0.563 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 |
| 9 | 0.688 | 0.688 | 0.625 | 0.688 | 0.688 | 0.688 | 0.688 | 0.688 | 0.688 |
| 10 | 0.813 | 0.875 | 0.688 | 0.813 | 0.813 | 0.813 | 0.813 | 0.750 | 0.813 |
| Average | 0.725 | 0.756 | 0.581 | 0.731 | 0.731 | 0.719 | 0.725 | 0.713 | 0.744 |

validation was implemented and the worst performance in terms of the proportion of cases correctly classified (PCCC) was with the sigmoid kernel and the best under the polynomial and AK4 kernels. The R code for reproducing the results in Table 8.2 is given in Appendix 2.

## 8.5 Kernel Methods for Categorical Response Variables

For categorical response variables, the loss function is the negative log-likelihood of the product of multinomial distributions and the expression that needs to be minimized is given next:

$$\underbrace{\min}_{\eta_0,\boldsymbol{\beta}}\left\{-\frac{1}{n}\left(\sum_{i=1}^{n}\sum_{c=1}^{C}I_{\{y_i=c\}}\left(\eta_{0c}+\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}_c\right)-\sum_{i=1}^{n}\log\left[\sum_{l=1}^{C}\exp\left(\eta_{0l}+\boldsymbol{k}_i^{\mathrm{T}}\boldsymbol{\beta}_l\right)\right]\right)+\frac{\lambda}{2}\sum_{l=1}^{C}\boldsymbol{\beta}_l^{\mathrm{T}}\boldsymbol{K}\boldsymbol{\beta}_l\right\}$$

The estimation process does not have an analytical solution, and gradient descent methods are used for the estimation of the required parameters. The optimization process is done with the same methods described in Chaps. 3 and 7 for categorical response variables. The following illustrative examples were implemented using the glmnet library.

Now the Data_Toy_EYT.R data set was used that was also used for illustrating kernels with binary response variables. The nine kernels were implemented but with the categorical response variable days to heading (DTHD). Again, the worst predictions occurred with the sigmoid kernel, but now the best predictions were achieved with the AK2 kernel (Table 8.3). The code given in Appendix 2 can be used for reproducing these results with two small modifications: (a) replace the response variable y2=Pheno$Height with y2=Pheno$DTHD and (b) in the specification of the model in glmnet, replace family='binomial' with family='multinomial'.

## 8.6 The Linear Mixed Model with Kernels

Under a linear mixed model (LMM) ($\boldsymbol{y} = \boldsymbol{C\theta} + \boldsymbol{K\beta} + \boldsymbol{e}$), every individual $i$ is associated with a genotype vector $\boldsymbol{x}_i^{\mathrm{T}}$ and a covariate vector $\boldsymbol{c}_i^{\mathrm{T}}$ (e.g., gender, age, herd, race, environment, etc.). Given a sample of individuals with a genotyped variants matrix $\boldsymbol{X} = \left[\boldsymbol{x}_1^{\mathrm{T}}, \boldsymbol{x}_2^{\mathrm{T}}, \ldots, \boldsymbol{x}_n^{\mathrm{T}}\right]^{\mathrm{T}}$ and matrix of incidence nuisance variables $\boldsymbol{C} = \left[\boldsymbol{c}_1^{\mathrm{T}}, \boldsymbol{c}_2^{\mathrm{T}}, \ldots, \boldsymbol{c}_n^{\mathrm{T}}\right]^{\mathrm{T}}$, relating some effect ($\boldsymbol{\theta}$) to the phenotype vector $\boldsymbol{y} = [y_1, y_2, \ldots, y_n]^T$ that follows a multivariate normal distribution.

$$\boldsymbol{y}|\boldsymbol{X},\boldsymbol{C} \sim N\left(\boldsymbol{C\theta}, \boldsymbol{K} + \boldsymbol{I}\sigma_e^2\right)$$

**Table 8.3** Prediction performance in terms of the proportion of cases correctly classified (PCCC) with ten fold cross-validation for the categorical trait days to heading (DTHD) of the Data_Toy_EYT.R data set with nine kernels

| Fold | Linear | Polynomial | Sigmoid | Gaussian | Exponential | AK1 | AK2 | AK3 | AK4 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.813 | 0.813 | 0.688 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 |
| 2 | 0.813 | 0.813 | 0.625 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 |
| 3 | 0.688 | 0.688 | 0.750 | 0.688 | 0.688 | 0.688 | 0.688 | 0.688 | 0.688 |
| 4 | 0.875 | 0.813 | 0.750 | 0.875 | 0.875 | 0.875 | 0.875 | 0.875 | 0.875 |
| 5 | 0.688 | 0.625 | 0.625 | 0.688 | 0.625 | 0.625 | 0.688 | 0.688 | 0.625 |
| 6 | 0.750 | 0.750 | 0.563 | 0.750 | 0.750 | 0.750 | 0.750 | 0.750 | 0.750 |
| 7 | 0.688 | 0.750 | 0.563 | 0.688 | 0.688 | 0.688 | 0.688 | 0.688 | 0.688 |
| 8 | 0.875 | 0.875 | 0.875 | 0.875 | 0.875 | 0.875 | 0.875 | 0.875 | 0.875 |
| 9 | 0.813 | 0.813 | 0.688 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 |
| 10 | 0.625 | 0.688 | 0.500 | 0.625 | 0.688 | 0.625 | 0.688 | 0.625 | 0.625 |
| Average | 0.763 | 0.763 | 0.663 | 0.763 | 0.763 | 0.756 | 0.769 | 0.763 | 0.756 |

Here, $\boldsymbol{K}$ is a valid kernel encoding genotypic covariance, as long as it is positive semi-definite and, again, represents similarities between genotyped individuals. Now the nonparametric function is $\boldsymbol{f}(\boldsymbol{X}) = \boldsymbol{K\beta}$ and the nonparametric coefficients, $\boldsymbol{\beta}$, and residuals can be assumed to be independently distributed as $\boldsymbol{\beta} \sim \boldsymbol{N}\left(\mathbf{0}, \boldsymbol{K}^{-1}\sigma_\beta^2\right)$ and $\boldsymbol{e} \sim \boldsymbol{N}\left(\mathbf{0}, \boldsymbol{I}\sigma_e^2\right)$. $\boldsymbol{\theta}$ is a vector of covariate coefficients (denoted as fixed effects), $\boldsymbol{I}$ is the $n \times n$ identity matrix, and $\sigma_e^2$ is the variance of the microenvironmental effects. Now under this LMM approach, the function to be minimized becomes

$$\underbrace{\min}_{\boldsymbol{\theta},\boldsymbol{\beta}} \quad J[\boldsymbol{\theta},\boldsymbol{\beta}|\lambda] = \underbrace{\min}_{\boldsymbol{\theta},\boldsymbol{\beta}} \quad \left\{\frac{1}{2\sigma_e^2}[\boldsymbol{y} - \boldsymbol{C\theta} - \boldsymbol{K\beta}]^{\mathrm{T}}[\boldsymbol{y} - \boldsymbol{C\theta} - \boldsymbol{K\beta}] + \frac{\lambda}{2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{K\beta}\right\}.$$

After setting the gradient of $J(.)$ with respect to $\boldsymbol{\theta}$ and $\boldsymbol{\beta}$ simultaneously to zero (Mallick et al. 2005; Gianola et al. 2006; Gianola and van Kaam 2008), the RKHS regression estimating equations can be formulated in matrix form given $\sigma_e^2$ and $\lambda$ as

$$\begin{bmatrix} \boldsymbol{C}^{\mathrm{T}}\boldsymbol{C} & \boldsymbol{C}^{\mathrm{T}}\boldsymbol{K} \\ \boldsymbol{K}^{\mathrm{T}}\boldsymbol{C} & \boldsymbol{K}^{\mathrm{T}}\boldsymbol{K} + \lambda\boldsymbol{K}\sigma_e^2 \end{bmatrix} \begin{bmatrix} \widehat{\boldsymbol{\theta}} \\ \widehat{\boldsymbol{\beta}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C}^{\mathrm{T}}\boldsymbol{y} \\ \boldsymbol{K}^{\mathrm{T}}\boldsymbol{y} \end{bmatrix} \tag{8.6}$$

Recall that $\boldsymbol{K}$ is symmetric, so $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{K} = \boldsymbol{K}^2$, and by multiplying the second system of (8.6) by $\boldsymbol{K}^{-1}$ (assuming the inverse exists), we obtain

$$\begin{bmatrix} \boldsymbol{C}^{\mathrm{T}}\boldsymbol{C} & \boldsymbol{C}^{\mathrm{T}}\boldsymbol{K} \\ \boldsymbol{I}^{\mathrm{T}}\boldsymbol{C} & \boldsymbol{K} + \lambda\boldsymbol{I}\sigma_e^2 \end{bmatrix} \begin{bmatrix} \widehat{\boldsymbol{\theta}} \\ \widehat{\boldsymbol{\beta}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{C}^{\mathrm{T}}\boldsymbol{y} \\ \boldsymbol{y} \end{bmatrix} \tag{8.7}$$

This avoids inverting $\boldsymbol{K}$ and forming $\boldsymbol{K}^{\mathrm{T}}\boldsymbol{K}$. Note that the variance of the nonparametric coefficient $\sigma_{\boldsymbol{\beta}}^2 = \lambda^{-1}$ may be interpreted as variation due to marked additive genomic variation.

The mixed model $\boldsymbol{y} = \boldsymbol{C\theta} + \boldsymbol{K\beta} + \boldsymbol{e}$ (reparametrization I) can be reparametrized as $\boldsymbol{y} = \boldsymbol{C\theta} + \boldsymbol{u} + \boldsymbol{e}$ (reparametrization II), where $\boldsymbol{u} = \boldsymbol{K\beta}$, but with $\boldsymbol{u}$ distributed as $\boldsymbol{u} \sim N\left(\mathbf{0}, \boldsymbol{K}\sigma_u^2\right)$, and $\sigma_u^2$ is the additive variance due to lines. Both parametrizations produce the same solution since they are equivalent, with the following peculiarities. Parametrization I has two main advantages: (1) kernel matrix $\boldsymbol{K}$ does not need to be inverted. The inverse of kernel matrix $\boldsymbol{K}$ may be time-consuming or unfeasible if the number of genotyped individuals is large, because the matrix is too dense. Currently, there is the need to invert the matrix up to 100,000×100,000. (2) Genome-enabled prediction of breeding values for any $t$ new genotyped individuals ($\widehat{\boldsymbol{u}}_{\text{new}}$) without phenotype can be done using a simple matrix–vector product $\widehat{\boldsymbol{u}}_{\text{new}} = \boldsymbol{K}_s\widehat{\boldsymbol{\beta}}$, where $\widehat{\boldsymbol{\beta}}$ are the $n$ nonparametric coefficients estimated from the $n$ individuals in the training set, $\boldsymbol{K}_s$ is a matrix of dimension $(t \times n)$ containing the genomic similarity values between the $t$ new individuals whose direct genomic merits are to be predicted and the individuals in the training set. When $\boldsymbol{K}$ is the genomic relationship matrix calculated as suggested by VanRaden (2008), the kernel is linear, but when $\mathbf{K}$ is

calculated with nonlinear kernels such as the Gaussian, exponential, polynomial, arc-cosine, sigmoid, etc., the same model can be used to capture nonlinear patterns better. This means that with the mixed model equations given above, it is possible to implement any of the proposed kernels since the only difference between them is the transformation performed on the input information to obtain a particular kernel. This means that the genomic relationship matrix used in a method known as genomic BLUP (GBLUP) is replaced by a more general kernel matrix that creates similarities among individuals, even if genetically unrelated. However, for a particular data set, some kernels will perform better and others worse, since the performance of the kernels is data-dependent. In our context, a kernel is any smooth function **K** defining a covariance structure among individuals. Also, as mentioned above, we can mix many kernels using a weighted sum or product of them to create new kernels. In general, as mentioned many times in this chapter, there is enough empirical evidence that kernel methods outperform conventional regression methods that are only able to capture linear patterns (Tusell et al. 2013; Long et al. 2010; Morota et al. 2013, 2014).

The solution of the mixed model equations given in (8.6 and 8.7) can be obtained using the rrBLUP package (Endelman 2011). This package is not restricted only to linear kernels since it is also useful for estimating marker effects by Ridge regression, and BLUPs calculated based on an additive relationship matrix. In this package, variance components are estimated by either maximum likelihood (ML) or restricted maximum likelihood (REML; default) using the spectral decomposition algorithm of Kang et al. (2008). The *R* function returns the variance components, the maximized log-likelihood (LL), the ML estimate for $\boldsymbol{\theta}$, and the BLUP solution for $\boldsymbol{u}$.

The basic function for implementing the kernel methods using the rrBLUP package is given next:

```
mixed.solve(y=y, Z=Z, K=K, X=X, method="REML"),
```

where $\boldsymbol{y}$, $\boldsymbol{Z}$, $\boldsymbol{K}$, and $\boldsymbol{X}$ are the vector of response variables, the design matrix of random effects, the kernel matrix, and the design matrix of fixed effects, respectively. The estimation method by default is REML, but the ML method is also allowed. The kernel matrix is calculated before using the mixed.solve() function. It is important to point out that the function kinship.BLUP allows implementing three kernel methods directly [linear kernel (RR), Gaussian kernel (GAUSS), and Exponential kernel (EXP)] under a mixed model approach.

```
kinship.BLUP(y=y[trn], G.train=W[trn,], G.pred=W[tst,], X=X
[trn,], K.method="GAUSS", mixed.method="REML"),
```

where **y**[trn] contains the training part of the response variable, **W**[trn,] contains the markers corresponding to the training set, **W**[tst,] contains the testing set of marker data, **X**[trn,] contains the fixed effects corresponding to the training set, the K. method="GAUSS" specifies that the Gaussian kernel will be implemented, and finally, mixed.method="REML" specifies any of the two estimation methods

**Table 8.4** Prediction performance in terms of mean square error (MSE) and Pearson's correlation (PC) for each fold for the wheat599 (Environment 4) data set under a mixed model and three kernels: linear, Gaussian, and Exponential. These kernels are the defaults programmed in the rrBLUP library

| | Linear | Gaussian | Exponential | Linear | Gaussian | Exponential |
|---|---|---|---|---|---|---|
| Fold | MSE | MSE | MSE | PC | PC | PC |
| 1 | 0.757 | 0.694 | 0.720 | 0.534 | 0.592 | 0.610 |
| 2 | 0.667 | 0.626 | 0.630 | 0.536 | 0.583 | 0.590 |
| 3 | 0.774 | 0.685 | 0.700 | 0.435 | 0.521 | 0.490 |
| 4 | 0.736 | 0.609 | 0.649 | 0.398 | 0.535 | 0.509 |
| 5 | 0.677 | 0.690 | 0.685 | 0.454 | 0.428 | 0.426 |
| 6 | 1.139 | 1.036 | 1.025 | 0.309 | 0.395 | 0.405 |
| 7 | 1.006 | 0.966 | 1.010 | 0.486 | 0.523 | 0.514 |
| 8 | 0.874 | 0.758 | 0.752 | 0.486 | 0.594 | 0.616 |
| 9 | 0.683 | 0.592 | 0.586 | 0.526 | 0.621 | 0.625 |
| 10 | 0.729 | 0.683 | 0.689 | 0.315 | 0.366 | 0.365 |
| Average | 0.804 | 0.734 | 0.745 | 0.448 | 0.516 | 0.515 |

REML or ML. As mentioned above, this kinship.BLUP() allows implementing three kernels: linear, Gaussian, and Exponential. Using the wheat599 data set used in the last examples, a ten fold cross-validation using the three default kernels was implemented.

Table 8.4 shows that under a mixed model approach using the default kernels [linear (RR), Gaussian (GAUSS), and Exponential (EXP)] available in the rrBLUP library, the best predictions were observed in Env4 of data set wheat599 with the Gaussian and Exponential kernels under both metrics. The R code for reproducing the results in Table 8.4 is given in Appendix 3.

Table 8.5 provides the results of nine kernels for the response variable of Env4. The building process was manual for the kernel matrices and the data set used for this example was the wheat599 data set. Results for each of the nine kernels are given for each fold and across the 10 folds. Table 8.5 shows that the best prediction performance was observed with the Gaussian kernel and the worst under the polynomial kernel. The R code for reproducing the results in Table 8.5 is given in Appendix 4.

## 8.7 Hyperparameter Tuning for Building the Kernels

Hand-tuning kernel functions can be time-consuming and requires expert knowledge. A tuned kernel can improve the trained model, if standard kernels are insufficient for achieving a good transformation. In this section, we illustrate how to tune kernels and we compare the standard kernel with a hand-tuned kernel. As pointed out in Chap. 4, one approach to tuning is to divide the data into a training set, a tuning set, and a testing set. The training set is for training the data, the tuning set is for

**Table 8.5** Prediction performance in terms of mean square error (MSE) for each fold for the wheat599 (Environment 4) data set under a mixed model, manually building the kernels linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4

| Fold | Linear | Polynomial | Sigmoid | Gaussian | Exponential | AK1 | AK2 | AK3 | AK4 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.763 | 0.848 | 0.772 | 0.688 | 0.691 | 0.714 | 0.694 | 0.693 | 0.693 |
| 2 | 0.667 | 0.737 | 0.664 | 0.632 | 0.630 | 0.645 | 0.630 | 0.627 | 0.626 |
| 3 | 0.780 | 0.817 | 0.799 | 0.701 | 0.725 | 0.728 | 0.713 | 0.714 | 0.716 |
| 4 | 0.736 | 0.672 | 0.756 | 0.596 | 0.655 | 0.670 | 0.639 | 0.634 | 0.631 |
| 5 | 0.682 | 0.727 | 0.703 | 0.722 | 0.695 | 0.681 | 0.701 | 0.707 | 0.710 |
| 6 | 1.139 | 1.077 | 1.176 | 1.015 | 1.051 | 1.094 | 1.057 | 1.047 | 1.040 |
| 7 | 1.012 | 1.078 | 1.023 | 0.987 | 0.986 | 0.977 | 0.969 | 0.971 | 0.975 |
| 8 | 0.875 | 0.927 | 0.899 | 0.758 | 0.756 | 0.805 | 0.767 | 0.761 | 0.758 |
| 9 | 0.684 | 0.742 | 0.708 | 0.590 | 0.602 | 0.630 | 0.608 | 0.606 | 0.606 |
| 10 | 0.730 | 0.696 | 0.729 | 0.677 | 0.701 | 0.725 | 0.705 | 0.700 | 0.697 |
| Average | 0.807 | 0.832 | 0.823 | 0.736 | 0.749 | 0.767 | 0.748 | 0.746 | 0.745 |

choosing the best hyperparameter combination, and the testing set is for evaluating the prediction performance with the best hyperparameters. However, when the data sets are small after selecting the best combination of hyperparameters, the training and tuning sets are joined into one data set and with this data set the model is refitted again with the best combination of hyperparameters, and finally, the prediction performance is evaluated with the testing set.

This conventional approach to tuning is illustrated next using the wheat599 data set. To illustrate how to choose the hyperparameter, we will work with the arc-cosine kernel where the hyperparameter to be tuned is the number of hidden layers, for which we used a grid of 10 values (1, 2,. . ., 9, 10). To be able to tune the number of hidden layers, first we divided the original data set into four mutually exclusive parts with four fold cross-validation. This is called the outer cross-validation strategy. Then three of these parts were used for training and the remaining for testing. A ten fold cross-validation was performed in each of the outer training sets; this is called inner cross-validation. Nine out of the ten formed the inner training set and the remaining the tuning set. Then for each of the outer folds, the grid was evaluated with 10 values in the grid for the number of hidden layers, with the inner ten fold cross-validation strategy and for each of the 10 tuning sets, the mean square error of prediction was computed for each of the 10 values of the hidden layers in the grid. Then the average mean square error of the 10 inner cross-validations (in each outer fold) was computed for each value in the grid; it was selected as the optimal number of hidden layers in the grid that provides the smallest MSE. Then the inner training and the tuning sets (inner testing) were joined together for refitting the model with the optimal number of hidden layers, and finally, the prediction performance also in terms of MSE was evaluated in the outer testing set. The average of the four values of the outer testing set is reported as the final prediction performance. Under this strategy, the optimal number of hidden layers is different for each fold.

Figure 8.2 shows that the optimal number of hidden layers is different in each fold. In folds 1 and 3 (Fig. 8.2a, c), the optimal number of hidden layers was equal to 3, in folds 3 and 4 (Fig. 8.2b, d), the optimal number of hidden layers was equal to 8. Finally, with these optimal values, the model was refitted with the information of the inner training + tuning set, and then for each fold, the mean square error (MSE) was calculated for each outer testing set; the MSEs were 0.6878 (fold 1), 0.6963 (fold 2), 0.9725 (fold 3), and 0.7212 (fold 4), with an MSE across folds equal to 0.7694. The R code for reproducing these results is given in Appendix 5.

## 8.8 Bayesian Kernel Methods

For a single environment, the model can be expressed as

$$\boldsymbol{y} = \mu\mathbf{1} + \boldsymbol{u} + \boldsymbol{e}, \tag{8.8}$$

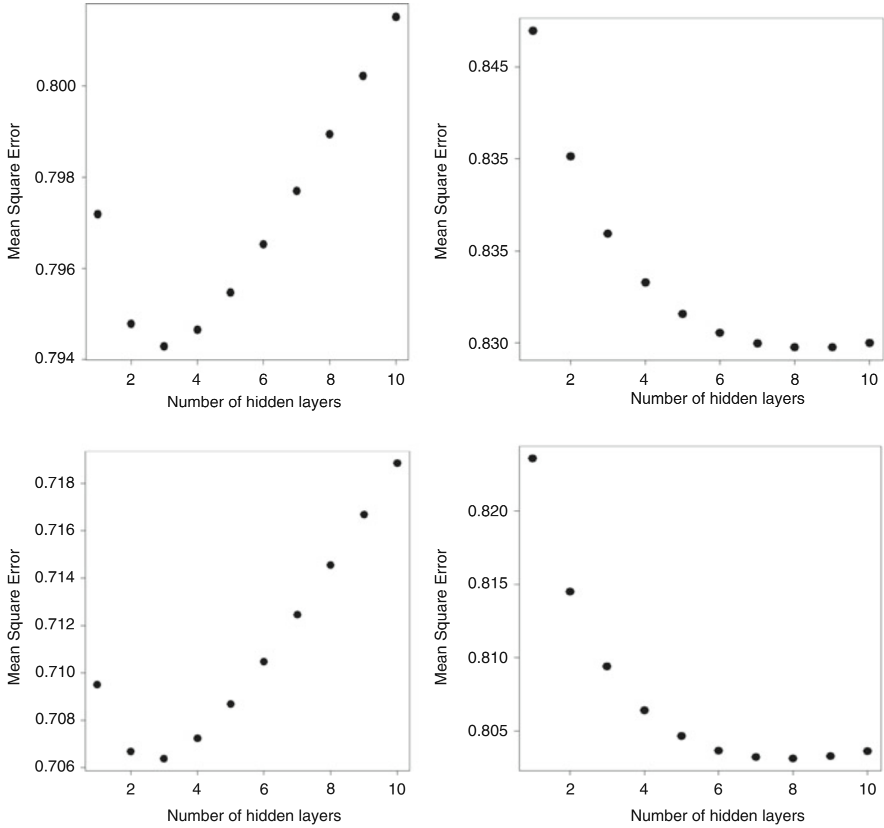



**Fig. 8.2** Optimal number of hidden layers in each fold using a grid with values 1 to 10 and an inner ten fold cross-validation

where $\mu$ is the overall mean, $\mathbf{1}$ is the vector of ones, and $\boldsymbol{y}$ is the vector of observations of size $n$. Moreover, $\boldsymbol{u}$ is the vector of genomic effects $\boldsymbol{u} \sim \boldsymbol{N}\left(\mathbf{0}, \sigma_u^2 \boldsymbol{K}\right)$, where $\sigma_u^2$ is the genomic variance estimated from the data, and matrix $\boldsymbol{K}$ is the kernel constructed with any of the kernel methods explained above (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, ...). The random residuals are assumed independent with normal distribution $\boldsymbol{e} \sim \boldsymbol{N}\left(\mathbf{0}, \sigma_e^2 \boldsymbol{I}\right)$, where $\sigma_e^2$ is the error variance.

Now the kernel Ridge regression is cast under a Bayesian framework with $\lambda = \sigma_e^2/\sigma_u^2$, where $\sigma_e^2$ and $\sigma_u^2$ are the residual and variance attached to $\boldsymbol{u}$, respectively. With a flat prior to mean parameter ($\mu$), $\sigma_e^2 \sim \chi_{v,S}^{-2}$, and the induced priors $\boldsymbol{u} \mid \sigma_u^2 \sim N_n\left(\mathbf{0}, \boldsymbol{K}\sigma_g^2\right)$ and $\sigma_u^2 \sim \chi_{v_u,\ S_u}^{-2}$, the full conditional posterior distribution of $\boldsymbol{u}$ in model (8.8) is given by

$$
\begin{aligned}
f(\boldsymbol{u}|-) &\propto L\left(\mu, \boldsymbol{u}, \sigma_e^2; \boldsymbol{y}\right) f\left(\boldsymbol{u}|\sigma_u^2\right) \\
&\propto \frac{1}{\left(2\pi\sigma_e^2\right)^{\frac{n}{2}}} \exp\left[-\frac{1}{2\sigma_e^2}\boldsymbol{y} - 1_n\mu - \boldsymbol{u}^2\right] \frac{1}{\left(\sigma_u^2\right)^{\frac{n}{2}}} \exp\left(\left[-\frac{1}{2\sigma_u^2}\boldsymbol{u}^{\mathrm{T}}\boldsymbol{K}^{-1}\boldsymbol{u}\right]\right) \\
&\propto \exp\left\{-\frac{1}{2}\left[(\boldsymbol{u} - \widetilde{\boldsymbol{u}})^{\mathrm{T}}\widetilde{\boldsymbol{K}}^{-1}(\boldsymbol{u} - \widetilde{\boldsymbol{u}})\right]\right\},
\end{aligned}
$$

where $\widetilde{\boldsymbol{K}} = \left(\sigma_u^{-2}\boldsymbol{K}^{-1} + \sigma_e^{-2}\boldsymbol{I}_n\right)^{-1}$ and $\widetilde{\boldsymbol{u}} = \sigma_e^{-2}\widetilde{\boldsymbol{K}}(\boldsymbol{y} - \mathbf{1}_n\mu)$, and from here $\boldsymbol{u} \mid - \sim N_n\left(\widetilde{\boldsymbol{u}}, \widetilde{\boldsymbol{K}}\right)$. Then the mean/mode of $\boldsymbol{u} \mid -$ is $\widetilde{\boldsymbol{u}} = \sigma_e^{-2}\widetilde{\boldsymbol{K}}(\boldsymbol{y} - \mathbf{1}_n\mu)$**,** which is also the BLUP of $\boldsymbol{u}$ under the mixed model equation of Henderson (1975). For this reason, model (8.8) is often referred to as GBLUP. However, here the genomic relationship matrix (GRM; or pedigree matrix P) was replaced by any kernel $\boldsymbol{K}$; for this reason, under a Bayesian framework, we call this model a Bayesian kernel BLUP, which is reduced to the pedigree (P) or Genomic (G) BLUP when we use the GRM or pedigree matrix as the kernel.

The full conditional posterior of the rest of the parameters is equal to the GBLUP model described in Chap. 6: $\mu \mid - \sim N\left(\widetilde{\mu}, \widetilde{\sigma}_0^2\right)$, where $\widetilde{\sigma}_0^2 = \frac{\sigma_e^2}{n}$ and $\widetilde{\mu} = \frac{1}{n}\mathbf{1}_n^T(\boldsymbol{y} - \boldsymbol{u})$; $\sigma_e^{-2} \mid - \sim \chi^{-2}_{\widetilde{v},\widetilde{S}}$, where $\widetilde{v} = v + n$ and $\widetilde{S} = S + \|\boldsymbol{y} - \mathbf{1}_n\mu - \boldsymbol{u}\|^2$; and $\sigma_u^2 \mid - \sim \chi^{-2}_{\widetilde{v}_u,\widetilde{S}_u}$, where $\widetilde{v}_u = v_u + n$ and $\widetilde{S}_u = \boldsymbol{u}^T\boldsymbol{K}^{-1}\boldsymbol{u}$. The Bayesian kernel BLUP, like the GBLUP, does not face the large $p$ and small $n$ problem, since due to the kernel trick, a problem of dimensionality $p$ is converted into an $n$-dimensional problem.

The Bayesian kernel BLUP model (8.8) can also be implemented easily with the BGLR R package, and when the hyperparameters $S$-$v$ and $S_u$-$v_u$ are not specified, $v = v_u = 5$ is used by default and the scale parameters are settled as in the BRR. However, a two-step process is required for its implementation: Step 1: Select and compute the kernel matrix to be used. Step 2: Use this kernel matrix to implement the model using the BGLR package.

The BGLR code to fit this model is

```
ETA = list( list( model = 'RHKS', K = K , df0 = vu, S0 = Su,  R2 = 1-R2)) )
 A   = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 =
R2)
```

When individuals had more than one replication, or a sophisticated experimental design was used for data collection, the Bayesian kernel BLUP model is specified in a more general way to take into account this structure, as follows:

$$
\boldsymbol{Y} = \mathbf{1}_n\mu + \boldsymbol{Zu} + \boldsymbol{e} \tag{8.9}
$$

with $\boldsymbol{Z}$ the incident matrix of the genomic effects. This model cannot be fitted directly in the BGLR and some precalculus is needed first to compute the "covariance" matrix of the predictor $\boldsymbol{Zu}$ in model (8.9): $\mathbf{K}_* = \mathrm{Var}(\boldsymbol{Zu}) = \boldsymbol{ZKZ}^{\mathrm{T}}$. The BGLR code for implementing this model is the following:

```
  Z = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
  K_start = Z%*%K%*%t(Z)
  ETA = list( list( model = 'RHKS', K = K_start , df0 = vu, S0 = Su,  R2 =
1-R2)) )
  A   = BGLR(y=y, ETA = ETA, nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 =
R2)
```

To illustrate how to implement the Bayesian kernel BLUP model in BGLR, some examples are provided next.

**Example 2** We again consider the prediction of grain yield (tons/ha) based on marker information. The data set used consists of 30 lines in four environments with one and two repetitions, and the genotype information consists of 500 markers for each line. The numbers of lines with one (two) repetition are 6 (24), 2 (28), 0 (30), and 3 (27) in Environments 1, 2, 3, and 4, respectively, resulting in 229 observations. The performance prediction of all these models was evaluated with 10 random partitions using a cross-validation strategy, where 80% of the complete data set was used to fit the model and the rest to evaluate the model in terms of the mean squared error (MSE) of prediction. Nine kernels were evaluated (linear=GBLUP, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4). The R code for implementing this model is given in Appendix 6.

The results for all kernels (shown in Table 8.6) were obtained by iterating 10,000 times the corresponding Gibbs sampler and discarding the first 1000 of them, using the default hyperparameter values implemented in BGLR. We can observe that the worst and second worst prediction performances were obtained under the sigmoid and linear (GBLUP) kernels, while the best and second-best predictions were obtained with polynomial and Gaussian kernels. However, it is important to point out that the differences between the best and worst predictions were small.

### *8.8.1 Extended Predictor Under the Bayesian Kernel BLUP*

The Bayesian kernel BLUP method can be extended, in terms of the predictor, to easily take into account the effects of other factors. For example, in addition to the genotype effect, the effects of environments and genotype × environment interaction terms can also be incorporated as

$$\boldsymbol{y} = \mu\mathbf{1} + \boldsymbol{Z}_E\boldsymbol{\beta}_E + \boldsymbol{u}_1 + \boldsymbol{u}_2 + \boldsymbol{\varepsilon}, \tag{8.10}$$

where $\boldsymbol{y} = [\boldsymbol{y}_1, \ldots, \boldsymbol{y}_I]'$ are the observations collected in each of the $I$ sites (or environments). The fixed effects of the environment are modeled with the incidence matrix of environments $\boldsymbol{Z}_E$, where the parameters to be estimated are the intercepts for each environment ($\boldsymbol{\beta}_E$) (other fixed effects can be incorporated into the model). In this model, $\boldsymbol{u}_1 \sim N\left(\mathbf{0}, \sigma^2_{u_1}\boldsymbol{K}_1\right)$ represents the genomic main effects, $\sigma^2_{u_1}$ is

**Table 8.6** Mean squared error (MSE) of prediction across 10 random partitions, with 80% for training and the rest for testing, under nine kernel methods

| Partition | Linear | Polynomial | Sigmoid | Gaussian | Exponential | AK1 | AK2 | AK3 | AK4 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.578 | 0.569 | 0.608 | 0.566 | 0.568 | 0.571 | 0.570 | 0.565 | 0.566 |
| 2 | 0.355 | 0.360 | 0.371 | 0.353 | 0.353 | 0.357 | 0.356 | 0.353 | 0.354 |
| 3 | 0.443 | 0.432 | 0.472 | 0.428 | 0.429 | 0.435 | 0.433 | 0.431 | 0.431 |
| 4 | 0.387 | 0.392 | 0.399 | 0.372 | 0.374 | 0.377 | 0.380 | 0.377 | 0.375 |
| 5 | 0.372 | 0.329 | 0.416 | 0.334 | 0.335 | 0.351 | 0.345 | 0.343 | 0.342 |
| 6 | 0.810 | 0.753 | 0.864 | 0.789 | 0.800 | 0.797 | 0.794 | 0.797 | 0.792 |
| 7 | 0.757 | 0.740 | 0.779 | 0.752 | 0.756 | 0.749 | 0.752 | 0.751 | 0.748 |
| 8 | 0.352 | 0.340 | 0.362 | 0.350 | 0.352 | 0.352 | 0.348 | 0.348 | 0.348 |
| 9 | 0.297 | 0.302 | 0.306 | 0.294 | 0.293 | 0.295 | 0.295 | 0.292 | 0.292 |
| 10 | 0.551 | 0.565 | 0.552 | 0.549 | 0.550 | 0.554 | 0.550 | 0.550 | 0.548 |
| Average | 0.490 | 0.478 | 0.513 | 0.479 | 0.481 | 0.484 | 0.482 | 0.481 | 0.480 |

the genomic variance component estimated from the data, and $\boldsymbol{K_1} = \boldsymbol{Z}_{u1}\boldsymbol{K}\boldsymbol{Z}'_{u1}$, where $\boldsymbol{Z}_{u1}$ relates the genotypes to the phenotypic observations. The random effect $\boldsymbol{u_2}$ represents the interaction between the genomic effects and environments and is modeled as $\boldsymbol{u_2} \sim N\left(\mathbf{0}, \sigma^2_{u_2}\boldsymbol{K_2}\right)$, where $\boldsymbol{K_2} = \left(\boldsymbol{Z}_{u1}\boldsymbol{K}\boldsymbol{Z}'_{u1}\right)^{\circ}\left(\boldsymbol{Z}_E\boldsymbol{Z}_E{}'\right)$, where $^{\circ}$ is the Hadamard product. The BGLR specification for this Bayesian kernel BLUP model with the extended predictor is exactly the same as the GBLUP method studied in Chap. 6, but instead of using the genomic relationship matrix (linear kernel), now any of the kernels mentioned above is specified:

```
XE = model.matrix(~0+Env,data=dat_F)[,-1]
K.E=XE%*%t(XE)
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
K_L=Z_L%*%K%*%t(Z_L) ###### K is the kernel matrix
K_LE= K.E*K_L
ETA_K=list(list(model='FIXED',X=XE),list(model='RKHS',K=K_L),
list(model='RKHS',K=K_LE))
 y_NA = y
 y_NA[Pos_tst] = NA
A = BGLR(y=y_NA,ETA=ETA_K,nIter = 1e4,burnIn = 1e3,verbose = FALSE,
nIter = 1e4, burnIn = 1e3, S0 = S, df0 = v, R2 = R^2)
```

Now to illustrate the Bayesian kernel BLUP with the extended predictor described in Eq. (8.10), we used a data set that contains 30 lines in four environments, and the genotyped information is composed of 500 markers for each line. We call this data set dat_ls_E2. Now only the following kernels were implemented: linear, polynomial, sigmoid, Gaussian, exponential, AK1, and AK4. The R code for reproducing the results in Table 8.7 is given in Appendix 7. We can observe that taking into account the genotype by environment interaction in the predictor, the best

**Table 8.7** Mean squared error (MSE) of prediction across 10 random partitions, with 80% for training and the rest for testing, under seven kernel methods with the predictor including the effects of environment + genotypes + genotype × environment interaction term. Here we used the dat_ls_E2 data set

| Partition | Linear | Polynomial | Sigmoid | Gaussian | Exponential | AK1 | AK4 |
|---|---|---|---|---|---|---|---|
| 1 | 0.729 | 0.662 | 0.775 | 0.665 | 0.930 | 0.700 | 0.742 |
| 2 | 0.533 | 0.559 | 0.596 | 0.499 | 0.573 | 0.515 | 0.496 |
| 3 | 0.691 | 0.629 | 0.724 | 0.633 | 0.634 | 0.654 | 0.620 |
| 4 | 0.646 | 0.621 | 0.678 | 0.631 | 0.748 | 0.626 | 0.633 |
| 5 | 0.517 | 0.550 | 0.488 | 0.517 | 0.490 | 0.519 | 0.516 |
| 6 | 0.674 | 0.650 | 0.683 | 0.597 | 0.586 | 0.640 | 0.607 |
| 7 | 0.419 | 0.435 | 0.474 | 0.376 | 0.558 | 0.403 | 0.397 |
| 8 | 0.400 | 0.409 | 0.406 | 0.359 | 0.349 | 0.381 | 0.361 |
| 9 | 0.618 | 0.611 | 0.641 | 0.589 | 0.586 | 0.605 | 0.587 |
| 10 | 0.539 | 0.494 | 0.567 | 0.473 | 0.493 | 0.507 | 0.485 |
| Average | 0.576 | 0.562 | 0.603 | 0.534 | 0.595 | 0.555 | 0.544 |

prediction performance was obtained with the Gaussian kernel while the worst was obtained under the sigmoid kernel.

It is important to point out that in the predictor under a Bayesian kernel BLUP using BGLR, as many terms as desired can be included, and the specification is very similar to how it was done with three terms in the predictor in this example. Using BGLR, the Bayesian kernel BLUP can be implemented for binary and ordinal response variables. Next, we provide one example for binary response variables and one for categorical response variables.

### 8.8.2 *Extended Predictor Under the Bayesian Kernel BLUP with a Binary Response Variable*

It is important to point out that it is feasible to implement the Bayesian kernel BLUP with binary response variables using the probit link function in BGLR. This implementation first requires calculating the kernel to be used; then with the following lines of code, the Bayesian kernel BLUP can be fitted for binary and categorical response variables:

```
XE = model.matrix(~0+Env,data=dat_F)[,-1]
K.E=XE%*%t(XE)
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
K_L=Z_L%*%K%*%t(Z_L)
K_LE= K.E*K_L
ETA_K=list(list(model='FIXED',X=XE),list(model='RKHS',K=K_L),
        list(model='RKHS',K=K_LE))
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K, response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
 Probs = A$probs[Pos_tst,]
```

When categorical response variables are used, two different things need to be modified to fit the model in BGLR. The first one is that we need to specify response_type="ordinal" and the other is that the outputs now are the probabilities that can be extracted with A$probs. When response_type="ordinal" is ignored, the response variable is assumed Gaussian by default.

To give an example with a binary response variable, we used the EYT Toy data set (Data_Toy_EYT. RData) that is preloaded in the BMTME library. This data set is composed of 40 lines, four environments (Bed5IR, EHT, Flat5IR, and LHT), and four response variables: DTHD, DTMT, GY, and Height. G_Toy_EYT is the genomic relationship matrix of dimension $40\times 40$. The first two variables are ordinal with three categories, the third is continuous (GY = Grain yield) and the last one (Height) is binary. In this example, we work with only the binary response variable (Height).

**Table 8.8** Proportion of cases correctly classified (PCCC) across 10 random partitions, with 80% for training and the rest for testing, under seven kernel methods with the predictor including the effects of environment + genotypes + genotype × environment interaction term with the Data_Toy_EYT with trait Height

| Partition | Linear | Polynomial | Sigmoid | Gaussian | Exponential | AK1 | AK4 |
|---|---|---|---|---|---|---|---|
| 1 | 0.781 | 0.781 | 0.750 | 0.719 | 0.625 | 0.813 | 0.844 |
| 2 | 0.594 | 0.625 | 0.594 | 0.719 | 0.750 | 0.813 | 0.844 |
| 3 | 0.688 | 0.625 | 0.688 | 0.656 | 0.719 | 0.688 | 0.688 |
| 4 | 0.688 | 0.656 | 0.656 | 0.563 | 0.656 | 0.781 | 0.781 |
| 5 | 0.406 | 0.500 | 0.406 | 0.563 | 0.531 | 0.531 | 0.563 |
| 6 | 0.656 | 0.656 | 0.656 | 0.656 | 0.688 | 0.688 | 0.719 |
| 7 | 0.625 | 0.625 | 0.625 | 0.656 | 0.688 | 0.688 | 0.719 |
| 8 | 0.719 | 0.688 | 0.688 | 0.719 | 0.719 | 0.719 | 0.719 |
| 9 | 0.500 | 0.563 | 0.500 | 0.688 | 0.719 | 0.750 | 0.719 |
| 10 | 0.531 | 0.563 | 0.531 | 0.688 | 0.594 | 0.688 | 0.688 |
| Average | 0.619 | 0.628 | 0.609 | 0.663 | 0.669 | 0.716 | 0.728 |

Table 8.8 gives the results of implementing the Bayesian kernel BLUP method under a binary response variable with seven kernels using the ETY Toy data set with trait Height. The best predictions using the EYT Toy data set were obtained with kernel AK4, and the worst was under kernel sigmoid. Again, we can see that, in general, most kernel methods outperform the linear kernel. The R code for reproducing the results in Table 8.8 is given in Appendix 8.

### 8.8.3 *Extended Predictor Under the Bayesian Kernel BLUP with a Categorical Response Variable*

The fitting process in BGLR for the categorical response variable is exactly the same as the binary response variable explained above. For this reason, the results given in Table 8.9 for the categorial response variable were obtained with the same R code given in Appendix 8 with the following two modifications: (a) y=dat_F$DTMT instead of y=dat_F$Height and b) yp_ts=apply(Probs,1,which.max) instead of yp_ts=apply(Probs,1,which.max)-1, since now the response variable has levels 1, 2, and 3.

Table 8.9 shows that the best prediction performance for the categorical response variable was observed in the polynomial kernel while the worst was under the AK4 kernel.

**Table 8.9** Proportion of cases correctly classified (PCCC) across 10 random partitions, with 80% for training and the rest for testing, under seven kernel methods with the predictor including effects of environment + genotypes + genotype × environment interaction term with the Data_Toy_EYT with the ordinal trait DTMT

| Fold | Linear | Polynomial | Sigmoid | Gaussian | Exponential | AK1 | AK4 |
|---|---|---|---|---|---|---|---|
| 1 | 0.656 | 0.656 | 0.656 | 0.688 | 0.688 | 0.688 | 0.625 |
| 2 | 0.688 | 0.688 | 0.688 | 0.656 | 0.688 | 0.688 | 0.688 |
| 3 | 0.813 | 0.813 | 0.813 | 0.813 | 0.813 | 0.781 | 0.781 |
| 4 | 0.719 | 0.719 | 0.719 | 0.656 | 0.656 | 0.688 | 0.656 |
| 5 | 0.563 | 0.625 | 0.563 | 0.656 | 0.656 | 0.625 | 0.656 |
| 6 | 0.719 | 0.750 | 0.750 | 0.781 | 0.750 | 0.750 | 0.750 |
| 7 | 0.625 | 0.656 | 0.625 | 0.625 | 0.594 | 0.625 | 0.594 |
| 8 | 0.594 | 0.594 | 0.594 | 0.594 | 0.594 | 0.594 | 0.594 |
| 9 | 0.688 | 0.688 | 0.688 | 0.656 | 0.656 | 0.625 | 0.625 |
| 10 | 0.750 | 0.719 | 0.781 | 0.750 | 0.719 | 0.750 | 0.719 |
| Average | 0.681 | 0.691 | 0.688 | 0.688 | 0.681 | 0.681 | 0.669 |

## 8.9 Multi-trait Bayesian Kernel

In BGLR, it is possible to fit multi-trait Bayesian kernel BLUP methods, and the fitting process is exactly the same as fitting multi-trait Bayesian GBLUP methods (see Chap. 6). The only difference is that instead of using a linear kernel, any kernel can be used. The basic R code for fitting multi-trait Bayesian kernel methods is given next:

```
  y_NA = data.matrix(y)
  y_NA[Pos_tst,] = NA
  A4= Multitrait(y = y_NA, ETA=ETA_K.Gauss,resCov = list(type ="UN",
S0=diag(4),df0= 5), nIter =10000, burnIn = 1000)
  Me_Gauss= PC_MM_f(y[Pos_tst,],A4$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  A4$ETAHat[Pos_tst,]
```

To illustrate the fitting process of a multi-trait Bayesian kernel BLUP method, we used the EYT Toy data set (Data_Toy_EYT. RData), but using the four response variables simultaneously (even though only GY is Gaussian, we assumed that the four response variables satisfy this assumption). The R code for its implementation is given in Appendix 9. Table 8.10 gives the results of the prediction performance in terms of the mean square error across the 10 random partitions, where the best kernel for prediction differs between environment-trait combinations. The polynomial kernel and the linear kernel were the best in 4 out of 16 environment-trait combinations.

**Table 8.10** Mean square error (MSE) of prediction across 10 random partitions, with 80% for training and the rest for testing, under seven kernel methods with the predictor including the effects of environment + genotypes + genotype × environment interaction term with the Data_Toy_EYT assuming that the multi-trait response is Gaussian

| Env | Trait | Linear | Polynomial | Sigmoid | Gaussian | Exponential | AK1 | AK4 |
|---|---|---|---|---|---|---|---|---|
| EHT | DTHD | 0.302 | 0.292 | 0.303 | **0.274** | 0.268 | 0.280 | 0.280 |
| EHT | DTMT | 0.164 | 0.180 | 0.162 | 0.154 | **0.152** | 0.158 | 0.158 |
| EHT | GY | 0.333 | 0.396 | **0.321** | 0.342 | 0.348 | 0.354 | 0.354 |
| EHT | Height | 0.167 | 0.149 | 0.162 | 0.156 | 0.150 | **0.148** | **0.148** |
| LHT | DTHD | 0.261 | 0.243 | 0.259 | 0.229 | **0.220** | 0.234 | 0.234 |
| LHT | DTMT | 0.357 | 0.341 | 0.360 | 0.322 | **0.310** | 0.326 | 0.326 |
| LHT | GY | 0.328 | **0.302** | 0.322 | 0.319 | 0.333 | 0.318 | 0.318 |
| LHT | Height | **0.228** | 0.230 | 0.233 | 0.232 | 0.230 | 0.232 | 0.232 |
| Bed5IR | DTHD | 0.150 | 0.160 | 0.151 | **0.148** | 0.153 | 0.155 | 0.155 |
| Bed5IR | DTMT | **0.202** | 0.228 | 0.199 | 0.203 | 0.210 | 0.214 | 0.214 |
| Bed5IR | GY | 0.138 | 0.138 | 0.136 | 0.136 | 0.135 | **0.134** | **0.134** |
| Bed5IR | Height | 0.189 | **0.178** | 0.189 | 0.185 | 0.184 | 0.182 | 0.182 |
| Flat5IR | DTHD | **0.186** | 0.190 | 0.189 | 0.194 | 0.200 | 0.191 | 0.191 |
| Flat5IR | DTMT | **0.211** | 0.224 | **0.211** | 0.216 | 0.226 | 0.219 | 0.219 |
| Flat5IR | GY | 0.477 | **0.465** | 0.478 | 0.478 | 0.469 | 0.471 | 0.471 |
| Flat5IR | Height | 0.179 | **0.178** | 0.180 | 0.183 | 0.182 | 0.183 | 0.183 |

## 8.10 Kernel Compression Methods

By kernel compression methods, we mean those tools that allow us to approximate kernels without affecting the prediction accuracy very much, but gaining a significant reduction in computational resources. There are many methods for compression of kernel methods. However, in this section, we only illustrate the method proposed by Cuevas et al. (2020). The basic idea of this method consists of approximating the original kernel using a small size ($m$) of the original $n$ observations (lines in the context of GS) available in the training set, which significantly reduces the required computational resources required to build the kernel matrix.

Before giving the details of the compression of kernels proposed by Cuevas et al. (2020), it is important to point out that model (8.8) can be reparametrized as Eq. (8.11) if the eigenvalue decomposition of the kernel matrix $\boldsymbol{K}$ is expressed as $\boldsymbol{US}^{1/2}\boldsymbol{S}^{1/2}\boldsymbol{U}'$,

$$\boldsymbol{y} = \mu 1_n + \boldsymbol{Pf} + \boldsymbol{\varepsilon}, \tag{8.11}$$

where $\boldsymbol{f} \sim N\left(\boldsymbol{0}, \sigma_f^2 \boldsymbol{I}_{r,r}\right)$ (where $r$ is the rank of $\boldsymbol{K}$) and $\boldsymbol{P} = \boldsymbol{US}^{1/2}$. Note that models (8.8) and (8.11) are equivalent. Model (8.11) can be fitted by the conventional Ridge regression model. This Ridge regression reparameterization most of the time is computationally very efficient, since most of the time $r < \min\ (n, p)$, which is common in multi-environment and/or multi-trait models. It should be noted that

only $r$ effects can be summarized and projected for $\boldsymbol{P}$ to explain the $n$ effects without any significant loss of precision with the available information.

Next, we describe the Cuevas et al. (2020) method for compressing the kernel matrix $\boldsymbol{K}$. First, the method approximates the original kernel matrix ($\boldsymbol{K}$) using a smaller sub-matrix $\boldsymbol{K}_{m,m}$ ($m < n$) constructed with $m$ out of $n$ lines. The rank of $K_{m,m}$ is $m$. Under the assumption that the row vectors are linearly independent, Williams and Seeger (2001) showed that the Nyström approximation of the kernel is as follows:

$$\boldsymbol{K} \approx \boldsymbol{Q} = \boldsymbol{K}_{n,m}\boldsymbol{K}_{m,m}^{-1}\boldsymbol{K}'_{n,m},$$

where $\boldsymbol{Q}$ will have the rank of $\boldsymbol{K}_{m,m}$, that is, $m$. The computation of this kernel is facilitated since it is not necessary to compute and store the original matrix $\boldsymbol{K}$, since only $\boldsymbol{K}_{m,m}$ and $\boldsymbol{K}_{n,m}$ are required.

Therefore, $\boldsymbol{K}_{m,m}$ can be computed with $m$ lines with all the $p$ markers, that is, $\boldsymbol{X}_{m,p}$. For the **linear kernel** (GBLUP), $\boldsymbol{K}_{m,m} = \frac{\boldsymbol{X}_{m,p}\boldsymbol{X}'_{m,p}}{p}$ and $\boldsymbol{K}_{n,m} = \frac{\boldsymbol{X}_{n,p}\boldsymbol{X}'_{m,p}}{p}$ which captures the relationship of all $n$ lines with the $m$ lines. Note that in the construction of $\boldsymbol{Q}$, all the $n$ lines and all the $p$ markers are considered, but not all their relationships are accounted for. For example, relationships $\boldsymbol{K}_{n-m,n-m} = \frac{\boldsymbol{X}_{n-m,p}\boldsymbol{X}'_{n-m,p}}{p}$ are not considered (where $n - m$ represents the complement to the $m$ lines). To try to explain this, we ordered the elements of matrix $\boldsymbol{Q}$ per block, such that $\boldsymbol{Q}_{n,n} = \begin{bmatrix} \boldsymbol{Q}_{m,m} & \boldsymbol{Q}_{m,n-m} \\ \boldsymbol{Q}_{n-m,m} & \boldsymbol{Q}_{n-m,n-m} \end{bmatrix}$.

Rassmussen and Williams (2006) showed that $\boldsymbol{Q}_{m,m} = \boldsymbol{K}_{m,m}$, $\boldsymbol{Q}_{n-m,m} = \boldsymbol{K}_{n-m,m}$, $\boldsymbol{Q}_{m,n-m} = \boldsymbol{K}_{m,n-m}$, and that the difference $\boldsymbol{K}_{n-m,n-m} - \boldsymbol{Q}_{n-m,n-m}$, that is, $\boldsymbol{K}_{n-m,n-m} - \boldsymbol{K}_{n-m,m}\boldsymbol{K}_{m,m}^{-1}\boldsymbol{K}_{m,n-m}$ is known as the Schur complement of $\boldsymbol{K}_{m,m}$ on $\boldsymbol{K}_{n,\,n}$. Then, because it is assumed that $\boldsymbol{K}_{m,m}$ and $\boldsymbol{K}_{n,n}$ are positive semi-definite, their difference is also positive semi-definite: $\boldsymbol{Q}_{n,n} = \begin{bmatrix} \boldsymbol{K}_{m,m} & \boldsymbol{K}_{m,n-m} \\ \boldsymbol{K}_{n-m,m} & \boldsymbol{Q}_{n-m,n-m} \end{bmatrix}$. Assuming the effects of $\boldsymbol{u}_{n\,-\,m} \mid \boldsymbol{u}_m$ are conditional and independent, Snelson and Ghahramani (2006) and Misztal et al. (2014) proposed substituting the diagonal of the differences of $\boldsymbol{Q}_{n\,-\,m,\,n\,-\,m}$ with the diagonal of $\boldsymbol{K}_{n\,-\,m,\,n\,-\,m}$.

In the method called projected process, Seeger et al. (2003) theoretically showed that using all lines and considering the minimum Kullback–Leibler distance $\mathrm{KL}(q(\boldsymbol{u}|\boldsymbol{y})\|p(\boldsymbol{u}|\boldsymbol{y}))$ justify that the matrix $\boldsymbol{K}$ in the prior distribution of $\boldsymbol{u}$ (of model 8.8) can be substituted for the $\boldsymbol{Q}$ approximations from Nyström (Titsias 2009). That is, the random genetic vectors have a normal distribution $\boldsymbol{u} \sim N\left(\boldsymbol{0}, \sigma_u^2\boldsymbol{Q}\right)$, where $\boldsymbol{Q} = \boldsymbol{K}_{n,m}\boldsymbol{K}_{m,m}^{-1}\boldsymbol{K}'_{n,m}$.

With these adjustments in the distribution of the random effects $\boldsymbol{u}$, we used model (8.8) for prediction. It is common to estimate parameters $\sigma_e^2$ and $\sigma_u^2$ of the model with the marginal likelihood and then predict the random effects using the inversion lemma, which is fast. Furthermore, if matrix $\boldsymbol{Q}$ is directly used in BGLR, there is no

advantage in terms of saving computational resources using the approximate kernel. Therefore, an eigen decomposition of $\boldsymbol{K}_{m,m}^{-1} = \boldsymbol{U}\boldsymbol{S}^{-1/2}\boldsymbol{S}^{-1/2}\boldsymbol{U}'$ is used where $\boldsymbol{U}$ are the eigenvectors of order $m \times m$ and $\boldsymbol{S}_{m,m}$ is a diagonal matrix of order $m \times m$ with the eigenvalues ordered from largest to smallest. These values are substituted in $\boldsymbol{Q}$ resulting in $\boldsymbol{u_n} \sim N\left(\boldsymbol{0}, \sigma_u^2 \boldsymbol{K}_{n,m}\boldsymbol{U}\boldsymbol{S}^{-1/2}\boldsymbol{S}^{-1/2}\boldsymbol{U}'\boldsymbol{K}'_{n,m}\right)$, and thus, thanks to the properties of the normal distribution, model (8.8) can be expressed like model (8.11) as

$$\boldsymbol{y} = \mu 1_n + \boldsymbol{Pf} + \boldsymbol{\varepsilon} \tag{8.12}$$

Model (8.12) is similar to model (8.11), except that $\boldsymbol{f}$ is a vector of order $m \times 1$ with a normal distribution of the form $\boldsymbol{f} \sim \boldsymbol{N}\left(\boldsymbol{0}, \sigma_f^2 \boldsymbol{I}_{m,m}\right)$, where $\boldsymbol{P} = \boldsymbol{K}_{m,\,n}\boldsymbol{U}\boldsymbol{S}^{-1/2}$ is now the design matrix. This implies estimating only $m$ effects that are projected into the $n$-dimensional space in order to predict $\boldsymbol{u_n}$ and explain $\boldsymbol{y}_n$. Note that model (8.12) has a Ridge regression solution, and thus available software for Bayesian Ridge regression like BGLR R or software for conventional Ridge regression like glmnet can be used for fitting model (8.12).

In summary, according to Cuevas et al. (2020), the approximation described above consists of the following steps:

Step 1. Computing the following matrices, matrix $\boldsymbol{K}_{m,m}$ from $m$ lines of the training set.
Step 2. Computing matrix $\boldsymbol{K}_{n,m}$.
Step 3. Eigenvalue decomposition of $\boldsymbol{K}_{m,m}$.
Step 4. Computing matrix $\boldsymbol{P} = \boldsymbol{K}_{n,m}\boldsymbol{U}\boldsymbol{S}^{-1/2}$.
Step 5. Fitting the model under a Ridge regression framework (like BGLR or glmnet) and making genomic-enabled predictions for future data.

With the following R code, the $\boldsymbol{P} = \boldsymbol{K}_{n,m}\boldsymbol{U}\boldsymbol{S}^{-1/2}$ (matrix design) can be computed under a linear kernel.

```
##################Linear approximate kernel#########################
Sparse_linear_kernel = function(m, X){
m = m
XF = X
p = ncol(XF)
pos_m = sample(1:nrow(XF),m)
####Step 1 compute K_m###############3
X_m = XF[pos_m,]
dim(X_m)
K_m = X_m% * %t(X_m)/p
dim(K_m)
######Step 2 compute K_n_m###########
K_n_m = XF% * %t(X_m)/p
dim(K_n_m)
######Step 3 compute eigenvalue decomposition of K_m######
EVD_K_m = eigen(K_m)
####Eigenvectors
U = EVD_K_m$vectors
```

```
###Eigenvalues###
S = EVD_K_m$values
####Square root of the inverse of eigenvalues#####
S_0.5_Inv = sqrt(1/S)
#####Diagonal matrix of square root of inverse of eigenvalues###
S_mat_Inv = diag(S_0.5_Inv)
#####Computing matrix P
P = K_n_m% * %U% * %S_mat_Inv
return(P)}
```

To use this function to create the design matrix $\boldsymbol{P} = \boldsymbol{K}_{n,m}\boldsymbol{U}\boldsymbol{S}^{-1/2}$, you need to provide the standardized matrix of markers $\boldsymbol{X}$, and the number of lines $m$, to be used for computing the approximate linear kernel. Then with this $\boldsymbol{P}$ you can implement the Ridge regression model under a Bayesian or conventional framework.

Table 8.11 indicates that the lower the value of $m$ for building the approximate kernel, the smaller the time required for its implementation in the four response variables (Env1, . . ., Env4). The table also shows that the lower the value of $m$, the worse the prediction performance in terms of MSE and PC. However, it is really interesting that with a reduction in the training set from 599 (all data) to 264 (only 44% of the total data set), the implementation time is reduced to almost half without any significant loss in terms of prediction performance. The complete R code to reproduce the results provided in Table 8.11 is given in Appendix 10.

The approximate kernel method can be used for any of the kernels studied before. The construction of the approximate Gaussian kernel matrix can be done with the following R function:

```
####################Gaussian kernel function#####################
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=XF,x2=XF, gamma=1/ncol(XF))

######################Approximate Guassian kernel#################
Sparse_Gaussian_kernel=function(m,X){
m=m
XF=X
p=ncol(XF)
pos_m=sample(1:nrow(XF),m)
####Step 1 compute K_m###############3
X_m=XF[pos_m,]
dim(X_m)
K_m=K.radial(x1=X_m,x2=X_m, gamma=1/p)

######Step 2 compute K_n_m###########
K_n_m=K.radial(x1=XF,x2=X_m, gamma=1/p)

############Step 3 compute eigenvalues decomposition of
K_m###########
```

**Table 8.11** Prediction performance in terms of mean square error (MSE) and average Pearson's correlation (PC) under the approximate linear kernel with the method proposed by Cuevas et al. (2020)

| | Env1 | | | Env2 | | | Env3 | | | Env4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $m$ | MSE | PC | Time | MSE | PC | Time | MSE | PC | Time | MSE | PC | Time |
| 15 | 0.914 | 0.293 | 12.410 | 0.904 | 0.312 | 12.430 | 0.941 | 0.248 | 12.830 | 0.905 | 0.310 | 12.860 |
| 32 | 0.852 | 0.372 | 15.860 | 0.865 | 0.369 | 14.480 | 0.910 | 0.303 | 15.220 | 0.877 | 0.351 | 14.760 |
| 74 | 0.804 | 0.435 | 18.960 | 0.810 | 0.431 | 20.790 | 0.873 | 0.361 | 20.000 | 0.824 | 0.423 | 20.720 |
| 132 | 0.740 | 0.494 | 33.050 | 0.785 | 0.458 | 27.150 | 0.869 | 0.369 | 26.550 | 0.798 | 0.453 | 26.740 |
| 264 | 0.716 | 0.523 | 53.250 | 0.749 | 0.497 | 43.720 | 0.852 | 0.393 | 42.090 | 0.794 | 0.457 | 42.630 |
| 599 | 0.713 | 0.524 | 98.800 | 0.741 | 0.506 | 86.470 | 0.844 | 0.405 | 86.290 | 0.785 | 0.468 | 87.220 |

Ten-fold cross-validation was implemented and the prediction performance is only reported for the testing set. Six values of training size $m$ were implemented: 15, 32, 74, 132, 264, and 599 (all data). Time reported: the implementation time in seconds. Data wheat599 was used

```
EVD_K_m=eigen(K_m)
####Eigenvectors
U=EVD_K_m$vectors
###Eigenvalues###
S=EVD_K_m$values
####Square root of the inverse of eigenvalues #####
S_0.5_Inv=sqrt(1/S)
#####Diagonal matrix of square root of inverse of eigenvalues###
S_mat_Inv=diag(S_0.5_Inv)
######Computing matrix P
P=K_n_m%*%U%*%S_mat_Inv
return(P)}
```

With this approximate kernel method, an equivalent to Table 8.11 was reproduced, but instead of using a linear kernel, a Gaussian kernel was used. The results for the same values of $m$ as those used in Table 8.11 ($m$ = 15, 32, 74, 132, 264, and 599), with the wheat599 data set for each of the four response variables are given in Table 8.12.

Again, we can see in Table 8.12 that the lower the training set ($m$) used for approximating the kernel, the lower the prediction performance, but the shorter the implementation time. However, it is very interesting to point out that with this data set, the approximation obtained when 44% (264 lines) of the original lines (599 lines) were used is quite good for the four response variables (E1,. . ., E4). However, the approximation to the full data set was slightly better under the approximate linear kernel (Table 8.11) than under the approximate Gaussian kernel (Table 8.12), but in general, the predictions obtained with the Gaussian kernel (full and approximated) were better than those obtained with the linear kernel (full and approximated). It is really important to point out that the R code given in Appendix 10 can be used for reproducing the results given in Table 8.12, but by replacing the function of the approximate linear kernel with the function for the approximate Gaussian kernel.

### 8.10.1 Extended Predictor Under the Approximate Kernel Method

Now we will illustrate the approximate kernel using the expanded predictor (8.10) that, in addition to the main effect of lines, contains the main effects of environments and the interaction term between environments and lines. Therefore, the approximate method is similar to the case of a single environment, that is, $\boldsymbol{u}_1 \sim N\left(\mathbf{0}, \sigma^2_{u_1}\boldsymbol{Q}^{u_1}\right)$, where $\boldsymbol{K}_1 \approx \boldsymbol{Q}^g = \boldsymbol{Z}_{u1}\left(\boldsymbol{K}_{n,m}\boldsymbol{K}^{-1}_{m,m}\boldsymbol{K}^T_{n,m}\right)\boldsymbol{Z}^T_{u1}$, whereas for the random interaction $\boldsymbol{u}_2 \sim N\left(\mathbf{0}, \sigma^2_{u_2}\boldsymbol{Q}^{u_2}\right)$, where $\boldsymbol{K}_2 \approx \boldsymbol{Q}^{u_2} = \left[\boldsymbol{Z}_{u1}\left(\boldsymbol{K}_{n,m}\boldsymbol{K}^{-1}_{m,m}\boldsymbol{K}^T_{n,m}\right)\boldsymbol{Z}^T_{u1}\right]^\circ\left[\boldsymbol{Z}_E\boldsymbol{Z}_E{}^T\right]$.

Also, we decomposed $\boldsymbol{K}^{-1}_{m,m}$ in such a way that model (8.10) could be approximated as

**Table 8.12** Prediction performance in terms of mean square error (MSE) and average Pearson's correlation (PC) under the approximate Gaussian kernel with the method proposed by Cuevas et al. (2020) using the wheat599 data set

| | | Env1 | | | Env2 | | | Env3 | | | Env4 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $m$ | MSE | PC | Time | MSE | PC | Time | MSE | PC | Time | MSE | PC | Time |
| 15 | 0.927 | 0.301 | 12.780 | 0.955 | 0.241 | 12.710 | 0.988 | 0.227 | 11.120 | 0.959 | 0.213 | 12.700 |
| 32 | 0.845 | 0.389 | 14.270 | 0.911 | 0.313 | 16.230 | 0.937 | 0.254 | 12.680 | 0.931 | 0.292 | 14.030 |
| 74 | 0.802 | 0.448 | 20.970 | 0.827 | 0.412 | 21.310 | 0.868 | 0.365 | 17.050 | 0.800 | 0.453 | 20.380 |
| 132 | 0.714 | 0.533 | 30.200 | 0.816 | 0.435 | 29.680 | 0.865 | 0.373 | 24.770 | 0.775 | 0.476 | 25.250 |
| 264 | 0.677 | 0.570 | 51.140 | 0.769 | 0.483 | 48.520 | 0.854 | 0.387 | 39.280 | 0.745 | 0.512 | 38.530 |
| 599 | 0.645 | 0.595 | 104.250 | 0.754 | 0.501 | 84.890 | 0.803 | 0.454 | 80.720 | 0.721 | 0.538 | 81.990 |

Ten-fold cross-validation was implemented and the prediction performance is only reported for the testing set. Six values of training size $m$ were implemented: 15, 32, 74, 132, 264, and 599 (all data). Time reported: the implementation time in seconds

$$\boldsymbol{y} = \mu\mathbf{1} + \boldsymbol{Z}_E\boldsymbol{\beta}_E + \boldsymbol{P}^{u_1}\boldsymbol{f} + \boldsymbol{P}^{u_2}\boldsymbol{l} + \boldsymbol{\varepsilon}, \tag{8.13}$$

where $\boldsymbol{P}^{u_1} = \boldsymbol{Z}_{u1}\boldsymbol{P} = \boldsymbol{Z}_{u1}\boldsymbol{K}_{n,m}\boldsymbol{US}^{-1/2}$ of order $n^* \times m$, with $n^* = n_1 + n_2 + \ldots + n_I$, with $\boldsymbol{f}$ a vector of $m \times 1$; $\boldsymbol{P}^{u_2} = \boldsymbol{P}^{u_1} : \boldsymbol{Z}_E$ of order $n^* \times mI$ and the vector $\boldsymbol{l}$ is of order $mI \times 1$, and the notation $\boldsymbol{P}^{u_1} : \boldsymbol{Z}_E$ denotes the interaction term between the design matrix $\boldsymbol{P}^{u_1}$ and $\boldsymbol{Z}_E$.

In summary, the suggested approximate method described above can be summarized as

Step 1. We assume that we have a matrix of markers **X** that contains the lines without replication, that is, each row corresponds to a different line. We assume that this matrix contains $L$ lines (rows) and $p$ markers (columns). Also, it is important to point out that this matrix is standardized by columns.
Step 2: We randomly select $m$ lines out of $L$ from the training set **X.**
Step 3. Next we construct matrices $\boldsymbol{K}_{m,m}$ and $\boldsymbol{K}_{L,m}$, from the matrix of markers as $\boldsymbol{K}_{m,m} = \frac{\boldsymbol{X}_{m,p}\boldsymbol{X}'_{m,p}}{p}$, $\boldsymbol{K}_{L,m} = \frac{\boldsymbol{X}_{L,p}\boldsymbol{X}'_{m,p}}{p}$
Step 4. Eigenvalue decomposition of $\boldsymbol{K}_{m,m}$.
Step 5. Computing matrix $\boldsymbol{P} = \boldsymbol{K}_{n,m}\boldsymbol{US}^{-1/2}$.
Step 6. Computing matrix $\boldsymbol{P}^{u_1} = \boldsymbol{Z}_{u1}\,\boldsymbol{P} = \boldsymbol{Z}_{u1}\boldsymbol{K}_{n,m}\boldsymbol{US}^{-1/2}$.
Step 7. Computing matrix $\boldsymbol{P}^{u_2} = \boldsymbol{P}^{u_1} : \boldsymbol{Z}_E$, where : denotes the interaction between the design matrix $\boldsymbol{P}^{u_1}$ and $\boldsymbol{Z}_E$.
Step 8. Fitting the model under a Ridge regression framework (like BGLR or glmnet) and making genomic-enabled predictions for future data.

It is important to point out that the extension of the approximate kernel method for an extended predictor requires that some lines were studied in some environments but not in all environments. This extended approximate kernel method is expected to be efficient when the number of environments is low and the number of lines is large.

To illustrate the extended approximate kernel method, we used the Data_Toy_EYT that contains four environments, four traits, and 40 observations in each environment. Here we only used the continuous trait (GY) as the response variable. The approximate kernels were built using only the lines (40 lines) from which the training set was built with $m = 4$, 8, 12, 16, 20, and 40 lines. Now instead of using only one kernel, we implemented five (linear, polynomial, sigmoid, Gaussian, and exponential). The R code for reproducing the results in Table 8.13 is given in Appendix 11.

We can see in Table 8.13 that there are differences in the prediction performance using different kernels. However, the approximate kernel, even with $m = 4$, many times outperformed the prediction performance of the exact kernel ($m = 40$), which implies that when the lines are quite correlated even with a small sample size $m$, approximating the kernel is enough to get good prediction performance. But the time gained using a sample size $m$, that is less than $n$ total number of lines, significantly reduces the implementation time, and this gain in implementation time is more relevant when the number of lines is very large, as was stated by Cuevas et al. (2020).

**Table 8.13** Prediction performance in terms of mean square error (MSE) and average Pearson's correlation (PC) under five approximate Gaussian kernel methods with the method proposed by Cuevas et al. (2020)

| Metrics | Kernel | $m = 4$ | $m = 8$ | $m = 12$ | $m = 16$ | $m = 20$ | $m = 40$ |
|---|---|---|---|---|---|---|---|
| MSE | Linear | 0.306 | 0.325 | 0.367 | 0.329 | 0.327 | 0.325 |
| PC | Linear | 0.939 | 0.936 | 0.926 | 0.935 | 0.935 | 0.936 |
| Time | Linear | 12.300 | 17.780 | 17.090 | 17.340 | 19.330 | 27.240 |
| MSE | Polynomial | 0.330 | 0.343 | 0.356 | 0.308 | 0.325 | 0.342 |
| PC | Polynomial | 0.934 | 0.931 | 0.927 | 0.939 | 0.935 | 0.933 |
| Time | Polynomial | 17.000 | 17.030 | 18.280 | 20.470 | 21.180 | 25.750 |
| MSE | Sigmoid | 0.383 | 0.295 | 0.335 | 0.314 | 0.328 | 0.324 |
| PC | Sigmoid | 0.923 | 0.941 | 0.934 | 0.937 | 0.935 | 0.936 |
| Time | Sigmoid | 19.400 | 18.550 | 17.170 | 19.770 | 20.450 | 23.550 |
| MSE | Gaussian | 0.325 | 0.337 | 0.351 | 0.335 | 0.331 | 0.340 |
| PC | Gaussian | 0.936 | 0.932 | 0.929 | 0.934 | 0.935 | 0.933 |
| Time | Gaussian | 18.530 | 19.300 | 20.090 | 18.730 | 22.360 | 30.630 |
| MSE | Exponential | 0.333 | 0.320 | 0.339 | 0.381 | 0.363 | 0.348 |
| PC | Exponential | 0.932 | 0.936 | 0.932 | 0.924 | 0.927 | 0.932 |
| Time | Exponential | 17.220 | 18.860 | 20.090 | 21.730 | 19.200 | 24.970 |

Ten fold cross-validation was implemented and the prediction performance is only reported for the testing set. Six values of training size $m$ were implemented: 4, 8, 12, 16, 20, and 40 (all data). Time reported: the implementation time in seconds. The data set used for this example was Data_Toy_EYT with trait GY as the response variable that was used before in Table 8.10

## 8.11 Final Comments

In the context of genomic prediction, arguably, genotypes and phenotypes may be linked in functional forms that are not well addressed by the linear additive models that are standard in quantitative genetics. Therefore, developing statistical machine learning models for predicting phenotypic values from all available molecular information and that are capable of capturing complex genetic network architectures is of great importance. Kernel Ridge regression methods are nonparametric prediction models proposed for this purpose. Their essence is to create a spatial distance-based relationship matrix called a kernel (Morota et al. 2013). For this reason, many kernel functions have been developed to capture complex nonlinear patterns that many times outperform conventional linear regression models in terms of prediction performance.

The kernel trick allows you to build nonlinear versions of any linear algorithms by replacing their independent variables (predictors) with a kernel function, giving them greater advantages.

1. The kernels are interpreted as scalar products in high-dimensional spaces.
2. There are kernels with great versatility and composite kernels can be built; some can be computed in closed form, while others require an iterative process.

3. The number of dimensions increases the complexity, and with it the risk of overfitting.
4. Kernel methods are an alternative to least square methods algorithms that control complexity through regularization.
5. Kernel methods guarantee existence and uniqueness, just like least square methods.
6. Nonlinear versions can be made using the kernel trick, obtaining statistical machines with great expressive capacity, but with training control.
7. Kernel statistical machine learning methods provide promising tools for large-scale and high-dimensional genomic data processing.
8. Kernel methods can also be viewed from a regression perspective and can be integrated with classical methods for gene prioritizing, prediction, and data fusion.
9. Kernel methods allow you to further improve the scalability of conventional machine learning methods and their versatility to work with heterogeneous inputs.
10. Kernel methods are remarkably flexible and elegant, as they are the predictive principle underlying most linear mixed models commonly used in plant breeding, as well as others used in spatial analysis of classification problems.
11. Kernel methods exploit complexity to improve prediction accuracy, but do not help very much to increase the understanding of the complexity.
12. Kernels based on data compression ideas are very promising for dealing with very large data sets, but software is needed, as well as more research to develop new methods and improve the existing methods.

## Appendix 1

R code for manually implementing nine kernels (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4) with a continuous response variable; the results are provided in Table 8.1.

```
rm(list=ls(all=TRUE))
library(plyr)
library(tidyr)
library(dplyr)
library(glmnet)
library(BGLR)
data(wheat) #Loads the wheat data set
y=wheat.Y
dim(y)
head(y)

XF=scale(wheat.X)
dim(XL)
head(XF[,1:5])
```

```
#######Linear Kernel###########
K.linear=function(x1, x2=x1,gamma=1) {gamma*(as.matrix(x1)%*%t(as.
matrix(x2))) }
K.lin=K.linear(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.lin)
X.lin=t(chol(K.lin))

#######Polynomial Kernel###########
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.poly)
X.poly=t(chol(K.poly))

#######Sigmoid Kernel###########
K.sigmoid=function(x1,x2=x1, gamma=1/ncol(XF), b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=XF,x2=XF)
dim(K.sig)
ei=eigen(K.sig)
pos_neg=which(ei$values<0)
Eigenv=ei$value
Eigenv[pos_neg]=0
Eigenv
X.sig=ei$vectors%*%diag(sqrt(Eigenv))%*%t(ei$vectors)

#######Radial Kernel###########
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.rad)
X.rad=t(chol(K.rad))

#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.exp)
X.exp=t(chol(K.exp))

###########Arc-cosine kernel with L=1#######
K.AC1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
```

```
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}

####Arc-cosine kernel with L>1########
AK1<-K.AC1(XF)
X.AK1=t(chol(AK1))
K.AC.L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC
 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
 }
  GC<-GC/median(GC)
 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}
AK1=AK1
AK2<-K.AC.L(GC=AK1,nl=2)
X.AK2=t(chol(AK2))
AK3<-K.AC.L(GC=AK1,nl=3)
X.AK3=t(chol(AK3))
AK4<-K.AC.L(GC=AK1,nl=4)
.AK4=t(chol(AK4))

###########Cholesky of each kernel#######
X_ker=list(X.lin=X.lin, X.poly=X.poly,X.sig=X.sig,X.rad=X.rad,X.
exp=X.exp,X.AK1=X.AK1, X.AK2=X.AK2,X.AK3=X.AK3,X.AK4=X.AK4)
kernel.name=c
("Linear","Polynomial","Sigmoid","Gaussian","Exponential", "AK1",
"AK2","AK3","AK4")

#####K-fold cross-validation
n=nrow(XF)
No.folds=10
set.seed(10)
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)

Y=y
Env_name=colnames(y)
```

```
rownames(Y)=1:n
Pred_all_kelnels<-data.frame()
for (t in 1:4){
y2=Y[,t]

results<-c() #save cross-validation results
for (k in 1:9){
MSE_Part=c()
for(r in 1:No.folds) {
#r=1
Xstar=X_ker[[k]]
rownames(Xstar)=1:n
X2=Xstar
actual_CV=r
y1=y2

positionTST=which(Grpv==r)

y_tr = y1[-positionTST] ; X_tr = X2[-positionTST,];
y_tst = y1[positionTST]; X_tst = X2[positionTST,]

A_RR = cv.glmnet(X_tr,y_tr,family='gaussian',
        alpha=0,type.measure='mse')
ypred= as.numeric(predict(A_RR,newx=X_tst,s='lambda.min',
type='class'))
Predicted=ypred
Observed=as.numeric(y_tst)

MSE=mean((Predicted-Observed)^2)
MSE_Part=c(MSE_Part,MSE)
}
MSE_Part
mean(MSE_Part)

results=c(results,mean(MSE_Part))
}
results1=t(results)
colnames(results1)=kernel.name
Pred_all_kelnels=rbind(Pred_all_kelnels,data.frame(Env=Env_name
[t],results1))
}
Pred_all_kelnels
write.csv(Pred_all_kelnels, file ="Kernel_Example_8.1v2.csv")
```

## Appendix 2

R code for manually implementing nine kernels (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4) with a binary response variable; the results are provided in Table 8.2.

```
rm(list=ls(all=TRUE))
library(plyr)
library(tidyr)
library(dplyr)
library(glmnet)

load("Data_Toy_EYT.RData")
ls()
Pheno=Pheno_Toy_EYT
dim(G_Toy_EYT)
XF=t(chol(G_Toy_EYT))
dim(XF)
head(XF[,1:5])

#######Linear Kernel###########
K.linear=function(x1, x2=x1,gamma=1) {gamma*(as.matrix(x1)%*%t(as.
matrix(x2))) }
K.lin=K.linear(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.lin)
X.lin=t(chol(K.lin))

#######Polynomial Kernel###########
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.poly)
X.poly=t(chol(K.poly))

#######Sigmoid Kernel###########
K.sigmoid=function(x1,x2=x1, gamma=1/ncol(XF), b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=XF,x2=XF, gamma=1/1*ncol(XF))
dim(K.sig)
ei=eigen(K.sig)
pos_neg=which(ei$values<0)
Eigenv=ei$value
Eigenv[pos_neg]=0
Eigenv
X.sig=ei$vectors%*%diag(sqrt(Eigenv))%*%t(ei$vectors)

#######Radial Kernel###########
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.rad)
X.rad=t(chol(K.rad))

#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
```

```
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.exp)
X.exp=t(chol(K.exp))

############Arc-cosine kernel with L=1#######
K.AC1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)
 return(GC1)
}

####Arc-cosine kernel with L>1########
AK1<-K.AC1(XF)
X.AK1=t(chol(AK1))
K.AC.L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC
 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)

  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
  }

 GC<-GC/median(GC)
 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}
AK1=AK1
AK2<-K.AC.L(GC=AK1,nl=2)
X.AK2=t(chol(AK2))
AK3<-K.AC.L(GC=AK1,nl=3)
X.AK3=t(chol(AK3))
AK4<-K.AC.L(GC=AK1,nl=4)
X.AK4=t(chol(AK4))
```

```
########Design matrices###
ZE=model.matrix(~0+as.factor(Pheno$Env))
ZL=model.matrix(~0+as.factor(Pheno$GID))
cbind(colnames(ZL), colnames(G_Toy_EYT))
###########Cholesky of each kernel#######
X_ker=list(X.lin=cbind(ZE,ZL%*%X.lin), X.poly=cbind(ZE,ZL%*%X.
poly),X.sig=cbind(ZE,ZL%*%X.sig),X.rad=cbind(ZE,ZL%*%X.rad),X.
exp=cbind(ZE,ZL%*%X.exp),X.AK1=cbind(ZE,ZL%*%X.AK1), X.AK2=cbind
(ZE,ZL%*%X.AK2),X.AK3=cbind(ZE,ZL%*%X.AK3),X.AK4=cbind(ZE,ZL%*%X.
AK4))
kernel.name=c
("Linear","Polynomial","Sigmoid","Gaussian","Exponential", "AK1",
"AK2","AK3","AK4")

#####K-fold cross-validation
n=nrow(Pheno)
No.folds=10
set.seed(10)
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)

########Response variable#################################
y2=Pheno$Height
results<-data.frame() #save cross-validation results
for(r in 1:No.folds) {
y1=y2
positionTST=which(Grpv==r)
PCCC_Part=c()
for (k in 1:9){
Xstar=X_ker[[k]]
rownames(Xstar)=1:n
X2=Xstar
y_tr = y1[-positionTST] ; X_tr = X2[-positionTST,];
y_tst = y1[positionTST]; X_tst = X2[positionTST,]

A_RR = cv.glmnet(X_tr,y_tr,family='binomial',
         alpha=0,type.measure='class')
ypred= as.numeric(predict(A_RR,newx=X_tst,s='lambda.min',
type='class'))
Predicted=ypred
Observed=as.numeric(y_tst)

PCCC=1-mean(Predicted!=Observed)
PCCC_Part=c(PCCC_Part,PCCC)
}
names(PCCC_Part)=kernel.name
results=rbind(results,data.frame(fold=r,t(PCCC_Part)))
}
results
apply(results[,-1],2,mean)
write.csv(results, file ="Kernel_Binary_Example_Table_8.2.csv")
```

# Appendix 3

R code for implementing three default kernels in rrBLUP (linear, Gaussian, and Exponential), and the results are provided in Table 8.4.

```
library(rrBLUP) #load rrBLUP
library(BGLR) #load BLR
data(wheat) #load wheat data
X=wheat.X
dim(X)
X <- 2*X-1 #recode genotypes
X=scale(X)
t=4
y <-wheat.Y[,t] #yields from E1
yy=y
MSE=function(yobserved,ypredicted){
 MSE=mean((yobserved-ypredicted)^2)
}
n_records=nrow(X)
n_folds=10
set.seed(10)
sets <- findInterval(cut(sample(1:n_records, n_records),
                 breaks=n_folds), 1:n_records)
results=data.frame()
for (i in 1:n_folds){
# i=1
trn <- which(sets!=i)
tst <- which(sets==i)
ans.RR<-kinship.BLUP(y=y[trn],
          G.train=X[trn,],G.pred=X[tst,])
#accuracy with RR
Cor_RR=cor(ans.RR$g.pred,yy[tst])
MSE_RR=MSE(ans.RR$g.pred,yy[tst])

ans.GAUSS<-kinship.BLUP(y=y[trn],
           G.train=X[trn,],G.pred=X[tst,],
           K.method="GAUSS")
#accuracy with GAUSS
Cor_GAUSS=cor(ans.GAUSS$g.pred,yy[tst])
MSE_GAUSS=MSE(ans.GAUSS$g.pred,yy[tst])

ans.EXP<-kinship.BLUP(y=y[trn],
           G.train=X[trn,],G.pred=X[tst,],
           K.method="EXP")
#accuracy with EXponential
Cor_EXP=cor(ans.EXP$g.pred,yy[tst])
MSE_EXP=MSE(ans.EXP$g.pred,yy[tst])

results=rbind(results,data.frame(Fold=i, MSE_RR=MSE_RR,
MSE_GAUSS=MSE_GAUSS, MSE_EXP=MSE_EXP,Cor_RR=Cor_RR,
Cor_GAUSS=Cor_GAUSS, Cor_EXP=Cor_EXP))
```

```
}
results

write.csv(results,file="Kernel_Mixed_Example_Table_8.4.csv")
```

## Appendix 4

R code for manually implementing nine kernels (linear, polynomial, sigmoid, Gaussian, exponential, AK1, AK2, AK3, and AK4); the results are provided in Table 8.5.

```
library(rrBLUP) #load rrBLUP
library(BGLR) #load BLR
data(wheat) #load wheat data
X=wheat.X
dim(X)
X <- 2*X-1 #recode genotypes

XF=scale(wheat.X)
dim(XF)
head(XF[,1:5])

#######Linear Kernel###########
K.linear=function(x1, x2=x1,gamma=1) {gamma*(as.matrix(x1)%*%t(as.
matrix(x2))) }
K.lin=K.linear(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.lin)

#######Polynomial Kernel###########
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=XF,x2=XF,gamma=1/ncol(XF))
dim(K.poly)

#######Sigmoid Kernel###########
K.sigmoid=function(x1,x2=x1, gamma=1/ncol(XF), b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=XF,x2=XF)
ei=eigen(K.sig)
pos_neg=which(ei$values<0)
Eigenv=ei$value
Eigenv[pos_neg]=0
Eigenv
K.sig=ei$vectors%*%diag((Eigenv))%*%t(ei$vectors)
#######Radial Kernel###########
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
```

```
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.rad)

#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=XF,x2=XF, gamma=1/ncol(XF))
dim(K.exp)
###########Arc-cosine kernel with L=1#######
K.AC1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}

####Arc-cosine kernel with L>1########
AK1<-K.AC1(XF)
K.AC.L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
  }
 GC<-GC/median(GC)
 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}
AK1=AK1
AK2<-K.AC.L(GC=AK1,nl=2)
AK3<-K.AC.L(GC=AK1,nl=3)
AK4<-K.AC.L(GC=AK1,nl=4)
```

```
###########Cholesky of each kernel#######
K_ker=list(K.lin=K.lin, K.poly=K.poly,K.sig=K.sig,K.rad=K.rad,K.
exp=K.exp,K.AK1=AK1, K.AK2=AK2,K.AK3=AK3,K.AK4=AK4)
kernel.name=c
("Linear","Polynomial","Sigmoid","Gaussian","Exponential", "AK1",
"AK2","AK3","AK4")

#####K-fold cross-validation
n=nrow(XF)
No.folds=10
set.seed(10)
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)
Env_name=colnames(wheat.Y)
t=4
 y2=wheat.Y[,t]
 results_ker=data.frame() #save cross-validation results

  for (k in 1:9){
  Kstar=K_ker[[k]]
  rownames(Kstar)=1:n
  results=data.frame()
  for(r in 1:No.folds) {
   y1=y2
   positionTST=which(Grpv==r)
   y1[positionTST]=NA
    I=diag(n)
    fit_Mix <- mixed.solve(y=y1,Z=I,K=Kstar)
    Pred_y=c(fit_Mix$beta)*rep(1,nrow(Kstar))+c(fit_Mix$u)
    MSE=mean((Pred_y[positionTST]-y2[positionTST])^2)
    results=rbind(results,data.frame(MSE=MSE))
   }
  results
   results_ker=rbind(results_ker,data.frame(Kernel=kernel.name[k],t
(results)))
   }
  results_ker
  colnames(results_ker)=c("Kernel",
"1","2","3","4","5","6","7","8","9","10")
  results_ker
write.csv(results_ker, file ="Example_Mixed_Table_8.5.csv")
```

## Appendix 5

R code for tuning the number of hidden layers in the arc-cosine kernel (Fig. 8.2).

```
library(rrBLUP) #load rrBLUP
library(BGLR) #load BLR
data(wheat) #load wheat data
X=wheat.X
```

```
dim(X)
X <- 2*X-1 #recode genotypes
XF=scale(wheat.X)
rownames(XF)=1:599
dim(XF)
head(XF[,1:5])

############Arc-cosine kernel with L=1#######
K.AC1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)
 return(GC1)
}

####Arc-cosine kernel with L>1########
AK1<-K.AC1(XF)
K.AC.L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){

  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1

  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
  }
 GC<-GC/median(GC)
 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}
AK1=AK1
AK2<-K.AC.L(GC=AK1,nl=2)
dim(AK2)

#####K-fold cross-validation
n=nrow(XF)
No.folds=4
set.seed(10)
```

```
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)
Env_name=colnames(wheat.Y)

t=4
rownames(wheat.Y)=1:599
y2=wheat.Y[,t]
results=data.frame()
for(r in 1:No.folds) {
#  r=4
  y1=y2
  positionTST=which(Grpv==r)
  positionTRN=which(Grpv!=r)
  n_inner=length(positionTRN)
  ICV= findInterval(cut(sample(1:length(positionTRN),n_inner),
breaks=No.folds),1:n_inner)
Lvec=1:10
Ave_MSE_L=c()
for (L in 1:length(Lvec)){
  results_Inner=c()
for (j in 1:No.folds){
  y1_trn=y1[positionTRN]
  y1_trnI=y1_trn
  AKL<-K.AC.L(GC=AK1,nl=L)
  Kstar_inner=AKL[positionTRN,positionTRN]
  pos_TST=which(ICV==j)
  y1_trnI[pos_TST]=NA
  length(y1_trn)
  I=diag(length(y1_trn))
  fit_MixI <- mixed.solve(y=y1_trnI,Z=I,K=Kstar_inner)
  Pred_yI=c(fit_MixI$beta)*rep(1,nrow(Kstar_inner))+c(fit_MixI$u)
  MSE_Inner=mean((Pred_yI[pos_TST]-y1_trn[pos_TST])^2)
  results_Inner=c(results_Inner,MSE_Inner)
  }
MSE_L=mean(results_Inner)
MSE_L
Ave_MSE_L=c(Ave_MSE_L,MSE_L)
opt_mse=which.min(Ave_MSE_L)
L_opt=Lvec[opt_mse]
}

AKL_opt<-K.AC.L(GC=AK1,nl=L_opt )
Kstar=AKL_opt
y1[positionTST]=NA
I=diag(n)
   fit_Mix <- mixed.solve(y=y1,Z=I,K=Kstar)
   Pred_y=c(fit_Mix$beta)*rep(1,nrow(Kstar))+c(fit_Mix$u)
   MSE=mean((Pred_y[positionTST]-y2[positionTST])^2)
   results=rbind(results,data.frame(MSE=MSE))}
 results
apply(results,2,mean)
par(mar=c(4,6,6,4))
plot(1:10,Ave_MSE_L, ylab="Mean Square Error",xlab="Number of hidden
layers",lwd=12, cex.axis =1.5, cex.lab = 2)
```

# Appendix 6

R code for implementing nine kernels under a Bayesian kernel BLUP approach with only genotypic effects in the predictor (Table 8.6).

```
rm(list=ls())
library(BGLR)
load('dat_ls_E1.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
#Marker data
dat_M = dat_ls$dat_M
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F,5)
dim(dat_F)

#Matrix design of markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
XM = scale(XM)
dim(XM)

n = dim(dat_F)[1]
y = dat_F$y

#10 random partitions
K = 10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)

#GBLUP=Linear kernel
dat_M = scale(dat_M)
G = tcrossprod(dat_M)/dim(dat_M)[2]
dim(G)
#Matrix design of GIDs
Z = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
Ga = Z%*%G%*%t(Z)
ETA_GB = list(list(model='RKHS',K = Ga))
#Tab = data.frame(PT = 1:K,MSEP = NA)
set.seed(1)
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
```

```
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_GB,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_GB[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_GB)
sd(Tab$MSEP_GB)

#######Polynomial Kernel###########
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
dim(K.poly)
K.poly.Exp= Z%*%K.poly%*%t(Z)
ETA_K.poly = list(list(model='RKHS',K=K.poly.Exp))

for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.poly,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_Poly[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_Poly)
sd(Tab$MSEP_Poly)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Sigmoid Kernel###########
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
#K.sig=K.sig+diag(599)*0.1

K.sig.Exp= Z%*%K.sig%*%t(Z)
ETA_K.sig = list(list(model='RKHS',K=K.sig.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.sig,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_sig[k] = mean((y[Pos_tst]-yp_ts)^2)
}
```

```
mean(Tab$MSEP_sig)
sd(Tab$MSEP_sig)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Gaussian or Radial Kernel###########
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.rad)

K.Gauss.Exp= Z%*%K.rad%*%t(Z)
ETA_K.Gauss = list(list(model='RKHS',K=K.Gauss.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.Gauss,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_Gauss[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_Gauss)
sd(Tab$MSEP_Gauss)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.exp)

K.Expo.Exp= Z%*%K.exp%*%t(Z)
ETA_K.Expo = list(list(model='RKHS',K=K.Expo.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.Expo,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_Expo[k] = mean((y[Pos_tst]-yp_ts)^2)
}
```

```
mean(Tab$MSEP_Expo)
sd(Tab$MSEP_Expo)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

############Arc-cosine kernel with deep=1#######
K.AK1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}

AK1<-K.AK1(dat_M)

K.AK1.Exp= Z%*%AK1%*%t(Z)
ETA_K.AK1 = list(list(model='RKHS',K=K.AK1.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.AK1,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_AK1[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_AK1)
sd(Tab$MSEP_AK1)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

####Arc-cosine kernel with deep=2#####
AK_L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){

  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
```

```
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)

  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))

 }

 GC<-GC/median(GC)

 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}

AK2<-AK_L(GC=AK1,nl=2)
K.AK2.Exp= Z%*%AK2%*%t(Z)
ETA_K.AK2 = list(list(model='RKHS',K=K.AK2.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.AK2,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_AK2[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_AK2)
sd(Tab$MSEP_AK2)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Arc-cosine kernel with deep=3
AK3<-AK_L(GC=AK1,nl=3)
K.AK3.Exp= Z%*%AK3%*%t(Z)
ETA_K.AK3 = list(list(model='RKHS',K=K.AK3.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.AK3,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_AK3[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_AK3)
sd(Tab$MSEP_AK3)
```

```
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

#######Arc-cosine kernel with deep=4
AK4<-AK_L(GC=AK1,nl=4)
K.AK4.Exp= Z%*%AK4%*%t(Z)
ETA_K.AK4 = list(list(model='RKHS',K=K.AK4.Exp))
for(k in 1:K)
{
 Pos_tst = PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.AK4,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
 yp_ts = A$yHat[Pos_tst]
 Tab$MSEP_AK4[k] = mean((y[Pos_tst]-yp_ts)^2)
}

mean(Tab$MSEP_AK4)
sd(Tab$MSEP_AK4)
cor(y[Pos_tst],yp_ts)
plot(y[Pos_tst],yp_ts);abline(a=0,b=1)

write.csv(Tab,file='Tab_MSEP-Ex1-Kernels.csv',row.names = FALSE)
```

# Appendix 7

R code for implementing seven kernels under a Bayesian kernel BLUP approach with effects of environment + genotype + genotype ×environment interaction in the predictor (Table 8.7).

```
rm(list=ls())
library(BGLR)
library(BMTME)
load('dat_ls_E2.RData',verbose=TRUE)
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
dim(dat_F)
#Marker data
dat_M = dat_ls$dat_M
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F)

head(dat_F,5)
#Matrix design for markers
```

```
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
dim(XM)
XM = scale(XM)
#Environment design matrix
XE = model.matrix(~0+Env,data=dat_F)[,-1]
K.E=XE%*%t(XE)
dim(K.E)

#GID design matrix and Environment-GID design matrix
#for RKHS models
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))

n=dim(dat_F)[1]
y=dat_F$y

#Number of random partitions
K=10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

########Linear kernel=GBLUP
dat_M=scale(dat_M)
G=tcrossprod(dat_M)/dim(dat_M)[2]
dim(G)
#Covariance matrix for Zg
K_L=Z_L%*%G%*%t(Z_L)
#Covariance matrix for random effects ZEg
K_LE= K.E*K_L

ETA_K.Linear=list(list(model='FIXED',X=XE),list(model='RKHS',
K=K_L),
        list(model='RKHS',K=K_LE))

#######Polynomial Kernel############
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
dim(K.poly)
K.poly.Exp=Z_L%*%K.poly%*%t(Z_L)

#Covariance matrix for random effects ZEg
K.GE.poly= K.E*K.poly.Exp
ETA_K.poly = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
poly.Exp),
          list(model='RKHS',K=K.GE.poly))

#######Sigmoid Kernel############
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
```

```
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))

K.sig.Exp= Z_L%*%K.sig%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.sig= K.E*K.sig.Exp
ETA_K.sig = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
sig.Exp),
          list(model='RKHS',K=K.GE.sig))

########Gaussian or Radial Kernel############
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.rad)

K.Gauss.Exp= Z_L%*%K.rad%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.Gauss=K.E*K.Gauss.Exp
ETA_K.Gauss=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Gauss.Exp),
         list(model='RKHS',K=K.GE.Gauss))

########Exponential Kernel############
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.exp)
K.Expo.Exp= Z_L%*%K.exp%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.Exp=K.E*K.Expo.Exp
ETA_K.Exp=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Expo.Exp),
        list(model='RKHS',K=K.GE.Exp))
############Arc-cosine kernel with deep=1#######
K.AK1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}
```

```
AK1<-K.AK1(dat_M)

K.AK1.Exp= Z_L%*%AK1%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.AK1=K.E*K.AK1.Exp
ETA_K.AK1=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK1.Exp),
        list(model='RKHS',K=K.GE.AK1))
####Arc-cosine kernel with deep=4#####
AK_L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
 }
  GC<-GC/median(GC)

 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}

AK4<-AK_L(GC=AK1,nl=4)
K.AK4.Exp= Z_L%*%AK4%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.AK4=K.E*K.AK4.Exp
ETA_K.AK4=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK4.Exp),
        list(model='RKHS',K=K.GE.AK4))

Tab1_m = data.frame(PT = 1:K,MSEP = NA)
Tab1_MSE = data.frame(PT = 1:K,MSEP = NA)
Tab1_Cor = data.frame(PT = 1:K,MSEP = NA)
for(k in 1:K)
{
 set.seed(1)
 Pos_tst =PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.Linear,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$Linear[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$Linear[k] = cor(y[Pos_tst],yp_ts)
```

```
  A = BGLR(y=y_NA,ETA=ETA_K.poly,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$poly[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$poly[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.sig,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$sig[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$sig[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.Gauss,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$Gauss[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$Gauss[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.Exp,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$Exp[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$Exp[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.AK1,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$AK1[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$AK1[k]  = cor(y[Pos_tst],yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.AK4,nIter = 1e4,burnIn = 1e3,verbose =
FALSE)
  yp_ts = A$yHat[Pos_tst]
  Tab1_MSE$AK4[k] = mean((y[Pos_tst]-yp_ts)^2)
  Tab1_Cor$AK4[k]  = cor(y[Pos_tst],yp_ts)
 }
Tab1_MSE
apply(Tab1_MSE[,-c(1:2)],2,mean)
write.csv(Tab1_MSE,file="Tab_MSEP.Ex2_kernels_New.csv")
```

## Appendix 8

R code for implementing seven kernels under a Bayesian kernel BLUP with a binary response variable with effects of environment + genotype + genotype ×environment interaction in the predictor (Table 8.8).

```
rm(list=ls())
library(BGLR)
```

```
library(BMTME)
load('Data_Toy_EYT.RData',verbose=TRUE)
ls()
#Phenotypic data
dat_F =Pheno_Toy_EYT
head(dat_F)
dim(dat_F)
#Marker data
dat_M =t(chol(G_Toy_EYT))
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F)

head(dat_F,5)
#Matrix design for markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]

XM =XM
#Environment design matrix
XE = model.matrix(~0+Env,data=dat_F)[,-1]
K.E=XE%*%t(XE)
dim(K.E)

#GID design matrix for lines
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
n=dim(dat_F)[1]
y=dat_F$Height

#Number of random partitions
K=10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))

########Linear kernel=GBLUP
G=G_Toy_EYT
dim(G)
#Covariance matrix for lines
K_L=Z_L%*%G%*%t(Z_L)
#Covariance matrix for GE term
K_LE= K.E*K_L
##########Predictor
ETA_K.Linear=list(list(model='FIXED',X=XE),list(model='RKHS',
K=K_L),
        list(model='RKHS',K=K_LE))

########Polynomial Kernel############
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}
```

```
K.poly=K.polynomial(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
dim(K.poly)
K.poly.Exp=Z_L%*%K.poly%*%t(Z_L)

#Covariance matrix for GE term
K.GE.poly= K.E*K.poly.Exp
ETA_K.poly = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
poly.Exp),
          list(model='RKHS',K=K.GE.poly))

########Sigmoid Kernel############
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))

K.sig.Exp= Z_L%*%K.sig%*%t(Z_L)
#Covariance matrix for GE term
K.GE.sig= K.E*K.sig.Exp
ETA_K.sig = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
sig.Exp),
         list(model='RKHS',K=K.GE.sig))

########Gaussian or Radial Kernel############
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.rad)

K.Gauss.Exp= Z_L%*%K.rad%*%t(Z_L)
#Covariance matrix for GE term
K.GE.Gauss=K.E*K.Gauss.Exp
ETA_K.Gauss=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Gauss.Exp),
         list(model='RKHS',K=K.GE.Gauss))

########Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.exp)
K.Expo.Exp= Z_L%*%K.exp%*%t(Z_L)
#Covariance matrix for GE term
K.GE.Exp=K.E*K.Expo.Exp
ETA_K.Exp=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Expo.Exp),
        list(model='RKHS',K=K.GE.Exp))
############Arc-cosine kernel with deep=1#######
K.AK1<-function(X){
```

```
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)
 return(GC1)
}

AK1<-K.AK1(dat_M)
K.AK1.Exp= Z_L%*%AK1%*%t(Z_L)
#Covariance matrix for GE term
K.GE.AK1=K.E*K.AK1.Exp
ETA_K.AK1=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK1.Exp),
        list(model='RKHS',K=K.GE.AK1))
####Arc-cosine kernel with deep=4#####
AK_L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){

  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1
  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
  }

 GC<-GC/median(GC)

 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}

AK4<-AK_L(GC=AK1,nl=4)
K.AK4.Exp= Z_L%*%AK4%*%t(Z_L)
#Covariance matrix for GE term
K.GE.AK4=K.E*K.AK4.Exp
ETA_K.AK4=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK4.Exp),
        list(model='RKHS',K=K.GE.AK4))
```

```
Tab1_PCCC = data.frame(PT = 1:K,PCCC = NA)

for(k in 1:K) {
 set.seed(1)
 Pos_tst =PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA_K.Linear,response_type="ordinal",nIter =
1e4,burnIn = 1e3,verbose = FALSE)
 Probs = A$probs[Pos_tst,]
 yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$Linear[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.poly,response_type="ordinal",nIter =
1e4,burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$poly[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.sig,response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$sig[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.Gauss,response_type="ordinal",nIter =
1e4,burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$Gauss[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.Exp,response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$Exp[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.AK1,response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$AK1[k] = 1-mean(y[Pos_tst]!=yp_ts)

  A = BGLR(y=y_NA,ETA=ETA_K.AK4,response_type="ordinal",nIter = 1e4,
burnIn = 1e3,verbose = FALSE)
  Probs = A$probs[Pos_tst,]
  yp_ts = apply(Probs,1,which.max)-1
  Tab1_PCCC$AK4[k] = 1-mean(y[Pos_tst]!=yp_ts)
 }
Tab1_PCCC
apply(Tab1_PCCC[,-c(1:2)],2,mean)

write.csv(Tab1_PCCC,file="Tab_PCCC.Ex3_kernels_Final.csv")
```

## Appendix 9

R code for implementing seven kernels under a multi-trait Bayesian kernel BLUP with a Gaussian response variable with effects of environment + genotype + genotype ×environment interaction in the predictor (Table 8.10).

```
rm(list=ls())
library(BGLR)
library(BMTME)
library(BGLR)
library(plyr)
library(tidyr)
library(dplyr)
load('Data_Toy_EYT.RData',verbose=TRUE)
ls()
#Phenotypic data
dat_F =Pheno_Toy_EYT
head(dat_F)
dim(dat_F)
#Marker data
dat_M =t(chol(G_Toy_EYT))
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F)

head(dat_F,5)
#Matrix design for markers
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[Pos,]
dim(XM)
XM =XM
#Environment design matrix
XE = model.matrix(~0+Env,data=dat_F)[,-1]
K.E=XE%*%t(XE)

#GID design matrix of lines
Z_L = model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))

n=dim(dat_F)[1]
head(dat_F)
y=dat_F[,3:6]
head(y)

#Number of random partitions
K=10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))
```

```
########Linear kernel=GBLUP
G=G_Toy_EYT
dim(G)
#Covariance matrix for Zg
K_L=Z_L%*%G%*%t(Z_L)
#Covariance matrix for random effects ZEg
K_LE= K.E*K_L

ETA_K.Linear=list(list(model='FIXED',X=XE),list(model='RKHS',
K=K_L),
        list(model='RKHS',K=K_LE))

########Polynomial Kernel############
K.polynomial=function(x1, x2=x1, gamma=1, b=0, p=3){
 (gamma*(as.matrix(x1)%*%t(x2))+b)^p}

K.poly=K.polynomial(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))
dim(K.poly)
K.poly.Exp=Z_L%*%K.poly%*%t(Z_L)

#Covariance matrix for GE
K.GE.poly= K.E*K.poly.Exp
ETA_K.poly = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
poly.Exp),
          list(model='RKHS',K=K.GE.poly))

########Sigmoid Kernel############
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }
K.sig=K.sigmoid(x1=dat_M,x2=dat_M,gamma=1/ncol(dat_M))

K.sig.Exp= Z_L%*%K.sig%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.sig= K.E*K.sig.Exp
ETA_K.sig = list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
sig.Exp),
         list(model='RKHS',K=K.GE.sig))

########Gaussian or Radial Kernel############
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
        Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}
K.rad=K.radial(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.rad)

K.Gauss.Exp= Z_L%*%K.rad%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.Gauss=K.E*K.Gauss.Exp
ETA_K.Gauss=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Gauss.Exp),
        list(model='RKHS',K=K.GE.Gauss))
```

```
#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
          Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

K.exp=K.exponential(x1=dat_M,x2=dat_M, gamma=1/ncol(dat_M))
dim(K.exp)
K.Expo.Exp= Z_L%*%K.exp%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.Exp=K.E*K.Expo.Exp
ETA_K.Exp=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
Expo.Exp),
         list(model='RKHS',K=K.GE.Exp))
###########Arc-cosine kernel with deep=1#######
K.AK1<-function(X){
 n<-nrow(X)
 cosalfa<-cor(t(X))
 angulo<-acos(cosalfa)
 mag<-sqrt(apply(X,1,function(x) crossprod(x)))
 sxy<-tcrossprod(mag)
 GC1<-(1/pi)*sxy*(sin(angulo)+(pi*matrix(1,n,n)-angulo)*cosalfa)
 GC1<-GC1/median(GC1)
 colnames(GC1)<-rownames(X)
 rownames(GC1)<-rownames(X)

 return(GC1)
}

AK1<-K.AK1(dat_M)

K.AK1.Exp= Z_L%*%AK1%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.AK1=K.E*K.AK1.Exp
ETA_K.AK1=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK1.Exp),
        list(model='RKHS',K=K.GE.AK1))
####Arc-cosine kernel with deep=4#####
AK_L<-function(GC,nl){
 n<-nrow(GC)
 GC1<-GC

 for ( l in 1:nl){
  Aux<-tcrossprod(diag(GC))
  cosalfa<-GC*(Aux^(-1/2))
  cosa<-as.vector(cosalfa)
  cosa[which(cosalfa>1)]<-1

  angulo<-acos(cosa)
  angulo<-matrix(angulo,n,n)
  GC<-(1/pi)*(Aux^(1/2))*(sin(angulo)+(pi*matrix(1,n,n)-angulo)
*cos(angulo))
 }
```

```
 GC<-GC/median(GC)
 rownames(GC)<-rownames(GC1)
 colnames(GC)<-colnames(GC1)
 return(GC)
}

AK4<-AK_L(GC=AK1,nl=4)
K.AK4.Exp= Z_L%*%AK4%*%t(Z_L)
#Covariance matrix for random effects ZEg
K.GE.AK4=K.E*K.AK4.Exp
ETA_K.AK4=list(list(model='FIXED',X=XE),list(model='RKHS',K=K.
AK4.Exp),
        list(model='RKHS',K=K.GE.AK4))

source('PC_MM.R')#See below

Tab1_Metrics = data.frame()

for(k in 1:K) {
#k=1
 set.seed(1)
 Pos_tst =PT[,k]
 y_NA = data.matrix(y)
 y_NA[Pos_tst,] = NA

 A1= Multitrait(y = y_NA, ETA=ETA_K.Linear,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
  Me_linear= PC_MM_f(y[Pos_tst,],A1$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  Me_linear

  A2= Multitrait(y = y_NA, ETA=ETA_K.poly,resCov = list(type ="UN",
S0=diag(4),df0= 5),
          nIter =10000, burnIn = 1000)
  Me_poly= PC_MM_f(y[Pos_tst,],A2$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  Me_poly

  A3= Multitrait(y = y_NA, ETA=ETA_K.sig,resCov = list(type ="UN",
S0=diag(4),df0= 5),
          nIter =10000, burnIn = 1000)
   Me_sig= PC_MM_f(y[Pos_tst,],A3$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  Me_sig

  A4= Multitrait(y = y_NA, ETA=ETA_K.Gauss,resCov = list(type ="UN",
S0=diag(4),df0= 5),
          nIter =10000, burnIn = 1000)
  Me_Gauss= PC_MM_f(y[Pos_tst,],A4$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
  Me_Gauss
```

```
 A5= Multitrait(y = y_NA, ETA=ETA_K.Exp,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
 Me_Exp= PC_MM_f(y[Pos_tst,],A5$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
 Me_Exp

 A6= Multitrait(y = y_NA, ETA=ETA_K.AK1,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
 Me_AK1= PC_MM_f(y[Pos_tst,],A6$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
 Me_AK1

 A7= Multitrait(y = y_NA, ETA=ETA_K.AK4,resCov = list(type ="UN",
S0=diag(4),df0= 5),
         nIter =10000, burnIn = 1000)
 Me_AK4= PC_MM_f(y[Pos_tst,],A7$ETAHat[Pos_tst,],Env=dat_F$Env
[Pos_tst])
 Me_AK4
 Tab1_Metrics=rbind(Tab1_Metrics, data.frame(Fold=k,
Trait=Me_linear[,1],Env=Me_linear[,2],MSE_linear=Me_linear[,4],
MSE_poly=Me_poly[,4],MSE_sig=Me_sig[,4],MSE_Gauss=Me_Gauss[,4],
MSE_Exp=Me_Exp[,4],MSE_AK1=Me_AK1[,4],MSE_AK4=Me_AK4[,4]))
 }
Tab1_Metrics
Tab_R = Tab1_Metrics%>%group_by(Env,Trait)%>%select(MSE_linear,
MSE_poly,MSE_sig,MSE_Gauss,MSE_Exp,MSE_AK1,MSE_AK4)%>%summarise
(Linear= mean(MSE_linear),
    Polynomial= mean(MSE_poly),Sigmoid= mean(MSE_sig),Gaussian= mean
(MSE_Gauss),Exponential= mean(MSE_Exp),AK1= mean(MSE_AK4),AK4= mean
(MSE_AK4))
Tab_R = as.data.frame(Tab_R)
Tab_R

write.csv(Tab_R, file="Tab_R_MSE_C.Ex5_kernels_multi_trait_Final.
csv")
```

## Appendix 10

R code for implementing the approximate kernel method proposed by Cuevas et al. (2020) with the wheat599 data set (Table 8.11).

```
rm(list=ls())
library(rrBLUP) #load rrBLUP
library(BGLR) #load BLR
data(wheat) #load wheat data
X=wheat.X
```

```
XF=scale(wheat.X)
rownames(XF)=1:599
dim(XF)
head(XF[,1:5])

############Linear kernel########################
Sparse_linear_kernel=function(m,X){
m=m
XF=X
p=ncol(XF)
pos_m=sample(1:nrow(XF),m)
####Step 1 compute K_m###############3
X_m=XF[pos_m,]
dim(X_m)
K_m=X_m%*%t(X_m)/p
dim(K_m)
######Step 2 compute K_n_m###########
K_n_m=XF%*%t(X_m)/p
dim(K_n_m)
######Step 3 compute eigenvalue decomposition of K_m######
EVD_K_m=eigen(K_m)
####Eigenvectors
U=EVD_K_m$vectors
###Eigenvalues###
S=EVD_K_m$values
####Square root of the inverse of eigenvalues #####
S_0.5_Inv=sqrt(1/S)
#####Diagonal matrix of square root of inverse of eigenvalues###
S_mat_Inv=diag(S_0.5_Inv)
######Computing matrix P
P=K_n_m%*%U%*%S_mat_Inv
return(P)}

#####K-fold cross-validation
n=nrow(XF)
No.folds=10
set.seed(2)
Grpv= findInterval(cut(sample(1:n,n),breaks=No.folds),1:n)
Grpv
Env_name=colnames(wheat.Y)
results_all_traits=data.frame()
for (t in 1:4){
t=t
rownames(wheat.Y)=1:599
y2=wheat.Y[,t]
mvec=c(15,32,74,132,264,599)
results_all=data.frame()

for (j in 1:6){
m=mvec[j]
P=Sparse_linear_kernel(m,X=XF)
results=data.frame()
start_time <- proc.time()
```

```
for(r in 1:No.folds) {
  y1=y2
  positionTST=which(Grpv==r)
  positionTRN=which(Grpv!=r)
  y1[positionTST] = NA
  ETA=list(list(model='BRR',X=P))
  A=BGLR(y=y1,ETA=ETA,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
  yHat= A$yHat[positionTST]
  MSE=mean((y2[positionTST]-yHat)^2)
  Cor=cor(y2[positionTST],yHat)
  results=rbind(results,data.frame(MSE=MSE, Cor=Cor))
  }
Summary=apply(results,2,mean)
end_time <- proc.time()
Time=c(end_time[1] - start_time[1])
Time
results_all=rbind(results_all,data.frame(m=m, MSE=Summary[1],
Cor=Summary[2], Time=Time))
}
results_all
results_all_traits=rbind(results_all_traits,data.frame
(results_all))
}
results_all_traits
write.csv(results_all_traits,file="Table8.11_Final.csv")
```

# Appendix 11

R code for implementing the approximate kernel method with five kernels with the Data_Toy_EYT data set (Table 8.13).

```
rm(list=ls())
library(BGLR)
library(BMTME)
load('Data_Toy_EYT.RData',verbose=TRUE)
ls()
#Phenotypic data
dat_F =Pheno_Toy_EYT
head(dat_F)
dim(dat_F)
#Marker data as Cholesky of the genomic relationship matrix
dat_M =t(chol(G_Toy_EYT))
dim(dat_M)

dat_F = transform(dat_F, GID = as.character(GID))
head(dat_F)

head(dat_F,5)
#Matrix design for markers
```

```
Pos = match(dat_F$GID,row.names(dat_M))
XM = dat_M[unique(Pos),]
dim(XM)

#######Gaussian Kernel function###########
l2norm=function(x){sqrt(sum(x^2))}
K.radial=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j])^2)))}

#######Polynomial Kernel###########
K.polynomial=function(x1, x2=x1, gamma=1, b=0, d=3)
 { (gamma*(as.matrix(x1)%*%t(x2))+b)^d}

#######Sigmoid Kernel###########
K.sigmoid=function(x1,x2=x1, gamma=1, b=0)
{ tanh(gamma*(as.matrix(x1)%*%t(x2))+b) }

#######Exponential Kernel###########
K.exponential=function(x1,x2=x1, gamma=1){
 exp(-gamma*outer(1:nrow(x1 <- as.matrix(x1)), 1:ncol(x2 <- t(x2)),
         Vectorize(function(i, j) l2norm(x1[i,]-x2[,j]))))}

###########Approximate Guassian Kernel#######################
Sparse_kernel=function(m,X,name){
 m=m
 XF=X
 p=ncol(XF)
 pos_m=sample(1:nrow(XF),m)
 ####Step 1 compute K_m###############3
 X_m=XF[pos_m,]
 dim(X_m)
 if (name=="Linear") {
  K_m=X_m%*%t(X_m)/p
  ######Step 2 compute K_n_m###########
  K_n_m=XF%*%t(X_m)/p
 } else if (name=="Polynomial") {
  K_m=K.polynomial(x1=X_m,x2=X_m,gamma=1/p)
  ######Step 2 compute K_n_m###########
  K_n_m=K.polynomial(x1=XF,x2=X_m,gamma=1/p)
 } else if  (name=="Sigmoid") {
  K_m=K.sigmoid(x1=X_m,x2=X_m,gamma=1/p)
  ######Step 2 compute K_n_m###########
  K_n_m=K.sigmoid(x1=XF,x2=X_m,gamma=1/p)
 }else if  (name=="Gaussian") {
  K_m=K.radial(x1=X_m,x2=X_m,gamma=1/p)
  ######Step 2 compute K_n_m###########
  K_n_m=K.radial(x1=XF,x2=X_m,gamma=1/p)
 } else {
  K_m=K.exponential(x1=X_m,x2=X_m,gamma=1/p)
  ######Step 2 compute K_n_m###########
```

```
  K_n_m=K.exponential(x1=XF,x2=X_m,gamma=1/p)
 }

 ######Step 3 compute eigenvalue decomposition of K_m######
 EVD_K_m=eigen(K_m)
 ####Eigenvectors
 U=EVD_K_m$vectors
 ###Eigenvalues###
 S=EVD_K_m$values
 ####Square root of the inverse of eigenvalues #####
 S_0.5_Inv=sqrt(1/S)
 #####Diagonal matrix of square root of inverse of eigenvalues###
 S_mat_Inv=diag(S_0.5_Inv)
 ######Computing matrix P
 P=K_n_m%*%U%*%S_mat_Inv
 return(P)}

#Environment design matrix
XE = model.matrix(~0+Env,data=dat_F)

#####Design matrix of lines
Z_L=model.matrix(~0+GID,data=dat_F,xlev = list(GID=unique
(dat_F$GID)))
dim(Z_L)
############Total observations in the data set and response variable
n=dim(dat_F)[1]
y=dat_F$GY

#Number of random partitions
K=10
set.seed(1)
PT = replicate(K,sample(n,0.20*n))
####Trainig sample size of lines m that will be used for training the
model
mvec=c(round(40*0.1),round(40*0.2),round(40*0.3),round(40*0.4),
round(40*0.5),round(40*1))
mvec
kernel_name=c("Linear","Polynomial", "Sigmoid", "Gaussian",
"Exponential")
results_all_kernels=data.frame()
for (i in 1:5) {
results_all=data.frame()
for (j in 1:6){
m=mvec[j]
P_Lines=Sparse_kernel(m=m,X=XM,name=kernel_name[i])
Z_Lines_Sparse=Z_L%*%P_Lines
#####Design matrix of lines x Environment interaction
Z_LE = model.matrix(~0+Z_Lines_Sparse:Env,data=dat_F)

ETA=list(list(model='FIXED',X=XE[,-1]),list(model='BRR',
X=Z_Lines_Sparse),
    list(model='BRR',X=Z_LE))
```

```
Tab1_Metrics= data.frame(PT = 1:K,MSE = NA)
start_time <- proc.time()
for(k in 1:K) {
 Pos_tst =PT[,k]
 y_NA = y
 y_NA[Pos_tst] = NA
 A = BGLR(y=y_NA,ETA=ETA,nIter = 1e4,burnIn = 1e3,verbose = FALSE)
 yp_ts = A$yHat
  Tab1_Metrics$MSE[k] = mean((y[Pos_tst]-yp_ts[Pos_tst])^2)
  Tab1_Metrics$Cor[k] = cor(y[Pos_tst],yp_ts[Pos_tst])
}
end_time <- proc.time()
Time=c(end_time[1] - start_time[1])
Metrics=apply(Tab1_Metrics[,-c(1)],2,mean)
results_all=rbind(results_all,data.frame(m=m, MSE=Metrics[1],
Cor=Metrics[2], Time=Time))
}
results_all_kernels=rbind(results_all_kernels,data.frame
(kernel=kernel_name[i], t(results_all)))

}
results_all_kernels
write.csv(results_all_kernels,
file="Table_8.13_results_kernels_Final.csv")
```

## References


Akhiezer NI, Glazman IM (1963) Theory of linear operators in Hilbert Space (Teoriia lineikykh operatorov v Gil'bertovom prostranstve), vol 1. M. Nestell, trans. from Russian. Frederick Ungar, New York

Buil A, Brown AA, Lappalainen T, Viñuela A, Davies MN, Zheng HF, Richards JB, Glass D, Small KS, Durbin R et al (2015) Gene-gene and gene-environment interactions detected by transcriptome sequence analysis in twins. Nat Genet 47:88–91

Cho Y, Saul LK (2009) Kernel methods for deep learning. In: NIPS'09 proceedings of the 22nd international conference on neural information processing systems, pp 342–350

Cordell HJ (2002) Epistasis: what it means, what it doesn't mean, and statistical methods to detect it in humans. Hum Mol Genet 11:2463–2468

Cordell HJ (2009) Detecting gene-gene interactions that underlie human diseases. Nat Rev Genet 10:392–404

Crossa J, de los Campos G, Pérez P, Gianola D, Burgueño J, Araus JL et al (2010) Prediction of genetic values of quantitative traits in plant breeding using pedigree and molecular markers. Genetics 186:713–724. https://doi.org/10.1534/genetics.110.118521

Cuevas J, Crossa J, Soberanis V, Pérez-Elizalde S, Pérez-Rodríguez P, de los Campos G, Montesinos-López OA, Burgueño J (2016) Genomic prediction of genotype × environment interaction kernel regression models. Plant Genome 9(3):1–20

Cuevas J, Crossa J, Montesinos-López OA, Burgueño J, Pérez-Rodríguez P, de los Campos G (2017) Bayesian Genomic prediction with genotype × environment kernel models. G3 7(1):41–53

Cuevas J, Granato I, Fritsche-Neto R, Montesinos-Lopez OA, Burgueño J, Bandeira e Sousa M, Crossa J (2018) Genomic-enabled prediction kernel models with random intercepts for multi-environment trials. G3 8(4):1347–1365

Cuevas J, Montesinos-López OA, Juliana P, Guzmán C, Pérez-Rodríguez P, González-Bucio J, Burgueño J, Montesinos-López A, Crossa J (2019) Deep kernel for genomic and near infrared predictions in multi-environment breeding trials. G3 9(9):2913–2924

Cuevas J, Montesinos-Lopez OA, Martini JW, Pérez-Rodríguez P, Lillemo M, Crossa J (2020) Approximate genome-based kernels models for large data sets including main effects and interactions. Front Genet 11:567757. https://doi.org/10.3389/fgene.2020.567757

de los Campos G, Gianola D, Rosa GJ, Weigel KA, Crossa J (2010) Semi-parametric genomic-enabled prediction of genetic values using reproducing kernel Hilbert spaces methods. Genet Res (Camb) 92:295–308. https://doi.org/10.1017/S0016672310000285

Endelman JB (2011) Ridge regression and other kernels for genomic selection with R package rrBLUP. Plant Genome 4:250–255. https://doi.org/10.3835/plantgenome2011.08.0024

Gianola D, van Kaam JBCHM (2008) Reproducing kernel Hilbert spaces regression methods for genomic assisted prediction of quantitative traits. Genetics 178:2289–2303. https://doi.org/10.1534/genetics.107.084285

Gianola D, Fernando RL, Stella A (2006) Genomic-assisted prediction of genetic value with semi parametric procedures. Genetics 173:1761–1776. https://doi.org/10.1534/genetics.105.049510

Golan D, Rosset S (2014) Effective genetic-risk prediction using mixed models. Am J Hum Genet 95:383–393

Hemani G, Shakhbazov K, Westra HJ, Esko T, Henders AK, McRae AF, Yang J, Gibson G, Martin NG, Metspalu A et al (2014) Detection and replication of epistasis influencing transcription in humans. Nature 508:249–253

Henderson C (1975) Best linear unbiased estimation and prediction under a selection model. Biometrics 31(2):423–447. https://doi.org/10.2307/2529430

Kang HM, Zaitlen NA, Wade CM, Kirby A, Heckerman D, Daly MJ, Eskin E (2008) Efficient control of population structure in model organism association mapping. Genetics 178:1709–1723

Lehner B (2011) Molecular mechanisms of epistasis within and between genes. Trends Genet 27: 323–331

Lin HT, Lin CJ (2003) A study on sigmoid kernels for SVM and the training of non-PSD kernels by SMO-type methods. Neural Comput 3:1–32

Long N, Gianola D, Rosa GJ, Weigel KA, Kranis A, González- Recio, O. (2010) Radial basis function regression methods for predicting quantitative traits using SNP markers. Genet Res 92: 209–225. https://doi.org/10.1017/S0016672310000157

Mallick BK, Ghosh D, Ghosh M (2005) Bayesian classification of tumours by using gene expression data. J R Stat Soc B 67:219–234

Misztal I, Legarra A, Aguilar I (2014) Using recursion to compute the inverse of the genomic relationship matrix. J Dairy Sci 97:3943–3952

Moore JH, Williams SM (2009) Epistasis and its implications for personal genetics. Am J Hum Genet 85:309–320

Morota G, Koyama M, Rosa GJM, Weigel KA, Gianola D (2013) Predicting complex traits using a diffusion kernel on genetic markers with an application to dairy cattle and wheat data. Genet Sel Evol 45:17. https://doi.org/10.1186/1297-9686-45-17

Morota G, Boddhireddy P, Vukasinovic N, Gianola D, DeNise S (2014) Kernel-based variance component estimation and whole-genome prediction of pre-corrected phenotypes and progeny tests for dairy cow health traits. Front Genet 5:56. https://doi.org/10.3389/fgene.2014.00056

Ober U, Erbe M, Long N, Porcu E, Schlather M, Simianer H (2011) Predicting genetic values: a kernel-based best linear unbiased prediction with genomic data. Genetics 188:695–708. https://doi.org/10.1534/genetics.111.128694

Pérez-Elizalde S, Cuevas J, Pérez-Rodríguez P, Crossa J (2015) Selection of the bandwidth parameter in a Bayesian kernel regression model for genomic-enabled prediction. J Agric Biol Environ Stat 20:512–532. https://doi.org/10.1007/s13253-015-0229-y

Rassmussen CE, Williams CK (2006) Gaussian processes for machine learning. MIT Press, Cambridge, MA. ISBN 0-262-18253-X

Schrodi SJ, Mukherjee S, Shan Y, Tromp G, Sninsky JJ, Callear AP, Carter TC, Ye Z, Haines JL, Brilliant MH et al (2014) Genetic-based prediction of disease traits: prediction is very difficult, especially about the future. Front Genet 5:162

Seeger M, Williams CKI, Lawrence N (2003) Fast forward selection to speed up sparse gaussian process regression. In: Bishop C, Frey BJ (eds) Proceedings of the ninth international workshop on artificial intelligence and statistics. Society for Artificial Intelligence and Statistics

Shawe-Taylor J, Cristianini N (2004) Kernel methods for pattern analysis. University Press, Cambridge, UK

Snelson E, Ghahramani Z (2006) Local and global sparse Gaussian process approximations. In: Meilia M, Shen X (eds) Proceedings of the eleven international workshop on artificial intelligence and statistics, Society for Artificial Intelligence and Statistics. Omnipress

Titsias MK (2009) Variational learning of inducing variables in sparse Gaussian Processes. In: van Dyk D, Welling M (eds) Proceedings of the eleven international workshop on artificial intelligence and statistics, Clearwater Beach, FL, 16-18 April 2009, vol 5, pp 567–574. JMLR W&CP 5

Tusell L, Pérez-Rodríguez P, Wu SF-L, Gianola D (2013) Genome-enabled methods for predicting litter size in pigs: a comparison. Animal 7:1739–1749. https://doi.org/10.1017/S1751731113001389

VanRaden PM (2008) Efficient methods to compute genomic predictions. J Dairy Sci 91:4414–4423. https://doi.org/10.3168/jds.2007-0980

Vapnik V (1998) Statistical learning theory. Wiley, Hoboken, NJ

Wahba G (1990) Spline models for observational data. Society for Industrial and Applied Mathematics, Philadelphia

Williams CKI, Seeger M (2001) Using the Nyström method to speed up kernel machines. In: Leen TK, Diettrich TG, Tresp V (eds) Advances in neural information processing systems 13. MIT Press, Cambridge, MA, pp 682–688

Zhang Z, Dai G, Jordan MI (2011) Bayesian generalized kernel mixed models. J Mach Learn Res 12:111–139

Zuk O, Hechter E, Sunyaev SR, Lander ES (2012) The mystery of missing heritability: genetic interactions create phantom heritability. Proc Natl Acad Sci U S A 109:1193–1198

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 9
# Support Vector Machines and Support Vector Regression

## 9.1 Introduction to Support Vector Machine

The Support Vector Machine (SVM) is one of the most popular and efficient supervised statistical machine learning algorithms, which was proposed to the computer science community in the 1990s by Vapnik (1995) and is used mostly for classification problems. Its versatility is due to the fact that it can learn nonlinear decision surfaces and perform well in the presence of a large number of predictors, even with a small number of cases. This makes the SVM very appealing for tackling a wide range of problems such as speech recognition, text categorization, image recognition, face detection, faulty card detection, junk mail classification, credit rating analysis, cancer and diabetes classification, among others (Attewell et al. 2015; Byun and Lee 2002). Most of the groundwork for the SVM was laid by Vladimir Vapnik while he was working on his Ph.D. thesis in the Soviet Union in the 1960s. Then Vapnik emigrated to U.S. in 1990 to work with AT&T. In 1992, Bernhard E. Boser, Isabelle M. Guyon, and Vladimir N. Vapnik suggested applying the kernel trick to maximum-margin hyperplanes to capture nonlinearities in classification problems. Finally, Cortes and Vapnik (1995) introduced the SVM to the world in its more efficient mode, and since the mid-1990s, the SVM has been a very popular topic in statistical machine learning.

The SVM method works by representing the observations (data) as points in space by mapping the original observations of different classes (categories) in such a way that they are divided by an evident gap that is as extensive as possible. The predictions of new observations are done by mapping these observations into the same space, and they are allocated to one or another category depending on which side of the gap they fall.

SVM methods are very efficient for classifying nonlinear separable patterns in part by the use of the kernel trick, explained in the previous chapter, which consists of transforming the original input information into a high-dimensional feature space by enlarging the feature space using functions of the predictors; this makes it

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_9

possible to accommodate a nonlinear boundary between the classes, without significantly increasing the computational cost.

As mentioned above, the SVM is a type of supervised learning method, which means that it cannot be implemented when the data do not have a dependent or output variable ($y$). Also, it is important to point out that the mathematics behind the SVM has been around for a long time and is quite complex, but the popularity of this method is very recent. The popularity of this method can be attributed to three main reasons: (a) the increase in computational power, (b) ample evidence of the high prediction performance of this method, and (c) the availability of user-friendly libraries in many languages that are able to implement the SVM method. For this reason, the SVM has been implemented in many domains that range from social science to natural sciences, since it is not only used for tasks relating to the prediction of categorical variables but also for the prediction of continuous outputs.

The mathematics of the SVM was originally developed for classifying binary outputs, and for this reason, this type of application is more popular and better understood. However, there is also evidence that the SVM is doing a good job predicting continuous outputs and novelty detection. The power of the SVM can be attributed to the following facts: (a) it is a kernel-based algorithm that has a sparse solution, since the prediction of new inputs is done by evaluating the kernel function in a subset of the training data points and (b) the estimation of the model parameters corresponds to a convex optimization problem, which means that they always provide a global optimum (Bishop 2006).

First, we will study the SVM for classification and then for the prediction of continuous outputs. To understand the SVM better, it is important to understand its ancestors, i.e., the maximum margin classifier and the support vector classifier. The *maximum margin classifier* is a simple and elegant method for classifying binary outputs that assume that the classes are separable by a linear boundary. However, it is not feasible to apply this method to many data sets since it requires the strong assumption that classes are separable by a linear boundary. The *support vector classifier* is an extension of the maximum margin classifier that allows to misclassify some of the training data and thus creates a separable linear boundary with a reasonable width (margin) that is more robust to overfitting. Finally, the *support vector machine* is a generalization of the support vector classifier that classifies the observations using nonlinear boundaries by expanding the feature space with the help of kernels (James et al. 2013).

## 9.2 Hyperplane

A *hyperplane* is a subspace whose dimension (cardinality) is one less than that of its original space. This means that the hyperplane of a $p$-dimensional space has a subspace of dimension $p - 1$. In Fig. 9.1 (left), we can see a two-dimensional space whose resulting hyperplane is a line, a flat one-dimensional subspace, while in Fig. 9.1 (right) there is a three-dimensional space whose hyperplane is a plane, a flat

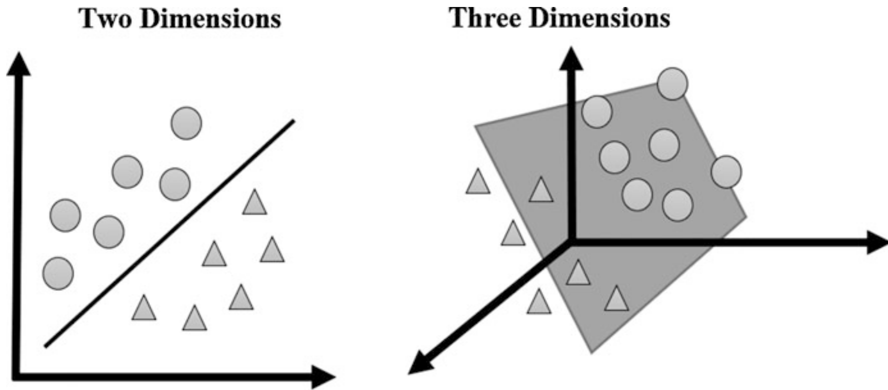



**Fig. 9.1** Hyperplanes in two (left) and three (right) dimensions

two-dimensional subspace. Although it is hard to visualize a hyperplane when the original space has a dimension of four or more, it still applies for the $(p - 1)$-dimensional flat subspace (James et al. 2013). In higher dimensions, it is useful to think of a hyperplane as a member of an affine family of $(p - 1)$-dimensional subspaces (affine spaces look and behave very similarly to linear spaces without the requirement to contain the origin), such that the whole space is partitioned into these family subspaces.

From a mathematical point of view, a hyperplane is defined as (James et al. 2013)

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 = 0 \tag{9.1}$$

for parameters $\beta_0$, $\beta_1$, $\beta_2$, and $\beta_3$. (9.1) "defines" a hyperplane, since any $X = (X_1, X_2, X_3)^T$ for which (9.1) holds is a point in the hyperplane. Equation (9.1) is the equation of a plane, since in three dimensions, as mentioned before, a hyperplane is a plane, as can be observed in Fig. 9.1 (right).

For the $p$-dimensional space, the dimension of the hyperplane generated is $p - 1$, and it is simply an extension of (9.1) as

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p = 0 \tag{9.2}$$

In the same way, any point $X = (X_1, X_2, \ldots . X_p)^T$ in the $p$-dimensional space that satisfies (9.2) defines a $(p - 1)$-dimensional hyperplane, which means that the hyperplane is formed by those points of $X$ that satisfy (9.2) (James et al. 2013). But those points of $X$ that do not satisfy (9.2) like, for example,

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p < 0 \tag{9.3}$$

There are points that satisfying (9.3) lie on one side of the hyperplane. Similarly, the $X$ points that correspond to

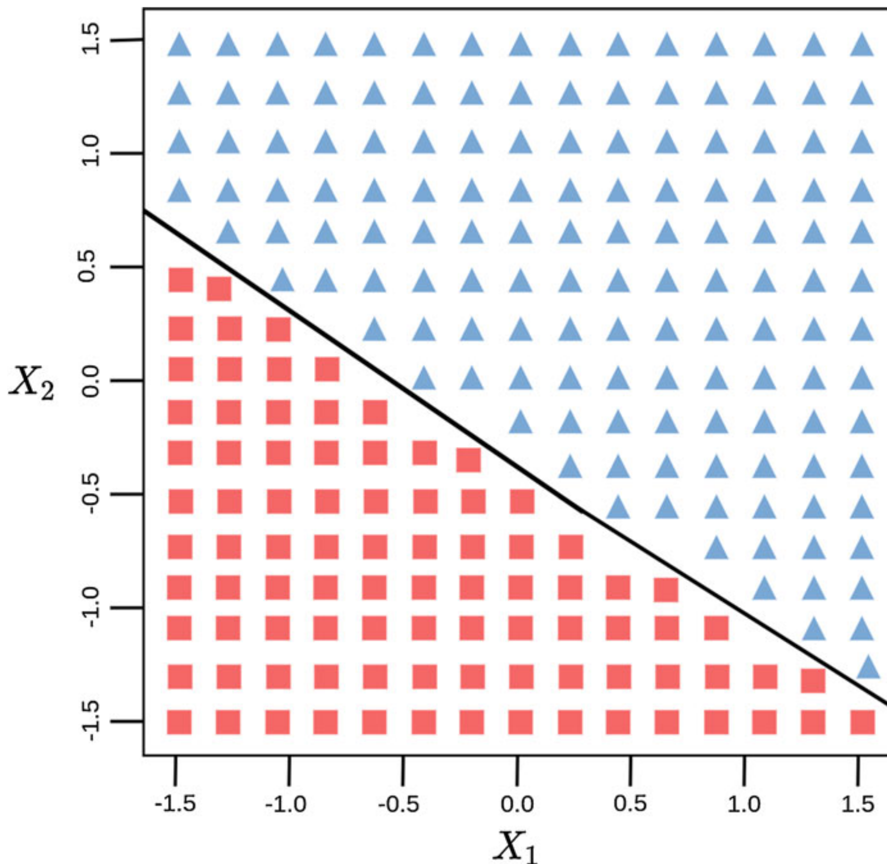



**Fig. 9.2** The hyperplane $1 + 2X_1 + 3X_2 = 0$ is shown. The blue region is the set of points for which $1 + 2X_1 + 3X_2 > 0$, and the red region is the set of points for which $1 + 2X_1 + 3X_2 < 0$ (James et al. 2013)

$$\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p > 0 \tag{9.4}$$

will lie on the other side of the hyperplane. This means that we can think of the hyperplane as a mechanism that can divide the $p$-dimensional space into two halves. By simply calculating the sign of the left-hand side of (9.2), one can determine on which side of the hyperplane a point lies (James et al. 2013). Figure 9.2 shows a hyperplane in two-dimensional space.

## 9.3 Maximum Margin Classifier

We assume that we measure a training sample with pairs $(y_i, \boldsymbol{x}_i^{\mathrm{T}})$ for $i = 1, 2, \ldots, n$, where $y_i$ is the response variable (output) for sample $i$, and $\boldsymbol{x}_i^{\mathrm{T}} = (x_{i1}, \ldots, x_{ip})$ is a $p$-dimensional vector of predictors (inputs) measured in sample $i$. We also assume that

the response variable is binary (two classes) and coded as 1 for representing class 1 and −1 for representing class 2. A fitting function of the form

$$f(\boldsymbol{x}_i) = \beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} \tag{9.5}$$

can be used for building a classifier based on the training data set, where $\beta_0$ is an intercept term, and $\boldsymbol{\beta}^{\mathrm{T}} = (\beta_1, \ldots, \beta_p)$ are the beta coefficients (weights) that need to be estimated to build the required classifier. Once the beta coefficients have been estimated, $\left(\widehat{\beta}_0, \widehat{\boldsymbol{\beta}}\right)$, they can be used to predict the output of a test observation that contains $\boldsymbol{x}_{i*}^{\mathrm{T}} = \left(x_{i1*}, \ldots, x_{ip*}\right)$ as a predictor. The prediction of this new test observation is labeled as 1 if $\widehat{f}(\boldsymbol{x}_{i*}) = \widehat{\beta}_0 + \boldsymbol{x}_{i*}^{\mathrm{T}}\widehat{\boldsymbol{\beta}}$ is positive, and labeled as −1 if $\widehat{f}(\boldsymbol{x}_{i*})$ is negative. $\widehat{f}(\boldsymbol{x}_{i*})$ is calculated with the estimates of the beta coefficients. Before estimating the required beta coefficients, we assume for the moment that the training data set is linearly separable in the predictor space, which means that there is at least one set of beta coefficient parameters, $(\beta_0, \boldsymbol{\beta})$, so that using the function given in (9.5), we can assume that $f(\boldsymbol{x}_i) < 0$, for observations having $y_i = -1$ and $f(\boldsymbol{x}_i) > 0$ for observations having $y_i = 1$, so that $y_i f(\boldsymbol{x}_i) > 0$ for all training observations (Bishop 2006).

Let us assume that 40 hybrids of maize were evaluated for the presence (1) or absence (−1) of a certain disease and that in addition to the output of interest, we also measured in each hybrid two predictors ($x_1$, $x_2$), which could be markers linked to this disease. Figure 9.3 shows that the 40 observations can be separated by a line into

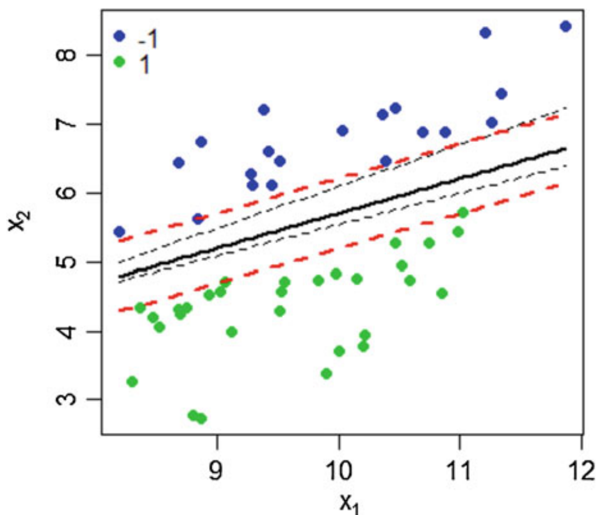



**Fig. 9.3** Synthetic linear separable data set with the hyperplane $1.3467 + 0.9927x_1 - 1.9779x_2 = 0$. The blue points correspond to the individuals with response −1 and the green points to individuals with response equal to 1. Note that the black dotted lines are two of many possible linear hyperplanes that correctly classify both classes. The black continuous line corresponds to the optimum hyperplane; the dotted red lines correspond to the maximum margin bounds and, relative to the rest, they play the biggest role in predicting new points

the two classes, that is, it is possible to construct a hyperplane that is able to perfectly classify both classes of the training data set. As can be seen in Fig. 9.3, the 1s (green points) and $-1$s (blue points) are each located in quite different areas of the two-dimensional space defined by the two predictors. For this reason, it is possible to perfectly separate the training data with a dividing line between the 1s (green points) and $-1$s (blue points). However, in Fig. 9.3, three possible dividing lines (two dotted lines and one continuous line) were used to separate the two classes perfectly, but of course the separation can be made with an infinite number of dividing lines. Therefore, the question of interest is: How to choose the dividing line in such a way that we can separate the training sample perfectly and, in addition, classify new samples with a low rate of misclassification? The answer to this question is not hard, but neither is it straightforward since there are many possible dividing lines when the pattern of the data is similar to the one shown in Fig. 9.3.

In terms of equations, this hyperplane has the property that

$$\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} < 0 \text{ if } y_i = -1,$$

and

$$\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} > 0 \text{ if } y_i = 1.$$

In its equivalent formulation, the hyperplane can be expressed as

$$y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) > 0$$

for all $i = 1, 2, \ldots, n$. When this separating hyperplane is found, a natural classifier is built and testing observations are classified depending on which side of the hyperplane they are located and, as mentioned above, test observation $\boldsymbol{x}_{i*}$ is used to calculate $\widehat{f}(\boldsymbol{x}_{i*}) = \widehat{\beta}_0 + \boldsymbol{x}_{i*}^{\mathrm{T}}\widehat{\boldsymbol{\beta}}$; if $\widehat{f}(\boldsymbol{x}_{i*})$ is negative, it is classified in class $-1$, but if $\widehat{f}(\boldsymbol{x}_{i*})$ is positive, it is classified in class 1. Large positive (or negative) values of $\widehat{f}(\boldsymbol{x}_{i*})$ indicate that we can be more confident about our class assignment for $\boldsymbol{x}_{i*}$, while values close to zero of $\widehat{f}(\boldsymbol{x}_{i*})$ indicate that we should be less certain of the class assignment of $\boldsymbol{x}_{i*}$. For this reason, classifiers based on a separating hyperplane require defining a linear decision boundary (James et al. 2013).

For data with patterns similar to the pattern in Fig. 9.3, the maximum margin classifier solves the problem of finding the “best” decision boundary by building two parallel lines on each side of the decision boundary and at the same distance from the decision boundary. The two lines should be as far apart as possible, taking care that any observation is within the space between them. The space between the two lines is called the margin, and it is a kind of “buffer zone” that is often also called width of the street. The preferred term is maximum margin or maximum width of the street, since intuitively it looks like it will improve the chances of correctly classifying even new observations not used for training. In other words, the strategy is to find the “street” which separates the data into two groups such that the street is as wide as

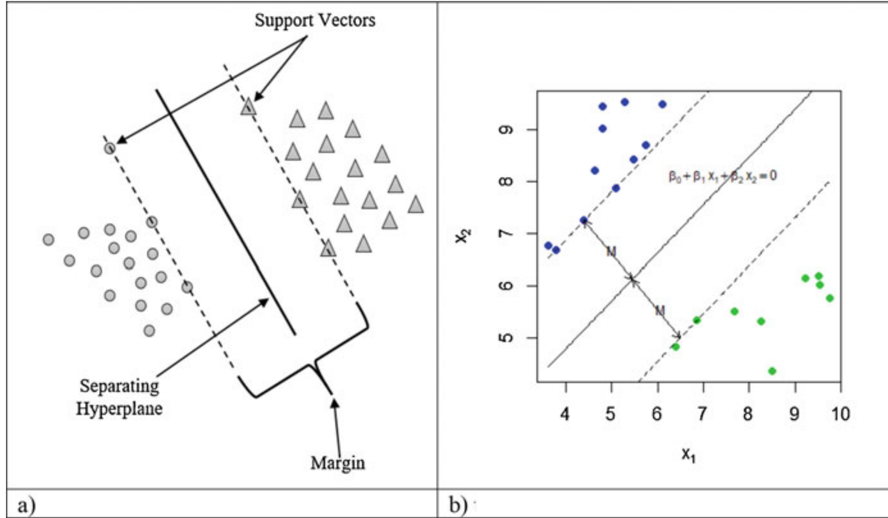



**Fig. 9.4** Maximum margin hyperplane when there are two separable classes. The maximum margin hyperplane is shown as a dashed line. The margin is the distance from the dashed line to any point on the solid line. The support vectors are the dots from each class that touch to the maximum margin hyperplane and each class must have a least one support vector. In (**a**) the two classes are circles and triangles and in (**b**) the two classes are dots in green and blue

possible, and the equation that would correspond to the "median" of this street. Our decision is made according to the position of a point relative to this median.

Figure 9.4a, b shows the margin ($M$), that is, the distance between any point and the hyperplane, while the whole width of the street is $2M$. The points touching this boundary are the support vectors (in Fig. 9.4a, the circles and triangles shown are the support vectors, while in Fig. 9.4b, they are green and blue dots) and each class must have at least one support vector. Here, the solid line maximizes the distance, so it is the best. It is possible to define the maximum margin hyperplane with only the support vectors, and for this reason, they provide a very compact way of storing a classification model, even if the number of predictors is very large.

The algorithm used to find the right support vectors relies on vector geometry and involves novel math that will be explained next.

### *9.3.1 Derivation of the Maximum Margin Classifier*

We assume that the training sample is linearly separable, that is, that there is a hyperplane that separates the training sample perfectly into two populations that can be labeled as 1s and $-1$s or as white and black or any other two labeling options. However, as pointed out before, there is an infinite number of such separating hyperplanes; for this reason, to select one reasonable hyperplane, we will choose

the hyperplane with the maximum margin ($M$). The continous line in Fig. 9.4 illustrates the best choice. Therefore, assuming that we have a training set with input information, $\boldsymbol{x}_i^{\mathrm{T}} = (x_1, \ldots, x_p)$ and with output information, $y_i \in (-1, 1)$ for $i = 1, \ldots, n$. Next, we derive the maximum margin hyperplane, which is the solution to the following optimization problem (James et al. 2013):

$$\underbrace{\text{maximize}}_{\beta_0, \beta_1, \beta_2, \ldots, \beta_p} M \tag{9.6}$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1,$$

$$y_i(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}) \geq M, \quad i = 1, \ldots, n$$

The term $y_i(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta})$ in the restrictions of (9.6) of this optimization problem is the distance between the $i$th observation and the decision boundary and is essential for correctly identifying classified observations on or beyond the margin boundary, given that $M$ is positive. $2M$ is the whole margin or width of the street (see Fig. 9.4b), since $M$ (half-width of the street) is the distance, centered on the decision boundary, to the margin boundary from the decision boundary. It is important to point out that the constraints given in (9.4) and (9.5) guarantee that each observation will fall on the correct side of the hyperplane and at a distance of at least $M$ from the hyperplane. The fact that the last restriction of (9.6) applies to all observations ($i = 1, \ldots, n$) means that no observations are inside the street (whole margin) or fences. Hence, the goal of the maximum margin hyperplane is to find the values of the beta coefficients, $\beta_0, \beta_1, \beta_2, \ldots, \beta_p$, that maximize the margin ($M$) avoiding that some observations are inside the fences (street).

To obtain the distance from a point to the hyperplane, consider point $\boldsymbol{x}$ in Fig. 9.5. Note that from any two points $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ lying in hyperplane $H$, we have that $\beta_0 + \boldsymbol{x}_1^{\mathrm{T}}\boldsymbol{\beta} = 0$ and $\beta_0 + \boldsymbol{x}_2^{\mathrm{T}}\boldsymbol{\beta} = 0$, which implies that $(\boldsymbol{x}_1 - \boldsymbol{x}_2)^{\mathrm{T}}\boldsymbol{\beta} = 0$. But because $\boldsymbol{x}_1 - \boldsymbol{x}_2$ is a vector in $H$, then $\boldsymbol{\beta}$ is orthogonal to $H$, and consequently also to the normalized $\boldsymbol{\beta}$ vector, $\boldsymbol{\beta}^* = \frac{\boldsymbol{\beta}}{||\boldsymbol{\beta}||}$ (see Fig. 9.5). To solve the optimization problem (9.6), it is very important to determine the distance (margin, $M$) from point $\boldsymbol{x}$ to hyperplane $H$, which is given by the norm of the projection vector of $\boldsymbol{x} - \boldsymbol{x}_i$ on vector $\boldsymbol{\beta}^*$, where $\boldsymbol{x}_i$ is the vector formed by the intersection point of vector $\boldsymbol{\beta}^*$ and the hyperplane. Recall that the projection of $\boldsymbol{a}$ onto $\boldsymbol{b}$ is equal to $P_{\boldsymbol{b}}(\boldsymbol{a}) = \left(\frac{\boldsymbol{a}^{\mathrm{T}}\boldsymbol{b}}{\left\|\boldsymbol{b}\right\|}\right)\left(\frac{\boldsymbol{b}}{\left\|\boldsymbol{b}\right\|}\right)$. Therefore, $P_{\boldsymbol{\beta}^*}(\boldsymbol{x} - \boldsymbol{x}_i) = \left(\frac{(\boldsymbol{x} - \boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\beta}^*}{\|\boldsymbol{\beta}^*\|^2}\right)\boldsymbol{\beta}^*$ but because $\|\boldsymbol{\beta}^*\| = 1$, then $P_{\boldsymbol{\beta}^*}(\boldsymbol{x} - \boldsymbol{x}_i) = \left((\boldsymbol{x} - \boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\beta}^*\right)\boldsymbol{\beta}^* = \frac{(\boldsymbol{x}^{\mathrm{T}}\boldsymbol{\beta} + \beta_0)\boldsymbol{\beta}^*}{\left\|\boldsymbol{\beta}\right\|}$. Therefore, the norm of $P_{\boldsymbol{\beta}^*}(\boldsymbol{x} - \boldsymbol{x}_i)$ is equal to the margin between the hyperplane and any of the support vectors (***M*),** which is equal to

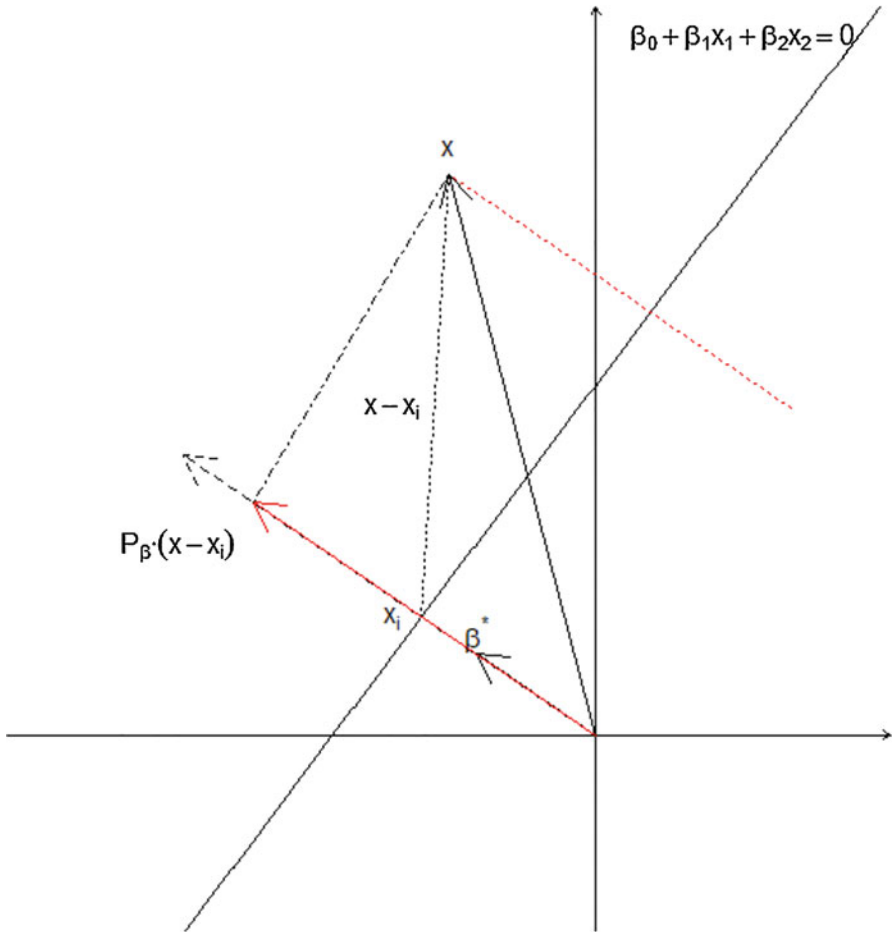



**Fig. 9.5** Distance from a point ($\boldsymbol{x}$) to a point ($\boldsymbol{x}_i$) in the hyperplane ($\beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0$)

$$\boldsymbol{M} = \frac{\left|\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right|\left\|\boldsymbol{\beta}^*\right\|}{\left\|\boldsymbol{\beta}\right\|} = \frac{\left|\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right|1}{\left\|\boldsymbol{\beta}\right\|} = 1/\|\boldsymbol{\beta}\|$$

This distance is equal to the distance (margin, $M$) from hyperplane ($h_0 = \beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} = 0$) to hyperplane ($h_1 = \beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta} = 1$). This means that the total distance is equal to $2\boldsymbol{M} = 2/\|\boldsymbol{\beta}\|$. This implies that maximizing $\boldsymbol{M} = 1/\|\boldsymbol{\beta}\|$ subject to the constraints of (9.6) is equivalent to minimizing ($\|\boldsymbol{\beta}\|$), subject to the same constraints.

Due to the fact that $\|\boldsymbol{\beta}\|$ is naturally nonnegative and that $\frac{\|\boldsymbol{\beta}\|^2}{2}$ is monotone increasing for $\|\boldsymbol{\beta}\| \geq \mathbf{0}$, we can now reformulate the optimization problem given in (9.6) as

$$\underbrace{\text{minimize}}_{\beta_0,\beta_1,\beta_2,\ldots,\beta_p} \frac{1}{2}\|\boldsymbol{\beta}\|^2 \tag{9.7}$$

$$y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) \geq 1, \quad i = 1, \ldots, n \tag{9.8}$$

To be able to solve the optimization problem, it is important to understand the Wolfe dual result, which is explained below. Also, remember that $\frac{1}{2}\|\boldsymbol{\beta}\|^2 = \frac{1}{2}\boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{\beta}$.

### 9.3.2 Wolfe Dual

Assume we have the following general optimization problem:

$$\underbrace{\text{minimize}}_{x} f(x) \quad x \in \mathbf{R}^n \tag{9.9}$$

$$\text{subject to } h_i(x) = 0 \;\; i = 1, \ldots, n \tag{9.10}$$

$$g_i(x) \geq 0, \quad i = 1, \ldots, p \tag{9.11}$$

Assume that we are searching for the minimization value of $f(x)$ in an $n$-dimensional space with $m$ equality constraints and $p$ inequality constraints. The Wolfe dual of this optimization problem is

$$\underbrace{\text{maximize}}_{x,\lambda,\mu} f(x) - \sum\nolimits_{i=1}^{m} \lambda_i h_i(x) - \sum\nolimits_{i=1}^{p} \alpha_i g_i(x) \tag{9.12}$$

$$\text{subject to} \quad \nabla f(x) - \sum\nolimits_{i=1}^{m} \lambda_i \nabla h_i(x) - \sum\nolimits_{i=1}^{p} \alpha_i \nabla g_i(x) = 0 \tag{9.13}$$

$$\alpha_i \geq 0, \quad i = 1, \ldots, p \tag{9.14}$$

This changes the searching space to an $(n + m + p)$-dimensional space, $x$, $\lambda$, $\alpha$, with $p + 1$ constraints. The Wolfe dual is a type of Lagrange dual problem. It is important to point out that the sign of the equality constraint does not matter, and we may define it as addition or subtraction, as we wish. However, the sign of the inequality constraint is crucial and should be negative for minimization and positive for maximization.

**Illustrative Example 9.1**

$$\underbrace{\text{minimize}}_{x} x^2 \tag{9.15}$$

$$\text{subject to} \quad x \geq 1 \tag{9.16}$$

Its dual version according to Wolfe is equal to

$$\underbrace{\text{maximize}}_{x,\alpha} f(x,\alpha) = x^2 - 2\alpha(x-1) \tag{9.17}$$

$$\text{subject to} \quad \frac{\partial f(x,\alpha)}{\partial x} = 2x - 2\alpha = 0$$
$$\text{and } \alpha \geq 0 \tag{9.18}$$

Then the last version of the Wolfe dual can be simplified as

$$\underbrace{\text{maximize}}_{\alpha} L(\lambda) = -\alpha^2 + 2\alpha \tag{9.19}$$

$$\text{subject to } \alpha \geq 0 \tag{9.20}$$

With this last version of the Wolfe dual, we obtained the solution to the original optimization problem with the solution for $x = 1$ and $\alpha = 1$.

**Illustrative Example 9.2**

$$\underbrace{\text{minimize}}_{x,y} x^2 + y^2 \tag{9.21}$$

$$\textit{subject to} \qquad x + y \geq 2 \tag{9.22}$$

Its dual version according to Wolfe is equal to

$$\underbrace{\text{maximize}}_{x,y,\alpha} f(x,y,\alpha) = x^2 + y^2 - 2\alpha(x+y-2) \tag{9.23}$$

$$\text{subject to} \quad \frac{\partial f(x,y,\alpha)}{\partial x} = 2x - 2\alpha = 0$$
$$\frac{\partial f(x,y,\alpha)}{\partial y} = 2y - 2\alpha = 0 \tag{9.24}$$
$$\text{and } \alpha \geq 0$$

The last version of the Wolfe dual can be simplified by replacing $x = y = \alpha$ in the dual version, and we obtained:

$$\underbrace{\text{maximize}}_{\alpha} L(\alpha) = -2\alpha^2 + 4\alpha \tag{9.25}$$

$$\text{subject to } \alpha \geq 0 \tag{9.26}$$

With this last version of the Wolfe dual, we obtained the solution to the original optimization problem with the solution for $x = y = 1$ and $\alpha = 1$.

Now that we understand the Wolfe dual result and how to use it to obtain optimal values from optimization problems, we will solve the optimization problem given in (9.7) and (9.8). First, we present its Wolfe dual version (maximization problem), which is equal to

$$L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{\beta}\|^2 - \sum_{i=1}^{n} \alpha_i \left[ y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) - 1 \right], \tag{9.27}$$

where $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)^{\mathrm{T}}$ and the auxiliary nonnegative variables $\alpha_i$ for $i = 1, 2, \ldots, n$ are called *Lagrange multipliers*. Setting the derivatives of $L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha})$ with regard to $\boldsymbol{\beta}$ and $\beta_0$ equal to zero, we obtain the following conditions:

$$\frac{\partial L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha})}{\partial \boldsymbol{\beta}} = \boldsymbol{\beta} - \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i = 0 \Rightarrow \boldsymbol{\beta} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i \tag{9.28}$$

$$\frac{\partial L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha})}{\partial \beta_0} = \sum_{i=1}^{n} \alpha_i y_i = 0 \Rightarrow \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{9.29}$$

$$\alpha_i \left[ y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) - 1 \right] = 0 \text{ for } i = 1, \ldots, n \Rightarrow \alpha_i = 0 \text{ and } y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) = 1 \tag{9.30}$$

The conditions that the solution must satisfy are called the Karush–Kuhn–Tucker conditions. They are required to ensure that the function is convex to guarantee a local optimum of nonlinear programming problems.

We can see from (9.30) that

(a) If $\alpha_i > 0$, then $y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) = 1$, or in other words, $\boldsymbol{x}_i$ is on the boundary of the slab.
(b) If $y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) > 1$, $\boldsymbol{x}_i$ is not on the boundary of the slab, and $\alpha_i = 0$.

From (9.28), we can see that the beta coefficients (with the exception of the intercept) of the maximum margin hyperplane problem are a linear combination of the training vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$. A vector $\boldsymbol{x}_i$ belongs to that expansion if, and only if, $\alpha_i \neq 0$ and these vectors are called support vectors. By condition (9.30), if $\alpha_i \neq 0$, then $y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) = 1$. Thus, support vectors lie on the marginal hyperplane $y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) = \pm 1$.

The maximum margin hyperplane is fully defined by support vectors. The definition of these hyperplanes is not affected by vectors that are not lying on the marginal hyperplanes, since in their absence, the solution for the maximum margin hyperplane remains unchanged.

By placing solutions (9.28) and (9.29) back into $L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\alpha})$, we obtain the Wolfe dual simplified version (maximization) of the optimization problem:

$$L(\boldsymbol{\alpha}) = \underbrace{\frac{1}{2}\left\|\sum_{i=1}^{n}\alpha_i y_i \boldsymbol{x}_i\right\|^2 - \sum_{i=1}^{n}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right)}_{-0.5\times\sum_{i=1}^{n}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right)} - \sum_{i=1}^{n}\alpha_i y_i \beta_0 + \sum_{i=1}^{n}\alpha_i \tag{9.31}$$

Simplifying (9.31) leads to the dual optimization problem for the maximum margin classifier

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = \sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right) \tag{9.32}$$

$$\text{subject to}: \alpha_i \geq 0 \text{ and } \sum_{i=1}^{n}\alpha_i y_i = 0 \text{ for } i = 1, \ldots, n \tag{9.33}$$

The dual problem that needs to be maximized in (9.32) and (9.33) for the maximum margin classifier is cast entirely in terms of the training data and depends only on dot (inner) products of data vectors, $\boldsymbol{x}_i$, $\boldsymbol{x}_j$, and not on the vectors themselves. The operation $\boldsymbol{x}_i \cdot \boldsymbol{x}_j$ denotes the dot product of vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. This means that we do not exactly need the exact data points, but only their inner products to compute our decision boundary. What it implies is that if we want to transform our existing data into a higher dimensional data, which in many cases helps us classify better (see the image below for an example), we need not compute the exact transformation of our data, we just need the inner product of our data in that higher dimensional space.

It is important to point out that the constraints in (9.33) are affine and convex. Also, (9.32) is infinitely differentiable and its Hessian is positive semi-definite which implies that the maximization problem in (9.32) and (9.33) is equivalent to a convex optimization problem. For these reasons, the maximum margin hyperplane provides a unique solution to the separating hyperplane problem and in general does a good job of classifying the testing data due to the fact that the maximization of the margin between the two classes is optimal. Therefore, the dual optimization problem has the following two advantages: (a) there is no need to access the original data, only the dot products and (b) the number of free parameters is bound by the number of support vectors and not by the number of variables (beneficial for high-dimensional problems).

Since $L(\boldsymbol{\alpha})$ is a quadratic function of $\boldsymbol{\alpha}$, this dual optimization problem is a quadratic problem, and standard quadratic programming solvers can be used to obtain the optimal solution for the maximum margin classifier. Once the optimization problem is solved and the values of $\boldsymbol{\alpha}$ are found, we proceed to obtain $\widehat{\boldsymbol{\beta}} = \sum_{i=1}^{n}\alpha_i y_i \boldsymbol{x}_i$. Then we obtain the value of the intercept $\beta_0$ by the fact that any support vector $\boldsymbol{x}_i$ satisfies $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = 1$, that is,

$$y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = y_i\left(\beta_0 + \sum_{j\in S}\alpha_j y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right)\right) = 1,$$

where the set of indices of the support vectors is denoted as $S$. Although we can solve this equation for the intercept $\beta_0$ using an arbitrarily chosen support sector $\boldsymbol{x}_i$, a numerically more stable solution is obtained by first multiplying by $y_i$ making use of $y_i^2 = 1$, and then averaging this equation over all support vectors and solving for $\beta_0$, which gives

$$\beta_0 = \frac{1}{N_S} \sum_{i \in S} \left( y_i - \sum_{j \in S} \alpha_j y_j (\boldsymbol{x}_i . \boldsymbol{x}_j) \right),$$

where $N_S$ is the total number of support vectors. The maximum margin classifier produces a function $\widehat{f}(\boldsymbol{x}_i) = \widehat{\beta}_0 + \boldsymbol{x}_i^{\mathrm{T}} \widehat{\boldsymbol{\beta}}$ that can be used to classify training and testing observations as

$$\widehat{y}_i = sign\left[\widehat{f}(\boldsymbol{x}_i)\right]$$

Due to the construction of this method, none of the training observations falls in the margin, but for testing observations this is not guaranteed. It is expected that the larger the margin in the training data, the better the classification for testing observations. This method is quite robust to misclassification of testing observations because its construction focuses only on the fraction of points that count (support points), and those that have $\alpha_i > 0$ for $i = 1, \ldots, n$, but of course, finding those support points requires using all the training data.

**Example 9.1 A Hand Computation of the Maximum Margin Classifier**
Let the data points and labels be as follows:

$$\boldsymbol{X} = \begin{bmatrix} 0.5 & 1 \\ -0.5 & 1 \\ -0.5 & -1 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix}, \quad \boldsymbol{Q} = \begin{bmatrix} -0.5 & -1 \\ 0.5 & -1 \\ -0.5 & -1 \end{bmatrix}$$

The matrix $\boldsymbol{Q}$ on the right incorporates the class labels, i.e., the rows are $x_i y_i$. Then

$$\boldsymbol{Q}\boldsymbol{Q}^{\mathrm{T}} = \begin{bmatrix} 1.25 & 0.75 & 1.25 \\ 0.75 & 1.25 & 0.75 \\ 1.25 & 0.75 & 1.25 \end{bmatrix}$$

The dual optimization problem is thus

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = \alpha_1 + \alpha_2 + \alpha_3 - \frac{1}{2}\begin{pmatrix} 1.25\alpha_1^2 + 0.75\alpha_1\alpha_2 + 1.25\alpha_1\alpha_3 + 0.75\alpha_2\alpha_1 \\ +1.25\alpha_2^2 + 0.75\alpha_2\alpha_3 + 1.25\alpha_3\alpha_1 + 0.75\alpha_3\alpha_2 + 1.25\alpha_3^2 \end{pmatrix}$$

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = \alpha_1 + \alpha_2 + \alpha_3 - \frac{1}{2} \times \left(1.25\alpha_1^2 + 1.5\alpha_1\alpha_2 + 2.5\alpha_1\alpha_3 + 1.25\alpha_2^2 + 1.5\alpha_2\alpha_3 + 1.25\alpha_3^2\right)$$

Subject to $a_1 \geq 0, a_2 \geq 0, a_3 \geq 0$ and since $\sum_{i=1}^{3} \alpha_i y_i = 0$, then $-\alpha_1 - \alpha_2 + \alpha_3 = 0$, which is equivalent to $\alpha_3 = \alpha_1 + \alpha_2$. While in practice such problems are solved by delicate quadratic optimization solvers, here we will show how to solve this toy problem by hand.

Using the equality constraint, we can eliminate one of the variables, say $\alpha_3$, and simplify the objective function to

$$\underbrace{\text{maximize}}_{\alpha} L(\boldsymbol{\alpha}) = -\frac{1}{2}\left(1.25\alpha_1^2 + 1.5\alpha_1\alpha_2 + 2.5\alpha_1(\alpha_1 + \alpha_2) + 1.25\alpha_2^2 + 1.5\alpha_2(\alpha_1 + \alpha_2) + 1.25(\alpha_1 + \alpha_2)^2\right) + 2\alpha_1 + 2\alpha_2$$

$$\underbrace{\text{maximize}}_{\alpha} L(\boldsymbol{\alpha}) = -\frac{1}{2}\left(5\alpha_1^2 + 8\alpha_1\alpha_2 + 4\alpha_2^2\right) + 2\alpha_1 + 2\alpha_2$$

By setting partial derivatives to 0, we obtain $-5\alpha_1 - 4\alpha_2 + 2 = 0$ and $-4\alpha_1 - 4\alpha_2 + 2 = 0$ (notice that, because the objective function is quadratic, these equations are guaranteed to be linear). We therefore obtain the solution $\alpha_1 = 0$ and $\alpha_2 = \alpha_3 = 0.5$. Recall that $\widehat{\boldsymbol{\beta}} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i = \boldsymbol{X}^{\mathbf{T}}\boldsymbol{Z}$, where $\boldsymbol{Z}^{\mathbf{T}} = (\boldsymbol{\alpha} \circ \boldsymbol{y})^{\mathbf{T}} = [-1, -1, 1][0, 0.5, 0.5] = [0, -0.5, 0.5]$, and $^\circ$ represents the cell-by-cell product between matrices or vectors.

Therefore,

$$\widehat{\boldsymbol{\beta}} = \boldsymbol{X}^{\mathbf{T}}\boldsymbol{Z} = \begin{bmatrix} 0.5 & -0.5 & -0.5 \\ 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 0 \\ -0.5 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

Next, we will calculate the intercept, $\beta_0 = \frac{1}{N_S}\sum_{i \in S}\left(y_i - \sum_{j \in S}\alpha_j y_j (\boldsymbol{x}_i . \boldsymbol{x}_j)\right)$, but first we calculate

$$\boldsymbol{y}^{*}-\boldsymbol{X}^{*}\boldsymbol{X}^{*\mathbf{T}}\boldsymbol{Z}^{*}=\begin{bmatrix}-1\\1\end{bmatrix}-\begin{bmatrix}-0.5&1\\-0.5&-1\end{bmatrix}\begin{bmatrix}-0.5&-0.5\\1&-1\end{bmatrix}\begin{bmatrix}-0.5\\0.5\end{bmatrix}=\begin{bmatrix}-1\\1\end{bmatrix}$$
$$-\begin{bmatrix}1.25&-0.75\\-0.75&1.25\end{bmatrix}\begin{bmatrix}-0.5\\0.5\end{bmatrix}=\begin{bmatrix}-1\\1\end{bmatrix}-\begin{bmatrix}-1\\1\end{bmatrix}=\begin{bmatrix}0\\0\end{bmatrix}$$

$\boldsymbol{y}^*$ is equal to $\boldsymbol{y}$ but without the rows for those *Lagrange multipliers* ($\alpha_i$) that are equal to zero. $\boldsymbol{X}^*$ is equal to $\boldsymbol{X}$ but without those $\alpha_i$ that are equal to zero, and $\boldsymbol{Z}^*$ is equal to $\boldsymbol{Z}$ but without those $\alpha_i$ that are equal to zero. $N_S = 2$ since only one $\alpha_i$ is equal to zero, and this was observation 1. Therefore, $\beta_0$ is

$$\beta_0 = \frac{1}{2}(0+0) = 0$$

Next we calculate the $\widehat{f}(\boldsymbol{x}_i)$ values

$$\begin{bmatrix}\widehat{f}(\boldsymbol{x}_1)\\\widehat{f}(\boldsymbol{x}_2)\\\widehat{f}(\boldsymbol{x}_3)\end{bmatrix}=\begin{bmatrix}0.5&1\\-0.5&1\\-0.5&-1\end{bmatrix}\begin{bmatrix}0\\-1\end{bmatrix}+\begin{bmatrix}0\\0\\0\end{bmatrix}=\begin{bmatrix}-1\\-1\\1\end{bmatrix}$$

Finally, we proceed to calculate the predicted values using $\widehat{y}_i = \text{sign}[\widehat{f(\boldsymbol{x}_i)}]$, then

$$\begin{bmatrix}\widehat{y}_1\\\widehat{y}_2\\\widehat{y}_3\end{bmatrix}=\begin{bmatrix}\text{sign}\left[\widehat{f}(\boldsymbol{x}_1)\right]\\\text{sign}\left[\widehat{f}(\boldsymbol{x}_2)\right]\\\text{sign}\left[\widehat{f}(\boldsymbol{x}_3)\right]\end{bmatrix}=\begin{bmatrix}\text{sign}(-1)\\\text{sign}(-1)\\\text{sign}(1)\end{bmatrix}=\begin{bmatrix}-1\\-1\\1\end{bmatrix}.$$

```
#####Calculations of the hard margin classifier with library e1071####
rm(list=ls())
library(BMTME)
library(e1071)
library(caret)

#########Input data
X1=data.frame(matrix(c(0.5,-0.5,-0.5,1,1,-1), ncol=2))
y1=c(1,1,-1)
dat=data.frame(y=as.factor(y1),x1=X1[,1],x2=X1[,2])

####### Fitting the SVM model with library e1071 #####################
fm1=svm(y=as.factor(y1), x=X1, kernel="linear", scale =F)
ypred=predict(fm1,X1)
Predicted=ypred
```

```
####Useful information that we can extract###########
head(fm1$fitted) ######predicted values of the training data
head(fm1$index) #######index of support vectors
head(fm1$SV,5) ###design matrix of X of support vectors
head(c(fm1$coefs)) ####Coefs=Support vectors*y_i
fm1$rho ##### Extracting the negative value of b (intercept)
Beta=t(fm1$coefs)%*%fm1$SV ###Option 1 for computation of beta
coefficients (weights)
Beta
#####Option 2 for computing beta coefficients
Beta_Coef=t(fm1$coefs)%*%as.matrix(X1)[fm1$index,]
head(Beta_Coef)
Alphas=c(fm1$coefs)*y1[fm1$index]  ### Lagrange multiplier's
coefficients
Alphas

> ####Output of implementing the svm in library
e1071############################
> head(fm1$fitted) ######predicted values of the training data
1 1 -1
> head(fm1$index)#######index of support vectors
[1] 2 3
> head(c(fm1$coefs)) ####Coefs=Support vectors*y_i
[1]  0.5 -0.5
> fm1$rho #####Extracting the negative value of b (intercept)
[1] 0
> #Find value of Beta coefficients=weights
> Beta=t(fm1$coefs)%*%fm1$SV ## Option 1 for computation of beta
coefficients (weights)
> Beta
  X1 X2
[1,]  0  1
> #####Option 2 of beta coefficients calculation
> Beta_Coef=t(fm1$coefs)%*%as.matrix(X1)[fm1$index,]
> head(Beta_Coef)
  X1 X2
[1,]  0  1
> ####### Lagrange multiplier alpha coefficients ############
> Alphas=c(fm1$coefs)*y1[fm1$index]
> Alphas
[1] 0.5 0.5
```

From the output of the svm() function in the e1071 library, we can see that fm1$fitted produced exactly the same predictions as those we obtained with hand computation. Also, the hand computation and the index of the output of the svm() function, as fm1$index, agree that the indices of the support vector are observations 2 and 3 since observation 1 is equal to zero. The coefficients that result from the product of the Lagrange multipliers with the response variable, $\alpha_j y_j$, also agree. Also, we obtained the same intercept using hand calculation as that extracted from the fitted model with the svm() function of the e1071 library. We found agreement between the at hand computation and the results of the e1071 library for the Lagrange multipliers.

## 9.4 Derivation of the Support Vector Classifier

The method just studied does a good job when the data are linearly separable, but what can we do when the data are not linearly separable? The solution is to create a *soft margin classifier* that allows some points to fall on the incorrect side of the margin by using slack variables ($\zeta_i$). Adding slack variables to the optimization problem allows some points to be on the wrong side of the margin and, consequently, to be misclassified (James et al. 2013). In Fig. 9.6 there are two points that fall on the wrong side of the boundary line with the corresponding slack term denoted as $\zeta_i$ (James et al. 2013).

The $\zeta_i$ called slack variables are used in optimization problems to define relaxed versions of some constraints. The slack variable, $\zeta_i$, measures the distance by which vector $\boldsymbol{x}_i$ violates the established inequality, $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) \geq 1$. For a hyperplane $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = 0$, an $\boldsymbol{x}_i$ vector with $\zeta_i > 0$ can be viewed as an outlier. Each $\boldsymbol{x}_i$ must be positioned on the correct side of the appropriate marginal hyperplane so as not to be considered an outlier. This implies that those vectors that fall between $0 < y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) < 1$ are correctly classified by the hyperplane $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = 0$ and are no longer considered outliers. By omitting the outlier observations, the training data can be classified correctly by hyperplane $y_i\left(\beta_0 + \boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right) = 0$ with margin $M = \|\boldsymbol{\beta}\|^{-1}$, which is called the *soft margin*, as opposed to the separable case that we call the *hard margin* classifier. For this reason, the soft margin classifier is more robust to individual observations and does a better job classifying the training and testing observations. However, this method does not guarantee that every observation is on the correct side of the margin and hyperplane since it allows some observations to be on the incorrect side of the margin or hyperplane. It is from this property that this method takes its name since the margin is *soft* in the sense that it can be violated by some of the training observations. As mentioned above, an observation can be not only on the wrong side of the hyperplane but also on the incorrect side of the margin.

Then the question of interest is: how to select the hyperplane in the non-separable case? One option is to choose the hyperplane with minimum empirical error. However, this option does not guarantee that a large margin can be found, and for choosing the right hyperplane, we need to find: (a) a balance between the limit of the total amount of slack due to outliers, measured as $\sum_{i=1}^{n}\zeta_i$ and (b) a hyperplane with a large margin, but if the margin is larger, more outliers are possible, which implies a larger amount of slack. The optimization problem now consists of finding a hyperplane that is able to classify most of the training observations in the two classes; this can be accomplished by obtaining the solution to the following optimization problem:

$$\underbrace{\text{maximize}}_{\beta_0,\beta_1,\beta_2,\ldots,\ \beta_p,\zeta_1,\ldots,\zeta_n} \quad M \tag{9.34}$$

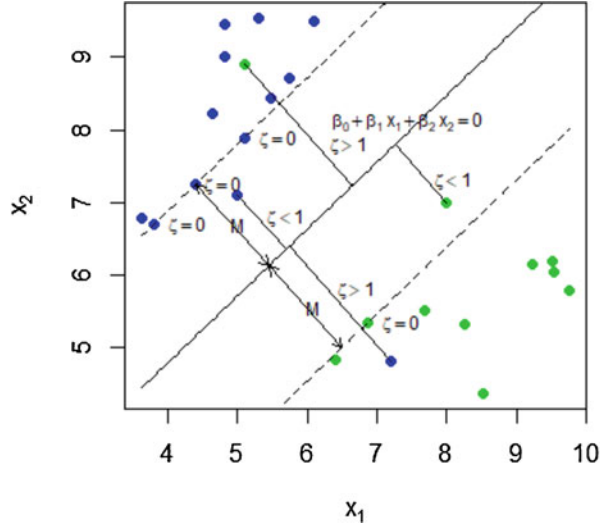



**Fig. 9.6** Soft margin support vector machine in non-separable data training. Dots with $0 \le \zeta_i \le 1$ or not labeled are correctly classified, while those with $\zeta_i > 1$ are on the wrong side of the decision boundary and incorrectly classified

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1, \tag{9.35}$$

$$y_i\left(\beta_0 + \sum_{j=1}^{p} \beta_{j1} x_{ij}\right) \ge M(1 - \zeta_i), \tag{9.36}$$

$$\zeta_i \ge 0, \sum_{i=1}^{n} \zeta_i \le T, \tag{9.37}$$

where $\beta_0, \beta_1, \beta_2, \ldots, \beta_p$ are the coefficients of the maximum margin hyperplane. $T$ is a nonnegative tuning parameter that determines the number and severity of the violations to the margin (and to the hyperplane) that we will tolerate, and it is seen as the total amount of errors allowed since it is the bound of the sum of $\zeta_i$'s. $T$ is like a budget for the amount that the margin can be violated by the $n$ observations. For $T$ close to zero, the soft-margin SVM allows very little error and is similar to the hard-margin classifier (James et al. 2013). The larger $T$ is, the more error is allowed, which in turn allows for wider margins. These parameters play a key role in controlling the bias-variance trade-off of this statistical learning method. In practice, $T$ is a hyperparameter that needs to be tuned, for example, by using cross-validation. $M$ is the width of the margin and we seek to make this quantity as large as possible. In (9.37), $\zeta_1, \ldots, \zeta_n$ are slack (error) variables that allow individual observations to be on the wrong side of the margin or the hyperplane. The slack variable $\zeta_i$ tells us where the $i$th observation is located, relative to the hyperplane and relative to the margin. If $\zeta_i = 0$, then the $i$th observation is on the correct side of the margin. If

$\zeta_i > 0$, then the $i$th observation is on the wrong side of the margin, and this means that the $i$th observation has broken the margin. If $\zeta_i > 1$, then it is on the wrong side of the hyperplane. If $T = 0$, this implies that no budget is available for violations to the margin, and it must be that $\zeta_1 = \ldots = \zeta_n = 0$, in which case the optimization problem is equal to that of the maximum margin hyperplane. The larger the budget $T$, the wider the margin and the larger the number of support vectors, which means that we are more tolerant of violations to the margin. In contrast, the lower the $T$, the narrower the margin and fewer support vectors are selected, which means less tolerance of violations to the margin. Similar to the maximum margin classifier, only the support vectors (observations that lie on the margin) and observations that violate the margin affect the hyperplane and the resulting classifier. However, all observations that lie strictly on the correct side of the margin do not affect the support vector classifier.

Since this method is based only on a small fraction of the training observations (support vectors), it is quite robust to the classification of new observations that are far away from the hyperplane. Once we solve (9.34)–(9.37), we classify a test observation $x^*$ by simply determining on which side of the hyperplane it lies. That is, we classify the test observation in the training/testing sets based on the sign of $f(x^*) = \widehat{\beta}_0 + \widehat{\beta}_1 x_1^* + \widehat{\beta}_2 x_2^* + \ldots + \widehat{\beta}_p x_p^*$; if $f(x^*) < 0$, then the observation is assigned to the class corresponding to $-1$, but if $f(x^*) > 0$, then the observation is assigned to the class corresponding to 1 (James et al. 2013), which is exactly as in the hard margin classification method described before.

Next, we present the Wolfe primal version (for minimization) of the support vector classifier, which is equal to

$$L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\epsilon}, \boldsymbol{\alpha}, \boldsymbol{\delta}) = \frac{1}{2} \|\boldsymbol{\beta}\|^2 - T\sum_{i=1}^{n} \zeta_i - \sum_{i=1}^{n} \alpha_i \left[ y_i \left( \beta_0 + \boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{\beta} \right) - 1 + \zeta_i \right] + \sum_{i=1}^{n} \delta_i \zeta_i, \tag{9.38}$$

where $\boldsymbol{\zeta} = (\zeta_1, \ldots, \zeta_n)^{\mathrm{T}}$, $\boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_n)^{\mathrm{T}}$, $\delta_i > 0$ for $i = 1, \ldots, n$ associated with the nonnegativity constraints of the slack variables, $\boldsymbol{\delta} = (\delta_1, \ldots, \delta_n)^{\mathrm{T}}$. By setting the derivatives of $L = L(\boldsymbol{\beta}, \beta_0, \boldsymbol{\zeta}, \boldsymbol{\alpha}, \boldsymbol{\delta})$ with regard to $\boldsymbol{\beta}$, $\beta_0$, and $\boldsymbol{\zeta}$ equal to zero, we obtain the following conditions:

$$\frac{\partial L}{\partial \boldsymbol{\beta}} = \boldsymbol{\beta} - \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i = 0 \;\Rightarrow\; \boldsymbol{\beta} = \sum_{i=1}^{n} \alpha_i y_i \boldsymbol{x}_i \tag{9.39}$$

$$\frac{\partial L}{\partial \beta_0} = -\sum_{i=1}^{n} \alpha_i y_i = 0 \Rightarrow \; \sum_{i=1}^{n} \alpha_i y_i = 0 \tag{9.40}$$

$$\frac{\partial L}{\partial \epsilon_i} = T - \alpha_i - \delta_i = 0 \Rightarrow \; \alpha_i + \delta_i = T \tag{9.41}$$

$$\alpha_i\left[y_i\left(\beta_0+\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right)-1+\zeta_i\right]=0 \text{ for } i=1,\ldots,n \Rightarrow \alpha_i = 0 \text{ and } y_i\left(\beta_0+\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{\beta}\right)=1-\zeta_i \tag{9.42}$$

$$\delta_i\zeta_i=0 \text{ for } i=1,\ldots,n \Rightarrow \delta_i=0 \text{ and } \zeta_i=0 \tag{9.43}$$

By placing solutions (9.39)–(9.43) back into $L=L(\boldsymbol{\beta},\beta_0,\boldsymbol{\zeta},\boldsymbol{\alpha},\boldsymbol{\delta})$, we obtain the Wolfe dual version (maximization problem) of the optimization problem

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha})=\sum\nolimits_{i=1}^{n}\alpha_i-\frac{1}{2}\sum\nolimits_{i=1}^{n}\alpha_i\alpha_j y_i y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right) \tag{9.44}$$

$$\text{subject to}: \quad 0\le\alpha_i\le T \text{ and } \sum\nolimits_{i=1}^{n}\alpha_i y_i=0 \text{ for } i=1,\ldots,n \tag{9.45}$$

This problem is very similar to the one in the previous section and, again, it is a convex quadratic programming problem that can be solved using conventional quadratic programming software since the objective function is concave and infinitely differentiable.

Again, the solution to $\boldsymbol{\alpha}$ in (9.44) can be used to make the predictions as follows:

$$\widehat{y}_i=\operatorname{sign}\left(\sum\nolimits_{i=1}^{N_S}\widehat{\alpha}_j y_j\left(\boldsymbol{x}_j.\boldsymbol{x}\right)+\widehat{\beta}_0\right),$$

where $N_S$ is the total number of support vectors lying on a marginal hyperplane, that is, those vectors $\boldsymbol{x}_i$ with $0\le\alpha_i\le T$ and $\widehat{\beta}_0=\frac{1}{N_S}\sum\limits_{i\in S}\left(y_i-\sum\limits_{j\in S}\widehat{\alpha}_j y_j\left(\boldsymbol{x}_i.\boldsymbol{x}_j\right)\right)$.

The predicted values depend only on the inner products between vectors and not directly on the vectors themselves; this fact is the key for expanding this method to define nonlinear decision boundaries.

## 9.5 Support Vector Machine

When the data are linearly inseparable in a low-dimensional space, it is possible to separate them in a higher dimensional space. For example, when all data points within a circle in a two-dimensional space belong to one class, those outside the circle belong to another class. In this case, it is not possible to use a straight line to separate the two classes, but by adding two more features, $x_1^2$, $x_2^2$, it is possible to do so, as can be seen in Fig. 9.7.

Figure 9.8 provides another example of a nonlinear problem that can be mapped to a linear problem by using a nonlinear transformation, $\varphi$, of the input data.

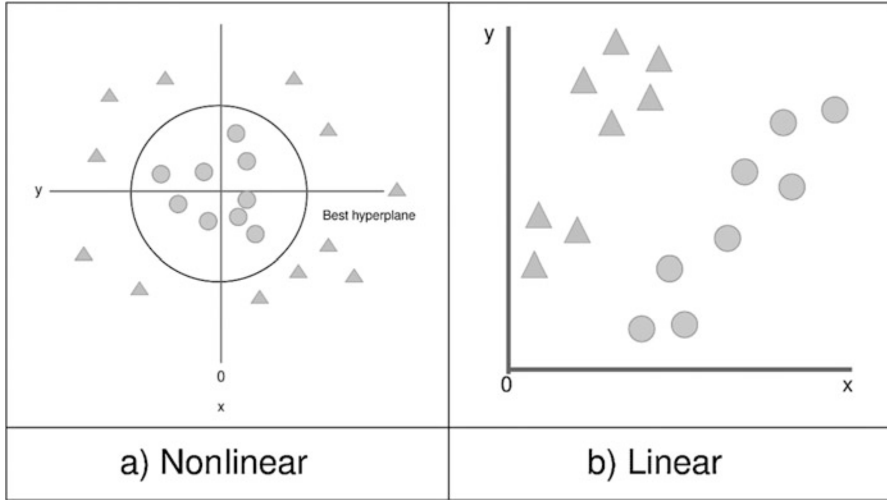



**Fig. 9.7** Transforming a nonlinear problem into a linear one

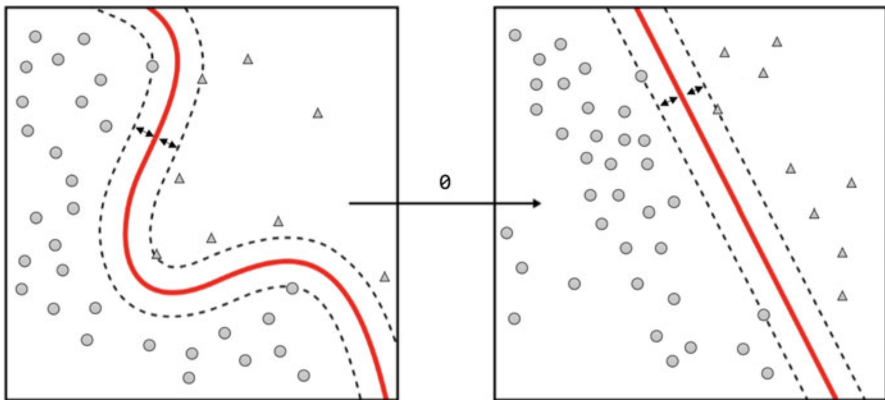
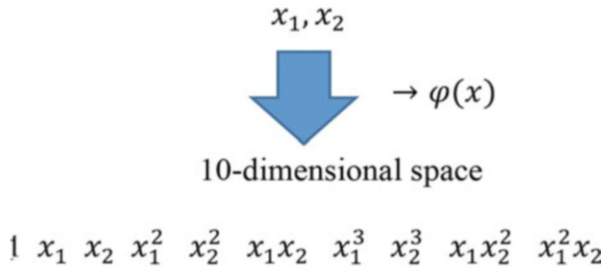



**Fig. 9.8** Transforming a complex nonlinear problem into a linear one

To see better how the transformation is implemented to expand the original input feature, we assume that we are dealing with two-dimensional data (i.e., in $\mathbb{R}^2$) and we will expand the input data using a polynomial kernel $(\boldsymbol{x}_i . \boldsymbol{x}_j + 1)^3$. The following illustration shows how this kernel maps the data.




$1 \ x_1 \ x_2 \ x_1^2 \ x_2^2 \ x_1x_2 \ x_1^3 \ x_2^3 \ x_1x_2^2 \ x_1^2x_2$

This means that if such a linear decision surface does exist, the data are mapped into a much higher dimensional space ("feature space") where the separating decision surface is found, and the feature space is constructed via a very smart statistical projection ("kernel trick") studied in detail in the previous chapter.

This means that the construction of a higher dimensional space is done in general terms as

$$\boldsymbol{x} \rightarrow \varphi(\boldsymbol{x}).$$

That is, training input samples are transformed into a feature space using a nonlinear function $\varphi(.)$.

**Kernel functions** We define a kernel function as being a real-valued function of two arguments, $K(\boldsymbol{x}, \boldsymbol{x}^{\mathrm{T}}) \in \mathbb{R}$, for $\boldsymbol{x}$, $\boldsymbol{x}^{\mathrm{T}} \in \mathbb{R}$. The function is typically symmetric (i.e., $K(\boldsymbol{x}, \boldsymbol{x}^{\mathrm{T}}) = K(\boldsymbol{x}^{\mathrm{T}}, \boldsymbol{x})$) and nonnegative (i.e., $K(\boldsymbol{x}, \boldsymbol{x}^{\mathrm{T}}) \geq 0$), so it can be interpreted as a measure of similarity, but this is not required.

By making the following substitution, we can build an optimization problem in the new space:

$$\boldsymbol{x}_i^{\mathrm{T}} \boldsymbol{x}_j \rightarrow \varphi(\boldsymbol{x}_i)^{\mathrm{T}} \varphi(\boldsymbol{x}_j).$$

This implies that the nonlinear *support vector machine* (SVM) is trained with the inner product $\varphi(\boldsymbol{x}_j)^{\mathrm{T}} \varphi(\boldsymbol{x}_j)$ as long as this inner product is known, which means that it does not matter if $\varphi(\boldsymbol{x}_j)$ is known. By using a kernel function, the kernel trick directly specifies the inner product:

$$\varphi(\boldsymbol{x}_i)^{\mathrm{T}} \varphi(\boldsymbol{x}_j) \rightarrow K(\boldsymbol{x}_i, \boldsymbol{x}_j)$$

Thanks to the kernel trick, the computational cost of training the SVM is independent of the dimensionality of the feature space. Some of the most popular kernels were described in the previous chapter and are: linear, polynomial, sigmoid,

Gaussian (radial), exponential, and arc-cosine (AK) with different numbers of hidden layers.

As pointed out above, the SVM kernel only needs the information of the kernel value $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$, assuming that this has been defined as was exemplified in the previous chapter. For this reason, nonvectorial patterns $\boldsymbol{x}$ such as sequences, trees, and graphs can be handled. It is important to point out that the kernel trick can be applied in unsupervised methods like cluster analysis and dimensionality reduction methods like principal component analysis, independent component analysis, etc.

The SVM is an extension of the *support vector classifier* when enlarging the feature space using kernels (James et al. 2013). This is possible thanks to the fact that the solution of the dual optimization problem for the support vector classifier does not directly depend on the input vectors but only on the inner products. Since positive definite symmetric (PDS) kernels implicitly define an inner product, we can extend the support vector classifier and combine it with an arbitrary PDS kernel $K$, by replacing each instance of an inner product $\boldsymbol{x}_i.\, \boldsymbol{x}_j$, with $K(\boldsymbol{x}_i, \boldsymbol{x}_j)$. This leads to a general form of the support vector classifier that is called SVM, which is the solution to the following optimization problem:

$$\underbrace{\text{maximize}}_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) = \sum\nolimits_{i=1}^{n} \alpha_i - \frac{1}{2}\sum\nolimits_{i=1}^{n} \alpha_i \alpha_j y_i y_j K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) \tag{9.46}$$

$$\text{subject to}: \quad 0 \leq \alpha_i \leq T \text{ and } \sum\nolimits_{i=1}^{n} \alpha_i y_i = 0 \text{ for } i = 1, \ldots, n \tag{9.47}$$

Again, we classify the test observation in the training/testing sets based on the sign of $f(\boldsymbol{x}) = \sum_{i=1}^{N_S} \widehat{\alpha}_i y_i K(\boldsymbol{x}_i, \boldsymbol{x}) + \widehat{\beta}_0 = (\widehat{\boldsymbol{\alpha}}^{\circ} \boldsymbol{y})^{\mathrm{T}} K(\boldsymbol{x}_i, \boldsymbol{x}) + \widehat{\beta}_0$; if $f(\boldsymbol{x}) < 0$, then the observation is assigned to the class corresponding to $-1$, but if $f(\boldsymbol{x}) > 0$, then the observation is assigned to the class corresponding to 1 (James et al. 2013). Also, $\widehat{\beta}_0 = \frac{1}{N_S} \sum_{i \in S} \left( y_i - \sum_{j \in S} \widehat{\alpha}_j y_j K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) \right)$, where $N_S$ is the total number of support vectors lying on a marginal hyperplane and $\sum_{j \in S} \widehat{\alpha}_j y_j K\left(\boldsymbol{x}_i, \boldsymbol{x}_j\right) = (\widehat{\boldsymbol{\alpha}}^{\circ} \boldsymbol{y})^{\mathrm{T}} \boldsymbol{K} \boldsymbol{e}_i$, where $\boldsymbol{e}_i$ is the $i$th unit vector, therefore $\varphi(\boldsymbol{x}_i) = \boldsymbol{K}\boldsymbol{e}_i$, that is $\varphi(\boldsymbol{x}_i)$ is the $i$th column of $\boldsymbol{K}$, for $i = 1, 2, \ldots, n$. We chose $f(\boldsymbol{x})$ as a nonlinear function of $\boldsymbol{x}$ and the possible kernels are those explained above: linear, polynomial, Gaussian, or sigmoid. With the exception of the linear kernel, all these kernels are nonlinear functions of $\boldsymbol{x}$, but with fewer parameters than quadratic, cubic, or a higher order expansion of $x$.

The SVM can be implemented with the R package e1071 in the R statistical software (R Core Team 2018) with linear, polynomial, Gaussian, and sigmoid kernels. This software also allows implementing the SVM method with ordinal data under the following two approaches:

### 9.5.1 One-Versus-One Classification

When we have categorical (multi-class) data with more than two classes under the *one*-versus-*one* classification approach, we construct $K(K-1)/2$ binary SVMs, each of which compares a pair of classes. Each SVM compares the $k$th class, coded as +1, to the $k$'th class, coded as $-1$. At prediction time, a voting scheme is applied: all $K(K-1)/2$ binary SVMs are applied to an unseen sample and the class that gets the highest number of "+1" predictions gets predicted by the combined classifier (James et al. 2013).

### 9.5.2 One-Versus-All Classification

The one-versus-all approach is an alternative when there are more than two categories ($K > 2$) and consists of fitting $K$ SVMs, each time comparing one of the $K$ classes to the remaining $K-1$ classes, that is, in learning the $k$th classifier we treat all points not in class $k$ as a single not-$k$ class by lumping them all together. To learn each of the two class classifiers, we temporarily assign labels to n training points: observations in class $k$ and not-$k$ are assigned temporary labels +1 and $-1$, respectively. Having done this for all $K$ classes, we then predict the value of $y_i$ for input $\boldsymbol{x}$ by taking

$$\widehat{y}_i = \mathrm{argmax}\left[f(\boldsymbol{x})_k\right] \text{ for } k = 1, 2, \ldots, K,$$

where $f(\boldsymbol{x})_k = \sum_{j=1}^{N_S} \widehat{\alpha}_{jk} y_j K_k\left(\boldsymbol{x}_j, \boldsymbol{x}\right) + \widehat{\beta}_{0k}$. That is, for an $\boldsymbol{x}$ input, we classified the $i$th observation in the class for which $f(\boldsymbol{x})_k$ $k = 1, 2, \ldots, K$ is largest even if this evaluation is negative, since this indicates that we have the highest level of confidence that the test observation belongs to the $k$th class rather than to any of the other classes. In general, SVMs are very competitive when you have a large number of features (independent variables), for example, in genomic selection and in text classification. SVMs with nonlinear kernels perform quite well in most cases and are usually head to head with random forests, that is, sometimes random forests work slightly better and sometimes SVMs win. It is effective when the number of independent variables is greater than the number of observations. However, there are no free lunches and SVMs have their difficulties. They can be computationally expensive at times. SVMs do not perform well with noisy data sets. That being said, one should be careful about when to choose and when not to choose SVM as the classifier to solve the problem at hand.

**Example 9.1 for binary data** For this example, we used the EYT Toy data set composed of 40 lines, four environments (Bed5IR, EHT, Flat5IR, and LHT), and four response variables: DTHD, DTMT, GY, and Height. G_Toy_EYT is the genomic relationship matrix of dimension $40 \times 40$. The first two variables are

ordinal with three categories, the third is continuous (GY = Grain yield) and the last one (Height) is binary. In this example, we work with only the binary response variable (Height).

First we load the data using load("Data_Toy_EYT.RData") using the following code:

```
rm(list=ls())
library(BMTME)
library(e1071)
library(caret)

load("Data_Toy_EYT.RData")
ls()

Gg=data.matrix(G_Toy_EYT)
G=Gg
```

This part of the code gives as output

```
> load("Data_Toy_EYT.RData")
> ls()
[1] "G_Toy_EYT"   "Pheno_Toy_EYT"
```

Here we can see two files: the first is the GRM and the second is the phenotypic information. Then, using data.matrix(G_Toy_EYT), we accommodate the GRM in the Gg object as a matrix.

```
> Gg=data.matrix(G_Toy_EYT)
```

With the next R code, we give a name to the phenotypic information; this is ordered as

```
Data.Final=Pheno_Toy_EYT
Data.Final=Data.Final[order(Data.Final$Env,Data.Final$GID),]
head(Data.Final)
```

The first six observations of this phenotypic information are

```
> head(Data.Final)
      GID   Env DTHD DTMT    GY Height
1 GID6569128 Bed5IR  1   1 6.119272   0
2 GID6688880 Bed5IR  2   2 5.855879   0
3 GID6688916 Bed5IR  2   2 6.434748   0
4 GID6688933 Bed5IR  2   2 6.350670   0
5 GID6688934 Bed5IR  1   2 6.523289   0
6 GID6688949 Bed5IR  1   2 5.984599   0
```

With the following code, we create the design matrices

```
########Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data.Final$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZT=model.matrix(~0+as.factor(Data.Final$Env))
Z2TG=model.matrix(~0+Z1G:as.factor(Data.Final$Env))
```

Then with the next part of the code, we prepare the information to create the folds for implementing a five-fold CV strategy.

```
##########Preparation for building the five-fold CV######
Data.Final_1=Data.Final[,c(1:3)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final$Env))
nCV=5
```

Using the latter information, we created the five-folds using the CV.KFold function of the BMTME package

```
#############Training-testing partitions#####################
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)
```

Then with the next code, we selected Height as the response variable, and we also got the number of rows in the data set

```
##########Selecting the Height, binary output######
y1=Data.Final$Height+1
y2=y1
n=dim(Data.Final)[1]
```

Next, we built the input data to implement the SVM method. To do this, we applied the following code, using only the information of environments and genotypic information of lines; PCCC_Part = c() is for saving the output of each testing fold.

```
##########Concatenating the information for input information####
X1=as.data.frame(cbind(ZT,Z1G))
dim(X1)
PCCC_Part=c()
```

The next code implements the five-fold:

```
for(r in 1:nCV) {
###### a) input, output, and testing set##################
X2=X1
```

```
y1=as.factor(y2)
positionTST=c(CrossV$CrossValidation_list[[r]])

###### b) Training and testing sets######################
X_tr=droplevels(X2[-positionTST,])
X_ts=droplevels(X2[positionTST,])
y_tr=y1[-positionTST] ###Training
y_ts=y1[positionTST]  ###Testing

####### c) Deleting columns with no variance##############
var_x=apply(X_tr,2,var)
length(var_x)
pos_var0=which(var_x>0)
length(pos_var0)
X_tr_New=X_tr[,pos_var0]
X_ts_New=X_ts[,pos_var0]

####### d) Fitting the model with SVM#####################
fm1=svm(y=y_tr,x=X_tr_New)
ypred=predict(fm1,X_ts_New)
Predicted=ypred
Observed=y_ts
xtab <- table(Observed, Predicted)
Conf_Matrix=confusionMatrix(xtab)

####### e) Calculating the accuracy in terms of PCCC#########
PCCC=Conf_Matrix$overall[1]
PCCC_Part=c(PCCC_Part,PCCC)
}
PCCC_Part
mean(PCCC_Part)
```

In part a) of this code, we updated in each fold the input information (X matrix), the output information (y1), and the testing set of each fold. It is important to point out that the output variable (y1), when this is binary or ordinal, it is required to define it as a factor for SVM methods. The outputs and inputs of each fold are obtained in part b) of the code. In part c) of the code, those columns of the input information with zero variance are deleted, since if they are not deleted, the SVM fails to converge. In part d) of the code, the SVM is fitted, where we only provide the training set of the input and output information. By default, the SVM implements the radial basis function, or Gaussian kernel, and also by default, the input is scaled internally with the SVM function. Further, in part d) of the code, the corresponding predictions for the testing set of each fold are obtained. In part e) of the code, the metric PCCC is calculated with the help of the caret package for each fold and saved in PCCC_Part.

Finally, the output in terms of PCCC for each fold is obtained with PCCC_Part, and the average of the five-fold in terms of PCCC is obtained with mean (PCCC_Part). The output of the implementation is given below.

```
> PCCC_Part
Accuracy Accuracy Accuracy Accuracy Accuracy
 0.68750 0.78125 0.84375 0.71875 0.68750
> mean(PCCC_Part)
[1] 0.74375
```

We can see that the highest prediction was obtained in fold 3 with PCCC = 0.84375, while the lowest was observed in folds 1 and 5 with PCCC = 0.68750. Finally, the average of the five-fold was equal to PCCC = 0.74375, which means that 74.375% of the cases were correctly classified in the testing sets. It is important to point out that this result was obtained without taking into account the genotype × environment interaction. The same code can be used taking into account the $G \times E$ by only replacing

$$\mathrm{X1} = \mathrm{as.data.frame(cbind(ZT, Z1G))}$$

with $G \times E$, using

$$\mathrm{X1} = \mathrm{as.data.frame(cbind(ZT, Z1G, Z2TG))}$$

With $G \times E$, the average PCCC = 0.5375, which is 20.625% lower than when the $G \times E$ term is ignored. It is important to point out that to fit a model with the svm() function without the $G \times E$ term, we can implement not only the Gaussian kernel (radial) but also the linear, polynomial, and sigmoid kernels, by only specifying in svm(y = y_tr,x = X_tr_New, kernel = "linear"), the required kernel as linear, polynomial, or sigmoid. The outputs using the four available kernels are given next:

```
> results
      Type   PCCC
1   radial 0.74375
2   linear 0.74375
3 polynomial 0.71250
4  sigmoid 0.74375
```

Here we can see that the PCCC for radial, linear, and sigmoid was 0.74375, and the lowest PCCC was with the polynomial kernel with a value of 0.71250, while with the $G \times E$ interaction term the PCCC were

```
> results
      Type   PCCC
1   radial 0.53750
2   linear 0.55625
3 polynomial 0.51250
4  sigmoid 0.56875
```

Next, we provide the R code for tuning the hyperparameters under the SVM method without taking into account the $G \times E$ interaction term. This code should be used after part ###c) by deleting columns with no variance ###, of the code given above inside the loop, with five-folds, but using only the information of fold = 2. The tuning function requires the method (in this case, an SVM method) to be tuned, and then the training and testing sets. Then we need to specify the type of kernel, which in this case was a linear kernel, and in ranges of the tune function are specified the grid of values to the cost. It is important that, for the linear kernel, only the cost parameter needs to be tuned, but for the radial kernel, the gamma parameter should also be tuned.

```
############Tuning process#####################
obj <- tune(svm, train.y=y_tr,train.x=X_tr_New,kernel="linear",
ranges = list(cost =seq(0.001,0.5,0.005)))
summary(obj)
plot(obj)

Par_cost=as.numeric(obj$best.parameters[1])
Par_cost
bestmod=obj$best.model
bestmod

#########Predictions for the testing set######################
ypred=predict(bestmod,X_ts_New)
Predicted=ypred
Observed=y_ts
xtab <- table(Observed, Predicted)
Conf_Matrix=confusionMatrix(xtab)

PCCC=Conf_Matrix$overall[1]
PCCC
```

Part of the output of this code is given below.

```
> summary(obj)

Parameter tuning of 'svm':

- sampling method: 10-fold cross validation

- best parameters:
 cost
 0.061

- best performance: 0.2423077

- Detailed performance results:
  cost   error dispersion
```

```
1  0.001 0.5698718 0.1057401
2  0.031 0.2653846 0.1101288
3  0.061 0.2423077 0.1124914
4  0.091 0.2576923 0.1033534
5  0.121 0.2980769 0.1116542
6  0.151 0.3057692 0.1029218
7  0.181 0.2903846 0.1245948
8  0.211 0.3057692 0.1259071
9  0.241 0.3057692 0.1259071
10 0.271 0.3134615 0.1333871
11 0.301 0.3134615 0.1333871
12 0.331 0.3134615 0.1333871
13 0.361 0.3134615 0.1333871
14 0.391 0.3134615 0.1333871
15 0.421 0.3134615 0.1333871
16 0.451 0.3134615 0.1333871
17 0.481 0.3217949 0.1373119
```

Here we can see that the lower error is obtained when the cost = 0.061. It is important to point out that these errors are calculated using a ten-fold cross-validation set with the training set of outer fold 2. The plot(obj) resulting from the tuning process is given below.

In Fig. 9.9, again we can see that the minimum average validation error corresponds to a cost value of 0.061, which was extracted with the R code as.numeric (obj$best.parameters[1]). Then we used bestmod = obj$best.model to extract the best model that corresponds to the model with the lower error in the validation set with a cost value of 0.061. Finally, the predictions for the outer testing set were obtained with ypred = predict(bestmod,X_ts_New), where the predictions are performed using the best model of the grid, which in this case has a cost value of 0.061. The resulting prediction in terms of PCCC was 0.71875.

It is important to point out that in the case of nonlinear hyperplanes, a gamma parameter also needs to be tuned. It is expected that the higher the gamma value, the

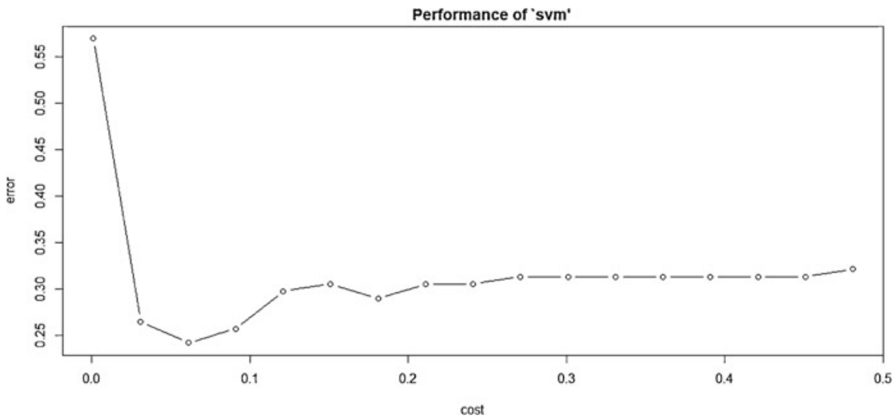



**Fig. 9.9** Average validation error of ten-fold cross-validation for the grid of cost values

better the fit to the training data set; for this reason, many times increasing the gamma parameter leads to overfitting. Now we implemented the SVM by tuning the gamma values and the cost of nonlinear kernels and only the cost for the linear hyperplane (the code we used is in Appendix 1), also ignoring the $G \times E$ interaction term. The results obtained for each type of kernel are given below.

```
> results
  Type       MSE
1 linear     0.74375
2 radial     0.74375
3 polynomial 0.74375
4 sigmoid    0.71875
```

We can see that tuning the parameters did not improve the prediction performance more than when using the default values for these hyperparameters. This means that many times the default values do a good job and that choosing the right values for the tuning process is challenging.

**Example 9.2 for ordinal data** Once again, we used the EYT Toy data set composed of 40 lines, four environments (Bed5IR, EHT, Flat5IR, and LHT), and four response variables: DTHD, DTMT, GY, and Height. But now we worked with the ordinary response variable (DTHD) that has three response options.

Since the data set was the same, the code used for its implementation was the same, but now we worked with the DTHD categorical variable. This means that the key modification was that now we used the DTHD as the response variable, which was chosen using the following code:

```
##########Selecting the DTHD, ordinal output######
y1=Data.Final$DTHD
y2=y1
n=dim(Data.Final)[1]
```

Also, by using five-folds without tuning and ignoring the $G \times E$ interaction term for the four types of kernels, we got the following results:

```
> results
  Type       PCCC
1 radial     0.76875
2 linear     0.65625
3 polynomial 0.76250
4 sigmoid    0.73125
```

Here the best predictions were obtained under the Gaussian or radial kernel with PCCC = 0.76875 and the worst under the linear kernel with PCCC = 0.65625. The complete code for reproducing these results is given in Appendix 2. Next, we

provide the performance also with five-fold cross-validation but tuning the gamma parameters and cost for nonlinear kernels, and only the cost for those with linear kernels.

```
> results
  Type       PCCC
1 linear     0.65625
2 radial     0.76875
3 polynomial 0.76250
4 sigmoid    0.74375
```

Once again, the best predictions were observed under the Gaussian kernel and the worst under the linear kernel; however, there was no improvement when using the default values for gamma and cost. In this example, we saw that implementation of the SVM method for binary or ordinal data with the e1071 library is the same, but taking care that the response variable is defined as a factor. However, this library works for binary data by default, since the machinery for SVM (derivations explained above) was designed for binary data. In the case of ordinal data (multi-class classification), by default, library e1071 implements the "*one*-vs-*one*" approach explained above, where $k(k - 1)/2$ binary classifiers are trained and the appropriate class is found by a voting scheme.

## 9.6 Support Vector Regression

The support vector regression (SVR) is inspired by the support vector machine algorithm for binary response variables. The main idea of the algorithm consists of only using residuals smaller in absolute value than some constant (called $\varepsilon$-sensitivity), that is, fitting a tube of $\varepsilon$ width to the data, as illustrated in Fig. 9.10.

Two sets of points are defined as in binary classification: those falling inside the tube, which are $\varepsilon$-close to the predicted function and thus not penalized, and those falling outside, which are penalized based on their distance from the predicted function, in a way that is similar to the penalization used by SVMs in classification. Due to the fact that the idea behind support vector regression (SVR) is very similar to SVM, which consists of finding a well-fitting hyperplane in a kernel-induced feature space that will have good generalization performance using the original features. For this reason, detailed SVR theory is not covered in this book, but interested readers can find this information in the following references: Burges (1998); Awad and Khanna (2015). Also, there is no agreement that the performance of SVR is better compared to any type of regression machines for predicting continuous outcomes. For this reason, next we will illustrate the implementation of SVR in the e1071 library.

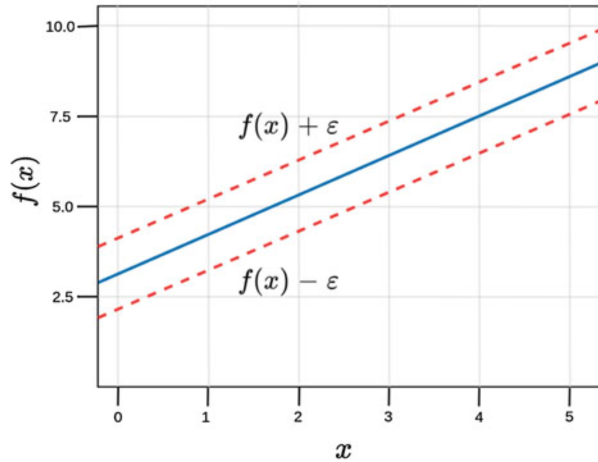



**Fig. 9.10** $\epsilon$-insensitive regression band. The solid blue line represents the estimated regression curve $f(x)$

**Example 9.3 for continuous data** Once again, we used the EYT Toy data set composed of 40 lines, four environments (Bed5IR, EHT, Flat5IR, and LHT), and four response variables, DTHD, DTMT, GY, and Height, but now we work with the GY variable, which is continuous.

Since the data set is the same, all the codes used for its implementation are the same, but now we work with the continuous GY variable. This continuous response variable was chosen using the following code:

```
##########Selecting the GY, continuous response variable######
y1=Data.Final$GY
y2=y1
n=dim(Data.Final)[1]
```

It is important to point out that the code for implementing SVR is exactly the same as that used to implement SVM, but with the difference that here it is not necessary to put the response variable (outcome) as a factor, since now the response variable is continuous. The code used now without $G \times E$ interaction is given in Appendix 3, but the only difference between this code and the code given in Appendix 2 is the following:

```
X2=X1
actual_CV=r
y1=y2
positionTST=c(CrossV$CrossValidation_list[[r]])
```

By using five-folds without tuning and ignoring the $G \times E$ interaction term for the five types of kernels, we got the following results:

```
> results
   Type       MSE
1  linear     0.3403227
2  radial     1.0868555
3  polynomial 2.8076426
4  sigmoid    0.4211818
```

Here we see that the best predictions were obtained with the linear kernel, the second best with the sigmoid kernel, and the worst with the polynomial kernel. When the $G \times E$ interaction term is taken into account, the prediction performance in terms of MSE is equal to

```
> results
   Type        MSE
1  linear      1.366566e+01
2  radial      2.052131e+00
3  polynomial  6.675209e+06
4  sigmoid     2.296421e+00
```

In general, taking into account the $G \times E$ interaction term produced a worse prediction performance. But now the best predictions were under the radial kernel and the worst under the polynomial kernel. These results were also obtained using Appendix 3, but with X1 = as.data.frame(cbind(ZT,Z1G,Z2TG)).

Finally, this chapter provides the fundamentals of support vector machines which were studied in considerable detail and in such a way that the user understands the basis of this powerful method. We provided many examples applied for genomic predictions that illustrated how to fit SVM methods for binary, ordinal, and continuous outcomes with and without genotype × environment interaction. We also provided the components needed to build some kernels manually, which is the key for capturing nonlinearities of the input data.

## Appendix 1

Tuning process for different types of kernels ignoring the $G \times E$ interaction term with a binary response variable denoted as Height.

```
rm(list=ls())
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(e1071)
library(caret)
#################Loading the data###############################
load("Data_Toy_EYT.RData")
ls()
```

```
Gg=data.matrix(G_Toy_EYT)
Data.Final=Pheno_Toy_EYT

##########Ordering the data#####################################
Data.Final=Data.Final[order(Data.Final$Env,Data.Final$GID),]

########Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data.Final$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZT=model.matrix(~0+as.factor(Data.Final$Env))
Z2TG=model.matrix(~0+Z1G:as.factor(Data.Final$Env))

############Preparation for training-testing sets############
Data.Final_1=Data.Final[,c(1:3)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final$Env))

#############Training-testing partitions#####################
nCV=5
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)

Y=Data.Final[,3:ncol(Data.Final)]
y1=Y$Height+1
y2=y1
n=dim(Y)[1]

##########Concatenating the information for input information####
X1=as.data.frame(cbind(ZT,Z1G))
dim(X1)

Pred_all_traits<-data.frame()

results<-data.frame() #save cross-validation results
Type=list("linear","radial","polynomial","sigmoid")
for (i in 1:4){
PCCC_Part=c()
for(r in 1:nCV) {
 ##### a) input, output, and testing set##################
X2=X1
actual_CV=r
y1=as.factor(y2)
positionTST=c(CrossV$CrossValidation_list[[r]])

###### b) Training and testing sets######################
X_tr=droplevels(X2[-positionTST,])
X_ts=droplevels(X2[positionTST,])
y_tr=y1[-positionTST] ###Training
y_ts=y1[positionTST]
```

```
####### c) Deleting columns with no variance#############
var_x=apply(X_tr,2,var)
length(var_x)
pos_var0=which(var_x>0)
length(pos_var0)
X_tr_New=X_tr[,pos_var0]
X_ts_New=X_ts[,pos_var0]

####### d) Grid and tuning process#############################
Nobs=nrow(X_tr_New)
Ncols=ncol(X_tr_New)
Ncols_2=Ncols-10

gamma_values=seq(1/(Ncols-3),1/(Ncols-20),1/(10*Ncols))

obj<-tune(svm,train.y=y_tr,train.x=X_tr_New, kernel=Type[[i]],
ranges=list(gamma=gamma_values, cost =seq(1.3, 2, 0.05)),tunecontrol
= tune.control(sampling = "fix"))

Par_gamma=as.numeric(obj$best.parameters[1])

Par_cost=as.numeric(obj$best.parameters[2])

Best.model=obj$best.model

#########e) predictions with the best model#####################
ypred=predict(Best.model,X_ts_New)
Predicted=ypred
Observed=y_ts
xtab <- table(Observed, Predicted)
Conf_Matrix=confusionMatrix(xtab)

#######Calculating the accuracy in terms of PCCC#########
PCCC=Conf_Matrix$overall[1]
PCCC_Part=c(PCCC_Part,PCCC)
}
PCCC_Part

results=rbind(results,data.frame(Type=Type[[i]], PCCC=mean
(PCCC_Part)))
}
results
```

# Appendix 2

Training SVM models for different types of kernels ignoring the $G \times E$ interaction term, without tuning, with the ordinal response variable DTHD with three classes.

```
rm(list=ls())
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(e1071)
library(caret)

########Loading the data#########################################
load("Data_Toy_EYT.RData")
ls()
Gg=data.matrix(G_Toy_EYT)
Data.Final=Pheno_Toy_EYT

########Ordering the data########################################
Data.Final=Data.Final[order(Data.Final$Env,Data.Final$GID),]

########Creating the design matrix of Lines ##################
Z1G=model.matrix(~0+as.factor(Data.Final$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZT=model.matrix(~0+as.factor(Data.Final$Env))
Z2TG=model.matrix(~0+Z1G:as.factor(Data.Final$Env))

######Preparation for building training-testing sets###########
Data.Final_1=Data.Final[,c(1:3)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final$Env))

#############Training-testing partitions####################
nCV=5
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)

##########Selecting the DTHD, ordinal output######
Y=Data.Final[,3:ncol(Data.Final)]
y1=Y$DTHD
y1
y2=y1
n=dim(Y)[1]

##########Concatenating the information for input information####
X1=as.data.frame(cbind(ZT,Z1G))
dim(X1)

Pred_all_traits<-data.frame()

results<-data.frame() #save cross-validation results
Type=list("radial","linear","polynomial","sigmoid")
```

```
for (i in 1:4){
 PCCC_Part=c()
for(r in 1:nCV) {
###### a) input, output, and testing set##################
X2=X1
actual_CV=r
y1=as.factor(y2)
positionTST=c(CrossV$CrossValidation_list[[r]])

###### b) Training and testing sets######################
X_tr=droplevels(X2[-positionTST,])
X_ts=droplevels(X2[positionTST,])
y_tr=y1[-positionTST] ###Training
y_ts=y1[positionTST]

####### c) Deleting columns with no variance##############
var_x=apply(X_tr,2,var)
length(var_x)
pos_var0=which(var_x>0)
length(pos_var0)
X_tr_New=X_tr[,pos_var0]
X_ts_New=X_ts[,pos_var0]

####### d) Fitting the model with SVM######################
fm1=svm(y=y_tr,x=X_tr_New,kernel=Type[[i]])
ypred=predict(fm1,X_ts_New)
Predicted=ypred
Observed=y_ts
xtab <- table(Observed, Predicted)
Conf_Matrix=confusionMatrix(xtab)

####### e) Calculating the accuracy in terms of PCCC#########
PCCC=Conf_Matrix$overall[1]
PCCC_Part=c(PCCC_Part,PCCC)
}
PCCC_Part

results=rbind(results,data.frame(Type=Type[[i]], PCCC=mean
(PCCC_Part)))
}
results
```

# Appendix 3

Training SVR models for different types of kernels ignoring the $G \times E$ interaction term, without tuning, with the continuous response variable GY.

```
rm(list=ls())
library(BMTME)
```

```
library(plyr)
library(tidyr)
library(dplyr)
library(e1071)

load("Data_Toy_EYT.RData")
ls()
Gg=data.matrix(G_Toy_EYT)
G=Gg

Data.Final=Pheno_Toy_EYT
Data.Final=Data.Final[order(Data.Final$Env,Data.Final$GID),]

#######Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data.Final$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZT=model.matrix(~0+as.factor(Data.Final$Env))
Z2TG=model.matrix(~0+Z1G:as.factor(Data.Final$Env))
nCV=5

Data.Final_1=Data.Final[,c(1:3)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final$Env))

############Training-testing partitions####################
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)

Y=Data.Final[,3:ncol(Data.Final)]
head(Y)
y1=Y$GY
y2=y1
n=dim(Y)[1]
##########Joining the information for input information####
X1=as.data.frame(cbind(ZT,Z1G))
dim(X1)

Pred_all_traits<-data.frame()

results<-data.frame() #save cross-validation results
Type=list("linear","radial","polynomial","sigmoid")
for (i in 1:4){
MSE_Part=c()
for(r in 1:nCV) {
#r=1
X2=X1
actual_CV=r
y1=y2
#y1=as.factor(y2)
positionTST=c(CrossV$CrossValidation_list[[r]])
```

```
##############Training and testing sets################
X_tr=droplevels(X2[-positionTST,])
X_ts=droplevels(X2[positionTST,])
y_tr=y1[-positionTST] ###Training
y_ts=y1[positionTST]

##########Deleting columns with no variance#############
var_x=apply(X_tr,2,var)
length(var_x)
pos_var0=which(var_x>0)
length(pos_var0)
X_tr_New=X_tr[,pos_var0]
X_ts_New=X_ts[,pos_var0]

#########Fitting the model with SVM#####################
fm1=svm(y=y_tr,x=X_tr_New,kernel=Type[[i]])
ypred=predict(fm1,X_ts_New)

Predicted=as.numeric(ypred)
Observed=as.numeric(y_ts)

MSE=mean((Predicted-Observed)^2)
MSE_Part=c(MSE_Part,MSE)
}
MSE_Part
mean(MSE_Part)

results=rbind(results,data.frame(Type=Type[[i]], MSE=mean
(MSE_Part)))
}
results
```

## References


Attewell P, Monaghan DB, Kwong D (2015) Data mining for the social sciences: an introduction. University of California Press, Oakland

Awad M, Khanna R (2015) Efficient learning machines—theories, concepts, and applications for engineers and system designers. Apress Open

Bishop CM (2006) Pattern recognition and machine learning. Springer Science + Business Media, LCC, New York

Burges CJ (1998) A tutorial on support vector machines for pattern recognition. Data Min Knowl Disc 2(2):121–167

Byun H, Lee SW (2002) Applications of support vector machines for pattern recognition: a survey. In: SVM ‘02 proceedings of the first international workshop on pattern recognition with support vector machines. Springer, London, pp 213–236

Cortes C, Vapnik V (1995) Support-vector network. Mach Learn 20:125
James G, Witten D, Hastie T, Tibshirani R (2013) An introduction to statistical learning: with applications in R. Springer, New York
R Core Team (2018) R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna. ISBN 3-900051-07-0. http://www.R-project.org/
Vapnik V (1995) The nature of statistical learning theory. Springer, New York

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 10
# Fundamentals of Artificial Neural Networks and Deep Learning

## 10.1 The Inspiration for the Neural Network Model

The inspiration for artificial neural networks (ANN), or simply neural networks, resulted from the admiration for how the human brain computes complex processes, which is entirely different from the way conventional digital computers do this. The power of the human brain is superior to many information-processing systems, since it can perform highly complex, nonlinear, and parallel processing by organizing its structural constituents (neurons) to perform such tasks as accurate predictions, pattern recognition, perception, motor control, etc. It is also many times faster than the fastest digital computer in existence today. An example is the sophisticated functioning of the information-processing task called human vision. This system helps us to understand and capture the key components of the environment and supplies us with the information we need to interact with the environment. That is, the brain very often performs perceptual recognition tasks (e.g., voice recognition embedded in a complex scene) in around 100–200 ms, whereas less complex tasks many times take longer even on a powerful computer (Haykin 2009).

Another interesting example is the sonar of a bat, since the sonar is an active echolocation system. The sonar provides information not only about how far away the target is located but also about the relative velocity of the target, its size, and the size of various features of the target, including its azimuth and elevation. Within a brain the size of a plum occur the computations required to extract all this information from the target echo. Also, it is documented that an echolocating bat has a high rate of success when pursuing and capturing its target and, for this reason, is the envy of radar and sonar engineers (Haykin 2009). This bat capacity inspired the development of radar, which is able to detect objects that are in its path, without needing to see them, thanks to the emission of an ultrasonic wave, the subsequent reception and processing of the echo, which allows it to detect obstacles in its flight with surprising speed and accuracy (Francisco-Caicedo and López-Sotelo 2009).

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_10

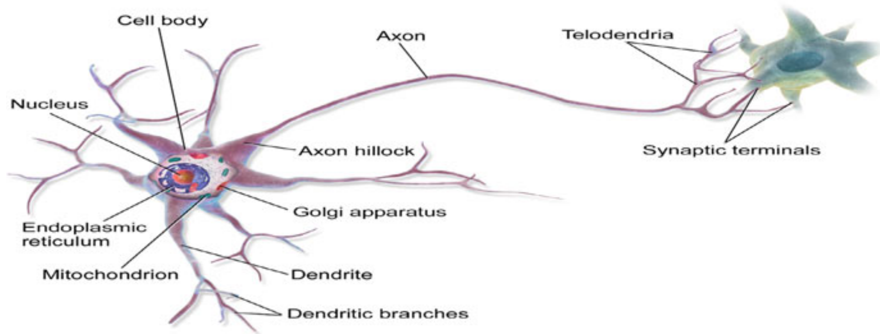



**Fig. 10.1** A graphic representation of a biological neuron

In general, the functioning of the brains of humans and other animals is intriguing because they are able to perform very complex tasks in a very short time and with high efficiency. For example, signals from sensors in the body convey information related to sight, hearing, taste, smell, touch, balance, temperature, pain, etc. Then the brain's neurons, which are autonomous units, transmit, process, and store this information so that we can respond successfully to external and internal stimuli (Dougherty 2013). The neurons of many animals transmit spikes of electrical activity through a long, thin strand called an axon. An axon is divided into thousands of terminals or branches, where depending on the size of the signal they synapse to dendrites of other neurons (Fig. 10.1). It is estimated that the brain is composed of around $10^{11}$ neurons that work in parallel, since the processing done by the neurons and the memory captured by the synapses are distributed together over the network. The amount of information processed and stored depends on the threshold firing levels and also on the weight given by each neuron to each of its inputs (Dougherty 2013).

One of the characteristics of biological neurons, to which they owe their great capacity to process and perform highly complex tasks, is that they are highly connected to other neurons from which they receive stimuli from an event as it occurs, or hundreds of electrical signals with the information learned. When it reaches the body of the neuron, this information affects its behavior and can also affect a neighboring neuron or muscle (Francisco-Caicedo and López-Sotelo 2009). Francisco-Caicedo and López-Sotelo (2009) also point out that the communication between neurons goes through the so-called synapses. A synapse is a space that is occupied by chemicals called neurotransmitters. These neurotransmitters are responsible for blocking or passing on signals that come from other neurons. The neurons receive electrical signals from other neurons with which they are in contact. These signals accumulate in the body of the neuron and determine what to do. If the total electrical signal received by the neuron is sufficiently large, the action potential can be overcome, which allows the neuron to be activated or, on the contrary, to remain inactive. When a neuron is activated, it is able to transmit an electrical impulse to the neurons with which it is in contact. This new impulse, for example, acts as an input

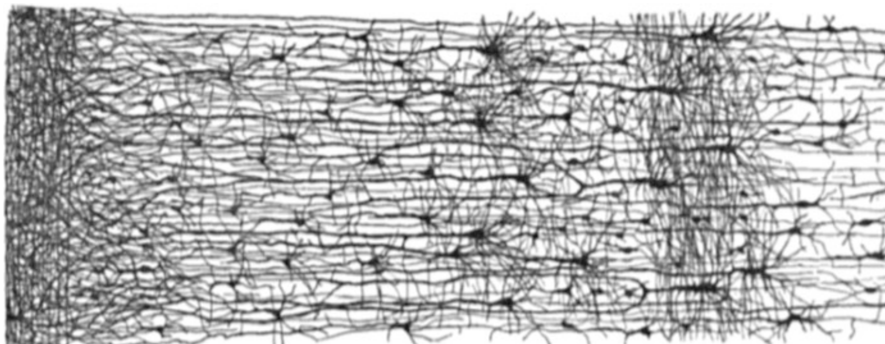


**Fig. 10.2** Multiple layers in a biological neural network of human cortex

to other neurons or as a stimulus in some muscles (Francisco-Caicedo and López-Sotelo 2009). The architecture of biological neural networks is still the subject of active research, but some parts of the brain have been mapped, and it seems that neurons are often organized in consecutive layers, as shown in Fig. 10.2.

ANN are machines designed to perform specific tasks by imitating how the human brain works, and build a neural network made up of hundreds or even thousands of artificial neurons or processing units. The artificial neural network is implemented by developing a computational learning algorithm that does not need to program all the rules since it is able to build up its own rules of behavior through what we usually refer to as "experience." The practical implementation of neural networks is possible due to the fact that they are massively parallel computing systems made up of a huge number of basic processing units (neurons) that are interconnected and learn from their environment, and the synaptic weights capture and store the strengths of the interconnected neurons. The job of the learning algorithm consists of modifying the synaptic weights of the network in a sequential and supervised way to reach a specific objective (Haykin 2009). There is evidence that neurons working together are able to learn complex linear and nonlinear input–output relationships by using sequential training procedures. It is important to point out that even though the inspiration for these models was quite different from what inspired statistical models, the building blocks of both types of models are quite similar. Anderson et al. (1990) and Ripley (1993) pointed out that neural networks are simply no more than *generalized nonlinear statistical models*. However, Anderson et al. (1990) were more expressive in this sense and also pointed out that "ANN are statistics for amateurs since most neural networks conceal the statistics from the user."

## 10.2 The Building Blocks of Artificial Neural Networks

To get a clear idea of the main elements used to construct ANN models, in Fig. 10.3 we provide a general artificial neural network model that contains the main components for this type of models.

$x_1, \ldots, x_p$ represents the information (input) that the neuron receives from the external sensory system or from other neurons with which it has a connection. $\boldsymbol{w} = (w_1, \ldots, w_p)$ is the vector of synaptic weights that modifies the received information emulating the synapse between the biological neurons. These can be interpreted as gains that can attenuate or amplify the values that they wish to propagate toward the neuron. Parameter $b_j$ is known as the bias (intercept or threshold) of a neuron. Here in ANN, learning refers to the method of modifying the weights of connections between the nodes (neurons) of a specified network.

The different values that the neuron receives are modified by the synaptic weights, which then are added together to produce what is called the *net input*. In mathematical notation, that is equal to

$$v_j = \sum_{j=1}^{p} \omega_{ij} x_j$$

This net input ($v_j$) is what determines whether the neuron is activated or not. The activation of the neuron depends on what we call the *activation function*. The net input is evaluated in this function and we obtain the output of the network as shown next:

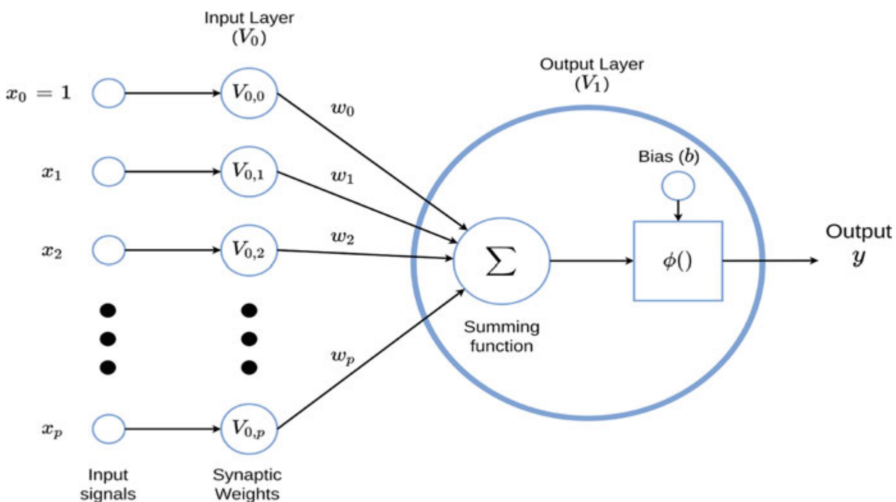



**Fig. 10.3** General artificial neural network model

$$y_j = g(v_j),$$

where $g$ is the activation function. For example, if we define this function as a unit step (also called threshold), the output will be 1 if the net input is greater than zero; otherwise the output will be 0. Although there is no biological behavior indicating the presence of something similar to the neurons of the brain, the use of the activation function is an artifice that allows applying ANN to a great diversity of real problems. According to what has been mentioned, output $y_j$ of the neuron is generated when evaluating the net input ($v_j$) in the activation function. We can propagate the output of the neuron to other neurons or it can be the output of the network, which, according to the application, will have an interpretation for the user. In general, the job of an artificial neural network model is done by simple elements called neurons. The signals are passed between neurons through connection links. Each connection link has an associated weight, which, in a typical neuronal network, multiplies the transmitted signal. Each neuron applies an activation function (usually nonlinear) to the network inputs (sum of the heavy input signals) for determining its corresponding sign. Later in this chapter, we describe the many options for activation functions and the context in which they can be used.

A unilayer ANN like that in Fig. 10.3 has a low processing capacity by itself and its level of applicability is low; its true power lies in the interconnection of many ANNs, as happens in the human brain. This has motivated different researchers to propose various topologies (architectures) to connect neurons to each other in the context of ANN. Next, we provide two definitions of ANN and one definition of deep learning:

*Definition 1*. An artificial neural network is a system composed of many simple elements of processing which operate in parallel and whose function is determined by the structure of the network and the weight of connections, where the processing is done in each of the nodes or computing elements that has a low processing capacity (Francisco-Caicedo and López-Sotelo 2009).

*Definition 2*. An artificial neural network is a structure containing simple elements that are interconnected in many ways with hierarchical organization, which tries to interact with objects in the real world in the same way as the biological nervous system does (Kohonen 2000).

*Deep learning model*. We define deep learning as a generalization of ANN where more than one hidden layer is used, which implies that more neurons are used for implementing the model. For this reason, an artificial neural network with multiple hidden layers is called a Deep Neural Network (DNN) and the practice of training this type of networks is called *deep learning (DL)*, which is a branch of statistical machine learning where a multilayered (deep) topology is used to map the relations between input variables (independent variables) and the response variable (outcome). Chollet and Allaire (2017) point out that DL puts the "emphasis on learning successive layers of increasingly meaningful representations." The adjective "deep" applies not to the acquired knowledge, but to the way in which the knowledge is acquired (Lewis 2016), since it stands for the idea of successive layers of

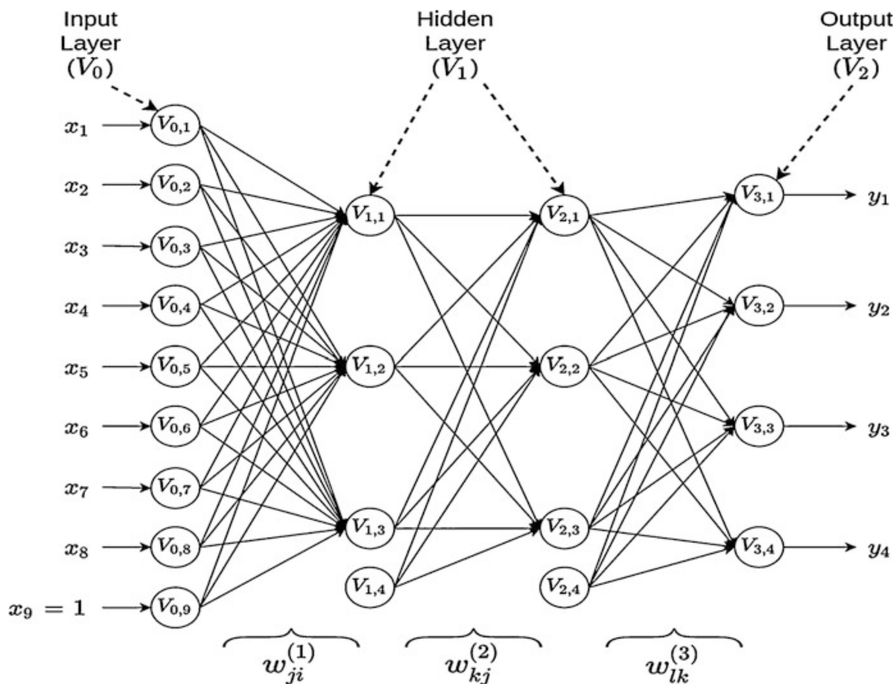



**Fig. 10.4** Artificial deep neural network with a feedforward neural network with eight input variables ($x1$, ... ,$x8$), four output variables ($y1$, $y2$, $y3$, $y4$), and two hidden layers with three neurons each

representations. The "deep" of the model refers to the number of layers that contribute to the model. For this reason, this field is also called layered representation learning and hierarchical representation learning (Chollet and Allaire 2017).

It is important to point out that DL as a subset of machine learning is an aspect of artificial intelligence (AI) that has more complex ways of connecting layers than conventional ANN, which uses more neurons than previous networks to capture nonlinear aspects of complex data better, but at the cost of more computing power required to automatically extract useful knowledge from complex data.

To have a more complete picture of ANN, we provide another model, which is a DL model since it has two hidden layers, as shown in Fig. 10.4.

From Fig. 10.4 we can see that an artificial neural network is a directed graph whose *nodes* correspond to neurons and whose *edges* correspond to *links* between them. Each neuron receives, as input, a weighted sum of the outputs of the neurons connected to its incoming edges (Shalev-Shwartz and Ben-David 2014). In the artificial deep neural network given in Fig. 10.4, there are four layers ($V_0$, $V_1$, $V_2$, and $V_3$): $V_0$ represents the input layer, $V_1$ and $V_2$ are the hidden layers, and $V_3$ denotes the output layer. In this artificial deep neural network, three is the number of layers of the network since $V_0$, which contains the input information, is excluded. This is also called the "depth" of the network. The size of this network is $|V| = \left|\bigcup_{t=0}^{\mathrm{T}} V_t\right| =$

$|9 + 4 + 4 + 4| = 21$. Note that in each layer we added +1 to the observed units to represent the node of the bias (or intercept). The width of the network is $\max|V_t| = 9$.

The analytical form of the model given in Fig. 10.4 for output $o$, with $d$ inputs, $M_1$ hidden neurons (units) in hidden layer 1, $M_2$ hidden units in hidden layer 2, and O output neurons is given by the following (10.1)–(10.3):

$$V_{1j} = g_1\left(\sum_{i=1}^{d} w_{ji}^{(1)} x_i\right) \text{ for } j = 1, \ldots, M_1 \tag{10.1}$$

$$V_{2k} = g_2\left(\sum_{j=1}^{M_1} w_{kj}^{(2)} V_{1j}\right) \text{ for } k = 1, \ldots, M_2 \tag{10.2}$$

$$y_l = g_3\left(\sum_{k=1}^{M_2} w_{lk}^{(3)} V_{2k}\right) \text{ for } l = 1, \ldots, O \tag{10.3}$$

where (10.1) produces the output of each of the neurons in the first hidden layer, (10.2) produces the output of each of the neurons in the second hidden layer, and finally (10.3) produces the output of each response variable of interest. The learning process is obtained with the weights ($w_{ji}^{(1)}, w_{kj}^{(2)}$, and $w_{lk}^{(3)}$), which are accommodated in the following vector: $\boldsymbol{w} = \left(w_{11}^{(1)}, w_{12}^{(1)}, \ldots, w_{1d}^{(1)}, w_{21}^{(2)}, w_{22}^{(2)}, \ldots, w_{2M_1}^{(2)}, w_{31}^{(3)}, w_{32}^{(3)}, \ldots, w_{3M_2}^{(3)}\right)$, $g_1$, $g_2$, and $g_3$ are the activation functions in hidden layers 1, 2, and the output layer, respectively.

The model given in Fig. 10.4 is organized as several interconnected layers: the input layer, hidden layers, and output layer, where each layer that performs nonlinear transformations is a collection of artificial neurons, and connections among these layers are made using weights (Fig. 10.4). When only one output variable is present in Fig. 10.4, the model is called univariate DL model. Also, when only one hidden layer is present in Fig. 10.4, the DL model is reduced to a conventional artificial neural network model, but when more than one hidden layer is included, it is possible to better capture complex interactions, nonlinearities, and nonadditive effects. To better understand the elements of the model depicted in Fig. 10.4, it is important to distinguish between the types of layers and the types of neurons; for this reason, next we will explain the type of layers and then the type of neurons in more detail.

(a) Input layer: It is the set of neurons that directly receives the information coming from the external sources of the network. In the context of Fig. 10.4, this information is $x1, \ldots, x8$ (Francisco-Caicedo and López-Sotelo 2009). Therefore, the number of neurons in an input layer is most of the time the same as the number of the input explanatory variables provided to the network. Usually input layers are followed by at least one hidden layer. Only in feedforward neuronal networks, input layers are fully connected to the next hidden layer (Patterson and Gibson 2017).

(b) Hidden layers: Consist of a set of internal neurons of the network that do not have direct contact with the outside. The number of hidden layers can be 0, 1, or more. In general, the neurons of each hidden layer share the same type of information; for this reason, they are called hidden layers. The neurons of the hidden layers can be interconnected in different ways; this determines, together with their number, the different topologies of ANN and DNN (Francisco-Caicedo and López-Sotelo 2009). The learned information extracted from the training data is stored and captured by the weight values of the connections between the layers of the artificial neural network. Also, it is important to point out that hidden layers are key components for capturing complex nonlinear behaviors of data more efficiently (Patterson and Gibson 2017).

(c) Output layer: It is a set of neurons that transfers the information that the network has processed to the outside (Francisco-Caicedo and López-Sotelo 2009). In Fig. 10.4 the output neurons correspond to the output variables $y1$, $y2$, $y3$, and $y4$. This means that the output layer gives the answer or prediction of the artificial neural network model based on the input from the input layer. The final output can be continuous, binary, ordinal, or count depending on the setup of the ANN which is controlled by the activation (or inverse link in the statistical domain) function we specified on the neurons in the output layer (Patterson and Gibson 2017).

Next, we define the types of neurons: (1) *input neuron.* A neuron that receives external inputs from outside the network; (2) *output neuron.* A neuron that produces some of the outputs of the network; and (3) *hidden neuron.* A neuron that has no direct interaction with the "outside world" but only with other neurons within the network. Similar terminology is used at the layer level for *multilayer neural networks.*

As can be seen in Fig. 10.4, the distribution of neurons within an artificial neural network is done by forming levels of a certain number of neurons. If a set of artificial neurons simultaneously receives the same type of information, we call it a layer. We also described a network of three types of levels called layers. Figure 10.5 shows another six networks with different numbers of layers, and half of them (Fig. 10.5a, c, e) are univariate since the response variable we wish to predict is only one, while the other half (Fig. 10.5b, d, f) are multivariate since the interest of the network is to predict two outputs. It is important to point out that subpanels a and b in Fig. 10.5 are networks with only one layer and without hidden layers; for this reason, this type of networks corresponds to conventional regression or classification regression models.

Therefore, the topology of an artificial neural network is the way in which neurons are organized inside the network; it is closely linked to the learning algorithm used to train the network. Depending on the number of layers, we define the networks as *monolayer* and *multilayer*; and if we take as a classification element the way information flows, we define the networks as feedforward or recurrent. Each type of topology will be described in another section.

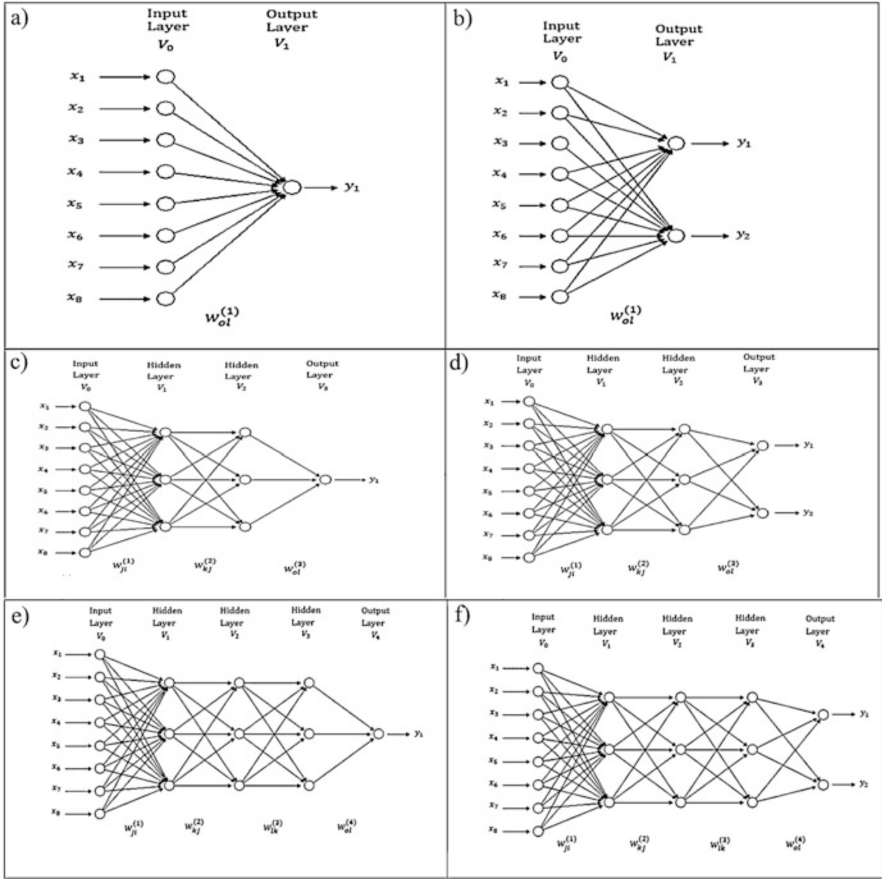



**Fig. 10.5** Different feedforward topologies with univariate and multivariate outputs and different number of layers. (**a**) Unilayer and univariate output. (**b**) Unilayer and multivariate output. (**c**) Three layer and univariate output. (**d**) Three layer and multivariate output. (**e**) Four layer univariate output. (**f**) Four layer multivariate output

In summary, an artificial (deep) neural network model is an information processing system that mimics the behavior of biological neural networks, which was developed as a generalization of mathematical models of human knowledge or neuronal biology.

## 10.3 Activation Functions

The mapping between inputs and a hidden layer in ANN and DNN is determined by activation functions. Activation functions propagate the output of one layer's nodes forward to the next layer (up to and including the output layer). Activation functions

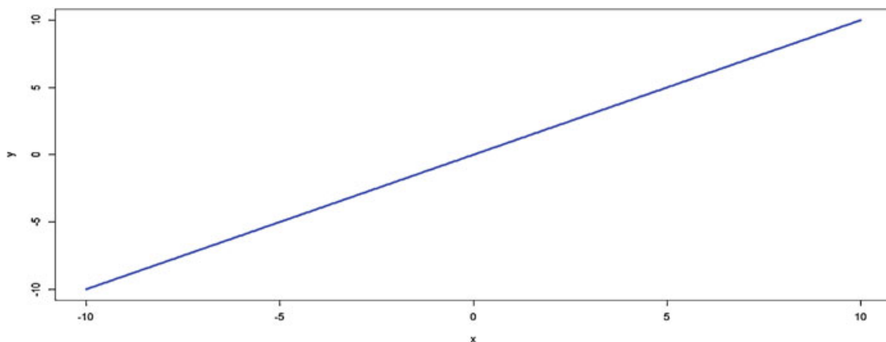



**Fig. 10.6** Representation of a linear activation function

are scalar-to-scalar functions that provide a specific output of the neuron. Activation functions allow nonlinearities to be introduced into the network's modeling capabilities (Wiley 2016). The activation function of a neuron (node) defines the functional form for how a neuron gets activated. For example, if we define a linear activation function as $g(z) = z$, in this case the value of the neuron would be the raw input, $x$, times the learned weight, that is, a linear model. Next, we describe the most popular activation functions.

### 10.3.1 Linear

Figure 10.6 shows a linear activation function that is basically the identity function. It is defined as $g(z) = Wz$, where the dependent variable has a direct, proportional relationship with the independent variable. In practical terms, it means the function passes the signal through unchanged. The problem with making activation functions linear is that this does not permit any nonlinear functional forms to be learned (Patterson and Gibson 2017).

### 10.3.2 Rectifier Linear Unit (ReLU)

The rectifier linear unit (ReLU) activation function is one of the most popular. The ReLU activation function is flat below some threshold (usually the threshold is zero) and then linear. The ReLU activates a node only if the input is above a certain quantity. When the input is below zero, the output is zero, but when the input rises above a certain threshold, it has a linear relationship with the dependent variable $g(z) = \max(0, z)$, as demonstrated in Fig. 10.7. Despite its simplicity, the ReLU activation function provides nonlinear transformation, and enough linear rectifiers can be used to approximate arbitrary nonlinear functions, unlike when only linear

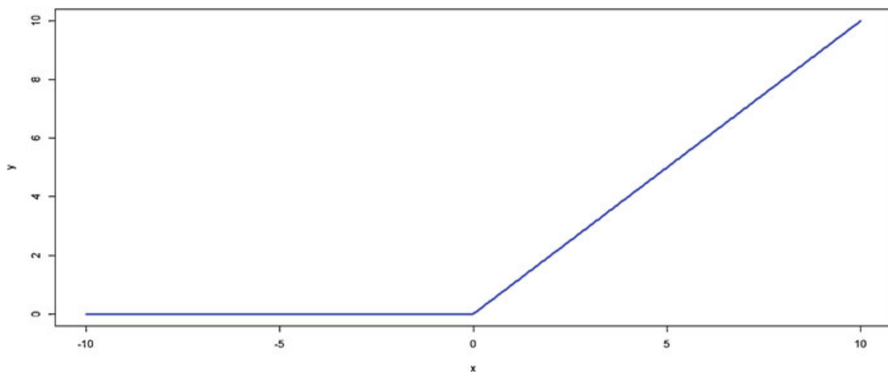



**Fig. 10.7** Representation of the ReLU activation function

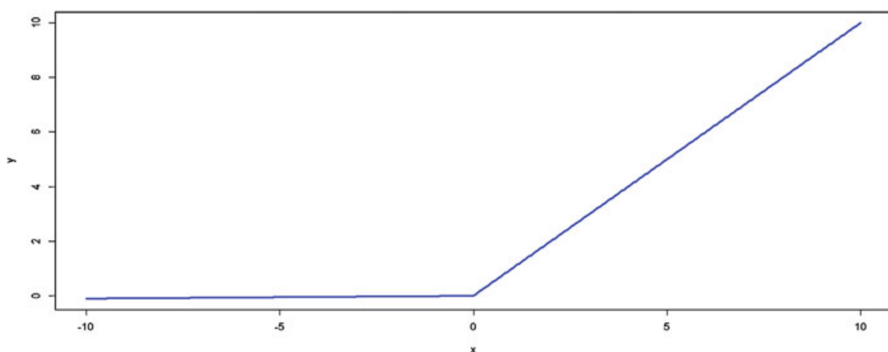



**Fig. 10.8** Representation of the Leaky ReLU activation function with $\alpha = 0.1$

activation functions are used (Patterson and Gibson 2017). ReLUs are the current state of the art because they have proven to work in many different situations. Because the gradient of a ReLU is either zero or a constant, it is not easy to control the vanishing exploding gradient issue, also known as the "dying ReLU" issue. ReLU activation functions have been shown to train better in practice than sigmoid activation functions. This activation function is the most used in hidden layers and in output layers when the response variable is continuous and larger than zero.

### *10.3.3 Leaky ReLU*

Leaky ReLUs are a strategy to mitigate the "dying ReLU" issue. As opposed to having the function be zero when $z < 0$, the leaky ReLU will instead have a small negative slope, $\alpha$,where $\alpha$ is a value between 0 and 1 (Fig. 10.8). In practice, some

success has been achieved with this ReLU variation, but results are not always consistent. The function of this activation function is given here:

$$g(z) = \begin{cases} z \text{ if } z > 0 \\ \alpha z \text{ otherwise} \end{cases}$$

### 10.3.4 Sigmoid

A sigmoid activation function is a machine that converts independent variables of near infinite range into simple probabilities between 0 and 1, and most of its output will be very close to 0 or 1. Like all logistic transformations, sigmoids can reduce extreme values or outliers in data without removing them. This activation function resembles an S (Wiley 2016; Patterson and Gibson 2017) and is defined as $g(z) = (1 + e^{-z})^{-1}$. This activation function is one of the most common types of activation functions used to construct ANNs and DNNs, where the outcome is a probability or binary outcome. This activation function is a strictly increasing function that exhibits a graceful balance between linear and nonlinear behavior but has the propensity to get "stuck," i.e., the output values would be very close to 1 or 0 when the input values are strongly positive or negative (Fig. 10.9). By getting "stuck" we mean that the learning process is not improving due to the large or small values of the output values of this activation function.

### 10.3.5 Softmax

Softmax is a generalization of the sigmoid activation function that handles multinomial labeling systems, that is, it is appropriate for categorical outcomes. Softmax is

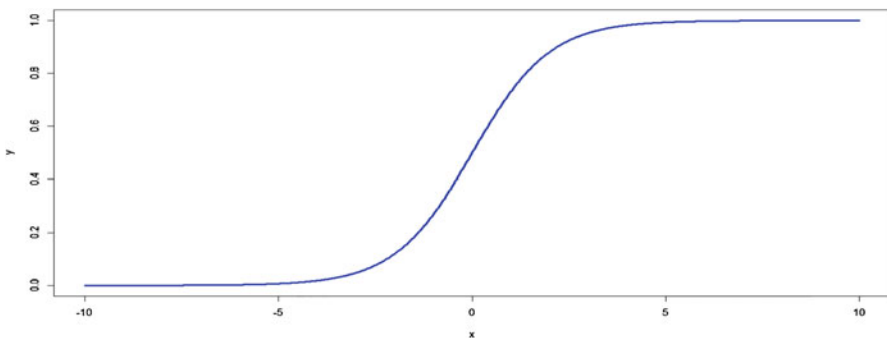



**Fig. 10.9** Representation of the sigmoid activation function

the function you will often find in the output layer of a classifier with more than two categories. The softmax activation function returns the probability distribution over mutually exclusive output classes. To further illustrate the idea of the softmax output layer and how to use it, let's consider two types of uses. If we have a multiclass modeling problem we only care about the best score across these classes, we'd use a softmax output layer with an *argmax()* function to get the highest score across all classes. For example, let us assume that our categorical response has ten classes; with this activation function we calculate a probability for each category (the sum of the ten categories is one) and we classify a particular individual in the class with the largest probability. It is important to recall that if we want to get binary classifications per output (e.g., "diseased and not diseased"), we do not want softmax as an output layer. Instead, we will use the sigmoid activation function explained before. The softmax function is defined as

$$g\left(z_j\right) = \frac{\exp\left(z_j\right)}{1 + \sum_{c=1}^{C} \exp\left(z_c\right)}, \quad j = 1, .., C$$

This activation function is a generalization of the sigmoid activation function that squeezes (force) a C dimensional vector of arbitrary real values to a C dimensional vector of real values in the range [0,1] that adds up to 1. A strong prediction would have a single entry in the vector close to 1, while the remaining entries would be close to 0. A weak prediction would have multiple possible categories (labels) that are more or less equally likely. The sigmoid and softmax activation functions are suitable for probabilistic interpretation due to the fact that the output is a probabilistic distribution of the classes. This activation function is mostly recommended for output layers when the response variable is categorical.

### *10.3.6 Tanh*

The hyperbolic tangent (Tanh) activation function is defined as $\tanh\left(\mathrm{z}\right) = \sinh\left(\mathrm{z}\right)/\cosh\left(\mathrm{z}\right) = \frac{\exp\left(z\right) - \exp\left(-z\right)}{\exp\left(z\right) + \exp\left(-z\right)}$. The hyperbolic tangent works well in some cases and, like the sigmoid activation function, has a sigmoidal ("S" shaped) output, with the advantage that it is less likely to get "stuck" than the sigmoid activation function since its output values are between $-1$ and 1, as shown in Fig. 10.10. For this reason, for hidden layers should be preferred the Tanh activation function. Large negative inputs to the tanh function will give negative outputs, while large positive inputs will give positive outputs (Patterson and Gibson 2017). The advantage of tanh is that it can deal more easily with negative numbers.

It is important to point out that there are more activations functions like the *threshold* activation function introduced in the pioneering work on ANN by McCulloch and Pitts (1943), but the ones just mentioned are some of the most used.

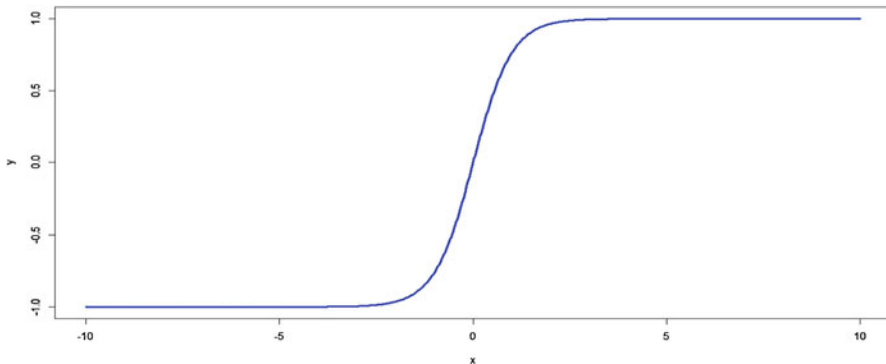



**Fig. 10.10** Representation of the tanh activation function

## 10.4 The Universal Approximation Theorem

The universal approximation theorem is at the heart of ANN since it provides the mathematical basis of why artificial neural networks work in practice for nonlinear input–output mapping. According to Haykin (2009), this theorem can be stated as follows.

Let $g()$ be a bounded, and monotone-increasing continuous function. Let $I_{m_0}$ denote the $m_0$-dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on $I_{m_0}$ is denoted by $C(I_{m_0})$. Then given any function $f \ni C(I_{m_0})$ and $\varepsilon > 0$, there is an integer $m_1$ and sets of real constants $\alpha_i$, $b_i$, and $w_{ij}$, where $i = 1,\ldots, m_1$ and $j = 1,\ldots, m_0$ such that we may define

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i g\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i\right) \tag{10.4}$$

as an approximate realization of function $f(\cdot)$; that is,

$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \varepsilon$$

For all $x_1, \ldots, x_{m_0}$ that lie in the input space.

$m_0$ represents the input nodes of a multilayer perceptron with a single hidden layer. $m_1$ is the number of neurons in the single hidden layer, $x_1, \ldots, x_{m_0}$ are the inputs, $w_{ij}$ denotes the weight of neuron $i$ in input $j$, $b_i$ denotes the bias corresponding to neuron $i$, and $\alpha_i$ is the weight of the output layer in neuron $i$.

This theorem states that any feedforward neural network containing a finite number of neurons is capable of approximating any continuous functions of arbitrary complexity to arbitrary accuracy, if provided enough neurons in even a single hidden layer, under mild assumptions of the activation function. In other words, this theorem says that any continuous function that maps intervals of real numbers to

some output interval of real numbers can be approximated arbitrarily and closely by a multilayer perceptron with just one hidden layer and a finite *very large number of neurons* (Cybenko 1989; Hornik 1991). However, this theorem only guarantees a reasonable approximation; for this reason, this theorem is an existence theorem. This implies that simple ANNs are able to represent a wide variety of interesting functions if given enough neurons and appropriate parameters; but nothing is mentioned about the algorithmic learnability of those parameters, nor about their time of learning, ease of implementation, generalization, or that a single hidden layer is optimum. The first version of this theorem was given by Cybenko (1989) for sigmoid activation functions. Two years later, Hornik (1991) pointed out that the potential of "ANN of being universal approximators is not due to the specific choice of the activation function, but to the multilayer feedforward architecture itself."

From this theorem, we can deduce that when an artificial neural network has more than two hidden layers, it will not always improve the prediction performance since there is a higher risk of converging to a local minimum. However, using two hidden layers is recommended when the data has discontinuities. Although the proof of this theorem was done for only a single output, it is also valid for the multi-output scenario and can easily be deduced from the single output case. It is important to point out that this theorem states that all activation functions will perform equally well in specific learning problems since their performance depends on the data and additional issues such as minimal redundancy, computational efficiency, etc.

## 10.5 Artificial Neural Network Topologies

In this subsection, we describe the most popular network topologies. An artificial *neural network topology* represents the way in which neurons are connected to form a network. In other words, the neural network topology can be seen as the relationship between the neurons by means of their connections. The topology of a neural network plays a fundamental role in its functionality and performance, as illustrated throughout this chapter. The generic terms *structure* and *architecture* are used as synonyms for network topology. However, caution should be exercised when using these terms since their meaning is not well defined and causes confusion in other domains where the same terms are used for other purposes.

More precisely, the topology of a neural network consists of its *frame* or *framework* of neurons, together with its *interconnection structure* or *connectivity:*

$$\text{Artificial neural network topology} \begin{cases} 1)\ \text{artificial neural framework} \\ 2)\ \text{interconnection structure.} \end{cases}$$

The next two subsections are devoted to these two components.

***Artificial neural framework*** Most neural networks, including many biological ones, have a layered topology. There are a few exceptions where the network is not explicitly layered, but those can usually be interpreted as having a layered topology, for example, in some *associative memory networks,* which can be seen as one-layer neural networks where all neurons function both as input and output units. At the framework level, neurons are considered abstract entities, therefore possible differences between them are not considered. The framework of an artificial neural network can therefore be described by the number of neurons, number of layers (denoted by $L$), and the size of the layer, which consists of the number of neurons in each of the layers.

***Interconnection structure*** The interconnection structure of an artificial neural network determines the way in which the neurons are linked. Based on a layered structure, several different kinds of connections can be distinguished (see Fig. 10.11): (a) *Interlayer connection:* This connects neurons in adjacent layers whose layer indices differ by one; (b) *Intralayer connection:* This is a connection between neurons in the same layer; (c) *Self-connection:* This is a special kind of intralayer connection that connects a neuron to itself; (d) *Supralayer connection:* This is a connection between neurons that are in distinct nonadjacent layers; in other words, these connections "cross" or "jump" at least one hidden layer.

With each connection *(interconnection*), a *weight (strength)* is associated which is a weighting factor that reflects its importance. This weight is a scalar value (a number), which can be positive *(excitatory)* or negative *(inhibitory).* If a connection has zero weight, it is considered to be nonexistent at that point in time.

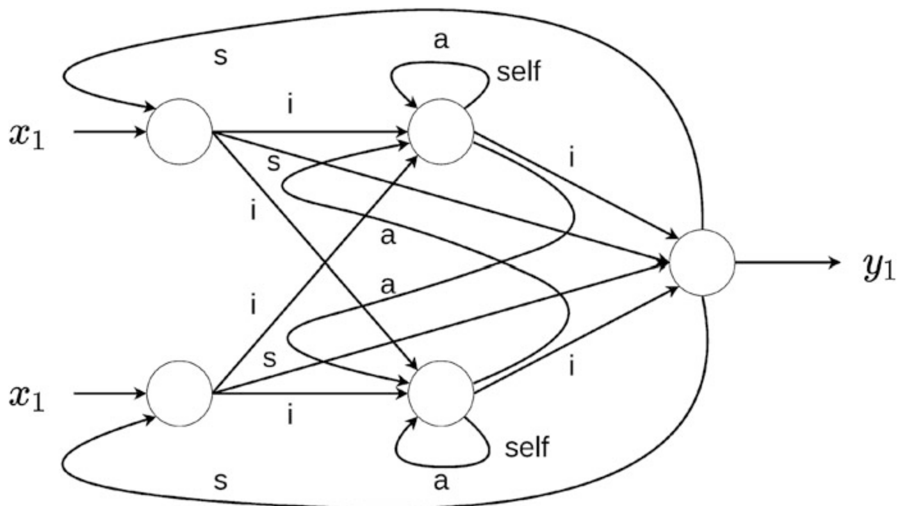



**Fig. 10.11** Network topology with two layers. (i) denotes the six interlayer connections, (s) denotes the four supralayered connections, and (a) denotes four intralayer connections of which two are self-connections

Note that the basic concept of layeredness is based on the presence of interlayer connections. In other words, every layered neural network has at least one interlayer connection between adjacent layers. If interlayer connections are absent between any two adjacent clusters in the network, a spatial reordering can be applied to the topology, after which certain connections become the interlayer connections of the transformed, layered network.

Now that we have described the two key components of an artificial neural network topology, we will present two of the most commonly used topologies.

***Feedforward network*** In this type of artificial neural network, the information flows in a single direction from the input neurons to the processing layer or layers (only interlayer connections) for monolayer and multilayer networks, respectively, until reaching the output layer of the neural network. This means that there are no connections between neurons in the same layer (no intralayer), and there are no connections that transmit data from a higher layer to a lower layer, that is, no supralayer connections (Fig. 10.12). This type of network is simple to analyze, but is not restricted to only one hidden layer.

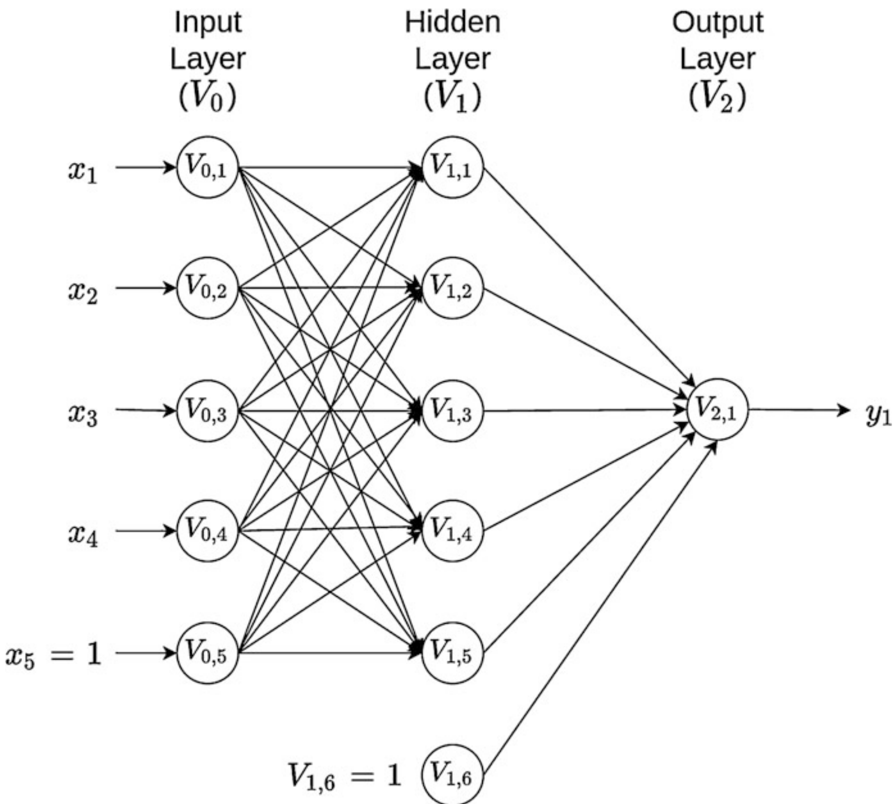



**Fig. 10.12** A simple two-layer feedforward artificial neural network

***Recurrent networks*** In this type of neural network, information does not always flow in one direction, since it can feed back into previous layers through synaptic connections. This type of neural network can be monolayer or multilayer. In this network, all the neurons have (1) incoming connections emanating from all the neurons in the previous layer, (2) ongoing connections leading to all the neurons in the subsequent layer, and (3) recurrent connections that propagate information between neurons of the same layer. Recurrent neural networks (RNNs) are different from a feedforward neural network in that they have at least one feedback loop since the signals travel in both directions. This type of network is frequently used in time series prediction since short-term memory, or delay, increases the power of recurrent networks immensely. In this case, we present an example of a recurrent two-layer neural network. The output of each neuron is passed through a delay unit and then taken to all the neurons, except itself. In Figs. 10.13 and 10.14, we can see that only one input variable is presented to the input units, the feedforward flow is computed, and the outputs are fed back as auxiliary inputs. This leads to a different set of hidden unit activations, new output activations, and so on. Ultimately, the activations stabilize, and the final output values are used for predictions.

However, it is important to point out out that despite the just mentioned virtues of recurrent artificial neural networks, they are still largely theoretical and produce mixed results (good and bad) in real applications. On the other hand, the feedforward networks are the most popular since they are successfully implemented in all areas of domain; the multilayer perceptron (MLP; that is, onother name give to feedforward networks) is the de facto standard artificial neural network topology (Lantz 2015). There are other DNN topologies like convolutional neural networks that are presented in Chap. 13, but they can be found also in books specializing in deep learning.

## 10.6 Successful Applications of ANN and DL

The success of ANN and DL is due to remarkable results on perceptual problems such as seeing and hearing—problems involving skills that seem natural and intuitive to humans but have long been elusive for machines. Next, we provide some of these successful applications:

(a) Near-human-level image classification, speech recognition, handwriting transcription, autonomous driving (Chollet and Allaire 2017)
(b) Automatic translation of text and images (LeCun et al. 2015)
(c) Improved text-to-speech conversion (Chollet and Allaire 2017)
(d) Digital assistants such as Google Now and Amazon Alexa
(e) Improved ad targeting, as used by Google, Baidu, and Bing
(f) Improved search results on the Web (Chollet and Allaire 2017)
(g) Ability to answer natural language questions (Goldberg 2016)
(h) In games like chess, Jeopardy, GO, and poker (Makridakis et al. 2018)
(i) Self-driving cars (Liu et al. 2017),

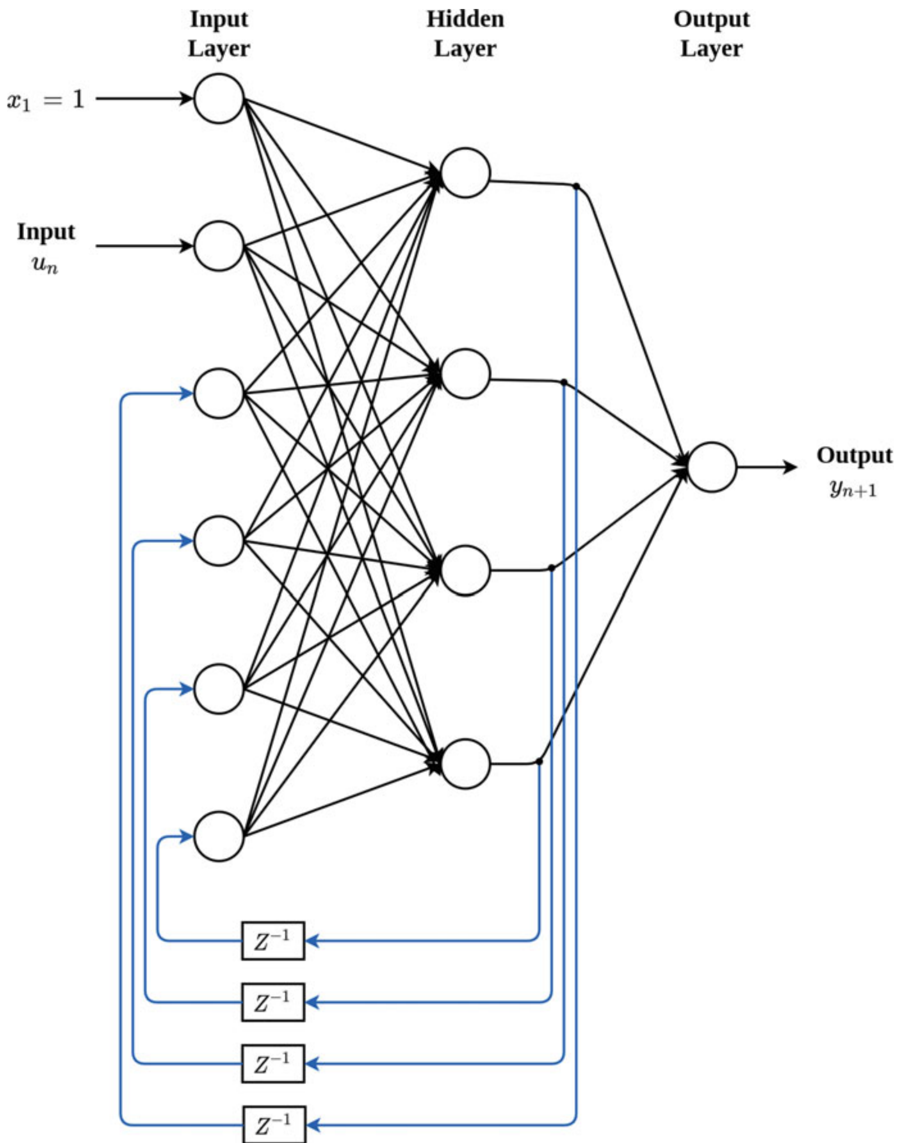



**Fig. 10.13** A simple two-layer recurrent artificial neural network with univariate output

(j) Voice search and voice-activated intelligent assistants (LeCun et al. 2015)
(k) Automatically adding sound to silent movies (Chollet and Allaire 2017)
(l) Energy market price forecasting (Weron 2014)
(m) Image recognition (LeCun et al. 2015)
(n) Prediction of time series (Dingli and Fournier 2017)
(o) Predicting breast, brain (Cole et al. 2017), or skin cancer

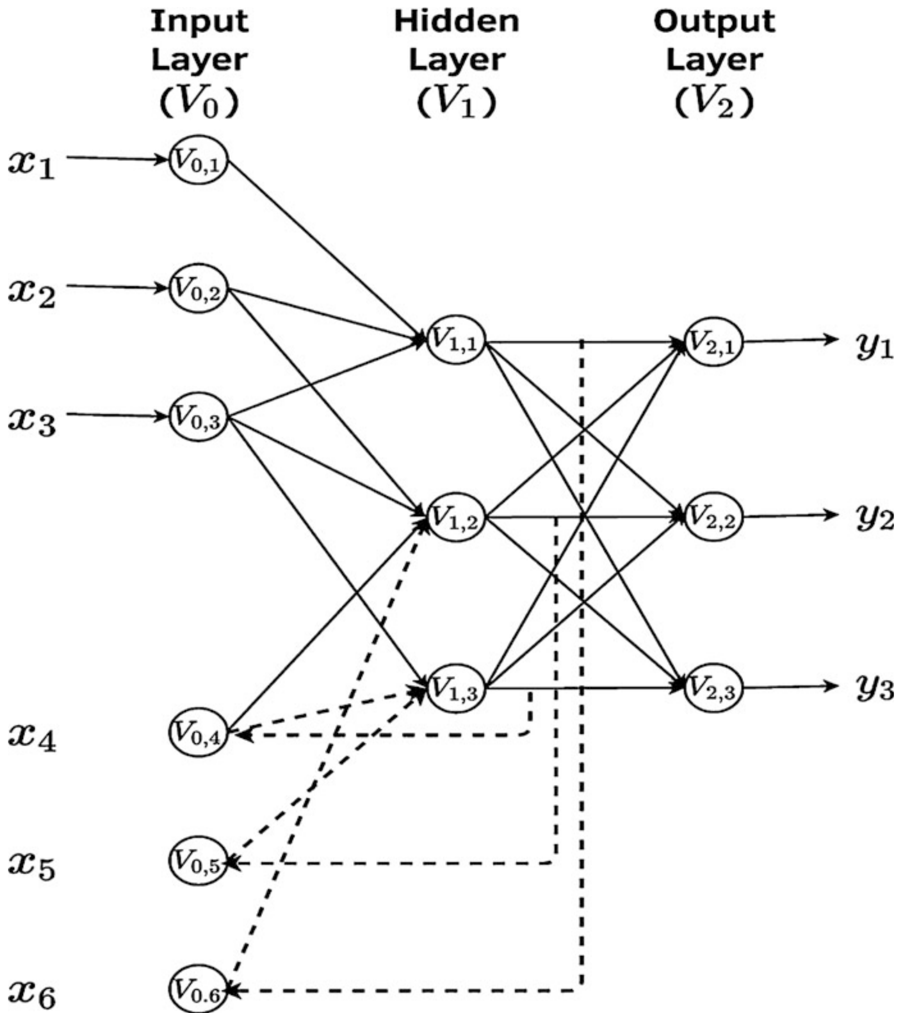



**Fig. 10.14** A two-layer recurrent artificial neural network with multivariate outputs

(p) Automatic image captioning (Chollet and Allaire 2017)
(q) Predicting earthquakes (Rouet-Leduc et al. 2017)
(r) Genomic prediction (Montesinos-López et al. 2018a, b)

It is important to point out that the applications of ANN and DL are not restricted to perception and natural language understanding, such as formal reasoning. There are also many successful applications in biological science. For example, deep learning has been successfully applied for predicting univariate continuous traits (Montesinos-López et al. 2018a), multivariate continuous traits (Montesinos-López et al. 2018b), univariate ordinal traits (Montesinos-López et al. 2019a), and

multivariate traits with mixed outcomes (Montesinos-López et al. 2019b) in the context of genomic-based prediction. Menden et al. (2013) applied a DL method to predict the viability of a cancer cell line exposed to a drug. Alipanahi et al. (2015) used DL with a convolutional network architecture (an ANN with convolutional operations; see Chap. 13) to predict specificities of DNA- and RNA-binding proteins. Tavanaei et al. (2017) used a DL method for predicting tumor suppressor genes and oncogenes. DL methods have also made accurate predictions of single-cell DNA methylation states (Angermueller et al. 2016). In the area of genomic selection, we mention two reports only: (a) McDowell and Grant (2016) found that DL methods performed similarly to several Bayesian and linear regression techniques that are commonly employed for phenotype prediction and genomic selection in plant breeding and (b) Ma et al. (2017) also used a DL method with a convolutional neural network architecture to predict phenotypes from genotypes in wheat and found that the DL method outperformed the GBLUP method. However, a review of DL application to genomic selection is provided by Montesinos-López et al. (2021).

## 10.7 Loss Functions

Loss function (also known as objective function) in general terms is a function that maps an event or values of one or more variables onto a real number intuitively representing some "cost" associated with the event. An optimization problem seeks to minimize a loss function. An objective function is either a loss function or its negative (in specific domains, variously called a reward function, a profit function, a utility function, a fitness function, etc.), in which case now the goal is a maximization process. In the statistical machine learning domain, a loss function tries to quantify how close the predicted values produced by an artificial neural network or DL model are to the true values. That is, the loss function measures the quality of the network's output by computing a distance score between the observed and predicted values (Chollet and Allaire 2017). The basic idea is to calculate a metric based on the observed error between the true and predicted values to measure how well the artificial neural network model's prediction matches what was expected. Then these errors are averaged over the entire data set to provide only a single number that represents how the artificial neural network is performing with regard to its ideal. In looking for this ideal, it is possible to find the parameters (weights and biases) of the artificial neural network that will minimize the "loss" produced by the errors. Training ANN models with loss functions allows the use of optimization methods to estimate the required parameters. Although most of the time it is not possible to obtain an analytical solution to estimate the parameters, very often good approximations can be obtained using iterative optimization algorithms like gradient descent (Patterson and Gibson 2017). Next, we provide the most used loss functions for each type of response variable.

### 10.7.1 Loss Functions for Continuous Outcomes

***Sum of square error loss*** This loss function is appropriate for continuous response variables (outcomes), assuming that we want to predict L response variables. The error (difference between observed ($y_{ij}$) and predicted ($\widehat{y}_{ij}$) values) in a prediction is squared and summed over the number of observations, since the training of the network is not local but global. To capture all possible trends in the training data, the expression used for sum of square error (SSE) loss is

$$L(\boldsymbol{w}) = \frac{1}{2}\sum\nolimits_{i=1}^{n}\sum\nolimits_{j=1}^{L}\left(\widehat{y}_{ij} - y_{ij}\right)^2$$

Note that $n$ is the size of your data set, and $L$, the number of targets (outputs) the network has to predict. It is important to point out that when there is only one response variable, the $L$ is dropped. Also, the division by two is added for mathematical convenience (which will become clearer in the context of its gradient in backpropagation). One disadvantage of this loss function is that it is quite sensitive to outliers and, for this reason, other loss functions have been proposed for continuous response variables. With the loss function, it is possible to calculate the loss score, which is used as a feedback signal to adjust the weights of the artificial neural network; this process of adjusting the weights in ANN is illustrated in Fig. 10.15 (Chollet and Allaire 2017). It is also common practice to use as a loss function, the SSE divided by the training sample ($n$) multiplied by the number of outputs ($L$).

Figure 10.15 shows that in the learning process of an artificial neural network are involved the interaction of layers, input data, loss function which defines the feedback signal used for learning, and the optimizer which determines how the learning proceeds and uses the loss value to update the network's weights. Initially, the weights of the network are assigned small random values, but when this provides an output far from the ideal values, it also implies a high loss score. But at each iteration of the network process, the weights are adjusted a little to reduce the

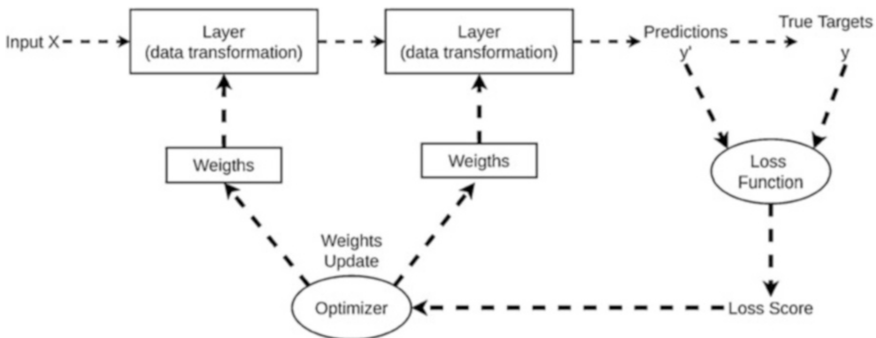



**Fig. 10.15** The loss score is used as a feedback signal to adjust the weights

difference between the observed and predicted values and, of course, to decrease the loss score. This is the basic step of the training process of statistical machine learning models in general, and when this process is repeated a sufficient number of times (on the order of thousands of iterations), it yields weight values that minimize the loss function. A network with minimal loss is one in which the observed and predicted values are very close; it is called a trained network (Chollet and Allaire 2017). There are other options of loss functions for continuous data like the sum of absolute percentage error loss (SAPE): $L(\boldsymbol{w})=\sum_{i=1}^{n}\sum_{j=1}^{L}\left|\frac{\widehat{y}_{ij}-y_{ij}}{y_{ij}}\right|$ and the sum of squared log error loss (Patterson and Gibson 2017): $L(\boldsymbol{w})=\sum_{i=1}^{n}\sum_{j=1}^{L}\left(\log\left(\widehat{y}_{ij}\right)-\log\left(y_{ij}\right)\right)^2$, but the SSE is popular in ANN and DL models due to its nice mathematical properties.

### 10.7.2 Loss Functions for Binary and Ordinal Outcomes

Next, we provide two popular loss functions for binary data: the hinge loss and the cross-entropy loss.

***Hinge loss*** This loss function originated in the context of the support vector machine for "maximum-margin" classification, and is defined as

$$L(\boldsymbol{w})=\sum_{i=1}^{n}\sum_{j=1}^{L}\max\left(0,y_{ij}\times\widehat{y}_{ij}\right)$$

It is important to point out that since this loss function is appropriate for binary data, the intended response variable output is denoted as +1 for success and −1 for failure.

***Logistic loss*** This loss function is defined as

$$L(\boldsymbol{w})=-\sum_{i=1}^{n}\sum_{j=1}^{L}\left[y_{ij}\times\log\left(\widehat{y}_{ij}\right)+\left(1-y_{ij}\right)\times\log\left(1-\widehat{y}_{ij}\right)\right]$$

This loss function originated as the negative log-likelihood of the product of Bernoulli distributions. It is also known as *cross-entropy* loss since we arrive at the logistic loss by calculating the cross-entropy (difference between two probability distributions) loss, which is a measure of the divergence between the predicted probability distribution and the true distribution. Logistic loss functions are preferred over the hinge loss when the scientist is mostly interested in the probabilities of success rather than in just the hard classifications. For example, when a scientist is interested in the probability that a patient can get cancer as a function of a set of covariates, the logistic loss is preferred since it allows calculating true probabilities.

When the number of classes is more than two according to Patterson and Gibson (2017), that is, when we are in the presence of categorical data, the loss function is known as categorical cross-entropy and is equal to

$$L(\boldsymbol{w}) = -\sum\nolimits_{i=1}^{n}\sum\nolimits_{j=1}^{L}\left[y_{ij} \times \log\left(\widehat{y}_{ij}\right)\right]$$

***Poisson loss*** This loss function is built as the minus log-likelihood of a Poisson distribution and is appropriate for predicting count outcomes. It is defined as

$$L(\boldsymbol{w}) = \sum_{i=1}^{n}\sum_{j=1}^{L}\left[\widehat{y}_{ij} - y_{ij}\log\left(\widehat{y}_{ij}\right)\right]$$

Also, for count data the loss function can be obtained under a negative binomial distribution, which can do a better job than the Poisson distribution when the assumption of equal mean and variance is hard to justify.

### *10.7.3 Regularized Loss Functions*

Regularization is a method that helps to reduce the complexity of the model and significantly reduces the variance of statistical machine learning models without any substantial increase in their bias. For this reason, to prevent overfitting and improve the generalizability of our models, we use regularization (penalization), which is concerned with reducing testing errors so that the model performs well on new data as well as on training data. Regularized or penalized loss functions are those that instead of minimizing the conventional loss function, $L(\boldsymbol{w})$, minimize an augmented loss function that consists of the sum of the conventional loss function and a penalty (or regularization) term that is a function of the weights. This is defined as

$$L(\boldsymbol{w}, \lambda) = L(\boldsymbol{w}) + 0.5 \times \lambda E_P,$$

where $L(\boldsymbol{w}, \lambda)$ is the regularized (or penalized) loss function, $\lambda$ is the degree or strength of the penalty term, and $E_P$ is the penalization proposed for the weights; this is known as the regularization term. The regularization term shrinks the weight estimates toward zero, which helps to reduce the variance of the estimates and increase the bias of the weights, which in turn helps to improve the out-of-sample predictions of statistical machine learning models (James et al. 2013). As you remember, the way to introduce the penalization term is using exactly the same logic used in Ridge regression in Chap. 3. Depending on the form of $E_P$, there is a name for the type of regularization. For example, when $E_P = \boldsymbol{w}^{\mathrm{T}}\boldsymbol{w}$, it is called Ridge penalty or weight decay penalty. This regularization is also called L2 penalty and has the effect that larger weights (positive or negative) result in larger penalties. On the

other hand, when $E_P = \sum_{p=1}^{P} |w_p|$, that is, when the $E_P$ term is equal to the sum of the absolute weights, the name of this regularization is Least Absolute Shrinkage and Selection Operator (Lasso) or simply L1 regularization. The L1 penalty produces a sparse solution (more zero weights) because small and larger weights are equally penalized and force some weights to be exactly equal to zero when the $\lambda$ is considerably large (James et al. 2013; Wiley 2016); for this reason, the Lasso penalization also performs variable selection and provides a model more interpretable than the Ridge penalty. By combining Ridge (L2) and Lasso (L1) regularization, we obtained Elastic Net regularization, where the loss function is defined as $L(\boldsymbol{w}, \lambda_1, \lambda_2) = L(\boldsymbol{w}) + 0.5 \times \lambda_1 \sum_{p=1}^{P} |w_p| + 0.5 \times \lambda_2 \sum_{p=1}^{P} w_p^2$, and where instead of one lambda parameter, two are needed.

It is important to point out that more than one hyperparameter is needed in ANN and DL models where different degrees of penalties can be applied to different layers and different hyperparameters. This differential penalization is sometimes desirable to improve the predictions in new data, but this has the disadvantage that more hyperparameters need to be tuned, which increases the computation cost of the optimization process (Wiley 2016).

In all types of regularization, when $\lambda = 0$ (or $\lambda_1 = \lambda_2 = 0$), the penalty term has no effect, but the larger the value of $\lambda$, the more the shrinkage and penalty grows and the weight estimates will approach zero. The selection of the appropriate value of $\lambda$ is challenging and critical; for this reason, $\lambda$ is also treated as a hyperparameter that needs to be tuned and is usually optimized by evaluating a range of possible $\lambda$ values through cross-validation. It is also important to point out that scaling the input data before implementing artificial neural networks is recommended, since the effect of the penalty depends on the size of the weights and the size of the weights depends on the scale of the data. Also, the user needs to recall from Chap. 3 where Ridge regression was presented, that the shrinkage penalty is applied to all the weights except the intercept or bias terms (Wiley 2016).

Another type of regularization that is very popular in ANN and DL is the dropout, which consists of setting to zero a random fraction (or percentage) of the weights of the input neurons or hidden neurons. Suppose that our original topology is like the topology given in Fig. 10.16.16a, where all the neurons are active (with weights different to zero), while when a random fraction of neurons is dropped out, this means that all its connections (weights) are set to zero and the topology with the dropout neurons (with weights set to zero) is observed in Fig. 10.16b. The contribution of those dropped out neurons to the activation of downstream neurons is temporarily removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. Dropout is only used during the training of a model but not when evaluating the skill of the model; it prevents the units from co-adapting too much.

This type of regularization is very simple and there is a lot of empirical evidence of its power to avoid overfitting. This regularization is quite new in the context of statistical machine learning and was proposed by Srivastava et al. (2014) in the paper *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. There are no

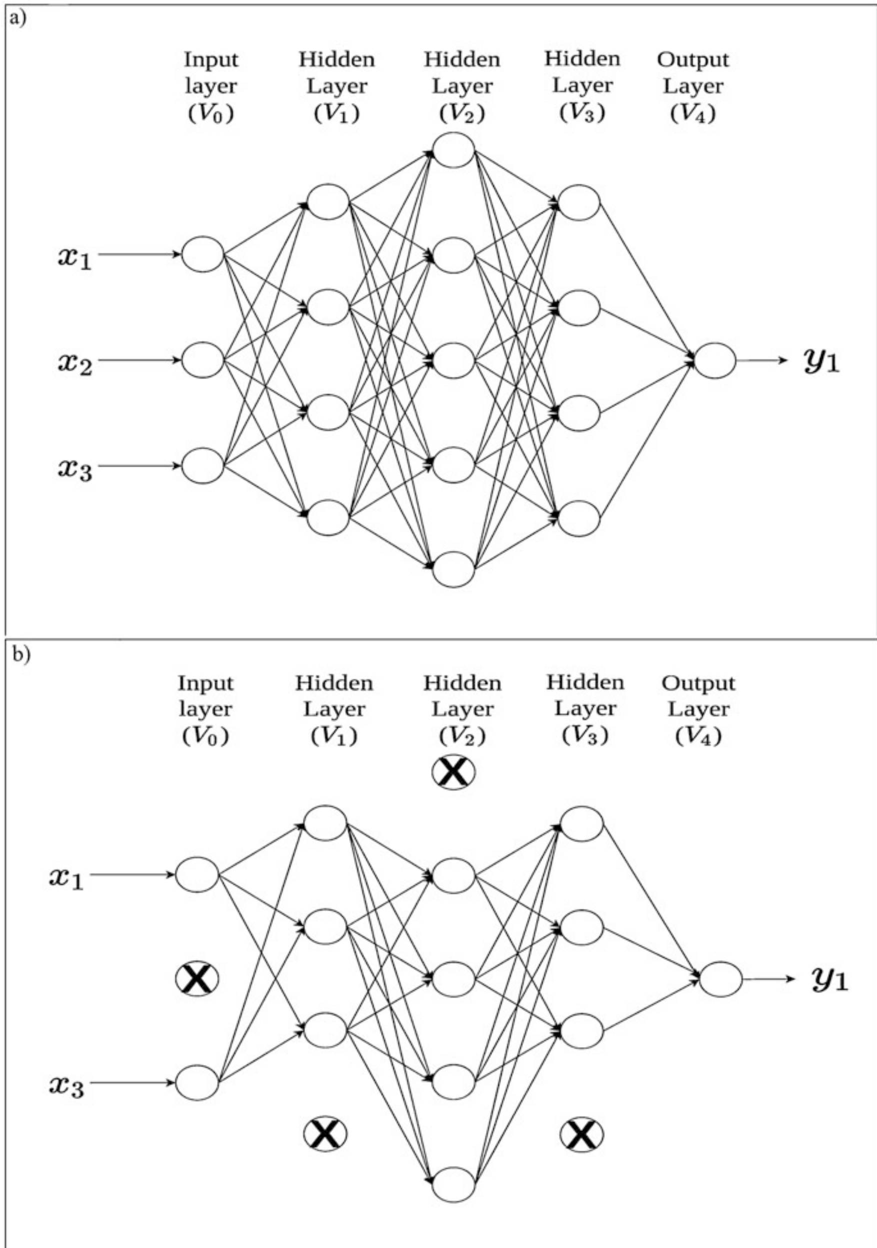



**Fig. 10.16** Feedforward neural network with four layers. (**a**) Three input neurons, four neurons in hidden layers 1 and 3, and five neurons in hidden layer 2 without dropout and (**b**) the same network with dropout; dropping out one in the input neuron, three neurons in hidden layers 1–3

unique rules to choose the percentage of neurons that will be dropped out. Some tips are given below to choose the % dropout:

(a) Usually a good starting point is to use 20% dropout, but values between 20% and 50% are reasonable. A percentage that is too low has minimal effect and a value that is too high results in underfitting the network.
(b) The larger the network, the better, when you use the dropout method, since then you are more likely to get a better performance, because the model had more chance to learn independent representations.
(c) Application of dropout is not restricted to hidden neurons; it can also be applied in the input layer. In both cases, there is evidence that it improves the performance of the ANN model.
(d) When using dropout, increasing the learning rate (learning rate is a tuning parameter in an optimization algorithm that regulates the step size at each epoch (iteration) while moving toward a minimum (or maximum) of a loss function) of the ANN algorithm by a factor of 10–100 is suggested, as well as increasing the momentum value (another tuning parameter useful for computing the gradient at each iteration), for example, from 0.90 to 0.99.
(e) When dropout is used, it is also a good idea to constrain the size of network weights, since the larger the learning rate, the larger the network weights. For this reason, constraining the size of network weights to less than five in absolute values with max-norm regularization has shown to improve results.

It is important to point out that all the loss functions described in the previous section can be converted to regularized (penalized) loss functions using the elements given in this section. The dropout method can also be implemented with any type of loss function.

### *10.7.4 Early Stopping Method of Training*

During the training process, the ANN and DL models learn in stages, from simple realizations to complex mapping functions. This process is captured by monitoring the behavior of the mean squared error that compares the match between observed and predicted values, which starts decreasing rapidly by increasing the number of epochs (epoch refers to one cycle through the full training data set) used for training, then decrease slowly when the error surface is close to a local minimum. However, to attain the larger generalization power of a model, it is necessary to figure out when it is best to stop training, which is a very challenging situation since a very early stopping point can produce underfitting, while a very late (no large) stopping point can produce overfitting of the training data. As mentioned in Chap. 4, one way to avoid overfitting is to use a CV strategy, where the training set is split into a training-inner and testing-inner set; with the training-inner set, the model is trained for the set of hyperparameters, and with the testing-inner (tuning) set, the power to predict out of sample data is evaluated, and in this way the optimal hyperparameters are

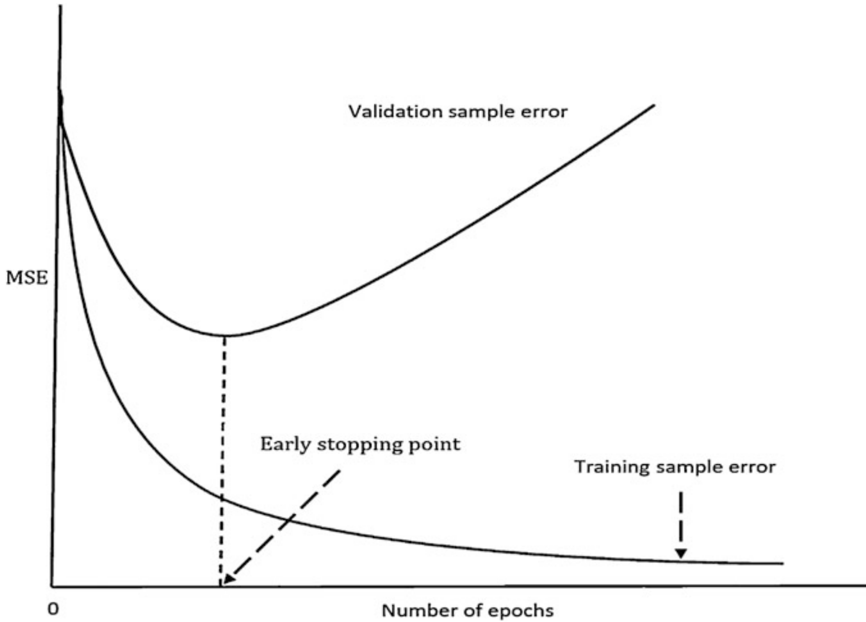



**Fig. 10.17** Schematic representation of the early stopping rule based on cross-validation (Haykin 2009)

obtained. However, we can incorporate the early stopping method to CV to fight better overfitting by using the CV strategy in the usual way with a minor modification, which consists of stopping the training section periodically (i.e., every so many epochs) and testing the model on the validation subset, after reaching the specified number of epochs (Haykin 2009). In other words, the stopping method combined with the CV strategy that consists of a periodic "estimation-followed-by-validation process" basically proceeds as follows:

(a) After a period of estimation (training)—every three epochs, for example—the weights and bias (intercept) parameters of the multilayer perceptron are all fixed, and the network is operated in its forward mode. Then the training and validation error are computed.
(b) When the validation prediction performance is completed, the estimation (training) is started again for another period, and the process is repeated.

Due to its nature (just described above), which is simple to understand and easy to implement in practice, this method is called *early stopping method of training*. To better understand this method, in Fig. 10.17 this approach is conceptualized with two learning curves, one for the training subset and the other for the validation subset. Figure 10.17 shows that the prediction power in terms of MSE is lower in the training set than in the validation set, which is expected. The *estimation learning curve* that corresponds to the training set decreases monotonically as the number of

epochs increases, which is normal, while the validation learning curve decreases monotonically to a minimum and then as the training continues, starts to increase. However, the estimation learning curve of the training set suggests that we can do better by going beyond the minimum point on the validation learning curve, but this is not really true since in essence what is learned beyond this point is the noise contained in the training data. For this reason, the minimum point on the validation learning curve could be used as a sensible criterion for stopping the training session. However, the validation sample error does *not* evolve as smoothly as the perfect curve shown in Fig. 10.17, over the number of epochs used for training, since the validation sample error many times exhibits few local minima of its own before it starts to increase with an increasing number of epochs. For this reason, in the presence of two or more local minima, the selection of a "slower" stopping criterion (i.e., a criterion that stops later than other criteria) makes it possible to attain a small improvement in generalization performance (typically, about 4%, on average) at the cost of a much longer training period (about a factor of four, on average).

## 10.8 The King Algorithm for Training Artificial Neural Networks: Backpropagation

The training process of ANN, which consists of adjusting connection weights, requires a lot of computational resources. For this reason, although they had been studied for many decades, few real applications of ANN were available until the mid-to-late 1980s, when the backpropagation method made its arrival. This method is attributed to Rumelhart et al. (1986). It is important to point out that, independently, other research teams around the same time published the backpropagation algorithm, but the one previously mentioned is one of the most cited. This algorithm led to the resurgence of ANN after the 1980s, but this algorithm is still considerably slower than other statistical machine learning algorithms. Some advantages of this algorithm are (a) it is able to make predictions of categorical or continuous outcomes, (b) it does a better job in capturing complex patterns than nearly any other algorithm, and (c) few assumptions about the underlying relationships of the data are made. However, this algorithm is not without weaknesses, some of which are (a) it is very slow to train since it requires a lot of computational resources because the more complex the network topology, the more computational resources are needed, this statement is true not only for ANN but also for any algorithm, (b) it is very susceptible to overfitting training data, and (c) its results are difficult to interpret (Lantz 2015).

Next, we provide the derivation of the backpropagation algorithm for the multilayer perceptron network shown in Fig. 10.18.

As mentioned earlier, the goal of the backpropagation algorithm is to find the weights of a multilayered feedforward network. The multilayered feedforward network given in Fig. 10.18 is able to approximate any function to any degree of

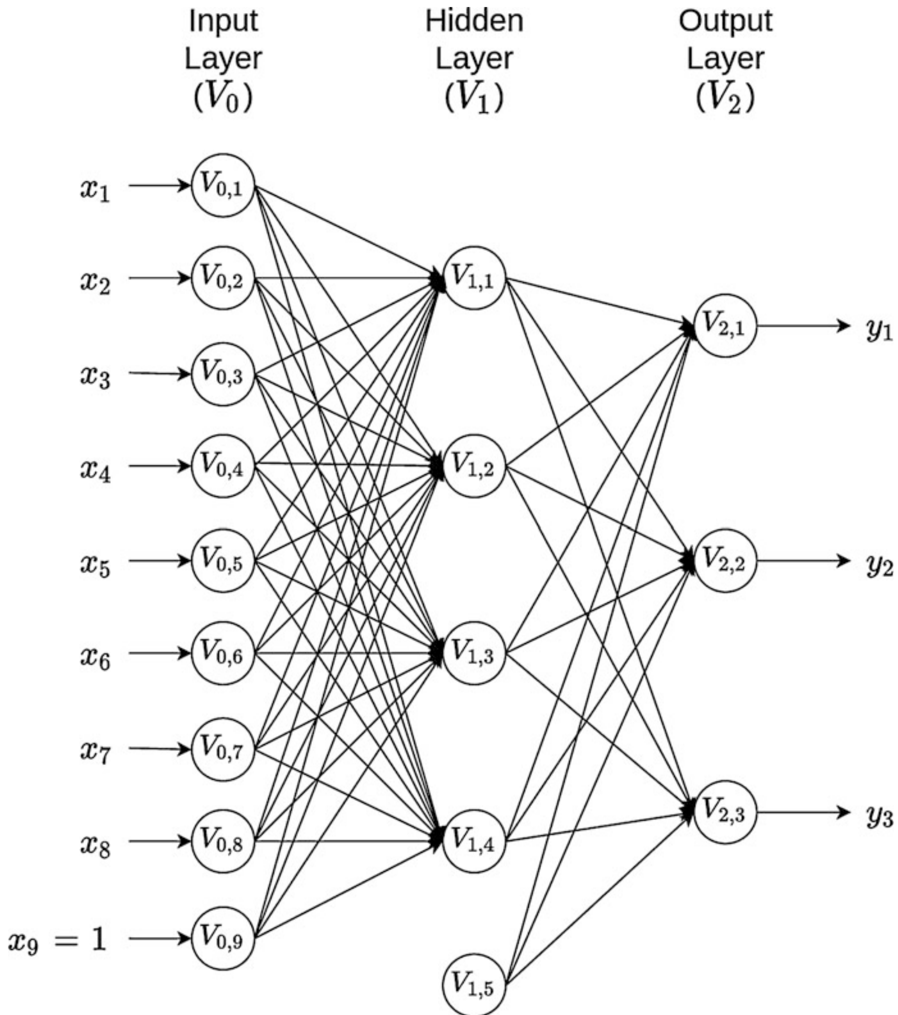



**Fig. 10.18** Schematic representation of a multilayer feedforward network with one hidden layer, eight input variables, and three output variables

accuracy (Cybenko 1989) with enough hidden units, as stated in the universal approximation theorem (Sect. 10.4), which makes the multilayered feedforward network a powerful statistical machine learning tool. Suppose that we provide this network with $n$ input patterns of the form

$$\boldsymbol{x}_i = [x_{i1}, \ldots, x_{iP}]^{\mathrm{T}},$$

where $\boldsymbol{x}_i$ denotes the input pattern of individual $i$ with $i = 1, \ldots, n$, and $x_{ip}$ denotes the input $p$th of $\boldsymbol{x}_i$. Let $y_{ij}$ denote the response variable of the $i$th individual for the $j$th

output and this is associated with the input pattern $\boldsymbol{x}_i$. For this reason, to be able to train the neural network, we must learn the functional relationship between the inputs and outputs. To illustrate the learning process of this relationship, we use the SSE loss function (explained in the section about "loss functions" to optimize the weights) which is defined as

$$E = \frac{1}{2}\sum\nolimits_{i=1}^{n}\sum\nolimits_{j=1}^{L}\left(\widehat{y}_{ij} - y_{ij}\right)^2 \tag{10.5}$$

Now, to explain how the backpropagation algorithm works, we will explain how information is first passed forward through the network. Providing the input values to the input layer is the first step, but no operation is performed on this information since it is simply passed to the hidden units. Then the net input into the $k$th hidden neuron is calculated as

$$z_{ik}^{(h)} = \sum\nolimits_{p=1}^{P} w_{kp}^{(h)} x_{ip} \tag{10.6}$$

Here $P$ is the total number of explanatory variables or input nodes, $w_{kp}^{(h)}$ is the weight from input unit $p$ to hidden unit $k$, the superscript, $h$, refers to hidden layer, and $x_{ip}$ is the value of the $p$th input for pattern or individual $i$. It is important to point out that the bias term ($b_j^{(h)}$) of neuron $k$ in the hidden layer has been excluded from (10.6) because the bias can be accounted for by adding an extra neuron to the input layer and fixing its value at 1. Then the output of the $k$ neuron resulting from applying an activation function to its net input is

$$V_{ik}^{(h)} = g^{(h)}\left(z_{ik}^{(h)}\right), \tag{10.7}$$

where $g^{(h)}$ is the activation function that is applied to the net input of any neuron $k$ of the hidden layer. In a similar vein, now with all the outputs of the neurons in the hidden layer, we can estimate the net input of the $j$th neuron of the output unit $j$ as

$$z_{ij}^{(l)} = \sum\nolimits_{k=1}^{M} w_{jk}^{(l)} V_{ik}^{(h)}, \tag{10.8}$$

where $M$ is the number of neurons in the hidden layer and $w_{jk}^{(l)}$ represents the weights from hidden unit $k$ to output $j$. The superscript, $l$,refers to output layer. Also, here the bias term ($b_j^{(l)}$) of neuron $j$ in the output layer was not included in (10.8) since it can be included by adding an extra neuron to the hidden layer and fixing its value at 1. Now, by applying the activation function to the output of the $j$th neuron of the output layer, we get the predicted value of the $j$th output as

$$\widehat{y}_{ij} = g^{(l)}\left(z_{ij}^{(l)}\right), \tag{10.9}$$

where $\widehat{y}_{ij}$ is the predicted value of individual $i$ in output $j$ and $g^{(l)}$ is the activation function of the output layer. We are interested in learning the weights ($w_{kp}^{(h)}$, $w_{jk}^{(l)}$) that minimize the sum of squared errors known as the mean square loss function (10.5), which is a function of the unknown weights, as can be observed in (10.6)–(10.8). Therefore, the partial derivatives of the loss function with respect to the weights represent the rate of change of the loss function with respect to the weights (this is the slope of the loss function). The loss function will decrease when moving the weights down this slope. This is the intuition behind the iterative method called backpropagation for finding the optimal weights and biases. This method consists of evaluating the partial derivatives of the loss function with regard to the weights and then moving these values down the slope, until the score of the loss function no longer decreases. For example, if we make the variation of the weights proportional to the negative of the gradient, the change in the weights in the right direction is reached. The gradient of the loss function given in (10.5) with respect to the weights connecting the hidden units to the output units ($w_{jk}^{(l)}$) is given by

$$\Delta w_{jk}^{(l)} = -\eta \frac{\partial E}{\partial w_{jk}^{(l)}}, \tag{10.10}$$

where $\eta$ is the learning rate that scales the step size and is specified by the user. To be able to calculate the adjustments for the weights connecting the hidden neurons to the outputs, $w_{jk}^{(l)}$, first we substitute (10.6)–(10.9) in (10.5), which yields

$$E = \frac{1}{2}\sum\nolimits_{i=1}^{n}\sum\nolimits_{j=1}^{L}\left(y_{ij} - g^{(l)}\left(\sum\nolimits_{k=1}^{M} w_{jk}^{(l)} g^{(h)}\left(\sum\nolimits_{p=1}^{P} w_{kp}^{(h)} x_{ip}\right)\right)\right)^2$$

Then, by expanding (10.10) using the change rule, we get

$$\Delta w_{jk}^{(l)} = -\eta \frac{\partial E}{\partial \widehat{y}_{ij}} \frac{\partial \widehat{y}_{ij}}{\partial z_{ij}^{(l)}} \frac{\partial z_{ij}^{(l)}}{\partial w_{jk}^{(l)}}$$

Next, we get each partial derivative

$$\frac{\partial E}{\partial \widehat{y}_{ij}} = -\left(y_{ij} - \widehat{y}_{ij}\right)$$

$$\frac{\partial \widehat{y}_{ij}}{\partial z_{ij}^{(l)}} = g^{(l)'}\left(z_{ij}^{(l)}\right) \tag{10.11}$$

$$\frac{\partial z_{ij}^{(l)}}{\partial w_{jk}^{(l)}} = V_{ik}^{(h)}$$

By substituting these partial derivatives in (10.10), we obtain the change in weights from the hidden units to the output units, $\Delta w_{jk}^{(l)}$, as

$$\Delta w_{jk}^{(l)} = \eta\left(y_{ij} - \widehat{y}_{ij}\right) g^{(l)'}\left(z_{ij}^{(l)}\right) V_{ik}^{(h)} = \eta \delta_{ij} V_{ik}^{(h)}, \tag{10.12}$$

where $\delta_{ij} = \left(y_{ij} - \widehat{y}_{ij}\right) g^{(l)'}\left(z_{ij}^{(l)}\right)$. Therefore, the formula used to update the weights from the hidden units to the output units is

$$w_{jk}^{(l)(t+1)} = w_{jk}^{(l)(t)} + \Delta w_{jk}^{(l)} = w_{jk}^{(l)(t)} + \eta \delta_{ij} V_{ik}^{(h)} \tag{10.13}$$

This equation reflects that the adjusted weights from (10.13) are added to the current estimate of the weights, $w_{jk}^{(l)(t)}$, to obtain the updated estimates, $w_{jk}^{(l)(t+1)}$.

Next, to update the weights connecting the input units to the hidden units, we follow a similar process as in (10.12). Thus

$$\Delta w_{kp}^{(h)} = -\eta \frac{\partial E}{\partial w_{kp}^{(h)}} \tag{10.14}$$

Using the chain rule, we get that

$$-\eta \frac{\partial E}{\partial w_{kp}^{(h)}} = \frac{\partial E}{\partial \widehat{y}_{ij}} \frac{\partial \widehat{y}_{ij}}{\partial z_{ij}^{(l)}} \frac{\partial z_{ij}^{(l)}}{\partial V_{ik}^{(h)}} \frac{\partial V_{ik}^{(h)}}{\partial z_{ik}^{(h)}} \frac{\partial z_{ik}^{(h)}}{\partial w_{kp}^{(h)}},$$

where $\frac{\partial E}{\partial \widehat{y}_{ij}}$ and $\frac{\partial \widehat{y}_{ij}}{\partial z_{ij}^{(l)}}$ are given in (10.11), while

$$\frac{\partial z_{ij}^{(l)}}{\partial V_{ik}^{(h)}} = w_{jk}^{(l)}$$

$$\frac{\partial V_{ik}^{(h)}}{\partial z_{ik}^{(h)}} = g^{(h)'}\left(z_{ik}^{(h)}\right) \tag{10.15}$$

$$\frac{\partial z_{ik}^{(h)}}{\partial w_{kp}^{(h)}} = x_{ip}$$

Substituting back into (10.14), we obtain the change in the weights from the input units to the hidden units, $\Delta w_{kp}^{(h)}$, as

$$\Delta w_{kp}^{(h)} = \eta \sum\nolimits_{j=1}^{L} \delta_{ij} w_{jk}^{(l)} g^{(h)'}\left(z_{ik}^{(h)}\right) x_{ip} = \eta \psi_{ik} x_{ip}, \tag{10.16}$$

where $\psi_{ik} = \sum\nolimits_{j=1}^{L} \delta_{ij} w_{jk}^{(l)} g^{(h)'}\left(z_{ik}^{(h)}\right)$. The summation over the number of output units is because each hidden neuron is connected to all the output units. Therefore, all the outputs should be affected if the weight connecting an input unit to a hidden unit changes. In a similar way, the formula for updating the weights from the input units to the hidden units is

$$w_{kp}^{(h)(t+1)} = w_{kp}^{(h)(t)} + \Delta w_{kp}^{(h)} = w_{kp}^{(h)(t)} + \eta \psi_{ik} x_{ip} \tag{10.17}$$

This equation also reflects that the adjusted weights from (10.17) are added to the current estimate of the weights, $w_{kp}^{(h)(t)}$, to obtain the updated estimates, $w_{kp}^{(h)(t+1)}$. Now we are able to put down the processing steps needed to compute the change in the network weights using the backpropagation algorithm. We define $\boldsymbol{w}$ as the entire collection of weights.

## *10.8.1 Backpropagation Algorithm: Online Version*

### 10.8.1.1 Feedforward Part

Step 1. Initialize the weights to small random values, and define the learning rate ($\eta$) and the minimum expected loss score (tol). By tol we can fix a small value that when this value is reached, the training process will stop.

Step 2. If the stopping condition is false, perform steps 3–14.

Step 3. Select a pattern $\boldsymbol{x}_i = [x_{i1}, \ldots, x_{iP}]^{\mathrm{T}}$ as the input vector sequentially ($i = 1$ till the number of samples) or at random.

Step 4. The net inputs of the hidden layer are calculated: $z_{ik}^{(h)} = \sum\nolimits_{p=0}^{P} w_{kp}^{(h)} x_{ip}$, $i = 1, \ldots, n$ and $k = 0, \ldots, M$.

Step 5. The outputs of the hidden layer are calculated: $V_{ik}^{(h)} = g^{(h)}\left(z_{ik}^{(h)}\right)$

Step 6. The net inputs of the output layer are calculated: $z_{ij}^{(l)} = \sum\nolimits_{k=0}^{M} w_{jk}^{(l)} V_{ik}^{(h)}, j = 1, \ldots, L$

Step 7. The predicted values (outputs) of the neural network are calculated: $\widehat{y}_{ij} = g^{(l)}\left(z_{ij}^{(l)}\right)$

Step 8. Compute the mean square error (loss function) for pattern $i$ error: $E_i = \frac{1}{2nL} \sum\nolimits_{j=1}^{L} \left(\widehat{y}_{ij} - y_{ij}\right)^2 + E_i$; then $E(\boldsymbol{w}) = E_i + E(\boldsymbol{w})$; in the first step of an epoch,

initialize $E_i = 0$. Note that the value of the loss function is accumulated over all data pairs, that is, $(y_{ij}, \boldsymbol{x}_i)$.

#### 10.8.1.2 Backpropagation Part

Step 9. The output errors are calculated: $\delta_{ij} = \left(y_{ij} - \widehat{y}_{ij}\right) g^{(l)'}\left(z_{ij}^{(l)}\right)$

Step 10. The hidden layer errors are calculated: $\psi_{ik} = g^{(h)'}\left(z_{ik}^{(h)}\right)\sum_{j=1}^{L}\delta_{ij}w_{jk}^{(l)}$

Step 11. The weights of the output layer are updated: $w_{jk}^{(l)(t+1)} = w_{jk}^{(l)(t)} + \eta\delta_{ij}V_{ik}^{(h)}$

Step 12. The weights of the hidden layer are updated: $w_{kp}^{(h)(t+1)} = w_{kp}^{(h)(t)} + \eta\psi_{ik}x_{ip}$

Step 13. If $i < n$, go to step 3; otherwise go to step 14.

Step 14. Once the learning of an epoch is complete, $i = n$; then we check if the global error is satisfied with the specified tolerance (tol). If this condition is satisfied we terminate the learning process which means that the network has been trained satisfactorily. Otherwise, go to step 3 and start a new learning epoch: $i = 1$, since $E(\boldsymbol{w}) < \text{tol}$.

The backpropagation algorithm is iterative. This means that the search process occurs over multiple discrete steps, each step hopefully slightly improving the model parameters. Each step involves using the model with the current set of internal parameters to make predictions of some samples, comparing the predictions to the real expected outcomes, calculating the error, and using the error to update the internal model parameters. This update procedure is different for different algorithms, but in the case of ANN, as previously pointed out, the backpropagation update algorithm is used.

### *10.8.2 Illustrative Example 10.1: A Hand Computation*

In this section, we provide a simple example that will be computed step by step by hand to fully understand how the training is done using the backpropagation method. The topology used for this example is given in Fig. 10.19.

The data set for this example is given in Table 10.1, where we can see that the data collected consist of four observations, the response variable ($y$) takes values

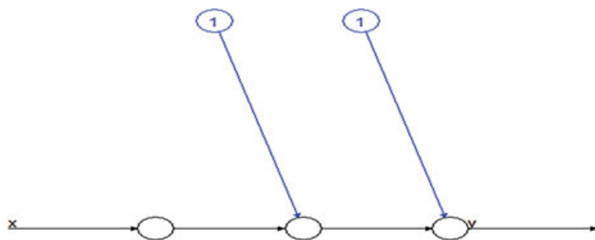



**Fig. 10.19** A simple artificial neural network with one input, one hidden layer with one neuron, and one response variable (output)

**Table 10.1** Input ($X$) and response variable ($y$) for four individuals (observations) and initial weights

| Observation | $X$ | $y$ | Output weights ($w_{jk}^{(l)}$) | Hidden weights ($w_{kp}^{(h)}$) |
|---|---|---|---|---|
| 1 | 0.33 | 0.9 | −1.5 | 1.86 |
| 2 | 0.95 | 0.6 | 4.4 | −3.3 |
| 3 | 0.27 | 0.95 | | |
| 4 | 1.3 | 0.7 | | |

between 0 and 1, and the input information is for only one predictor ($x$). Additionally, Table 10.1 gives the starting values for the hidden weights ($w_{kp}^{(h)}$) and for the output weights ($w_{jk}^{(l)}$). It is important to point out that due to the fact that the response variable is in the interval between zero and one, we will use the sigmoid activation function for both the hidden layer and the output layer. A learning rate ($\eta$) equal to 0.1 and tolerance equal to 0.025 were also used.

The backpropagation algorithm described before was given for one input pattern at a time; however, to simplify the calculations, we will implement this algorithm using the four patterns of data available simultaneously using matrix calculations. For this reason, first we build the design matrix of inputs and outputs:

$$\boldsymbol{X} = \begin{bmatrix} 1 & 0.33 \\ 1 & 0.95 \\ 1 & 0.27 \\ 1 & 1.3 \end{bmatrix}, \quad \boldsymbol{y} = \begin{bmatrix} 0.9 \\ 0.6 \\ 0.95 \\ 0.7 \end{bmatrix}$$

We also define the vectors of the starting values of the hidden and output weights:

$$\boldsymbol{w}^{(h)} = \begin{bmatrix} 1.86 \\ -3.3 \end{bmatrix}, \quad \boldsymbol{w}^{(l)} = \begin{bmatrix} -1.5 \\ 4.4 \end{bmatrix}.$$

Here we can see that $P = 1$,and $M = 2$. Next we calculate the net inputs for the hidden layer as

$$\boldsymbol{z}^{(h)} = \boldsymbol{X}\boldsymbol{w}^{(h)} = \begin{bmatrix} 1 & 0.33 \\ 1 & 0.95 \\ 1 & 0.27 \\ 1 & 1.2 \end{bmatrix} \begin{bmatrix} 1.86 \\ -3.3 \end{bmatrix} = \begin{bmatrix} 0.771 \\ -1.275 \\ 0.969 \\ -2.430 \end{bmatrix}$$

Now the output for the hidden layer is calculated using the sigmoid activation function

$$\boldsymbol{V}^{(h)} = \begin{bmatrix} V_{11}^{(h)} = 1/(1 + \exp(-z_{11})) \\ V_{21}^{(h)} = 1/(1 + \exp(-z_{21})) \\ V_{31}^{(h)} = 1/(1 + \exp(-z_{31})) \\ V_{41}^{(h)} = 1/(1 + \exp(-z_{41})) \end{bmatrix} = \begin{bmatrix} 0.6837 \\ 0.2184 \\ 0.7249 \\ 0.0809 \end{bmatrix},$$

where $V_{ik}^{(h)} = g^{(h)}\left(z_{ik}^{(h)}\right), i = 1, \ldots, 4$ and $g^{(h)}(z) = 1/(1 + \exp(-z))$, which can be replaced by another desired activation function. Then the net inputs for the output layer are calculated as follows:

$$\boldsymbol{z}^{(l)} = \left[\mathbf{1}, \boldsymbol{V}^{(h)}\right]\boldsymbol{w}^{(l)} = \begin{bmatrix} 1 & 0.6837 \\ 1 & 0.2184 \\ 1 & 0.7249 \\ 1 & 0.0809 \end{bmatrix} \begin{bmatrix} -1.5 \\ 4.4 \end{bmatrix} = \begin{bmatrix} 1.5084 \\ -0.5390 \\ 1.6896 \\ -1.1440 \end{bmatrix}$$

The predicted values (outputs) of the neural network are calculated as

$$\widehat{\boldsymbol{y}} = \begin{bmatrix} \widehat{y}_1 = 1/(1 + \exp(-z_1)) \\ \widehat{y}_2 = 1/(1 + \exp(-z_2)) \\ \widehat{y}_3 = 1/(1 + \exp(-z_3)) \\ \widehat{y}_4 = 1/(1 + \exp(-z_4)) \end{bmatrix} = \begin{bmatrix} 0.8188 \\ 0.3684 \\ 0.8442 \\ 0.2416 \end{bmatrix},$$

where $\widehat{y}_i = g^{(l)}\left(z_{i1}^{(l)}\right), i = 1, \ldots, 4$ and $g^{(l)}(z) = 1/(1 + \exp(-z))$. Next the output errors are calculated using the Hadamard product, $\circ$, (element-wise matrix multiplication) as

$$\boldsymbol{\delta}^{(l)} = (\boldsymbol{y} - \widehat{\boldsymbol{y}})\circ\widehat{\boldsymbol{y}}\circ(1 - \widehat{\boldsymbol{y}})$$

$$= \left(\begin{matrix} 0.9 \\ 0.6 \\ 0.95 \\ 0.7 \end{matrix} - \begin{matrix} 0.8188 \\ 0.3684 \\ 0.8442 \\ 0.2416 \end{matrix}\right) \circ \begin{pmatrix} 0.8188 \\ 0.3684 \\ 0.8442 \\ 0.2416 \end{pmatrix} \circ \left(\begin{matrix} 1 \\ 1 \\ 1 \\ 1 \end{matrix} - \begin{matrix} 0.8188 \\ 0.3684 \\ 0.8442 \\ 0.2416 \end{matrix}\right) = \begin{bmatrix} 0.0120 \\ 0.0539 \\ 0.0139 \\ 0.0839 \end{bmatrix}$$

The hidden layer errors are calculated as

$$\boldsymbol{\psi} = \left[\boldsymbol{V}^{(h)} \circ \left(1 - \boldsymbol{V}^{(h)}\right)\right] \circ \boldsymbol{\delta}^{(l)} \boldsymbol{w}_1^{T(l)} = \left[\begin{pmatrix} 0.6837 \\ 0.2184 \\ 0.7249 \\ 0.0809 \end{pmatrix} \circ \left(\begin{matrix} 1 \\ 1 \\ 1 \\ 1 \end{matrix} - \begin{matrix} 0.6837 \\ 0.2184 \\ 0.7249 \\ 0.0809 \end{matrix}\right)\right] \circ \left\{ \begin{bmatrix} 0.0120 \\ 0.0539 \\ 0.0139 \\ 0.0839 \end{bmatrix} [4.4] \right\}$$

$$= \begin{bmatrix} 0.0114 \\ 0.0405 \\ 0.0122 \\ 0.0275 \end{bmatrix},$$

where $\boldsymbol{w}_1^{(l)}$ is $\boldsymbol{w}^{(l)}$ without the weight of the intercept, that is, without the first element.

The weights of the output layer are updated:

$$\boldsymbol{w}^{(l)(2)} = \boldsymbol{w}^{(l)(1)} + \eta \left[\mathbf{1}, \boldsymbol{V}^{(h)}\right]^{\mathbf{T}} \boldsymbol{\delta}^{(l)}$$

$$\boldsymbol{w}^{(l)(2)} = \begin{bmatrix} -1.5 \\ 4.4 \end{bmatrix} + 0.1 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0.6837 & 0.2184 & 0.7249 & 0.0809 \end{bmatrix} \begin{bmatrix} 0.0120 \\ 0.0539 \\ 0.0139 \\ 0.0839 \end{bmatrix}$$

$$= \begin{bmatrix} -1.4836 \\ 4.4037 \end{bmatrix},$$

where 2 denotes that the output weights are for epoch number 2. Then the weights for epoch 2 of the hidden layer are obtained with

$$\boldsymbol{w}^{(h)(2)} = \boldsymbol{w}^{(h)(1)} + \eta \boldsymbol{X}^{\mathrm{T}} \boldsymbol{\psi}$$

$$\boldsymbol{w}^{(h)(2)} = \begin{bmatrix} 1.86 \\ -3.3 \end{bmatrix} + 0.1 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0.33 & 0.95 & 0.27 & 1.3 \end{bmatrix} \begin{bmatrix} 0.0114 \\ 0.0405 \\ 0.0122 \\ 0.0275 \end{bmatrix} = \begin{bmatrix} 1.8692 \\ -3.2918 \end{bmatrix}$$

We check to see if the global error is satisfied with the specified tolerance (tol). Since $E(\boldsymbol{w}) = \frac{1}{2n} \sum_{i=1}^{n} (\widehat{y}_i - y_i)^2 = 0.03519 > \text{tol} = 0.025,$ this means that we need to increase the number of epochs to satisfy the $\text{tol} = 0.025$ specified.

Epoch 2. Using the updated weights of epoch 1, we obtain the new weights after epoch 2. First for the output layer:

$$\boldsymbol{w}^{(l)(3)} = \begin{bmatrix} -1.4836 \\ 4.4037 \end{bmatrix} + 0.1 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0.6863 & 0.2213 & 0.7272 & 0.0824 \end{bmatrix} \begin{bmatrix} 0.0111 \\ 0.0526 \\ 0.0132 \\ 0.0842 \end{bmatrix}$$

$$= \begin{bmatrix} -1.4675 \\ 4.4073 \end{bmatrix}$$

And next for the hidden layer:

$$\boldsymbol{w}^{(h)(3)} = \begin{bmatrix} 1.8692 \\ -3.2918 \end{bmatrix} + 0.1 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0.33 & 0.95 & 0.27 & 1.3 \end{bmatrix} \begin{bmatrix} 0.0106 \\ 0.0399 \\ 0.0115 \\ 0.0280 \end{bmatrix}$$

$$= \begin{bmatrix} 1.8782 \\ -3.2838 \end{bmatrix}$$

Now the predicted values are $\widehat{y}_1 = 0.8233$, $\widehat{y}_2 = 0.3754$, $\widehat{y}_3 = 0.8480$, and $\widehat{y}_4 = 0.2459$, and again we found that $E(\boldsymbol{w}) = \frac{1}{2n}\sum_{i=1}^{n}(\widehat{y}_i - y_i)^2 = 0.03412 > \text{tol} = 0.025$. This means that we need to continue the number of epochs to be able to satisfy the tol $= 0.025$ specified. The learning process by decreasing the MSE is observed in Fig. 10.20, where we can see that tol $= 0.025$ is reached in epoch number 13, with an MSE $= E(\boldsymbol{w}) = 0.02425$.

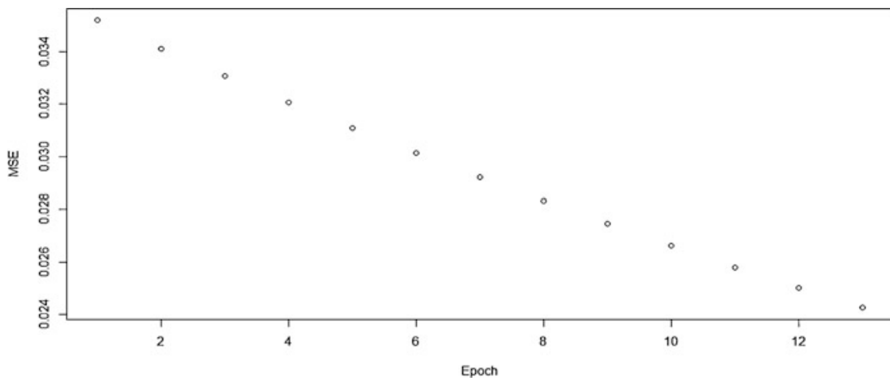



**Fig. 10.20** Behavior of the learning process by monitoring the MSE for Example 10.1—a hand computation

### 10.8.3 *Illustrative Example 10.2—By Hand Computation*

Table 10.2 gives the information for this example; the data collected contain five observations, the response variable ($y$) has a value between $-1$ and 1, and there are three inputs (predictors). Table 10.2 also provides the starting values for the hidden weights ($w_{kp}^{(h)}$) and for the output weights ($w_{jk}^{(l)}$). Due to the fact that the response variable is in the interval between $-1$ and 1, we will use the hyperbolic tangent activation function (Tanh) for the hidden and output layers. Now we used a learning rate ($\eta$) equal to 0.05 and a tolerance equal to 0.008 (Fig. 10.21).

Here the backpropagation algorithm was implemented using the five patterns of data simultaneously using matrix calculations. Again, first we represent the design matrix of inputs and outputs:

**Table 10.2** Inputs ($x1$, $x2$, and $x3$) and response variable ($y$) for four observations and initial weights

| Observation | $X = \text{c}(x1,x2,x3)$ | $y$ | Output weights ($w_{jk}^{(l)}$) | Hidden weights ($w_{1p}^{(h)}, w_{2p}^{(h)}$) |
|---|---|---|---|---|
| 1 | 0.15,0.20,0.37 | 0.88 | −1.5 | 1.4, 0.6 |
| 2 | 0.05,0.30,0.55 | 0.20 | 3.9 | −1.3,−0.48 |
| 3 | 0.45,0.20,0.42 | −0.8 | 0.27 | −0.8,0.06 |
| 4 | 0.35,0.10,0.22 | 0.62 | | −1.2,0.009 |
| 5 | 0.30,0.41,0.70 | −0.8 | | |

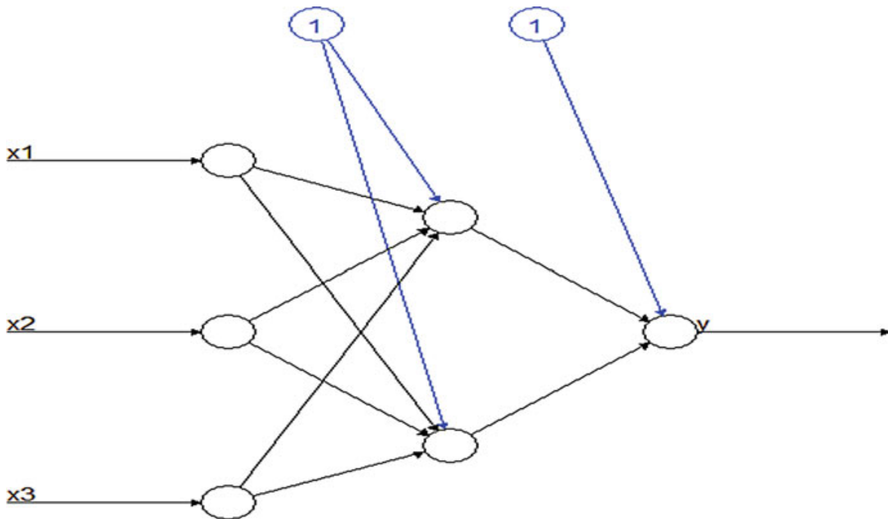



**Fig. 10.21** A simple artificial neural network with three inputs, one hidden layer with two neurons, and one response variable (output)

$$X = \begin{bmatrix} 1 & 0.15 & 0.20 & 0.37 \\ 1 & 0.05 & 0.30 & 0.55 \\ 1 & 0.45 & 0.20 & 0.42 \\ 1 & 0.35 & 0.10 & 0.22 \\ 1 & 0.30 & 0.41 & 0.70 \end{bmatrix}, \quad y = \begin{bmatrix} 0.88 \\ 0.20 \\ -0.8 \\ 0.62 \\ -0.8 \end{bmatrix}$$

Then we define the vectors of starting values of the hidden ($w^{(h)}$) and output ($w^{(l)}$) weights:

$$w^{(h)} = \begin{bmatrix} 1.4 & 0.6 \\ -1.3 & -0.48 \\ -0.8 & 0.060 \\ -1.2 & 0.009 \end{bmatrix}, \quad w^{(l)} = \begin{bmatrix} -1.5 \\ 3.9 \\ 0.27 \end{bmatrix}.$$

Now $P = 3$ and $M = 3$. Next, we calculate the net inputs for the hidden layer as

$$z^{(h)} = Xw^{(h)} = \begin{bmatrix} 1 & 0.15 & 0.20 & 0.37 \\ 1 & 0.05 & 0.30 & 0.55 \\ 1 & 0.45 & 0.20 & 0.42 \\ 1 & 0.35 & 0.10 & 0.22 \\ 1 & 0.30 & 0.41 & 0.70 \end{bmatrix} \begin{bmatrix} 1.4 & 0.6 \\ -1.3 & -0.48 \\ -0.8 & 0.060 \\ -1.2 & 0.009 \end{bmatrix}$$

$$= \begin{bmatrix} 0.601 & 0.5433 \\ 0.435 & 0.5989 \\ 0.151 & 0.3998 \\ 0.601 & 0.4399 \\ -0.158 & 0.4869 \end{bmatrix}$$

Now with the *tanh* activation function, the output of the hidden layer is calculated:

$$V^{(h)} = \begin{bmatrix} V_{11}^{(h)} = \tanh(z_{11}) & V_{12}^{(h)} = \tanh(z_{12}) \\ V_{21}^{(h)} = \tanh(z_{21}) & V_{22}^{(h)} = \tanh(z_{22}) \\ V_{31}^{(h)} = \tanh(z_{31}) & V_{32}^{(h)} = \tanh(z_{32}) \\ V_{41}^{(h)} = \tanh(z_{41}) & V_{42}^{(h)} = \tanh(z_{42}) \\ V_{51}^{(h)} = \tanh(z_{51}) & V_{52}^{(h)} = \tanh(z_{52}) \end{bmatrix} = \begin{bmatrix} 0.5378 & 0.4955 \\ 0.4095 & 0.5363 \\ 0.1499 & 0.3798 \\ 0.5378 & 0.4136 \\ -0.1567 & 0.4518 \end{bmatrix}$$

Again $V_{ik}^{(h)} = g^{(h)}\left(z_{ik}^{(h)}\right), i = 1; k = 1, 2$, and $g^{(h)}(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$, which can also be replaced by another activation function. Then the net inputs for the output layer are calculated as

$$\boldsymbol{z}^{(l)} = \left[\mathbf{1}, \boldsymbol{V}^{(h)}\right]\boldsymbol{w}^{(l)} = \begin{bmatrix} 1 & 0.5378 & 0.4955 \\ 1 & 0.4095 & 0.5363 \\ 1 & 0.1499 & 0.3798 \\ 1 & 0.5378 & 0.4136 \\ 1 & -0.1567 & 0.4518 \end{bmatrix} \begin{bmatrix} -1.5 \\ 3.9 \\ 0.27 \end{bmatrix} = \begin{bmatrix} 0.7311 \\ 0.2418 \\ -0.8130 \\ 0.7089 \\ -1.9891 \end{bmatrix}$$

The predicted values (outputs) of the neural network are calculated as

$$\widehat{\boldsymbol{y}} = \begin{bmatrix} \widehat{y}_{11} = \tanh(z_{11}) \\ \widehat{y}_{21} = \tanh(z_{21}) \\ \widehat{y}_{31} = \tanh(z_{31}) \\ \widehat{y}_{41} = \tanh(z_{41}) \\ \widehat{y}_{51} = \tanh(z_{51}) \end{bmatrix} = \begin{bmatrix} 0.6237 \\ 0.2372 \\ -0.6712 \\ 0.6100 \\ -0.9633 \end{bmatrix},$$

where $\widehat{y}_{ij} = g^{(l)}\left(z_{i1}^{(l)}\right), i = 1, \ldots, 5$ and $g^{(l)}(z) = \tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$. The output errors are calculated as

$$\boldsymbol{\delta}^{(l)} = (\boldsymbol{y} - \widehat{\boldsymbol{y}}) \circ \left(1 - \widehat{\boldsymbol{y}}^2\right) = \left(\begin{matrix} 0.88 \\ 0.20 \\ -0.8 \\ 0.62 \\ -0.8 \end{matrix} - \begin{matrix} 0.6237 \\ 0.2372 \\ -0.6712 \\ 0.6100 \\ -0.9633 \end{matrix}\right) \circ \left(\begin{matrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{matrix} - \begin{matrix} 0.3890 \\ 0.0563 \\ 0.4506 \\ 0.3721 \\ 0.9279 \end{matrix}\right)$$

$$= \begin{pmatrix} 0.1566 \\ -0.0351 \\ -0.0707 \\ 0.0063 \\ 0.0118 \end{pmatrix}$$

The hidden layer errors are calculated as

$$\boldsymbol{\psi} = \left[\left(1 - \boldsymbol{V}^{(h)2}\right)\right] \circ \boldsymbol{\delta}^{(l)} \boldsymbol{w}_1^{\mathrm{T}(h)}$$

$$= \begin{pmatrix} 0.7108 & 0.7545 \\ 0.8323 & 0.7124 \\ 0.9775 & 0.8558 \\ 0.7108 & 0.8289 \\ 0.9754 & 0.7959 \end{pmatrix} \circ \left[\begin{pmatrix} 0.1566 \\ -0.0351 \\ -0.0707 \\ 0.0063 \\ 0.0118 \end{pmatrix} \begin{bmatrix} 3.9 & 0.27 \end{bmatrix}\right]$$

$$= \begin{bmatrix} 0.4341 & 0.0319 \\ -0.1139 & -0.0068 \\ -0.2697 & -0.01635 \\ 0.0174 & 0.0014 \\ 0.0448 & 0.0025 \end{bmatrix},$$

where $\boldsymbol{w}_1^{(h)}$ is $\boldsymbol{w}^{(h)}$ without the weights of the intercepts, that is, without the first row.

The weights of the output layer are updated:

$$\boldsymbol{w}^{(l)(2)} = \boldsymbol{w}^{(l)(1)} + \eta\left[\mathbf{1}, \boldsymbol{V}^{(h)}\right]^{\mathbf{T}} \boldsymbol{\delta}^{(l)}$$

$$\boldsymbol{w}^{(l)(2)} = \begin{bmatrix} -1.5 \\ 3.9 \\ 0.27 \end{bmatrix} + 0.05 \begin{bmatrix} 1 & 1 & & 1 & 1 & 1 \\ 0.5378 & 0.4095 & 0.1499 & 0.5378 & -0.1567 \\ 0.4955 & 0.5363 & 0.3798 & 0.4136 & 0.4518 \end{bmatrix}$$

$$\times \begin{pmatrix} 0.1566 \\ -0.0351 \\ -0.0707 \\ 0.0063 \\ 0.0118 \end{pmatrix} = \begin{bmatrix} -1.4965 \\ 3.9030 \\ 0.2720 \end{bmatrix}$$

Number 2 in $\boldsymbol{w}^{(l)(2)}$ indicates that output weights are for epoch number 2. The weights of the hidden layer in epoch 2 are obtained with

$$\boldsymbol{w}^{(h)(2)} = \boldsymbol{w}^{(h)(1)} + \eta \boldsymbol{X}^{\mathrm{T}} \boldsymbol{\psi}$$

$$\boldsymbol{w}^{(h)(2)} = \begin{bmatrix} 1.4 & 0.6 \\ -1.3 & -0.48 \\ -0.8 & 0.060 \\ -1.2 & 0.009 \end{bmatrix} + 0.05 \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0.15 & 0.05 & 0.45 & 0.35 & 0.30 \\ 0.20 & 0.30 & 0.20 & 0.10 & 0.41 \\ 0.37 & 0.55 & 0.42 & 0.22 & 0.70 \end{bmatrix}$$

$$\times \begin{bmatrix} 0.4341 & 0.0319 \\ -0.1139 & -0.0068 \\ -0.2697 & -0.01635 \\ 0.0174 & 0.0014 \\ 0.0448 & 0.0025 \end{bmatrix}$$

$$= \begin{bmatrix} 1.4056 & 0.6006 \\ -1.3021 & -0.4801 \\ -0.7990 & 0.0601 \\ -1.1990 & 0.00917 \end{bmatrix}$$

We check to see if the global errors are satisfied with the specified tolerance (tol). $E(\boldsymbol{w}) = \frac{1}{2n}\sum_{i=1}^{n}(\widehat{y}_i - y_i)^2 = 0.01104 > \text{tol} = 0.008$ which means that we have to continue with the next epoch by cycling the training data again.

Epoch 2. Using the updated weights of epoch 1, we obtain the new weights for epoch 2.

For the output layer, these are

$$\boldsymbol{w}^{(l)(3)} = \begin{bmatrix} -1.4965 \\ 3.9030 \\ 0.2720 \end{bmatrix} + 0.05 \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0.5419 & 0.4147 & 0.1550 & 0.5414 & -0.1508 \\ 0.4960 & 0.5368 & 0.3804 & 0.4142 & 0.4524 \end{bmatrix}$$

$$\times \begin{pmatrix} 0.1442 \\ -0.0578 \\ -0.0810 \\ -0.0016 \\ 0.0123 \end{pmatrix}$$

$$= \begin{bmatrix} -1.4958 \\ 3.9050 \\ 0.2727 \end{bmatrix}$$

While for the hidden layer, they are

$$\boldsymbol{w}^{(h)(3)} = \begin{bmatrix} 1.4056 & 0.6006 \\ -1.3021 & -0.4801 \\ -0.7990 & 0.0601 \\ -1.1990 & 0.00917 \end{bmatrix} + 0.05 \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 0.15 & 0.05 & 0.45 & 0.35 & 0.30 \\ 0.20 & 0.30 & 0.20 & 0.10 & 0.41 \\ 0.37 & 0.55 & 0.42 & 0.22 & 0.70 \end{bmatrix}$$

$$\times \begin{bmatrix} 0.3975 & 0.0296 \\ -0.1867 & -0.0112 \\ -0.3087 & -0.01885 \\ -0.0045 & -0.00037 \\ 0.0468 & 0.0027 \end{bmatrix} = \begin{bmatrix} 1.4028 & 0.6007 \\ -1.3059 & 0.4803 \\ -0.8000 & 0.06011 \\ -1.2017 & 0.0091 \end{bmatrix}$$

Now the predicted values are $\widehat{y}_1 = 0.6372$, $\widehat{y}_2 = 0.2620$, $\widehat{y}_3 = -0.6573$, $\widehat{y}_4 = 0.6226$, $\widehat{y}_5 = -0.9612$, and the $E(\boldsymbol{w}) = \frac{1}{2n}\sum_{i=1}^{n}(\widehat{y}_i - y_i)^2 = 0.01092 > \text{tol} = 0.008$, which means that we have to continue with the next epoch by cycling the training data again. Figure 10.22 shows that the $E(\boldsymbol{w}) = 0.00799 < \text{tol} = 0.008$ until epoch 83.

In this algorithm, zero weights are not an option because each layer is symmetric in the weights flowing to the different neurons. Then the starting values should be close to zero and can be taken from random uniform or Gaussian distributions (Efron

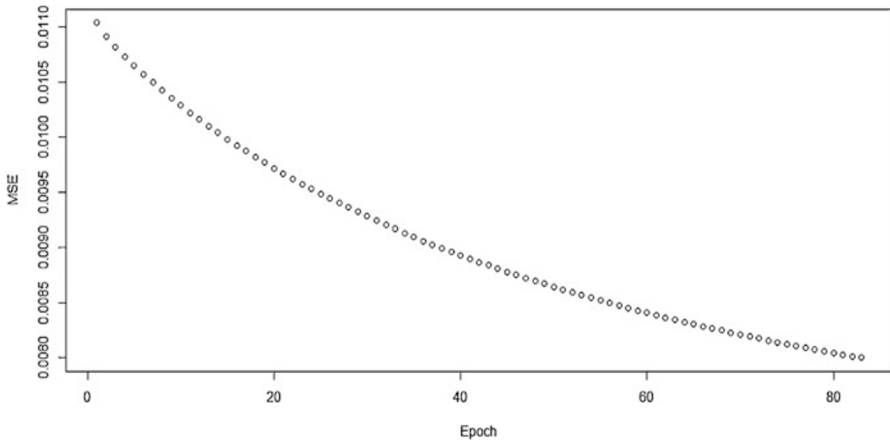



**Fig. 10.22** Behavior of the learning process by monitoring the MSE for Example 10.2—a hand computation

and Hastie 2016). One of the disadvantages of the basic backpropagation algorithm just described above is that the learning parameter $\eta$ is fixed.

## References

Alipanahi B, Delong A, Weirauch MT, Frey BJ (2015) Predicting the sequence specificities of DNA- and RNA-binding proteins by deep learning. Nat Biotechnol 33:831–838

Anderson J, Pellionisz A, Rosenfeld E (1990) Neurocomputing 2: directions for research. MIT, Cambridge

Angermueller C, Pärnamaa T, Parts L, Stegle O (2016) Deep learning for computational biology. Mol Syst Biol 12(878):1–16

Chollet F, Allaire JJ (2017) Deep learning with R. Manning Publications, Manning Early Access Program (MEA), 1st edn

Cole JH, Rudra PK, Poudel DT, Matthan WA, Caan CS, Tim D, Spector GM (2017) Predicting brain age with deep learning from raw imaging data results in a reliable and heritable biomarker. NeuroImage 163(1):115–124. https://doi.org/10.1016/j.neuroimage.2017.07.059

Cybenko G (1989) Approximations by superpositions of sigmoidal functions. Math Control Signal Syst 2:303–314

Dingli A, Fournier KS (2017) Financial time series forecasting—a deep learning approach. Int J Mach Learn Comput 7(5):118–122

Dougherty G (2013) Pattern recognition and classification-an introduction. Springer Science + Business Media, New York

Efron B, Hastie T (2016) Computer age statistical inference. Algorithms, evidence, and data science. Cambridge University Press, New York

Francisco-Caicedo EF, López-Sotelo JA (2009) Una approximación práctica a las redes neuronales artificiales. Universidad del Valle, Cali

Goldberg Y (2016) A primer on neural network models for natural language processing. J Artif Intell Res 57(345):420

Haykin S (2009) Neural networks and learning machines, 3rd edn. Pearson Prentice Hall, New York

Hornik K (1991) Approximation capabilities of multilayer feedforward networks. Neural Netw 4: 251–257

James G, Witten D, Hastie T, Tibshirani R (2013) An introduction to statistical learning with applications in R. Springer, New York

Kohonen T (2000) Self-organizing maps. Springer, Berlin

Lantz B (2015) Machine learning with R, 2nd edn. Packt Publishing Ltd, Birmingham

LeCun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436–444

Lewis ND (2016) Deep learning made easy with R. A gentle introduction for data science. CreateSpace Independent Publishing Platform

Liu S, Tang J, Zhang Z, Gaudiot JL (2017) CAAD: computer architecture for autonomous driving. ariv preprint ariv:1702.01894

Ma W, Qiu Z, Song J, Cheng Q, Ma C (2017) DeepGS: predicting phenotypes from genotypes using Deep Learning. bioRxiv 241414. https://doi.org/10.1101/241414

Makridakis S, Spiliotis E, Assimakopoulos V (2018) Statistical and Machine Learning forecasting methods: concerns and ways forward. PLoS One 13(3):e0194889. https://doi.org/10.1371/journal.pone.0194889

McCulloch WS, Pitts W (1943) A logical calculus of the ideas immanent in nervous activity. Bull Math Biophys 5:115–133

McDowell R, Grant D (2016) Genomic selection with deep neural networks. Graduate Theses and Dissertations, p 15973. https://lib.dr.iastate.edu/etd/15973

Menden MP, Iorio F, Garnett M, McDermott U, Benes CH et al (2013) Machine learning prediction of cancer cell sensitivity to drugs based on genomic and chemical properties. PLoS One 8: e61318

Montesinos-López A, Montesinos-López OA, Gianola D, Crossa J, Hernández-Suárez CM (2018a) Multi-environment genomic prediction of plant traits using deep learners with a dense architecture. G3: Genes, Genomes, Genetics 8(12):3813–3828. https://doi.org/10.1534/g3.118.200740

Montesinos-López OA, Montesinos-López A, Crossa J, Gianola D, Hernández-Suárez CM et al (2018b) Multi-trait, multi-environment deep learning modeling for genomic-enabled prediction of plant traits. G3: Genes, Genomes, Genetics 8(12):3829–3840. https://doi.org/10.1534/g3.118.200728

Montesinos-López OA, Vallejo M, Crossa J, Gianola D, Hernández-Suárez CM, Montesinos-López A, Juliana P, Singh R (2019a) A benchmarking between deep learning, support vector machine and bayesian threshold best linear unbiased prediction for predicting ordinal traits in plant breeding. G3: Genes, Genomes, Genetics 9(2):601–618

Montesinos-López OA, Martín-Vallejo J, Crossa J, Gianola D, Hernández-Suárez CM, Montesinos-López A, Juliana P, Singh R (2019b) New deep learning genomic prediction model for multi-traits with mixed binary, ordinal, and continuous phenotypes. G3: Genes, Genomes, Genetics 9(5):1545–1556

Montesinos-López OA, Montesinos-López A, Pérez-Rodríguez P, Barrón-López JA, Martini JWR, Fajardo-Flores SB, Gaytan-Lugo LS, Santana-Mancilla PC, Crossa J (2021) A review of deep learning applications for genomic selection. BMC Genomics 22:19

Patterson J, Gibson A (2017) Deep learning: a practitioner's approach. O'Reilly Media

Ripley B (1993) Statistical aspects of neural networks. In: Bornndorff-Nielsen U, Jensen J, Kendal W (eds) Networks and chaos—statistical and probabilistic aspects. Chapman and Hall, London, pp 40–123

Rouet-Leduc B, Hulbert C, Lubbers N, Barros K, Humphreys CJ et al (2017) Machine learning predicts laboratory earthquakes. Geophys Res Lett 44(28):9276–9282

Rumelhart DE, Hinton GE, Williams RJ (1986) Learning internal representations by backpropagating errors. Nature 323:533–536

Shalev-Shwartz, Ben-David (2014) Understanding machine learning: from theory to algorithms. Cambridge University Press, New York

Srivastava N, Hinton G, Krizhevsky A, Sutskever I, Salakhutdinov R (2014) Dropout: a simple way to prevent neural networks from overfitting. J Mach Learn Res 15(6):1929–1958

Tavanaei A, Anandanadarajah N, Maida AS, Loganantharaj R (2017) A deep learning model for predicting tumor suppressor genes and oncogenes from PDB structure. bioRiv 177378. https://doi.org/10.1101/177378

Weron R (2014) Electricity price forecasting: a review of the state-of-the-art with a look into the future. Int J Forecast 30(4):1030–1081

Wiley JF (2016) R deep learning essentials: build automatic classification and prediction models using unsupervised learning. Packt Publishing, Birmingham, Mumbai

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 11
# Artificial Neural Networks and Deep Learning for Genomic Prediction of Continuous Outcomes

## 11.1 Hyperparameters to Be Tuned in ANN and DL

Successful applications of ANN or DL are related to the user's ability to choose the right hyperparameters. Hyperparameter selection is done with the goal that a model neither underfits nor overfits the training data sets. However, this task is challenging because the number of hyperparameters required in ANN and DL is large. Below we provide a list of the hyperparameters that need to be tuned for implementing ANN and DL models.

1. Network topology
2. Activation functions
3. Loss function
4. Number of hidden layers
5. Number of neurons in each layer
6. Regularization type
7. Learning rate
8. Number of epochs and number of batches
9. Normalization scheme for input data

Next, we provide some general tips for choosing each hyperparameter.

### 11.1.1 Network Topology

Choosing the neural network topology is a critical step for the successful implementation of ANN and DL models. A safe approach is to choose a feedforward network with a large amount of neurons since there is a lot of empirical evidence that this neural network performs a good optimization process and finds proper weights while at the same time avoiding overfitting. Even though there is empirical evidence that

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_11

other topologies also work well in some circumstances, the application of DL is still at an early stage (experimental stage) with no definite conclusions. Picking the right network topology is more an art than a science and only practice will make a proper neural network architect (Chollet and Allaire 2017). However, with images data the most powerful DNN are convolutional neural networks, see Chap. 13.

### *11.1.2 Activation Functions*

The principal rule for choosing activation functions for output layers depends on the type of response variable at hand. For this reason, for binary data, the best option is the sigmoid activation function, for ordinal or multiclass classification problems, the softmax activation function, and for continuous data, the linear activation function is suggested. However, even though there are no clear rules for choosing the activation functions for hidden layers, there is a lot of evidence that using sigmoid activation functions is not a good choice for hidden layers since sigmoid functions very often discard information due to saturation in both forward and backpropagation. RELU activation functions are the most used for hidden neurons, as can be seen in modern deep learning networks today, but RELU activation functions have the problem that some neurons may never activate across the entire training data set. For this reason, the suggestion is to use the leaky RELU activation function in hidden layers which have no zero gradient for all input values (Patterson and Gibson 2017).

### *11.1.3 Loss Function*

Choosing the right loss function is a key component for the successful implementation of ANN and DL models. Next are given some general rules for selecting the right loss function: (a) when the response variable is continuous, use the sum of squares error loss as the primary option and the sum of absolute percentage error loss as the secondary option; (b) when the response variable is binary, use the hinge loss or logistic (cross-entropy) loss; (c) however, if you are interested in the probability of success of each individual, the logistic loss is the best option; and (d) for categorical responses, the categorical cross-entropy loss is the most used, but when the response variable is a count, a reasonable choice is the Poisson loss or negative binomial loss.

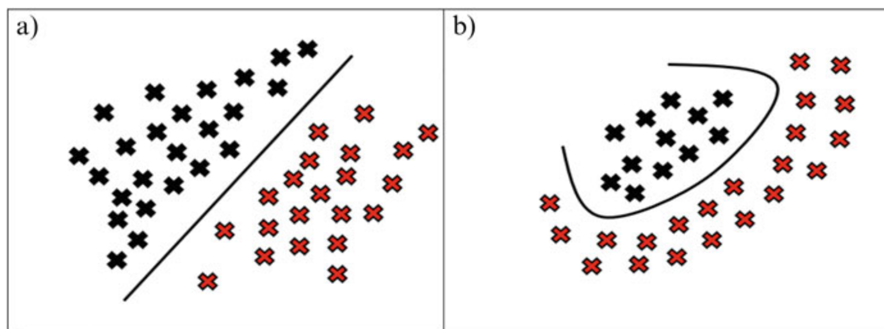



**Fig. 11.1** Schematic representation of linearly and nonlinearly separable patterns. (**a**) Linearly separable patterns. (**b**) Nonlinearly separable patterns

**Table 11.1** Guide for determining the number of hidden layers

| Number of hidden layers | Result |
|---|---|
| None | Only capable of representing linear separable functions or decisions |
| 1 | Can approximate well any functions that contain continuous mapping from one finite space to another |
| 2 | Represent a decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any degree of accuracy |

### *11.1.4 Number of Hidden Layers*

Selecting the number of hidden layers is a very important issue in the configuration process of DL models. According to the universal approximation theorem (Cybenko 1989), a network with one hidden layer and a large number of neurons is enough to approximate any arbitrary function to the desired degree of accuracy. However, the reality is that many times the required number of neurons is so large that its implementation is not practical. Also, there is a lot of evidence that more than one hidden layer is required when the data are not separated linearly (Fig. 11.1b). Therefore, if your data are linearly separable (Fig. 11.1a) (which you often unknow by the time you begin coding an ANN or DL model), then you don't need any hidden layers at all. Of course, you don't need an ANN or DL model to resolve your data either, but it will still do the job. One hidden layer is sufficient for most problems.

In general, neural networks with two hidden layers can represent functions with any kind of shape. There are currently no strong theoretical reasons to use neural networks with any more than two hidden layers, but empirical evidence shows that using more than two hidden layers can capture in a better way non-linear patterns and complex interactions (Chollet and Allaire 2017). Further, for many practical problems, there's no reason to use any more than one hidden layer. Problems that require two hidden layers are not very common. Differences between the number of hidden layers are summarized in Table 11.1.

### 11.1.5 Number of Neurons in Each Layer

The number of neurons in the network is very important since using too few neurons in the hidden layers will result in underfitting, while using too many neurons in the hidden layers can result in severe overfitting. For the input layer, the number of neurons is predetermined by the number of input features (predictors) of the input data. Also, in the output layer, the number of neurons is determined by the number of response variables (outputs) to be predicted or by the number of classes of the response variable when it is ordinal or categorical. For example, if we want to predict grain yield, a continuous outcome, using information of 3000 markers as input variables, this means that the input layer should have 3000 neurons and the output layer just 1, since we are only interested in one response variable that is continuous. However, if instead of predicting a continuous outcome we are interested in predicting an categorical output with ten categories, in this case the required number of neurons in the output layer should be equal to the number of categories in the categorical response variable. But defining the number of neurons in the hidden layers is challenging, and it is left to the user to decide which number to use prior to training the model. According to the literature review we performed, we found that there is no unique and reliable rule for determining the required number of neurons in the hidden layers, since this number depends on (a) the number of input neurons; (b) the amount of training data; (c) the quality of the training data; and (d) the complexity of the learning task (Lantz 2015).

As a general rule, the more complex the network topology (with more neurons, layers, etc.), the more powerful the network for learning more complex problems. That is, the more neurons and layers are used, the better the representation of the training data, which means that the predicted values should be very close to the true values observed in the training set; however, this runs the risk of overfitting and may generalize poorly to new data. We also need to be aware that increasing the complexity of the network also implies that more computational resources are required for its implementation (Lantz 2015). However, as pointed out before, there is empirical evidence that only one hidden layer with a large number of neurons is enough even for complex applications, since increasing the number of layers increases the computational resources required by the complexity of the neural network. For this reason, in general, increasing the number of neurons instead of the number of layers is suggested, for example, if we have a neural network with 100 neurons in the single hidden layer, computationally it is better to increase to 200 neurons in the same hidden layer than to add a second hidden layer with 100 neurons. Of course, this suggestion is not always valid since there are many applications in the fields of pattern recognition and image classification, among others, were to be able to create adequately complex decision boundaries. It is very important to add more hidden layers to better capture the complexity of the phenomenon we want to predict.

Given the difficulties mentioned above, there are two very general approaches for specifying the required number of neurons in the hidden layers. The first one, called

the *backward* approach, suggests starting with a very large number of neurons and evaluating their performance, and then decreasing the number of neurons until there is no more gain in the reduction of the testing error. The second one, called the *forward* approach, suggests starting with half (50%) of the input neurons and then increasing the number of neurons until no significant gain in the prediction performance of sample data is observed. Since there is no consensus about how to choose the required number of hidden neurons, next we provide some general rules: (a) the number of hidden neurons should be in the range between the size of the input layer and the size of the output layer; (b) the number of hidden neurons should be 2/3 of the input layer size, plus the size of the output layer; (c) the number of hidden neurons should be less than twice the input layer size; (d) Hecht-Nielsen (1987) suggested using $2P + 1$ hidden neurons (where $P$ is the number of inputs) for a network with one hidden layer to compute an arbitrary function; (e) Lippmann (1987) suggested that the maximum number of neurons required for the single hidden layer should be equal to $L \times (P + 1)$, where $L$ is the number of output neurons; and (f) for a single hidden layer classifier, an upper bound is about $2P + 1$ hidden neurons, whichever is larger.

But in practical terms, the optimal size of the hidden layer is usually between the size of the input and size of the output layers. In some problems, one could probably achieve a decent performance (even without a second optimization step) by setting the hidden layer configuration following just two rules:

1. the number of hidden layers is equal to one and
2. the number of neurons in that layer is the mean of the neurons in the input and output layers.

These rules are only starting points that you may want to consider. Ultimately, the selection of the neurons of your neural network will come down to trial and error following a backward or forward approach. Of course, the idea is to use as few neurons as possible that provide good performance in validation data sets. In general, rules (e) and (f) many times are conservative given that fewer neurons are often sufficient, since in a lot of cases, using only a small number of hidden neurons in the artificial neural network offers a tremendous amount of learning ability. Also, we need to recall that according to the universal approximation theorem, an artificial neural network with at least one hidden layer with a lot of neurons is a good universal function approximator (Lantz 2015).

### *11.1.6 Regularization Type*

The main goal of regularization is to avoid overfitting, and the basic idea is to penalize the weights in such a way that the model increases its prediction performance in out-of-sample data. The most common methods are the L2 (Ridge penalization), L1 (Lasso penalization), L2-L1 (Elastic Net penalization), and the dropout method. The L1 regularization forces many weights to be exactly zero, which

provides a sparse solution, while with L2 regularization, the larger the weight, the stronger the penalization. The L2-L1 is somewhere between the L2 and L1 regularization, while the dropout method fixes a certain percentage of weights to be exactly zero in the input or hidden neurons. However, it is not easy to suggest one of them, since all of them do a reasonable job in particular contexts, but any of them works well for all types of data. But, in general, the L2 and dropout regularizations perform well in many applications.

### *11.1.7 Learning Rate*

The learning rate modifies weights and thresholds (bias) of the neural network, in order for a given input to the network to produce a favored output. It also determines the speed for convergence of the backpropagation algorithm to an optimal solution. Typically, a value of 0.001 or less is used to initialize the learning rate; however, in general, it is a hard task to choose the appropriate learning rate. Before explaining the problems involved in choosing the learning rate, we will first mention that the backpropagation algorithm can stop because (1) the number of maximum epochs allowed was reached and (2) an optimal solution was found (Baysolow II 2017). Figure 11.2 illustrates how to reach an optimal solution with an optimal learning rate.

Next, we explain the consequences of choosing a learning rate that is too small or too large. When the learning rate is very small, it is more likely to stop the backpropagation algorithm in a non-optimal condition since the first stopping condition is satisfied. Although apparently this can be fixed by increasing the number of epochs, this solution is not optimal since increasing the number of epochs considerably increases the required computation resources. On the other hand, when the learning rate is very large, it is possible not to reach the optimal solution due to the fact that the loss function in each epoch may overcorrect and give updated values for the coefficient that are far too small or far too large (Figs. 11.3 and 11.4). For this reason, the suggestion is to try several learning rate values and see how the algorithm performs across each epoch (Baysolow II 2017).

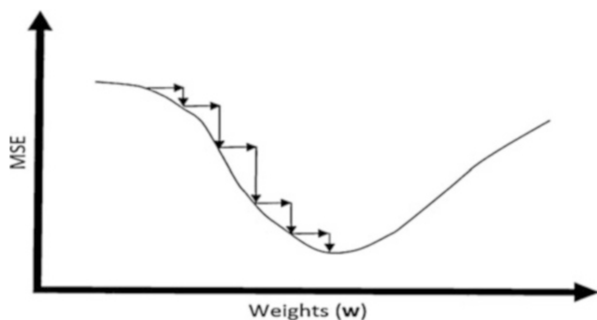



**Fig. 11.2** Optimal learning rate for efficient error minimization (Samarasinghe 2007)

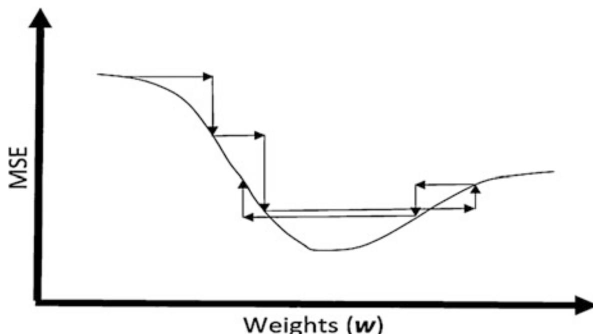



**Fig. 11.3** The effect of a high learning rate on the learning process (Samarasinghe 2007)

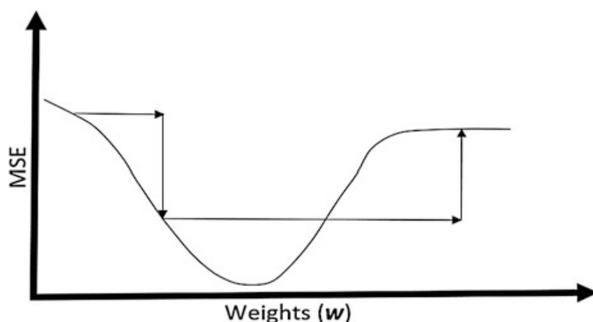



**Fig. 11.4** The effect on the learning process of a learning rate that is too high. (Samarasinghe 2007)

### *11.1.8 Number of Epochs and Number of Batches*

Before defining an epoch, we will define a sample (individual) as a single row of data that contains inputs that are fed into the algorithm and an output (prediction) that is compared to the true value to calculate an error. The number of epochs is a hyperparameter that defines the number of times that the learning algorithm will work across the entire training data set. One epoch means that each sample in the training data set has had an opportunity to update the internal model parameters. An epoch is comprised of one or more batches. For example, assume you have a data set with 600 samples (rows of data) and you choose a batch size of 15 and 500 epochs. This means that the data set will be divided into 40 batches, each with 15 samples. The model weights will be updated after each batch of 15 samples. This also means that one epoch will involve 40 batches or 40 updates of 15 samples to be completed. With 500 epochs, the model will be exposed to or will pass through the whole data set 500 times. That is, a total of $40 \times 500 = 20{,}000$ batches during the entire training process. Therefore, with this example, it is clear that the number of batches in each epoch is equal to the number of samples in the data sets divided by the batch size. To better understand these two hyperparameters, you can think of a for-loop over the number of epochs where each loop proceeds over batches of the training data set. Within this for-loop is another nested for-loop that iterates over each batch of samples, where one batch has the specified “batch size” number of samples.

The number of epochs is traditionally large, often hundreds or thousands, allowing the learning algorithm to run until the error from the model has been sufficiently minimized. You may see examples of the number of epochs in the literature and in tutorials set to 10, 100, 500, 1000, and larger. This means that the number of epochs can be set to an integer value between one and a very large value. You can run the algorithm for as long as you like and even stop using other criteria besides a fixed number of epochs, such as a change (or lack of change) in the model error over time. The more complex the features and relationships in your data, the more epochs you'll require for your model to learn, adjust the weights, and minimize the loss function. With regard to the batch size, that is, the number of samples processed before the model is updated, this hyperparameter should be more than or equal to one and less than or equal to the number of samples in the training data set. Popular batch sizes include 32, 64,128 samples and so on.

It is common to create line plots that show epochs along the $x$-axis as time, and the error or skill of the model on the $y$-axis. These plots were mentioned in Chap. 10 and were called learning curves under the *early stopping rule*. These plots can help to diagnose whether the model has over learned, under learned, or is suitably fit to the training data set. Since there are no magic rules on how to configure these parameters, you must try different values and see what works best for your problem.

### *11.1.9 Normalization Scheme for Input Data*

It is generally good practice to scale input variables to have the same scale. When input variables have different scales, the scale of the weights of the network will, in turn, vary accordingly. This introduces a problem when using weight regularization because the absolute or squared values of the weights must be added for use in the penalty. Also, without normalization, the input variable with greater numeric range has larger impact than the one having less numeric range, and this could, in turn, impact prediction accuracy. This problem can be addressed by the following methods applied to each input variable:

Standardizing. Subtract the mean and divide by the standard deviation. Many times it is good practice to only divide each input by the standard deviation without centering.

Min-max normalization. Subtract from each observation the min value of the input and divide this by the range (max value-min value) of the input. The disadvantage of the min-max normalization technique is that it tends to bring the data toward the mean. Details of this method and other methods of scaling were provided in Chap. 2 of this book.

## 11.2 Popular DL Frameworks

Given that the adoption of DL has proceeded at an alarming pace, the maturity of the ecosystem has also shown phenomenal improvement. Thanks to many large tech organizations and open-source initiatives, we now have a plethora of options to choose from. Due to the evolution of the software industry, today it is far easier to develop high-end software than it was a few years back, thanks to the available tools that have automated complex problems in a way that's simple to use.

Given the maturity of software tools available today, we can afford to automate several complexities that happen in the background. These tools are nothing but building blocks for software systems. You technically don't need to start from scratch; you can instead rely on available powerful tools that have matured significantly to take care of several software-building services. Given the level of abstraction a framework provides, we can classify it as a:

(a) Low-level DL framework and
(b) High-level DL framework

A few of the popular low-level frameworks for DL are Theano, Torch, PyTorch, MxNet, TensorFlow, etc. The previously mentioned frameworks can be defined as the first level of abstraction for DL models. You would still need to write fairly long codes and scripts to get your DL model ready, although much less than using just Python or C++. The advantage of using the first-level abstraction is the flexibility it provides in designing a model. However, to simplify the process of DL models, we have frameworks that work on the second level of abstraction called High-Level DL Frameworks. Rather than using the previously mentioned frameworks directly, we can use a new framework on top of an existing framework and thereby simplifying DL model development even further. The most popular high-level DL framework that provides a second-level abstraction to DL model development is Keras. Other frameworks like Gluon, Lasagne, and so on are also available, but Keras has been the most widely adopted one.

Keras is a high-level neural network API written in Python, and it can help you in developing a fully functional DL model with few lines of code. Since it is written in Python, it has a larger community of users and supporters and is extremely easy to get started on. The simplicity of Keras is that it helps users to quickly develop DL models and provides a ton of flexibility while still being a high-level API. This really makes Keras a special framework to work with. By far the most widely adopted usage of Keras is with TensorFlow as a back end (i.e., Keras as a high-level DL API and TensorFlow as its low-level API back end). In a nutshell, the code you write in Keras gets converted to TensorFlow, which then runs on a computing instance. You can read more about Keras and its recent developments here: https://keras.io/.

For this reason, in this book, we use Keras for implementing ANN and DL models (Allaire and Chollet 2019). The examples that follow use this framework.

## 11.3 Optimizers

The optimizer is a key component of the model training process. Up to this point, we are only familiar with the backpropagation algorithm that provides feedback to the model to reach a good trained process. The loss function of ANN is the main element to understand how well or poorly the current set of weights has performed on the training sample. The next step for the model is to reduce the loss. How does it know what steps or updates it should perform on the weights to reduce the loss? As we observed in the previous chapter, the optimizer function is a mathematical algorithm that uses calculus (derivatives and partial derivatives) to estimate how much change the network will need in the loss function to be able to reach the minimum. The change in the loss function, which would be a decrease or increase, helps in finding the direction of the change required in the weight of the connection. The math equations required are specific for each optimizer and are beyond the scope of this book (Géron 2019), but the general issues are those provided in Chap. 10 for the back propagation algorithm.

*Batch Gradient Descent (BGD).* The implementation of BGD requires training the model with the full data sets at every step. As a result, it is considerably slow on very large training sets. The BGD will eventually converge to the optimal solution for convex functions with a fixed learning rate, but requires a considerable amount of time (Géron 2019). Under BGD, the cost function has the shape of a bowl when the inputs have the same scale, but an elongated bowl with different scales of the features is also possible. Figure 11.5 shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).

Other optimizers based on gradient descent available in Keras are listed next.

*Stochastic Gradient Descent (SGD).* SGD, at the extreme, simply chooses a random instance in the training set at each step and calculates the gradients only in that single instance, that is, SGD uses batch_ size = 1 in the model training function.

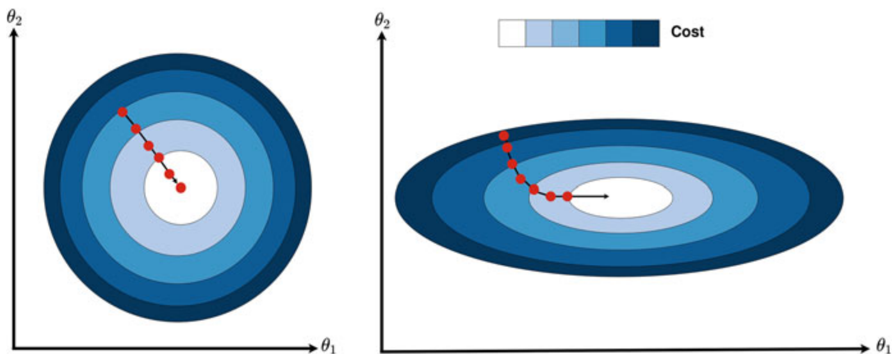



**Fig. 11.5** Gradient descent with and without input (feature) scaling (Géron 2019). On the left are shown the two features on the same scale that reach the target faster, while on the right the two features have different scales and take longer to reach the target

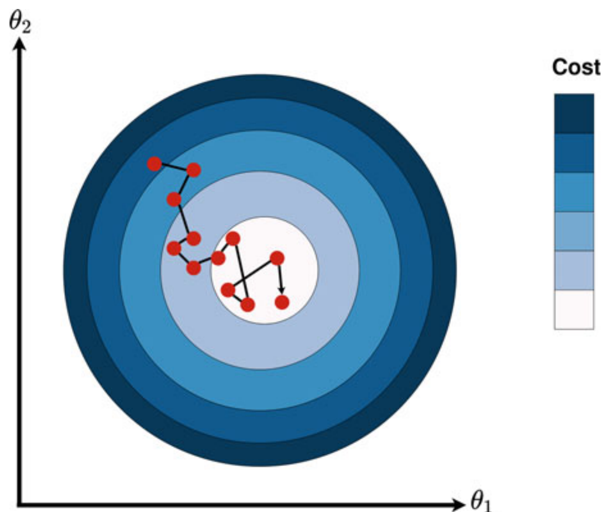



**Fig. 11.6** Stochastic gradient descent (Géron 2019)

Obviously, this makes the algorithm much faster, since in each iteration, it has less data to manipulate (Fig. 11.6). To reduce high fluctuations in SGD optimizations, a better approach would reduce the number of affected iterations in a mini-batch. Mini-batch gradient descent is a modification of the gradient descent method that splits the training data set into small batches that are used to calculate model prediction error and update model coefficients. This approach has been more successful and results in a more fluid training process. The batch size is usually set to powers of 2 (i.e., 32, 64, 128, etc.) (Géron 2019).

*Adaptive Moment Estimation (Adam).* This is by far the most popular and widely used optimizer in DL. In most cases, you can blindly choose the Adam optimizer and forget about the optimization alternatives. This optimization technique computes an adaptive learning rate for each parameter. It defines momentum and variance of the gradient of the loss and leverages a combined effect to update the weight parameters. The variance and momentum together help smooth the learning curve and effectively improve the learning process.

*Other important optimizers*. There are many other popular optimizers for DL models. We will not discuss all of them in this book. We will list only a few of the most popular optimization alternatives used and available within Keras (Adadelta, RMSProp, Adagrad, Adamax, Nadam). There is no perfect optimization method since each of them has its own pros and cons. Each optimizer requires tunable parameters like learning rate, momentum, and decay. The saddle point and vanishing gradient are two problems often faced in DL. These problems can be explored in more detail while choosing the best optimizer for your problem. But Adam always works fine in most cases (Géron 2019).

## 11.4 Illustrative Examples

**Example 11.1 MaizeToy**
In this example, we use the MaizeToy data set. This data set is preloaded in the BMTME library, and for this reason it is important to first install this library and also the Keras library. To see the information of this data set, we can use:

```
rm(list=ls())
library(keras)
library(BMTME)
##########Set seed for reproducible results###################
use_session_with_seed(45)
#################Loading the MaizeToy data sets#############
data("MaizeToy")
ls()
head(phenoMaizeToy)
```

Using this code, we can see in the console that this data set has two objects: "genoMaizeToy" and "phenoMaizeToy." The first contains the genomic relationship matrix (GRM) of the lines, while the second contains the phenotypic information for which the first six observations resulting from using the head() command are shown:

```
> head(phenoMaizeToy)
  Line Env   Yield  ASI  PH
1 CKDHL0008  EBU   6.88 2.7 226
2 CKDHL0008  KAK   4.83 1.7 227
3 CKDHL0008  KTI   4.84 3.0 240
4 CKDHL0039  EBU   6.85 1.3 239
5 CKDHL0039  KAK   5.23 0.7 212
6 CKDHL0039  KTI   6.18 2.0 238
```

Here we can observe that in this data set, the first column contains the names of the lines, the second contains the name of the environments, while columns 3–5 contain the measured traits: grain yield (Yield), anthesis-silking interval (ASI), and plant height (PH).

Next we order the data, first in terms of environments and then in terms of lines inside each environment, using the following commands:

```
##################Ordering the data ###############################
phenoMaizeToy<-(phenoMaizeToy[order(phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames(phenoMaizeToy)=1:nrow(phenoMaizeToy)
head(phenoMaizeToy,5)
```

The output of five observations using the head() function is equal to:

```
> head(phenoMaizeToy,5)
  Line Env   Yield ASI  PH
1 CKDHL0008 EBU   6.88 2.7 226
2 CKDHL0039 EBU   6.85 1.3 239
3 CKDHL0042 EBU   6.37 2.3 238
4 CKDHL0050 EBU   4.98 3.1 239
5 CKDHL0060 EBU   7.07 1.4 242
```

Here it is clear that the data are ordered first in terms of environments and then in terms of lines inside each environment. Next, we build the design matrices required to implement a model with a predictor with information of environments, lines, and genotype by environment interaction. To do this, the following code can be used:

```
#################Design
matrices######################################
LG <- cholesky(genoMaizeToy)
ZG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line))
Z.G <- ZG %*% LG
Z.E <- model.matrix(~0 + as.factor(phenoMaizeToy$Env))
ZEG<-model.matrix(~0 +as.factor(phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker(diag(length(unique(phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky(G2)
Z.EG <- ZEG %*% LG2
```

It is important to point out that the design matrices are built using the model matrix command of R that is specialized for this task, but the resulting design matrices of lines and genotype by environment interaction are post-multiplied by the Cholesky decomposition of the GRM of lines or genotype × environment. In the next code, we select the available response variables and create the training–testing set using a CV strategy with five-fold.

```
#############Selecting the response variable#####################
Y <- as.matrix(phenoMaizeToy[, -c(1, 2)])
####Training-testing sets using the BMTME package##############
pheno <- data.frame(GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
         Response = phenoMaizeToy[, 3])
CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)
```

The Y matrix contains information of the three traits available in the MaizeToy data sets. The pheno variable is an auxiliary object that contains information of lines, information of environments, and the response variable of only one trait to be able to create a training-testing partition according to a five-fold cross-validation strategy where information of some lines is missing in some environments but not in all. The

CrossV$CrossValidation_list argument was used to print the testing observations for each fold.

```
> CrossV$CrossValidation_list
$partition1
 [1] 88 79 59 67  3 36 90 14 23 78 10 27 26 24 74  9 13 72

$partition2
 [1] 12 39 30 69 41 65 83 63 87 57 48 53  2 38 20 31 43 29

$partition3
 [1] 89 80 75 17 11 54 33 52 28 84 56  7 35 37 42 21 34 70

$partition4
 [1] 81 58 44  8 73 19  4 66 51  6 86 60 40 76 61 16 15 64

$partition5
 [1] 45 82 18  5 71 47 46 49 85 50 25 55 62 22 32 68  1 77
```

We used the following code to select only one trait to be our response variable (Yield), and we also collected in an X matrix the information of environments, lines, and genotype by environment that will be the input of our artificial deep neural network model:

```
################Part 0: Final X and y before splitting ################
y=(phenoMaizeToy[, 3])
length(y)
X=cbind(Z.E,Z.G,Z.EG)
dim(X)
```

The output of this code shows that the total number of observations is 90, that is, 30 for each environment, and the number of input variables (predictors) is 123, which corresponds to the number of columns in matrix X. For the moment, we will only work with one-fold of the five training–testing sets, and we will also specify the number of epochs and units (neurons) used to implement our first artificial deep neural network model in Keras.

```
####################One holdout cross-validation#################
tst_set=CrossV$CrossValidation_list[[2]]
No_Epoch=1000
N_Units=33
X_trn=X[-tst_set,]
X_tst=X[tst_set,]
y_trn=y[-tst_set]
y_tst=y[tst_set]
```

Before specifying the model, we will explain the pipe operator (%>%) that connects functions or operations together in the Keras library. We will be using

the pipe operator (%>%) to add layers to the network. The pipe operator allows us to pass the value on its left as the first argument to the function on its right.

Next, we define the *artificial neural network model* and *compile* it in Keras:

```
build_model <- function() {
model <- keras_model_sequential()
model %>%
 layer_dense(units =N_Units, activation = "relu", input_shape = c(dim
(X_trn)[2])) %>%
 layer_dropout(rate = 0.0) %>%
 layer_dense(units = 1, activation = "linear")

model %>% compile(
 loss = "mse",
 optimizer = "rmsprop",
 metrics = c("mse"))
model}

model <- build_model()
model %>% summary()
```

The first part of this code specifies the type and topology of the model. With keras_model_sequential we are using Keras to implement a feedforward (or densely connected) network. The sequential model is composed of a linear stack of layers. Layers consist of the input layer, hidden layers, and an output layer. Sequential models are appropriate when the information available is in two dimensions: one dimension for samples (observations or individuals) and the other for predictors (features), since the available data set is a matrix where the rows correspond to observations and the columns correspond to the features at hand. The first layer_dense is used to specify the first hidden layer and also the required number of neurons (units), and in the input_shape is specified the dimension of the input variables (in this case, it is equal to the number of columns in the training set of the predictors, that is, 123). Also specified here is the linear activation function which gives as output positive values. Next in layer_dropout() we are instructing Keras that any of the weights of the neurons of the first hidden layer will be set to zero, that is, all specified neurons will be used since rate $= 0$. Then the second layer_dense() is used to specify the output layer, which only used one neuron since the response variable under study is continuous and unique (univariate); in this layer we used the linear activation function, which is a good choice for continuous outputs. An input shape argument is not required in the second or other hidden layers since its input shape is automatically inferred as being the output of the previous layer.

Next, the compilation step is done in which the learning process of the network is configured and where you should specify the optimizer, the loss function(s), and the metric used for monitoring the match between observed and predicted values. In this case, for the optimizer we specified "rmsprop," which is a gradient-descent-based algorithm that combines Adagrad and Adadelta adaptive learning ability. Adagrad is a gradient-descent-based algorithm that accumulates previous costs to do adaptive

learning. Adadelta is a gradient-descent-based algorithm that uses Hessian approximation to do adaptive learning. The loss function and metric used in this case is the mean squared error (mse) since the response variable is quantitative. The last two lines of this code provide a summary of the model that will be implemented.

```
> model <- build_model()
> model %>% summary()
___________________________________________________________________________
__________________________
Layer (type)                  Output Shape                  Param #
===========================================================================
=================================
dense_1 (Dense)                 (None, 33)                    4092
___________________________________________________________________________
__________________________
dropout_1 (Dropout)               (None, 33)                    0
___________________________________________________________________________
__________________________
dense_2 (Dense)                 (None, 1)                    34
===========================================================================
=================================
Total params: 4,126
Trainable params: 4,126
Non-trainable params: 0
```

From this summary output of the neural model that will be implemented, we can see that there are 33 neurons (since we decided to use 33 neurons) in the hidden layer, and any of their weights will be fixed at zero since the dropout percentage is zero. For this reason, the number of parameters (weights) in the first hidden layer is equal to $123 \times 33 + 33 = 4092$, and to 34 in the second layer since there are 34 weights that connect from the hidden layer to the output layer; this corresponds to 33 units plus an intercept (bias) term. For this reason, the total number of parameters to estimate is equal to 4126.

We then provide the code for training (fitting) the model and also specify a plot to visualize training and validation metrics by epoch:

```
print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat("\n")
    cat(".")
  })
##########Internally this part of the code split the training set in
training-inner and validation###
model_fit <- model %>% fit(
  X_trn, y_trn,
  shuffle=F,
  epochs =No_Epoch, batch_size =72,
  validation_split =0.2,
  verbose = 0,
  callbacks = list(print_dot_callback))
###########Plot history ##################
plot(model_fit)
```

(continued)

(continued)

The first part of the code is a custom callback function for replacing the default training output with a single dot per epoch, while the second part is for fitting the artificial neural model and storing training stats. Also, in this second part of the code is specified the input and output training information, the number of epochs and batch size, and the size of the inner validation set, which in this case was 0.2 or 20%. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics on this data at the end of each epoch. The validation data are selected from the last samples in the X and y data provided, before shuffling (permining the data). The goal is to shuffle the training data before each epoch when TRUE is specified, but not when FALSE is specified. The plot of model_fit is for visualizing the model's training progress using the metrics stored in the model_fit variable. We want to use these data to determine how long to train before the model stops making progress.

In Fig. 11.7, the mean square error is plotted in the bottom panel, and the loss in the top panel. In this plot you can see that after 250 epochs, there is no relevant improvement in the prediction performance of the validation set. Next, we calculate the MSE for the training set; the predictions and MSE of the testing set were obtained using the following code:

```
############MSE for No_epochs=1000 in the outer testing set##########
model_fit$metrics$val_mse[No_Epoch]
pred=model%>% predict(X_tst)
Pred=c(pred)
Obs=y_tst
MSE_First=mean((Obs-Pred)^2)
MSE_First
```

The first part of the code extracts the MSE for the training set implemented with 1000 epochs, which is equal to 4.5429; the second manually calculates the MSE for the testing set, which is equal to 1.8688. Next, we refit the model but using the *early*

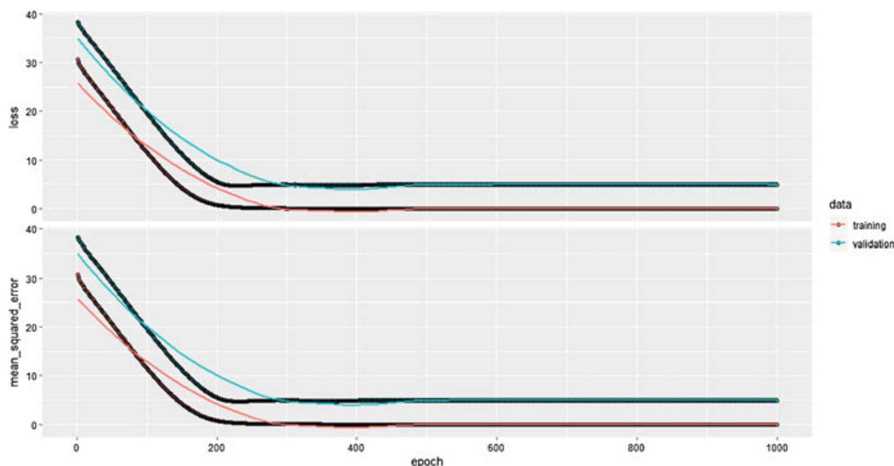



**Fig. 11.7** Training and validation metrics

*stopping approach* to stop the training process when there is no more improvement in terms of prediction accuracy.

```
#######Fitting the model with early stopping#######
######## The patience parameter is the amount of epochs to check for
improvement.
early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min', patience =50)

model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
 X_trn, y_trn,
 shuffle=F,
 epochs =No_Epoch, batch_size =72,
 validation_split = 0.2,
 verbose=0,  callbacks = list(early_stop,print_dot_callback))

########################Plot history ############################
length(model_fit_Final$metrics$mean_squared_error)
plot(model_fit_Final)
```

The first part of this code is a function that will help to implement the early stopping process in Keras, and the function to monitor is the validation loss score. The patience parameter is the number of epochs with no improvement, after which training will be stopped. The second part is exactly the same as the one used before for fitting the model, with the difference that now the training process will stop according to the information provided in the early stopping function. The code length(model_fit_Final$metrics$mse) prints the number of epochs during the stopping process. Finally, the history of the training and validation process is depicted in this plot (Fig. 11.8).

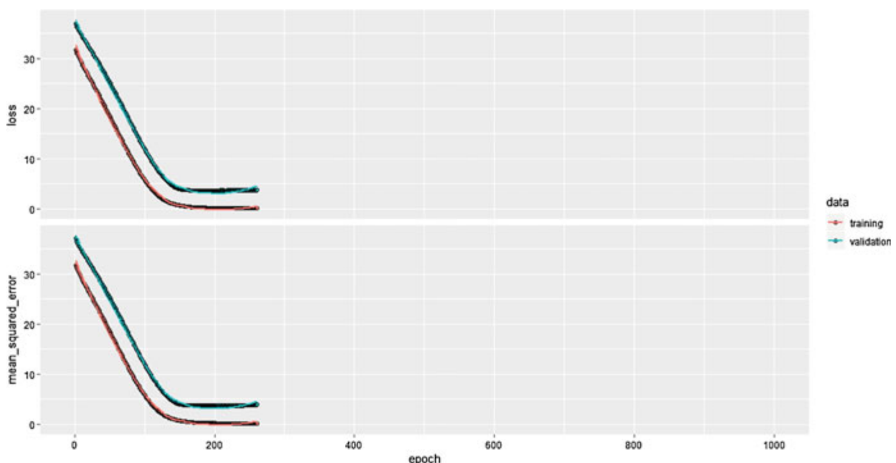



**Fig. 11.8** Training and validation metrics under the stopping approach

In the plot it is clear that the stop occurred in epoch 282, which means that the training process was done with this number of epochs and not with 1000 as before. Finally, the following code was used to obtain the predicted values and MSE for the testing set using the training information with 282 epochs. Two MSE were also printed, one using the original 1000 epochs (MSE_First) and the last one with 282 epochs.

```
###############Prediction of testing set#########################
predicted=model_Final %>% predict(X_tst)
Predicted=c(predicted)
Observed=y_tst
plot(Observed,Predicted)
MSE=mean((Observed-predicted)^2)
MSE
Obs_Pred=cbind(Observed,predicted)
colnames(Obs_Pred)=c("Observed","Predicted")
Obs_Pred

########Results with and without early stopping MSE and MSE_First##
MSE_First
MSE
```

Finally, this code, in addition to providing the observed and predicted values, of which we show the first six observations, also provides the MSE of both training processes:

```
> head(Obs_Pred)
      Observed Predicted
[1,]  7.55     5.824002
[2,]  5.91     4.806885
[3,]  5.62     4.344528
[4,]  5.39     4.817317
[5,]  5.23     4.776768
[6,]  6.39     5.039447
>
>
> ########Results with and without early stopping are printed as MSE and
MSE_Firt, respectively ##
> MSE_First
[1] 1.868842
> MSE
[1] 1.806609
```

From this output it is clear that when the training process was done with the *early stopping approach*, the MSE of the testing set was better by 3.4% (even though maybe it is not so relevant since the predictions are still quite far from the true values), than when using all the epochs originally used. However, it is important to point out that the early stopping method did not always improve the prediction performance.

**Example 11.2 MaizeToy Example with a Tuning Process**

The last example illustrated how to use the early stopping rule to train ANN or DL models. However, that example only used one partition of the data sets (one TRN with 80% and one TST with 20%) and one value of neurons. However, if another TRN–TST split and number of neurons are used, the expected MSE will be different. For this reason, next we split the training into five TRN–TST inner subsets and also train the ANN model with a grid of two values for the neurons and two dropout values.

Tuning a DNN model often requires exploring many hyperparameters to find the best combination that produces the best out-of-sample predictions. The best way to approach this in Keras is by defining external flags (a map of the model parameter name and an arrangement of values to try) for key parameters, which you may want to vary instead of progressively changing your source code of the training script each time. In other words, this is a way in Keras for specifying a grid search. The flags() function provides a flexible mechanism for defining flags and varying them across training runs. For this reason, for tuning hyperparameters in Keras, we will now use the tfruns package (Allaire 2018), which uses flags and provides a lot of flexibility for tracking, visualizing, and managing training runs and experiments in R. This package allows tracking the metrics, hyperparameters, source code of every training run, and output, and visualizes the results of individual runs and comparisons between runs. For this reason, first we created the Keras script with flags called Code_Tuning_With_Flags_00.R, as follows:

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.01),
 flag_numeric("val_split",0.2),
 flag_numeric("reg_l1",0.001))

####b) Defining the DNN model
build_model<-function() {
model <- keras_model_sequential()
model %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1,
input_shape = c(dim(X_trII)[2])) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units=1, activation ="linear")

#####c) Compiling the DNN model
model %>% compile(
 loss = "mse",
 optimizer =optimizer_adam(FLAGS$learning_rate),
 metrics = c("mse"))
model}
```

```
model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")})

early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =50)

###########d) Fitting the DNN model################
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
 X_trII, y_trII,
 epochs =FLAGS$Epoch1, batch_size =FLAGS$batchsize1,
 shuffled=F,
 validation_split = FLAGS$val_split,
 verbose=0,callbacks = list(early_stop,print_dot_callback))
```

In part a) of the above code, we declare the flags for each of the candidate hyperparameters to be tuned. It is important to point out that the type of flag that is used depends on the type of hyperparameter, and there are four types of flags: a) flag_integer() for integer values like 1, 2, 3, 4, etc., b) flag_numeric() for floating point values like 2.5, 4.4. −1.23, 1.35, etc., c) flag_boolean() for logical values (e.g., true, false, 1, 0), and d) flag_string(), for example, character values (e.g., "RELU").

Next we incorporate the flag parameters within our model. In part b) of the code, a feedforward neural network is defined, but incorporating the flags for units and hidden activation function allows us to vary the values of these hyperparameters and not only work with the literal values of these hyperparameters. The definition of the DNN model differs from the previous DNN model in the use of flags. In part c), the compilation of the DNN model is done using flags and in this part the only flag used was for the learning rate. Finally, in part d), the model is fitted; flags were also used here but now for epochs, batch_size, and validation_split.

Next we created a grid of hyperparameters and we called the last R code (Code_Tuning_With_Flags_00.R) to be able to vary each flag with a grid of values using the tfruns::tuning_run() function where the grid of values for each of the hyperparameters is specified. The following code performs a tuning process for choosing the optimal hyperparameters. The data used here are the same as before (MaizeToy data sets), but it is important to point out that to be able to use this code, first all the codes that were used in Example 11.1 should be used, until part 0: Part 0: Final X and y before splitting.

```
################a) Inner cross-validation########################
nCVI=5 ####Number of folds for inner CV
Hyperpar=data.frame()
for (i in 1:nCVI){
 Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
```

```
 X_trII=X_trn[Sam_per,]
 y_trII=y_trn[Sam_per]

############b) Grid search using the tuning_run() function of tfruns
package########
runs.sp<-tuning_run("Code_Tuning_With_Flags_00.R",runs_dir =
'_tuningE1',flags=list(dropout1= c(0,0.05),
                              units = c(33,67),
                              activation1=("relu"),
                              batchsize1=c(28),
                              Epoch1=c(1000),
                              learning_rate=c(0.001),
                              val_split=c(0.2)),sample=1,confirm =
                              FALSE,echo =F)

#####c) Saving each combination of hyperparameters in the Hyperpar
data.frame
  #### Ordering in the same way all grids
  runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_dropout1),]
  runs.sp$grid_length=1:nrow(runs.sp)
  Parameters=data.frame(grid_length=runs.sp$grid_length,
metric_val_mse=runs.sp$metric_val_mse,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_activation1=runs.
sp$flag_activation1)
  Hyperpar=rbind(Hyperpar,data.frame(Parameters))
 }
#####d) Summarizing the five inner fold by hyperparameter combination
 Hyperpar %>%
  group_by(grid_length) %>%
  summarise(val_mse=mean(metric_val_mse),
 dropout1=mean(flag_dropout1),
 units=mean(flag_units),
 batchsize1=mean(flag_batchsize1),
 learning_rate=mean(flag_learning_rate),
 epochs=mean( epochs_completed)) %>%
  select(grid_length,val_mse,dropout1,units,batchsize1,
learning_rate, epochs) %>%
  mutate_if(is.numeric, funs(round(., 3))) %>%
  as.data.frame() -> Hyperpar_Opt

######e) Selecting the optimal hyperparameters###########
 Min=min(Hyperpar_Opt$val_mse)
 pos_opt=which(Hyperpar_Opt$val_mse==Min)
 pos_opt=pos_opt[1]
 Optimal_Hyper=Hyperpar_Opt[pos_opt,]
 #####Selecting the best hyperparameters
 Drop_O=Optimal_Hyper$dropout1
 Epoch_O=round(Optimal_Hyper$epochs,0)
 Units_O=round(Optimal_Hyper$units,0)
 activation_O=unique(Hyperpar$flag_activation1)
 batchsize_O=round(Optimal_Hyper$batchsize1,0)
 lr_O=Optimal_Hyper$learning_rate
```

```
print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")})

########f) Refitting the model with the optimal values#################
model_Sec<-keras_model_sequential()
model_Sec %>%
 layer_dense(units =Units_O , activation =activation_O, input_shape =
c(dim(X_trn)[2])) %>%
 layer_dropout(rate =Drop_O) %>%
 layer_dense(units =1, activation =activation_O)

model_Sec %>% compile(
 loss = "mean_squared_error",
 optimizer = optimizer_adam(lr=lr_O),
 metrics = c("mean_squared_error"))

ModelFited <-model_Sec %>% fit(
 X_trn, y_trn,
 epochs=Epoch_O, batch_size =batchsize_O, #####validation_split=0.2,
early_stop,
 verbose=0,callbacks=list(print_dot_callback))

############g) Prediction of testing set ##########################
Yhat=model_Sec%>% predict(X_tst)
y_p=Yhat
y_p_tst=as.numeric(y_p)
y_tst=y[tst_set]
plot(y_tst,y_p_tst)
MSE=mean((y_tst-y_p_tst)^2)
MSE
```

In part a) of the above code, the number of inner TRN–TST partitions that will be implemented is specified; in this case, there are five. Then a data.frame called Hyperpar is defined in which the results of each combination of hyperparameters resulting from each inner partition will be saved. Then we start the loop for implementing the five inner CV strategies for selecting the best hyperparameters. Then the outer training set is shuffled (permuted; using sample without replacement) in such a way that the TRN–TST partition constructed internally in Keras using the validation_set function does not use the same validation and training set in each inner partition.

Part b) of this code specifies the grid for each hyperparameter. It is important to point out that two values were used, 0 (0%) and 0.05 (5%), for dropout. Also, two values, 33 and 67, were used for units, while for the rest of the hyperparameters only one value was used. For this reason, a total of four hyperparameter combinations should be evaluated. It is important to point out that you can put many more values for each hyperparameter, for example, using a grid of 4 values for each hyperparameter implies a total of $4^7 = 16{,}384$ combinations since 7 hyperparameters

can be tuned with the flags specified in the above code, which illustrates that the hyperparameter search space explodes quickly with DL models since there are many hyperparameters that need to be tuned. For this reason, the tuning_run () function, in addition to implementing by default a full Cartesian grid search (with sample = 1), also allows implementing a random grid search by specifying a value between 0 and 1 in the sample command inside the tuning_run() function. For example, by specifying sample = 0.25, only 25% of the total combinations of hyperparameters should be evaluated.

In part c) of the code, first the outputs of the tuning_run function are ordered, that were saved in the runs.sp object. The ordering should be done for all those hyperparameters that have more than one value for evaluation. We did the ordering process only for dropout1 and units (hyperparameter neurons) that have two values each to be evaluated. However, if more hyperparameters contain more than one value to be evaluated, they should also be ordered. Then we added to the runs.sp data frame the unique consecutive number for each unique combination of hyperparameters under study. Then in parameters, in addition to the grid length, the important hyperparameters are saved for each inner fold in the Hyperpar data frame. In part d) the average prediction performance is obtained for each of the four hyperparameters under study in this problem. In e) the best combination of hyperparameters is selected. Then in part f), the model with the optimal hyperparameters is refitted, and finally in part g), the predictions of the second-outer cross-validation are obtained.

Before giving the output of the refitted model, we printed the average prediction performance of the five inner cross-validations of the four combinations of hyperparameters in the grid search for the outer fold cross-validation of the second partition:

```
> Hyperpar_Opt
  grid_length val_mse dropout1 units batchsize1 learning_rate epochs
1 1             8.093  0.00     33    28         0.001         298.0
2 2             1.008  0.05     33    28         0.001         299.4
3 3            13.309  0.00     67    28         0.001         177.2
4 4             0.971  0.05     67    28         0.001         247.6
```

We can see that the best combination of hyperparameters, in partition 2, of the inner cross-validation is combination 4 (with dropout = 0.05, units = 67, and epochs = 247.6) since it has the lowest MSE (0.971) in the testing set (validation set). Then with the optimal values of dropout, epochs, and neurons, the ANN was refitted, but using the whole second outer training set, and then, with the final trained ANN, the predictions for the outer testing set were obtained. Finally, the plot of the observed and predicted values of the outer testing set was done, as shown in Fig. 11.9. The MSE was calculated and was equal to (0.810), which is lower than the MSE obtained (1.222) using only one split (inner TRN–TST) for partition 2.

Finally, using the code given in Appendix 1, we implemented the above code using the five outer folds and five inner folds, and the MSE resulting from the

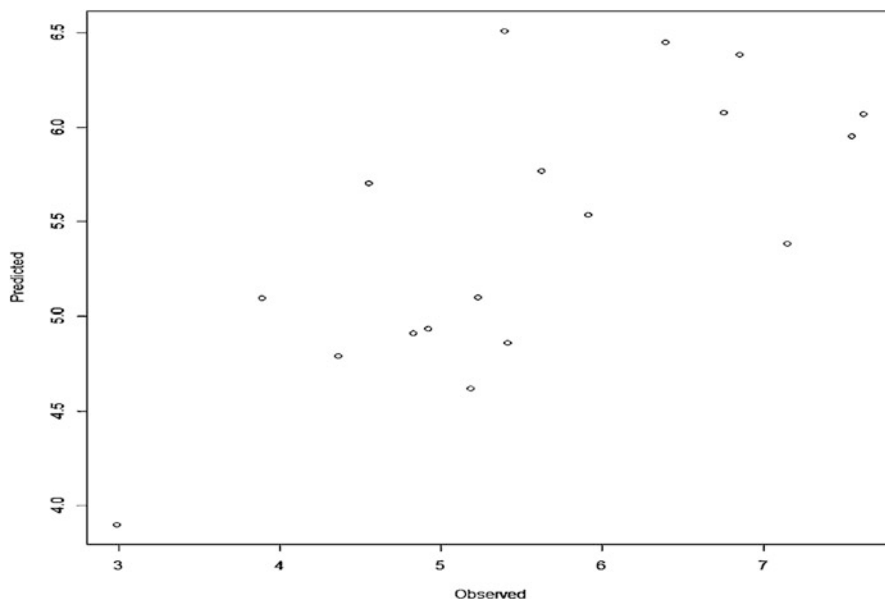



**Fig. 11.9** Observed versus predicted values for fold 2 trained with a grid search

average of the five outer CV is equal to 0.892; it is slightly larger than when using only fold 2, which means that the variability of fold to fold is considerably large.

**Example 11.3 MaizeToy Example with Five-fold CV with Dropout**

Next, we perform a five-fold CV for the MaizeToy data sets but with different percentages of dropout, a type of regularization. It is important to point out that now, in addition to reporting the MSE by fold, we also provide the MSE for each environment. First, we provide the MSE for each environment, then we average the five-folds, using as input the information of environments + the information of lines (taking into account the marker information). The code used to obtain these results is given in Appendix 2. This code also uses the Code_Tuning_With_Flags_00.R, but now the grid search is performed not only for two hyperparameters since we use two values for the following hyperparameters: units, batchsize, and learning_rate, which means that the total combinations to be evaluated are equal to $2^3 = 8$, while the remaining four hyperparameters, validation split, dropout, activation function, and number of epochs, were fixed at 0.20, 0, relu, and 1000, respectively. Next we provide the key part of the Appendix 2 code:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_00.R",flags=list
(dropout1= c(0),
                                 units = c(33,67),
                                 activation1=("relu"),
                                 batchsize1=c(28,56),
```

```
                    Epoch1=c(1000),
                    learning_rate=c(0.001,0.01),
                val_split=c(0.2)), sample=1, confirm=FALSE,echo =F),
```

where it is clear that the % of dropout value was fixed at zero. Of course, when you want to use other dropout percentages, you only need to specify in dropout1 the new value in the above code for tuning and it is modified in the Code_Tuning_With_Flags_00.R the following part of the code

```
layer_dropout(rate = 0.05) %>%
```

This means that when the Code_Tuning_With_Flags_00.R is executed with 5% dropout, in this code, the line above is modified with a rate = 0.05, but if you want to run with 25% dropout, the rate parameter is changed to 0.25. This code in Appendix 2 is run for the following dropout percentages: 0%, 5%, 15%, 25%, and 35%. This code was run once for a grid of the five dropout percentage values, but first without taking into account the genotype by environment interaction and then with the genotype by environment interaction. It is important to point out that by changing the value of the sample argument in the tuning_run() function, we can implement the random search grid that tests only the proportion of hyperparameter combinations that is specified in the sample. For example, if we specify sample = 0.5, this means that only half of the total number of hyperparameter combinations in the grid will be evaluated, while if we specify sample = 0.10, only 10% of the total number of hyperparameter combinations should be evaluated. However, to use the code in Appendix 2 with a value of sample less than one, it is also important to modify the number of inner testing CV to 1 (nCVI = 1, ####Number of folds for inner CV) since with more than two (nCVI = 2, 3, 4, . . .), the R code given in Appendix 2 is not valid.

Table 11.2 shows that in general, the prediction performance in terms of MSE and MAAPE improved using different dropout percentages, since the best prediction performance was observed using 15% of dropout (rows in bold).

Table 11.3 gives the prediction performance using the same dropout percentages using the code given in Appendix 2, but now taking into account as predictors the information corresponding to the $G \times E$ interaction term which is obtained by replacing, in Appendix 2, X = cbind(Z.E, Z.G) with X = cbind(Z.E, Z.G, Z.EG). This table shows that there is a gain in prediction performance with a dropout percentage larger than zero and the best predictions are obtained with 15% dropout. However, it is clear that the predictions are worse when the $G \times E$ interaction term is ignored, which can be attributed in part to the fact that the number of data sets used is small and that ANN with at least one hidden layer can capture complex interactions by the nonlinear transformations performed in the hidden layer. Perhaps for this reason, in many cases providing as input the information corresponding to the $G \times E$ interaction term is not required. It is possible to increase the prediction accuracy of this model with the $G \times E$ interaction term by increasing the number of neurons in the hidden layer, but this also increases the computational resources required.

**Table 11.2** Prediction performance of trait yield with five-fold CV with different dropout percentages without the genotype × environment interaction term

| % Dropout | Environment | Trait | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| 0 | EBU | Yield | 1.538 | 0.350 | 0.174 | 0.018 |
| 0 | KAK | Yield | 0.443 | 0.097 | 0.103 | 0.016 |
| 0 | KTI | Yield | 1.222 | 0.185 | 0.154 | 0.021 |
| 5 | EBU | Yield | 1.371 | 0.315 | 0.163 | 0.016 |
| 5 | KAK | Yield | 0.407 | 0.091 | 0.105 | 0.012 |
| 5 | KTI | Yield | 1.140 | 0.206 | 0.144 | 0.021 |
| **15** | **EBU** | **Yield** | **1.465** | **0.278** | **0.168** | **0.018** |
| **15** | **KAK** | **Yield** | **0.380** | **0.066** | **0.101** | **0.009** |
| **15** | **KTI** | **Yield** | **1.055** | **0.214** | **0.138** | **0.018** |
| 25 | EBU | Yield | 1.540 | 0.332 | 0.174 | 0.020 |
| 25 | KAK | Yield | 0.473 | 0.112 | 0.112 | 0.016 |
| 25 | KTI | Yield | 1.194 | 0.184 | 0.144 | 0.018 |
| 35 | EBU | Yield | 1.348 | 0.227 | 0.169 | 0.018 |
| 35 | KAK | Yield | 0.411 | 0.110 | 0.100 | 0.016 |
| 35 | KTI | Yield | 1.280 | 0.183 | 0.150 | 0.019 |

**Table 11.3** Prediction performance of trait yield with five-fold CV with different dropout percentages with the $G \times E$ interaction term

| % Dropout | Environment | Trait | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| 0 | EBU | Yield | 1.772 | 0.137 | 0.176 | 0.012 |
| 0 | KAK | Yield | 0.696 | 0.212 | 0.126 | 0.020 |
| 0 | KTI | Yield | 1.422 | 0.255 | 0.150 | 0.015 |
| 5 | EBU | Yield | 10.849 | 9.412 | 0.289 | 0.125 |
| 5 | KAK | Yield | 5.518 | 5.003 | 0.246 | 0.135 |
| 5 | KTI | Yield | 6.801 | 5.397 | 0.279 | 0.128 |
| **15** | **EBU** | **Yield** | **1.680** | **0.235** | **0.168** | **0.012** |
| **15** | **KAK** | **Yield** | **0.536** | **0.094** | **0.116** | **0.011** |
| **15** | **KTI** | **Yield** | **1.426** | **0.243** | **0.153** | **0.015** |
| 25 | EBU | Yield | 1.867 | 0.259 | 0.185 | 0.010 |
| 25 | KAK | Yield | 0.592 | 0.162 | 0.116 | 0.017 |
| 25 | KTI | Yield | 1.393 | 0.285 | 0.150 | 0.013 |
| 35 | EBU | Yield | 1.771 | 0.289 | 0.175 | 0.016 |
| 35 | KAK | Yield | 0.581 | 0.128 | 0.113 | 0.015 |
| 35 | KTI | Yield | 1.413 | 0.294 | 0.147 | 0.015 |

**Example 11.4 MaizeToy Example with More Than One Hidden Layer with Inner CV**

The goal of this section is to show how to train DL models with different number of layers and how they impact the prediction performance. Now the grid contains the number of neurons (33, 67), three dropout percentages (0, 5, and 10%) and two batch

size values (28, 56), and again the number of epochs is chosen using the early stopping approach. All 12 combinations are evaluated, since they were implemented using the full grid search with sample = 1. It is important to point out that this grid was implemented with an outer 5FCV and inner 5FCV. The code used for implementing this grid with four hidden layers is given in Appendix 4 (this code call the code given in Appendix 3 that contains the flags). It is very similar to the code used in Appendix 2, but one of the most important parts is how the layers are added to this code. Next we provide the part where the layers are added.

```
model %>%
 layer_dense(units =Units_O , activation =activation_O, input_shape =
c(dim(X_trn)[2])) %>%
 layer_dropout(rate =Drop_O) %>%
 layer_dense(units =Units_O , activation =activation_O) %>%
 layer_dropout(rate =Drop_O) %>%
 layer_dense(units =Units_O , activation =activation_O) %>%
 layer_dropout(rate =Drop_O) %>%
 layer_dense(units =Units_O , activation =activation_O) %>%
 layer_dropout(rate =Drop_O) %>%
 layer_dense(units =1)
```

This part of the code that forms part of Appendix 4 is for a DL model with four hidden layers; the lines that start with layer_dense() specified all the layers, but the first four correspond to the hidden layers and the last one to the output layer. Also, this DL model corresponds to a neural network with five layers.

Table 11.4 shows the prediction performance with one to four different hidden layers; the prediction performance is affected by the number of hidden layers used, but in this case, the best predictions across environments were obtained with three hidden layers.

**Table 11.4** Prediction performance of trait yield with five outer fold CV, five inner fold CV, with different dropout percentages without the $G \times E$ interaction term. Env denotes environments

| No. layers | Environment | Trait | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| 1 | EBU | Yield | 1.489 | 0.316 | 0.170 | 0.018 |
| 1 | KAK | Yield | 0.426 | 0.088 | 0.102 | 0.015 |
| 1 | KTI | Yield | 1.251 | 0.147 | 0.153 | 0.020 |
| 2 | EBU | Yield | 1.456 | 0.260 | 0.168 | 0.016 |
| 2 | KAK | Yield | 0.427 | 0.110 | 0.103 | 0.015 |
| 2 | KTI | Yield | 1.118 | 0.115 | 0.151 | 0.016 |
| **3** | **EBU** | **Yield** | **1.372** | **0.212** | **0.157** | **0.015** |
| **3** | **KAK** | **Yield** | **0.412** | **0.062** | **0.101** | **0.007** |
| **3** | **KTI** | **Yield** | **1.134** | **0.203** | **0.148** | **0.022** |
| 4 | EBU | Yield | 1.724 | 0.291 | 0.184 | 0.017 |
| 4 | KAK | Yield | 0.350 | 0.060 | 0.092 | 0.008 |
| 4 | KTI | Yield | 1.320 | 0.298 | 0.162 | 0.024 |

**Table 11.5** Prediction performance of trait yield with five outer fold CV, one inner fold CV, with different hidden layers without the $G \times E$ interaction term and the inner CV

| No. layers | Env | Trait | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| 1 | EBU | Yield | 1.2767 | 0.1807 | 0.1469 | 0.0131 |
| 1 | KAK | Yield | 0.3508 | 0.0613 | 0.0962 | 0.0123 |
| 1 | KTI | Yield | 1.0319 | 0.1332 | 0.138 | 0.0135 |
| 2 | EBU | Yield | 1.1731 | 0.1579 | 0.1408 | 0.012 |
| 2 | KAK | Yield | 0.4397 | 0.1274 | 0.1049 | 0.0161 |
| 2 | KTI | Yield | 1.2271 | 0.1211 | 0.151 | 0.0134 |
| **3** | **EBU** | **Yield** | **1.1565** | **0.1844** | **0.1435** | **0.007** |
| **3** | **KAK** | **Yield** | **0.4852** | **0.1157** | **0.1138** | **0.0186** |
| **3** | **KTI** | **Yield** | **1.1017** | **0.1213** | **0.1376** | **0.0155** |
| 4 | EBU | Yield | 1.3575 | 0.1894 | 0.1602 | 0.0077 |
| 4 | KAK | Yield | 0.4864 | 0.1152 | 0.1238 | 0.0189 |
| 4 | KTI | Yield | 1.2323 | 0.2545 | 0.1481 | 0.0092 |

**Example 11.5 MaizeToy Example with More Than One Hidden Layer Without Inner CV**
The goal of this example is to implement exactly the same grid used in Example 11.4, but instead of implementing the five inner fold CV, we implemented this with only one inner random partition. The code used here is the one given in Appendix 5 with only one inner cross-validation.

Table 11.5 shows that the best predictions are observed with three hidden layers, but it is important to point out that in general the prediction performances using only one inner random partition were not quite similar to performances using five inner fold CV, as can be observed in Table 11.4 in Example 11.4.

**Example 11.6 MaizeToy Example with One Hidden Layer with Inner CV and Ridge, Lasso, and Elastic Net (Ridge-Lasso) Regularization**
The code used for implementing a grid search with an outer and inner CV with Ridge regularization (L2) is similar to the one given in Appendix 2, with the difference that now for specifying the model the following part of the code is used:

```
model_Sec %>%
 layer_dense(units =Units_O , activation =activation_O,
kernel_regularizer=regularizer_l2(Reg_l2_O), input_shape = c(dim
(X_trn)[2])) %>%
 layer_dropout(rate =Drop_O) %>%
 layer_dense(units =1, activation ="linear")
```

The only new part of this code is the part that contains kernel_regularizer = regularizer_l2(Reg_l2_O) that specifies Ridge regularization, and in Ridge_values, the following values were used: 0.001, 0.01, 0.1, 0.5, 1. The grid here is composed of four values due to two unit values (33, 67) and two batch size values (28, 56); we used sample = 1, which means that the four hyperparameters

**Table 11.6** Prediction performance of trait yield with five outer fold CV and five-fold inner CV with different values of L2 regularization (0.001, 0.01, 0.1, 0.5, 1) without the $G \times E$ interaction term

| L2 value | Environment | Trait | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| 0.001 | EBU | Yield | 1.639 | 0.231 | 0.166 | 0.011 |
| 0.001 | KAK | Yield | 0.495 | 0.118 | 0.105 | 0.008 |
| 0.001 | KTI | Yield | 1.366 | 0.268 | 0.150 | 0.017 |
| **0.01** | **EBU** | **Yield** | **1.387** | **0.119** | **0.169** | **0.013** |
| **0.01** | **KAK** | **Yield** | **0.338** | **0.080** | **0.085** | **0.008** |
| **0.01** | **KTI** | **Yield** | **1.132** | **0.217** | **0.149** | **0.023** |
| 0.1 | EBU | Yield | 1.371 | 0.252 | 0.168 | 0.022 |
| 0.1 | KAK | Yield | 0.384 | 0.090 | 0.093 | 0.010 |
| 0.1 | KTI | Yield | 1.165 | 0.247 | 0.144 | 0.026 |
| 0.5 | EBU | Yield | 1.372 | 0.315 | 0.167 | 0.026 |
| 0.5 | KAK | Yield | 0.428 | 0.132 | 0.101 | 0.015 |
| 0.5 | KTI | Yield | 1.194 | 0.234 | 0.141 | 0.025 |
| 1 | EBU | Yield | 1.413 | 0.342 | 0.168 | 0.027 |
| 1 | KAK | Yield | 0.461 | 0.151 | 0.107 | 0.019 |
| 1 | KTI | Yield | 1.215 | 0.233 | 0.142 | 0.025 |

were evaluated. Each of these values will add Ridge_val×weight_coefficient_value to the total loss of the network. Since this penalty is only added at training time, the loss for this network will be much bigger at training than at testing time. To reproduce these results, you can replace in Appendices A3 and A4 the specification of the model provided above each time you need to modify in kernel_regularizer = regularizer_l2(Reg_l2_O) the corresponding value of the Ridge regularization parameter.

Table 11.6 gives the output without taking into account the $G \times E$ interaction using the following L2 regularization values of lambda, 0.001, 0.01, 0.1, 0.5, 1, implemented with the early stopping method. We can see in Table 11.6 that the best performance was obtained using the value of 0.01 for the L2 regularization hyperparameter.

Table 11.7 provides the output without taking into account the $G \times E$ interaction for the L1 regularization for the same grid containing the following values of lambda, 0.001, 0.01, 0.1, 0.5, 1, two neuron values (33, 67), two batch size values (28, 56), and 1000 epochs implemented with early stopping. The performance was very similar using different values of L1 for regularization, but the performance under the L1 regularization using a value of 0.01 was slightly better.

In similar fashion, Table 11.8 gives the output without taking into account the $G \times E$ interaction for five regularization parameters (0.001, 0.01, 0.1, 0.5, 1) using the Elastic Net regularization (L1-L2) for two neuron values (33, 67), two batch size values (28, 56), and 1000 epochs implemented with early stopping. We can see in Table 11.8 that the best performance was obtained using the value of 0.001 of the L1-L2 regularization parameter across environments under the MAAPE and a value of 0.01 of the L1-L2 regularization parameter under the MSE.

**Table 11.7** Prediction performance of trait yield with five outer fold CV, five inner fold CV with different values of L1 regularization (0.001, 0.01, 0.1, 0.5, 1) in the grid without the $G \times E$ interaction term

| L1 value | Environment | Trait | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| 0.001 | EBU | Yield | 1.5623 | 0.2358 | 0.1668 | 0.0154 |
| 0.001 | KAK | Yield | 0.4312 | 0.099 | 0.0959 | 0.0038 |
| 0.001 | KTI | Yield | 1.2691 | 0.2402 | 0.1459 | 0.0178 |
| **0.01** | **EBU** | **Yield** | **1.4894** | **0.2366** | **0.1762** | **0.0186** |
| **0.01** | **KAK** | **Yield** | **0.4366** | **0.0717** | **0.1055** | **0.0113** |
| **0.01** | **KTI** | **Yield** | **1.1368** | **0.2443** | **0.1365** | **0.0285** |
| 0.1 | EBU | Yield | 1.4492 | 0.3605 | 0.1679 | 0.0275 |
| 0.1 | KAK | Yield | 0.4813 | 0.1466 | 0.1138 | 0.0203 |
| 0.1 | KTI | Yield | 1.2664 | 0.2405 | 0.1439 | 0.0236 |
| 0.5 | EBU | Yield | 1.6529 | 0.389 | 0.1669 | 0.0288 |
| 0.5 | KAK | Yield | 1.0277 | 0.2506 | 0.183 | 0.025 |
| 0.5 | KTI | Yield | 1.2195 | 0.2598 | 0.1403 | 0.0232 |
| 1 | EBU | Yield | 1.6669 | 0.3923 | 0.1675 | 0.029 |
| 1 | KAK | Yield | 1.0326 | 0.2575 | 0.1829 | 0.026 |
| 1 | KTI | Yield | 1.2108 | 0.2631 | 0.1398 | 0.0233 |

**Table 11.8** Prediction performance of trait yield with five outer fold CV, five inner fold CV, with different values of L1-L2 regularization (0.001, 0.01, 0.1, 0.5, 1) in the grid without the $G \times E$ interaction term

| L1_L2 value | Environment | Trait | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| 0.001 | EBU | Yield | 1.4893 | 0.1938 | **0.167** | 0.0154 |
| 0.001 | KAK | Yield | 0.3991 | 0.1053 | **0.0909** | 0.0067 |
| 0.001 | KTI | Yield | 1.2316 | 0.2226 | **0.1459** | 0.0181 |
| 0.01 | EBU | Yield | **1.4719** | 0.2554 | 0.1761 | 0.0207 |
| 0.01 | KAK | Yield | **0.4286** | 0.0767 | 0.1038 | 0.0096 |
| 0.01 | KTI | Yield | **1.1693** | 0.2441 | 0.139 | 0.0276 |
| 0.1 | EBU | Yield | 1.4484 | 0.3632 | 0.1677 | 0.0277 |
| 0.1 | KAK | Yield | 0.5088 | 0.164 | 0.1141 | 0.0213 |
| 0.1 | KTI | Yield | 1.2798 | 0.2451 | 0.1436 | 0.0236 |
| 0.5 | EBU | Yield | 1.6546 | 0.3837 | 0.1672 | 0.0289 |
| 0.5 | KAK | Yield | 1.0238 | 0.2445 | 0.1825 | 0.0247 |
| 0.5 | KTI | Yield | 1.2159 | 0.2599 | 0.1398 | 0.023 |
| 1 | EBU | Yield | 1.6574 | 0.3836 | 0.1674 | 0.0288 |
| 1 | KAK | Yield | 1.0372 | 0.256 | 0.1833 | 0.026 |
| 1 | KTI | Yield | 1.2208 | 0.2609 | 0.1402 | 0.0234 |

**Table 11.9** Prediction performance of trait yield with five outer and five inner fold CV with different different optimizers without the genotype × environment interaction term

| Optimizer | Environment | Trait | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| optimizer_adam | EBU | Yield | 1.7009 | 0.2456 | 0.1679 | 0.0098 |
| optimizer_adam | KAK | Yield | 0.6735 | 0.1827 | 0.1254 | 0.019 |
| optimizer_adam | KTI | Yield | 1.4192 | 0.295 | 0.1505 | 0.0157 |
| optimizer_sgd | EBU | Yield | 1.457 | 0.3499 | 0.1702 | 0.0264 |
| optimizer_sgd | KAK | Yield | 0.4953 | 0.1385 | 0.1074 | 0.014 |
| optimizer_sgd | KTI | Yield | 1.2861 | 0.2312 | 0.1457 | 0.0236 |
| optimizer_rmsprop | EBU | Yield | 1.7165 | 0.2187 | 0.1682 | 0.0093 |
| optimizer_rmsprop | KAK | Yield | 0.862 | 0.2246 | 0.1484 | 0.0242 |
| optimizer_rmsprop | KTI | Yield | 1.5172 | 0.2486 | 0.1582 | 0.0103 |
| optimizer_adagrad | EBU | Yield | 32.0716 | 2.8222 | 0.7224 | 0.009 |
| optimizer_adagrad | KAK | Yield | 19.0737 | 1.3223 | 0.7157 | 0.0146 |
| optimizer_adagrad | KTI | Yield | 26.0565 | 1.9791 | 0.7131 | 0.0079 |
| optimizer_adadelta | EBU | Yield | 39.5482 | 2.2787 | 0.7805 | 0.0018 |
| optimizer_adadelta | KAK | Yield | 24.0182 | 0.9201 | 0.7783 | 0.007 |
| optimizer_adadelta | KTI | Yield | 33.2928 | 2.8686 | 0.7779 | 0.0041 |
| **optimizer_adamax** | **EBU** | **Yield** | **1.4955** | **0.1743** | **0.1664** | **0.0117** |
| **optimizer_adamax** | **KAK** | **Yield** | **0.427** | **0.117** | **0.0932** | **0.0093** |
| **optimizer_adamax** | **KTI** | **Yield** | **1.2982** | **0.2195** | **0.1522** | **0.0229** |
| optimizer_nadam | EBU | Yield | 1.709 | 0.2438 | 0.168 | 0.0098 |
| optimizer_nadam | KAK | Yield | 0.6759 | 0.1839 | 0.1255 | 0.0191 |
| optimizer_nadam | KTI | Yield | 1.4208 | 0.2942 | 0.1506 | 0.0155 |

**Example 11.7 MaizeToy Example with One Hidden Layer with Inner CV, a Dropout Rate of 5%, and Different Optimizers**
In Table 11.9 are given the results of evaluating seven types of optimizers for a five outer and five inner fold cross-validations without taking into account the genotype×environment interaction. The code given in Appendix 2 was used in this implementation, but with a 0% dropout rate and with only one hidden layer. The neuron grid consists of two neurons (33 and 67), two batch sizes (28, 56) and they were implemented with early stopping to find the optimal number of epochs, which was fixed at 1000. Since the code given in Appendix 2 was used each time, this code was run by replacing in the optimizer each of the seven types of optimizers. The results in Table 11.9 show that the best performance in terms of prediction accuracy was found under the **optimizer_adamax** and the worst under the **optimizer_adagrad.**

Finally, with these examples we illustrated how to use DNN models for implementing genomic prediction models for continuous outcomes in Keras. Most of the key elements for implementing DNN models were explained to enable you to implement these models on your own data without any complications. However, as mentioned before, the DNN training process is challenging due to the complexity of tuning hyperparameters, but thanks to functions like the tfruns, this work is not as hard. However, more research and automatization of all these activities are required to be able to implement DNN models more quickly and efficiently since the tuning process still requires a lot of time.

# Appendix 1

MaizeToy example with five-fold CV.

```
rm(list=ls())
library(lsa)
library(keras)
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(tfruns)
options(bitmapType='cairo')

################Set seed for reproducible results###################
use_session_with_seed(45)

####################Loading the MaizeToy data sets##################
data("MaizeToy")
head(phenoMaizeToy)

#################Ordering the data ################################
phenoMaizeToy<-(phenoMaizeToy[order(phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames(phenoMaizeToy)=1:nrow(phenoMaizeToy)
head(phenoMaizeToy,8)

#######################Design matrices###########################
LG <- cholesky(genoMaizeToy)
ZG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line))
Z.G <- ZG %*%LG
Z.E <- model.matrix(~0 + as.factor(phenoMaizeToy$Env))
ZEG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker(diag(length(unique(phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky(G2)
Z.EG <- ZEG %*% LG2

###############Selecting the response variable#####################
Y <- as.matrix(phenoMaizeToy[, -c(1, 2)])

#########Training-testing sets using the BMTME package#############
pheno <- data.frame(GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
         Response = phenoMaizeToy[, 3])

CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)
```

```
################Final X and y for fitting the model###################
y=(phenoMaizeToy[, 3])
X=cbind(Z.E,Z.G)

#####################Outer cross-validation#######################
digits=4
Names_Traits=colnames(Y)
results=data.frame()
t=1
for (o in 1:5){
#o=2
 tst_set=CrossV$CrossValidation_list[[o]]
 X_trn=(X[-tst_set,])
 X_tst=(X[tst_set,])
 y_trn=(y[-tst_set])
 y_tst=(y[tst_set])

 ###################Inner cross-validation########################
 nCVI=5 ####Number of folds for inner CV
 Hyperpar=data.frame()
 for (i in 1:nCVI){
  set.seed(i+100)
  Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
  X_trII=X_trn[Sam_per,]
  y_trII=y_trn[Sam_per]

  #########Using the tuning_run function for tuning
hyperparameters#####
  runs.sp<-tuning_run("Code_Tuning_With_Flags_00.R",flags=list
(dropout1= c(0,0.05), units = c(33,67), activation1=("relu"),
batchsize1=c(28), Epoch1=c(1000), learning_rate=c(0.001),
val_split=c(0.2)), sample=1,confirm =FALSE,echo =F)

  ###### Ordering in the same way all grids
  runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_dropout1),]

  runs.sp$grid_length=1:nrow(runs.sp)
  Parameters=data.frame(grid_length=runs.sp$grid_length,
metric_val_mse=runs.sp$metric_val_mse,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_activation1=runs.
sp$flag_activation1)
  Hyperpar=rbind(Hyperpar,data.frame(Parameters))
 }
 Hyperpar %>%
  group_by(grid_length) %>%
  summarise(val_mse=mean(metric_val_mse),
 dropout1=mean(flag_dropout1),
 units=mean(flag_units),
 batchsize1=mean(flag_batchsize1),
 learning_rate=mean(flag_learning_rate),
```

```
 epochs=mean( epochs_completed)) %>%
  select(grid_length,val_mse,dropout1,units,batchsize1,
learning_rate, epochs) %>%
  mutate_if(is.numeric, funs(round(., 3))) %>%
  as.data.frame() -> Hyperpar_Opt
########################Optimal hyperparameters#################
 Min=min(Hyperpar_Opt$val_mse)
 pos_opt=which(Hyperpar_Opt$val_mse==Min)
 pos_opt=pos_opt[1]
 Optimal_Hyper=Hyperpar_Opt[pos_opt,]
 #####Selecting the best hyperparameters
 Drop_O=Optimal_Hyper$dropout1
 Epoch_O=round(Optimal_Hyper$epochs,0)
 Units_O=round(Optimal_Hyper$units,0)
 activation_O=unique(Hyperpar$flag_activation1)
 batchsize_O=round(Optimal_Hyper$batchsize1,0)
 lr_O=Optimal_Hyper$learning_rate

 print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
   if (epoch %% 20 == 0) cat("\n")
   cat(".")})

 ###########Refitting the model with the optimal values#############
 model_Sec<-keras_model_sequential()

 model_Sec %>%
  layer_dense(units =Units_O , activation =activation_O, input_shape =
c(dim(X_trn)[2])) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =1)

 model_Sec %>% compile(
  loss = "mean_squared_error",
  optimizer = optimizer_adam(lr=lr_O),
  metrics = c("mean_squared_error"))

 ModelFited <-model_Sec %>% fit(
  X_trn, y_trn,
  epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
  verbose=0,callbacks=list(print_dot_callback))

 ###############Prediction of testing set #########################
 Yhat=model_Sec%>% predict(X_tst)
 y_p=Yhat
 y_p_tst=as.numeric(y_p)
 y_tst=y[tst_set]
 MSE=mean((y_tst-y_p_tst)^2)
 MSE
```

```
 ###########Saving the predictions of each outer testing set##########
 results<-rbind(results, data.frame(Partition=o,MSE_Env=MSE))
 cat("CV=",o,"\n")
}

####Average prediction performance
MSE_global=mean(results$MSE_Env)
MSE_global
```

## Appendix 2

MaizeToy example with five-fold CV with dropout.

```
rm(list=ls())
library(lsa)
library(keras)
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(tfruns)
options(bitmapType='cairo')

################Set seed for reproducible results###################
use_session_with_seed(45)

#####################Loading the MaizeToy data sets#################
data("MaizeToy")
head(phenoMaizeToy)

###################Ordering the data ###############################
phenoMaizeToy<-(phenoMaizeToy[order(phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames(phenoMaizeToy)=1:nrow(phenoMaizeToy)
head(phenoMaizeToy,8)

#####################Design matrices##############################
LG <- cholesky(genoMaizeToy)
ZG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line))
Z.G <- ZG %*%LG
Z.E <- model.matrix(~0 + as.factor(phenoMaizeToy$Env))
ZEG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker(diag(length(unique(phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky(G2)
Z.EG <- ZEG %*% LG2
```

```
###############Selecting the response variable#####################
Y <- as.matrix(phenoMaizeToy[, -c(1, 2)])

#########Training-testing sets using the BMTME package###############
pheno <- data.frame(GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
          Response = phenoMaizeToy[, 3])

CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)

################Function for averaging the predictions##############
summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
 results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(MSE = mean((Predicted-Observed)^2),
      MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed)))) %>%
  select(Environment, Trait, Partition, MSE, MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Environment, Trait) %>%
  summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
     SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>
%
  select(Environment, Trait, MSE, SE_MSE, MAAPE, SE_MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
        complete = presum,
        extended = {
         finalSum$Partition <- 'All'
         presum$Partition <- as.character(presum$Partition)
         presum$SE_MSE <- NA
         presum$SE_MAAPE <- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

################Final X and y for fitting the model###################
y=(phenoMaizeToy[, 3])
X=cbind(Z.E,Z.G)
Drop_per=c(0,0.05,0.15,0.25,0.35)
Final_results=data.frame()
for (e in 1:5){
 #e=1
 ############Outer cross-validation######################
 digits=4
```

```
Names_Traits=colnames(Y)
results=data.frame()
t=1
for (o in 1:5){
  #o=1
  tst_set=CrossV$CrossValidation_list[[o]]
  X_trn=(X[-tst_set,])
  X_tst=(X[tst_set,])
  y_trn=(y[-tst_set])
  y_tst=(y[tst_set])

  ################Inner cross-validation###########################
  nCVI=5  ####Number of folds for inner CV
  Hyperpar=data.frame()
  for (i in 1:nCVI){
   #i=1
   Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
   X_trII=X_trn[Sam_per,]
   y_trII=y_trn[Sam_per]

   #########Using the tuning_fun function for tuning
hyperparameters#####
   runs.sp<-tuning_run("Code_Tuning_With_Flags_00.R",flags=list
(dropout1= Drop_per[e],
                                        units = c(33,67),
                                        activation1=("relu"),
                                        batchsize1=c(28,56),
                                        Epoch1=c(1000),
                                        learning_rate=c(0.001,0.01),
                                        val_split=c(0.2)),sample=1,
                                        confirm =FALSE,echo =F)

   ###### Ordering in the same way all grids
   runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_batchsize1,
runs.sp$flag_learning_rate),]

   runs.sp$grid_length=1:nrow(runs.sp)
   Parameters=data.frame(grid_length=runs.sp$grid_length,
metric_val_mse=runs.sp$metric_val_mse,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_activation1=runs.
sp$flag_activation1)
   Hyperpar=rbind(Hyperpar,data.frame(Parameters))
  }
  Hyperpar %>%
   group_by(grid_length) %>%
   summarise(val_mse=mean(metric_val_mse),
        dropout1=mean(flag_dropout1),
        units=mean(flag_units),
        batchsize1=mean(flag_batchsize1),
        learning_rate=mean(flag_learning_rate),
        epochs=mean( epochs_completed)) %>%
```

```
   select(grid_length,val_mse,dropout1,units,batchsize1,
learning_rate, epochs) %>%
   mutate_if(is.numeric, funs(round(., 3))) %>%
   as.data.frame() -> Hyperpar_Opt
  ############Optimal hyperparameters############
  Min=min(Hyperpar_Opt$val_mse)
  pos_opt=which(Hyperpar_Opt$val_mse==Min)
  Optimal_Hyper=Hyperpar_Opt[pos_opt,]
  pos_opt=pos_opt[1]
  #####Selecting the best hyperparameters
  Drop_O=Optimal_Hyper$dropout1
  Epoch_O=round(Optimal_Hyper$epochs,0)
  Units_O=round(Optimal_Hyper$units,0)
  activation_O=unique(Hyperpar$flag_activation1)
  batchsize_O=round(Optimal_Hyper$batchsize1,0)
  lr_O=Optimal_Hyper$learning_rate

  print_dot_callback <- callback_lambda(
   on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat("\n")
    cat(".")})

  ############Refitting the model with the optimal values############
  model_Sec<-keras_model_sequential()

  model_Sec %>%
   layer_dense(units =Units_O , activation =activation_O, input_shape
= c(dim(X_trn)[2])) %>%
   layer_dropout(rate =Drop_O) %>%
   layer_dense(units =1)

  model_Sec %>% compile(
   loss = "mean_squared_error",
   optimizer = optimizer_adam(lr=lr_O),
   metrics = c("mean_squared_error"))

  ModelFited <-model_Sec %>% fit(
   X_trn, y_trn,
   epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
   verbose=0,callbacks=list(print_dot_callback))

  ####Prediction of testing set ##########################
  Yhat=model_Sec%>% predict(X_tst)
  y_p=Yhat
  y_p_tst=as.numeric(y_p)

  ###########Saving the predictions of each outer testing
set#################
  results<-rbind(results, data.frame(Position=tst_set,
                  Environment=CrossV$Environments[tst_set],
                  Partition=o,
```

```
                  Units=Units_O,
                  Epochs=Epoch_O,
                  Observed=round(y[tst_set], digits), #$response,
digits),
                  Predicted=round(y_p_tst, digits),
                  Trait=Names_Traits[t]))
  cat("CV=",o,"\n")
 }

 results

 ############Average of prediction performance####################
 Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
 Pred_Summary
 Final_results=rbind(Final_results,data.frame(Pred_Summary))
}
Final_results
write.csv(Final_results,
file="Example5_AppendixA2_modified_Table11.2.csv")
```

## Appendix 3

Flags with four hidden layers.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
 flag_numeric("val_split",0.2))

####b) Defining the DNN model
build_model<-function() {
model <- keras_model_sequential()
model %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1,
input_shape = c(dim(X_trII)[2])) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units=1, activation =”linear”)
```

```
#####c) Compiling the DNN model
model %>% compile(
 loss = "mse",
 optimizer =optimizer_adam(lr=FLAGS$learning_rate),
 metrics = c("mse"))
model}

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")})

early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =50)

###########d) Fitting the DNN model################
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
 X_trII, y_trII,
 epochs =FLAGS$Epoch1, batch_size =FLAGS$batchsize1,
 shuffled=F,
 validation_split =FLAGS$val_split,
 verbose=0,callbacks = list(early_stop,print_dot_callback))
```

## Appendix 4

MaizeToy example with five outer fold CV and five-folds inner CV without the $G \times E$ interaction term and four hidden layers.

```
rm(list=ls())
library(lsa)
library(keras)
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)
library(tfruns)
library(tfestimators) # provides grid search & model training
interface

#########Set seed for reproducible results##################
use_session_with_seed(45)

################Loading the MaizeToy data sets###############
data("MaizeToy")
head(phenoMaizeToy)
```

```
#################Ordering the data ##################################
phenoMaizeToy<-(phenoMaizeToy[order(phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames(phenoMaizeToy)=1:nrow(phenoMaizeToy)
head(phenoMaizeToy,8)

####################Design matrices################################
LG <- cholesky(genoMaizeToy)
ZG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line))
Z.G <- ZG %*%LG
Z.E <- model.matrix(~0 + as.factor(phenoMaizeToy$Env))
ZEG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker(diag(length(unique(phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky(G2)
Z.EG <- ZEG %*% LG2

############Selecting the response variable#######################
Y <- as.matrix(phenoMaizeToy[, -c(1, 2)])

####Training–testing sets using the BMTME package###############
pheno <- data.frame(GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
          Response = phenoMaizeToy[, 3])

CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)

################Function for averaging the predictions############
summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
 results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(MSE = mean((Predicted-Observed)^2),
      MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed)))) %>%
  select(Environment, Trait, Partition, MSE, MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Environment, Trait) %>%
  summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
     SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>
%
  select(Environment, Trait, MSE, SE_MSE, MAAPE, SE_MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
        complete = presum,
        extended = {
         finalSum$Partition <- 'All'
```

```
        presum$Partition <- as.character(presum$Partition)
        presum$SE_MSE <- NA
        presum$SE_MAAPE <- NA
        rbind(presum, finalSum)
       }
 )
 return(out)
}

################Final X and y for fitting the model###################
y=(phenoMaizeToy[, 3])
X=cbind(Z.E,Z.G)

#####################Outer sross-validation#######################
digits=4
Names_Traits=colnames(Y)
results=data.frame()
t=1
for (o in 1:5){
 #o=1
 tst_set=CrossV$CrossValidation_list[[o]]
 X_trn=(X[-tst_set,])
 X_tst=(X[tst_set,])
 y_trn=(y[-tst_set])
 y_tst=(y[tst_set])

 ##################Inner cross-validation#######################
 nCVI=5  ####Number of folds for inner CV
 Hyperpar=data.frame()
 for (i in 1:nCVI){
  #i=1
  Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
  X_trII=X_trn[Sam_per,]
  y_trII=y_trn[Sam_per]

  #########Using the tuning_fun function for tuning
hyperparameters#####
  runs.sp<-tuning_run("Code_Tuning_With_Flags_00_4HL.R",flags=list
(dropout1= c(0,0.05,0.1),
                              units = c(33,67),
                              activation1=("relu"),
                              batchsize1=c(28,56),
                              Epoch1=c(1000),
                              learning_rate=c(0.001),
                         val_split=c(0.2)),sample=1,confirm =FALSE,
echo =F)
  ###### Ordering in the same way all grids
  runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_dropout1,
runs.sp$flag_batchsize1),]

  runs.sp$grid_length=1:nrow(runs.sp)
  Parameters=data.frame(grid_length=runs.sp$grid_length,
```

```
metric_val_mse=runs.sp$metric_val_mse,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_activation1=runs.
sp$flag_activation1)
  Hyperpar=rbind(Hyperpar,data.frame(Parameters))
 }
 Hyperpar %>%
  group_by(grid_length) %>%
  summarise(val_mse=mean(metric_val_mse),
       dropout1=mean(flag_dropout1),
       units=mean(flag_units),
       batchsize1=mean(flag_batchsize1),
       learning_rate=mean(flag_learning_rate),
       epochs=mean( epochs_completed)) %>%
  select(grid_length,val_mse,dropout1,units,batchsize1,
learning_rate, epochs) %>%
  mutate_if(is.numeric, funs(round(., 3))) %>%
  as.data.frame() -> Hyperpar_Opt
 ############Optimal hyperparameters############
 Min=min(Hyperpar_Opt$val_mse)
 pos_opt=which(Hyperpar_Opt$val_mse==Min)
 pos_opt=pos_opt[1]
 Optimal_Hyper=Hyperpar_Opt[pos_opt,]
 #####Selecting the best hyperparameters
 Drop_O=Optimal_Hyper$dropout1
 Epoch_O=round(Optimal_Hyper$epochs,0)
 Units_O=round(Optimal_Hyper$units,0)
 activation_O=unique(Hyperpar$flag_activation1)
 batchsize_O=round(Optimal_Hyper$batchsize1,0)
 lr_O=Optimal_Hyper$learning_rate

 print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
   if (epoch %% 20 == 0) cat("\n")
   cat(".")})

 ###########Refitting the model with the optimal values################
 model_Sec<-keras_model_sequential()
 model_Sec %>%
  layer_dense(units =Units_O , activation =activation_O, input_shape =
c(dim(X_trn)[2])) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =1)

 model_Sec %>% compile(
  loss = "mean_squared_error",
```

```
  optimizer = optimizer_adam(),
  metrics = c("mean_squared_error"))

 ModelFited <-model_Sec %>% fit(
  X_trn, y_trn,
  epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
  verbose=0,callbacks=list(print_dot_callback))

 ################Prediction of testing set ########################
 Yhat=model_Sec%>% predict(X_tst)
 y_p=Yhat
 y_p_tst=as.numeric(y_p)

 ###########Saving the predicctions of each outer testing
set#################
 results<-rbind(results, data.frame(Position=tst_set,
                         Environment=CrossV$Environments[tst_set],
                         Partition=o,
                         Units=Units_O,
                         Epochs=Epoch_O,
                         Observed=round(y[tst_set], digits),
                         #$response, digits),
                         Predicted=round(y_p_tst, digits),
                         Trait=Names_Traits[t]))
 cat("CV=",o,"\n")
}

results

#############Average of prediction performance####################
Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
Pred_Summary
```

# Appendix 5

MaizeToy example with five-fold CV with only one inner CV.

```
rm(list=ls())
library(lsa)
library(keras)
library(BMTME)
library(plyr)
library(tidyr)
library(dplyr)

###############Set seed for reproducible results###################
use_session_with_seed(45)
```

```
####################Loading the MaizeToy data sets##################
data("MaizeToy")
head(phenoMaizeToy)

#################Ordering the data #################################
phenoMaizeToy<-(phenoMaizeToy[order(phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames(phenoMaizeToy)=1:nrow(phenoMaizeToy)
head(phenoMaizeToy,8)

####################Design matrices################################
LG <- cholesky(genoMaizeToy)
ZG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line))
Z.G <- ZG %*%LG
Z.E <- model.matrix(~0 + as.factor(phenoMaizeToy$Env))
ZEG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker(diag(length(unique(phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky(G2)
Z.EG <- ZEG %*% LG2

###############Selecting the response variable#####################
Y <- as.matrix(phenoMaizeToy[, -c(1, 2)])

##########Training-testing sets using the BMTME package#############
pheno <- data.frame(GID = phenoMaizeToy[, 1], Env = phenoMaizeToy[, 2],
          Response = phenoMaizeToy[, 3])

CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)

###############Grid of hyperparameters#############################
Stage <- expand.grid(units_M =seq(33,67,33),epochs_M=1000, Dropout=c
(0.0,0.05,0.15, 0.25, 0.35))

################Function for averaging the predictions############
summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
 results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(MSE = mean((Predicted-Observed)^2),
      MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed)))) %>%
  select(Environment, Trait, Partition, MSE, MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Environment, Trait) %>%
  summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
     SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>
%
  select(Environment, Trait, MSE, SE_MSE, MAAPE, SE_MAAPE) %>%
```

```
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
        complete = presum,
        extended = {
         finalSum$Partition <- 'All'
         presum$Partition <- as.character(presum$Partition)
         presum$SE_MSE <- NA
         presum$SE_MAAPE <- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

################Final X and y for fitting the model###################
y=(phenoMaizeToy[, 3])
X=cbind(Z.E,Z.G)

#####################Outer cross-validation#######################
digits=4
Names_Traits=colnames(Y)
results=data.frame()
t=1
for (o in 1:5){
tst_set=CrossV$CrossValidation_list[[o]]
X_trn=(X[-tst_set,])
X_tst=(X[tst_set,])
y_trn=(y[-tst_set])
y_tst=(y[tst_set])

nCVI=1 ####Number of folds for inner CV
i=1
############Matrices for saving the output of inner CV################
Tab_pred_MSE=matrix(NA,ncol=length(Stage[,1]), nrow= nCVI)
Tab_pred_Epoch=matrix(NA,ncol=length(Stage[,1]), nrow= nCVI)
Tab_pred_Units=matrix(NA,ncol=length(Stage[,1]), nrow= nCVI)
Tab_pred_Drop=matrix(NA,ncol=length(Stage[,1]), nrow= nCVI)
X_trI=X_trn
y_trI=y_trn

for (stage in seq_len(dim(Stage)[1])) {
 X_trII=X_trI
 y_trII=y_trI
 units_M <- Stage[stage, 1]
 epochs_M <- Stage[stage, 2]
 Drop_per= Stage[stage, 3]

build_model<-function() {
model <- keras_model_sequential()
```

```
model %>%
 layer_dense(units =units_M, activation = "relu", input_shape = c(dim
(X_trII)[2])) %>%
 layer_dropout(rate=Drop_per) %>%
 layer_dense(units=1)

model %>% compile(
 loss = "mse",
 optimizer =optimizer_adam(),
 metrics = c("mse"))
model}

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")
 })

################Fitting the model with early stopping############
early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =50)

##############Fit of the model for each values of the grid############
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
 X_trII, y_trII,
 epochs =epochs_M, batch_size =72,
 ###shuffled=F,
 validation_split = 0.2,
 verbose=0,callbacks = list(early_stop, print_dot_callback))
##############Saving the output of each hyperparameter##############
No.Epoch_Min=length(model_fit_Final$metrics$val_mse)
Min_MSE=model_fit_Final$metrics$val_mean_squared_error[No.
Epoch_Min]

Tab_pred_MSE[i,stage]=Min_MSE
Tab_pred_Units[i,stage]=units_M
Tab_pred_Epoch[i,stage]=No.Epoch_Min[1]
Tab_pred_Drop[i,stage]=Drop_per
}

################Selecting the optimal hyperparameters############
Median_MSE_Inner=apply(Tab_pred_MSE,2,median)
Units_Inner=apply(Tab_pred_Units,2,max)
Drop_Inner=apply(Tab_pred_Drop,2,max)
Epoch_Inner=apply(Tab_pred_Epoch,2,median)
Pos_Min_MSE=which(Median_MSE_Inner==min(Median_MSE_Inner))
Units_O=Units_Inner[Pos_Min_MSE]
```

```
Epoch_O= Epoch_Inner[Pos_Min_MSE]
Drop_O= Drop_Inner[Pos_Min_MSE]

##############Refitting the model with the optimal values############
model_Sec<-keras_model_sequential()
model_Sec %>%
 layer_dense(units =Units_O , activation ="relu", input_shape = c(dim
(X_trn)[2])) %>%
 layer_dropout(rate =Drop_O) %>%
 layer_dense(units =1)

model_Sec %>% compile(
 loss = "mean_squared_error",
 optimizer = optimizer_adam(),
 metrics = c("mean_squared_error"))

ModelFited <-model_Sec %>% fit(
 X_trn, y_trn,
 epochs=Epoch_O, batch_size =30,
####### validation_split=0.2,early_stop,
 verbose=0, callbacks=list(print_dot_callback))

################Prediction of testing set ##########################
Yhat=model_Sec %>% predict(X_tst)
y_p=Yhat
y_p_tst=as.numeric(y_p)

#########Saving the predicctions of each outer testing set###########
results<-rbind(results, data.frame(Position=tst_set,
                         Environment=CrossV$Environments[tst_set],
                         Partition=o,
                         Units=Units_O,
                         Epochs=Epoch_O,
                         Drop_Out=Drop_O,
                         Observed=round(y[tst_set], digits),
                         #$response, digits),
                         Predicted=round(y_p_tst, digits),
                         Trait=Names_Traits[t]))
cat("CV=",o,"\n")
}
results

############Average of prediction performance####################
Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
Pred_Summary
```

## References

Allaire JJ (2018) Tfruns: training run tools for 'tensorflow'. https://CRAN.R-project.org/package=tfruns
Allaire JJ, Chollet F (2019) Keras: R interface to Keras. https://CRAN.R-project.org/package=keras
Baysolow II T (2017) Introduction to deep learning using R. A step-by step guide to learning and implementing Deep learning models using R. Apress
Chollet F, Allaire JJ (2017) Deep learning with R. Manning Publications, Manning Early Access Program (MEA), 1st edn
Cybenko G (1989) Approximations by superpositions of sigmoidal functions. Math Control Signal Syst 2:303–314
Géron A (2019) Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. Concepts, tools, and techniques to build intelligent systems. O'Reilly Media, Inc., California
Hecht-Nielsen R (1987) Counterpropagation networks. Appl Opt 26:4979–4984
Lantz B (2015) Machine learning with R, 2nd edn. Packt Publishing Ltd, Birmingham
Lippmann R (1987) An introduction to computing with neural nets. IEEE ASSP Mag 4(2):4–22
Patterson J, Gibson A (2017) Deep learning: a practitioner's approach. O'Reilly Media
Samarasinghe S (2007) Neural networks for applied sciences and engineering. From fundamentals to complex pattern recognition. Auerbach Publications, Taylor & Francis Group, Boca Raton, New York

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 12
# Artificial Neural Networks and Deep Learning for Genomic Prediction of Binary, Ordinal, and Mixed Outcomes

## 12.1 Training DNN with Binary Outcomes

Before starting with the examples, we explain in general terms the process to follow to train DNN for binary outcomes. When training binary outcomes, it is important to denote the values of the response variable as 0 and 1, where 0 denotes the absence of the disease and 1 its presence; any other two types of interest should be denoted as 0 and 1. Below we provide some key elements to train this type of DNNs more efficiently:

(a) *Define the DNN model in Keras.* As in the previous chapters, we again focus only on fully connected neural networks that consist of stacking fully connected networks to all neurons (units). We suggest using the RELU activation function or some of the following activation functions (leaky RELU, tanh, exponential linear unit, etc.) for hidden layers, but for the output layer we suggest using a single-unit layer with the sigmoid activation function to guarantee that the output is a probability between 0 and 1. Also, the first layer requires the input shape (features) information; however, this is not required for the following layers since they automatically infer the shape from the previous layer.

    The construction of a DNN in Keras for binary outcomes again needs to start by initializing a sequential model using the keras_model_sequential() function which allows implementing a series of layer functions that create a linear stacking of layers. The summary () can also be used to print a summary of our DNN model. The number of neurons in the output layer is 1 since we are dealing with a univariate binary outcome with two categories in the outcome variable.

(b) *Configuring and compiling the model.* At this stage of the training process, the loss function, the optimizer, and the metric for evaluating the prediction performance should be defined. Regarding the loss function, most of the time binary_crossentropy is suggested for binary outcomes, while categorical_crossentropy is suggested for categorical or ordinal data. As it was studied in the previous chapter, the mean_squared_error loss is the most popular

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_12

for continuous outcomes. However, it is also possible to use the mean_squared_error loss for binary and categorical outcomes, but binary_crossentropy and categorical_crossentropy are the best choices when we are dealing with DNN models that output probabilities because they measure the distance between probability distributions, that is, between the ground truth distribution and the predictions obtained. As was mentioned in the previous chapter, the optimizer plays a really important role when updating model parameters (weights and biases). There is no specific optimizer for each type of response variable, and as we studied in the previous chapter, there are at least seven optimizers available in the Keras library. However, we usually use the Adam optimizer since it performs well in many cases. Finally, regarding the type of metric for evaluating the prediction performance for binary and categorical data, we use the accuracy metric which measures the proportion of cases that are correctly classified.

(c) *Fitting the model*. At this stage, we need to specify the number of epochs (i.e., the number of times the algorithm uses the entire training data set) and the batch size (size of the sample to be passed through the entire algorithm in each epoch) because if the training data consist of 1000 observations and we use a batch size = 50, we will need 20 iterations per epoch. Here, we should specify the validation split when you are in the tuning process (the value of the validation split is between 0 and 1) or specify the validation data set that should be used. For example, if you specified a validation_split = 0.3, this means that 30% of the original training data should be used as the validation set, and the remaining 70% of the observations will be used for training the model; the prediction performance of the model is evaluated with the validation set. Also, if you want to use the early stopping method, this should be specified in callbacks exactly as was done in the previous chapter for continuous outcomes.

(d) *Evaluating the prediction performance*. For binary outcomes, we suggest using the predict_classes() function that requires the information of the independent variables of the testing set as input for which the predictions are required. The predict_classes() function gives binary results (0 or 1) as output. However, practitioners can also use the predict() function, which provides probabilities for each category as outputs that need to be converted to 0 and 1 using a threshold value, for example, observations with probabilities larger or equal to 0.5 should be classified as 1 and observations with probabilities smaller than 0.5 should be classified as 0. Next, we provide the first illustrative example for training DNN with binary outcomes.

**Example 12.1**

**Binary outcomes**. This toy data set is called EYT and is composed of four environments (Bed5IR, EHT, Flat5IR, and LHT), 40 lines in each environment, and contains four traits (DTHD, DTMT, GY, and Height). Traits DTHD and DTMT are ordinal traits, GY is a continuous trait, and Height is a binary trait. This data set

contains a genomic relationship matrix of $40 \times 40$ that corresponds to the similarity between lines.

The first eight observations of this data set are given below.

```
> head(Data_Pheno,8)
       GID    Env DTHD DTMT       GY Height
1 GID6569128 Bed5IR    1    1 6.119272      0
2 GID6688880 Bed5IR    2    2 5.855879      0
3 GID6688916 Bed5IR    2    2 6.434748      0
4 GID6688933 Bed5IR    2    2 6.350670      0
5 GID6688934 Bed5IR    1    2 6.523289      0
6 GID6688949 Bed5IR    1    2 5.984599      0
7 GID6689407 Bed5IR    1    2 6.436980      0
8 GID6689482 Bed5IR    3    3 6.052307      1
```

We can see that the ordinal traits (DTHD and DTMT) have three levels denoted as 1, 2, and 3, and the binary trait (Height) has two levels denoted as 0 and 1. We will create flags for the tuning process. The following code is used to create the flags; it is called Code_Tuning_With_Flags_Bin.R.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
 flag_numeric("val_split",0.2))

####b) Defining the DNN model
build_model<-function() {
model <- keras_model_sequential()
model %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1,
input_shape = c(dim(X_trII)[2])) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units=1, activation ="sigmoid")

#####c) Compiling the DNN model
model %>% compile(
 loss = "binary_crossentropy",
 optimizer =optimizer_adam(lr=FLAGS$learning_rate),
```

```
 metrics =c('accuracy'))
model}

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")})

early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =50)

###########d) Fitting the DNN model#################
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
 X_trII, y_trII,
 epochs =FLAGS$Epoch1, batch_size =FLAGS$batchsize1,
 shuffled=F,
 validation_split =FLAGS$val_split,
 verbose=0,callbacks = list(early_stop,print_dot_callback))
```

In a) are given the default flag values for % of dropout, number of units, activation function for hidden layers, batch size, number of epochs, learning rate, and validation split. In b) the DNN model is defined and the flag parameters are incorporated within our DNN model. It should be pointed out that the RELU activation function is used for the hidden layers, but the sigmoid activation function is used for the output layer to guarantee a probability as output between 0 and 1. Only one unit is specified for the output layer since we are interested in predicting only a univariate binary outcome. It is clear that our DNN model contains four hidden layers since the layer_dense() function was specified five times, but the last one corresponds to the output layer.

The model is compiled in part c) of the code, and the important things to note are that (a) now the loss function is binary_crossentropy, which is appropriate for binary response variables, (b) the optimizer specified was the Adam optimizer, optimizer_adam(), which is not specific for binary data, and (c) the metrics specified for evaluating the prediction performance is the accuracy that measures the proportion of correctly classified cases. In part d) the model is fitted using the number of epochs, the batch size, and the validation split specified in the flags (part a). In this case, the fitting process was done using the early stopping method.

Then, the above codes named "Code_Tuning_With_Flags_Bin.R" are called in the code given in Appendix 1. The code given in Appendix 1 executes the grid search using the library tfruns (Allaire 2018) with the tuning_run() function. The implemented grid search is shown below:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_Bin_4HL.R",runs_dir =
'_tuningE1',
```

```
flags=list(dropout1= c(0,0.05),
     units = c(67,100),
     activation1=("relu"),
     batchsize1=c(28),
     Epoch1=c(1000),
     learning_rate=c(Learn_val[e]),
     val_split=c(0.2)),sample=1,confirm =FALSE,echo =F)
```

The grid search is composed of only four combinations of hyperparameters that resulted from using two values of dropout (0, 0.05) and two units (67, 100). We used this small grid search because it is often not possible to do a full cartesian grid search with many values of each hyperparameter due to time and computational constraints. The code given in Appendix 1 was run five times, and each time it was run for a specific value of learning rate. It is important to note that the prediction performance is reported using not only the PCCC but also the Kappa coefficient, the sensitivity, and the specificity. Table 12.1 indicates that the best prediction performance was obtained using a learning rate (learn_val) of 0.01 across environments.

**Table 12.1** Prediction performance for binary outcomes for five different values of learning rate using four hidden layers with the RELU activation function with five outer fold cross-validations and five inner fold cross-validations

| learn_val | Env | PCCC | SE_PCCC | Kappa | SE_Kappa | Sensitivity | Specificity |
|---|---|---|---|---|---|---|---|
| 0.001 | Bed5IR | 0.724 | 0.091 | 0.419 | 0.203 | 0.683 | 0.703 |
| 0.001 | EHT | 0.900 | 0.045 | 0.800 | 0.090 | 0.883 | 0.927 |
| 0.001 | Flat5IR | 0.654 | 0.043 | 0.191 | 0.122 | 0.704 | 0.500 |
| 0.001 | LHT | 0.513 | 0.058 | 0.010 | 0.127 | 0.523 | 0.510 |
| **0.01** | **Bed5IR** | **0.792** | **0.072** | **0.590** | **0.138** | **0.827** | **0.767** |
| **0.01** | **EHT** | **0.900** | **0.063** | **0.779** | **0.139** | **0.943** | **0.883** |
| **0.01** | **Flat5IR** | **0.668** | **0.033** | **0.323** | **0.073** | **0.780** | **0.600** |
| **0.01** | **LHT** | **0.584** | **0.097** | **0.234** | **0.137** | **0.650** | **0.607** |
| 0.1 | Bed5IR | 0.577 | 0.047 | 0.184 | 0.088 | 0.567 | 0.633 |
| 0.1 | EHT | 0.721 | 0.101 | 0.481 | 0.167 | 0.653 | 0.860 |
| 0.1 | Flat5IR | 0.604 | 0.044 | 0.146 | 0.117 | 0.860 | 0.372 |
| 0.1 | LHT | 0.576 | 0.070 | 0.189 | 0.131 | 0.750 | 0.574 |
| 0.5 | Bed5IR | 0.445 | 0.070 | −0.013 | 0.013 | 0.429 | 0.470 |
| 0.5 | EHT | 0.484 | 0.094 | 0.164 | 0.109 | 0.513 | 0.717 |
| 0.5 | Flat5IR | 0.484 | 0.069 | −0.032 | 0.032 | 0.608 | 0.220 |
| 0.5 | LHT | 0.392 | 0.089 | 0.080 | 0.080 | 0.519 | 0.413 |
| 1 | Bed5IR | 0.413 | 0.042 | 0.025 | 0.025 | 0.354 | 0.565 |
| 1 | EHT | 0.473 | 0.082 | 0.000 | 0.000 | 0.325 | 0.571 |
| 1 | Flat5IR | 0.462 | 0.078 | 0.000 | 0.000 | 0.625 | 0.353 |
| 1 | LHT | 0.402 | 0.092 | 0.000 | 0.000 | 0.374 | 0.421 |

## 12.2 Training DNN with Categorical (Ordinal) Outcomes

When training DNN for categorical or ordinal outcomes, the C levels of the response variable are 0, 1, 2, . . ., $C - 1$. For example, if the categorical or ordinal response variable has three levels (no infection, middle level of infection, and total level of infection), they should be denoted as 0, 1, and 2, where 0 denotes no infection, 1 middle level of infection, and 2 total level of infection. Another example is that assume you are interested in training a machine for classification with orange, mandarin, tangerine, and lemon as outcomes; you can denote orange with 0, mandarin with 1, tangerine with 2, and lemon with 3. Of course you can choose different values for each fruit, but because there are four categories, you will use 0, 1, 2, and 3 to denote the four fruits even though this is a nominal variable. Next, we provide some key elements to train this type of DNN models more efficiently.

*Define the DNN model in Keras.* The training process is equal to the training of binary response variables, except that in the output layer we suggest using the softmax activation function with a number of units equal to the number of categories; this guarantees that the output of each category is a probability between 0 and 1, and that the sum of these (all categories) probabilities is 1. Before starting the training process, you need to convert to dummy variables the categorical (or ordinal) response variable using the to_categorical() function that needs, as input, the vector of the categorical response variable and the number of classes that the response has. This way of coding the categorical and ordinal responses is called one-hot encoding or categorical encoding. It consists of embedding each level (label) of the categorical response variable as an all-zero vector with 1 in the place of the label index. For example, suppose that your response variable contains the following values: $y = (0, 2, 4, 1, 3, 0, 1, 3, 4, 2)$. Then the vector of response variable is transformed to $yf = to\ _\ categorical(y, 5)$ that produces the following result:

| *y* | *yf* | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 |

This means that the dependent variable is no longer a vector, because it is a matrix of zeros and ones; for this reason, the number of units required for the output layer is equal to the number of classes. In this example, the number of units required for the output layer should be five.

(a) *Configuring and compiling the model*. The only difference when compiling binary response variables and ordinal (or categorical) outcomes is that now using the categorical_crossentroy loss as the loss function is recommended.
(b) *Fitting the model*. Everything that was explained for binary data also applies to ordinal and categorical outcomes.
(c) *Evaluating the prediction performance*. For ordinal and categorical outcomes, we suggest using the predict_classes() function because it produces, as output, values of the categorical or ordinal data in the scale of the response variable, that is, 0, 1, . . ., $C - 1$. However, you can also use the predict() function, which will provide you with probabilities for each category and the sum of all of them is equal to 1. These probabilities need to be converted to the original response variable (0, 1, . . ., $C - 1$). This conversion can be done by assigning each observation to the category with the largest probability (Allaire and Chollet 2019). Next, we provide one illustrative example for a DNN with an ordinal outcome.

**Example 12.2**
**Ordinal outcome**. This example uses the same data as Example 12.1 (Toy_EYT data set), but now we use the ordinal trait DTHD as the response variable. For the tuning process, we first created the flags which are given next and should be placed in a file called Code_Tuning_With_Flags_Ordinal_4HL2.R, as it is called in Appendix 2.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
 flag_numeric("val_split",0.2))

####b) Defining the DNN model
build_model<-function() {
model <- keras_model_sequential()
model %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1,
input_shape = c(dim(X_trII)[2])) %>%
 layer_batch_normalization() %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_batch_normalization() %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_batch_normalization() %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
```

```
 layer_batch_normalization() %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units=3, activation ="softmax")

#####c) Compiling the DNN model
model %>% compile(
 loss = "categorical_crossentropy",
 optimizer =optimizer_adam(lr=FLAGS$learning_rate),
 metrics =c('accuracy'))
model}

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")})

early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =50)

###########d) Fitting the DNN model#################
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
 X_trII, y_trII,
 epochs =FLAGS$Epoch1, batch_size =FLAGS$batchsize1,
 shuffled=F,
 validation_split =FLAGS$val_split,
 verbose=0,callbacks = list(early_stop,print_dot_callback))
```

In a) are given the default flag values for some hyperparameters; the DNN is defined in b) and is very similar to the definition of the DNN for binary outcomes, except that in the output layer the softmax activation function is used, which is appropriate for categorical or ordinal data, and now instead of one unit, three are used in the output layer since this is the number of classes of the DTHD ordinal response variable. Another important difference in the definition of the DNN model is that now we used the layer_batch_normalization() function just after specifying each hidden layer. The layer_batch_normalization() function is used to help with a problem called internal covariate shift that consists of changing the distribution of network activations due to the change in network parameters during training. Therefore, the layer_batch_normalization() function improves the training process by reducing the internal covariate shift by fixing the distribution of the layer inputs x as the training progresses (Ioffe and Szegedy 2015; LeCun et al. 1998; Wiesler and Ney 2011), and the internal covariate shift is reduced by linearly transforming the input to have zero means and unit variances and decorrelating the input information. This process is done in the input of each layer to fix the distributions of inputs that would remove the effects of the internal covariate shift (Ioffe and Szegedy 2015).

In part c) of the code, the DNN model is compiled; this is the same as the compilation process of binary outcomes, except that now a categorical_crossentropy loss function should be used because the response variable is ordinal or categorical. The fitting process, part d), is the same as that for binary outcomes.

Before using these flags, the response variable was converted to dummy variables using the to_categorical() function. Next, we show the first eight observations of the testing set of the first partition used in the code given in Appendix 2.

```
> y_tst= to_categorical(y[tst_set],nclas)
> cbind(y[tst_set],y_tst)
     [,1] [,2] [,3] [,4]
 [1,]  2    0    0    1
 [2,]  0    1    0    0
 [3,]  2    0    0    1
 [4,]  2    0    0    1
 [5,]  1    0    1    0
 [6,]  0    1    0    0
 [7,]  0    1    0    0
 [8,]  2    0    0    1
```

Here we observe that the first column corresponds to the original ordinal score of the response variable with levels 0, 1, and 2, that is, nclas = 3, while the remaining three columns are the three dummy variables created using the to_categorical() function since the original response variable has three types. From the output produced using the to_categorical() function, we can see that the first observation in the last column is a 1, while columns 2 and 3 have values of 0, since the original categorical response variable is 2. In the second observation, we can see that since the original categorical score is 0, the 1 appears in the second column, and values of 0 in the remaining columns. In the fifth observation, we can see that the original categorical score is 1; for this reason, in the third column, there is a 1 and in the remaining columns there is a value of 0.

The code given above is called Code_Tuning_With_Flags_Ordinal_4HL2.R in the code given in Appendix 2. The code given in Appendix 2 does the grid search using the library tfruns (Allaire 2018) and the tuning_run() function. The implemented grid search is shown below:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_Ordinal_4HL2.R",
runs_dir = '_tuningE1',
          flags=list(dropout1= c(0,0.05),
               units = c(67,100),
               activation1=("relu"),
               batchsize1=c(28),
               Epoch1=c(1000),
               learning_rate=c(Learn_val[e]),
               val_split=c(0.2)),sample=1,confirm =FALSE,echo =F)
```

**Table 12.2** Prediction performance for ordinal outcomes for different values of learning rate with four hidden layers

| Learn_val | Env | PCCC | SE_PCCC | Kappa | SE_Kappa | Sensitivity | Specificity |
|---|---|---|---|---|---|---|---|
| 0.005 | Bed5IR | 0.692 | 0.085 | 0.535 | 0.109 | 0.783 | 0.625 |
| 0.005 | EHT | 0.693 | 0.075 | 0.439 | 0.082 | 0.542 | 0.167 |
| 0.005 | Flat5IR | 0.757 | 0.071 | 0.617 | 0.101 | 0.854 | 0.542 |
| 0.005 | LHT | 0.702 | 0.056 | 0.517 | 0.084 | 0.810 | 0.333 |
| **0.01** | **Bed5IR** | **0.734** | **0.097** | **0.585** | **0.145** | **0.750** | **0.556** |
| **0.01** | **EHT** | **0.743** | **0.075** | **0.537** | **0.139** | **0.660** | **0.250** |
| **0.01** | **Flat5IR** | **0.702** | **0.089** | **0.529** | **0.137** | **0.850** | **0.567** |
| **0.01** | **LHT** | **0.718** | **0.055** | **0.519** | **0.084** | **0.658** | **0.750** |
| 0.015 | Bed5IR | 0.691 | 0.111 | 0.502 | 0.169 | 0.710 | 0.333 |
| 0.015 | EHT | 0.689 | 0.053 | 0.462 | 0.103 | 0.625 | 0.167 |
| 0.015 | Flat5IR | 0.685 | 0.087 | 0.523 | 0.131 | 0.767 | 0.367 |
| 0.015 | LHT | 0.725 | 0.097 | 0.544 | 0.158 | 0.906 | 0.313 |
| 0.03 | Bed5IR | 0.684 | 0.087 | 0.500 | 0.118 | 0.883 | 0.542 |
| 0.03 | EHT | 0.733 | 0.037 | 0.373 | 0.167 | 0.900 | 0.167 |
| 0.03 | Flat5IR | 0.633 | 0.091 | 0.417 | 0.132 | 0.883 | 0.400 |
| 0.03 | LHT | 0.676 | 0.126 | 0.494 | 0.184 | 0.833 | 0.250 |
| 0.06 | Bed5IR | 0.720 | 0.097 | 0.503 | 0.177 | 0.704 | 0.500 |
| 0.06 | EHT | 0.713 | 0.033 | 0.425 | 0.078 | 0.733 | 0.333 |
| 0.06 | Flat5IR | 0.695 | 0.062 | 0.519 | 0.091 | 0.817 | 0.250 |
| 0.06 | LHT | 0.658 | 0.068 | 0.442 | 0.104 | 0.761 | 0.125 |

The grid search was used (sample = 1) for the tuning process and, for the four hyperparameter combinations (two values of dropout and two values of units), were evaluated.

Table 12.2 gives the results of implementing the code given in Appendix 2 for five different values of learning rate (0.005, 0.01, 0.015, 0.03, and 0.06), where the best predictions were obtained with a learning rate value of 0.01 across environments. These results give evidence that the prediction performance depends considerably on the value of the hyperparameter called learning rate.

## 12.3 Training DNN with Count Outcomes

Remember that count data (0, 1, 2, . . .) are usually modeled with Poisson regression or negative binomial regression in the statistical world. In the world of DNN, only the Poisson DNN model has been available until now in Keras and its key elements imitate those of the generalized linear models of the statistical world. For this reason, its construction in Keras uses as loss function the minus log-likelihood of a Poisson distribution; for the output layer, you can use the exponential activation function (inverse of log link in generalized linear models) that is available in Keras, which

guarantees only positive outcomes. Also a RELU activation function can be used to guarantee a positive outcome.

In general, the definition of a DNN model for count outcomes in Keras is very similar to what we have studied before for continuous, binary, and categorical data for univariate responses. If we are interested in predicting only one response variable, we only need to specify one unit in the output layer, but if we are interested in predicting five count response variables, we need to specify a unit for each response we wish to predict. Also, the process of adding the hidden layers is exactly the same as was done for continuous, binary, and categorical (or ordinal) data with the same activation functions, for example, RELU for all the hidden layers, since the fitting process of the model is exactly the same as the fitting process of continuous, binary, and ordinal univariate outcomes. However, some of the key differences are in the compilation and prediction process. In the compilation process, we need to specify the “poisson” loss function that was created as the minus log-likelihood of the Poisson distribution, and for the metric, we can still use the mean squared error metric; however, for the prediction process, we should use the predict() function that will always produce positive values only if we specify an appropriate activation function in the output layer like the exponential activation function. Next, we provide an illustrative example for training DNN with count outcomes.

**Example 12.3**
**Count data.** This toy data set contains 115 lines, evaluated in three environments (Batan2012, Batan2014, and Chunchi2014) and in each environment two blocks were created. The total number of observations of this data set is 649. The data set is denoted as Data_Count_Toy.RData. The count response variable has a minimum value of 0 and a maximum value of 17.

Modeling and predicting count data is not only important in plant breeding, but also very common in areas such as health, finance, social science, etc. Generalized linear models have been widely used for modeling count response variables, but many times fail to capture complex data patterns. For this reason, nonlinear Poisson regressions under the umbrella of deep artificial neural networks are of paramount importance for modeling count data and improving the prediction accuracy. The details for implementing DNN models for count data are given below.

The tuning process was done by creating flags which are given below; they should be placed in a file named: Code_Tuning_With_Flags_Count_Lasso.R, that is used in Appendix 3.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.0),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
```

```
 flag_numeric("val_split",0.2),
 flag_numeric("Lasso_par",0.001))

####b) Defining the DNN model
build_model<-function() {
model <- keras_model_sequential()
model %>%
 layer_dense(units =FLAGS$units, kernel_regularizer=regularizer_l1
(FLAGS$Lasso_par),activation =FLAGS$activation1, input_shape = c(dim
(X_trII)[2])) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, kernel_regularizer=regularizer_l1
(FLAGS$Lasso_par),activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, kernel_regularizer=regularizer_l1
(FLAGS$Lasso_par),activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, kernel_regularizer=regularizer_l1
(FLAGS$Lasso_par),activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units=1, activation ="exponential")

#####c) Compiling the DNN model
model %>% compile(
 loss = "poisson",
 optimizer =optimizer_adam(lr=FLAGS$learning_rate),
 metrics =c('mse'))
model}

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")})

early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =30)

###########d) Fitting the DNN model################
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
 X_trII, y_trII,
 epochs =FLAGS$Epoch1, batch_size =FLAGS$batchsize1,
 shuffled=T,
 validation_split =FLAGS$val_split,
 verbose=0,callbacks = list(early_stop,print_dot_callback))
```

Again, in part a) are defined the default flags, in part b) the DNN for count data is defined, where the significant difference is that the exponential activation function is

**Table 12.3** Prediction performance of count data for different values of regularization with four hidden layers and Ridge regularization

| relularization value | Environment | Trait | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| 0.005 | Batan2012 | Count | 1.787 | 0.2466 | 0.8752 | 0.0276 |
| 0.005 | Batan2014 | Count | 1.7613 | 0.1572 | 0.8864 | 0.0171 |
| 0.005 | Chunchi2014 | Count | 15.0395 | 2.2348 | 0.6663 | 0.0122 |
| 0.01 | Batan2012 | Count | 2.5557 | 0.1534 | 0.9026 | 0.0467 |
| 0.01 | Batan2014 | Count | 2.2408 | 0.0878 | 0.9093 | 0.0109 |
| 0.01 | Chunchi2014 | Count | 13.9118 | 2.176 | 0.5964 | 0.0227 |
| 0.015 | Batan2012 | Count | 1.8293 | 0.3306 | 0.8662 | 0.0378 |
| 0.015 | Batan2014 | Count | 1.7998 | 0.2559 | 0.8874 | 0.0226 |
| 0.015 | Chunchi2014 | Count | 15.1706 | 1.8295 | 0.6334 | 0.0153 |
| 0.03 | Batan2012 | Count | 2.2634 | 0.2003 | 0.9072 | 0.0295 |
| 0.03 | Batan2014 | Count | 2.2277 | 0.2537 | 0.9055 | 0.0174 |
| 0.03 | Chunchi2014 | Count | 13.0655 | 1.7221 | 0.5856 | 0.0199 |
| 0.06 | Batan2012 | Count | 2.2384 | 0.2008 | 0.8975 | 0.016 |
| 0.06 | Batan2014 | Count | 2.0308 | 0.2055 | 0.9078 | 0.0275 |
| 0.06 | Chunchi2014 | Count | 13.8051 | 1.3726 | 0.6511 | 0.0391 |

used for the output layer; we should point out that in each hidden layer, the Lasso (L1) regularization is also used in addition to dropout. In part c) the relevant parts are (1) the specification of the Poisson loss function and (2) the use of the mean squared error as a metric for evaluating the prediction performance, while the fitting process is exactly the same as was done for continuous, binary, and categorial or ordinal outcomes.

Then, the flags above are called in Appendix 3 that used the following grid search:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_Count_Lasso.R",
             runs_dir = '_tuningE1',
             flags=list(dropout1= c(0),
             units = c(67,150),
             activation1=("relu"),
             batchsize1=c(28),
             Epoch1=c(1000),
             learning_rate=c(Learn_val[e]),
             val_split=c(0.2),
           Lasso_par=c(0.001,0.01)), sample=1,confirm =FAL SE,echo =F)
```

From the above random grid search, we observe that four is the total number of hyperparameters that form the grid (two regularization parameters and two units) and should be evaluated. The code given in Appendix 3 was used to get the results given in Table 12.3, as well as for evaluating the prediction performance with five regularization values (0.005, 0.01, 0.015, 0.03, and 0.06) and with five outer fold and five inner fold cross-validations and with four hidden layers.

## 12.4 Training DNN with Multivariate Outcomes

Training models with multivariate outcomes are very important for plant breeders since they are interested in predicting more than one trait. In the context of plant breeding, multivariate models for the prediction of more than one trait simultaneously are called multi-trait models. There is evidence that multi-trait models capture the complex relationships between traits more efficiently and, for this reason, many times they improve the prediction performance when compared with univariate models. Statistical multi-trait models capture the correlation between traits and also the correlation between lines.

There is evidence, and not only in genomic selection, that the larger the correlation between traits, the better the prediction performance of multi-trait analysis (Jia and Jannink 2012; Jiang et al. 2015). Authors like He et al. (2016) and Schulthess et al. (2017) found a significant improvement of multi-trait analysis with regard to univariate analysis, while Calus and Veerkamp (2011), Montesinos-López et al. (2016), Montesinos-López et al. (2018a, b), and Montesinos-López et al. (2019) found modest improvement of multivariate analysis when compared to univariate analysis. Also in the context of multi-trait models, it helps to clarify the relationship and the effect of each studied independent variable on the dependent multivariate variables (Castro et al. 2013; Huang et al. 2015).

Despite the positive advantages of statistical multi-trait models mostly for continuous outcomes, it has not been possible to develop efficient models for other types of multivariate response variables (binary, categorical, and count) and efficient models for mixed outcomes (continuous, binary, categorical, and count) are still lacking. There have been some developments in the statistical literature, but most of them are not efficient for large data sets. However, as shown in two publications by Montesinos-López et al. (2018b, c) and Montesinos-López et al. (2019), the deep learning methodology has the power to efficiently implement univariate and multivariate models for each type of response variables, and even for mixed outcomes (Chollet and Alliare 2017). For this reason, in this section, we will show how to implement multi-trait analysis for continuous outcomes, multi-trait binary outcomes, multi-trait categorical outcomes, multi-trait count outcomes, and multi-trait mixed outcomes with a combination of at least two types of response variables (continuous and binary; continuous and categorical; continuous and count; categorical and count; etc.) and even with four types of response variables for continuous, binary, categorical, and count outcomes. Next, we will illustrate the DNN training process with multi-trait outcomes with all traits as continuous.

### *12.4.1 DNN with Multivariate Continuous Outcomes*

The data set used for illustrating how to train multivariate continuous outcomes is called MaizeToy data set. This data set contains 30 lines that were measured in three

environments (EBU, KAT, and KTI); for this reason, the total number of observations is 90. Also, three continuous traits were measured for each observation, and these traits were Yield, ASI, and PH. The genomic information is contained in the genomic relationship matrix denoted as genoMaizeToy.R.

Next, we provide the default flags for training continuous outcomes. These flags should be placed in the R (R Core Team 2019) code called Code_Tuning_With_Flags_MT_normal.R.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
 flag_numeric("val_split",0.2))

####b) Defining the multi-trait DNN model
input <- layer_input(shape=dim(X_trII)[2],name="covars")

# add hidden layers
base_model <- input %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units=1, name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units= 1, name="response_2")

# add output 3
yhat3 <- base_model %>%
 layer_dense(units= 1,name="response_3")

#c) Compiling the multi-trait model
model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3)) %>% compile(optimizer =optimizer_adam
(lr=FLAGS$learning_rate),
    loss=list(response_1="mse",response_2="mse", response_3="mse"),
     metrics=list(response_1="mse",response_2="mse",
response_3="mse"),
     loss_weights=list(response_1=0.99,response_2=1.8,
response_3=0.069))
```

```
print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")
 })

early_stop <- callback_early_stopping(monitor = c("val_loss"),
mode='min', patience =50)

# d) Fitting multi-trait model
model_fit <- model %>%
 fit(x=X_trII,
   y=list(response_1=y_trII[,1],response_2=y_trII[,2],
response_3=y_trII[,3]),
   epochs=FLAGS$Epoch1,
   batch_size =FLAGS$batchsize1,
   validation_split=FLAGS$val_split,
   verbose=0, callbacks = list(early_stop,print_dot_callback))
```

In part a) the default flags which are defined exactly as for univariate outcomes are defined. In part b) the multi-trait DNN model is defined, with layer_input(), the dimension (number of independent variables) of the input information is then provided and the hidden layers are added. In this case, only three hidden layers were specified and each layer had dropout that was added with layer_dropout(). Then three output layers were added that correspond to each of the three traits of the mult-trait DNN model. The three output layers use one unit and the linear activation function (not necessary to specify which) since the three traits are assumed to be continuous. Next, in part c), the compilation process is done; here, as in univariate DNN models, we need to specify the loss function, the optimizer, and the metrics used to evaluate the prediction performance. The specified optimizer is exactly the same as in the univariate DNN models. However, for the specification of the loss function and metrics we need to specify a loss function and metrics for each trait (outcome), that need to be in agreement with the type of response of each trait. Since in this example we have three continuous traits, we specified as a loss function and metric for each trait the mean_squared error (mse); however, for traits with different types of outcome, the practitioner should specify a loss function and metric appropriate for each type of trait. Two differences in the compilation process of multi-trait DNN models with regard to univariate DNN models are found. The first one is that in the Keras we need to specify the traits under study using a list() where, separated by a comma, the names of the traits under study are provided. The second one is that we need to specify the loss_weights for each trait. One simple approach is to use the same weights for all traits: for example, (1,1,1), if we have three traits or (0.3333,0.3333, 0.3333), this approach is valid when all traits are on the same scale, but when traits are on different scales, we suggest using different weights for each continuous trait. In this example, different weights were used, since each trait has a different scale and the weights were built as follows: (1) first we calculated the median of each trait, (2) then we calculated the 0.25 and 0.75 quantiles for each

trait, (3) then we calculated the maximum distance in terms of absolute value between the median and both quantiles, (4) then we used as the weight for the first trait (GY) its calculated distance, and (5) then we used as weight for the second trait the value obtained by dividing the distance of the first trait by the distance of the second trait, and finally the weight for the third trait was also obtained by dividing the distance of the first trait by the distance of the third trait. Finally, the fitting process is done in part d), and it is the same as the fitting process for the univariate DNN model, except that the training set of the response variables is provided as a list. These steps are only suggestions that can work for some data sets, but there is no guarantee that they can work for all data sets.

Next, we called the flag Code_Tuning_With_Flags_MT_normal.R. In Appendix 4, it is used to implement the tuning process for selecting the best combination of hyperparameters, and after selecting the best combination of hyperparameters, the multi-trait DNN with the optimum hyperparameters is refitted, and the prediction performance using cross-validation is evaluated with this refitted model. The grid search implemented in this code is given next:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_normal.R",runs_dir
= '_tuningE1',
          flags=list(dropout1= c(0,0.05),
               units = c(56,97),
               activation1=("relu"),
               batchsize1=c(22),
               Epoch1=c(1000),
               learning_rate=c(0.001),
             val_split=c(0.25)), sample=0.5, confirm =FALSE,echo =F)
```

The results of implementing the last random grid search (sample = 0.5) that consists of four combinations of hyperparameters resulting from two dropout values (0, 0.05) and two values of units (56, 97) are given in Table 12.4, which shows that the best predictions were obtained for trait PH in terms of Pearson's correlation and MAAPE.

### 12.4.2 DNN with Multivariate Binary Outcomes

For illustrating the process of training multivariate binary DNN models, we used the same data set (Data_Toy_EYT.RData) as in Example 12.1 in this chapter. As was explained in Example 12.1, this data set contains four environments (Bed5IR, EHT, Flat5IR, and LHT), 40 lines in each environment, a genomic relationship matrix of order 40 $\times$ 40, four traits (DTHD, DTMT, GY, and Height) of which DTHD and DTMT are ordinal traits, trait GY is continuous, and trait Height is binary. However, for the implementation of the multivariate binary DNN model, we converted ordinal traits DTHD and DTMT into binary traits, making 0 the levels of 1, and 1 the levels

**Table 12.4** Prediction performance of three continuous traits with three hidden layers without genotype × environment interaction

| Environment | Trait | Pearson | SE_Pearson | MAAPE | SE_MAAPE | MSE | SE_MSE |
|---|---|---|---|---|---|---|---|
| EBU | Yield | 0.2643 | 0.1593 | 0.1797 | 0.0214 | 1.871 | 0.3754 |
| KAK | Yield | 0.2834 | 0.1287 | 0.0835 | 0.0134 | 0.3899 | 0.1735 |
| KTI | Yield | 0.088 | 0.1992 | 0.1756 | 0.0158 | 1.5099 | 0.3677 |
| EBU | ASI | 0.0924 | 0.2167 | 0.2936 | 0.0324 | 0.5816 | 0.1267 |
| KAK | ASI | −0.0234 | 0.2562 | 0.6847 | 0.0597 | 1.2333 | 0.2179 |
| KTI | ASI | −0.1824 | 0.2311 | 0.2631 | 0.0314 | 1.1295 | 0.4023 |
| EBU | PH | 0.4658 | 0.149 | 0.0628 | 0.0122 | 351.9016 | 158.214 |
| KAK | PH | 0.5608 | 0.0711 | 0.0346 | 0.0028 | 87.6928 | 9.4417 |
| KTI | PH | 0.4839 | 0.1397 | 0.0613 | 0.0106 | 289.4436 | 83.5068 |

of 2 and 3. For this reason, the illustration of the multivariate binary DNN model was done using the following three traits: DTHD, DTMT, and Height.

The default flags for training multivariate binary outcomes are given next. They should be put in the R code (R Core Team 2019) called Code_Tuning_With_Flags_MT_Binary.R.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
 flag_numeric("val_split",0.2))

####b) Defining the multi-trait DNN model
input <- layer_input(shape=dim(X_trII)[2],name="covars")

# add hidden layers
base_model <- input %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units=1,activation="sigmoid", name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units= 1, activation="sigmoid",name="response_2")

# add output 3
yhat3 <- base_model %>%
 layer_dense(units= 1, activation="sigmoid",name="response_3")

#c) Compiling the multi-trait model
model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3)) %>%
 compile(optimizer =optimizer_adam(lr=FLAGS$learning_rate),
     loss=list(response_1="binary_crossentropy",
response_2="binary_crossentropy",response_3=
"binary_crossentropy"),
     metrics=list(response_1="accuracy",response_2="accuracy",
response_3="accuracy"),
     loss_weights=list(response_1=1,response_2=1,response_3=1))
###1,3.2,0.024
```

```
print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")
 })

early_stop <- callback_early_stopping(monitor = c("val_loss"),
mode='min', patience =50)

# d) Fitting multi-trait model
model_fit <- model %>%
 fit(x=X_trII,
   y=list(response_1=y_trII[,1],response_2=y_trII[,2],
response_3=y_trII[,3]),
   epochs=FLAGS$Epoch1,
   batch_size =FLAGS$batchsize1,
   validation_split=FLAGS$val_split,
   verbose=0, callbacks = list(early_stop,print_dot_callback))
```

Also in part a) are given the default flags for implementing the multi-trait binary DNN model. In part b) is defined the multi-trait binary DNN model where we can see that the process of adding the dimension of the input and the hidden layers is exactly the same as for the multi-trait continuous DNN model. Also, the number of output layers depends on the number of traits under study, that is, if there are three traits under study, we need to specify three output layers with one unit for each output layer, as was done in continuous multivariate outcomes, but the key difference is that now for binary multivariate outcomes, instead of using the linear activation function in the output layer, we use the sigmoid activation function for each of the output layers, since we are dealing with binary multivariate outcomes. The multi-trait binary DNN model is compiled in part c); this process is very similar to the compilation process of a multi-trait continuous DNN model, except that now we use the binary_crossentropy loss function for each trait under study, we also use accuracy as the metric for each of the binary traits and we use the same weight (in this case, 1, 1, and 1) for each of the three traits. We now use the same weights because the three traits under study are in the same type of response variable and scale. Finally, in part d) is given the code for the fitting process, which is exactly the same as for multi-trait continuous DNN models.

These flags (Code_Tuning_With_Flags_MT_Binary.R) can be used with Appendix 4 with some small modifications such as those just described above, which are related to (1) the activation function used for the output layers, (2) the loss function used for each trait since now we should use binary_crossentropy, (3) the metrics used for each trait since now we should use accuracy for each trait, (4) the weights used in the compilation process since now we will use the same weight for each trait, and (5) the prediction process since we will replace the following code of Appendix 4:

```
# predict values for test set
Yhat<-predict(model,X_tst)%>%
 data.frame()%>%
 setNames(colnames(y_trn))
YP=Yhat
```

With the following code to guarantee binary predictions:

```
# predict values for test set
Yhat<-predict(model,X_tst)%>%
 data.frame()%>%
 setNames(colnames(y_trn))
YP=matrix(NA,ncol=ncol(y2),nrow=nrow(Yhat))
head(Yhat)
P_T1=ifelse(Yhat[,1]>0.5,1,0)

P_T2=ifelse(Yhat[,2]>0.5,1,0)
P_T3=ifelse(Yhat[,3]>0.5,1,0)
YP[,1]=P_T1
YP[,2]=P_T2
YP[,3]=P_T3
```

The random grid search was used since sample = 0.5, for the prediction under a multi-trait binary DNN model, as is shown next:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Binary.R",runs_dir
= '_tuningE1',
          flags=list(dropout1= c(0,0.05),
                units = c(56,97),
                activation1=("relu"),
                batchsize1=c(22),
                Epoch1=c(1000),
                learning_rate=c(0.001),
               val_split=c(0.25)), sample=0.5,confirm =FALSE,echo =F)
```

The important thing about this random grid search is that the tuning_run() function calls the flags developed above for the training process of multi-trait binary DDN models. Also, the total number of hyperparameters of the grid search is four, since we set a unique value for all the hyperparameters, except for the hyperparameters dropout with two values (0, 0.05) and the units with another two values (56, 97). It is important to point out that of the four total hyperparameter combinations, only two were evaluated, since we implemented a random grid search by specifying sample = 0.5.

In Table 12.5 is given the prediction performance for the average of the five outer fold cross-validations used for training the multi-trait binary DNN model, the prediction performance reported as metrics, the PCCC, the Kappa coefficient, the sensitivity, and the specificity. From these results, it is evident that the best predictions belong to trait DTHD and trait DTMT. For all trait-environment

**Table 12.5** Prediction performance of multivariate binary data for three hidden layers with genotype × environment interaction and with five outer fold cross-validations

| Trait | Env | PCCC | SE_PCCC | Kappa | SE_Kappa | Sensitivity | Specificity |
|---|---|---|---|---|---|---|---|
| DTHD | Bed5IR | 0.800 | 0.109 | 0.533 | 0.209 | 0.700 | 0.888 |
| DTHD | EHT | 0.775 | 0.047 | 0.452 | 0.104 | 0.767 | 0.808 |
| DTHD | Flat5IR | 0.775 | 0.061 | 0.437 | 0.192 | 0.700 | 0.819 |
| DTHD | LHT | 0.750 | 0.040 | 0.344 | 0.140 | 0.693 | 0.751 |
| DTMT | Bed5IR | 0.800 | 0.109 | 0.533 | 0.209 | 0.888 | 0.700 |
| DTMT | EHT | 0.775 | 0.047 | 0.452 | 0.104 | 0.808 | 0.767 |
| DTMT | Flat5IR | 0.775 | 0.061 | 0.437 | 0.192 | 0.819 | 0.700 |
| DTMT | LHT | 0.750 | 0.040 | 0.344 | 0.140 | 0.751 | 0.693 |
| Height | Bed5IR | 0.575 | 0.031 | 0.115 | 0.065 | 0.520 | 0.607 |
| Height | EHT | 0.850 | 0.061 | 0.670 | 0.123 | 0.760 | 0.910 |
| Height | Flat5IR | 0.725 | 0.073 | 0.363 | 0.152 | 0.850 | 0.550 |
| Height | LHT | 0.750 | 0.040 | 0.492 | 0.074 | 0.800 | 0.767 |

combinations, the PCCC was larger than 0.5, that is, larger than the probability of a correct classification by chance since we are dealing with binary outcomes.

### *12.4.3 DNN with Multivariate Ordinal Outcomes*

To illustrate the training of multivariate ordinal or categorical DNN models, we used the same data set (Data_Toy_EYT.RData) used in this chapter in Example 12.1 and before for training multivariate binary DNN models. However, now the implementation of the multivariate ordinal DNN model was done with traits DTHD, DTMT, and GY, but since GY is a continuous trait, this was converted to an ordinal outcome in the following way: if GY is less than 3.2, then the outcome was set to 0, but if $3.2 < GY < 5.8$, the outcome was set to 1, while if $5.8 < GY < 6.2$, the ordinal outcome was set to 2, and finally, if $GY > 6.2$, the outcome was set to 3. This means the training of the multivariate ordinal DNN model was done with three traits (DTHD, DTMT, and GY) where the first two had three levels and the last one had four levels.

Next are provided the default flags that we suggest placing in an R (R Core Team 2019) file called Code_Tuning_With_Flags_MT_Ordinal.R.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
 flag_numeric("val_split",0.2))
```

```
####b) Defining the multi-trait ordinal DNN model
input <- layer_input(shape=dim(X_trII)[2],name="covars")

# add hidden layers
base_model <- input  %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units=3,activation="softmax", name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units= 3, activation="softmax",name="response_2")

# add output 3
yhat3 <- base_model %>%
 layer_dense(units= 4, activation="softmax",name="response_3")

#c) Compiling the multi-trait ordinal DNN model
model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3)) %>%compile(optimizer =optimizer_adam
(lr=FLAGS$learning_rate),
loss=list(response_1="categorical_crossentropy",
response_2="categorical_crossentropy",
response_3="categorical_crossentropy"),
metrics=list(response_1="accuracy",response_2="accuracy",
response_3="accuracy"),
loss_weights=list(response_1=1,response_2=1,response_3=1))

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")
 })

early_stop <- callback_early_stopping(monitor = c("val_loss"),
mode='min', patience =50)

# d) Fitting multi-trait model
model_fit <- model %>%
 fit(x=X_trII,
   y=list(response_1=y_trII[,1],response_2=y_trII[,2],
response_3=y_trII[,3]),
   epochs=FLAGS$Epoch1,
   batch_size =FLAGS$batchsize1,
   validation_split=FLAGS$val_split,
   verbose=0, callbacks = list(early_stop,print_dot_callback))
```

The default flags for implementing the multi-trait ordinal DNN model are in part a). In part b) is defined the multi-trait categorical DNN model, for which its implementation is equal to binary outcomes, with two exceptions: (1) now the softmax activation function is used for each trait in the output layer and (2) the number of units in each output layer depends on the number of categories of each trait under study, in this case, 3, 3, and 4 for the first, second, and third traits, respectively. With regard to the compilation process (part c), it is the same as the compilation of the multi-trait binary DNN model, except that now the categorical_crossentropy loss function is used for each trait under study. Finally, the code for the fitting process (part d) is exactly the same as for multi-trait binary DNN models.

The flags given above for training multivariate ordinal outcomes (Code_Tuning_With_Flags_MT_Ordinal.R) can be used with Appendix 4 with the following modifications: (1) use the softmax activation function for each output layer, (2) use the categorical_crossentropy for each trait, (3) use accuracy as the metric for each trait, (4) use the same weights for each trait, and (5) replace the following code of Appendix 4:

```
# predict values for test set
Yhat<-predict(model,X_tst)%>%
 data.frame()%>%
 setNames(colnames(y_trn))
YP=Yhat
```

Use the following code to obtain categorical outcomes as predictions:

```
Yhat<-predict(model,X_tst)%>%
 data.frame()%>%
 setNames(colnames(y_trn))
YP=matrix(NA,ncol=ncol(y),nrow=nrow(Yhat))
head(Yhat)
P_T1=(apply(data.matrix(Yhat[,1:3]),1,which.max)-1)
P_T2=(apply(data.matrix(Yhat[,4:6]),1,which.max)-1)
P_T3=(apply(data.matrix(Yhat[,7:10]),1,which.max)-1)
YP[,1]=P_T1
YP[,2]=P_T2
YP[,3]=P_T3
```

The random grid search (since sample = 0.5) for the prediction of a multi-trait categorical DNN model is given next:

```
 runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Ordinal.R",
runs_dir = '_tuningE1',
           flags=list(dropout1= c(0,0.05),
                 units = c(56,97),
                 activation1=("relu"),
                 batchsize1=c(22),
                 Epoch1=c(1000),
                 learning_rate=c(0.001),
                val_split=c(0.25)), sample=0.5,confirm =FALSE,echo =F)
```

**Table 12.6** Prediction performance of multivariate ordinal data for three hidden layers with genotype × environment interaction and five outer fold cross-validation

| Trait | Env | PCCC | SE_PCCC | Kappa | SE_Kappa | Sensitivity | Specificity |
|---|---|---|---|---|---|---|---|
| **DTHD** | **Bed5IR** | **0.625** | **0.0685** | **0.408** | **0.0922** | **0.6** | **0.5** |
| **DTHD** | **EHT** | **0.625** | **0.0884** | **0.398** | **0.1414** | **0.6833** | **0.25** |
| **DTHD** | **Flat5IR** | **0.6** | **0.0919** | **0.426** | **0.1224** | **0.7467** | **0.55** |
| DTHD | LHT | 0.575 | 0.0637 | 0.287 | 0.0855 | 0.6167 | 0.0833 |
| DTMT | Bed5IR | 0.525 | 0.0612 | 0.246 | 0.1179 | 0.8 | 0.3917 |
| DTMT | EHT | 0.65 | 0.0729 | 0.444 | 0.1136 | 0.4583 | 0.6533 |
| DTMT | Flat5IR | 0.575 | 0.05 | 0.391 | 0.0448 | 0.78 | 0.3083 |
| DTMT | LHT | 0.55 | 0.0848 | 0.24 | 0.1589 | 0.7917 | 0.3958 |
| GY | Bed5IR | 0.4 | 0.1275 | 0.141 | 0.1666 | NaN | 0.3 |
| GY | EHT | 0.625 | 0.0395 | 0.23 | 0.0949 | NaN | 0.2 |
| GY | Flat5IR | 0.45 | 0.075 | 0.156 | 0.0846 | NaN | 0.4333 |
| GY | LHT | 0.425 | 0.0306 | −0.06 | 0.0579 | 0.5667 | 0.3733 |

This tuning process was implemented using a random grid search since sample = 0.5 was specified inside the running_run() function, which means that only two of the four hyperparameter combinations should be evaluated at each iteration of the deep learning algorithm.

The results of implementing the multi-trait ordinal DNN model using the best combination of hyperparameters resulting from the above research grid are given in Table 12.6, where we can see that the metrics used for evaluating the prediction performance were the same as those used for the multi-trait binary outcomes, but now the best predictions were observed in trait DTHD and the worst in trait GY. However, note that by chance alone now for traits DTHD and DTMT the probability is 1/3 while for trait GY this probability is ¼ since for the first two traits there are three levels, while for the third trait there are four response options.

### *12.4.4 DNN with Multivariate Count Outcomes*

To illustrate the training process of the multivariate count DNN model, we used the data called Data_Multi_Count_Toy.RData, which is a modified version of the data called Data_ Count_Toy.RData. The basic modification is that the modified version has two traits instead of one, which are denoted as y1 and y2. For this reason, again this data set contains 115 lines, evaluated in three environments (Batan2012, Batan2014, and Chunchi2014) and the total number of observations of this data set is 298. Next, we provide the default flags that we suggest placing in an R file called Code_Tuning_With_Flags_MT_Count.R.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
 flag_numeric("val_split",0.2))

####b) Defining the multi-trait count DNN model
input <- layer_input(shape=dim(X_trII)[2],name="covars")

# add hidden layers
base_model <- input %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units=1,activation="exponential", name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units=1, activation="exponential",name="response_2")

#c) Compiling the multi-trait ordinal DNN model
model <- keras_model(input,list(response_1=yhat1,response_2=yhat2))
%>%
 compile(optimizer =optimizer_adam(lr=FLAGS$learning_rate),
     loss=list(response_1="poisson",response_2="poisson"),
     metrics=list(response_1="mse",response_2="mse"),
     loss_weights=list(response_1=1,response_2=1))

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")
 })

early_stop <- callback_early_stopping(monitor = c("val_loss"),
mode='min', patience =50)

# d) Fitting multi-trait count DNN model
model_fit <- model %>%
 fit(x=X_trII,
   y=list(response_1=y_trII[,1],response_2=y_trII[,2]),
   epochs=FLAGS$Epoch1,
```

```
batch_size =FLAGS$batchsize1,
validation_split=FLAGS$val_split,
verbose=0, callbacks = list(early_stop,print_dot_callback))
```

The default flags for the multi-trait count DNN model are in part a). The process of building the multi-trait count DNN model (part b) is very similar to the building of continuous multivariate outcomes, except that now the exponential activation function should be used. The compilation process (part c) is the same as the compilation of the multi-trait continuous DNN model, except that now the Poisson loss function is used for each trait under study, and the fitting process (part d) is exactly the same as for multi-trait continuous DNN models.

The flags given above for training multivariate count outcomes (Code_Tuning_With_Flags_MT_Count. R) can be used with Appendix 4 with the following modifications: (1) use the exponential activation function for each output layer, (2) use the Poisson loss function for each trait, and (3) use the same weights for each trait.

Next, we give the random grid search used for tuning a multi-trait count DNN model:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Count.R",runs_dir
= '_tuningE1',
          flags=list(dropout1= c(0,0.05),
                units = c(56,97),
                activation1=("relu"),
                batchsize1=c(22),
                Epoch1=c(1000),
                learning_rate=c(0.001),
               val_split=c(0.2)),sample=0.5,confirm =FALSE,echo =F)
```

Again, we used the tuning_run() function to perform the tuning process which now was done with a random grid search since we specified sample = 0.5. Here the total number of hyperparameters is four, since only hyperparameters % of dropout and number of units have two values.

The prediction performance of training the multi-trait count data is given in Table 12.7. Now the metrics used for evaluating the accuracy were the mean square error of prediction and MAAPE. The best predictions across environments belong to

**Table 12.7** Prediction performance of multivariate count data for three hidden layers with genotype × environment interaction and five outer fold cross-validation

| Trait | Environment | MAAPE | SE_MAAPE | MSE | SE_MSE |
|---|---|---|---|---|---|
| y1 | Batan2012 | 0.7975 | 0.0498 | 1.6228 | 0.1696 |
| y1 | Batan2014 | 0.6739 | 0.0648 | 1.175 | 0.3795 |
| y1 | Chunchi2014 | 0.7933 | 0.0517 | 22.2328 | 3.3194 |
| y2 | Batan2012 | 0.9184 | 0.0534 | 1.0801 | 0.3943 |
| y2 | Batan2014 | 0.7897 | 0.0619 | 1.0169 | 0.5013 |
| y2 | Chunchi2014 | 0.5128 | 0.0254 | 9.5645 | 0.7366 |

trait y1 in environments Batan2012 and Batan2014, and to trait y2 in environment Chunchi2014. Results given in Table 12.7 belong to the best combination of hyperparameters of the above grid. The comparison of prediction performance between traits in MSE terms is not valid when the traits are on different scales.

### *12.4.5 DNN with Multivariate Mixed Outcomes*

Finally, in this chapter, we provided key elements for training DNN models for binary, categorical, count, and continuous outcomes. However, with the implementation of these DNN models, it is clear that DNN models are a novel tool for training univariate and multivariate genomic prediction models for binary, categorical, count, and mixed outcomes. The power to train univariate and multivariate nonlinear regression with count data is unique to deep learning models since similar tools in conventional statistical learning are not efficient and only a few are available. But the gain in training multivariate DNN models is only due to the increase in the sample size by modeling more than one trait simultaneously since the DNN models just studied do not take into account a variance–covariance matrix of traits to capture the correlation between traits. However, we also emphasize that the training process of DNN models is very challenging since more time, thought, experimentation, and resources are required for training these models than other statistical machine learning models studied in this book, since for most of the statistical machine learning algorithms, the search space for finding the optimum combination of hyperparameters is small compared to DNN models. For these reasons, we strongly suggest using the random grid search (or other new approaches like Bayesian optimization or genetic algorithms) since the larger the number of hyperparameters, the bigger the number of hyperparameter combinations that need to be evaluated due to the quick explosion in the number of hyperparameter combinations. For this reason, the use of the random grid search that explores only a fraction of the total combination of hyperparameters is more efficient.

Next the default flags are given in an R file called Code_Tuning_With_Flags_MT_Mixed.R.

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
 flag_numeric("val_split",0.2))

####b) Defining the DNN model
input <- layer_input(shape=dim(X_trII)[2],name="covars")
```

```
# add hidden layers
base_model <- input %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation=FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units = 3,activation="softmax", name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units = 1, activation="exponential",name="response_2")

# add output 3
yhat3 <- base_model %>%
 layer_dense(units = 1, name="response_3")

# add output 3
yhat4 <- base_model %>%
 layer_dense(units =1, activation="sigmoid",name="response_4")

#c) Compiling the multi-trait mixed DNN model
Model=keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3,response_4=yhat4)) %>%
 compile(optimizer =optimizer_adam(lr=FLAGS$learning_rate),
loss=list(response_1="categorical_crossentropy",response_2="mse",
response_3="mse",response_4="binary_crossentropy"), metrics=list
(response_1="accuracy",response_2="mse",response_3="mse",
response_4="accuracy"), loss_weights=list(response_1=1,
response_2=1,response_3=1,response_4=1))

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")
 })

early_stop <- callback_early_stopping(monitor = c("val_loss"),
mode='min', patience =50)

# d) Fitting multi-trait mixed DNN model
model_fit <- model %>%
fit(x=X_trII,y=list(response_1=y_trII[,1],response_2=y_trII[,2],
response_3=y_trII[,3],response_4=y_trII[,4]), epochs=FLAGS$Epoch1,
batch_size =FLAGS$batchsize1, validation_split=FLAGS$val_split,
verbose=0, callbacks =list(early_stop,print_dot_callback))
```

The default flags for the multi-trait mixed outcome DNN model are in part a). The building process of the multi-trait mixed outcome DNN model (part b) is different only in the specification of the outputs. Since now we have specified an activation function for each response variable, the first response variable is an ordinal output with three categories. For this reason, we specified three neurons and the softmax activation function. The second response variable is a count, and for this reason, we specified one unit (neuron) and the exponential activation function. The third response variable is continuous, and for this reason, we specified only one neuron and the linear activation function (default activation function). Finally, the last response variable is binary, and for this reason, we specified only one neuron and the sigmoid activation function. With regard to the compilation process (part c), we had to specify a mixture loss function due to the fact that we had multivariate mixed outcomes. For this reason, the mixture loss function in this example is composed of four types of losses: "categorical_crossentropy," "mse," "mse," and "binary_crossentropy," which are appropriate for a mixed outcome of ordinal, count, continuous, and binary response variables. The same type of mixture is required for specifying the metrics to evaluate the prediction accuracy to guarantee that they are in agreement with the response variables. For this reason, in this case, the specification of the metrics contains: "accuracy," "mse," "mse," and "accuracy." The first metric is for the ordinal response variable, the second for the count output, the third for the continuous outcome, and the last one for the binary outcome. The fitting process (part d) is exactly the same as the fitting process of the previous multivariate deep learning models with only one type of scale in the output.

The flags given above for training multivariate mixed outcomes (Code_Tuning_With_Flags_MT_Mixed. R) can be used with Appendix 5.

Next is given the grid search used for tuning a multi-trait mixed outcome DNN model:

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Mixed.R",runs_dir =
'_tuningE1',
          flags=list(dropout1= c(0,0.05),
                units = c(56,97),
                activation1=("relu"),
                batchsize1=c(30),
                Epoch1=c(1000),
                learning_rate=c(0.01),
               val_split=c(0.10)),sample=0.5,confirm =FALSE,echo =F)
```

Again, we used the tuning_run() function to perform the tuning process, which now was done with the full grid search since we specified sample = 1. Here the total number of hyperparameters is four, since hyperparameters % of dropout and number of units have only two values.

The prediction performance of training multi-trait mixed outcome data is given in Table 12.8. The metrics used for evaluating the accuracy were MAAPE for the continuous and count response variables, and the proportion of cases correctly classified (PCCC) for the ordinal and binary traits. The best predictions for the two

**Table 12.8** Prediction performance of multivariate mixed outcome data for three hidden layers with genotype × environment interaction and five outer fold cross-validations

| Trait | Env | MAAPE | SE_MAAPE | PCCC | SE_PCCC |
|---|---|---|---|---|---|
| DTHD | Bed5IR | – | – | 0.600 | 0.073 |
| DTHD | EHT | – | – | 0.675 | 0.064 |
| DTHD | Flat5IR | – | – | 0.550 | 0.050 |
| DTHD | LHT | – | – | 0.650 | 0.073 |
| DTMT | Bed5IR | 0.628 | 0.065 | – | – |
| DTMT | EHT | 0.609 | 0.059 | – | – |
| DTMT | Flat5IR | 0.669 | 0.038 | – | – |
| DTMT | LHT | 0.678 | 0.092 | – | – |
| GY | Bed5IR | 0.106 | 0.010 | – | – |
| GY | EHT | 0.124 | 0.016 | – | – |
| GY | Flat5IR | 0.169 | 0.032 | – | – |
| GY | LHT | 0.154 | 0.005 | – | – |
| Height | Bed5IR | – | – | 0.575 | 0.050 |
| Height | EHT | – | – | 0.850 | 0.073 |
| Height | Flat5IR | – | – | 0.725 | 0.047 |
| Height | LHT | – | – | 0.675 | 0.050 |

traits evaluated with MAAPE were those of trait GY, while for the ordinal and binary traits we can see that the PCCC was better for the trait Height that includes only two categories (Table 12.8). Again, these predictions belong to the best combination of hyperparameters of the above grid. Practitioners should remember that we only reported the MAAPE for the count and continuous traits, since it made no sense for the ordinal and binary outcomes. Similarly, we only reported the PCCC for the binary and ordinal traits since it made no sense for the count and continuous outcomes.

## Appendix 1

R code for training a univariate binary outcome with four hidden layers

```
rm(list=ls())
library(BMTME)
library(tensorflow)
library(keras)
library(caret)
library(plyr)
library(tidyr)
library(dplyr)
library(tfruns)
options(bitmapType='cairo')
```

```
##########Set seed for reproducible results###################
use_session_with_seed(64)

###########Loading the EYT_Toy data set#######################
load("Data_Toy_EYT.RData")

#############Genomic relationship matrix (GRM)#################
Gg=data.matrix(G_Toy_EYT)
G=Gg
dim(G)

############Phenotypic data ###################################
Data_Pheno=Pheno_Toy_EYT
head(Data_Pheno)

summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
 # if (!inherits(object, "BMTMECV")) stop("This function only works for
objects of class 'BMTMECV'")
 results$Observed=as.factor(results$Observed)
 results$Predicted=as.factor(results$Predicted)
 results %>%
  group_by(Env, Partition) %>%
  summarise(PCCC=confusionMatrix(table(Observed,Predicted))$
overall[1],
      PCCC_Lower=confusionMatrix(table(Observed,Predicted))$overall
[3],
      PCCC_Upper=confusionMatrix(table(Observed,Predicted))$overall
[4],
       Kappa=confusionMatrix(table(Observed,Predicted))$overall[2],
      Sensitivity=confusionMatrix(table(Observed,Predicted))$byClas
[1],
       Specificity=confusionMatrix(table(Observed,Predicted))$byClas
[2]) %>%
  select(Env, Partition, PCCC,PCCC_Lower,PCCC_Upper,Kappa,
Sensitivity,Specificity) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Env) %>%
  summarise(SE_PCCC= sd(PCCC, na.rm = T)/sqrt(n()), PCCC = mean(PCCC,
na.rm = T),
      SE_PCCC_Lower= sd(PCCC_Lower, na.rm = T)/sqrt(n()), PCCC_Lower =
mean(PCCC_Lower, na.rm = T),
      SE_PCCC_Upper= sd(PCCC_Upper, na.rm = T)/sqrt(n()), PCCC_Upper=
mean(PCCC_Upper, na.rm = T),
      SE_Kappa= sd(Kappa, na.rm = T)/sqrt(n()), Kappa = mean(Kappa, na.
rm = T),
       SE_Sensitivity= sd(Sensitivity, na.rm = T)/sqrt(n()),
Sensitivity = mean(Sensitivity, na.rm = T),
      SE_Specificity= sd(Specificity, na.rm = T)/sqrt(n()), Specificity =
mean(Specificity, na.rm = T)) %>%
```

```
  select(Env,PCCC, SE_PCCC,PCCC_Lower, SE_PCCC_Lower,
      PCCC_Upper,SE_PCCC_Upper,Kappa,SE_Kappa, Sensitivity,
SE_Sensitivity,Specificity,SE_Specificity ) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
        complete = presum,
        extended = {
         finalSum$Partition <- 'All'
         presum$Partition <- as.character(presum$Partition)
         presum$SE_PCCC <- NA
         presum$SE_PCCC_Lower <- NA
         presum$SE_PCCC_Upper <- NA
         presum$SE_Kappa <- NA
         presum$Sensitivity<- NA
         presum$Specificity<- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

########Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data_Pheno$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZE=model.matrix(~0+as.factor(Data_Pheno$Env))
Z2GE=model.matrix(~0+Z1G:as.factor(Data_Pheno$Env))
nCV=5  ###Number of outer Cross-validation

############Selecting the response variable#######################
Y <- as.matrix(Data_Pheno[, -c(1, 2)])

####Training testing sets using the BMTME package###############
pheno <- data.frame(GID =Data_Pheno[, 1], Env =Data_Pheno[, 2],
          Response =Data_Pheno[, 3])

CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)

########Here are printed the testing observations of each fold#####
CrossV$CrossValidation_list

#######Final X and y=Height to use for training the model##############
y=(Data_Pheno[, 6])
length(y)
y
X=cbind(ZE,Z1G)
dim(X)
Learn_val=c(0.001,0.01,0.1,0.5,1)
Final_results=data.frame()
```

```
for (e in 1:5){
#e=1

 digits=4
 Names_Traits=colnames(Y)
 results=data.frame()
 t=1

 for (o in 1:5){
#  o=2
  tst_set=CrossV$CrossValidation_list[[o]]
  X_trn=(X[-tst_set,])
  X_tst=(X[tst_set,])
  y_trn=y[-tst_set]
  y_tst=y[tst_set]

  ################Inner cross-validation##########################
  nCVI=5  ####Number of folds for inner CV
  Hyperpar=data.frame()
  for (i in 1:nCVI){
 #  i=1
   Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
   X_trII=X_trn[Sam_per,]
   y_trII=y_trn[Sam_per]

   #####a) Grid search using the tuning_run() function of tfruns
package########
   runs.sp<-tuning_run("Code_Tuning_With_Flags_Bin_4HL.R",runs_dir
= '_tuningE1',
             flags=list(dropout1= c(0,0.05),
                  units = c(67,100),
                  activation1=("relu"),
                  batchsize1=c(28),
                  Epoch1=c(1000),
                  learning_rate=c(Learn_val[e]),
                  val_split=c(0.2)),sample=1,confirm =FALSE,echo =F)
   runs.sp[,2:5]
   ###b) ###### Ordering in the same way all grids
   runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_dropout1),]
   runs.sp$grid_length=1:nrow(runs.sp)
Parameters=data.frame(grid_length=runs.sp$grid_length,
metric_val_acc=runs.sp$metric_val_acc,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_activation1=runs.
sp$flag_activation1)
   Hyperpar=rbind(Hyperpar,data.frame(Parameters))
  }
  Hyperpar %>%
   group_by(grid_length) %>%
   summarise(val_acc=mean(metric_val_acc),
        dropout1=mean(flag_dropout1),
        units=mean(flag_units),
```

```
        batchsize1=mean(flag_batchsize1),
        learning_rate=mean(flag_learning_rate),
        epochs=mean( epochs_completed)) %>%
   select(grid_length,val_acc,dropout1,units,batchsize1,
learning_rate, epochs) %>%
   mutate_if(is.numeric, funs(round(., 3))) %>%
   as.data.frame() -> Hyperpar_Opt
  ############Optimal hyperparameters############
  Max=max(Hyperpar_Opt$val_acc)
  pos_opt=which(Hyperpar_Opt$val_acc==Max)
  pos_opt=pos_opt[1]
  Optimal_Hyper=Hyperpar_Opt[pos_opt,]
  #####Selectiong the best hyperparameters
  Drop_O=Optimal_Hyper$dropout1
  Epoch_O=round(Optimal_Hyper$epochs,0)
  Units_O=round(Optimal_Hyper$units,0)
  activation_O=unique(Hyperpar$flag_activation1)
  batchsize_O=round(Optimal_Hyper$batchsize1,0)
  lr_O=Optimal_Hyper$learning_rate

  print_dot_callback <- callback_lambda(
   on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat("\n")
    cat(".")})

  ###########Refitting the model with the optimal values#############
  model_Sec<-keras_model_sequential()
  model_Sec %>%
   layer_dense(units =Units_O , activation =activation_O, input_shape
= c(dim(X_trn)[2])) %>%
   layer_dropout(rate =Drop_O) %>%
   layer_dense(units =Units_O, activation =activation_O) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, activation =activation_O) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, activation =activation_O) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =1, activation ="sigmoid")

  model_Sec %>% compile(
   loss = "binary_crossentropy",
   optimizer = optimizer_adam(lr_O),
   metrics = c('accuracy'))

  ModelFinal <-model_Sec %>% fit(
   X_trn, y_trn,
   epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
   verbose=0,callbacks=list(print_dot_callback))

  ####e) Prediction of testing set ##########################
  predicted=model_Sec %>% predict_classes(X_tst)
```

```
  Predicted=predicted
  Observed=y[tst_set]
  results<-rbind(results, data.frame(Position=tst_set,
                    Env=CrossV$Environments[tst_set],
                    Partition=o,
                    Units=Units_O,
                    Epochs=Epoch_O,
               Observed=round(Observed, digits), #$response, digits),
                    Predicted=round(Predicted, digits),
                    Trait=Names_Traits[t]))
  cat("CV=",o,"\n")
 }
 results

 Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
 Pred_Summary

 Final_results=rbind(Final_results,data.frame(learn_val=Learn_val
[e],Pred_Summary))
}
Final_results
write.csv(Final_results,file="Appendix1_Bin_Chapter12_modified.
csv")
```

## Appendix 2

R code for training a univariate categorical or ordinal outcome with four hidden layers

```
rm(list=ls())
library(BMTME)
library(tensorflow)
library(keras)
library(caret)
library(plyr)
library(tidyr)
library(dplyr)
library(tfruns)
options(bitmapType='cairo')

##########Set seed for reproducible results###################
use_session_with_seed(64)

###########Loading the EYT_Toy data set#######################
load("Data_Toy_EYT.RData")

#############Genomic relationship matrix (GRM)#################
Gg=data.matrix(G_Toy_EYT)
```

```
G=Gg
dim(G)

###########Phenotypic data ####################################
Data_Pheno=Pheno_Toy_EYT
head(Data_Pheno)

summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
 # if (!inherits(object, "BMTMECV")) stop("This function only works for
objects of class 'BMTMECV'")
 results$Observed=as.factor(results$Observed)
 results$Predicted=as.factor(results$Predicted)
 results %>%
  group_by(Env, Partition) %>%
  summarise(PCCC=as.numeric(confusionMatrix(table(Observed,
Predicted))$overall[1]),
       PCCC_Lower=as.numeric(confusionMatrix(table(Observed,
Predicted))$overall[3]),
       PCCC_Upper=as.numeric(confusionMatrix(table(Observed,
Predicted))$overall[4]),
       Kappa=as.numeric(confusionMatrix(table(Observed,Predicted))$
overall[2]),
       Sensitivity=as.numeric(confusionMatrix(table(Observed,
Predicted))$byClas[1]),
       Specificity=as.numeric(confusionMatrix(table(Observed,
Predicted))$byClas[2])) %>%
  select(Env, Partition, PCCC,PCCC_Lower,PCCC_Upper,Kappa,
Sensitivity,Specificity) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Env) %>%
  summarise(SE_PCCC= sd(PCCC, na.rm = T)/sqrt(n()), PCCC = mean(PCCC,
na.rm = T),
      SE_PCCC_Lower= sd(PCCC_Lower, na.rm = T)/sqrt(n()), PCCC_Lower =
mean(PCCC_Lower, na.rm = T),
      SE_PCCC_Upper= sd(PCCC_Upper, na.rm = T)/sqrt(n()), PCCC_Upper=
mean(PCCC_Upper, na.rm = T),
      SE_Kappa= sd(Kappa, na.rm = T)/sqrt(n()), Kappa = mean(Kappa, na.
rm = T),
       SE_Sensitivity= sd(Sensitivity, na.rm = T)/sqrt(n()),
Sensitivity = mean(Sensitivity, na.rm = T),
      SE_Specificity= sd(Specificity, na.rm = T)/sqrt(n()), Specificity =
mean(Specificity, na.rm = T)) %>%
  select(Env,PCCC, SE_PCCC,PCCC_Lower,SE_PCCC_Lower,PCCC_Upper,
SE_PCCC_Upper,Kappa,SE_Kappa, Sensitivity,SE_Sensitivity,
Specificity,SE_Specificity ) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
```

```
        complete = presum,
        extended = {
         finalSum$Partition <- 'All'
         presum$Partition <- as.character(presum$Partition)
         presum$SE_PCCC <- NA
         presum$SE_PCCC_Lower <- NA
         presum$SE_PCCC_Upper <- NA
         presum$SE_Kappa <- NA
         presum$SE_Sensitivity<- NA
         presum$SE_Specificity<- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}
##############Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data_Pheno$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZE=model.matrix(~0+as.factor(Data_Pheno$Env))
Z2GE=model.matrix(~0+Z1G:as.factor(Data_Pheno$Env))
nCV=5  ###Number of outer Cross-validation

############Selecting the response variable######################
Y <- as.matrix(Data_Pheno[, -c(1, 2)])

########Training testing sets using the BMTME package##############
pheno <- data.frame(GID =Data_Pheno[, 1], Env =Data_Pheno[, 2],
          Response =Data_Pheno[, 3])

CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)

########Here are printed the testing observations of each fold#####
CrossV$CrossValidation_list

#######Final X and y=DTHD to use for training the model################
y=c(Data_Pheno[, 3])-1
nclas=length(unique(y))
length(y)
X=cbind(ZE,Z1G)
dim(X)
Learn_val=c(0.005,0.01,0.015,0.03,0.06)
Final_results=data.frame()
for (e in 1:5){
 digits=4
 Names_Traits=colnames(Y)
 results=data.frame()
 t=1

 for (o in 1:5){
  #o=2
  tst_set=CrossV$CrossValidation_list[[o]]
```

```
  X_trn=(X[-tst_set,])
  X_tst=(X[tst_set,])
  y_trn= to_categorical(y[-tst_set],nclas)
  y_tst= to_categorical(y[tst_set],nclas)

  ################Inner cross-
validation####################################
  nCVI=5 ####Number of folds for inner CV
  Hyperpar=data.frame()
  for (i in 1:nCVI){
   #i=1
   Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
   X_trII=X_trn[Sam_per,]
   y_trII=y_trn[Sam_per,]

   #####a) Grid search using the tuning_run() function of tfruns
package########
   runs.sp<-tuning_run("Code_Tuning_With_Flags_Ordinal_4HL2.R",
runs_dir = '_tuningEO',
            flags=list(dropout1= c(0,0.05),
                  units = c(67,100),
                  activation1=("relu"),
                  batchsize1=c(28),
                  Epoch1=c(1000),
                  learning_rate=c(Learn_val[e]),
                  val_split=c(0.2)),sample=1,confirm =FALSE,echo =F)
   runs.sp[,2:5]
   ###b) ###### Ordering in the same way all grids
   runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_dropout1),]

   runs.sp$grid_length=1:nrow(runs.sp)
   Parameters=data.frame(grid_length=runs.sp$grid_length,
metric_val_acc=runs.sp$metric_val_acc,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_activation1=runs.
sp$flag_activation1)
   Hyperpar=rbind(Hyperpar,data.frame(Parameters))
  }
  Hyperpar %>%
   group_by(grid_length) %>%
   summarise(val_acc=mean(metric_val_acc),
        dropout1=mean(flag_dropout1),
        units=mean(flag_units),
        batchsize1=mean(flag_batchsize1),
        learning_rate=mean(flag_learning_rate),
        epochs=mean( epochs_completed)) %>%
   select(grid_length,val_acc,dropout1,units,batchsize1,
learning_rate, epochs) %>%
   mutate_if(is.numeric, funs(round(., 3))) %>%
   as.data.frame() -> Hyperpar_Opt
  ############Optimal hyperparameters############
  Max=max(Hyperpar_Opt$val_acc)
```

```
  pos_opt=which(Hyperpar_Opt$val_acc==Max)
  pos_opt=pos_opt[1]
  Optimal_Hyper=Hyperpar_Opt[pos_opt,]
  #####Selectiong the best hyperparameters
  Drop_O=Optimal_Hyper$dropout1
  Epoch_O=round(Optimal_Hyper$epochs,0)
  Units_O=round(Optimal_Hyper$units,0)
  activation_O=unique(Hyperpar$flag_activation1)
  batchsize_O=round(Optimal_Hyper$batchsize1,0)
  lr_O=Optimal_Hyper$learning_rate
  print_dot_callback <- callback_lambda(
   on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat("\n")
    cat(".")})

  ############Refitting the model with the optimal values###########
  model_Sec<-keras_model_sequential()
  model_Sec %>%
   layer_dense(units =Units_O , activation =activation_O, input_shape
= c(dim(X_trn)[2])) %>%
   layer_batch_normalization() %>%
   layer_dropout(rate =Drop_O) %>%
   layer_dense(units =Units_O, activation =activation_O) %>%
   layer_batch_normalization() %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, activation =activation_O) %>%
   layer_batch_normalization() %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, activation =activation_O) %>%
   layer_batch_normalization() %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =nclas, activation ="softmax")

  model_Sec %>% compile(
   loss = "categorical_crossentropy",
   optimizer = optimizer_adam(lr_O),
   metrics = c('accuracy'))

  ModelFinal <-model_Sec %>% fit(
   X_trn, y_trn,
   epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
   verbose=0,callbacks=list(print_dot_callback))

  ####e) Prediction of testing set ##########################
  predicted=model_Sec %>% predict_classes(X_tst)
  Predicted=predicted
  Observed=y[tst_set]
  results<-rbind(results, data.frame(Position=tst_set,
                   Env=CrossV$Environments[tst_set],
                   Partition=o,
                   Units=Units_O,
                   Epochs=Epoch_O,
```

```
               Observed=round(Observed, digits), #$response, digits),
                    Predicted=round(Predicted, digits),
                    Trait=Names_Traits[t]))
  cat("CV=",o,"\n")
 }
 results

 Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
 Pred_Summary
 Final_results=rbind(Final_results,data.frame(learn_val=Learn_val
[e],Pred_Summary))
}
Final_results
write.csv(Final_results,file="Appendix1_Ord_Chapter12_modifiedV2.
csv")
```

# Appendix 3

R code for training a univariate count outcome with four hidden layers and Ridge regularization

```
rm(list=ls(all=TRUE))
library(plyr)
library(tidyr)
library(dplyr)
library(BMTME)
library(tensorflow)
library(keras)
library(tfruns)

#Loading Data_Count_Toy
load('Data_Count_Toy.RData',verbose=TRUE)
ls()

####Phenotipic data
dat_F=Pheno_Toy_Count
head(dat_F)
dim(dat_F)

######Genomic relationship matrix###
G =G_Toy_Count
dim(G)
G_0.5 = cholesky(G, tolerance = 1e-9)

#####Design matrices#####
ZL = model.matrix(~0+as.factor(GID),data=dat_F)
ZL = ZL%*%G_0.5
X_L = model.matrix(~0+Loc,data=dat_F)
```

```
dim(X_L)
X_LM = model.matrix(~0+ZL:Loc,data=dat_F)
X_Block = model.matrix(~0+as.factor(Block):Loc,data=dat_F)
dim(X_Block )

#################
#########Function for averaging the predictions############
summary.BMTMECV <- function(results, information = 'compact', digits =
4, ...) {
 results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(MSE = mean((Predicted-Observed)^2),
      MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed)))) %>%
  select(Environment, Trait, Partition, MSE, MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Environment, Trait) %>%
  summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
     SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>
%
  select(Environment, Trait, MSE, SE_MSE, MAAPE, SE_MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
        complete = presum,
        extended = {
         finalSum$Partition <- 'All'
         presum$Partition <- as.character(presum$Partition)
         presum$SE_MSE <- NA
         presum$SE_MAAPE <- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

##Number of fold cross-validation and TRN and TST sets
nCV=5
Data.Final_1=dat_F[,c(1,2,6)]
colnames(Data.Final_1)=c("Line","Env","Response")
Env=unique(Data.Final_1$Env)
nI=length(unique(Data.Final_1$Env))

############Training testing partitions####################
CrossV<-CV.KFold(Data.Final_1, K =nCV, set_seed=123)

X = cbind(X_L,ZL,X_Block,X_LM)
y=dat_F$y
```

```
Learn_val=c(0.005,0.01,0.015,0.03,0.06)
Final_results=data.frame()
digits=4
for (e in 1:5){
#e=5
 # Names_Traits=colnames(y)
 results=data.frame()
 t=1
for (o in 1:5){
  # o=2
  tst_set=CrossV$CrossValidation_list[[o]]
  X_trn=(X[-tst_set,])
  X_tst=(X[tst_set,])
  y_trn=y[-tst_set]
  y_tst=y[tst_set]

  ################Inner cross-validation#########################
  nCVI=5 ####Number of folds for inner CV
  Hyperpar=data.frame()
  for (i in 1:nCVI){
   #i=1
   Sam_per=sample(1:nrow(X_trn),nrow(X_trn))
   X_trII=X_trn[Sam_per,]
   y_trII=y_trn[Sam_per]

   #####a) Grid search using the tuning_run() function of tfruns
package########
   runs.sp<-tuning_run("Code_Tuning_With_Flags_Count_Lasso.R",
             flags=list(dropout1= c(0),
                  units = c(67,150),
                  activation1=("relu"),
                  batchsize1=c(28),
                  Epoch1=c(1000),
                  learning_rate=c(Learn_val[e]),
                  val_split=c(0.2),
               Lasso_par=c(0.001,0.01)),sample=1,confirm =FALSE,echo
=F)
   runs.sp[,2:5]
   ###b) ###### Ordering in the same way all grids
  runs.sp=runs.sp[order(runs.sp$flag_units,runs.sp$flag_Lasso_par),]

   runs.sp$grid_length=1:nrow(runs.sp)
   Parameters=data.frame(grid_length=runs.sp$grid_length,
metric_val_mse=runs.sp$metric_val_mse,flag_dropout1=runs.
sp$flag_dropout1,flag_units=runs.sp$flag_units, flag_batchsize1=runs.
sp$flag_batchsize1,epochs_completed=runs.sp$epochs_completed,
flag_learning_rate=runs.sp$flag_learning_rate, flag_Lasso_par=runs.
sp$flag_Lasso_par, flag_activation1=runs.sp$flag_activation1)
   Hyperpar=rbind(Hyperpar,data.frame(Parameters))
  }
  Hyperpar %>%
   group_by(grid_length) %>%
   summarise(val_mse=mean(metric_val_mse),
```

```
        dropout1=mean(flag_dropout1),
        units=mean(flag_units),
        batchsize1=mean(flag_batchsize1),
        learning_rate=mean(flag_learning_rate),
        Lasso_par=mean(flag_Lasso_par),
        epochs=mean( epochs_completed)) %>%
   select(grid_length,val_mse,dropout1,units,batchsize1,
learning_rate, Lasso_par,epochs) %>%
   mutate_if(is.numeric, funs(round(., 3))) %>%
   as.data.frame() -> Hyperpar_Opt
  ############Optimal hyperparameters###########
  Min=min(Hyperpar_Opt$val_mse)
  pos_opt=which(Hyperpar_Opt$val_mse==Min)
  pos_opt=pos_opt[1]
  Optimal_Hyper=Hyperpar_Opt[pos_opt,]
  #####Selecting the best hyperparameters
  Drop_O=Optimal_Hyper$dropout1
  Epoch_O=round(Optimal_Hyper$epochs,0)
  Units_O=round(Optimal_Hyper$units,0)
  activation_O=unique(Hyperpar$flag_activation1)
  batchsize_O=round(Optimal_Hyper$batchsize1,0)
  lr_O=Optimal_Hyper$learning_rate
  Lasso_par_O=Optimal_Hyper$Lasso_par

  print_dot_callback <- callback_lambda(
   on_epoch_end = function(epoch, logs) {
    if (epoch %% 20 == 0) cat("\n")
    cat(".")})

  ###########Refitting the model with the optimal values#############
  model_Sec<-keras_model_sequential()
  model_Sec %>%
   layer_dense(units =Units_O, kernel_regularizer=regularizer_l1
(Lasso_par_O),activation =activation_O, input_shape = c(dim
(X_trII)[2])) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, kernel_regularizer=regularizer_l1
(Lasso_par_O),activation =activation_O) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, kernel_regularizer=regularizer_l1
(Lasso_par_O),activation =activation_O) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =Units_O, kernel_regularizer=regularizer_l1
(Lasso_par_O),activation =activation_O) %>%
   layer_dropout(rate=Drop_O) %>%
   layer_dense(units =1, activation ="exponential")

  model_Sec %>% compile(
   loss = "poisson",
   optimizer = optimizer_adam(lr_O),
   metrics = c('mse'))
```

```
  ModelFinal <-model_Sec %>% fit(
   X_trn, y_trn,
   epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
   verbose=0,callbacks=list(print_dot_callback))

  ####e) Prediction of testing set ##########################
  predicted=model_Sec %>% predict(X_tst)
  Predicted=predicted
  Observed=y[tst_set]
  results<-rbind(results, data.frame(Position=tst_set,
                    Environment=CrossV$Environments[tst_set],
                    Partition=o,
                    Units=Units_O,
                    Epochs=Epoch_O,
               Observed=round(Observed, digits), #$response, digits),
                    Predicted=round(Predicted, digits),
                    Trait="Count"))
  cat("CV=",o,"\n")
 }
 results

 Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
 Pred_Summary

 Final_results=rbind(Final_results,data.frame(learn_val=Learn_val
[e],Pred_Summary))
}
Final_results
write.csv(Final_results,file="Appendix3_Count_Chapter12_modifiedv2.
csv")
```

# Appendix 4

R code for training a multi-trait normal outcome with three hidden layers

```
rm(list=ls())
library(BMTME)
library(tensorflow)
library(keras)
library(plyr) ####ply
library(tidyr)
library(dplyr)
library(tfruns)
options(bitmapType='cairo')

##########Set seed for reproducible results###################
use_session_with_seed(64)
```

```
################Loading the MaizeToy Data set###############
data("MaizeToy")
ls()
head(phenoMaizeToy)

##########Ordering the data ################################
phenoMaizeToy<-(phenoMaizeToy[order(phenoMaizeToy$Env,
phenoMaizeToy$Line),])
rownames(phenoMaizeToy)=1:nrow(phenoMaizeToy)
head(phenoMaizeToy,5)

################Design matrices################################
LG <- cholesky(genoMaizeToy)
ZG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line))
Z.G <- ZG %*%LG
Z.E <- model.matrix(~0 + as.factor(phenoMaizeToy$Env))
ZEG <- model.matrix(~0 + as.factor(phenoMaizeToy$Line):as.factor
(phenoMaizeToy$Env))
G2 <- kronecker(diag(length(unique(phenoMaizeToy$Env))), data.matrix
(genoMaizeToy))
LG2 <- cholesky(G2)
Z.EG <- ZEG %*% LG2

###########Selecting the response variable######################
Data_Pheno=phenoMaizeToy
Y <- as.matrix(Data_Pheno[, -c(1, 2)])

####Training testing sets using the BMTME package##############
pheno <- data.frame(GID =Data_Pheno[, 1], Env =Data_Pheno[, 2],
          Response =Data_Pheno[, 3])

#CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)
CrossV <- CV.RandomPart(pheno, NPartitions = 5, PTesting = 0.2,
set_seed = 123)

summary.BMTMECV <- function(results, information='compact',
digits=4, ...) {
# if (!inherits(object, "BMTMECV")) stop("This function only works for
objects of class 'BMTMECV'")
  results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(Pearson = cor(Predicted, Observed, use = 'pairwise.
complete.obs'),
  MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed))),
  MSE= mean((Observed-Predicted)^2)) %>%
  select(Environment, Trait, Partition, Pearson, MAAPE,MSE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Environment, Trait) %>%
  summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
```

```
  SE_Pearson = sd(Pearson, na.rm = T)/sqrt(n()), Pearson = mean
(Pearson, na.rm = T),
  SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>%
  select(Environment, Trait, Pearson, SE_Pearson, MAAPE, SE_MAAPE,
MSE, SE_MSE ) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
        complete = presum,
        extended = {
         finalSum$Partition <- 'All'
         presum$Partition <- as.character(presum$Partition)
         presum$SE_Pearson <- NA
         presum$SE_MAAPE <- NA
         presum$SE_MSE <- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

#######Final X and y=Height to use for training the model#############
tst_set=CrossV$CrossValidation_list[[1]]

#####X training and testing####
X=cbind(Z.E,Z.G)
dim(X)

y=(Data_Pheno[, 3:5])
y2=y

####Weights calculation
max_Dif_q1=max(quantile(y[,1],p=0.75)-quantile(y[,1],p=0.5),
quantile(y[,1],p=0.5)-quantile(y[,1],p=0.25))
max_Dif_q2=(max(quantile(y[,2],p=0.75)-quantile(y[,2],p=0.5),
quantile(y[,2],p=0.5)-quantile(y[,2],p=0.25)))
max_Dif_q3=(max(quantile(y[,3],p=0.75)-quantile(y[,3],p=0.5),
quantile(y[,3],p=0.5)-quantile(y[,3],p=0.25)))
w1=max_Dif_q1
w2=max_Dif_q1/max_Dif_q2
w3=max_Dif_q1/max_Dif_q3

###################Outer cross-validation######################
digits=4
n=dim(X)[1]
Names_Traits=colnames(Y)
results=data.frame()
t=1
```

```
for (o in 1:5){
### o=1
  tst_set=CrossV$CrossValidation_list[[o]]

 X_trn=(X[-tst_set,])
 X_tst=(X[tst_set,])
 y_trn=(y[-tst_set,])
 y_tst=(y[tst_set,])

################Inner cross-validation############################
  X_trII=X_trn
  y_trII=y_trn

  #####a) Grid search using the tuning_run() function of tfruns
package########
  runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_normal.R",
runs_dir = '_tuningE1',
           flags=list(dropout1= c(0,0.05),
                units = c(56,97),
                activation1=("relu"),
                batchsize1=c(22),
                Epoch1=c(1000),
                learning_rate=c(0.001),
              val_split=c(0.25)),sample=0.5,confirm =FALSE,echo =F)
  runs.sp[,23:27]

  ###b) Decreasing order of prediction performance of each combination
of the grid
  runs=runs.sp[order(runs.sp$metric_val_loss , decreasing = T), ]
  runs
  runs[,23:27]
  dim(runs)[1]

  ####c) Selecting the best combination of hyperparameters ####
  pos_opt=dim(runs)[1]
  opt_runs=runs[pos_opt,]

  ####d) Renaming the optimal hyperparametes
  Drop_O=opt_runs$flag_dropout1
  Epoch_O=opt_runs$epochs_completed
  Units_O=opt_runs$flag_units
  activation_O=opt_runs$flag_activation1
  batchsize_O=opt_runs$flag_batchsize1
  lr_O=opt_runs$flag_learning_rate

 ###########Refitting the model with the optimal values#############
 ### add covariates
 input <- layer_input(shape=dim(X_trn)[2],name="covars")

 ### add hidden layers
 base_model <- input  %>%
  layer_dense(units =Units_O, activation=activation_O) %>%
```

```
  layer_dropout(rate = Drop_O) %>%
  layer_dense(units =Units_O, activation=activation_O) %>%
  layer_dropout(rate = Drop_O) %>%
  layer_dense(units =Units_O, activation=activation_O) %>%
  layer_dropout(rate = Drop_O)

 # add output 1
 yhat1 <- base_model %>%
  layer_dense(units =1, name="response_1")

 # add output 2
 yhat2 <- base_model %>%
  layer_dense(units = 1, name="response_2")

 # add output 3
 yhat3 <- base_model %>%
  layer_dense(units = 1, name="response_3")

 # build multi-output model
 model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3)) %>%
  compile(optimizer =optimizer_adam(lr=lr_O),
      loss=list(response_1="mse",response_2="mse",
response_3="mse"),
      metrics=list(response_1="mse",response_2="mse",
response_3="mse"),
     loss_weights=list(response_1=w1,response_2=w2,response_3=w3))

 print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
   if (epoch %% 20 == 0) cat("\n")
   cat(".")
  })
 early_stop <- callback_early_stopping(monitor = c("val_loss"),
mode='min', patience =50)

 # fitting the model
 model_fit <- model %>%
  fit(x=X_trn,
    y=list(response_1=y_trn[,1],response_2=y_trn[,2],
response_3=y_trn[,3]),
    epochs=Epoch_O,
    batch_size =batchsize_O,
    verbose=0, callbacks = list(print_dot_callback))

 # predict values for test set
 Yhat<-predict(model,X_tst)%>%
  data.frame()%>%
  setNames(colnames(y_trn))
 YP=Yhat
```

```
 results<-rbind(results, data.frame(Position=tst_set,
                  Environment=CrossV$Environments[tst_set],
                  Partition=o,
                  Units=Units_O,
                  Epochs=Epoch_O,
                  Observed=round(y[tst_set,], digits), #$response,
digits),
                  Predicted=round(YP, digits)))
 cat("CV=",o,"\n")
}
results
nt=3

Pred_all_traits=data.frame()
for (i in 1:nt){
 #i=1
 pos_i_obs=5+i
 pos_i_pred=8+i
 results_i=results[,c(1:5,pos_i_obs,pos_i_pred)]
 results_i$Trait=Names_Traits[i]
 Names_results_i=colnames(results_i)
 colnames(results_i)=c(Names_results_i
[1:5],"Observed","Predicted","Trait")

 Pred_Summary=summary.BMTMECV(results=results_i, information =
'compact', digits = 4)
 Pred_Summary
 Pred_all_traits=rbind(Pred_all_traits,Pred_Summary)
}
Pred_all_traits
write.csv(Pred_all_traits,file="Multi-trait-predictions_Normal2.
csv")
```

## Appendix 5

R code for training a multi-trait mixed outcome with three hidden layers

```
rm(list=ls())
library(BMTME)
library(tensorflow)
library(keras)
library(caret)
library(plyr)
library(tidyr)
library(dplyr)
library(tfruns)
options(bitmapType='cairo')
```

```
##########Set seed for reproducible results###################
use_session_with_seed(64)

###########Loading the EYT_Toy data set#######################
load("Data_Toy_EYT.RData")

#############Genomic relationship matrix (GRM)################
Gg=data.matrix(G_Toy_EYT)
G=Gg

############Phenotypic data ##################################
Data_Pheno=Pheno_Toy_EYT

########Creating the design matrix of lines ##################
Z1G=model.matrix(~0+as.factor(Data_Pheno$GID))
L=t(chol(Gg))
Z1G=Z1G%*%L
ZE=model.matrix(~0+as.factor(Data_Pheno$Env))
Z2GE=model.matrix(~0+Z1G:as.factor(Data_Pheno$Env))
nCV=5

summary.BMTMECV <- function(results, information='compact',
digits=4, ...) {
 # if (!inherits(object, "BMTMECV")) stop("This function only works for
objects of class 'BMTMECV'")

 results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(Pearson = cor(Predicted, Observed, use = 'pairwise.
complete.obs'),
       MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed))),
       PCCC = 1-sum(Observed!=Predicted)/length(Observed)) %>%
  select(Environment, Trait, Partition, Pearson, MAAPE,PCCC) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Environment, Trait) %>%
  summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
       SE_Pearson = sd(Pearson, na.rm = T)/sqrt(n()), Pearson = mean
(Pearson, na.rm = T),
      SE_PCCC = sd(PCCC, na.rm = T)/sqrt(n()), PCCC = mean(PCCC, na.rm =
T)) %>%
  select(Environment, Trait, Pearson, SE_Pearson, MAAPE, SE_MAAPE,
PCCC, SE_PCCC ) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
        complete = presum,
        extended = {
```

```
        finalSum$Partition <- 'All'
        presum$Partition <- as.character(presum$Partition)
        presum$SE_Pearson <- NA
        presum$SE_MAAPE <- NA
        presum$SE_PCCC <- NA
        rbind(presum, finalSum)
       }
 )
 return(out)
}

############Selecting the response variable#######################
Y <- as.matrix(Data_Pheno[, -c(1, 2)])

########Training testing sets using the BMTME package###############
pheno <- data.frame(GID =Data_Pheno[, 1], Env =Data_Pheno[, 2],
          Response =Data_Pheno[, 3])

#CrossV <- CV.KFold(pheno, DataSetID = 'GID', K = 5, set_seed = 123)
CrossV <- CV.RandomPart(pheno, NPartitions =5, PTesting = 0.2, set_seed
= 123)

#####X training and testing####
X=cbind(ZE,Z1G,Z2GE)
dim(X)

y=Data_Pheno[, c(3,4,5,6)]
summary(y[,3])
y[,1]=y[,1]-1
y[,2]=y[,2]-1

############Outer cross-validation#######################
digits=4
n=dim(X)[1]
Names_Traits=colnames(y)
results=data.frame()
t=1
for (o in 1:5){
# o=1
 tst_set=CrossV$CrossValidation_list[[o]]
 X_trn=(X[-tst_set,])
 X_tst=(X[tst_set,])
 y_trn=(y[-tst_set,])
 y_tst=(y[tst_set,])

 y_trn[,1]=to_categorical(y_trn[,1], 3)
 y_tst[,1]=to_categorical(y_tst[,1], 3)

 ##################Inner cross-validation#########################
 X_trII=X_trn
 y_trII=y_trn
```

```
#####a) Grid search using the tuning_run() function of tfruns
package########
runs.sp<-tuning_run("Code_Tuning_With_Flags_MT_Mixed.R",runs_dir
= '_tuningE1',
          flags=list(dropout1= c(0,0.05),
                units = c(56,97),
                activation1=("relu"),
                batchsize1=c(30),
                Epoch1=c(1000),
                learning_rate=c(0.01),
                val_split=c(0.10)),sample=1,confirm =FALSE,echo =F)
runs.sp[,23:27]

###b) Decreasing order of prediction performance of each combination of
the grid
runs=runs.sp[order(runs.sp$metric_val_loss , decreasing = T), ]

####c) Selecting the best combination of hyperparameters ####
pos_opt=dim(runs)[1]
opt_runs=runs[pos_opt,]

####d) Renaming the optimal hyperparametes
Drop_O=opt_runs$flag_dropout1
Epoch_O=opt_runs$epochs_completed
Units_O=opt_runs$flag_units
activation_O=opt_runs$flag_activation1
batchsize_O=opt_runs$flag_batchsize1
lr_O=opt_runs$flag_learning_rate

###########Refitting the model with the optimal values################
### add covariates
input <- layer_input(shape=dim(X_trn)[2],name="covars")

### add hidden layers
base_model <- input %>%
 layer_dense(units =Units_O, activation=activation_O) %>%
 layer_dropout(rate = Drop_O) %>%
 layer_dense(units =Units_O, activation=activation_O) %>%
 layer_dropout(rate = Drop_O) %>%
 layer_dense(units =Units_O, activation=activation_O) %>%
 layer_dropout(rate = Drop_O)

# add output 1
yhat1 <- base_model %>%
 layer_dense(units =3,activation="softmax", name="response_1")

# add output 2
yhat2 <- base_model %>%
 layer_dense(units = 1, activation="exponential",name="response_2")
```

```
# add output 3
yhat3 <- base_model %>%
  layer_dense(units = 1, name="response_3")

# add output 4
yhat4 <- base_model %>%
  layer_dense(units = 1, activation="sigmoid", name="response_4")

# build multi-output model
model <- keras_model(input,list(response_1=yhat1,response_2=yhat2,
response_3=yhat3,response_4=yhat4)) %>%
  compile(optimizer =optimizer_adam(lr=lr_O),
      loss=list(response_1="categorical_crossentropy",
response_2="mse",response_3="mse",
response_4="binary_crossentropy"),
      metrics=list(response_1="accuracy",response_2="mse",
response_3="mse",response_4="accuracy"),
      loss_weights=c(response_1=1,response_2=1,response_3=1,
response_4=1))

print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
   if (epoch %% 20 == 0) cat("\n")
   cat(".")
  })
early_stop <- callback_early_stopping(monitor = c("val_loss"),
mode='min', patience =50)
#,early_stop

# fitting the model
model_fit <- model %>%
  fit(x=X_trn,
    y=list(response_1=y_trn[,1],response_2=y_trn[,2],
response_3=y_trn[,3],response_4=y_trn[,4]),
    epochs=Epoch_O,
    batch_size =batchsize_O,
    verbose=0, callbacks = list(print_dot_callback))

# predict values for test set
Yhat<-predict(model,X_tst)%>%
  data.frame()%>%
  setNames(colnames(y_trn))
YP=matrix(NA,ncol=ncol(y),nrow=nrow(Yhat))
head(Yhat)
P_T1=(apply(data.matrix(Yhat[,1:3]),1,which.max)-1)
P_T4=ifelse(c(Yhat[,6])>0.5,1,0)
YP[,1]=P_T1
YP[,2]=c(Yhat[,4])
YP[,3]=c(Yhat[,5])
YP[,4]=P_T4
head(YP)
```

```
 results<-rbind(results, data.frame(Position=tst_set,
                    Environment=CrossV$Environments[tst_set],
                    Partition=o,
                    Units=Units_O,
                    Epochs=Epoch_O,
                    Observed=round(y[tst_set,], digits), #$response,
digits),
                    Predicted=round(YP, digits)))
 cat("CV=",o,"\n")
}
results
nt=4

Pred_all_traits=data.frame()
for (i in 1:nt){
 #i=1
 Names_Traits
 pos_i_obs=5+i
 pos_i_pred=9+i
 results_i=results[,c(1:5,pos_i_obs,pos_i_pred)]
 results_i$Trait=Names_Traits[i]
 Names_results_i=colnames(results_i)
 colnames(results_i)=c(Names_results_i
[1:5],"Observed","Predicted","Trait")

 Pred_Summary=summary.BMTMECV(results=results_i, information =
'compact', digits = 4)
 Pred_Summary
 Pred_all_traits=rbind(Pred_all_traits,data.frame
(Trait=Names_Traits[i],Pred_Summary))
}
Pred_all_traits
Res_Sum=Pred_all_traits[,-c(3:5)]
Res_Sum
write.csv(Res_Sum,file="Multi-trait-predictions_Mixed5.csv")
```

## References

Allaire JJ (2018) Tfruns: training run tools for 'tensorflow'. https://CRAN.R-project.org/package=tfruns

Allaire JJ, Chollet F (2019) Keras: R interface to Keras'. https://CRAN.R-project.org/package=keras

Calus MP, Veerkamp RF (2011) Accuracy of multi-trait genomic selection using different methods. Genetics Selection Evolution 43(1):26. https://doi.org/10.1186/1297-9686-43-26

Castro AFNM, Castro RV, Oliveira CAO, Lima JE, Santos RC, Pereira BLC, Alves ICN (2013) Multivariate analysis for the selection of eucalyptus clones destined for charcoal production. Pesq Agrop Brasileira 48(6):627–635

Chollet F, Allaire JJ (2017) Deep learning with R. Manning Publications, Manning Early Access Program (MEA), 1st edn

He D, Kuhn D, Parida L (2016) Novel applications of multitask learning and multiple output regression to multiple genetic trait prediction. Bioinformatics 32(12):i37–i43. https://doi.org/10.1093/bioinformatics/btw249

Huang M, Chen L, Chen Z (2015) Diallel analysis of combining ability and heterosis for yield and yield components in rice by using positive loci. Euphytica 205(1):37–50

Ioffe S, Szegedy C (2015) Batch normalization: accelerating deep network training by reducing internal covariate shift. arXiv Preprint arXiv:1502.03167

Jia Y, Jannink J-L (2012) Multiple-trait genomic selection methods increase genetic value prediction accuracy. Genetics 192(4):1513–1522. https://doi.org/10.1534/genetics.112.144246

Jiang J, Zhang Q, Ma L, Li J, Wang Z, Liu JF (2015) Joint prediction of multiple quantitative traits using a Bayesian multivariate antedependence model. Heredity 115(1):29–36

LeCun Y, Bottou L, Orr G, Muller K (1998) Efficient backprop. In: Orr G, Muller K (eds) Neural networks: tricks of the trade. Springer

Montesinos-López OA, Montesinos-López A, Crossa J, Toledo F, Pérez-Hernández O, Eskridge KM, Rutkoski J (2016) A genomic Bayesian multi-trait and multi-environment model. G3: Genes, Genomes, Genetics 6(9):2725–2744

Montesinos-López A, Montesinos-López OA, Gianola D, Crossa J, Hernández-Suárez CM (2018a) Multivariate Bayesian analysis of on-farm trials with multiple-trait and multiple-environment data. Agron J 111(6):2658–2669. https://doi.org/10.2134/agronj2018.06.0362

Montesinos-López OA, Montesinos-López A, Gianola D, Crossa J, Hernández-Suárez CM (2018b) Multi-trait, multi-environment deep learning modeling for genomic-enabled prediction of plant. G3: Genes, Genomes, Genetics 8(12):3829–3840

Montesinos-López A, Montesinos-López OA, Gianola D, Crossa J, Hernández-Suárez CM (2018c) Multi-environment genomic prediction of plant traits using deep learners with a dense architecture. G3: Genes, Genomes, Genetics 8(12):3813–3828. https://doi.org/10.1534/g3.118.200740

Montesinos-López OA, Martín-Vallejo J, Crossa J, Gianola D, Hernández-Suárez CM, Montesinos-López A, Juliana P, Singh R (2019) New deep learning genomic prediction model for multi-traits with mixed binary, ordinal, and continuous phenotypes. G3: Genes, Genomes, Genetics 9(5):1545–1556

R Core Team (2019) R: a language and environment for statistical computing. R Foundation for Statistical Computing, Vienna. ISBN 3-900051-07-0. http://www.R-project.org/

Schulthess AW, Zhao Y, Longin CFH, Reif JC (2017) Advantages and limitations of multiple-trait genomic prediction for Fusarium head blight severity in hybrid wheat (*Triticum aestivum* L.). Theor Appl Genet 131(3):685–701. https://doi.org/10.1007/s00122-017-3029-7

Wiesler S, Ney H (2011) A convergence analysis of log-linear training. In: Shawe-Taylor J, Zemel RS, Bartlett P, Pereira FCN, Weinberger KQ (eds), Advances in neural information processing systems, vol 24. Granada, pp 657–665

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 13
# Convolutional Neural Networks

## 13.1 The Importance of Convolutional Neural Networks

Convolutional neural networks (CNNs or ConvNets) are a specialized form of deep neural networks for analyzing input data that contain some form of spatial structure (Goodfellow et al. 2016). CNNs are primarily used to solve problems of computer vision (such as self-driving cars, robotics, drones, security, medical diagnoses, treatments for the visually impaired, etc.) using images as input, since it takes advantage of the grid structure of the data. CNNs work by first deriving low-level representations, local edges, and points, and then composing higher level representations, overall shapes, and contours. The name of these deep neural networks is due to the fact that they apply convolutions, a type of linear mathematical operation. CNNs use convolution in at least one of their layers, instead of a general matrix multiplication, as do the feedforward deep neural networks studied in the previous chapters. Compared to other classification algorithms, the preprocessing required in a CNN is considerably lower. For these reasons, CNNs have been used to identify faces, individuals, tumors, objects, street signs, etc. Thanks to successful commercial applications of this type of deep neural networks, the term "deep learning" was coined and it is the most popular name given to artificial neural networks with more than one hidden layer. Also, their popularity is attributed in part to the fact that the best CNNs today reach or exceed human-level performance, a feat considered impossible by most experts in computer vision only a few decades back.

The inspiration for CNNs came from a cat's visual cortex (Hubel and Wiesel 1959), which has small regions of cells that are sensitive to specific regions in the visual field; this means that when specific areas of the visual field are excited, then those cells in the visual cortex will be activated as well. In addition, it is important to point out that the excited cells depend on the shape and orientation of the objects in the visual field. This implies that horizontal edges excite some neurons, while vertical edges excite other neurons (Hubel and Wiesel 1959).

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_13

Successful applications of CNNs are not limited to the examples given above, since they have also been used for solving problems related to natural language processing using digitalized text as input, but they are also a powerful tool for analyzing words as discrete textual units (Patterson and Gibson 2017). In addition, they have been applied (a) to sound data when they are represented as a spectrogram, (b) for predicting time series (one-dimensional data) that use historical information as input, and (c) for sentiment analysis. In general, CNNs are very popular for dealing with input data that have some structure (as image data) (one, two, three, or more dimensions), video data (three dimensions), genetic data, etc.

CNNs are deep neural networks that most efficiently take advantage of both the extraordinary computing power and very large data sets that are available in many fields nowadays. Also, CNNs bypass the need to explicitly define which independent variables (inputs) should be included or selected for the analysis, since they optimize a complete end-to-end process to map data samples to outputs that are consistent with the large, labeled data sets used for training a deep neural network (Wang et al. 2019). Empirical evidence shows that CNNs are a powerful tool for image analysis using many layers of filters. In addition, the improvements in accuracy have been so extraordinarily large in the last few years that they have changed the research landscape in this area. Low-level image features (e.g., detecting horizontal or vertical edges, bright points, or color variations) are captured by the first filters and subsequent layers capture increasingly complicated combinations of earlier features, and when the training data set is large enough, CNNs most of the time outperform other machine learning algorithms (Wang et al. 2019). In addition, CNNs are expected to outperform feedforward deep neural networks since they exploit part of the correlation between nearby pixel positions. For the classification problem of attempting to label which of 1000 different objects are in an image, results have improved from 84.6% (in 2012) to 96.4% in 2015. For this reason, CNNs are being applied to complex tasks in plant genomics like (a) root and shoot feature identification (Pound et al. 2017), (b) leaf counting (Dobrescu et al. 2017; Giuffrida et al. 2018), (c) classification of biotic and abiotic stresses (Ghosal et al. 2018), (d) counting seeds per pot (Uzal et al. 2018), (e) detection of wheat spikes (Hasan et al. 2018), and (f) estimating plant morphology and developmental stages (Wang et al. 2019). These examples show their great potential for accurately estimating plant phenotypes directly from images.

This chapter provides the basic concepts and definitions related to CNNs, as well as the main elements to implement this type of deep learning models. Finally, some examples of CNNs in the context of genomic prediction are provided.

## 13.2 Tensors

Nowadays, all current machine learning algorithms use tensors as their basic data structure. Tensors are of paramount importance in machine learning and are so fundamental that Google's framework for machine learning is called TensorFlow. For this reason, here we define this concept in detail.

By tensors we understand a generalization of vectors and matrices to an arbitrary number of dimensions.

**0D tensors** These tensors are scalars represented by only one number; for this reason, we call them zero-dimensional tensors (0D tensors) (Chollet and Allaire 2017). For example, a 0D tensor is when you measure the temperature on a specific day in the city of Colima, Mexico, which can be 30 °C. Another example can be the tons per hectare produced by a hybrid in a specific year or location, which can be 8 tons per hectare.

**1D tensors** These tensors are vectors as defined in linear algebra, that is, a one-dimensional array of numbers; for this reason, they are called one-dimensional tensors (1D tensors) (Chollet and Allaire 2017). This type of tensor has exactly one axis. For example, the grain yield of the same hybrid measured in five environments is a 1D tensor; in this case, it is equal to GY = c(9,7.8,6.4,8.2,5.9). Another example of a 1D tensor is the temperature measured over the last 7 days (1 week) in Colima, Mexico, that is, Tem = c(28, 29, 30, 32, 33, 34, 34.5).

**2D tensors** These tensors are arrays of numbers in two dimensions (rows and columns); for this reason, they are called two-dimensional tensors (2D tensors) or matrix arrays. An example of a 2D tensor can be the grain yield measured over 3 years in five environments for the same hybrid (Table 13.1). Now this tensor has two axes.

Another example is the information of an image taken in grayscale, since an image is nothing but a matrix of pixel values. For example, an image in grayscale can be represented in two dimensions (height and width) and assuming that the image's height = 5 pixels and width = 5 pixels, the 2D tensor is given in Table 13.2.

Most of the data sets used in this book are 2D tensors, since the samples (or observations) are in the rows (first axis) and the variables measured for each observation are in the columns (second axis).

**3D tensors** These tensors are arrays of numbers in three dimensions; for this reason, they are called three-dimensional tensors (3D tensors). An example of a 3D tensor is the example given above assuming the trait was measured in 3 years, five environments, and three traits (Fig. 13.1).

**Table 13.1** Example of a 2D tensor

| Year | Env1 | Env2 | Env3 | Env4 | Env5 |
|---|---|---|---|---|---|
| Year 1 | 9 | 7.8 | 6.4 | 8.2 | 5.9 |
| Year 2 | 7 | 6.8 | 6 | 7.5 | 5.5 |
| Year 3 | 8.0 | 7.2 | 6.3 | 7.8 | 5.7 |

**Table 13.2** Example of a 2D tensor of an image in black and white, 5 pixels in height and 5 pixels in width

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

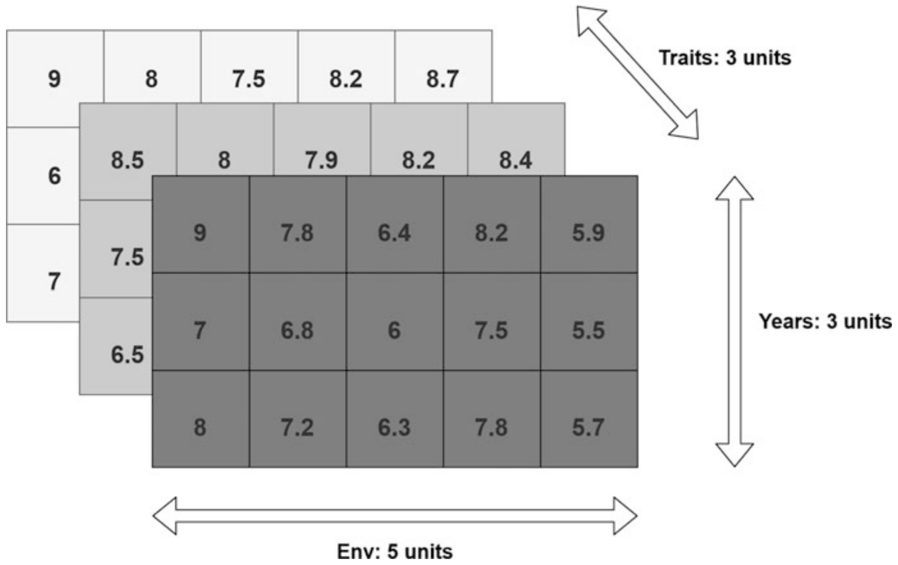



**Fig. 13.1** A 3D tensor for grain yield measured in five environments, 3 years, and three traits

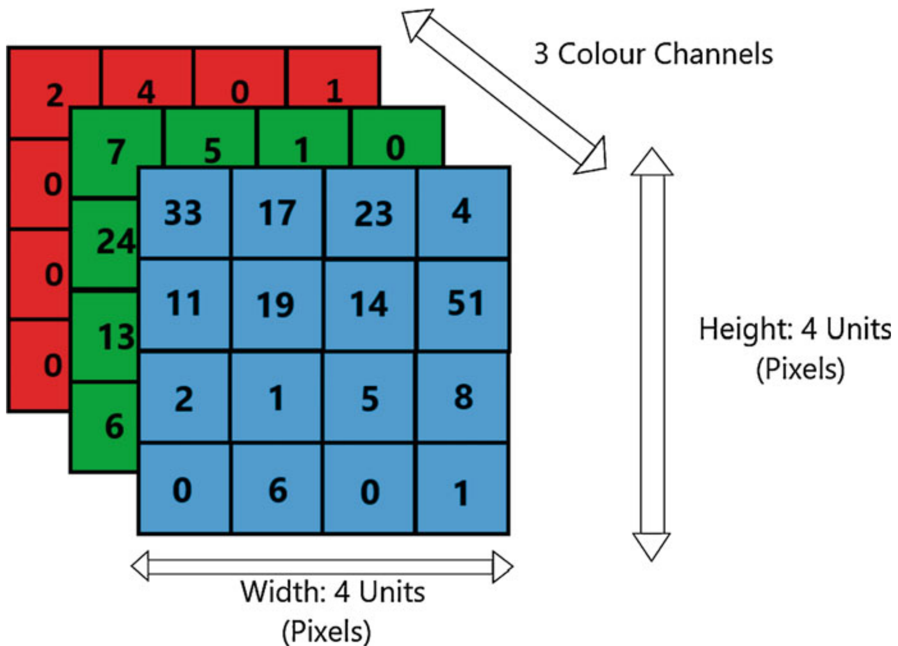



**Fig. 13.2** A 3D tensor of a Red-Green-Blue (RGB) image of a dimension of $4 \times 4 \times 3$

Other examples are color images in 3D, which, in addition to height and width given in pixels, also have another axis for depth. This can be appreciated in Fig. 13.2. There's no problem in understanding the height and width of an image, but the depth

is not clear. For this reason, the depth is defined as the axis that will encode the type of colors. For example, images called Red-Green-Blue (RGB) have a depth of 3, since they encode the three types of primary colors in this axis (see Fig. 13.2).

**4D tensors** This array is obtained when we pack a 3D tensor in a new array. One example is adding one more dimension to the example of measuring grain yield in 3 years, five environments, and three types of traits. This dimension can be the samples (Lines), which means that we are measuring a 3D tensor for each line. The same idea can be used for the example of the 3D tensor of an RGB image, but assuming that a color image (RGB) is measured on several samples (plants or objects), which transforms the information of a 3D tensor to a 4D tensor.

For practical purposes, in R we should use the array_reshape() function to reshape a tensor. As an illustration, first we will use the data in Table 13.1 corresponding to a 2D tensor and we put this information in the following matrix:

```
GY=matrix(c
(9,7.8,6.4,8.2,5.9,7,6.8,6,7.5,5.5,8.0,7.2,6.3,7.8,5.7), ncol=5,
byrow=T)
GY
> GY
     [,1] [,2] [,3]  [,4] [,5]
[1,]   9   7.8  6.4   8.2  5.9
[2,]   7   6.8  6.0   7.5  5.5
[3,]   8   7.2  6.3   7.8  5.7
```

Now I will reshape the GY 2D tensor to a 1D tensor.

```
GY1=array_reshape(GY)
```

This produces a column vector that has 15 elements, a 1D tensor, and the transpose of this 1D tensor is equal to

```
GY1=array_reshape(GY, dim=c(15,1))
> t(GY1)
 [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]  [,12]  [,13] [,14] [,15]
 [1,]   9    7.8  6.4  8.2  5.9  7   6.8  6    7.5  5.5 8   7.2    6.3   7.8   5.7
```

Now the GY 2D tensor will be reshaped to a 3D tensor of dimension $1 \times 5 \times 3$; the R code is

```
GY2=array_reshape(GY, dim=c(1,5,3))
```

That output is a 3D tensor

```
> GY2
, , 1
     [,1] [,2] [,3] [,4] [,5]
[1,]   9   8.2  6.8  5.5  6.3
```

```
, , 2
     [,1] [,2] [,3] [,4] [,5]
[1,]  7.8  5.9    6    8  7.8
, , 3
     [,1]  [,2] [,3] [,4] [,5]
[1,]  6.4     7  7.5  7.2  5.7
```

Where , ,1; , , 2; and , , 3 are the depths 1, 2, and 3 of the 3D tensor, and for each, a vector of dimension $1 \times 5$ is given as output.

Finally, we will reshape GY2, the 3D tensor to a 2D tensor of dimension $15 \times 3$.

```
GY3 = array_reshape(GY2, dim = c(5,3))
```

That produces as output

```
> GY3
     [,1] [,2] [,3]
[1,]  9.0  7.8 6.4
[2,]  8.2  5.9 7.0
[3,]  6.8  6.0 7.5
[4,]  5.5  8.0 7.2
[5,]  6.3  7.8 5.7
```

In similar fashion, you can reshape the last 2D tensor to a 4D tensor of dimension $5 \times 1 \times 1 \times 3$, using the following R code:

```
GY4 = array_reshape(GY3, dim = c(5,1,1,3))
```

That produces as output each of the components of the 4D tensor.

```
> GY4
, , 1, 1
      [,1]
[1,]  9.0
[2,]  8.2
[3,]  6.8
[4,]  5.5
[5,]  6.3

, , 1, 2
      [,1]
[1,]  7.8
[2,]  5.9
[3,]  6.0
[4,]  8.0
[5,]  7.8

, , 1, 3
     [,1]
[1,]  6.4
[2,]  7.0
```

```
[3,] 7.5
[4,] 7.2
[5,] 5.7
```

The output is in three blocks, and each block contains a 2D tensor of dimension $5 \times 1$. The last two numbers of the name of each block refer to the positions in dimensions 3 and 4, respectively, where for all blocks, the third position has a 1, since it only has one dimension, but dimension 4 (four position) contains numbers 1, 2, and 3, since 3 components form this dimension.

## 13.3 Convolution

Convolution comes from a Latin word meaning "to convolve" which means to roll together; it is a mathematical operation that merges two sets of information. Mathematically, convolution is a mathematical operation that is performed on two functions to produce a third one that is usually interpreted as a modified (filtered) version of one of the original functions (Berzal 2018). The convolution of the functions $f$ and $g$, which are usually denoted by an * or a $\star$, is defined as the integral of the product of two functions after one of them is reflected and moved:

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(t)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(t)d\tau$$

In digital signal processing, when we use discrete signals, the previous integral becomes a summation (Berzal 2018):

$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} f[g][n-m] = \sum_{m=-\infty}^{\infty} f[n-m]g[m]$$

Normally, one of the convolution operands is the signal that we want to process, $x[n]$, and the other corresponds to the filter, $h[n]$, with which we process the signal. When the filter is finite and defined only in the domain $\{0, 1, ..., K-1\}$, the convolution operation consists of, for each value of the signal, performing K multiplications and $K-1$ sums (Berzal 2018):

$$(x \star h)[n] = \sum_{k=0}^{K-1} h[k] \times [n-k]$$

According to Berzal (2018), the convolution operation, which so far we have defined on functions of a variable, can be easily extended to the multidimensional case. In the case of discrete signals defined on two variables and those that apply a filter of size $K_1 \times K_2$, the convolution is calculated using the following expression:

$$(x \star h)[n_1, n_2] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} h[k_1, k_2] \times [n_1 - k_1, n_2 - k_2]$$

In digital image processing, in which the variables $[n_1, n_2]$ correspond to the coordinates $[x, y]$ of the pixels of an image, the minus sign that appears in the definition of the convolution operation is usually replaced by a plus sign, so that the convolution is calculated as (Berzal 2018):

$$(x \star h)[x, y] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} h[k_1, k_2] \times [x + k_1, y + k_2]$$

Technically, it is not a convolution, but a similar operation called cross-correlation. In the discrete case for real signals, the only difference is that the filter used $h[x, y]$ appears reflected with respect to the formal definition of convolution (Berzal 2018). Given that the difference is minimal, in many cases we talk about convolution. The input to a convolution is a raw data or a feature map output from another convolution (Patterson and Gibson 2017). Finally, we can say that the convolution operation is a fancy kind of multiplication that is key in signal processing.

To better understand what the convolution operation is, let's use a simple example. Suppose we have a gray color image captured by a camera. This image, $4 \times 4$ pixels, can be represented by a matrix of dimension $4 \times 4$ (left of Fig. 13.3) to which we apply a filter (middle part of Fig. 13.3) of dimension $2 \times 2$. Then a dot product operation between each patch (square matrix of the same size as the filter) of the original image and the filter is performed that produces the output given on the right, where we can see that the first patch starts in the upper left-hand corner of the underlying image, and we move the filter across the image step by step until it reaches the lower right-hand corner. At each step, the filter is multiplied by the input data values within its bounds, creating a single entry in the output feature map (Patterson and Gibson 2017). We end up with a convolved matrix of dimension $3 \times 3$, of lower dimension than the original matrix. The size of the step in which we move the filter across the image is called the stride; this means that you can move the filter to the right one, two, three, etc., steps at a time.

The convolutional operation in Fig. 13.3 is not performed on one pixel at a time, but by taking square patches of pixels that are then passed through a filter. The filter is a square matrix of smaller dimensions than the original image and is the same size as the patch. The filter is called a kernel (a key word in support vector machine but with a different meaning) whose job is to find patterns in the pixels of the image. If the patch matrix and the filter have low values in the same positions, the dot products will be small; otherwise, they will be high. You can use a matrix of dimension $5 \times 5$ or $7 \times 7$ as a kernel, but most of the time the dimension of the kernel is a $3 \times 3$ matrix.

The output matrix resulting from the convolutional operation is called the *activation map*. The number of columns in this matrix is called the width and depends on

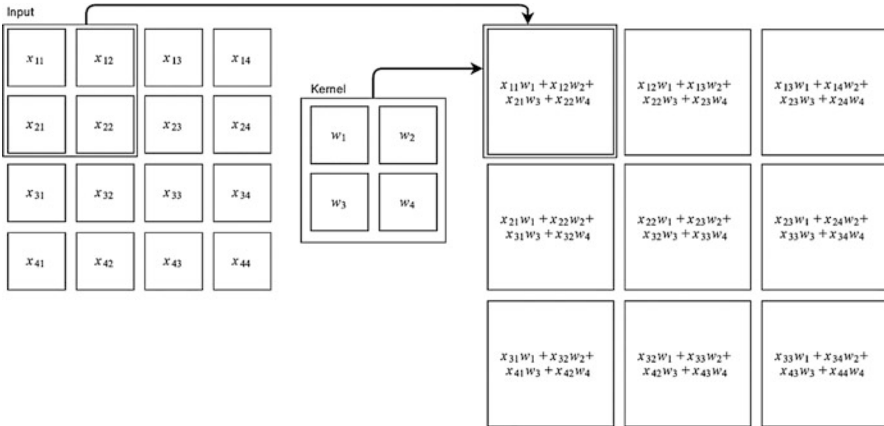



**Fig. 13.3** Example of a 2D convolution with a kernel (filter) of dimension 2 × 2 and stride of 1

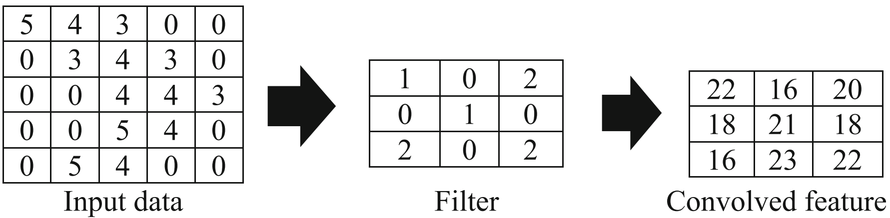



**Fig. 13.4** Example of a convolution operation with a stride of 1

the step size that the filter takes to traverse the underlying image. Larger strides (step size) lead to fewer steps and result in a smaller activation map matrix. The power of convolutional neural networks is related to the convolutional operation, since the larger the stride and the filter, the smaller the dimension of the *activation map* matrix produced, which considerably reduces the computational resources required.

Figure 13.3 shows that the input image has height ($L_{h0} = 4$), width ($L_{w0} = 4$), and depth ($L_{d0} = 1$), and the filters have height ($F_{h0} = 2$), width ($F_{w0} = 2$), and depth ($F_{d0} = 1$). Assuming the use of a stride ($S_0 = 1$), the output matrix is also called an activation map. After performing the convolutional operation, the input image has height $L_{h1} = L_{h0} - F_{h0} + 1 = 4 - 2 + 1 = 3$, width $L_{w1} = L_{w0} - F_{w0} + 1 = 4 - 2 + 1 = 3$, and depth $L_{d1} = L_{d0} = F_{d0} = 1$ (this did not change). In general, the height, width, and depth of the activation map, with a stride larger than 1 in the *lth* layer, can be calculated as $L_{h(l+1)} = (L_{h(l)} - F_{h(l)})/S_{(l)} + 1$, $L_{w(l+1)} = (L_{w(l)} - F_{w(l)})/S_{(l)} + 1$, and $L_{d(l+1)} = L_{d(l)} = F_{d(l)}$, respectively.

To completely understand the convolutional operation, below we provide other examples.

The input image in Fig. 13.4 has height ($L_{h0} = 5$), width ($L_{w0} = 5$), and depth ($L_{d0} = 1$), and the filters have height ($F_{h0} = 3$), width ($F_{w0} = 3$), and depth ($F_{d0} = 1$)

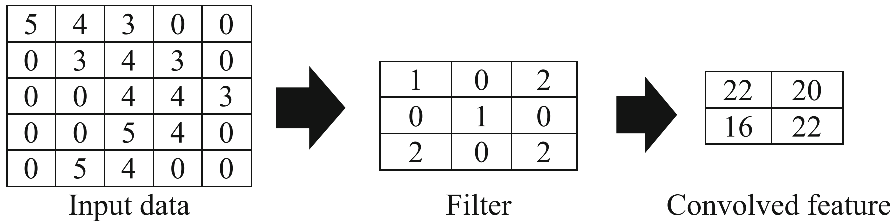



**Fig. 13.5** Example of a convolution operation with a stride of 2

and since a stride size of 1 will be used, the activation map, after performing the convolutional operation, has height $L_{h1} = L_{h0} - F_{h0} + 1 = 5 - 3 + 1 = 3$, width $L_{w1} = L_{w0} - F_{w0} + 1 = 5 - 3 + 1 = 3$, and depth $L_{d1} = L_{d0} = F_{d0} = 1$ (this did not change).

The input image in Fig. 13.5 has height ($L_{h0} = 5$), width ($L_{w0} = 5$), and depth ($L_{d0} = 1$), and the filter has height ($F_{h0} = 3$), width ($F_{w0} = 3$), and depth ($F_{d0} = 1$), but now using a stride of 2 ($S_0 = 2$). Therefore, the activation map, after performing the convolutional operation, has height $L_{h1} = (L_{h0} - F_{h0})/S_0 + 1 = (5 - 3)/2 + 1 = 2$, width $L_{w1} = (L_{w0} - F_{w0})/S_0 + 1 = (5 - 3)/2 + 1 = 2$, and depth $L_{d1} = L_{d0} = F_{d0} = 1$ (this did not change).

## 13.4 Pooling

Pooling is a mathematical operation that is also required in CNNs. A pooling operation replaces the output of the convolution operation at a certain location with summary statistics of the nearby outputs. The pooling operation is called downsampling or subsampling in machine learning. The two most popular pooling operations are the max pooling and the average pooling and, as in convolution, this process is applied one patch at a time. Max pooling performs dimensional reduction and de-noising, while average pooling mostly performs dimensional reduction. The max pooling operation summarizes the input as the maximum within a rectangular neighborhood, but does not introduce any new parameter to the CNN, with the advantage that the total number of parameters in the model is reduced considerably due to this operation. For example, assuming that Fig. 13.6 is the input (which can be raw information of the image or the convolved information of an image) and we apply the max pooling operation with a filter (kernel) of dimension $2 \times 2$, the max operation takes the largest of each of the four numbers in the filter area, and the process of pooling with the filter of dimension $2 \times 2$ starts in the upper left-hand corner of the input image; since we use a step of one, we move the filter across the image in steps of one until we reach the lower right-hand corner. At each step, for the four elements within the filter bounds, we get the largest value, with which we create the output matrix. The first output value for the first four values (in the upper left-hand corner) is 7 since this is the maximum value of the four elements that conform

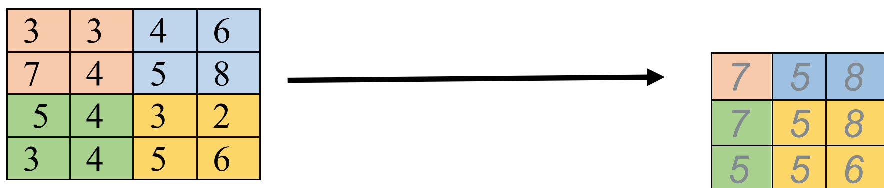



**Fig. 13.6** Max pooling with $2 \times 2$ filters and stride of 1

to the bounds of the filter (3, 3, 7, 4). The second value (after moving the filter one column to the right) is 5 since after the first slide of the filter, this is the maximum value that corresponds to the max of 3, 4, 4, and 5, while the last value is 6, which corresponds to a max of 3, 2, 5, and 6 in the last slice (lower right-hand corner). In this case, the size of the output matrix using a stride of 1 is 3, which is smaller than the original input matrix. By doing pooling after convolution, we threw away some information; since the network does not care about the exact location of a pattern, it is enough to know whether the pattern is present or not.

In general, the output after applying the max pooling or average pooling operation depends on the size of the small grid region of size $P_{h(l)} \times P_{w(l)}$ and the stride size $S_{(l)}$. For this reason, the height, width, and depth of the activation map, after applying the pooling operation with a stride larger than 1 in the *lth* layer, can be calculated as $L_{h(l+1)} = (L_{h(l)} - P_{h(l)})/S_{(l)} + 1$, $L_{w(l+1)} = (L_{w(l)} - P_{w(l)})/S_{(l)} + 1$, and $L_{d(l+1)} = L_{d(l)}$, respectively. For example, Fig. 13.6 contains an image of height ($L_{h0} = 4$), width ($L_{w0} = 4$), and depth ($L_{d0} = 1$), and using filters with height ($P_{h0} = 2$), width ($P_{w0} = 2$), depth ($P_{d0} = 1$), and a stride ($S_0 = 1$), the output activation map, after performing the max pooling operation, has height $L_{h1} = (L_{h0} - P_{h0})/S_0 + 1 = 4 - 2 + 1 = 3$, width $L_{w1} = (L_{w0} - P_{w0})/S_0 + 1 = 4 - 2 + 1 = 3$, and depth $L_{d1} = L_{d0} = 1$ (this did not change).

However, if we apply a stride of 2 to the same input of Fig. 13.6, the output matrix is of dimension $2 \times 2$, since the output activation map after performing the max pooling operation has height $L_{h1} = (L_{h0} - P_{h0})/S_0 + 1 = (4 - 2)/2 + 1 = 2$, width $L_{w1} = (L_{w0} - P_{w0})/S_0 + 1 = (4 - 2)/2 + 1 = 2$, and depth $L_{d1} = L_{d0} = 1$ (this did not change), that is, the dimension of the output matrix after pooling is halved (Fig. 13.7). A stride of 2 means that the filter is moved to the right two columns at a time.

Figure 13.8 illustrates, for the same input image given in the two previous figures, the average pooling with a $2 \times 2$ filter (kernel) and a stride of 1; the only difference is that now, instead of getting the maximum of each of the four values, the average is calculated in the bounds of the filter. For the first four values corresponding to the upper left-hand corner of Fig. 13.8, we calculated the average of the four values (3, 3, 7, 4) and we got $17/4 = 4.25$. The average of the four values corresponding to the remaining slices to the right of the filter from the upper left-hand corner to the lower right-hand corner of the input image was calculated in the same fashion.

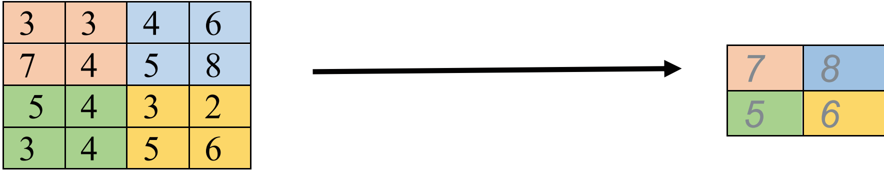



**Fig. 13.7** Max pooling with 2 × 2 filters and stride of 2

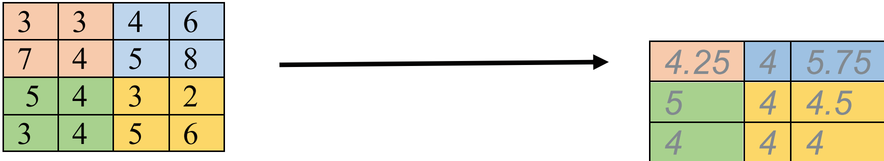



**Fig. 13.8** Average pooling with 2 × 2 filters and stride of 1

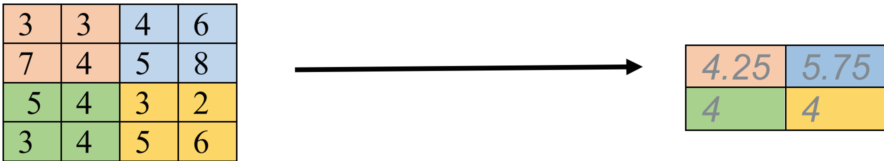



**Fig. 13.9** Average pooling with 2 × 2 filters and stride of 2

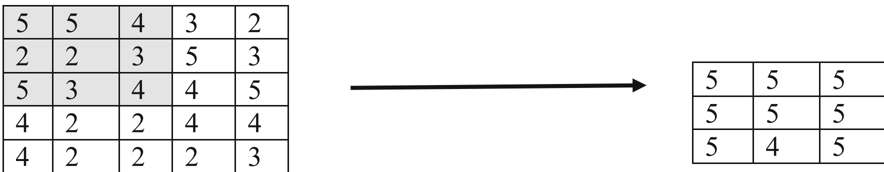



**Fig. 13.10** Max pooling with 3 × 3 filters and stride of 1

Figure 13.9 illustrates average pooling with a filter of size 2 × 2 and a stride of 2, where the output is part of the output of Fig. 13.8, but without the elements in column 2 and row 2, due to the fact that the stride used in Fig. 13.9 is 2.

The size of the filter is not restricted to a dimension of 2 × 2; this can be of any size but as we pointed out above, the most common filter size is dimension 2 × 2, but for example, filters of size 6 × 6 or 8 × 8 or any other filter size can be applied. Figure 13.10 illustrates the use of a filter of size 3 × 3 with the max pooling operation. In this case, for the first nine values that correspond to the bounds of the filter, the maximum value is 5. The image has height ($L_{h0} = 5$), width ($L_{w0} = 5$), and depth ($L_{d0} = 1$), using filters with height ($P_{h0} = 3$), width ($P_{w0} = 3$), depth ($P_{d0} = 1$), and a stride ($S_0 = 1$). Therefore, the output activation map after performing the max pooling operation has height $L_{h1} = (L_{h0} - P_{h0})/S_0 + 1 = 5 - 3 + 1 = 3$, width $L_{w1} = (L_{w0} - P_{w0})/S_0 + 1 = 5 - 3 + 1 = 3$, and depth $L_{d1} = L_{d0} = 1$ (this did not change).

The pooling operation is done at the level of each activation map, whereas the convolutional operation simultaneously uses all feature maps in combination with a filter to produce a single feature value; independent pooling operates on each feature map to produce other feature maps, which means that pooling does not change the number of feature maps. In simple terms, the depth of the layer created by pooling is the same as the depth of the layer on which the pooling operation was performed. Among the many pooling operations, the max pooling operation is the most popular downsampling operation and the second is the average pooling operation. Pooling layers that consist of layers in neural networks that apply pooling operations to the input information are commonly inserted between successive convolutional layers. Convolutional layers followed by pooling layers are applied to progressively reduce the spatial size (width and height) of the data representation (Patterson and Gibson 2017). Besides reducing the size of the representation, pooling layers also help to control overfitting. When we have input information in three dimensions (3D tensor), the pooling operation works independently on every depth slice of the input, which means that only the width and the height of the tensor are reduced in size but not the depth. Most of the time, the pooling operation is performed on square grid patches of the image. In a 2D-CNN, each pixel within the image is represented by its $x$ and $y$ positions as well as the depth, representing image channels (red, green, and blue). The filter in this example is $2 \times 2$ pixels. It moves over the images both horizontally and vertically.

## 13.5 Convolutional Operation for 1D Tensor for Sequence Data

The convolutional operation described above is for 2D tensors but can also be applied to sequence data in 1D tensors, but instead of extracting 2D patches from the image tensor, now 1D patches (subsequences) are extracted from the sequences (Chollet and Allaire 2017), as can be observed in Fig. 13.11.

This type of 1D-convolutional operation works to capture temporal relationships; for this reason, they are good for identifying local patterns in a sequence. Since this operation is performed in every patch, a pattern learned at a certain position in a sequence can be better recognized in a different position. This 1D-convolutional operation is useful when you are interested in capturing interesting features from shorter (fixed length) segments of the data set and where the specific location of the feature within the sequence data is not significant. This 1D-convolutional operation is very practical for dealing with time series data or for analyzing signal data over a fixed-length period. Also, the pooling operation can be performed in a similar fashion as was illustrated in Fig. 13.11 for the convolutional operation in one-dimensional data. Also, 1D-CNNs are different from 2D-CNNs since 1D-CNNs allow using larger filter sizes. In a 1D-CNN, a filter of size 7 or 9 contains only 7 or 9 feature vectors, whereas in a 2D-CNN, a filter of size 7 will contain 49 feature vectors, making it a very broad selection.

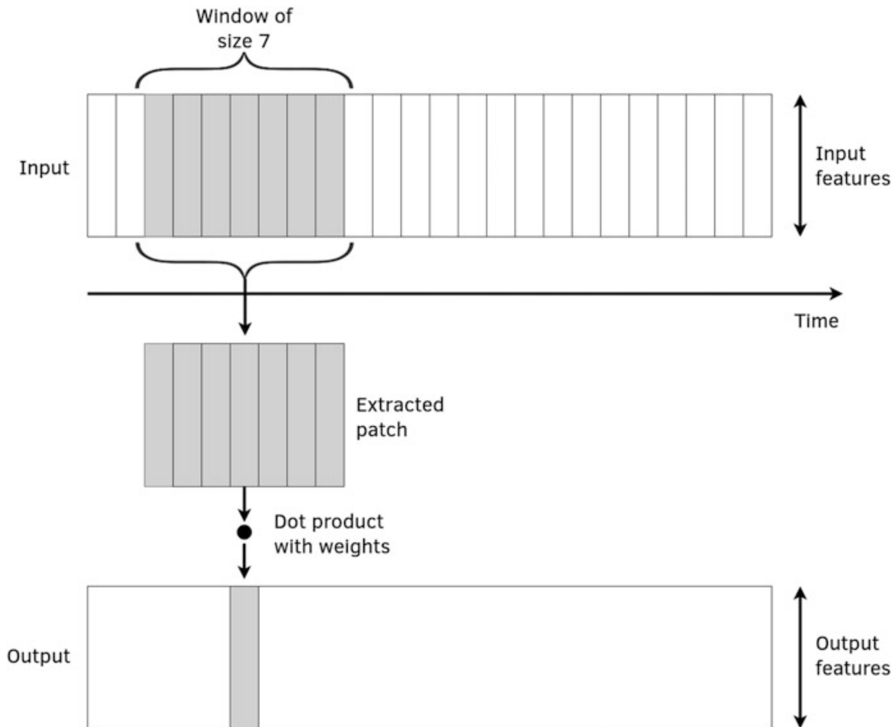



**Fig. 13.11** Illustration of how the 1D-convolutional operation works in a sequence of data

## 13.6 Motivation of CNN

Feedforward networks or fully connected networks studied in the two previous chapters use all the information of the image as input. This means that if we have images of 9500 × 8800 pixels, the dimension of the input should be 83,600,000 pixels since this topology uses as input the stacked information of the complete image, which is computationally expensive. However, by applying convolutions and pooling, CNNs reduce the dimension of the image considerably, without relevant loss of critical information, which is easier to process. CNNs are also preferred over feedforward networks with image input because they are not only good at learning features but are also scalable to massive data sets, since CNNs can extract high-level features such as edges from the input image without significant loss of information. The process of how the convolutional and the pooling operations are included in the training process of deep learning is illustrated in Fig. 13.12. Figure 13.12 shows that, most of the time, a convolutional layer is composed of three stages: first, the convolution operation is applied to the input, followed by a nonlinear transformation

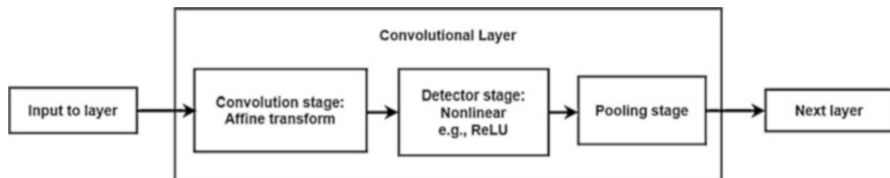



**Fig. 13.12** Illustration of the application of a convolutional layer that consists of the convolution operation followed by the ReLU nonlinear transformation and the pooling operation to reduce the input data

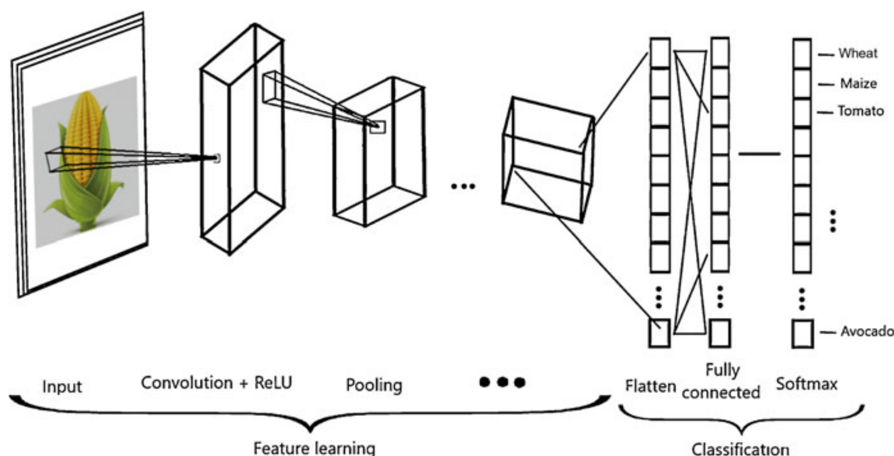



**Fig. 13.13** Convolutional neural network

(like ReLU, hyperbolic tangent, or another activation function); then the pooling operation is applied. With this convolutional layer, we significantly reduce the size of the input without relevant loss of information. The convolutional layer picks up different signals of the image by passing many filters over each image, which is key for reducing the size of the original image (input) without losing critical information, and in early convolutional layers we capture the edges in the image. CNN that applies convolutional layers are scalable and robust by learning different portions of a feature space. It is important to point out that the input to CNN is not restricted to tensors in two dimensions and can be of one, two, three, or more dimensions.

The output of the first convolutional layer can be followed by more convolutional layers, or its output can be stacked (flattened) directly to be used as input for a feedforward network, as can be observed in Fig. 13.13.

Compared to other machine learning algorithms, CNNs require less preprocessing since the convolutional layers (convolution + nonlinear transformation + pooling) make it possible to filter the noise and capture the spatial and

temporal dependencies of the input more efficiently. CNNs also provide more efficient fitting because they reduce the number of parameters that need to be estimated due to the reduction in the size of the input, and allow parameter sharing, and also due to the fact that the input is connected only to some neurons. Also, CNNs allow reusing weights in many circumstances, which facilitates the training process even with data sets that are not really large. Figure 13.13 indicates that depending on the complexity of the input (images), the number of convolutional layers can be more than one to be able to capture low-level details with more precision, but for sure the computational resources need to increase. Figure 13.13 also shows that after convolutional layers have been performed, the output of the last convolutional layer is flattened into a column vector (stacked as one long vector) and fed into a typical feedforward deep neural network for regression or classification purposes; backpropagation is applied to every iteration of training.

## 13.7 Why Are CNNs Preferred over Feedforward Deep Neural Networks for Processing Images?

To explain why the feedforward networks studied in the previous chapters (Chaps. 10, 11, and 12) are not the best option for processing images, we assume that we have as input images in the RGB format, that is, in a 3D tensor with $256 \times 256 \times 3$ pixels. This means that we have a matrix of dimension $256 \times 256$ for each of the three colors (Red, Green, and Blue). Stacking the 3D tensor on a 1D tensor implies that the input consists of 196,608 columns, which requires learning the same amount of weights (parameters) for each node in the first hidden layer of a feedforward network, also called fully connected deep neural network. Assuming that we used 300 neurons in the first hidden layer, this implies that we need to estimate 58,982,400.00 weights only for the first hidden layer, which is computationally very demanding. However, the greater the number of parameters that need to be estimated, the larger the training set that should be used to avoid overfitting, which of course increases the computational resources needed for the training process. On the other hand, there is evidence that CNNs require at most half of the parameters needed by a feedforward deep network and improve accuracy and reduce the training time substantially. There is also evidence that feedforward deep neural networks most of the time do not improve accuracy substantially after adding 5 hidden layers, while CNNs can continue improving prediction accuracy by adding more hidden layers; for this reason, there are applications of CNNs that use up to 150 hidden layers. Another disadvantage of feedforward deep neural networks is that they do not take advantage of structural information (correlation between pixels) since for processing these networks, the 3D tensor format of the images is transformed into a linear 1D tensor, which causes a loss of this structural information. Figure 13.14 shows how the information of an image in a 3D tensor is

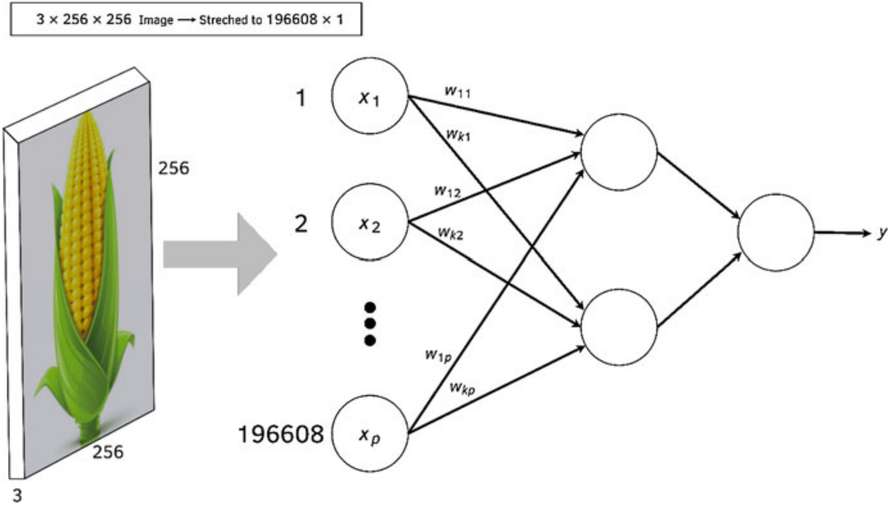



**Fig. 13.14** Image of a 3D tensor converted to a 1D tensor that is used as input for a fully feedforward deep neural network

converted to a linear 1D tensor as input for a feedforward deep neural network. Figure 13.14 also shows that the input image in a 3D tensor format with depth = 3 (three colors), width and size of 256 pixels produces a total input of size $256 \times 256 \times 3 = 196,608$ pixels.

Since CNNs can process the original 2D image at each depth, they are able to recognize shapes and capture the correlation between pixels using filter matching. Next, we want to develop a machine learning algorithm for the automatic classification of diseased and non-diseased wheat plants using images in a 3D format of dimension $256 \times 256 \times 3$ (Varma and Das 2018). Since we want to develop a classifier for binary outcomes under a linear model like logistic regression, the predictor (pre-activation) should be

$$z_i = \sum_{j=1}^{196,608} w_j x_{ij} + b$$

The weights $w_j$, $1 \leq j \leq 196{,}608$, can be interpreted as a filter for the category of interest (diseased plants), so that the classification operation can be interpreted as a filter matching the input image, as shown in Fig. 13.15.

This interpretation of linear filtering using filter matching gives evidence that it is possible to improve the system, but instead of using a filter that attempts to match the entire image, we can use smaller filters that try to match objects in local portions of the image. This has the following advantages: (a) smaller filters need a smaller filter size and, therefore, fewer parameters and (b) even if the object being detected moves around the image, the same filter can still be used, which leads to invariance translation (Varma and Das 2018). This is the main idea behind CNNs, as illustrated

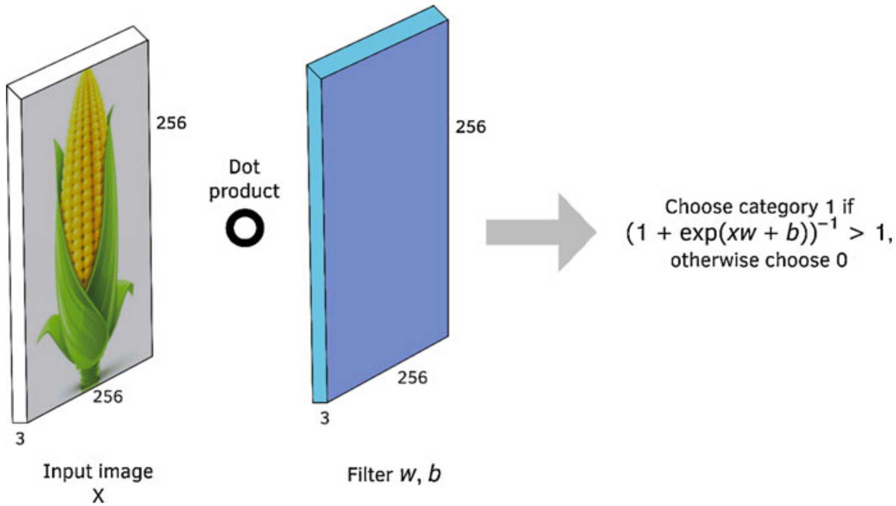



**Fig. 13.15** Classification using filter matching

in Fig. 13.16. The increasing number of filters in a particular layer increases the number of feature maps (i.e., depth) in the next layer, and the number of feature maps in the next layer depends on the number of filters we use for the convolutional operation in the previous layer. For this reason, it is possible that the number of feature maps is different in different hidden layers. It is noted that filters in early layers capture primitive shapes (vertical and horizontal edges, corners, points, etc.), while filters in the later layers can capture more complex compositions.

The important issues related to CNNs are explained in parts (a) to (d) of Fig. 13.16.

Part (a) of Fig. 13.16 tries to explain the key differences between filter matching in feedforward deep neural networks and CNNs. Feedforward deep neural networks use a larger filter than CNNs; CNN filters maintain the depth size but the height and width are smaller than the original height and width of the original image. This is illustrated in part (a) of Fig. 13.16, where a filter of size $7 \times 7 \times 3$ is used for an image of size $256 \times 256 \times 3$.

Part (b) of Fig. 13.16 exemplifies the filter matching operation using CNNs, and we can see clearly that the matching process is done locally, that is, in small patches or overlapping patches of an image. Filter matching is done by computing the pre-activation ($z_i$) and activation ($y_i$) values with the following equations. This is done at each position of the filter.

$$z_i = \sum_{j=1}^{147} w_j x_{ij} + b \tag{13.1}$$

$$y_i = g(z_i) \tag{13.2}$$

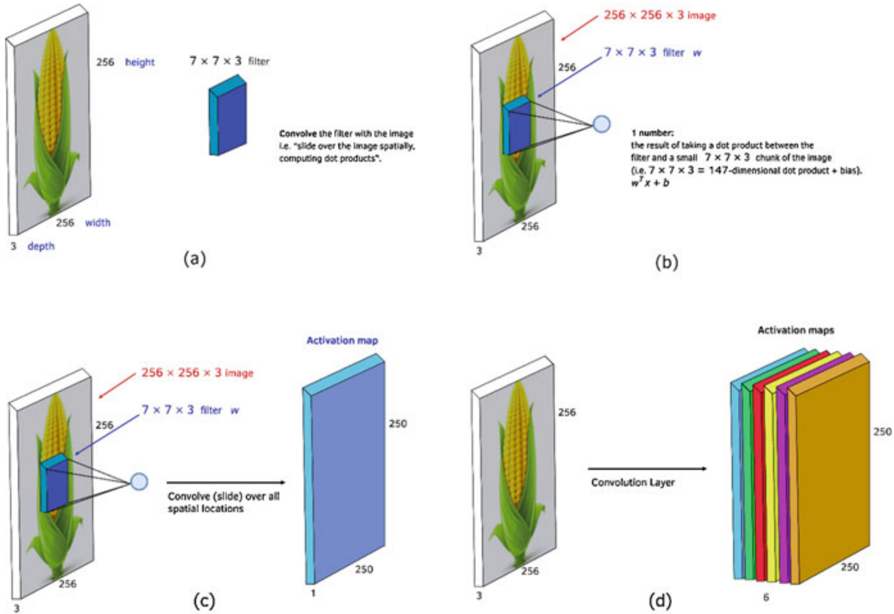



**Fig. 13.16** Illustrating local filters and feature maps in CNNs

The pixel values in Eqs. (13.1) and (13.2), $x_{ij}$, $i = 1, 2, \ldots, 147$, are called the local receptive field, which corresponds to the local patch of the image of size $7 \times 7 \times 3$ and changes as the filter slides vertically and horizontally across the whole image. The filter has only $7 \times 7 \times 3 + 1 = 148$ parameters, a lot less than $256 \times 256 \times 3 + 1 = 196,609$ parameters needed for the feedforward deep neural network given in Fig. 13.15. This reduction in the required number of parameters is due to the fact that under CNNs, parameters are shared. The local image patch used each time, as well as the filter, are 3D tensor structures, but the multiplication operation of Eq. (13.1) uses a stretched out vectorized version of the two tensors (Varma and Das 2018). The convolutional operation results in much sparser neural networks than the fully connected networks.

The filter is moved across the image, as can be seen in part (c) of Fig. 13.16, and a new value of $z_i$ is computed with Eq. (13.1) at each position, which produces an output matrix of size $250 \times 250$. This output matrix is called the feature map (or activation map). Convolution is the name given to this operation that computes the dot product at each position in which we slide the filters across the image. All the nodes in the feature map are tuned to detect the same feature in the input layer at different positions of the image since the same filters are used for all nodes in the feature map. This means that CNNs are able to detect objects regardless of their location in the image and, for this reason, CNNs possess the property of translational invariance (Varma and Das 2018).

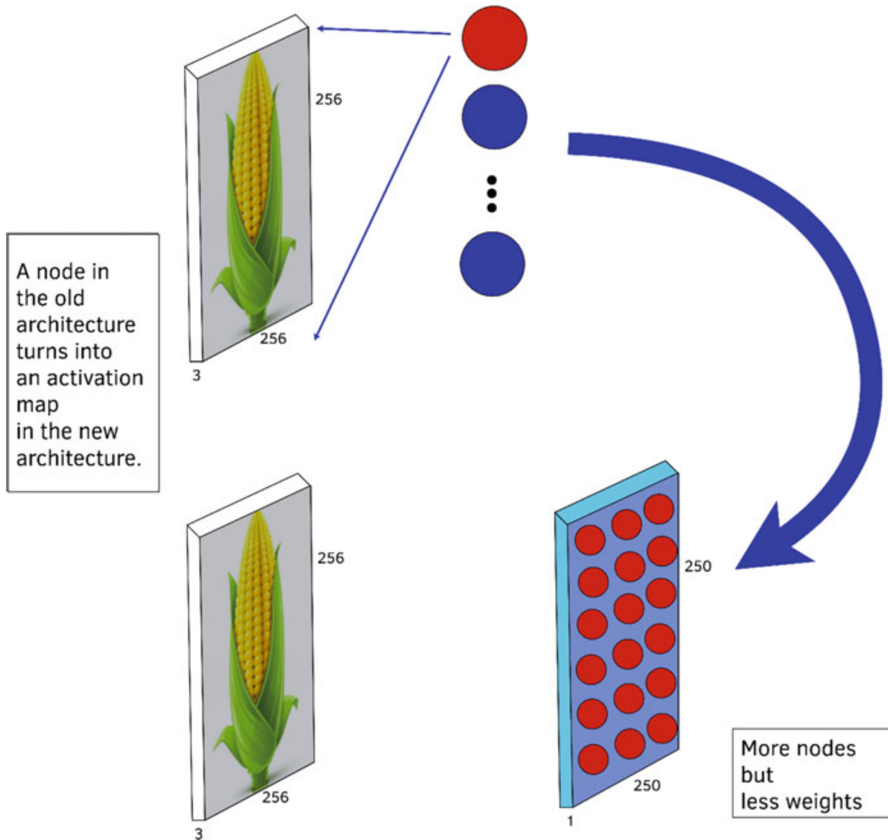



**Fig. 13.17** Relation between CNNs and deep feedforward networks

The examples given above illustrate how CNNs work using a single filter and are able to capture or detect a single pattern in the input image. For these reasons, we need to add many more filters to detect many patterns, each of which produces its own feature map, as can be seen in part (d) of Fig. 13.16. For example, only vertical edges can be detected with feature map 1, while only horizontal edges can be detected with feature map 2. This means that a hidden CNN consists of a stack of feature maps.

The entire input plane spans the filter in feedforward deep neural networks, looking for patterns that span the entire image. On the other hand, CNNs use smaller filters to detect smaller shapes built hierarchically into bigger shapes and objects to capture in more detail the information of the image. This is practical since real-world images are built with smaller patterns that rarely span the entire image. Thanks to the translational invariance property, any shape can be detected regardless of its location in the image plane.

Figure 13.17 also contrasts the feedforward deep neuronal network with CNNs. The top part of Fig. 13.17 shows the input image and the neurons in the first hidden

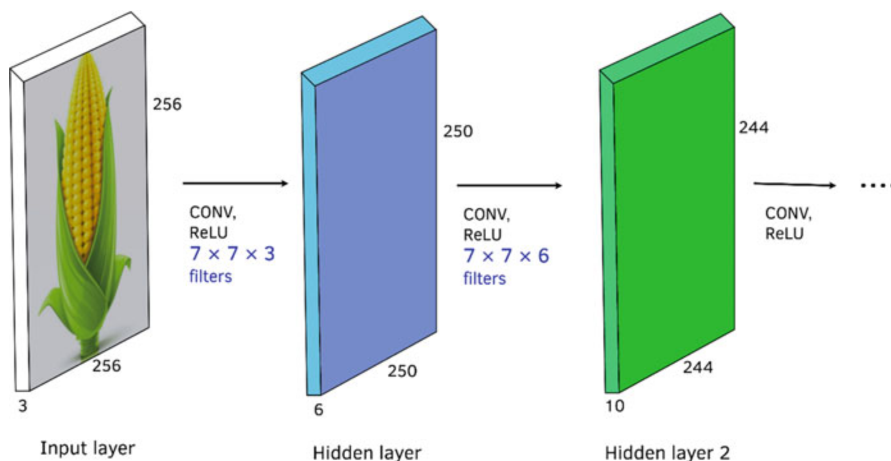



**Fig. 13.18** CNNs with multiple hidden layers

layer; when a particular pattern in the image is detected, a neuron is activated, but each neuron looks for a particular pattern. On the other hand, in CNNs, each neuron of the hidden layer is replaced by a feature map with multiple sub-neurons. A local filter is used to compute the activation value at each sub-neuron in an activation map which in different parts of the image looks for the same pattern. For this reason, in CNNs as many feature maps are needed as neurons in feedforward deep neural networks. Figure 13.17 also shows that the number of weights in the algorithm using CNNs is considerably reduced at the expense of the increase in the number of neurons. However, the increase in the number of neurons increases the computational resources required, but this is the price we need to pay for the reduction in the number of parameters that we need to estimate during the training process (Varma and Das 2018).

Figure 13.18 illustrates how to implement more than one hidden layer in a deep neural network using the convolutional operation. In this figure, we can see that it is almost the same as when a convolutional operation is performed between the input layer and the first hidden layer. Figure 13.18 also shows that the second convolutional hidden layer added contains ten feature maps, each generated with filters of size $7 \times 7 \times 10$, and the depth of this second hidden layer was inherited from the first hidden layer; for this reason, this is not a free parameter. In CNNs, the initial hidden layers are able to detect simple shapes, which allows the later layers to detect more complex shapes. But to be able to successfully implement a CNN, we need to tune: (a) the number of feature maps (number of filters) needed in each hidden layer, (b) the size of the filter, and (c) the stride size (Varma and Das 2018).

## 13.8 Illustrative Examples

**Example maize with CNNs** This data set contains 101 maize lines, each of which was studied in two environments (FLAT5I and BED2I) and genotyped with 101 markers. This first example will use CNNs in one dimension, which are useful for capturing spatial relationships between genomic markers and also in time series data.

The phenotypic information contains three columns: one for environments (Env), the other for genotypes (GID), and the last one for grain yield (Yield):

```
> head(Pheno)
      GID      Env    Yield
1 GID7459918 FLAT5I 5.481080
2 GID7461686 FLAT5I 6.395386
3 GID7462121 FLAT5I 6.633570
4 GID7462462 FLAT5I 5.358746
5 GID7624708 FLAT5I 6.281305
6 GID7624898 FLAT5I 6.746165
```

With command tail(), we can see the last six observations of a data set:

```
> tail(Pheno)
        GID      Env   Yield
197 GID8057500 BED2I 4.658421
198 GID8057905 BED2I 5.235285
199 GID8058141 BED2I 4.768317
200 GID8058432 BED2I 4.899983
201 GID8059034 BED2I 4.820649
202 GID8059268 BED2I 5.097920
```

Where we can see that there are 202 observations and 101 genotypes, and each of these genotypes was measured in two environments, as can be seen next.

```
> Env=unique(Pheno$Env)
> Env
[1] FLAT5I BED2I
Levels: FLAT5I BED2I
```

Then, with the following lines of code, the design matrices of environment and genotypes are created:

```
Z.E<-model.matrix(~0 + as.factor(Pheno$Env))
Z.G<-model.matrix(~0 + as.factor(Pheno$GID))
Z.G=Z.G%*%Markers_Final
```

Next, with the following lines of code, the training-testing sets are created using the BMTME package:

```
pheno <- data.frame(GID =Pheno[, 1], Env =Pheno[, 2],
          Response =Pheno[, 3])
CrossV <- CV.KFold(Pheno, DataSetID = 'GID', K = 5, set_seed = 123)
```

Then we select the response variable and create the input matrix of information:

```
y=(Pheno[, 3])
X=cbind(Z.E,Z.G)
```

The whole data set contains 202 observations and the input information contains 103 independent variables.

From this output and input information (y,X), we extracted a training sample of 161 observations for training (y_trn, X_trn), and the remaining observations were used for testing (y_tst, X_tst). Using this output and input, next we show the basic CNN in one dimension (1D-CNN), that consists of stacking layer_conv_1d() and layer_max_pooling_1d. The basic difference in the implementation of 1D-CNN is in the specification of the model, which is given next.

```
######Specification of the 1D-CNN model#############
 model_Sec<-keras_model_sequential()
 model_Sec %>%
  layer_conv_1d(filters=32, kernel_size=5, activation ="relu",
input_shape=c(NULL, dim(X_trn)[2],1)) %>%
  layer_max_pooling_1d(pool_size =2) %>%
  layer_conv_1d(filters=64, kernel_size=5, activation ="relu") %>%
  layer_max_pooling_1d(pool_size =4, strides=1) %>%
  layer_conv_1d(filters=64, kernel_size=5, activation ="relu") %>%
  layer_max_pooling_1d(pool_size =6,strides=2) %>%
  layer_flatten() %>%
  layer_dense(units =500 , activation ="relu") %>%
  layer_dropout(rate =0) %>%
  layer_dense(units =1)

 summary(model_Sec)
```

As mentioned above, the layer_conv_1d() is used to specify hidden layers for CNNs in one dimension that need an input of two dimensions, where the first dimension is due to the number of independent variables in the training set, while the second is the spatial dimension. Then inside this layer, the following needs to be specified: the number of filters, the activation function, the kernel size, and the stride which by default is 1. The number of filters and the kernel size should be specified by the user or tuned. Then after a layer_conv_1d(), a pooling layer is used with the function layer_max_pooling_1d(), that need as input the parameters pool size and stride which is by default equal to 2 for pooling layers. The pool size should be specified by the user, and in this case, values of 3, 5, or 7 are very common. This scheme is in agreement with the one provided in Fig. 13.12, which illustrates that a convolutional layer consists of a convolutional operation followed by ReLU nonlinear transformation and the pooling operation to reduce the input data.

However, the ReLU and pooling operations after the convolutional operation are not mandatory. Three convolutional layers are specified in this example, and the only difference between them is the number of filters, the type of activation function, the pool size, and the stride specified in each convolutional layer. After the third convolutional layer, a flatten layer is specified using the function layer_flatten() which reshapes the tensor to have the shape that is equal to the number of elements contained in the tensor, not including the batch dimension. After this layer, the layers are fully connected, as those studied in feedforward neural networks; the first contains 500 units and the output layer has only one layer since we want to predict a continuous outcome. Note that 1D-CNNs process input patches independently since they are not sensitive to the order of the timesteps.

Next is a summary of the 1D-CNN model.

Since the input has 103 time points and 32 filters were used in the first convolutional layer, the output for this layer is $(103 - 5 + 1) = 99$ time points and 32 filters, since by default, this convolutional operation has stride = 1. Therefore, for this layer, the required number of parameters is equal to $(32 + 1) \times 5 = 192$. Then for the first max pooling operation, the output contains $(99 - 2)/2 + 1 = 49$ time points and 32 is the depth; since this is not affected by the pooling operation, here and in all max pooling operations, there are no parameters to estimate (Fig. 13.19). For the second convolutional operation, the time points required are equal to $(49 - 5) + 1 = 45$ with a depth equal to 64, since this was the number of

```
Model: "sequential_8"
________________________________________________________________________________
Layer (type)                      Output Shape                  Param #
================================================================================
conv1d_17 (Conv1D)                (None, 99, 32)                192
________________________________________________________________________________
max_pooling1d_17 (MaxPooling1D)   (None, 49, 32)                0
________________________________________________________________________________
conv1d_18 (Conv1D)                (None, 45, 64)                10304
________________________________________________________________________________
max_pooling1d_18 (MaxPooling1D)   (None, 42, 64)                0
________________________________________________________________________________
conv1d_19 (Conv1D)                (None, 38, 64)                20544
________________________________________________________________________________
max_pooling1d_19 (MaxPooling1D)   (None, 17, 64)                0
________________________________________________________________________________
flatten_5 (Flatten)               (None, 1088)                  0
________________________________________________________________________________
dense_10 (Dense)                  (None, 500)                   544500
________________________________________________________________________________
dropout_5 (Dropout)               (None, 500)                   0
________________________________________________________________________________
dense_11 (Dense)                  (None, 1)                     501
================================================================================
Total params: 576,041
Trainable params: 576,041
Non-trainable params: 0
```

**Fig. 13.19** Summary of the 1D-CNN with three convolutional layers and two feedforward neural network layers

filters specified for this convolutional operation. For this second convolutional operation, $32 \times 5 \times 64 + 64 = 10{,}304$ parameters are required and need to be estimated. For the second pooling layer, the output contains the same 64 filters as the depth but the number of time points now is equal to $(45 - 4)/1 + 1 = 42$ since now the stride = 1 (Fig. 13.19). The third convolutional layer also has 64 filters since this was the value specified for this layer, but the number of time points of the output now is equal to $(42 - 5) + 1 = 38$, while for this operation, the number of parameters needed to be estimated is equal to $64 \times 5 \times 64 + 64 = 20{,}544$ (Fig. 13.19). Then the third pooling operation needs $(38 - 6)/2 + 1 = 17$ time points since the stride is equal to 2 and the pool size is equal to 6. Then the flattened layer stacks the final output of the third pooling layer; for this reason, it produces an output of $17 \times 64 = 1088$. Since in the first feedforward neural network, we specifically used 500 neurons, the number of parameters that need to be estimated is equal to $500 \times (1088 + 1) = 544{,}500$. Finally, in the output layer of the feedforward deep neural network, 501 parameters are required, since 500 weights +1 intercept need to be estimated (Fig. 13.19).

Next, we provide the flags needed under a 1D-CNN to implement a grid search to tune the required hyperparameters. The following code creates the flags; this code is called: Code_Tuning_With_Flags_CNN_1D_1HL_2CHL.R

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",32),
 flag_integer("Epoch1",500),
 flag_numeric("learning_rate", 0.001),
 flag_integer("filters",64),
 flag_numeric("KernelS", 3),
 flag_numeric("PoolS", 2),
 flag_numeric("val_split",0.2))

####b) Defining the DNN model
build_model<-function() {
model <- keras_model_sequential()
model %>%
 layer_conv_1d(filters=FLAGS$filters, kernel_size=FLAGS$KernelS,
activation =FLAGS$activation1,input_shape=c(dim(X_trII)[2],1)) %>%
 layer_max_pooling_1d(pool_size =FLAGS$PoolS) %>%
 layer_conv_1d(filters=FLAGS$filters, kernel_size=FLAGS$KernelS,
activation =FLAGS$activation1) %>%
 layer_max_pooling_1d(pool_size =FLAGS$PoolS) %>%
 layer_flatten() %>%
 layer_dense(units=FLAGS$units , activation =FLAGS$activation1) %>%
 layer_dropout(rate=FLAGS$dropout1) %>%
 layer_dense(units =1)
```

```
#####c) Compiling the DNN model
model %>% compile(
 loss = "mse",
 optimizer =optimizer_adam(lr=FLAGS$learning_rate),
 metrics = c("mse"))
model}

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")})

early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =50)

###########d) Fitting the DNN model################
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
 X_trII, y_trII,
 epochs =FLAGS$Epoch1, batch_size =FLAGS$batchsize1,
 shuffled=F,
 validation_split =FLAGS$val_split,
 verbose=0,callbacks = list(early_stop,print_dot_callback))
```

In a) are given the default flag values for the number of filters, kernel size, pool size, etc. In b) the DNN model is defined and the flag parameters are incorporated within our DNN model. But instead of specifying only dense layers (layer_dense) like in feedforward networks, first we specified the convolutional layers followed by pooling layers, and the specified dense layers are at the end. But before specifying dense layers, a flatten layer is specified that transforms the input from a tensor of 2D or more dimensions to a vector (tensor of one dimension). The specification of the dense layers is exactly the same as in all the feedforward networks studied before; the new part is the specification of the convolutional layers. In this code, only convolutional layers are only specified in 1D; for this reason, the keras function used to fit this DNN model is layer_conv_1d(), while for the pooling process, the layer_max_pooling_1d() function is used. The convolutional layers in 1D take as input 2D tensors with shape (samples, time features) and also return a shaped 2D tensor.

The first axis is a 1D window (patch) in the temporal axis, while the second is the input tensor (Chollet and Allaire 2017). CNNs in 1D can recognize temporal patterns in a sequence and since the same input transformation is performed at every subsequence (window or patch), a pattern learned in a specific position of the sequence can later be recognized at a difference position, making CNNs in 1D translation invariant (for temporal translations). CNNs in 1D are insensitive to the order of the timesteps (beyond the size of the convolutional windows) since they

process input patches (subsequences) independently; for this reason, they should be a good choice when global ordering of the sequence is not fundamentally meaningful in the processing of temporal patterns. In this example, two convolutional layers were specified; for this reason, in the above code (part b) appears in the specification of the deep learning model two times layer_conv_1d() and layer_max_pooling_1d(), before the flatten layer, which is specified as layer_flatten(). However, the number of the convolutional layers to be specified is problem-dependent, and more than one can be used. Also, the number of dense layers after the flatten layer can be more than one. Finally, it is important to mention that in the output layer, we only specified one neuron without specifying the activation function since we want to predict a continuous outcome, and by default, the linear activation function will be used. For all activation functions, except the output layer, we specified the relu activation function to capture no linear patterns. The input_shape for CNN in one dimension needs an input of two dimensions; for this reason, we specified the first dimension as the number of columns in the matrix design (X) and 1, since we only have data in one dimension. In CNNs in 1D, it is practical to use kernel sizes of 7 or 9.

In part c) of the code, the model is compiled with a loss function and as metric the mean squared error (MSE). Since the response we want to predict is continuous, here we also specified the optimizer_adam() as optimizer. In part d) the model is fitted using the number of epochs, batch size, and validation split as specified in the flags (part a). The fitting process in this case was done using the early stopping method.

The code given above called Code_Tuning_With_Flags_CNN_1D_1HL_2CHL.R is in the code given in Appendix 1. The code given in Appendix 1 executes the grid search using the library tfruns (Allaire 2018) and the tuning_run() function. The grid search implemented is shown below.

```
 runs.sp<-tuning_run("Code_Tuning_With_Flags_00_CNN_1D_3HL_3CHL.
R",runs_dir= '_tuningE1',flags=list(dropout1= c(0,0.05),
           units = seq(dim(X)[2]/2,2*dim(X)[2],40),
           activation1=("relu"),
           batchsize1=c(32),
           Epoch1=c(500),
           learning_rate=c(0.001),
           filters=c(32,64),
           KernelS=c(3),
           PoolS=c(1),
           val_split=c(0.2)),sample=1,confirm=FALSE,echo =F)
```

The grid search is composed of 16 combinations of hyperparameters: two values of dropout, four units and two values of filters, and one value for the remaining hyperparameters. The code given in Appendix 1 was run for five-fold, and each time four were used for training and the remaining for testing. The prediction performance is reported in terms of MSE and mean arctangent absolute percentage error (MAAPE) due to the fact that the outcome we want to predict is continuous. Table 13.3 indicates a similar prediction performance in both environments.

**Table 13.3** Prediction performance for different values for hidden layers (HL) in the feedforward deep neuronal network part and different hidden convolutional layers (HCL) with five outer fold cross-validation

| HL | HCL | Environment | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|---|
| 1 | 1 | FLAT5I | 1.0864 | 0.1819 | 0.1319 | 0.0111 |
| 1 | 1 | BED2I | 1.0869 | 0.2357 | 0.1904 | 0.0213 |
| **1** | **2** | **FLAT5I** | **0.3906** | **0.0375** | **0.0737** | **0.0036** |
| 1 | 2 | BED2I | 0.2979 | 0.0578 | 0.0922 | 0.0092 |
| 1 | 3 | FLAT5I | 0.4612 | 0.0563 | 0.0814 | 0.0049 |
| 1 | 3 | BED2I | 0.2719 | 0.0639 | 0.0845 | 0.0091 |
| 2 | 1 | FLAT5I | 0.4316 | 0.0148 | 0.0797 | 0.0016 |
| 2 | 1 | BED2I | 0.2145 | 0.0331 | 0.079 | 0.0047 |
| 2 | 2 | FLAT5I | 0.4604 | 0.0387 | 0.0814 | 0.0061 |
| 2 | 2 | BED2I | 0.2167 | 0.0435 | 0.0767 | 0.0073 |
| 2 | 3 | FLAT5I | 15.1851 | 14.8425 | 0.3065 | 0.2428 |
| 2 | 3 | BED2I | 8.2881 | 8.0526 | 0.321 | 0.2357 |
| 3 | 1 | FLAT5I | 0.4575 | 0.0495 | 0.0862 | 0.0064 |
| **3** | **1** | **BED2I** | **0.169** | **0.0273** | **0.0697** | **0.0075** |
| 3 | 2 | FLAT5I | 0.68 | 0.0839 | 0.1007 | 0.0081 |
| 3 | 2 | BED2I | 0.2333 | 0.0282 | 0.0819 | 0.0052 |
| 3 | 3 | FLAT5I | 0.5161 | 0.0375 | 0.0838 | 0.0053 |
| 3 | 3 | BED2I | 0.2663 | 0.0572 | 0.0878 | 0.0095 |

## 13.9 2D Convolution Example

The implementation of 2D-CNNs requires input for each observation in at least a 2D format. For this reason, these CNNs are useful for capturing spatial relationships in the input data; for this reason, they are most popular when using image information as input. In the context of genomic selection, the input data (SNPs) are not in this 2D format; for this reason, to use 2D-CNNs, a set of encoding methods was developed to overcome these constraints. We used one-hot encoding, which is simply recoding the three SNP genotypes as three 0/1 dummy variables. Using this approach, nonlinear relationships can be modeled using nonlinear activation functions in the first layer. Under one-hot coding, each marker is represented by a three-dimensional vector with 1 at the index for one genotype; the rest of them are set at 0. To illustrate this, we assume that the genotypes [AA, Aa, aa] are represented as [1, 0, 0], [0, 1, 0], and [0, 0, 1], respectively (Fig. 13.20). Liu et al. (2019), Bellot et al. (2018), and Pérez-Enciso and Zingaretti (2019) used this approach, but Liu et al. (2019) also used one-hot coding taking into account the missing values; the input vector is four-dimensional because the additional dimension was required to accommodate the missing values.

Before we explain the basic issues to implement a 2D-CNN, we note that we will use the same data sets as were used for illustrating the 1D-CNN, which contains 202 observations; the input information contains 101 independent variables, but with

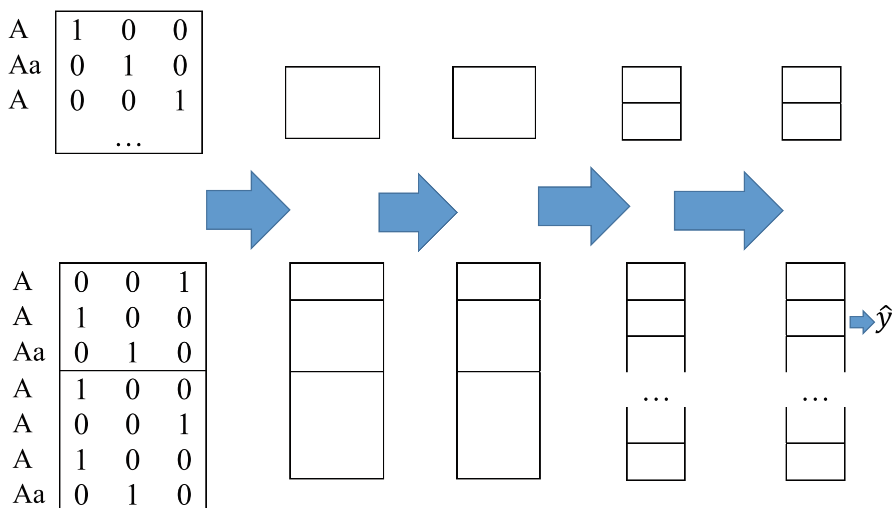



**Fig. 13.20** CNN with genotypes that are one-hot encoded. A denotes the homozygous, a is the reference homozygous, and Aa is the heterozygous. Processed features with two CNN layers are then passed to the output processing block, which contains a flatten layer and a fully connected dense layer

the difference that now a two-dimensional matrix was created for each observation. This means that each observation input has a height of 101 and a width equal to 3, which is equivalent to an image of 101 $\times$ 3.

Again from output and input information (y,X), we extracted a training set with 161 observations (y_trn, X_trn) and a testing set (y_tst, X_tst) with 41 observations. Next we show how 2D-CNNs are implemented by stacking layer_conv_2d() and layer_max_pooling_2d layers.

```
 model_Sec<-keras_model_sequential()
 model_Sec %>%
 layer_conv_2d(filters=32, kernel_size=c(3,3), activation ="relu",
input_shape=c(dim(X_trn)[2], dim(X_trn)[3], dim(X_trn)[4])) %>%
 layer_max_pooling_2d(pool_size =c(5,1), strides =2) %>%
 layer_conv_2d(filters=64,kernel_size=c(3,1), activation ="relu") %>%
 layer_max_pooling_2d(pool_size =c(3,1), strides=3) %>%
 layer_conv_2d(filters=128, kernel_size=c(3,1), activation ="relu") %
>%
 layer_max_pooling_2d(pool_size =c(2,1)) %>%
  layer_flatten() %>%
  layer_dense(units =384, activation ="relu",
kernel_regularizer=regularizer_l2(0.001)) %>%
  layer_dropout(rate =0) %>%
  layer_dense(units =1)

 summary(model_Sec)
```

As mentioned above, a 2D-CNN consists of stacking layer_conv_2d() and layer_max_pooling_2d(). Layer_conv_2d() requires at least three arguments which are the number of filters, kernel size, activation function and input shape, at least for the first convolutional layer. All three parameters need to be specified by the user; with regard to the kernel size, the user needs to specify the height and width of the kernel, which can be different. The activation function can be any of the ones existing in deep learning, but most of the time, the ReLU activation function is used in CNNs. With regard to the input_shape for 2D-CNNs, it needs to be three-dimensional: the first should be the height (101 in this example), the second the width (3 in this example), and the last one the depth (1 in this example) of the image or input.

For the layer_max_pooling_2d(), at least two inputs are required: one is the height and width of the patch that will be pooled and the second is the stride size. In this example, a layer_conv_2d() is followed by a layer_max_pooling_2d(), which form a convolutional layer; for this reason, in this example, three convolutional layers were implemented since the layer_conv_2d() and the layer_max_pooling_2d () appear three times. After the last convolutional layer, layer_flatten() was used that flattened the output of the last layer_max_pooling_2d(). Then a feedforward layer was used for which the user provides the number of neurons. Finally, the output layer that belongs to the feedforward layers is specified. In Fig. 13.21 is given the summary of the output shape of each convolutional and feedforward layer produced for the keras code given above for this 2D-CNN example.

Since the 2D input has order $101 \times 3$, and in the first convolutional layer 32 filters were used, the output of this layer has a height equal to $(101 - 3 + 1) = 99$, width equal to $3 - 3 + 1 = 1$ since the stride $= 1$, and 32 filters (specified by the user). The

```
Model: "sequential_6"
________________________________________________________________________________
Layer (type)                        Output Shape                    Param #
================================================================================
conv2d_18 (Conv2D)                  (None, 99, 1, 32)               320
________________________________________________________________________________
max_pooling2d_18 (MaxPooling2D)     (None, 48, 1, 32)               0
________________________________________________________________________________
conv2d_19 (Conv2D)                  (None, 46, 1, 64)               6208
________________________________________________________________________________
max_pooling2d_19 (MaxPooling2D)     (None, 15, 1, 64)               0
________________________________________________________________________________
conv2d_20 (Conv2D)                  (None, 13, 1, 128)              24704
________________________________________________________________________________
max_pooling2d_20 (MaxPooling2D)     (None, 6, 1, 128)               0
________________________________________________________________________________
flatten_6 (Flatten)                 (None, 768)                     0
________________________________________________________________________________
dense_12 (Dense)                    (None, 384)                     295296
________________________________________________________________________________
dropout_6 (Dropout)                 (None, 384)                     0
________________________________________________________________________________
dense_13 (Dense)                    (None, 1)                       385
================================================================================
Total params: 326,913
Trainable params: 326,913
Non-trainable params: 0
________________________________________________________________________________
```

**Fig. 13.21** Summary of the 2D-CNN with three convolutional layers and two feedforward neural network layers

required number of parameters for this layer is equal to (32 × 9 + 32) = 320. Then for the first max pooling operation, the output has a width of 1, a height equal to (99 − 5)/2 + 1 = 48, and a depth equal to 32. Since this is not affected by the pooling operation, here and in all max pooling operations there are no parameters to estimate (Fig. 13.21). For the second convolutional operation, the width is equal to 1 and the height is equal to (48 − 3) + 1 = 46 with a depth equal to 64, since this was the number of filters specified for this convolutional operation. For the second convolutional operation, 32 × 3 × 64 + 64 = 6208 parameters are required and need to be estimated. For the second pooling layer, the output contains the same 64 filters as the depth, plus a width of 1; the height is equal to (46 − 3)/3 + 1 = 15 since now the stride = 3 (Fig. 13.21). The third convolutional layer has 128 filters since this was the value specified for this layer, but the output now has a width equal to 1 and a height equal to (15 − 3) + 1 = 13, while the number of parameters that need to be estimated for this operation is equal to 64 × 3 × 128 + 128 = 24,704 (Fig. 13.21). The third pooling operation also has a width of 1 and a height of (13 − 2)/2 + 1 = 6 since the stride and the pooling size were equal to 2. The flattened layer stacked the final output of the third pooling layer and produced an output of 6 × 128 = 768. Since in the first feedforward neural network, we specified using 384 neurons, the number of parameters that need to be estimated is equal to 384 × (768 + 1) = 295,296. Finally, in the output layer of the feedforward deep neural network, 385 parameters are required, since 384 weights +1 intercept need to be estimated (Fig. 13.21).

Again, we will create flags for the tuning process. The following code creates the flags under 2D-CNNs and is called Code_Tuning_With_Flags_CNN_2D_2HL_1CHL.R. The details of this code are given next:

```
####a) Declaring the flags for hyperparameters
FLAGS = flags(
 flag_numeric("dropout1", 0.05),
 flag_integer("units",33),
 flag_string("activation1", "relu"),
 flag_integer("batchsize1",56),
 flag_integer("Epoch1",1000),
 flag_numeric("learning_rate", 0.001),
 flag_integer("filters",5),
 flag_integer("KernelS",3),
 flag_integer("PoolS",1),
 flag_numeric("val_split",0.2))

####b) Defining the DNN model
build_model<-function() {
model<-keras_model_sequential()
model %>%
 layer_conv_2d(filters=FLAGS$filters, kernel_size=c(FLAGS$KernelS,
FLAGS$KernelS), activation =FLAGS$activation1,input_shape=c(dim
(X_trII)[2],dim(X_trII)[3],dim(X_trII)[4])) %>%
 layer_flatten() %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
```

```
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =FLAGS$units, activation =FLAGS$activation1) %>%
 layer_dropout(rate =FLAGS$dropout1) %>%
 layer_dense(units =1)

#####c) Compiling the DNN model
model %>% compile(
 loss = "mse",
 optimizer =optimizer_adam(lr=FLAGS$learning_rate),
 metrics = c("mse"))
model}

model<-build_model()
model %>% summary()

print_dot_callback <- callback_lambda(
 on_epoch_end = function(epoch, logs) {
  if (epoch %% 20 == 0) cat("\n")
  cat(".")})

early_stop <- callback_early_stopping(monitor = "val_loss",
mode='min',patience =50)

###########d) Fitting the DNN model################
model_Final<-build_model()
model_fit_Final<-model_Final %>% fit(
 X_trII, y_trII,
 epochs =FLAGS$Epoch1, batch_size =FLAGS$batchsize1,
 shuffled=F,
 validation_split =FLAGS$val_split,
 verbose=0,callbacks = list(early_stop,print_dot_callback))
```

In part a) of the above code are given the default flag values for the number of filters, kernel size, pool size, etc. In part b) the DNN model with one convolutional layer is defined, and at the end, two dense layers are specified. Before specifying the dense layers, a flatten layer is specified to transform the input from a tensor of 2D or more dimensions to a vector (tensor of one dimension). The specification of the dense layer is exactly the same as all the feedforward networks and 1D-CNNs studied previously; the new part is the specification of the convolutional layers in 2D tensors. Now we are specifying 2D convolutions with the function layer_conv_2d() and max pooling with the function layer_max_pooling_2d(). The convolutional layers need input information in three dimensions (height, width, depth) not including the batch dimension, and return a 3D tensor as output (Chollet and Allaire 2017). CNNs in 2D also have the translation invariance property, since they can recognize local patterns in images, and a pattern learned on a specific position of the 2D tensor can later be recognized anywhere, due to the fact that the same transformation is performed in all patches extracted from the original image. As mentioned in this example, one convolutional layer was specified (see part b of the code) using the layer_conv_2d() and layer_max_pooling_2d() before the flatten

layer and the layer_flatten(); however, the user can implement more than one convolutional layer and dense layers depending on the problem at hand. In this example, only one neuron is specified in the output layer because we want to predict a continuous outcome. Except in the output layer, we used the ReLU activation function to capture nonlinear patterns. We specified a 3D tensor in the input_shape() function with the first argument representing the height, the second the width, and last one the depth. In CNNs in 2D, it is very popular to use kernel sizes of $3 \times 3$ or $5 \times 5$.

In part c) the model is compiled using the MSE as the loss function and metric since the outcome of interest is continuous, and optimizer_adam() was specified as the optimizer. In part d) the model is fitted using the number of epochs, batch size, and validation split as specified in the flags (part a). The fitting process in this case was done using the early stopping method.

The code given above was put in a file called Code_Tuning_With_Flags_CNN_2D_2HL_2CHL.R, which is used in Appendix 2 to implement a 2D tensor deep neural network with convolutional layers for predicting a continuous response variable. The code given in Appendix 2 executes a grid search using the library tfruns (Allaire 2018) and the tuning_run() function; the grid search that was implemented is shown below.

```
runs.sp<-tuning_run("Code_Tuning_With_Flags_00_CNN_2D_2HL_1CHL.
R",runs_dir = '_tuningE1',flags=list(dropout1= c(0,0.05),
                                units = seq(dim(X)[2]/2,2*dim(X)[2],40),
                                   activation1=("relu"),
                                   batchsize1=c(32),
                                   Epoch1=c(500),
                                   learning_rate=c(0.001),
                                   filters=c(32,64),
                                   KernelS=c(3),
                          PoolS=c(1),                          val_split=c
(0.2)),sample=1,confirm=FALSE,echo =F)
```

The grid search is composed of 16 combinations of hyperparameters: two values of dropout, four units and two values of filters, and one value for the remaining hyperparameters. The code given in Appendix 2 was run for five-fold and each time four folds were used for training and the remaining for testing. The prediction performance is reported in terms of MSE and MAAPE due to the fact that the outcome we want to predict is continuous. Table 13.4 shows a similar prediction performance in both environments.

Table 13.4 indicates that the best predictions are observed with only one hidden layer since with more hidden layers the results are slightly worse. To implement the model with one and three hidden layers, we modified the number of hidden layers in Code_Tuning_With_Flags_00_CNN_2D_2HL_2CHL.R and in the code given in Appendix 2.

Finally, as mentioned earlier, CNNs can capture the spatial structure between genomic markers, and for this reason, these deep neural network methods have been

**Table 13.4** Prediction performance under a 2D-CNN, with one hidden convolutional layer (HCL) and 1, 2, and 3 hidden layers (HL) in the fully connected layers. Five outer fold cross-validation was implemented

| Environment | HL | MSE | SE_MSE | MAAPE | SE_MAAPE |
|---|---|---|---|---|---|
| FLAT5I | 1 | 1.4724 | 0.2758 | 0.1616 | 0.0163 |
| BED2I | 1 | 1.2572 | 0.2889 | 0.2095 | 0.0287 |
| FLAT5I | 2 | 1.5223 | 0.4893 | 0.1577 | 0.0276 |
| BED2I | 2 | 1.3068 | 0.3928 | 0.207 | 0.0428 |
| FLAT5I | 3 | 1.7384 | 0.4147 | 0.1718 | 0.0239 |
| BED2I | 3 | 1.5394 | 0.4565 | 0.2271 | 0.0396 |

implemented in genomic selection by some authors like Liu et al. (2019), Bellot et al. (2018), Pérez-Enciso and Zingaretti (2019), Ma et al. (2018), Zou et al. (2019), and Waldmann et al. (2020). But the outperformance of CNNs with regard to statistical genomic selection models is modest, and a lot of time is needed to tune these CNNs. In genomics, most applications concern functional genomics, with examples that include predicting the sequence specificity of DNA- and RNA-binding proteins and of enhancer and cis-regulatory regions, methylation status, gene expression, and control of splicing (Waldmann et al. 2020). Deep learning has been especially successful when applied to regulatory genomics, by using architectures directly adapted from modern computer vision and natural language-processing applications.

## 13.10 Critics of Deep Learning

To finish this chapter, we need to mention some of the problems of deep learning to have both faces of the same coin. This is important since in the media deep learning is sold as the panacea that will solve all association and prediction problems using data, and for this reason, it is not necessary to learn any other machine learning or statistical learning algorithm—which is totally wrong. Next, we will provide some arguments about why it is totally wrong to think that it is enough to learn only deep learning algorithms and no other statistical machine learning methods: (A) Most deep learning models are not interpretable and when inference (association) is the goal, simple statistical learning methods are always the best option; in general, deep learning-based solutions lack mathematical elegance and offer very little interpretability of the found solution or understanding of the underlying phenomena. (B) Many times simple statistical machine learning algorithms, such as multiple regression or logistic regression, work just fine for the required model accuracy. This is due in part to the "*No free lunch theorem*" that states that there is no one model that works for every problem since the assumptions of a great model for one problem might not hold for another problem, and also due to the size of the training set, since the lower the training set, the worse the performance of deep learning models. Also, since many times the data at hand have a clearly linear pattern and there is no reason

to solve this problem with a nonlinear model (e.g., multilayer perceptron), this means that different problems need a different best method. Also, when the input is not structured and of high quality, many times conventional statistical machine learning methods are enough to get prediction performance that is good enough. (C) Most conventional statistical machine learning algorithms do not require massive computations that need to be run on computer clusters or graphic processing units, nor a complex tuning process that needs wise optimization algorithms that require effective initializations and gradual stochastic gradient learning to perform reasonably well. (D) The many successful applications of deep learning are supported only by great empirical evidence with few theoretical understanding of the underlying paradigm. Moreover, the optimization employed in the learning process is highly non-convex and intractable from a theoretical viewpoint.

For the reasons mentioned above, Rami (2017) stated that deep learning is "alchemy," since many times it is not possible to successfully implement it due to the fact that the loss function is a non-convex function that only guarantees a local minimum and even when a lot of time is used for tuning, it is not possible or easy to arrive at a reasonable solution. This is true because tuning methods used to select hyperparameters require brute force and their implementation consists of a trial-and-error process that is mostly art and little science. Le Cun, one of the great promoters of deep learning, argues that deep learning is "alchemy" in the following way: "In the history of science and technology, the engineering artifacts have almost always preceded the theoretical understanding: the lens and the telescope preceded optic theory, the steam engine preceded thermodynamics, the computer preceded computer science. Why? Because theorists will spontaneously study 'simple' phenomena, and will not be enticed to study complex ones until there is a practical importance to it" (Elad 2017).

However, successful applications of deep learning can be seen in many areas and the power of this tool, even in the many problems mentioned above, will continue reshaping and influencing not only our everyday lives but also many other areas of science. For this reason, it is important to adopt the good things from this field in our own field to take advantage of this power tool and help provide the strong theoretical background that is needed for this field to continue growing, improving, and reshaping our everyday lives and our ways of doing science.

Also, DL methods have some really good advantages, since they can efficiently handle natural data in their raw form, which is not possible with most statistical machine learning models (Lecun et al. 2015). DL has also proven to provide models with higher accuracy that are efficient at discovering patterns in high-dimensional data, making them applicable in a variety of domains. They also make it possible to more efficiently incorporate all omics data (Metabolomics, Proteomics, and Transcriptomics) in the same model. However, it is difficult to implement DL models in genomic data sets, since they usually represent a very large number of variables and a small number of samples; they also offer a lot of opportunities to design specific topologies (deep neural networks) that can deal in a better way with the type of data present in genomic selection.

## Appendix 1

Code for implementing 1D-CNNs with three convolutional layers

```
#setwd("~/Osval/Osval/DL_Phil")
rm(list=ls())
library(BMTME)
library(tensorflow)
library(keras)
library(plyr) ####ply
library(tidyr)
library(dplyr)
library(tfruns)
library(devtools)
options(bitmapType='cairo')

#use_session_with_seed(45)
tensorflow::tf$random$set_seed(45)
#########Loading the data

load("Data_Corn_Toy.RData")
######Phenotypic information
Pheno<-Pheno_Corn_Toy
head(Pheno)
tail(Pheno)
Env=unique(Pheno$Env)
Env

Markers_Final=Markers_Corn_Toy
dim(Markers_Final)

#####Genomic relationship matrix
G=G_Corn_Toy

################Design matrices#################################
Z.E<-model.matrix(~0 + as.factor(Pheno$Env))
Z.G<-model.matrix(~0 + as.factor(Pheno$GID))
dim(Z.G)
Markers=Markers_Final
Z.G=Z.G%*%Markers
dim(Z.G)

############Selecting the response variable######################
Y<-as.matrix(phenoMaizeToy[, -c(1, 2)])
####Training testing sets using the BMTME package###############
pheno <- data.frame(GID =Pheno[, 1], Env =Pheno[, 2],
         Response =Pheno[, 3])

CrossV <- CV.KFold(Pheno, DataSetID = 'GID', K = 5, set_seed = 123)
```

```
################Function for averaging the predictions############
summary.BMTMECV <- function(results, information= 'compact', digits =
4, ...) {
 results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(MSE = mean((Predicted-Observed)^2),
      MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed)))) %>%
  select(Environment, Trait, Partition, MSE, MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Environment, Trait) %>%
  summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
     SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>
%
  select(Environment, Trait, MSE, SE_MSE, MAAPE, SE_MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
        complete = presum,
        extended = {
         finalSum$Partition <- 'All'
         presum$Partition <- as.character(presum$Partition)
         presum$SE_MSE <- NA
         presum$SE_MAAPE <- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

#######Final X and y for fitting the model###################
y=(Pheno[, 3])
X=cbind(Z.E,Z.G)
dim(X)

############Outer cross-validation#######################
digits=4
Names_Traits=colnames(Y)
results=data.frame()
t=1

for (o in 1:5){
# o=1
 tst_set=CrossV$CrossValidation_list[[o]]
 X_trn=X[-tst_set,]
 X_trn=array_reshape(X_trn,dim=c(dim(X_trn)[1],dim(X_trn)[2],1))
 X_tst=X[tst_set,]
 X_tst=array_reshape(X_tst,dim=c(dim(X_tst)[1],dim(X_tst)[2],1))
```

```
 y_trn=(y[-tst_set])
 y_tst=(y[tst_set])
 dim(X_trn)
 dim(X_tst)

 ################Inner cross-validation###########################
 X_trII=X_trn
 y_trII=y_trn
 #########Using the tuning_fun function for tuning
hyperparameters#####
 runs.sp<-tuning_run("Code_Tuning_With_Flags_00_CNN_1D_1HL_3CHL.
R",runs_dir = '_tuningE1',flags=list(dropout1= c(0,0.05),
                                  units = seq(dim(X)[2]/2,2*dim(X)[2],40),
                                     activation1=("relu"),
                                     batchsize1=c(32),
                                     Epoch1=c(500),
                                     learning_rate=c(0.001),
                                     filters=c(32,64),
                                     KernelS=c(3),
                                     PoolS=c(1),
                            val_split=c(0.2)),sample=1,confirm =FALSE,echo
=F)

 # Ordering in decreasing order the prediction performance in the grid
 #runs=runs.sp[order(runs.sp$metric_val_mean_squared_error,
decreasing = TRUE), ]
 runs=runs.sp[order(runs.sp$metric_val_mse, decreasing = TRUE), ]
 dim(runs)[1]
 ###########Selecting the best combination in terms of prediction
performance####
 pos_opt=dim(runs)[1]
 opt_runs=runs[pos_opt,]

 #####Selecting the best hyperparameters
 Drop_O=opt_runs$flag_dropout1
 Epoch_O=opt_runs$epochs_completed
 Units_O=opt_runs$flag_units
 activation_O=opt_runs$flag_activation1
 batchsize_O=opt_runs$flag_batchsize1
 lr_O=opt_runs$flag_learning_rate
 filters_O=opt_runs$flag_filters
 Kernel_O=opt_runs$flag_KernelS
 pool_Size_O=opt_runs$flag_PoolS

 print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
   if (epoch %% 20 == 0) cat("\n")
   cat(".")})

 ###########Refitting the model with the optimal values##############
 model_Sec<-keras_model_sequential()
 model_Sec %>%
```

```
  layer_conv_1d(filters=filters_O, kernel_size=Kernel_O, activation
=activation_O,input_shape=c(dim(X_trII)[2],1)) %>%
  layer_max_pooling_1d(pool_size =pool_Size_O) %>%
  layer_conv_1d(filters=filters_O, kernel_size=Kernel_O, activation
=activation_O) %>%
  layer_max_pooling_1d(pool_size =pool_Size_O) %>%
  layer_conv_1d(filters=filters_O, kernel_size=Kernel_O, activation
=activation_O) %>%
  layer_max_pooling_1d(pool_size =pool_Size_O) %>%
  layer_flatten() %>%
  layer_dense(units =Units_O , activation =activation_O) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =1)

 model_Sec %>% compile(
  loss = "mse",
  optimizer = optimizer_adam(lr=lr_O),
  metrics = c("mse"))

 ModelFited <-model_Sec %>% fit(
  X_trn, y_trn,
  epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
  verbose=0,callbacks=list(print_dot_callback))

 ####Prediction of testing set ##########################
 Yhat=model_Sec%>% predict(X_tst)
 y_p=Yhat
 y_p_tst=as.numeric(y_p)

 ###########Saving the predictions of each outer testing set##########
 results<-rbind(results, data.frame(Position=tst_set,
                  Environment=CrossV$Environments[tst_set],
                  Partition=o,
                  Units=Units_O,
                  Epochs=Epoch_O,
                  Observed=round(y[tst_set], digits), #$response,
digits),
                  Predicted=round(y_p_tst, digits),
                  Trait=Names_Traits[t]))
 cat("CV=",o,"\n")
}

results
#############Average of prediction performance##################
Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
Pred_Summary
write.csv(Pred_Summary,file="Corn_Toy_DL_CNN_1D_1HL_3CHL_Env+G.
csv")
```

## Appendix 2

Code for implementing 2D-CNNs with one convolutional hidden layer

```
rm(list=ls())
library(BMTME)
library(tensorflow)
library(keras)
library(plyr) ####ply
library(tidyr)
library(dplyr)
library(tfruns)
library(devtools)
options(bitmapType='cairo')

#use_session_with_seed(45)
tensorflow::tf$random$set_seed(0)

###Loading the data
load("Data_Corn_Toy.RData")

######Phenotypic information
Pheno<-Pheno_Corn_Toy
Env=unique(Pheno$Env)
Env

Markers_Final=Markers_Corn_Toy
dim(Markers_Final)
nclas=length(unique(as.factor(Markers_Final)))
nclas
unique(as.factor(Markers_Final))
head(Markers_Final[,1:4])
nmark=ncol(Markers_Final)
nrows=nrow(Markers_Final)
Mat_Mark=matrix(0, nrow=nrow(Markers_Final),ncol=3*ncol
(Markers_Final))
for (i in 1:nrows){
#i=1
Mat_i=to_categorical(Markers_Final[i,],nclas)
Mat_i2=t(Mat_i)
Mat_Mark[i,]=array_reshape(Mat_i2,dim=c(1,3*nmark))
}
dim(Mat_Mark)
head(Mat_Mark[,1:10])

#####Genomic relationship matrix
G=G_Corn_Toy

################Design matrices################################
Z.E<-model.matrix(~0 + as.factor(Pheno$Env))
Z.G<-model.matrix(~0 + as.factor(Pheno$GID))
```

```
dim(Z.G)
Markers=Mat_Mark
Z.G=Z.G%*%Markers
dim(Z.G)

###########Selecting the response variable######################
Y<-as.matrix(phenoMaizeToy[, -c(1, 2)])
####Training testing sets using the BMTME package###############
pheno <- data.frame(GID =Pheno[, 1], Env =Pheno[, 2],
          Response =Pheno[, 3])

CrossV <- CV.KFold(Pheno, DataSetID = 'GID', K = 5, set_seed = 123)

################Function for averaging the predictions############
summary.BMTMECV <- function(results, information= 'compact', digits =
4, ...) {
 results %>%
  group_by(Environment, Trait, Partition) %>%
  summarise(MSE = mean((Predicted-Observed)^2),
      MAAPE = mean(atan(abs(Observed-Predicted)/abs(Observed)))) %>%
  select(Environment, Trait, Partition, MSE, MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> presum

 presum %>% group_by(Environment, Trait) %>%
  summarise(SE_MAAPE = sd(MAAPE, na.rm = T)/sqrt(n()), MAAPE = mean
(MAAPE, na.rm = T),
     SE_MSE = sd(MSE, na.rm = T)/sqrt(n()), MSE = mean(MSE, na.rm = T)) %>
%
  select(Environment, Trait, MSE, SE_MSE, MAAPE, SE_MAAPE) %>%
  mutate_if(is.numeric, funs(round(., digits))) %>%
  as.data.frame() -> finalSum

 out <- switch(information,
        compact = finalSum,
        complete = presum,
        extended = {
         finalSum$Partition <- 'All'
         presum$Partition <- as.character(presum$Partition)
         presum$SE_MSE <- NA
         presum$SE_MAAPE <- NA
         rbind(presum, finalSum)
        }
 )
 return(out)
}

#######Final X and y for fitting the model###################
y=(Pheno[, 3])
#Z.G=Z.G[1:5,1:21]
#Z.G
#nmark=7
```

```
Z.GMar=array_reshape(Z.G,dim=c(nrow(Z.G),nmark,3,1),order="F")
Z.GMar
dim(Z.GMar)
X=Z.GMar
dim(X)

############Outer cross-validation#######################
digits=4
Names_Traits=colnames(Y)
results=data.frame()
t=1
#
for (o in 1:5){
 tst_set=CrossV$CrossValidation_list[[o]]
 X_trn=X[-tst_set, , ,]
 X_trn=array_reshape(X_trn,dim=c(dim(X_trn)[1],nmark,3,1))
 X_tst=X[tst_set, , ,]
 X_tst=array_reshape(X_tst,dim=c(dim(X_tst)[1],nmark,3, 1))
 y_trn=(y[-tst_set])
 y_tst=(y[tst_set])
 dim(X_trn)
 dim(X_tst)

 ################Inner cross-validation###########################
 X_trII=X_trn
 y_trII=y_trn
 #######Using the tuning_fun function for tuning hyperparameters#####
 runs.sp<-tuning_run("Code_Tuning_With_Flags_00_CNN_2D_2HL_1CHL.
R",runs_dir = '_tuningE1',flags=list(dropout1= c(0,0.05),
                              units = seq(20,100,25),
                              activation1=("relu"),
                              batchsize1=c(16),
                              Epoch1=c(500),
                              learning_rate=c(0.01),
                              filters=c(28,56),
                              KernelS=c(3),
                              reg_l2=c(0.001,0.01),
                              PoolS=c(1),
                      val_split=c(0.5)),sample=1,confirm =FALSE,echo
=F)

 # Ordering in decreasing order the prediction performance in the grid
 #runs=runs.sp[order(runs.sp$metric_val_mean_squared_error,
decreasing = TRUE), ]
 runs=runs.sp[order(runs.sp$metric_val_mse, decreasing = TRUE), ]
 dim(runs)[1]
 ###########Selecting the best combination in terms of prediction
performance####
 pos_opt=dim(runs)[1]
 opt_runs=runs[pos_opt,]

 #####Selecting the best hyperparameters
 Drop_O=opt_runs$flag_dropout1
```

```
 Epoch_O=opt_runs$epochs_completed
 Units_O=opt_runs$flag_units
 activation_O=opt_runs$flag_activation1
 batchsize_O=opt_runs$flag_batchsize1
 lr_O=opt_runs$flag_learning_rate
 filters_O=opt_runs$flag_filters
 Kernel_O=opt_runs$flag_KernelS
 pool_Size_O=opt_runs$flag_PoolS
 L2_O=opt_runs$flag_reg_l2

 print_dot_callback <- callback_lambda(
  on_epoch_end = function(epoch, logs) {
   if (epoch %% 20 == 0) cat("\n")
   cat(".")})

 ###########Refitting the model with the optimal values##############
 model_Sec<-keras_model_sequential()
 model_Sec %>%
  layer_conv_2d(filters=filters_O, kernel_size=c(Kernel_O,Kernel_O),
activation =activation_O,kernel_regularizer=regularizer_l2(L2_O),
input_shape=c(dim(X_trII)[2],dim(X_trII)[3],dim(X_trII)[4])) %>%
  layer_max_pooling_2d(pool_size =c(1,1)) %>%
  layer_flatten() %>%
  layer_dense(units =Units_O, activation =activation_O,
kernel_regularizer=regularizer_l2(L2_O)) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =Units_O, activation =activation_O,
kernel_regularizer=regularizer_l2(L2_O)) %>%
  layer_dropout(rate =Drop_O) %>%
  layer_dense(units =1)

 model_Sec %>% compile(
  loss = "mae",
  optimizer = optimizer_adam(lr=lr_O),
  metrics = c("mae"))

 ModelFited <-model_Sec %>% fit(
  X_trn, y_trn,
  epochs=Epoch_O, batch_size =batchsize_O,
#####validation_split=0.2,early_stop,
  verbose=0,callbacks=list(print_dot_callback))

 ####Prediction of testing set ##########################
 Yhat=model_Sec%>% predict(X_tst)
 y_p=Yhat
 y_p_tst=as.numeric(y_p)

 ###########Saving the predictions of each outer testing set##########
 results<-rbind(results, data.frame(Position=tst_set,
                  Environment=CrossV$Environments[tst_set],
                  Partition=o,
                  Units=Units_O,
```

```
                    Epochs=Epoch_O,
                    Observed=round(y[tst_set], digits), #$response,
digits),
                    Predicted=round(y_p_tst, digits),
                    Trait=Names_Traits[t]))
 cat("CV=",o,"\n")
}

results

#############Average of prediction performance####################
Pred_Summary=summary.BMTMECV(results=results, information =
'compact', digits = 4)
Pred_Summary
write.csv(Pred_Summary,file="Corn_Toy_DL_CNN_2D_2HL_1CHL.csv")
```

## References


Allaire JJ (2018) Tfruns: training run tools for 'tensorflow'. https://CRAN.R-project.org/package=tfruns

Bellot P, De Los Campos G, Pérez-Enciso M (2018) Can deep learning improve genomic prediction of complex human traits? Genetics 210:809–819

Berzal F (2018) Redes neuronales and deep learning. Editor Fernando Berzal

Chollet F, Allaire JJ (2017) Deep learning with R. Manning Publications, Manning Early Access Program (MEA), 1st edn

Dobrescu A, Valerio Giuffrida M, Tsaftaris SA (2017) Leveraging multiple datasets for deep leaf counting. In: Proceedings of the IEEE International Conference on Computer Vision, pp 2072–2079

Elad M (2017) Deep, Deep Trouble. Deep learning's impact on image processing, mathematics and humanity. https://sinews.siam.org/Details-Page/deep-deep-trouble-4

Ghosal S, Blystone D, Singh AK et al (2018) An explainable deep machine vision framework for plant stress phenotyping. Proc Natl Acad Sci U S A 115(18):4613

Giuffrida MV, Doerner P, Tsaftaris SA (2018) Pheno-deep counter: a unified and versatile deep learning architecture for leaf counting. Plant J 96(4):880–890

Goodfellow I, Bengio Y, Courville A (2016) Deep learning. MAMIT Press, Cambridge

Hasan MM, Chopin JP, Laga H et al (2018) Detection and analysis of wheat spikes using convolutional neural networks. Plant Methods 14(1):100

Hubel D, Wiesel T (1959) Receptive fields of single neurons in the cat's striate cortex. J Physiol 124(3):574–591

Lecun Y, Bengio Y, Hinton G (2015) Deep learning. Nature 521(7553):436–444. https://doi.org/10.1038/nature14539

Liu Y, Wang D, He F, Wang J, Joshi T, Xu D (2019) Phenotype prediction and genome-wide association study using deep convolutional neural network of soybean. Front Genet 10:1091. https://doi.org/10.3389/fgene.2019.01091

Ma W, Qiu Z, Song J, Li J, Cheng Q, Zhai J et al (2018) A deep convolutional neural network approach for predicting phenotypes from genotypes. Planta 248:1307–1318. https://doi.org/10.1007/s00425-018-2976-9

Patterson J, Gibson A (2017) Deep learning: a practitioner's approach. O'Reilly Media

Pérez-Enciso M, Zingaretti LM (2019) A guide on deep learning for complex trait genomic prediction. Genes 10:553

Pound MP, Atkinson JA, Townsend AJ et al (2017) Deep machine learning provides state-of-the-art performance in image-based plant phenotyping. Gigascience 6(10):1–10
Rami (2017) Ali Rahimi's talk at NIPS(NIPS 2017 Test-of-time award presentation). https://www.youtube.com/watch?v=x7psGHgatGM
Uzal LC, Grinblat GL, Namías R et al (2018) Seed-per-pod estimation for plant breeding using deep learning. Comput Electron Agric 150:196–204
Varma S, Das S (2018) Deep learning. https://srdas.github.io/DLBook/index.html
Waldmann P, Pfeiffer C, Mészáros G (2020) Sparse convolutional neural networks for genome-wide prediction. Front Genet 11:25. https://doi.org/10.3389/fgene.2020.00025
Wang X, Xuan H, Evers B, Shrestha S, Pless R, Poland J (2019) High-throughput phenotyping with deep learning gives insight into the genetic architecture of flowering time in wheat. GigaScience 8:1–11
Zou J, Huss M, Abid A, Mohammadi P, Torkamani A, Telenti A (2019) A primer on deep learning in genomics. Nat Genet 51:12–18. https://doi.org/10.1038/s41588-018-0295-5

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 14
# Functional Regression

## 14.1 Principles of Functional Linear Regression Analyses

The general functional linear regression model with scalar response ($Y$) and one functional covariate ($x(\cdot)$) is defined by

$$Y = \mu + \int_0^T x(t)\beta(t)dt + \epsilon, \tag{14.1}$$

where $x(t)$ and $\beta(t)$ are a centered functional covariate and the functional coefficient regression, respectively, and $\epsilon$ is a random error often assumed to have a normal distribution with mean 0 and variance $\sigma^2$. Functional regression replaces the linear predictor of a linear regression model by integrating the product of a coefficient function $\beta(t)$ and centered covariate $x(t)$, which corresponds to a continuous non-delaying process.

Determining the infinite-dimensional beta coefficients $\beta(t)$ from a finite number of observations of the model (1) is a very difficult task. Indeed, it is almost always possible to find a function $\beta(t)$ satisfying the model with an error equal to 0, and there is an infinite number of these functions that give the same predictions (Ramsay et al. 2009). There are several procedures to solve this problem (Cardot and Sarda 2006); one of them is based on basis expansion (Fourier, B-splines, etc.) and will be adopted and described here.

A basis expansion solution is obtained by first representing the beta coefficient function $\beta(t)$ as

$$\beta(t) = \sum_{l=1}^{L_1} \beta_l \phi_l(t), \tag{14.2}$$

where $\phi_l(\cdot)$, $l = 1, \ldots, L_1$, is a collection of functions corresponding to the first $L_1$ elements of basis for a function space and $\beta_l$ are constants that depend on the

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_14

function to be represented (Ramsay et al. 2009). Then, by assuming this form for $\beta(t)$, model (14.1) can be expressed as

$$Y = \mu + \sum_{l=1}^{L_1} \beta_l \int_0^T x(t)\phi_l(t)dt + \epsilon = \mu + \boldsymbol{x}^{\mathrm{T}}\boldsymbol{\beta}_0 + \epsilon$$
$$= \boldsymbol{x}^{*\mathrm{T}}\boldsymbol{\beta} + \epsilon, \tag{14.3}$$

where $\boldsymbol{x}^* = [1, \boldsymbol{x}^{\mathrm{T}}]^{\mathrm{T}}$, $\boldsymbol{x} = [x_1, \ldots, x_{L_1}]^{\mathrm{T}}$, $x_l = \int_0^T x(t)\phi_l(t)dt$, $l = 1, \ldots, L_1$. So, if $y_i$, $i = 1, \ldots, n$, are independent observations of model (14.1), corresponding to covariate functions $x_i(\cdot)$, $i = 1, \ldots, n$, a basis expansion solution for the beta coefficient function is obtained by estimating the parameters involved in model (14.3), and then substituting $\widehat{\boldsymbol{\beta}} = \left[\mu, \widehat{\beta}_1, \ldots, \widehat{\beta}_{L_1}\right]^{\mathrm{T}}$ in (14.2) to obtain a basis-based estimation of $\beta(t)$:

$$\widehat{\beta}(t) = \sum_{l=1}^{L_1} \widehat{\beta}_l \phi_l(t).$$

When smoothing in the function coefficient is desired, one way to take more control of this is by using a roughness penalty, which combined with a high-dimensional basis could reduce the possibility of not considering some important features or taking into account some extraneous features (Ramsay et al. 2009). However, sometimes we can obtain good results without recurring to this, if the number of basis functions is smaller than the number of individuals in the sample (Ramsay et al. 2009).

Assuming that random errors are independently and identically distributed as a normal random variable with mean 0 and variance $\sigma^2$, $\epsilon_1, \ldots, \epsilon_n \sim iid\ N(0, \sigma^2)$, then $Y_i = \mu + \sum_{l=1}^{L_1} x_{il}\beta_l + \epsilon_i \sim N\left(\mu + \sum_{l=1}^{L_1} x_{il}\beta_l, \sigma^2\right)$, $x_{il} = \int_0^T x_i(t)\phi_l(t)dt$, $i = 1, \ldots, n$, $l = 1, \ldots, L_1$. So, the maximum likelihood estimation of parameters $\boldsymbol{\beta}$ and $\sigma^2$ is given by

$$\widehat{\boldsymbol{\beta}} = \left(\boldsymbol{X}^{*\mathrm{T}}\boldsymbol{X}^*\right)^{-1}\boldsymbol{X}^{*\mathrm{T}}\boldsymbol{y} \tag{14.4}$$

$$\widehat{\sigma}^2 = \frac{1}{n}\left(\boldsymbol{y} - \boldsymbol{X}^*\widehat{\boldsymbol{\beta}}\right)^{\mathrm{T}}\left(\boldsymbol{y} - \boldsymbol{X}^*\widehat{\boldsymbol{\beta}}\right), \tag{14.5}$$

where $\boldsymbol{X}^* = [\mathbf{1}_n\ \ \boldsymbol{X}]$, $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]^{\mathrm{T}}$ is assumed to be of full column rank ($n > L_1$), $\mathbf{1}_n$ is a vector of dimension $n \times 1$ with all its entries equal to 1 and $\boldsymbol{x}_i = [x_{i1}, \ldots, x_{iL_1}]^{\mathrm{T}}, i = 1, \ldots, n$.

However, in practice, the functional covariate is often unknown and not continuously observed. Usually, it is only measured in a finite number of points $t_1 < t_2 < \ldots < t_m$ in time or another domain. So, to complete the solution described before, the usual approach is also to assume that the covariate function can be represented as a linear combination of a set of basis functions ($\psi_l(\cdot)$, $l = 1, \ldots, L_2$)

$$x_i(t) = \sum_{o=1}^{L_2} c_{io}\psi_o(t), \tag{14.6}$$

where $c_{io}$, $o = 1, \ldots, L_2$, are constants to be determined for each observation, $i = 1, \ldots, n$. Usually, this is determined by least squares, in which case, by assuming that all curves were observed at the same time points, this can be computed as

$$\widehat{\boldsymbol{c}}_i = \left[\widehat{c}_{i1}, \ldots, \widehat{c}_{iL_2}\right]^{\mathrm{T}} = \left(\boldsymbol{\Psi}^{\mathrm{T}}\boldsymbol{\Psi}\right)^{-1}\boldsymbol{\Psi}^{\mathrm{T}}\boldsymbol{x}_i(\boldsymbol{t}), \tag{14.7}$$

where $\boldsymbol{\Psi}$ is a matrix of dimension $m \times L_2$ given by

$$\boldsymbol{\Psi} = \begin{bmatrix} \psi_1(t_1) & \cdots & \psi_{L_2}(t_1) \\ \psi_1(t_2) & \ddots & \psi_{L_2}(t_2) \\ \vdots & \vdots & \vdots \\ \psi_1(t_m) & \cdots & \psi_{L_2}(t_m) \end{bmatrix} \tag{14.8}$$

and $\boldsymbol{x}_i(\boldsymbol{t}) = [x_i(t_1), \ldots, x_i(t_m)]^{\mathrm{T}}$ is the vector with the actual values where the covariate curve of individual $i$ was observed. With this, the elements of $\boldsymbol{x}_i = \left[x_{i1}, \ldots, x_{iL_1}\right]^{\mathrm{T}}$, $x_{il} = \int_0^T x_i(t)\phi_l(t)dt$, can be re-expressed as $x_{il} = \int_0^T x_i(t)\phi_l(t)dt = \sum_{o=1}^{L_2}\widehat{c}_{io}\int_0^T \psi_o(t)\phi_l(t)dt = \boldsymbol{x}_l^{**\mathrm{T}}\widehat{\boldsymbol{c}}_i$ , with $\boldsymbol{x}_l^{**} = \left[x_{l1}^*, \ldots, x_{lL_2}^*\right]^{\mathrm{T}}$ and $x_{lo}^* = \int_0^T \phi_l(t)\psi_o(t)dt$, $o = 1, \ldots, L_2$ and $l = 1, \ldots, L_1$. From here, $\boldsymbol{x}_i$ can be expressed as

$$\boldsymbol{x}_i = \begin{bmatrix} \boldsymbol{x}_1^{**\mathrm{T}}\widehat{\boldsymbol{c}}_i \\ \vdots \\ \boldsymbol{x}_{L_1}^{**\mathrm{T}}\widehat{\boldsymbol{c}}_i \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_1^{**\mathrm{T}} \\ \vdots \\ \boldsymbol{x}_{L_1}^{**\mathrm{T}} \end{bmatrix}\widehat{\boldsymbol{c}}_i = \begin{bmatrix} \int_0^T \phi_1(t)\psi_1(t)dt & \cdots & \int_0^T \phi_1(t)\psi_{L_2}(t)dt \\ \vdots & \ddots & \vdots \\ \int_0^T \phi_{L_1}(t)\psi_m(t)dt & \cdots & \int_0^T \phi_{L_1}(t)\psi_{L_2}(t)dt \end{bmatrix}\widehat{\boldsymbol{c}}_i = \boldsymbol{Q}\widehat{\boldsymbol{c}}_i,$$

where

$$\boldsymbol{Q} = \begin{bmatrix} \int_0^T \phi_1(t)\psi_1(t)dt & \cdots & \int_0^T \phi_1(t)\psi_{L_2}(t)dt \\ \vdots & \ddots & \vdots \\ \int_0^T \phi_{L_1}(t)\psi_m(t)dt & \cdots & \int_0^T \phi_{L_1}(t)\psi_{L_2}(t)dt \end{bmatrix}.$$

Now, matrix $\boldsymbol{X}^*$ can be computed as

$$\boldsymbol{X}^* = [\,\mathbf{1}_n \quad \boldsymbol{X}\,], \tag{14.9}$$

where

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^{\mathrm{T}} \\ \vdots \\ \boldsymbol{x}_n^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} \widehat{\boldsymbol{c}}_1^{\mathrm{T}}\boldsymbol{Q}^{\mathrm{T}} \\ \vdots \\ \widehat{\boldsymbol{c}}_n^{\mathrm{T}}\boldsymbol{Q}^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} \widehat{\boldsymbol{c}}_1^{\mathrm{T}} \\ \vdots \\ \widehat{\boldsymbol{c}}_n^{\mathrm{T}} \end{bmatrix}\boldsymbol{Q}^{\mathrm{T}} = \begin{bmatrix} \boldsymbol{x}_1(\boldsymbol{t})^{\mathrm{T}}\boldsymbol{\Psi}\left(\boldsymbol{\Psi}^{\mathrm{T}}\boldsymbol{\Psi}\right)^{-1} \\ \vdots \\ \boldsymbol{x}_n(\boldsymbol{t})^{\mathrm{T}}\boldsymbol{\Psi}\left(\boldsymbol{\Psi}^{\mathrm{T}}\boldsymbol{\Psi}\right)^{-1} \end{bmatrix}\boldsymbol{Q}^{\mathrm{T}}$$
$$= \boldsymbol{X}^{**}\boldsymbol{\Psi}\left(\boldsymbol{\Psi}^{\mathrm{T}}\boldsymbol{\Psi}\right)^{-1}\boldsymbol{Q}^{\mathrm{T}}$$

with $\boldsymbol{X}^{**} = [\boldsymbol{x}_1(\boldsymbol{t}) \quad \cdots \quad \boldsymbol{x}_n(\boldsymbol{t})]^{\mathrm{T}}$. Finally, the complete practical solution of the parameter estimates is obtained with (14.4) and (14.5) but replacing $\boldsymbol{X}^{*}$ as computed in (14.9).

There are several proposals to choose the "optimal" number of bases ($L_1$) to represent the $\beta(\cdot)$ function coefficient. One way is by means of the Bayesian information criterion (Górecki et al. 2018), which is defined as follows:

$$\mathrm{BIC} = -2\ell\left(\widehat{\boldsymbol{\beta}}, \widehat{\sigma}^2; \boldsymbol{y}\right) + (L_1 + 1)\log(n),$$

where $\ell\left(\widehat{\boldsymbol{\beta}}, \widehat{\sigma}^2; \boldsymbol{y}\right)$ is the log-likelihood evaluated in the maximum likelihood estimation of parameters $\boldsymbol{\beta}$ and $\sigma^2$. This is a compromise between the fit of the model (first term) and its complexity (second term, the number of parameters in the model). In general, the model with the lowest BIC is preferred. In particular, with this criterion, the "optimal" number of basis functions corresponds to the lowest BIC.

When smoothing is required in the curve to be estimated, one way to control it is through the introduction of a penalty term, as will be described later. However, sometimes good results can be obtained without the need for this, as long as the number of basis functions relative to the amount of data is kept small (Ramsay et al. 2009).

To choose the "optimal" number of basis functions ($L_2$) to represent the functional covariate data, we can also use the BIC criteria. To do this, consider that each curve is observed with error under the following model:

$$x_i(t_j) = \sum\nolimits_{o=1}^{L_2} c_{io}\psi_o(t_j) + \epsilon_{ij},$$

where for each $i = 1, \ldots, n$, $\epsilon_{ij}$, $j = 1, \ldots, m$, are independent random variables with distribution $N\left(0, \sigma_x^2\right)$. Then the likelihood of the parameters to be estimated ($\boldsymbol{c}_i = (c_{i1}, \ldots, c_{iL_2})^{\mathrm{T}}$ and $\sigma_x^2$) is given by

$$L\left(\boldsymbol{c}_i,\sigma_x^2;\boldsymbol{x}_i(\boldsymbol{t})\right)=\prod_{j=1}^{m} f_{nx_i\left(t_j\right)}\left(x_i\left(t_j\right)\right)$$
$$=\left(\frac{1}{2\pi\sigma_x^2}\right)^{m/2}\exp\left[-\frac{1}{2\sigma_x^2}\sum_{j=1}^{m}\left(x_i\left(t_j\right)-\sum_{o=1}^{L_2}c_{io}\psi_o\left(t_j\right)\right)^2\right].$$

From this, the maximum likelihood of the parameters of $\boldsymbol{c}_i$ and $\sigma_x^2$ are $\widehat{\boldsymbol{c}}_i = \left[\widehat{c}_{i1},\ldots,\widehat{c}_{iL_2}\right]^{\mathrm{T}} = \left(\boldsymbol{\Psi}^{\mathrm{T}}\boldsymbol{\Psi}\right)^{-1}\boldsymbol{\Psi}^{\mathrm{T}}\boldsymbol{x}_i(\boldsymbol{t})$ and $\widehat{\boldsymbol{\sigma}}_x^{\mathbf{2}} = \frac{1}{m}\sum_{j=1}^{m}\left(x_i\left(t_j\right)-\sum_{o=1}^{L_2}\widehat{c}_{io}\psi_o\left(t_j\right)\right)^2$, respectively. So, before fitting the regression model, the "optimal" value of $L_2$ that will represent each curve in the sample can be chosen with the smallest value of BIC in the corresponding model:

$$\mathrm{BIC} = -2\log\left[L\left(\widehat{\boldsymbol{c}}_i,\widehat{\boldsymbol{\sigma}}_x^{\mathbf{2}};\boldsymbol{x}_i(\boldsymbol{t})\right)\right]+(L_2+1)\log(m).$$

A global value of $L_2$ can be adopted as suggested by Górecki et al. (2018), and the mode of the "optimal" values obtained across all the represented curves. Note that the maximum likelihood estimate of $\boldsymbol{c}_i$ is the same as the least square estimate mentioned above.

Another alternative to the BIC approach is to choose the "optimal" number of basis functions by estimating the predictive ability obtained by using different values of $L_2$ and selecting the value with the best predictive performance (Ruppert et al. 2003). One way to do this is by using the leave-one-out cross-validation (LOOCV) with mean squared error of prediction as the criterion to measure the predictive performance:

$$\mathrm{CV}_1(L_2)=\sum\nolimits_{j=1}^{m}\left(x\left(t_j\right)-\widehat{x}_{-j}\left(t_j\right)\right)^2,$$

where $\widehat{x}_{-j}\left(t_j\right)$ is the predicted value of point $j$, $x(t_j)$, obtained by doing the representation of the function with $L_2$ bases but without this point, that is,

$$\widehat{x}_{-j}\left(t_j\right)=\sum\nolimits_{o=1}^{L_2}\widehat{c}_{-j,o}\psi_o\left(t_j\right),$$

where $\widehat{\boldsymbol{c}}_{-j} = \left[\widehat{c}_1^{-j},\ldots,\widehat{c}_{L_2}^{-j}\right]^{\mathrm{T}} = \left(\boldsymbol{\Psi}_{-j}^{\mathrm{T}}\boldsymbol{\Psi}_{-j}\right)^{-1}\boldsymbol{\Psi}_{-j}^{\mathrm{T}}\boldsymbol{x}_{-j}$ and $\boldsymbol{\Psi}_{-j}$ is a matrix of dimension $(m-1)\times L_2$, like the matrix design basis defined in (14.8) over $L_2$ basis functions, but without row $j$, and $\boldsymbol{x}_{-j}$ is the same as the vector that contains the observed values of the latent function, $\boldsymbol{x}(\boldsymbol{t})$, but removing the value of its position $j$. For a specific basis, the optimal number of basis is the one with the lowest value of $\mathrm{CV}_1(L_2)$.

## 14.2 Basis Functions

A “base” is a set of basis functions ($\phi_l$, $l = 1, 2, 3, \ldots$) such that “any” function ($x(t)$) can be approximated as well as required, by means of a linear combination of $L_2$ of these functions:

$$x(t) = \sum\nolimits_{l=1}^{L_2} c_l \phi_l(t),$$

where $c_l$ are values that will determine the function.

In general, to represent data in functions by means of basis functions, you need to

(a) Choose suitable basis functions (polynomial basis, Fourier basis, B-spline basis, etc.).
(b) Determine the number of basis functions to consider ($L_2$).
(c) Estimate the coefficients $c_l$, $l = 1, \ldots, L_2$.

The degree of smoothness of function $x(t)$ depends on the value of $L_2$ that is chosen (a small value of $L_2$ causes more smoothing of the curves) and the optimum value for $L_2$ selected using the Bayesian information criterion (BIC) (Górecki et al. 2018) or with cross-validation, as described before.

### 14.2.1 *Fourier Basis*

The Fourier basis is often used for periodic or near-periodic data and is often useful for expanding functions with weak local characteristics and with an approximately constant curvature. It is not appropriate for data with discontinuities in the function or in low order derivatives (Ramsay et al. 2009).

The Fourier basis is created by the following functions:

$$\phi_1(t) = \frac{1}{\sqrt{P}}, \phi_2(t) = \frac{1}{\sqrt{\frac{P}{2}}}\sin(\omega t), \phi_3 = \frac{1}{\sqrt{\frac{P}{2}}}\sin(\omega t), \phi_4(t) = \frac{1}{\sqrt{\frac{P}{2}}}\cos(2\omega t),$$

$$\phi_5 = \frac{1}{\sqrt{\frac{P}{2}}}\cos(2\omega t), \phi_6(t) = \frac{1}{\sqrt{\frac{P}{2}}}\cos(3\omega t), \quad \phi_7 = \frac{1}{\sqrt{P/2}}\cos(3\omega t), \ldots,$$

where $\omega$ is related to period $P$ by $\omega = 2\pi/P$, and in practical applications, this is often taken as the range of $t$ values where the data are observed (Ramsay et al. 2009).

The graph on interval (0,8) of the first five of these functions (0, 8) with period 4 is given in Fig. 14.1. The vertical dotted lines are the end of the subinterval that corresponds to the period of each function. This figure can be reproduced by the following R code:

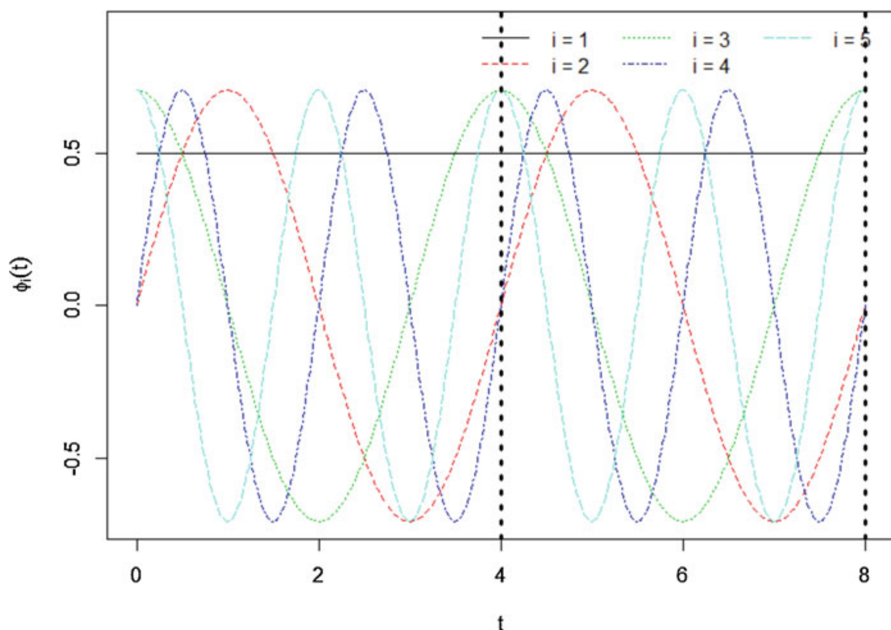



**Fig. 14.1** Graph of the first five elements of the Fourier basis with period 4 on interval (0,8)

```
library(fda)
BF = create.fourier.basis(rangeval=c(0,8),nbasis=5,period=4)
plot(BF,xlab='t',ylab=expression(phi[i](t)),ylim=c(-.7,.9), lty=1:
5,lwd=1)
abline(v=seq(4,8,4),lty=3,lwd=3)
legend('topright',paste('i = ',1:5,sep=''),lty=1:5,col=1:5,
bty='n',lwd=1,ncol=3)
```

### *14.2.2 B-Spline Basis*

A B-spline basis is typically used for nonperiodic data where the underlying function is locally smooth. The coefficients of a B-spline basis can be calculated quickly; they form a very flexible system because very good approximations of functions can be obtained even with a relatively small number of basis functions.

A B-spline is a type of spline that is a piecewise-polynomial continuous function and has a specific number of continuous derivatives on an interval. Specifically, a $q + 1$-order spline with interior knots $T_j, j = 1, \ldots, K$ (usually placed to take into account the data change points) on the observation interval $[0, T] = (0, T_1] \cup (T_1, T_2] \cup \ldots \cup (T_K, T]$ is a continuous function $s_q$ such that in each subinterval $(T_{j-1}, T_j]$ there is a polynomial of degree $q$+1 that has continuous derivatives of order $q-1$ in

each knot, that is, the $d$th derivate of $s_d$, $s_q^{(d)}\left(T_j\right)$, is a continuous function in each knot, for each $d = 1, \ldots, q - 1$ (Quarteroni et al. 2000; Hastie et al. 2009).

Indeed, a $q + 1$-order B-spline basis is a basis for the $q + 1$-order spline function space on a given sequence of knots, that is, any spline function of order $q + 1$ can be represented as a linear combination of $B$-splines, and unlike other bases, the truncated power basis, for example, is very attractive numerically (Quarteroni et al. 2000; Hastie et al. 2009).

Once chosen, the order ($q + 1$) and the interior knots $T_j$, $j = 1, \ldots, K$,of a spline, because we need $K + 1$ polynomials (one for each of the $K + 1$ intervals) and $Kq$ constraints (continuity of the B-spline in its interior knots + continuity of derivatives of order $q - 1$ in each knot), the number of basis functions is given by

$$L = (q + 1)(K + 1) - Kq = q + K + 1 = \text{order} + \text{number of interior knots}.$$

Practically, the position of the knots can be chosen according to the data change points or by allowing the observation time to determine their positions at appropriate percentiles (Fig. 14.2). For example, in R, if we want a B-spline of order 4 on the interval (0, 12) with three specific interior knots ($T_1 = 3$, $T_2 = 6$, $T_3 = 9$), this can be defined, plotted, and evaluated by applying the following code:

```
Order = 4 ; breaks = seq(0,12,3)
BS = create.bspline.basis(rangeval=c(0,12),norder=Order, breaks=breaks)
#Number of basis functions: 4 + 3
BS$nbasis
#Graphic of the seven basis functions
plot(BS,xlab='t')

#Evaluation of these seven basis functions in
tv= seq(0,12, length=100)
EBS = eval.basis(tv, basisobj=BS)
head(EBS)
matplot(tv,EBS,add=TRUE,type='o',pch='+')
```

Alternatively, if we want a B-spline basis of some order ($q + 1$) with a specific number of basis ($L_2$), in R it can be obtained similarly as before (Fig. 14.3). For example, a B-spline of order 4 with six bases will contain two (6–4) interior knots equally spaced and can be obtained with the following code:

```
Order = 4; nbasis = 6
BS = create.bspline.basis(rangeval=c(0,12),norder = Order,nbasis =
nbasis)
#Graphic of the 6 basis functions
plot(BS,xlab='t')
#Evaluation of this 6 basis functions in
tv= seq(0,12,length=100)
EBS = eval.basis(tv, basisobj=BS)
head(EBS)
matplot(tv,EBS,add=TRUE,type='o',pch='+')
```

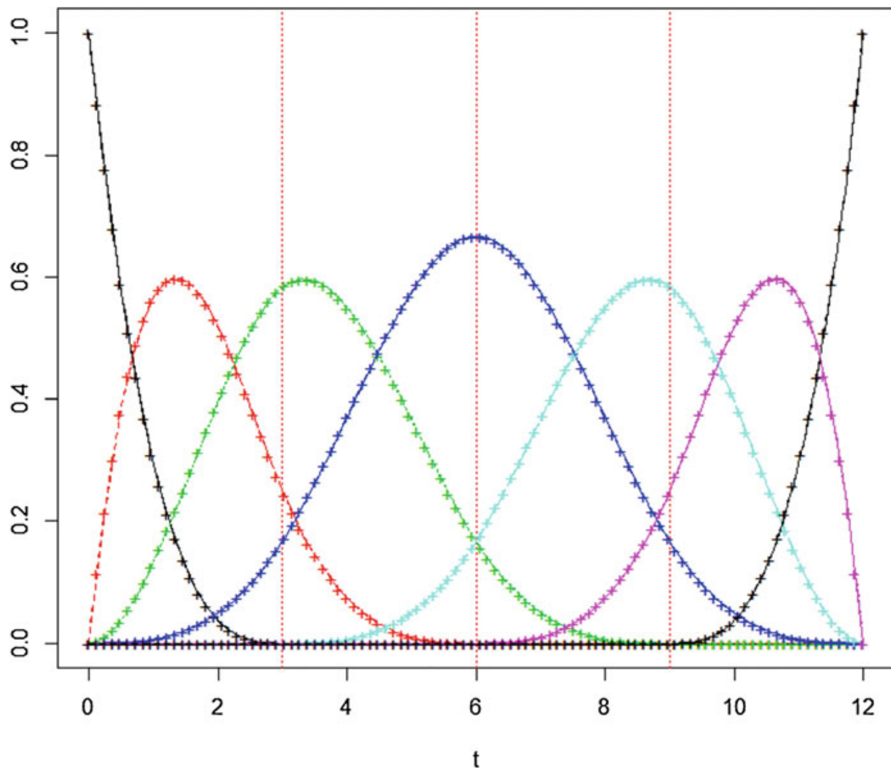



**Fig. 14.2** Graphic of the seven B-spline basis functions of order 4 on interval (0,12) and interior knots $T_1 = 3$, $T_2 = 6$, and $T_3 = 9$

A B-spline basis of the same order and the same number of basis functions (Fig. 14.4), but with specific positions of the interior knots ($T_1 = 2$ and $T_2 = 7$), can be obtained by adding the argument breaks = c(0,2,7,12) to the function create. bspline.basis:

```
breaks = c(0,2,7,12)
Order = 4; nbasis = 6
BS = create.bspline.basis(rangeval=c(0,12),norder = Order,nbasis =
nbasis,
            breaks= breaks)
#Graphic of the six basis functions
plot(BS,xlab='t')
```

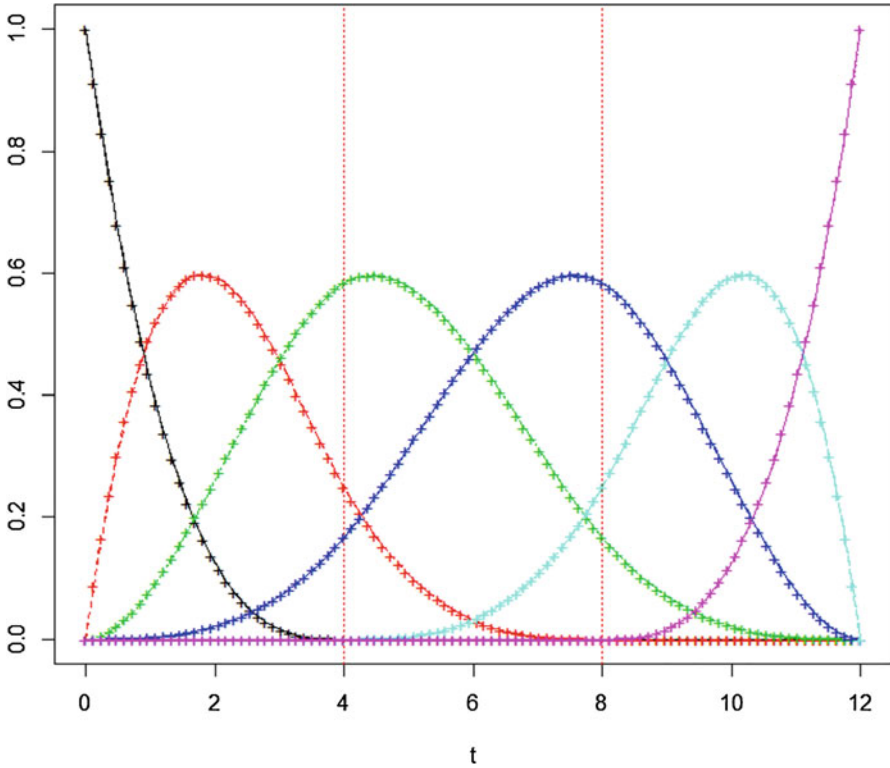



**Fig. 14.3** Graphic of the six B-spline basis functions of order 4 on interval (0,12)

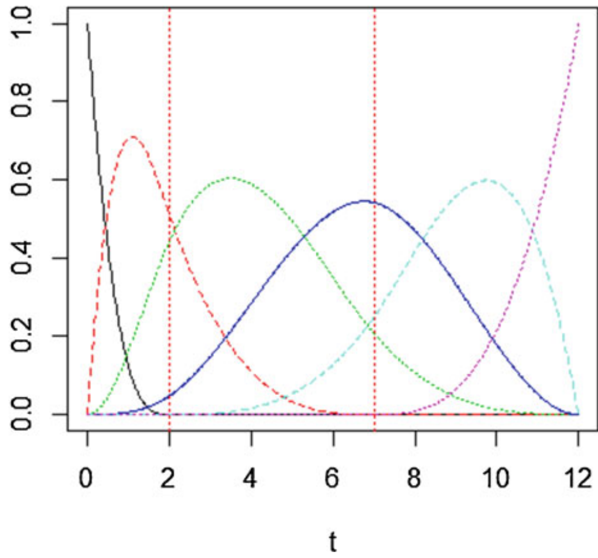



**Fig. 14.4** Graphic of the six B-spline basis functions of order 4 on interval (0,12) with specific interior knots: $T_1 = 2$ and $T_2 = 7$

## 14.3 Illustrative Examples

**Example 14.1**
To illustrate how to use and get a better picture of the performance of Fourier and B-spline basis to represent functions, suppose that there is information on only 30 (tv) equispaced evaluations (xv) of an unknown function in interval (0,12):

tv = seq(0,12,length = 30)

xv = c(0.9, 0.924, 0.9461, 0.9658, 0.983, 0.9971, 1.008, 1.0152, 1.0187, 1.0181, 1.0133, 1.0042, 0.9906, 0.9727, 0.9504, 0.9237, 0.8928, 0.8579, 0.8192, 0.777, 0.7314, 0.683, 0.632, 0.5788, 0.5238, 0.4675, 0.4103, 0.3526, 0.2949, 0.2376).

The graphical representation of this information is given in Fig. 14.5, together with three representations of this using Fourier basis (5, 21, and 29 basis functions) with period 30 (range of the observation domain). From this we can see that a poor representation was obtained with five basis functions, while with the other numbers of basis used (21 and 29), almost equal and reasonable representations were obtained, except in the boundaries of the interval, which is related to the Gibbs

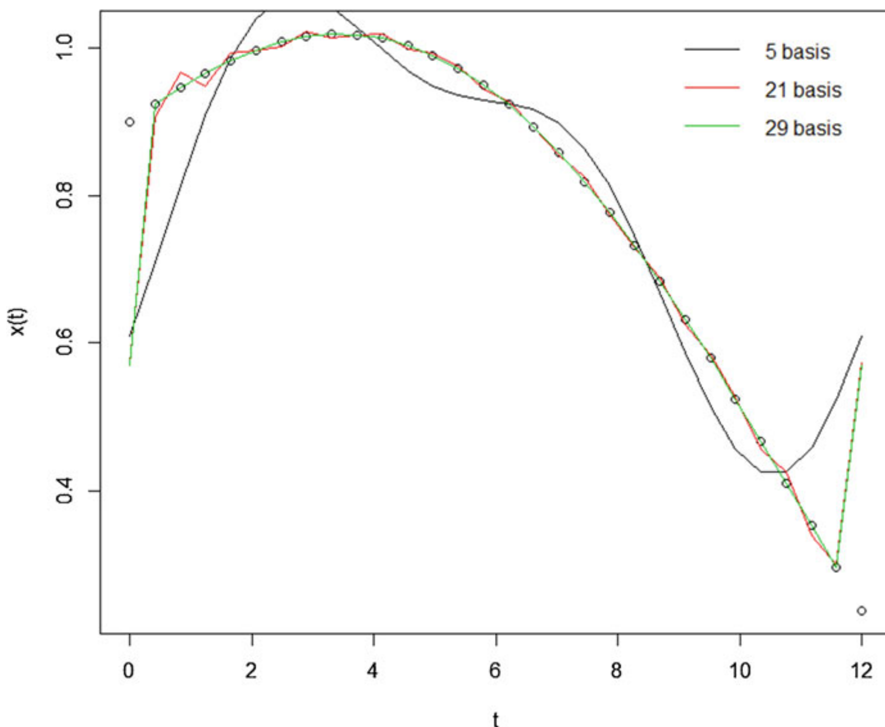



**Fig. 14.5** Graphical representation of 30 evaluations (points) of a function in 30 equispaced time points in interval (0, 12), and representation of this using 5 (in black), 21 (in red), and 29 (in green) Fourier basis functions

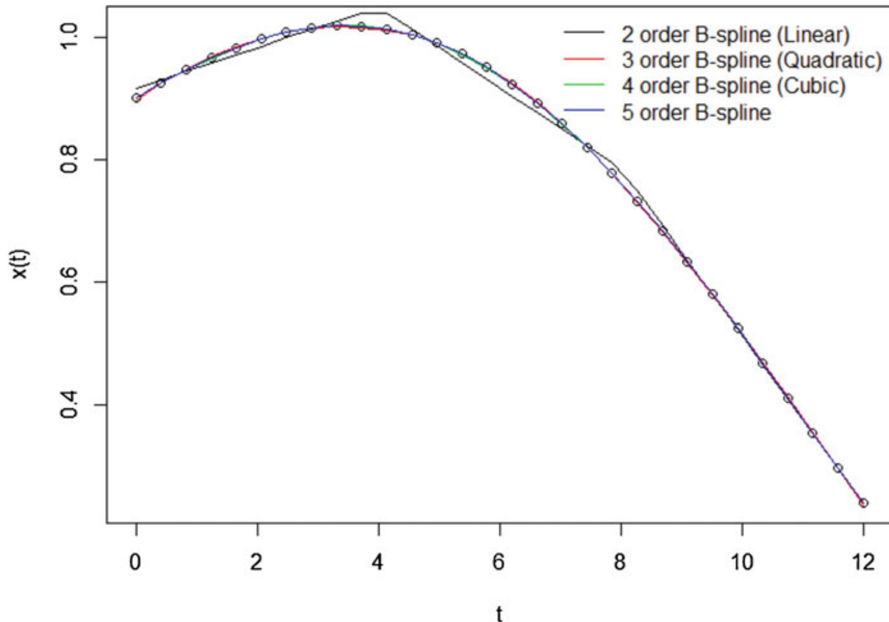



**Fig. 14.6** Graphical representation of 30 evaluations (points) of a function in 30 equispaced time points in interval (0, 12), and representation of this using B-splines of order 2 (shown in black), 3 (in red), 4 (in green), and 5 (in blue), each with two interior knots at $T_1 = 4$ and $T_2 = 8$

phenomenon, a spurious oscillation at the interval boundaries of an expansion in a Fourier series of a nonperiodic function (Shizgal and Jung 2003).

Figure 14.6 shows the same data but now together with representations of the latent function obtained by using B-splines of order 2, 3, 4, and 5, all with two interior knots ($T_1 = 4$ and $T_2 = 8$), to which correspond $2 + 2 = 4$, $2 + 3 = 5$, $2 + 4 = 6$, and $2 + 5 = 7$ basis functions, respectively. From this figure we can observe that the representation of order 3 is satisfactory, and this almost coincides with the representations obtained with orders 4 and 5.

```
#R code for B-spline representation
plot(tv,xv,type='p',xlab='t',ylab='x(t)')
breaks = seq(0,12,length=4)
Orderv = 2:5
for(i in 1:4)
{
    #A linear B-spline with two interior knots
    Order = Orderv[i]
    Degree = Order-1
    #No of basis functions= Order + length(breaks)-2
    nB = Order + length(breaks)-2
    BBS = create.bspline.basis(rangeval=c(0,12),norder=Order,
    breaks=breaks)
```

```
    EBBS = eval.basis(tv, basisobj=BBS)
    cv  = solve(t(EBBS)%*%EBBS)%*%t(EBBS)%*%xv
    xv_p = EBBS%*%cv
    lines(tv,xv_p,col=i)
}
legend('topright',c('2 order B-spline (Linear)', '3 order B-spline
(Quadratic)',
          '4 order B-spline (Cubic)','5 order B-spline'),
   lty=c(1,1,1,1),col=1:4,bty='n')
```

Now, by using 3, 5, 7, and 10 B-spline basis of orders 2, 3, 4, and 5, respectively, the resulting representations are shown in Fig. 14.7. From this we can see that with five B-spline basis of order 3 (only two interior knots equally spaced were required), the representation started to be satisfactory, indicating flexibility in the B-spline basis.

R code to reproduce Fig. 14.7

```
plot(tv,xv,type='p',xlab='t',ylab='x(t)')
Orderv = 2:5
#No of basis functions = Order + length(breaks)-2
nBv = c(3,5,7,10)
#Interior points
K = nBv - Orderv
for(i in 1:4)
{
    BBS = create.bspline.basis(rangeval=c(0,12),nbasis=nBv[i],
                    norder=Orderv[i])
    EBBS = eval.basis(tv, basisobj=BBS)
    cv  = solve(t(EBBS)%*%EBBS)%*%t(EBBS)%*%xv
    xv_p = EBBS%*%cv
```

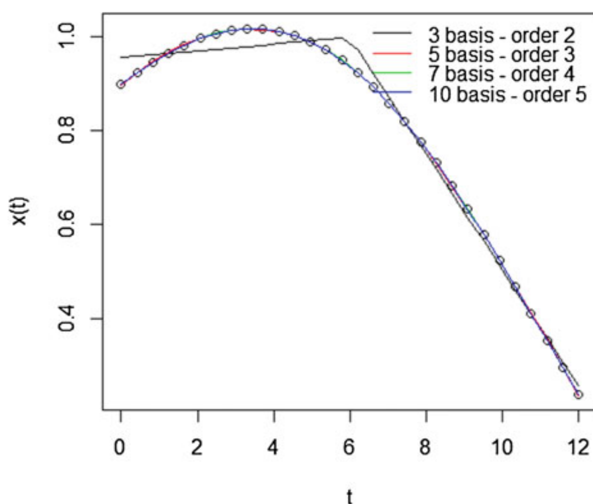



**Fig. 14.7** Graphical representation of 30 evaluations (points) of a function in 30 equispaced time points in interval (0,12), and B-spline representation using 3, 5, 7, and 10 basis functions of orders 2, 3, 4, and 5, respectively

```
    lines(tv,xv_p,col=i)
}
legend('topright',paste(nBv,' basis - order ', Orderv,sep=''),
   lty=c(1,1,1,1),col=1:4,bty='n')
```

In general, more flexible curves can be obtained by increasing the order or the number of knots in the B-spline. However, overfitting and an increase in the variance can occur if the number of knots is increased, while an inflexible function with more bias may result by decreasing the number of knots (Perperoglou et al. 2019).

**Example 14.2**

Now to illustrate how to use the BIC and LOOCV as criteria to choose the number of basis functions in a Fourier or B-spline (for a chosen order) representation, we retake the data points in Example 14.1 but perturbed by a random Gaussian noise (see Fig. 14.8).

For these data, Fig. 14.9 shows the value of the BIC criterion corresponding to the use of different numbers of basis functions in Fourier and B-spline representations. In both cases, the lowest value of this criterion was obtained with five basis functions and in all cases, the BIC criterion was better with the B-spline. The representation of the data points with this optimal number of basis functions is also shown in Fig. 14.8, where we can visually judge the better representation of the B-spline, because this resembles the non-perturbed latent function presented in Example 14.1. Similar results were obtained when using the LOOCV strategy: 7 and 6 basis for Fourier and B-spline were required, respectively, and the implied representation can also be observed in Fig. 14.9. These figures can be reproduced by the following R code:

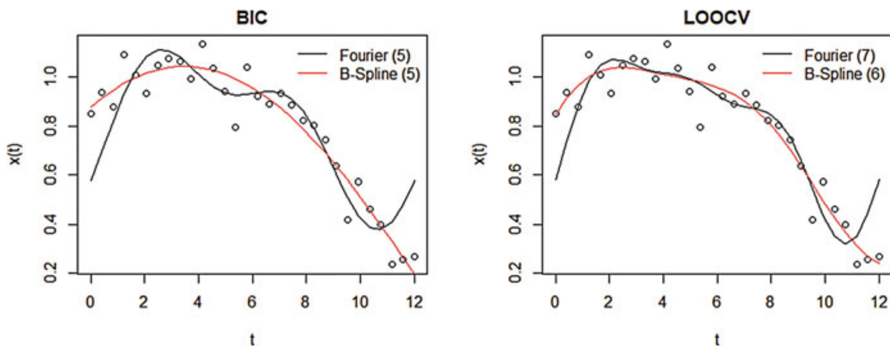



**Fig. 14.8** Graphical representation of the perturbed 30 data points in Example 14.1 and Fourier and B-spline representations using 5, 6, and 7 basis functions ($L_2$) in both. The left panel is the optimal representation obtained with the BIC and the right panel corresponds to the optimal representation obtained with the LOOCV

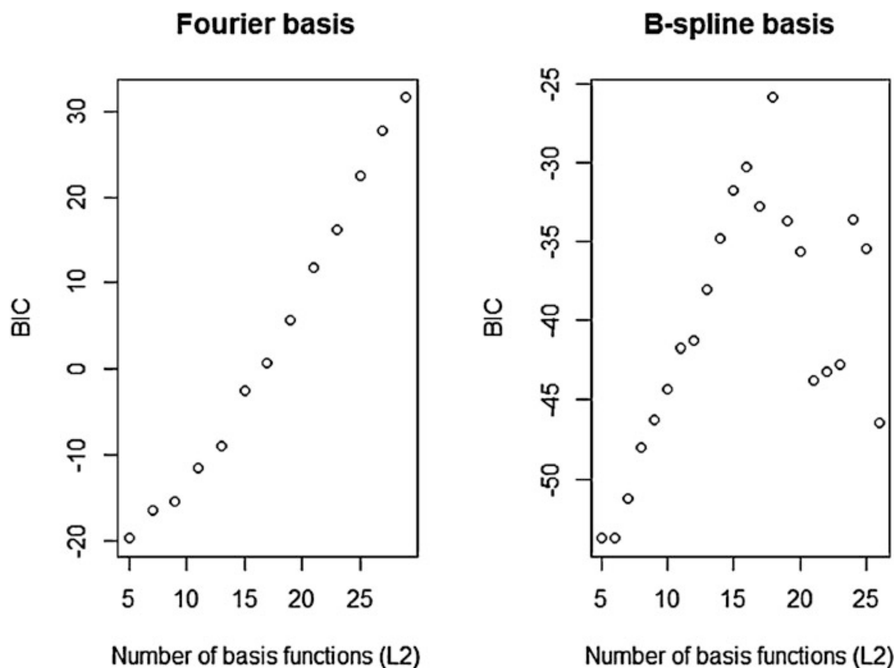



**Fig. 14.9** Behavior of the BIC for different numbers of basis *functions* ($L_2$) in Fourier and B-spline representations, for the perturbed data set of Example 14.1

```
rm(list=ls())
#------
#Example 14.2: Perturbed data points of Example 14.1
#------
library(fda)
#Time points where the functions was observed
tv = seq(0,12,length=30)
#Values of the function in 30 points
xv = c(0.9, 0.924, 0.9461, 0.9658, 0.983, 0.9971, 1.008, 1.0152, 1.0187,
1.0181, 1.0133, 1.0042, 0.9906, 0.9727, 0.9504, 0.9237, 0.8928, 0.8579,
0.8192, 0.777, 0.7314, 0.683, 0.632, 0.5788, 0.5238, 0.4675, 0.4103,
0.3526, 0.2949, 0.2376)
set.seed(1)
xv = xv + rnorm(length(xv),0,0.10*mean(xv))
#plot(tv,xv,type='p',xlab='t',ylab='x(t)')
#Fourier
library(fda)
m = length(xv)
nbFv = seq(5,m-1,2)
BICFv = rep(0, length(nbFv))
AICFv = BICFv
for(l in 1:length(nbFv))
{ BF=create.fourier.basis(rangeval=c(0,12),nbasis=nbFv[l],
period=diff(range(tv)))
```

```
 EBF = eval.basis(tv, basisobj=BF)
 cv = solve(t(EBF)%*%EBF)%*%t(EBF)%*%xv
 xv_p = EBF%*%cv
 sigma2 = mean((xv-xv_p)^2)
 ll = sum(dnorm(xv,xv_p,sqrt(sigma2),log = TRUE))
 BICFv[l] = -2*ll+(dim(EBF)[2]+1)*log(m)
 AICFv[l] = -2*ll+2*(dim(EBF)[2]+1)
}
#B-spline
library(fda)
Order = 4
#No of basis functions = Order + length(breaks)-2
nbBSv = (Order+1):(m-4)
BICBSv = rep(0,length(nbBSv))
for(l in 1:length(nbBSv))
{
 BBS = create.bspline.basis(rangeval=c(0,12),norder=Order,nbasis =
 nbBSv[l])
 EBBS = eval.basis(tv, basisobj=BBS)
 cv = solve(t(EBBS)%*%EBBS)%*%t(EBBS)%*%xv
 xv_p = EBBS%*%cv
 muv = EBBS%*%cv
 sigma2 = mean((xv-muv)^2)
 ll = sum(dnorm(xv,muv,sqrt(sigma2),log = TRUE))
 BICBSv[l] = -2*ll+(nbBSv[l]+1)*log(m)
}
#Behavior of the BIC for different models obtained using various
#number of Fourier or B-spline basis functions
par(mfrow=c(1,2))
plot(nbFv,BICFv,xlab='Number of basis functions (L2)', ylab='BIC',
   main='Fourier basis')
plot(nbBSv,BICBSv,xlab='Number of basis functions (L2)',ylab='BIC',
   main='B-spline basis')
#Fourier and B-spline representations with optimal number of
#basis functions as chosen by BIC
#Fourier
nboF_BIC = nbFv[which.min(BICFv)]
nboF_BIC
plot(tv,xv,type='p',xlab='t',ylab='x(t)',
   main='')#Optimal representation using BIC')
BF = create.fourier.basis(rangeval=c(0,12),nbasis=nboF_BIC,
              period=diff(range(tv)))
EBF = eval.basis(tv, basisobj=BF)
cv = solve(t(EBF)%*%EBF)%*%t(EBF)%*%xv
xv_p = EBF%*%cv
lines(tv,xv_p,col=1)
#B-spline
nboB_BIC = nbBSv[which.min(BICBSv)]
nboB_BIC
BBS = create.bspline.basis(rangeval=c(0,12),norder=Order,
              nbasis = nboB_BIC)
EBBS = eval.basis(tv, basisobj=BBS)
cv = solve(t(EBBS)%*%EBBS)%*%t(EBBS)%*%xv
```

```
xv_p = EBBS%*%cv
lines(tv,xv_p,col=2)
legend('topright',paste(c('Fourier (','B-Spline ('),
           c(nboF_BIC,nboB_BIC),c(')',')'),sep=''),
   col=1:2,lty=rep(1,2),bty='n')
#Choosing the optimal number of basis functions using 1FCV
PRESS_f<-function(A)
{
 Res = residuals(A)
 sum((Res/(1-hatvalues(A)))^2)
}
#Fourier basis
CVFv = nbFv
for(l in 1:length(nbFv))
{  BF=create.fourier.basis(rangeval=c(0,12),nbasis=nbFv[l],
period=diff(range(tv)))
 EBF = eval.basis(tv, basisobj=BF)
 A = lm(xv~0+EBF)
 CVFv[l] = PRESS_f(A)
}
plot(nbFv,CVFv,xlab='No. of basis functions',ylab='PRESS')
nboF = nbBSv[which.min(CVFv)]
nboF

#B-spline basis
CVBSv = nbBSv
for(l in 1:length(nbBSv))
{
 BBS = create.bspline.basis(rangeval=c(0,12),norder=Order,nbasis =
 nbBSv[l])
 EBBS = eval.basis(tv, basisobj=BBS)
 A = lm(xv~0+EBBS)
 CVBSv[l] = PRESS_f(A)
}
plot(nbBSv[1:5],CVBSv[1:5],xlab='No. of basis',ylab='PRESS')
nboBS = nbBSv[which.min(CVBSv)]
nboBS

#Fourier and B-spline representations with optimal number of
#basis functions as choosen by BIC
par(mfrow=c(1,2))
plot(tv,xv,type='p',xlab='t',ylab='x(t)',
  main='BIC')
#Fourier
BF = create.fourier.basis(rangeval=c(0,12),nbasis=nboF_BIC,
            period=diff(range(tv)))
EBF = eval.basis(tv, basisobj=BF)
cv = solve(t(EBF)%*%EBF)%*%t(EBF)%*%xv
xv_p = EBF%*%cv
lines(tv,xv_p,col=1)
#B-spline
BBS = create.bspline.basis(rangeval=c(0,12),norder=Order,
             nbasis = nboB_BIC)
```

```
EBBS = eval.basis(tv, basisobj=BBS)
cv = solve(t(EBBS)%*%EBBS)%*%t(EBBS)%*%xv
xv_p = EBBS%*%cv
lines(tv,xv_p,col=2)
legend('topright',paste(c('Fourier (','B-Spline ('),c(5,5),
           c(')',')'),sep=''),
    col=1:2,lty=rep(1,2),bty='n')

#Optimal representation with 1FCV
plot(tv,xv,type='p',xlab='t',ylab='x(t)',
  main='LOOCV')
#Fourier
BF = create.fourier.basis(rangeval=c(0,12),nbasis=nboF,
            period=diff(range(tv)))
EBF = eval.basis(tv, basisobj=BF)
cv = solve(t(EBF)%*%EBF)%*%t(EBF)%*%xv
xv_p = EBF%*%cv
lines(tv,xv_p,col=1)
#B-spline
BBS = create.bspline.basis(rangeval=c(0,12),norder=Order,
            nbasis = nboBS)
EBBS = eval.basis(tv, basisobj=BBS)
cv = solve(t(EBBS)%*%EBBS)%*%t(EBBS)%*%xv
xv_p = EBBS%*%cv
lines(tv,xv_p,col=2)
legend('topright',paste(c('Fourier (','B-Spline ('),c(nboF,nboBS),
           c(')',')'),sep=''),
    col=1:2,lty=rep(1,2),bty='n')
```

**Example 14.3**

Now we will consider the prediction of wheat grain yield (tons/ha) using hyper-spectral image data. For this example, we consider part of the data used in Montesinos-López et al. (2017a, b): 20 lines and three environments. For each individual plant, the reflectance, $x(t_j)$, of its leaves was measured at $m = 250$ wavelengths (from 392 to 851 nm were measured) and at different stages of its growth, but the information used here corresponds to one of these stages.

Figure 14.10 shows the measured reflectance corresponding to 60 observations, where the colors of these observations indicate that they belong to the same environment. The Fourier and B-spline representations of all these curves are shown in Fig. 14.11, where the number of basis used in each case were 29 and 16, respectively; they are the medians of the most frequently selected number of basis functions (29 for Fourier, and 12, 16, and 73 for B-spline) by the BIC across all the curves (see Appendix 1 for the R code to reproduce these results).

The observed values versus the predicted values of the response, corresponding to the Fourier and B-spline representations of the covariate, are shown in Fig. 14.12; in both cases, $L_1 = 21$ basis functions were used to represent the beta coefficient

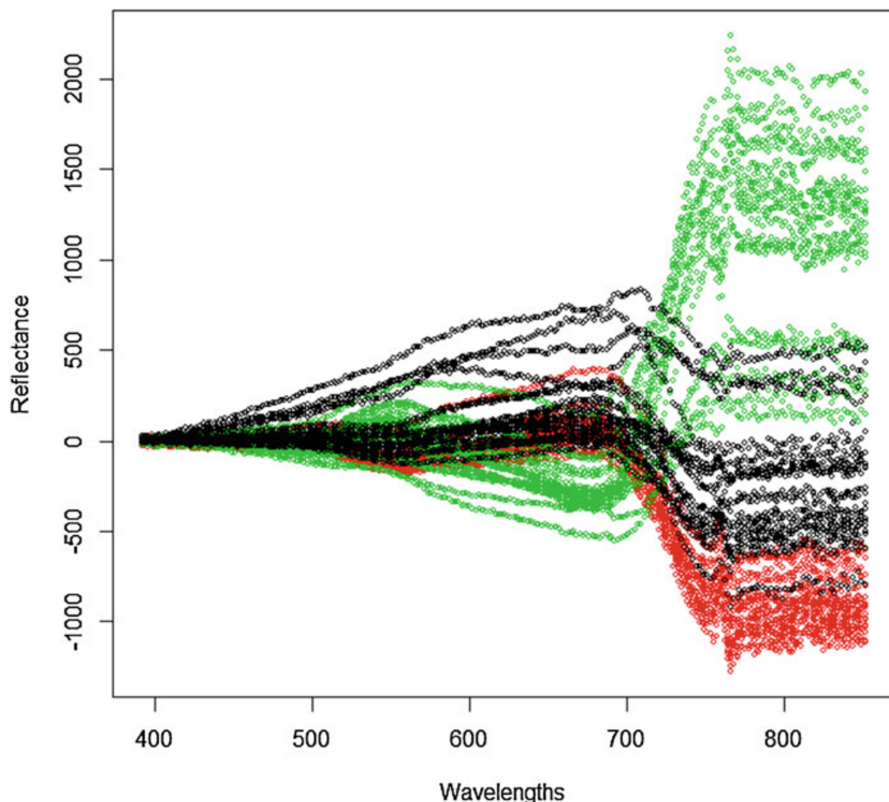



**Fig. 14.10** Reflectance of leaves of 60 individuals measured in 250 wavelengths (29, 394, 394, . . ., 851). The colors indicate the environment where the individuals were measured

function $\beta(t)$. The fitted model appears to give almost the same results with both representations.

To let the data speak for themselves about a reasonable value for $L_1$ to represent $\beta(t)$, the BIC was used in both representations. For the Fourier case, $L_1 = 11$ was the optimal value, while $L_1 = 14$ was the optimal value for the B-spline basis (see Appendix 1 for the R code). The predicted and residual values obtained with these optimal representations are shown in Fig. 14.13, from which it is difficult to choose the best one because they gave almost the same results.

Because the B-spline appears to give a better covariate representation (see Fig. 14.11) and a similar predicted value as that given by the Fourier basis (Fig. 14.13), we can take a more informed decision in terms of the prediction accuracy of the response, with both representations. To this end, we used ten random partitions, where in each partition, 20% of the total data set was used to measure prediction accurary and the rest was used to fit (train) the model. The results are shown in Table 14.1 and we can see that, on average, the Fourier basis was favored,

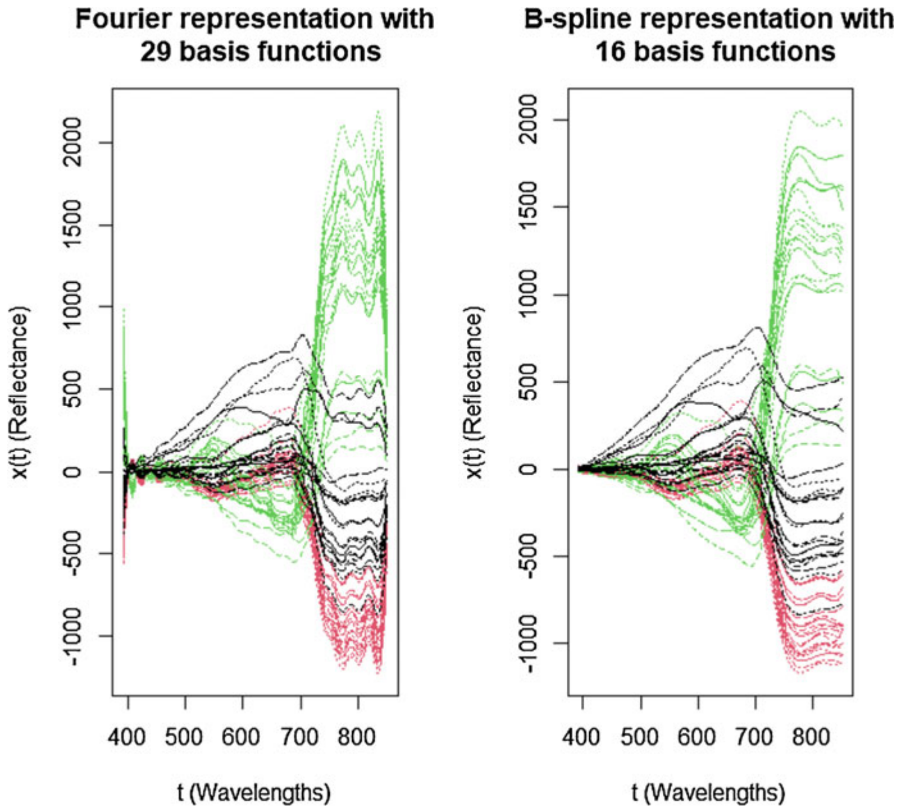



**Fig. 14.11** Fourier and B-spline representations with the "optimal" number of basis *functions* obtained with the BIC across all the curves

because on average across all the partitions, this type of basis provided lower MSE in 6 out of 10 partitions; for this reason, it is considered the best option. The same conclusion was reached by comparing the BIC values of the corresponding Fourier (149.8048) and B-spline (159.5854) representations.

## 14.4 Functional Regression with a Smoothed Coefficient Function

As mentioned earlier, in the representation of the functional predictor ($x(t)$), one way to control the smoothness when determining the beta coefficient function, $\beta(t)$, is by introducing a regularization term:

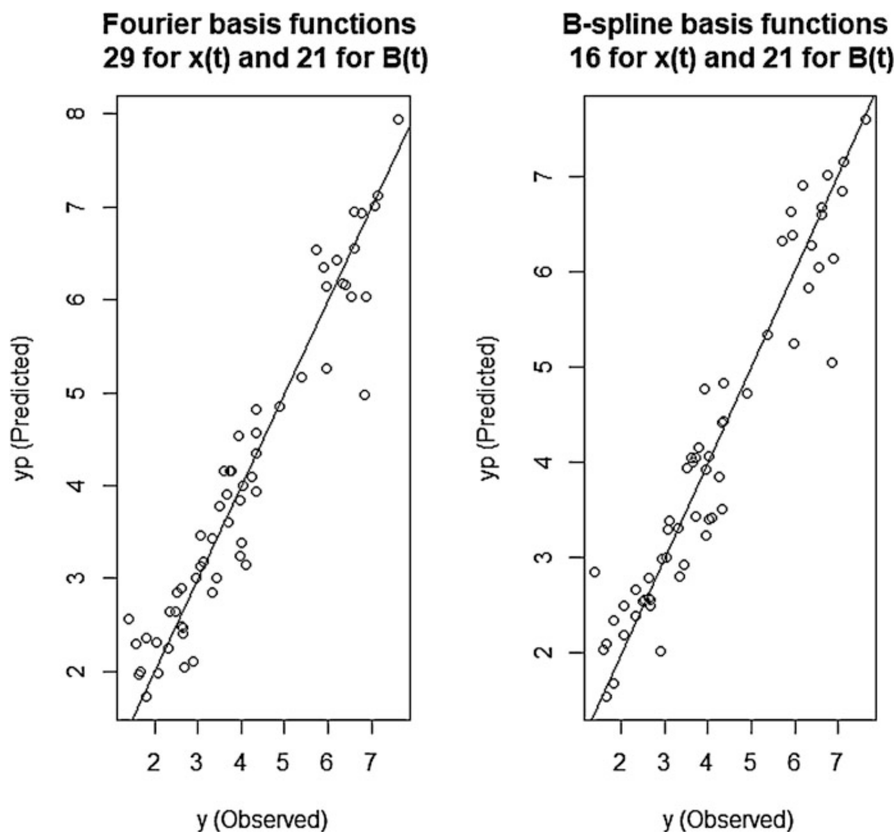



**Fig. 14.12** Graph displaying the observed versus fitted values with two functional regression models: the left panel was obtained by using 29 Fourier basis *functions* for the covariate function and 21 Fourier basis *functions* for $\beta(t)$; the right panel was obtained by using 16 B-spline basis *functions* for the covariate function and 21 B-spline basis *functions* for $\beta(t)$

$$\mathrm{SSE}_{\lambda}(\boldsymbol{\beta}) = \sum\nolimits_{i=1}^{n} \left( y_i - \mu - \sum\nolimits_{l=1}^{L_1} x_{il}\beta_l \right)^2 + \lambda J_{\beta}, \tag{14.10}$$

where $J_\beta$ is the penalty term and $\lambda$ is a smoothing parameter that represents a compromise between the fit of the model to the data (first term) and the smoothness of the function $\beta(\cdot)$ (second term). When $\lambda = 0$, the problem is reduced to that of least squares (or maximum likelihood under normal errors) where there is no penalty, and when $\lambda$ increases, the roughness is highly penalized to the extent that $\beta(t)$ can be constant.

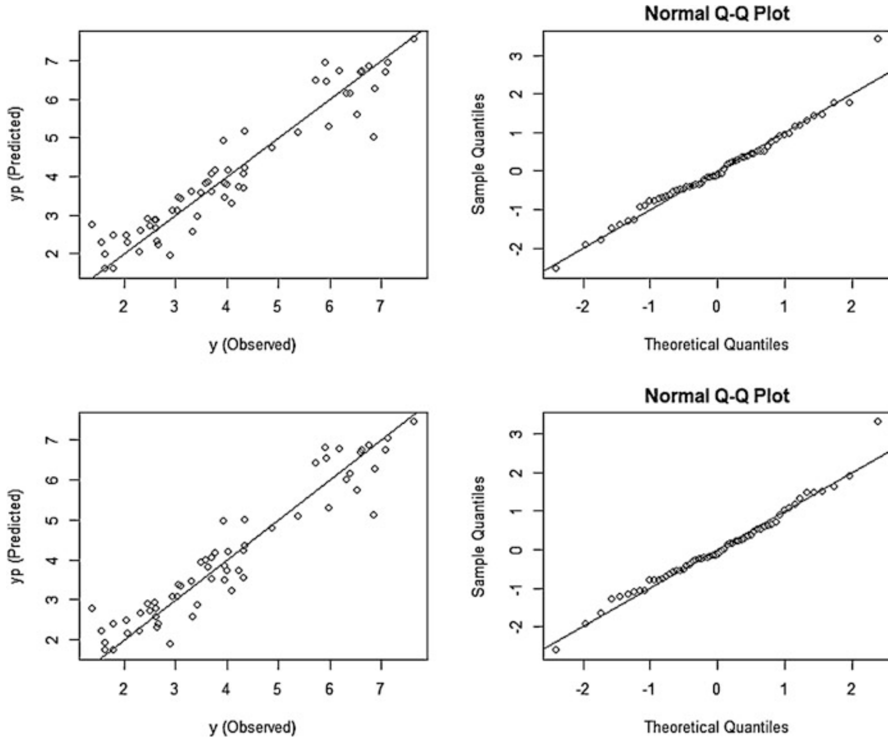



**Fig. 14.13** Observed versus predicted values and normal Q-Q plot of the residuals obtained with Fourier (above) and B-spline (below) representations of both sets of covariates ($L_2$=29 Fourier basis functions and $L_2$=16 B-spline basis functions) and beta coefficient functions ($L_1 = 11$ Fourier basis functions and $L_1 = 14$ B-spline Fourier basis functions)

**Table 14.1** Mean square error (MSE) of prediction for 10 random partitions for Fourier and B-spline representations

| Partition | Fourier | B-spline |
|---|---|---|
| | MSE | MSE |
| 1 | 0.1717 | 0.2993 |
| 2 | 0.6049 | 0.6487 |
| 3 | 0.4385 | 0.4559 |
| 4 | 0.5006 | 0.4910 |
| 5 | 0.6164 | 0.6048 |
| 6 | 0.5403 | 0.6453 |
| 7 | 0.1456 | 0.1396 |
| 8 | 0.5004 | 0.5737 |
| 9 | 0.7551 | 0.7435 |
| 10 | 0.8042 | 0.9250 |
| Average (SD) | 0.5077 (0.216) | 0.5526 (0.2221) |

Often the penalty term $J_\beta$ is based on the integrated $p$th order derivatives (Usset et al. 2016):

$$J_\beta = \int_0^T \left[\frac{d^p}{dt^p}\beta(t)\right]^2 dt, \tag{14.11}$$

where $\frac{d^p}{dt^p}\beta(t)$ is a derivative of order $p$ of the function $\beta(t)$. With the representation (14.2) of $\beta(t)$, $J_\beta$ can be expressed as

$$J_\beta = \boldsymbol{\beta}^{\mathrm{T}}\boldsymbol{P}\boldsymbol{\beta},$$

where $P$ is a square matrix with entries $P_{ij} = \int_0^T \phi_i^{(p)}(t)\phi_j^{(p)}(t)$, $i, j = 1, \ldots, L_1$, and $\phi_i^{(p)}(t)$ is a derivate of order $p$ of $\phi_i(t)$. Typical chosen values of $p$ are 1 and 2.

A smoothed solution of the function $\beta(t)$ can be obtained by minimizing (14.10) with respect to the parameters $\beta_l$, $l = 1, \ldots, L_1$. However, because this solution depends on the smoothing parameter, this needs to be determined. For this reason, as in the Ridge and Lasso regression models described in early chapters, here a cross-validation method is adopted first, and a Bayesian approach will be described later.

Under the penalty term (14.11), the penalized sum of squared errors (14.10) can be written as

$$\mathrm{SSE}_\lambda(\boldsymbol{\beta}) = ||\boldsymbol{y} - \mathbf{1}_n\mu - \boldsymbol{X}^*\boldsymbol{\beta}^*||^2 + \lambda\boldsymbol{\beta}^{*\mathrm{T}}\boldsymbol{D}\,\boldsymbol{\beta}^* = SSE_\lambda(\boldsymbol{\beta}^*), \tag{14.12}$$

where $\boldsymbol{X}^* = \boldsymbol{X}\boldsymbol{\Gamma}$, $\boldsymbol{\beta}^* = \boldsymbol{\Gamma}^{\mathrm{T}}\boldsymbol{\beta}$, and $\boldsymbol{P} = \boldsymbol{\Gamma}\mathbf{D}\boldsymbol{\Gamma}^{\mathrm{T}}$ is the spectral decomposition of the penalty matrix $\boldsymbol{P}$. Note that when the matrix $\boldsymbol{P}$ is not of full rank, the penalty term in (14.12) is reduced to $\lambda\boldsymbol{\beta}^{*\mathrm{T}}\boldsymbol{D}\,\boldsymbol{\beta}^* = \lambda\boldsymbol{\beta}_1^{*\mathrm{T}}\boldsymbol{D}_1\,\boldsymbol{\beta}_1^*$, where $\boldsymbol{D}_1$ is $\boldsymbol{D}$ but without the rows and columns corresponding to the eigenvalues equal to 0 of $\boldsymbol{P}$. So, the corresponding smoothed solution of $\beta(t)$ can be obtained as

$$\widehat{\beta}(t) = \sum_{l=1}^{L_1}\widehat{\beta}_l\phi_l(t),$$

where $\widehat{\boldsymbol{\beta}} = \boldsymbol{\Gamma}\widehat{\boldsymbol{\beta}}^*$ and $\widehat{\boldsymbol{\beta}}^*$ is the solution of (14.12), which also can be obtained with the *glmnet* R package.

**Example 14.4**

To exemplify the penalized estimation of functional regression (14.10) with penalty (14.11), here we retake the data used in Example 14.3. To compare the prediction accuracy of this with the non-penalized functional regression described in the previous section, 100 random partitions were used, and in each, 80% of the data set was used to train the model and the rest to evaluate the prediction performance. When training the model, an inner five-fold cross-validation was used to choose the optimal parameter ($\lambda$) and estimate the $\beta_l$'s coefficients. This was done using Fourier and B-spline basis in the representation of the beta function, and in both cases, two

basis were used. In both cases, the penalty matrix (14.11) and the elements of its spectral decomposition (14.12) can be computed in R as

```
#Penalty matrix of derivative of order p
P_mat = eval.penalty(basisobj =Phi,Lfdobj=p)
ei_P = eigen(P_mat)
#Espectral descompositin of P_mat
gamma = ei_P$vectors #Γ
dv = ei_P$values#Eigenvalues of P_mat, elements of diagonal of D
dv = ifelse(dv<1e-10,0,dv)
```

where Phi is a created basis in R (Fourier or B-spline) and $p$ is a nonnegative integer for the order of the derivative in the penalty matrix to be used. Once the penalty matrix is computed, the training of the model in (14.12) can be done in R as

```
Xa = X_F%*%gamma
A_PFR = cv.glmnet(x =Xa,y = y ,alpha = 0, nfolds=k, penalty.factor=dv,
        standardize=FALSE,maxit=1e6)
```

where Xa is $\boldsymbol{X}^{*}$, $y$ is the vector with the corresponding values of the response variable, $k$ is an integer used to specify the inner $k$ cross-validation to train the model and choose the "optimal" value of the smoothing parameter, and $dv$ are the eigenvalues of the penalty matrix used to indicate different penalties of the $\beta_j^*$ as required in (14.12), and standardize = FALSE to indicate the non-required standardization of the columns of $\boldsymbol{X}^{*}$. See Appendix 2 for a complete R code used to obtain the results of this example.

When using Fourier representation and first derivative penalization, for the 100 random partitions, in Fig. 14.14 is shown the MSE obtained with the penalized functional regression (PFR) against the MSE corresponding to the non-penalized functional regression (FR): in 78 out of 100 partitions, the PFR resulted in a better MSE (0.7465 vs. 0.5545 on average). Furthermore, in the partitions where the FR was better (20%), the average MSE of the PFR was 35.48% greater (0.4051 vs. 0.5561), while in those where the PFR was better, the average MSE obtained with the FR was 51.55% greater (0.8428 vs. 0.5561).

Now, for the B-spline representation and first derivative penalization, the corresponding results are also shown in Fig. 14.14. In this case, in 55 out of 100 partitions, the PFR resulted in a better MSE (0.5822 vs. 0.5630 on average). Furthermore, in the partitions where the FR was better (20%), the average MSE of the PFR was 27.18% greater (0.4887 vs. 0.6216), while in those where the PFR was better, the average MSE obtained with the FR was 27.87% greater (0.6587 vs. 0.5151). So, for the Fourier basis representation, the results obtained when using penalization in the estimation of the beta function $\beta(\cdot)$ differ from those obtained when no penalization is used, while with the B-spline representation the difference is negligible. This is perhaps because of the natural smoothing of the B-spline relative to the Fourier basis.

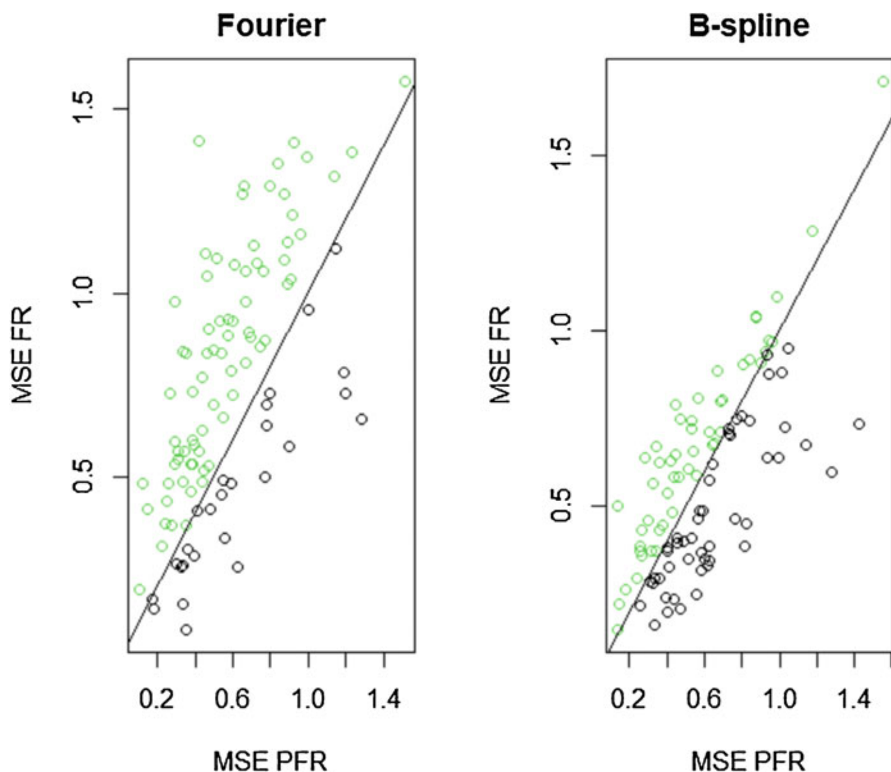



**Fig. 14.14** Mean square error (MSE) of prediction for penalized functional regression (PFR) versus MSE of functional regression (FR). Shown in green are the cases where the MSE obtained with the PFR is less than the FR

For second derivative penalization using Fourier representation, in 74 out of 100 partitions and on average, the MSE of PFR also resulted better than the FR (0.7465 vs. 0.5754). Indeed, in the cases where PFR was worse, the average MSE of this was 34.87% greater than the corresponding FR, and in the cases where FR was worse, the average MSE was 51.78% greater.

For B-spline basis with second derivative penalization, on average the FR was better than the PFR (0.5822 vs. 0.6009), but this resulted in a smaller MSE only in 50 out of 100 partitions. Additionally, in the case where FR was better, the average MSE of the PFR was 34.48% greater than the average of FR, while in the other half of the cases, the average MSE of the FR was only 24.25% greater than the average MSE of PFR.

## 14.5 Bayesian Estimation of the Functional Regression

Similar to what was done in Chaps. 3 and 6, the penalized sum squared of error solution in (14.12) with penalty term (14.11) coincides with the mean/mode of the posterior distribution of $\boldsymbol{\beta}$, in the multiple linear regression $y_i = \mu + \sum_{l=1}^{L_1} x_{il}\beta_l + \epsilon_i$ ($\boldsymbol{y} = \mathbf{1}_n\mu + \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$), with prior distribution $\boldsymbol{\beta} \sim N\left(\mathbf{0}, \sigma_\beta^2 \boldsymbol{P}^{-1}\right)$, with $\sigma_\beta^2 = \frac{\lambda}{\sigma^2}$. So, from here a Bayesian formulation (PBFR) for the smoothed solution of the coefficient function ($\beta(t)$) in the functional regression model (14.1) can be completed by assuming the following priors for the rest of the parameters: $\sigma^2 \sim \chi_{v,S}^{-2}$ and $\sigma_\beta^2 \sim \chi_{v_\beta,\ S_\beta}^{-2}$, where $\chi_{v,S}^{-2}$ denotes a scaled inverse Chi-squared distribution with shape parameter $v$ and scale parameter $S$. When $\boldsymbol{P}$ is not of full rank, little change is needed over the prior distribution of the $\boldsymbol{\beta}$ coefficients: for example, in the Fourier basis, the first element of this is the constant function, so entries in the first row and first column of $\boldsymbol{P}$ are equal to 0.

Similarly, the Bayesian formulation of regression models can be expressed as $\boldsymbol{y} = \mathbf{1}_n\mu + \boldsymbol{X}^*\boldsymbol{\beta}^* + \boldsymbol{\epsilon}$, with the same prior distribution, except that now $\boldsymbol{\beta}^* \sim N\left(\mathbf{0}, \sigma_\beta^2 \boldsymbol{I}_{L_1}\right)$ and $\boldsymbol{X}^* = \boldsymbol{X}\boldsymbol{\Gamma}\boldsymbol{D}^{-1/2}$, where $\boldsymbol{D}^{-\frac{1}{2}}$ is the inverse of the squared root matrix of $\boldsymbol{D}$. This equivalence is the same as Bayesian Ridge Regression (BRR) described in Chap. 6, so this can be implemented with the BGLR R package.

**Example 14.5**

Here we continue with the data set of Example 14.3, but now in another 100 random partitions, we added the Bayesian prediction to explore the prediction performance. Part of the code for implementing this model is given next, but the complete code used is given in **Appendix 3**.

```
#Number of Fourier and B-spline basis functions to represent the
covariable
#function
nbFo = 29
nbBSo = 16

#Functional regression with Fourier and B-splines
#Computing X for Fourier representation with L1 = 21 Fourier basis
functions
#for the beta function
L1 = 21
P = diff(range(Wv))
Psi_F = create.fourier.basis(range(Wv), nbasis = nbFo,
              period = P)
SF_mat = smooth.basisPar(argvals = Wv,
            y =t(X_W), fdobj = Psi_F,lambda=0,
            Lfdobj = 0)
Phi_F = create.fourier.basis(range(Wv), nbasis = L1,
              period = P)
```

```
Q   = inprod(Phi_F,Psi_F)
c_mat = t(SF_mat$fd$coefs)
X_F  = c_mat%*%t(Q)
dim(X_F)

#Computing X for B-spline representation with L1 = 21 B-spline basis
functions
#for the beta function
L1 = 21
Psi_BS = create.bspline.basis(range(Wv), nbasis = nbBSo)
SBS_mat = smooth.basisPar(argvals = Wv,
              y =t(X_W), fdobj = Psi_BS,lambda=0,
              Lfdobj = 0)
Phi_BS = create.bspline.basis(range(Wv), nbasis = L1,
                norder= 4)
Q   = inprod(Phi_BS,Psi_BS)
c_mat = t(SBS_mat$fd$coefs)
X_BS  = c_mat%*%t(Q)
dim(X_BS)
```

**#Smoothing estimation using Fourier representation for the functional**
**#covariable and L1 = 21 Fourier basis functions for the beta function**

```
par(mfrow=c(1,2))
library(glmnet)
#Penalization with first derivate
P_mat_F = eval.penalty(Phi_F,Lfdobj=1)
ei_P = eigen(P_mat_F)
gamma = ei_P$vectors
dv = ei_P$values
dv = ifelse(dv<1e-10,0,dv)

X_Fa = X_F%*%gamma

#Grid of lambda values obtained by varying the proportion of variance
explained
#by the functional predictor
lamb_FR_f<-function(Xa,dv,K=100,li=1e-1,ls=1-1e-12)
{
 Pos = which(dv<1e-10)
 D_inv = diag(1/dv[-Pos])
 PEV = seq(li,ls,length=K)
 Xa  = Xa[,-Pos]
 lambv = (1-PEV)/PEV*mean(diag(Xa%*%D_inv%*%t(Xa)))
 lambv = exp(seq(min(log(lambv)),max(log(lambv)),length=K))
 sort(lambv,decreasing = TRUE)
}
lambda = lamb_FR_f(X_Fa,dv,K=1e2)

y=yv
library(BGLR)
#Linear predictor specification in BGLR for the PBFR model
Pos = which(dv>0)
```

```
#Matrix design for the non-penalized columns of X^a
X_Fa1 = X_Fa[,-Pos]
#Matrix design for the penalized columns of X^a
X_Fa2 = X_Fa[,Pos]%*%diag(1/sqrt(dv[Pos]))
ETA = list(list(X=X_Fa1,model='FIXED'),list(X=X_Fa2,model='BRR'))

#Linear predictor specification in BGLR for the BFR model
ETA_NP = list(list(X=X_F[,1],model='FIXED'),list(X=X_F[,-1],
model='BRR'))

#Random cross-validation
RP = 100
set.seed(1)
MSEP_df = data.frame(RP=1:RP,MSEP_PFR=NA,MSEP_FR=NA,lamb_o=NA)
for(p in 1:RP)
{
 Pos_tst = sample(n,0.20*n)
 X_F_tr = X_F[-Pos_tst,]; n_tr = dim(X_F_tr)[1]
 y_tr  = y[-Pos_tst]; y_tst = y[Pos_tst]

 #FR
 dat_df = data.frame(y=y,X=X_F)
 A_F = lm(y~.,data=dat_df[-Pos_tst,])
 yp_tst = predict(A_F,newdata = dat_df[Pos_tst,])
 MSEP_df$MSEP_FR[p] = mean((y_tst-yp_tst)^2)

 #PFR with first derivative
 A_PFR = cv.glmnet(x =X_Fa[-Pos_tst,],y = y[-Pos_tst] ,alpha = 0,
          nfolds=5,lambda=lambda/n_tr,
          penalty.factor=dv,
          standardize=FALSE,maxit=1e6)
 MSEP_df$lamb_o[p] = A_PFR$lambda.min
 yp_tst = predict(A_PFR,newx=X_Fa[Pos_tst,], s="lambda.min")[,1]
 MSEP_df$MSEP_PFR[p] = mean((y_tst-yp_tst)^2)

 #BGLR
 y_NA = y
 y_NA[Pos_tst] = NA
 A_BGLR = BGLR(y_NA,ETA= ETA,nIter=1e4, burnIn = 1e3,verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR[p] = mean((y_tst-yp_tst)^2)

 A_BGLR = BGLR(y_NA,ETA= ETA_NP,nIter=1e4, burnIn = 1e3,
verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR_NP[p] = mean((y_tst-yp_tst)^2)

 cat('Partition = ', p,'\n')

}
MSEP_df
```

With the Fourier basis and first derivative penalization, the average MSE (SD) across 100 random partitions were 0.7744(0.3412), 0.6338(0.2871), 0.8585 (0.2929), and 0.8092(0.2848) for the functional regression (FR), penalized functional regression (PFR), penalized Bayesian functional regression (PBFR), and Bayesian functional regression (BFR, $\boldsymbol{\beta} \sim N\left(\mathbf{0}, \sigma_{\beta}^{2}\boldsymbol{I}_{L_1}\right)$), respectively. In 75, 81, and 77 out of the 100 random partitions, the MSE of the PFR was better than the partitions obtained with the FR, PBFR, and BFR, respectively. And only in 5 out of 100 cases, the MSE of the PBFR was better than the BFR, making the penalty term in the Bayesian estimation non-important and indeed harmful (see **Appendix 3** for the R code used).

With the B-spline basis and first derivative penalization, the PFR (0.5718 (0.2669)) also resulted better on average than FR (0.6012(0.2681)), PBFR (0.8475 (0.3333)), and BFR (0.7982(0.3112)). Here, in 62 out of 100 random partitions, the MSE of the PFR was less than the MSE of the FR, while in 84 and 80 out of the 100 random partitions, they were better than the PBFR and BFR, respectively. Also, taking into account the penalty term in the Bayesian prediction was not so important because in only 8 out of the 100 random partitions, the MSE of the PBFR was less than the MSE corresponding to the BFR (see **Appendix 3** for the R code used).

When using the penalty matrix based on second derivatives, in each case (Fourier and B-spline), the results were similar.

The Bayesian formulation can be extended easily to take into account the effects of other factors. For example, in Example 14.5, the effects of the environment can be added as

$$\boldsymbol{y} = \mathbf{1}_n\mu + \boldsymbol{X}_E\boldsymbol{\beta}_E + \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}, \tag{14.13}$$

where $\boldsymbol{X}_E$ is the design matrix of the environments and $\boldsymbol{\beta}_E$ is the vector with the environment effects, and the Bayesian formulation can be completed by assuming a prior distribution for $\boldsymbol{\beta}_E$. As was described in Chap. 6 in the BGLR package, there are several options for this: FIXED, BRR, BayesA, BayesB, BayesC, and BL. In the next example, the first one is used.

**Example 14.6**

This is a continuation of Example 14.5 used to illustrate the performance when adding environmental information to the prediction task by using the Bayesian formulation (14.13). The resulting models were evaluated with 100 random partitions with both Fourier and B-spline basis and first derivative penalization.

The key code for implementing this example is given below.

```
For Fourier basis:
#Matrix design of environment
X_E = model.matrix(~0+Env,data=dat_F)[,-1]
#Linear predictor to PBFR + Env effect
ETA = list(list(X=X_E,model='FIXED'),list(X=X_Fa1,model='FIXED'),
list(X=X_Fa2,model='BRR'))
```

```
# Linear predictor to BFR + Env effect
ETA_NP = list(list(X=X_E,model='FIXED'),list(X=X_F[,1],
                      model='FIXED'),list(X=X_F[,-1],model='BRR'))
```

**For B-spline basis:**

```
#Matrix design of environment
X_E = model.matrix(~0+Env,data=dat_F)[,-1]
#Linear predictor to PBFR + Env effect
ETA = list(list(X=X_E,model='FIXED'),list(X=X_Fa1,model='FIXED'),
     list(X=X_Fa2,model='BRR'))
#Linear predictor to BFR + Env effect
ETA_NP = list(list(X=X_E,model='FIXED'),list(X=X_BS,model='BRR'))

#Random cross-validation
RP = 100
set.seed(1)
MSEP_df = data.frame(RP=1:RP,MSEP_PFR=NA,MSEP_FR=NA,lamb_o=NA)
for(p in 1:RP)
{
 Pos_tst = sample(n,0.20*n)
 X_F_tr = X_F[-Pos_tst,]; n_tr = dim(X_F_tr)[1]
 y_tr  = y[-Pos_tst]; y_tst = y[Pos_tst]

 #FR
 dat_df = data.frame(y=y,X=X_F)
 A_F = lm(y~.,data=dat_df[-Pos_tst,])
 yp_tst = predict(A_F,newdata = dat_df[Pos_tst,])
 MSEP_df$MSEP_FR[p] = mean((y_tst-yp_tst)^2)

 #PFR with first derivative
 A_PFR = cv.glmnet(x =X_Fa[-Pos_tst,],y = y[-Pos_tst] ,alpha = 0,
         nfolds=5,lambda=lambda/n_tr,
         penalty.factor=dv,
         standardize=FALSE,maxit=1e6)
 MSEP_df$lamb_o[p] = A_PFR$lambda.min
 yp_tst = predict(A_PFR,newx=X_Fa[Pos_tst,], s="lambda.min")[,1]
 MSEP_df$MSEP_PFR[p] = mean((y_tst-yp_tst)^2)

 #BGLR
 y_NA = y
 y_NA[Pos_tst] = NA
 A_BGLR = BGLR(y_NA,ETA= ETA,nIter=1e4, burnIn = 1e3,verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR[p] = mean((y_tst-yp_tst)^2)

 A_BGLR = BGLR(y_NA,ETA= ETA_NP,nIter=1e4, burnIn = 1e3,
verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR_NP[p] = mean((y_tst-yp_tst)^2)
  cat('Partition = ', p,'\n')
 }
```

**Table 14.2** Mean square error (MSE) of prediction for 100 random partitions for Fourier and B-spline representations, where the PFR and FR are classic functional regression models used in Example 14.4, and PBFR and BFR are model (14.13) with prior $\boldsymbol{\beta} \sim N\left(\mathbf{0}, \sigma_\beta^2 \boldsymbol{P}^{-1}\right)$ (with penalization matrix based on the first derivative) and $\boldsymbol{\beta} \sim N\left(\mathbf{0}, \sigma_\beta^2 \boldsymbol{I}_{L_1}\right)$ (without penalization), respectively, for the functional term

| | | PFR | FR | PBFR | BFR |
|---|---|---|---|---|---|
| Fourier | **Mean** | 0.6029 | 0.7797 | 0.4580 | 0.4382 |
| | **SD** | 0.2988 | 0.3593 | 0.2058 | 0.1986 |
| B-spline | **Mean** | 0.6138 | 0.6133 | 0.4304 | 0.4176 |
| | **SD** | 0.2695 | 0.2568 | 0.1729 | 0.1668 |

The results are shown in Table 14.2, where the third and fourth rows correspond to the average (Mean) and standard deviation (SD) of the MSE, when using Fourier basis ($L_2$=29 for covariate representation and $L_1 = 21$ for the beta function $\beta(\cdot)$) in the four fitted models (PFR, FR, PBFR, and BFR, with the environment effects added in the predictor of the model). With respect to the classical functional regression models (PFR and FR), both Bayesian models with environment (PBFR and BFR) effects resulted in a better performance, but again, the Bayesian model without penalization matrix was better (0.4382 vs. 0.4580) (see **Appendix 4** for the whole code).

For the B-spline basis, similar results were obtained, but again, the difference between the PFR and FR was not so important as observed before (see **Appendix 4**). So, in both cases (Fourier and B-spline), adding the environment information to the model improved the prediction performance. In general, the extra information can be added and explored easily with the BGLR package to determine the importance of this in the prediction task of interest.

Further information can be easily added to the model without many complications. See Chap. 6 for more detailed information on how to do this with the BGLR R package.

**Example 14.7**

This example is an extension of Example 14.6 by adding the interaction of the environment with the hyper-spectral data (the functional covariable) in the predictor of the model. The corresponding term is given by

$$\int_0^T x(t)\beta_e(t)dt,$$

where $\beta_e(t)$ is the coefficient function corresponding to the functional part that represents the interaction between the $e$th environment and reflectance for wavelength $t$ (in general, the functional covariate measured in time $t$), to allow the effect of reflectance to vary by environment (see Montesinos-López et al. 2017b). By assuming that there are $n_e$ observations in environment $e$, $e = 1, \ldots, I$, the corresponding

re-expressed model after representing the coefficient function $\beta_e(t)$ in terms of the same basis functions used for $\beta(t)$, $\beta_e(t) = \sum_{l=1}^{L_{1e}} \beta_{el}\phi_l(t)$, is given by

$$\boldsymbol{y} = \mathbf{1}_n\mu + \boldsymbol{X}_E\boldsymbol{\beta}_E + \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{X}_{EF}\boldsymbol{\beta}_{EF} + \boldsymbol{\epsilon}, \tag{14.14}$$

where $\mathbf{1}_n$ is a vector of dimension $n \times 1$ with all its entries equal to 1, $\boldsymbol{X} = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]^{\mathrm{T}}$ and $\boldsymbol{x}_i = [x_{i1}, \ldots, x_{iL_1}]^{\mathrm{T}}, i = 1, \ldots, n = n_1 + \cdots + n_I$, as defined in (14.4) and (14.5), $\boldsymbol{X}_E$ is the design matrix of the environments and $\boldsymbol{\beta}_E$ is the vector with the environment effects (see 14.13), $\boldsymbol{X}_{EF}$ is the design matrix of the interaction effecs of enviroment-reflectance and $\boldsymbol{\beta}_{EF}$ is the corresponding interaction effects of environment-reflectance, as given by

$$\boldsymbol{X}_{EF} = \begin{bmatrix} \boldsymbol{x}_1^{\mathrm{T}} & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots \\ \boldsymbol{x}_{n_1}^{\mathrm{T}} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{x}_{n_1+1}^{\mathrm{T}} & \vdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \boldsymbol{x}_{n_1+n_2}^{\mathrm{T}} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{x}_{n-n_I+1}^{\mathrm{T}} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{x}_n^{\mathrm{T}} \end{bmatrix} \text{ and } \boldsymbol{\beta}_{EF} = \begin{bmatrix} \boldsymbol{\beta}_1 \\ \boldsymbol{\beta}_2 \\ \vdots \\ \boldsymbol{\beta}_I \end{bmatrix}.$$

This model was also evaluated with 100 random partitions with both Fourier and B-spline basis and with (PBFR (14.14)) and without (BFR (14.14)) first derivative penalization. The results are shown in Table 14.3, together with the prediction performance of the model with no interaction term (model 14.13). We can observe that by adding the interaction of environment with the functional covariate, both Bayesian models (PBFR and BFR) resulted in a reduction on average of about 35% of the MSE (PBFR (14.13) vs. PBFR (14.14) and BFR (14.13) vs. BFR (14.14)), and again the Bayesian model without penalization matrix was better (0.2955 vs. 0.2899)

**Table 14.3** Mean squared error of prediction (MSE) for 100 random partitions for Fourier and B-spline representations, where PBFR (14.13) and BFR (14.13) are MSE of the model (14.13) with and without penalization matrix based on the first derivative in the functional term ($\boldsymbol{X\beta}$), and PBFR (14.14) and BFR (14.14) are for model (14.14) with and without penalization matrix based on the first derivative in the functional terms ($\boldsymbol{X\beta}, \boldsymbol{X}_{EF}\boldsymbol{\beta}_{EF}$)

| | | PBFR (14.13) | BFR (14.13) | PBFR (14.14) | BFR (14.14) |
|---|---|---|---|---|---|
| Fourier | **Mean** | 0.4563 | 0.4387 | 0.2955 | 0.2899 |
| | **SD** | 0.2056 | 0.1992 | 0.1363 | 0.1313 |
| B-spline | **Mean** | 0.4510 | 0.4361 | 0.2814 | 0.2818 |
| | **SD** | 0.1892 | 0.1837 | 0.1149 | 0.1224 |

in the Fourier basis, while in the B-spline basis the Bayesian model with penalization matrix was better (0.2814 vs. 0.2818).

Finally, in this chapter, we gave the basic theory of functional regression and we provided examples to illustrate this methodology for genomic prediction using the *glmnet* and BGLR packages. The examples show in detail how to implement functional regression analysis in a more straightforward way by taking advantage of the existing software for genomic selection. Also, the examples are done with small data sets so that the user can run them on his/her own computer and can understand the implementation process well.

## Appendix 1

```
rm(list=ls(all=TRUE))
#Example 14.3
load('dat_ls.RData')
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
#Wavelengths data
dat_W = dat_ls$dat_WL
colnames(dat_W)[1:8]
head(dat_W)[,1:8]

#Wavelengths used
Wv = as.numeric(substring(colnames(dat_W)[-(1:2)],2))
#Reflectance in each individual
X_W = unique(dat_W[,-(1:2)])
X_W =  scale(X_W,scale=FALSE)
#Reflectance behavior as a function of wavelength for 10 individuals
W = matrix(Wv,nr=length(Wv),nc=60,byrow = FALSE)
matplot(W,t(X_W),xlab='Wavelengths',ylab='Reflectance',
    col=dat_W$Env,pch=1,cex=0.5)

#Optimal number of Fourier basis functions to represent the functional
covariable
#of each individual
library(fda)
Perd = diff(range(Wv))
plot(Wv,X_W[1,])
m =  length(Wv)
n =  dim(dat_F)[1]
nbF = seq(3,m/2,2)
BIC_mat =  matrix(NA,nr=length(nbF),nc = n)
for(l in 1:length(nbF))
{
 #Fourier basis for x(t)
 Psi_F = create.fourier.basis(range(Wv)+c(0,0), nbasis = nbF[l],
```

```
              period = Perd)
 SF_mat = smooth.basisPar(argvals = Wv,
             y =t(X_W), fdobj = Psi_F,lambda=0,
             Lfdobj = 0)
 #plot(SF_mat,col=dat_F$Env,xlab='t',ylab='x(t)')
 Res_mat = t(residuals(SF_mat))
 sigmav = sqrt(rowMeans(Res_mat^2))
 ll_f<-function(Res)
 {
  sigma = sqrt(mean(Res^2))
  sum(dnorm(Res,0,sigma,log=TRUE))
 }
 llv = apply(Res_mat,1,ll_f)
 BIC_v = -2*llv+log(m)*(nbF[l]+1)
 BIC_mat[l,] = BIC_v
 cat('l=',l,'\n')
}
#Optimal nbF in each curve obtained with the BIC
nbFov = apply(BIC_mat,2,function(x)nbF[which.min(x)])
plot(nbFov,xlab='Individual',ylab='Optimal number of basis functions
choosen by BIC')

#The median value of the more often selected number of basis functions
across all curves
Tb  = table(nbFov)
nbFo = as.numeric(names(Tb))[which(Tb==max(Tb))]
nbFo
nbFo = median(nbFo)
nbFo

#Optimal number of B-spline basis functions to represent the functional
covariable
#of each individual
nbBS = seq(4,m/2,1)
BIC_mat = matrix(NA,nr=length(nbBS),nc = n)
for(l in 1:length(nbBS))
{
 #Fourier basis for x(t)
 Psi_BS = create.bspline.basis(range(Wv)+c(0,0), nbasis = nbBS[l],
norder = 4)
 SBS_mat = smooth.basisPar(argvals = Wv,
             y =t(X_W), fdobj = Psi_BS,lambda=0,
             Lfdobj = 0)
 #plot(SBS_mat,col=dat_F$Env,xlab='t',ylab='x(t)')
 Res_mat = t(residuals(SBS_mat))
 sigmav = sqrt(rowMeans(Res_mat^2))
 ll_f<-function(Res)
 {
  sigma = sqrt(mean(Res^2))
  sum(dnorm(Res,0,sigma,log=TRUE))
 }
 llv = apply(Res_mat,1,ll_f)
 #BIC_v = -2*llv+log(m)*(nbBS[l]-4+nbBS[l]+1)
```

```
 BIC_v = -2*llv+log(m)*(nbBS[l]+1)
 BIC_mat[l,] = BIC_v

 cat('l=',l,'\n')
}
#Optimal nbBS for each curve obtained with the BIC
nbBSov = apply(BIC_mat,2,function(x)nbBS[which.min(x)])
plot(nbBSov,xlab='Individual',ylab='Optimal number of basis
functions choosen by BIC')

Tb = table(nbBSov)
barplot(Tb)
nbBSov = as.numeric(names(Tb))[which(Tb==max(Tb))]
nbBSo =  median(nbBSov)
nbBSo

#Fourier and B-spline representations with the "optimal" number
#of basis functions obtained across the curves using the BIC
par(mfrow=c(1,2))
#Fourier representation of all curves using 29 Fourier basis functions
Psi_F = create.fourier.basis(range(Wv), nbasis = nbFo,
             period = Perd)
SF_mat = smooth.basisPar(argvals = Wv,
            y =t(X_W), fdobj = Psi_F,lambda=0,
            Lfdobj = 0)
matplot(Wv,fitted(SF_mat),col=dat_F$Env,xlab='t (Wavelengths)',
  ylab='x(t) (Reflectance)',pch=1,type='l',
  main=paste('Fourier representation with\n ',nbFo,'basis
functions',sep=' '))

#B-spline representation of all curves using 16 B-spline basis
functions
Psi_BS = create.bspline.basis(range(Wv), nbasis = nbBSo)
SBS_mat = smooth.basisPar(argvals = Wv,
           y =t(X_W), fdobj = Psi_BS,lambda=0,
           Lfdobj = 0)
matplot(Wv,fitted(SBS_mat),col=dat_F$Env,xlab='t (Wavelengths)',
  ylab='x(t) (Reflectance)',pch=1,type='l',
  main=paste('B-spline representation with\n ',nbBSo,
           'basis functions'',sep=' '))

#Functional regression with Fourier and B-splines
#Computing X for Fourier representation with L1 =  21 Fourier basis
#functions for the beta function
L1 = 21
Phi_F = create.fourier.basis(range(Wv), nbasis = L1,
              period = Perd)
Q   = inprod(Phi_F,Psi_F)
c_mat = t(SF_mat$fd$coefs)
X_F  = c_mat%*%t(Q)
dim(X_F)
```

```
X_F_df = data.frame(dat_W[,1:2],X_F)
#write.csv(X_F_df,file='X_F_df.csv')

#Computing X for B-spline representation with L1 = 21 B-spline basis
functions
#for the beta function
L1 = 21
Phi_BS = create.bspline.basis(range(Wv), nbasis = L1,
              norder= 4)
Q   = inprod(Phi_BS,Psi_BS)
c_mat = t(SBS_mat$fd$coefs)
X_BS  = c_mat%*%t(Q)
dim(X_BS)
X_BS_df = data.frame(dat_W[,1:2],X_BS)
#write.csv(X_BS_df,file='X_BS_df.csv')

#OLS estimation using Fourier representation with L1 = 21 Fourier basis
#functions for the beta function
par(mfrow=c(1,2))
yv = dat_F$y
A_F = lm(yv~X_F)
betav = coef(A_F)
#Plot of the 21 estimated Bj's
#par(mar=c(5,5.1,2,2))
#plot(betav[-1],xlab=expression(j),ylab=expression(hat(beta)[j]))
#betaf_F = eval.basis(Wv,Phi_F)%*%betav[-1]
#plot(Wv,betaf_F)
#Fitted values
yp = fitted(A_F)
plot(yv,yp,xlab='y (Observed)',ylab='yp (Predicted)',
  main='Fourier basis functions \n 29 for x(t) and 21 for B(t)')
abline(a=0,b=1)
mean((yv-yp)^2)

#OLS estimation using Fourier representation with L1 = 21 B-spline basis
#functions for the beta function
yv = dat_F$y
A_BS = lm(yv~X_BS)
betav = coef(A_BS)
#Plot of the 21 estimated Bj's
#par(mar=c(5,5.1,2,2))
#plot(betav[-1],xlab=expression(j),ylab=expression(hat(beta)[j]))
#betaf_BS = eval.basis(Wv,Phi_BS)%*%betav[-1]
#plot(Wv,betaf_F)
#Fitted values
yp = fitted(A_BS)
plot(yv,yp,xlab='y (Observed)',ylab='yp (Predicted)',
  main='B-spline basis functions \n 16 for x(t) and 21 for B(t)')
abline(a=0,b=1)
mean((yv-yp)^2)
#BIC to choose the optimal number of basis functions for the beta
#function (L1) in the Fourier representation of this
nbF_B = seq(3,29,2)
```

```
Tab = data.frame()
for(i in 1:length(nbF_B))
{
 Phi_F = create.fourier.basis(range(Wv), nbasis = nbF_B[i],
              period = Perd)
 Q = inprod(Phi_F,Psi_F)
 c_mat = t(SF_mat$fd$coefs)
 X_F = c_mat%*%t(Q)
 A = lm(yv~X_F)
 BIC = BIC(A)
 Tab = rbind(Tab,data.frame(nb = nbF_B[i],BIC=BIC))
}
plot(Tab$nb,Tab$BIC,xlab='Número de bases',ylab='BIC')
nbFo_B = Tab$nb[which.min(Tab$BIC)]
nbFo_B

#Observed and fitted values with "optimal" number of basis functions for
#the beta function (L1) with the Fourier representation
L1 = nbFo_B
Phi_F = create.fourier.basis(range(Wv), nbasis = L1,
              period = Perd)
Q   = inprod(Phi_F,Psi_F)
c_mat = t(SF_mat$fd$coefs)
X_F  = c_mat%*%t(Q)
dim(X_F)
A_F = lm(yv~X_F)
betav = coef(A_F)
betaf_F = eval.basis(Wv,Phi_F)%*%betav[-1]
plot(Wv,betaf_F)

#Fitted values
yp = fitted(A_F)
plot(yv,yp,xlab='y (Observed)',ylab='yp (Predicted)')
abline(a=0,b=1)
mean((yv-yp)^2)

#Q-Q Normal
ev = residuals(A_F)
sigma2 = mean(ev^2)
qqnorm(ev/sqrt(sigma2))
abline(a=0,b=1)

#BIC to choose the optimal number of basis functions for the beta
#function (L1) in the B-spline representation of this
nbBS_B = seq(5,45,1)
Tab = data.frame()
for(i in 1:length(nbBS_B))
{
 Phi_BS = create.bspline.basis(range(Wv), nbasis = nbBS_B[i],
              norder = 4)
 Q = inprod(Phi_BS,Psi_BS)
 c_mat = t(SBS_mat$fd$coefs)
 X_BS = c_mat%*%t(Q)
```

```
 A = lm(yv~X_BS)
 BIC = BIC(A)
 Tab = rbind(Tab,data.frame(nb = nbBS_B[i],BIC=BIC))
}
plot(Tab$nb,Tab$BIC,xlab='Number of basis functions',ylab='BIC')
nbBSo_B = Tab$nb[which.min(Tab$BIC)]
nbBSo_B

#Observed and fitted values with "optimal" number of basis functions for
#the beta function (L1) with the B-spline representation
L1 = nbBSo_B
Phi_BS = create.bspline.basis(range(Wv), nbasis = L1, norder = 4)
Q   = inprod(Phi_BS,Psi_BS)
c_mat = t(SBS_mat$fd$coefs)
X_BS  = c_mat%*%t(Q)
dim(X_BS)
A_BS = lm(yv~X_BS)
betav = coef(A_BS)
betaf_BS = eval.basis(Wv,Phi_BS)%*%betav[-1]
plot(Wv,betaf_BS)
#lines(Wv,betaf_F,col=2)

par(mfrow=c(2,2))
#Fourier
#Fitted values
yp = fitted(A_F)
plot(yv,yp,xlab='y (Observed)',ylab='yp (Predicted)')
abline(a=0,b=1)
mean((yv-yp)^2)

#Q-Q Normal
ev = residuals(A_F)
sigma2 = mean(ev^2)
qqnorm(ev/sqrt(sigma2))
abline(a=0,b=1)
#B-spline
#Fitted values
yp = fitted(A_BS)
#par(mfrow=c(1,2),oma = c(0, 0, 2, 0))
plot(yv,yp,xlab='y (Observed)',ylab='yp (Predicted)')#,main='B-
spline representation')
abline(a=0,b=1)
mean((yv-yp)^2)
#Q-Q Normal
ev = residuals(A_BS)
sigma2 = mean(ev^2)
qqnorm(ev/sqrt(sigma2))
abline(a=0,b=1)
#mtext("Title for Two Plots", outer = T, cex = 1.5)
mean((yv-yp)^2)

#BIC of the models corresponding to the optimal Fourier and B-spline
#representations
```

```
BIC(A_F)
BIC(A_BS)

#Random partition to measure the prediction performance
#of Fourier and B-spline representations
set.seed(1)
MSEP = data.frame()
for(p in 1:10)
{
 Pos_j = sample(n,.20*n)
 dat_F_j = dat_F[-Pos_j,]
 X_W_j  = X_W[-Pos_j,]
 #Fourier
 A_F = lm(yv[-Pos_j]~X_F[-Pos_j,])
 #Fitted values
 #yp = predict(A_F,newdata = data.frame(X_F[Pos_j,]))
 yp_F = cbind(1,X_F[Pos_j,])%*%coef(A_F)
 #B-spline
 A_BS = lm(yv[-Pos_j]~X_BS[-Pos_j,])
 yp_BS = cbind(1,X_BS[Pos_j,])%*%coef(A_BS)

 MSEP =  rbind(MSEP,data.frame(MSEP_Fourier = mean((yv[Pos_j]-yp_F)
^2),
               MSEP_BS = mean((yv[Pos_j]-yp_BS)^2)))
}
MSEP
```

## Appendix 2 (Example 14.4)

```
rm(list=ls(all=TRUE))
library(fda)
#Example 14.4 (data set of Example 14.3)
load('dat_ls.RData')
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
yv =  dat_F$y
n =  length(yv)

#Wavelengths data
dat_W = dat_ls$dat_WL
colnames(dat_W)[1:8]
head(dat_W)[,1:8]

#Wavelengths used
Wv = as.numeric(substring(colnames(dat_W)[-(1:2)],2))
#Reflectance in each individual
X_W = unique(dat_W[,-(1:2)])
```

```
X_W = scale(X_W,scale=FALSE)
#Reflectance behavior as a function of wavelength for 10 individuals
W = matrix(Wv,nr=length(Wv),nc=60,byrow = FALSE)
matplot(W,t(X_W),xlab='Wavelengths',ylab='Reflectance',
    col=dat_W$Env,pch=1,cex=0.5)

#Number of Fourier and B-spline basis functions to represent the
covariable
#function
nbFo  = 29
nbBSo = 16

#Matrix design X computed with Fourier and B-spline bases
#Computing X for Fourier representation with L1 = 21 Fourier basis
functions
#for the beta function
L1 = 21
P = diff(range(Wv))
Psi_F = create.fourier.basis(range(Wv), nbasis = nbFo,
               period = P)
SF_mat = smooth.basisPar(argvals = Wv,
             y =t(X_W), fdobj = Psi_F,lambda=0,
             Lfdobj = 0)
Phi_F = create.fourier.basis(range(Wv), nbasis = L1,
               period = P)
Q   = inprod(Phi_F,Psi_F)
c_mat = t(SF_mat$fd$coefs)
X_F  = c_mat%*%t(Q)
dim(X_F)

#Computing X for B-spline representation with L1 =  21 B-spline basis
functions
#for the beta function
L1 = 21
Psi_BS = create.bspline.basis(range(Wv), nbasis = nbBSo)
SBS_mat = smooth.basisPar(argvals = Wv,
             y =t(X_W), fdobj = Psi_BS,lambda=0,
             Lfdobj = 0)
Phi_BS = create.bspline.basis(range(Wv), nbasis = L1,
               norder= 4)
Q   = inprod(Phi_BS,Psi_BS)
c_mat = t(SBS_mat$fd$coefs)
X_BS  = c_mat%*%t(Q)
dim(X_BS)

#Smoothing estimation using Fourier representation for the functional
#covariable and L1 = 21 Fourier basis functions for the beta function
par(mfrow=c(1,2))
library(glmnet)
#Penalization with first derivate (Changing the value Lfdobj=1 to
Lfdobj=2, the penalty matrix #with second derivates is obtained)
P_mat_F = eval.penalty(Phi_F,Lfdobj=1)
ei_P = eigen(P_mat_F)
```

```
gamma = ei_P$vectors
dv = ei_P$values
dv = ifelse(dv<1e-10,0,dv)

X_Fa = X_F%*%gamma

#Grid of lambda values obtained by varying the proportion of variance
explained
#by the functional predictor
lamb_FR_f<-function(Xa,dv,K=100,li=1e-1,ls=1-1e-12)
{
 Pos = which(dv<1e-10)
 D_inv = diag(1/dv[-Pos])
 PEV = seq(li,ls,length=K)
 Xa  = Xa[,-Pos]
 lambv = (1-PEV)/PEV*mean(diag(Xa%*%D_inv%*%t(Xa)))
 lambv = exp(seq(min(log(lambv)),max(log(lambv)),length=K))
 sort(lambv,decreasing = TRUE)
}

lambda = lamb_FR_f(X_Fa,dv,K=1e2)

#Random cross-validation
y=yv
RP = 100
set.seed(1)
MSEP_df = data.frame(RP=1:RP,MSEP_PFR=NA,MSEP_FR=NA,lamb_o=NA)
for(p in 1:RP)
{
 Pos_tst = sample(n,0.20*n)
 X_F_tr = X_F[-Pos_tst,]; n_tr = dim(X_F_tr)[1]
 y_tr  = y[-Pos_tst]; y_tst = y[Pos_tst]

 #FR
 dat_df = data.frame(y=y,X=X_F)
 A_F = lm(y~.,data=dat_df[-Pos_tst,])
 yp_tst = predict(A_F,newdata = dat_df[Pos_tst,])
 MSEP_df$MSEP_FR[p] = mean((y_tst-yp_tst)^2)

 #PFR with first derivative
 A_PFR = cv.glmnet(x =X_Fa[-Pos_tst,],y = y[-Pos_tst] ,alpha = 0,
          nfolds=5,lambda=lambda/n_tr,penalty.factor=dv,
          standardize=FALSE,maxit=1e6)
 #plot(A_PFR)
 MSEP_df$lamb_o[p] = A_PFR$lambda.min
 yp_tst = predict(A_PFR,newx=X_Fa[Pos_tst,], s="lambda.min")[,1]
 MSEP_df$MSEP_PFR[p] = mean((y_tst-yp_tst)^2)
 cat('Partition = ', p,'\n')

}
MSEP_df
```

```
#Mean and SD of MSEP across 100 RP
apply(MSEP_df[,-1],2,function(x)c(mean(x),sd(x)))
mean(MSEP_df$MSEP_PFR<MSEP_df$MSEP_FR)

plot(MSEP_df$MSEP_PFR,MSEP_df$MSEP_FR,
   col = ifelse(MSEP_df$MSEP_PFR<MSEP_df$MSEP_FR,
          3,1),xlab='MSEP PFR',ylab='MSEP FR',
   main='Fourier')
abline(a=0, b=1)

#Smoothing estimation using B-spline for the functional
#covariable and L1 = 21 B-spline basis functions for the beta function
library(glmnet)
#Penalization with first derivate (Changing the value Lfdobj=1 to
Lfdobj=2, the penalty #matrix with second derivates is obtained)
P_mat_F = eval.penalty(Phi_BS,Lfdobj=1)
ei_P = eigen(P_mat_F)
gamma = ei_P$vectors
dv = ei_P$values
dv = ifelse(dv<1e-10,0,dv)
X_Fa = X_BS%*%gamma

#Grid of lambda values obtained by varying the proportion of variance
explained
#by the functional predictor
lamb_FR_f<-function(Xa,dv,K=100,li=1e-1,ls=1-1e-12)
{
 Pos = which(dv<1e-10)
 D_inv = diag(1/dv[-Pos])
 PEV = seq(li,ls,length=K)
 Xa  = Xa[,-Pos]
 lambv = (1-PEV)/PEV*mean(diag(Xa%*%D_inv%*%t(Xa)))
 lambv = exp(seq(min(log(lambv)),max(log(lambv)),length=K))
 sort(lambv,decreasing = TRUE)
}

lambda = lamb_FR_f(X_Fa,lambv,K=1e2)

#Random cross-validation
y=yv
RP = 100
set.seed(1)
MSEP_df = data.frame(RP=1:RP,MSEP_PFR=NA,MSEP_FR=NA,lamb_o=NA)
for(p in 1:RP)
{
 Pos_tst = sample(n,0.20*n)
 X_F_tr = X_BS[-Pos_tst,]; n_tr = dim(X_F_tr)[1]
 y_tr  = y[-Pos_tst]; y_tst = y[Pos_tst]

 #FR
 dat_df = data.frame(y=y,X=X_BS)
 A_F = lm(y~.,data=dat_df[-Pos_tst,])
```

```
 yp_tst  = predict(A_F,newdata = dat_df[Pos_tst,])
 MSEP_df$MSEP_FR[p] = mean((y_tst-yp_tst)^2)

 #PFR with first derivative
 A_PFR = cv.glmnet(x =X_Fa[-Pos_tst,],y = y[-Pos_tst] ,alpha = 0,
          nfolds=5,lambda=lambda/n_tr,penalty.factor=dv,
          standardize=FALSE,maxit=1e6)
 #plot(A_PFR)
 MSEP_df$lamb_o[p] = A_PFR$lambda.min
 yp_tst = predict(A_PFR,newx=X_Fa[Pos_tst,], s="lambda.min")[,1]
 MSEP_df$MSEP_PFR[p] = mean((y_tst-yp_tst)^2)
 cat('Partition = ', p,'\n')

}
MSEP_df

#Mean and SD of MSEP across 100 RP
apply(MSEP_df[,-1],2,function(x)c(mean(x),sd(x)))
mean(MSEP_df$MSEP_PFR<MSEP_df$MSEP_FR)

plot(MSEP_df$MSEP_PFR,MSEP_df$MSEP_FR,
   col = ifelse(MSEP_df$MSEP_PFR<MSEP_df$MSEP_FR,
         3,1),xlab='MSEP PFR',ylab='MSEP FR',
   main='B-spline')
abline(a=0,b=1)
```

## Appendix 3 (Example 14.5)

```
rm(list=ls(all=TRUE))
library(fda)
#Example 14.5
load('dat_ls.RData')
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
yv =  dat_F$y
n =  length(yv)

#Wavelengths data
dat_W = dat_ls$dat_WL
colnames(dat_W)[1:8]
head(dat_W)[,1:8]

#Wavelengths used
Wv = as.numeric(substring(colnames(dat_W)[-(1:2)],2))
#Reflectance in each individual
X_W = unique(dat_W[,-(1:2)])
X_W =  scale(X_W,scale=FALSE)
```

```
#Number of Fourier and B-spline basis functions to represent the
covariable
#function
nbFo  = 29
nbBSo = 16

#Functional regression with Fourier and B-splines
#Computing X for Fourier representation with L1 =  21 Fourier basis
functions
#for the beta function
L1 = 21
P = diff(range(Wv))
Psi_F = create.fourier.basis(range(Wv), nbasis = nbFo,
              period = P)
SF_mat = smooth.basisPar(argvals = Wv,
            y =t(X_W), fdobj = Psi_F,lambda=0,
            Lfdobj = 0)
Phi_F = create.fourier.basis(range(Wv), nbasis = L1,
              period = P)
Q   = inprod(Phi_F,Psi_F)
c_mat = t(SF_mat$fd$coefs)
X_F  = c_mat%*%t(Q)
dim(X_F)

#Computing X for B-spline representation with L1 = 21 B-spline basis
functions
#for the beta function
L1 = 21
Psi_BS = create.bspline.basis(range(Wv), nbasis = nbBSo)
SBS_mat = smooth.basisPar(argvals = Wv,
            y =t(X_W), fdobj = Psi_BS,lambda=0,
            Lfdobj = 0)
Phi_BS = create.bspline.basis(range(Wv), nbasis = L1,
              norder= 4)
Q   = inprod(Phi_BS,Psi_BS)
c_mat = t(SBS_mat$fd$coefs)
X_BS  = c_mat%*%t(Q)
dim(X_BS)

#Smoothing estimation using Fourier representation for the functional
#covariable and L1 = 21 Fourier basis functions for the beta function
par(mfrow=c(1,2))
library(glmnet)
#Penalization with first derivate
P_mat_F = eval.penalty(Phi_F,Lfdobj=1)
ei_P = eigen(P_mat_F)
gamma = ei_P$vectors
dv = ei_P$values
dv = ifelse(dv<1e-10,0,dv)

X_Fa = X_F%*%gamma
```

```
#Grid of lambda values obtained by varying the proportion of variance
explained
#by the functional predictor
lamb_FR_f<-function(Xa,dv,K=100,li=1e-1,ls=1-1e-12)
{
 Pos = which(dv<1e-10)
 D_inv = diag(1/dv[-Pos])
 PEV = seq(li,ls,length=K)
 Xa  = Xa[,-Pos]
 lambv = (1-PEV)/PEV*mean(diag(Xa%*%D_inv%*%t(Xa)))
 lambv = exp(seq(min(log(lambv)),max(log(lambv)),length=K))
 sort(lambv,decreasing = TRUE)
}
lambda = lamb_FR_f(X_Fa,dv,K=1e2)

y=yv
library(BGLR)
#Linear predictor specification in BGLR for the PBFR model
Pos = which(dv>0)
#Matrix design for the non-penalized columns of X^a
X_Fa1 = X_Fa[,-Pos]
#Matrix design for the penalized columns of X^a
X_Fa2 = X_Fa[,Pos]%*%diag(1/sqrt(dv[Pos]))
ETA = list(list(X=X_Fa1,model='FIXED'),list(X=X_Fa2,model='BRR'))

#Linear predictor specification in BGLR for the BFR model
ETA_NP = list(list(X=X_F[,1],model='FIXED'),list(X=X_F[,-1],
model='BRR'))

#Random cross-validation
RP = 100
set.seed(1)
MSEP_df = data.frame(RP=1:RP,MSEP_PFR=NA,MSEP_FR=NA,lamb_o=NA)
for(p in 1:RP)
{
 Pos_tst = sample(n,0.20*n)
 X_F_tr  = X_F[-Pos_tst,]; n_tr = dim(X_F_tr)[1]
 y_tr   = y[-Pos_tst]; y_tst = y[Pos_tst]

 #FR
 dat_df = data.frame(y=y,X=X_F)
 A_F = lm(y~.,data=dat_df[-Pos_tst,])
 yp_tst = predict(A_F,newdata = dat_df[Pos_tst,])
 MSEP_df$MSEP_FR[p] = mean((y_tst-yp_tst)^2)

 #PFR with first derivative
 A_PFR = cv.glmnet(x =X_Fa[-Pos_tst,],y = y[-Pos_tst] ,alpha = 0,
          nfolds=5,lambda=lambda/n_tr,
          penalty.factor=dv,
          standardize=FALSE,maxit=1e6)
 MSEP_df$lamb_o[p] = A_PFR$lambda.min
```

```
 yp_tst = predict(A_PFR,newx=X_Fa[Pos_tst,], s="lambda.min")[,1]
 MSEP_df$MSEP_PFR[p] = mean((y_tst-yp_tst)^2)

 #BGLR
 y_NA = y
 y_NA[Pos_tst] = NA
 A_BGLR = BGLR(y_NA,ETA= ETA,nIter=1e4, burnIn = 1e3,verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR[p] = mean((y_tst-yp_tst)^2)

 A_BGLR = BGLR(y_NA,ETA= ETA_NP,nIter=1e4, burnIn = 1e3,
verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR_NP[p] = mean((y_tst-yp_tst)^2)

 cat('Partition = ', p,'\n')

}
MSEP_df

#Mean and SD of MSEP across 10 RP
Tab = apply(MSEP_df[,-1],2,function(x)c(mean(x),sd(x)))
Tab

#Smoothing estimation using B-spline representation for the functional
#covariable and L1 = 21 B-spline basis functions for the beta function
library(glmnet)
#Penalization with first derivate
P_mat_F = eval.penalty(Phi_BS,Lfdobj=1)
ei_P = eigen(P_mat_F)
gamma = ei_P$vectors
dv = ei_P$values
dv = ifelse(dv<1e-10,0,dv)
X_Fa = X_BS%*%gamma

#Grid of lambda values obtained by varying the proportion of variance
explained
#by the functional predictor
lamb_FR_f<-function(Xa,dv,K=100,li=1e-1,ls=1-1e-12)
{
 Pos = which(dv<1e-10)
 D_inv = diag(1/dv[-Pos])
 PEV = seq(li,ls,length=K)
 Xa = Xa[,-Pos]
 lambv = (1-PEV)/PEV*mean(diag(Xa%*%D_inv%*%t(Xa)))
 lambv = exp(seq(min(log(lambv)),max(log(lambv)),length=K))
 sort(lambv,decreasing = TRUE)
}

lambda = lamb_FR_f(X_Fa,dv,K=1e2)
```

```
#Random cross-validation
y=yv
library(BGLR)
Pos = which(dv>0)
X_Fa1 = X_Fa[,-Pos]
X_Fa2 = X_Fa[,Pos]%*%diag(1/sqrt(dv[Pos]))
ETA = list(list(X=X_Fa1,model='FIXED'),list(X=X_Fa2,model='BRR'))
ETA_NP = list(list(X=X_BS,model='BRR'))
RP = 100
set.seed(1)
MSEP_df = data.frame(RP=1:RP,MSEP_PFR=NA,MSEP_FR=NA,lamb_o=NA)
for(p in 1:100)
{
 Pos_tst = sample(n,0.20*n)
 X_F_tr = X_BS[-Pos_tst,]; n_tr = dim(X_F_tr)[1]
 y_tr  = y[-Pos_tst]; y_tst = y[Pos_tst]

 #FR
 dat_df = data.frame(y=y,X=X_BS)
 A_F = lm(y~.,data=dat_df[-Pos_tst,])
 yp_tst = predict(A_F,newdata = dat_df[Pos_tst,])
 MSEP_df$MSEP_FR[p] = mean((y_tst-yp_tst)^2)

 #PFR with first derivative
 A_PFR = cv.glmnet(x =X_Fa[-Pos_tst,],y = y[-Pos_tst] ,alpha = 0,
          nfolds=5,lambda=lambda/n_tr,penalty.factor=dv,
          standardize=FALSE,maxit=1e6)
 #plot(A_PFR)
 MSEP_df$lamb_o[p] = A_PFR$lambda.min
 yp_tst = predict(A_PFR,newx=X_Fa[Pos_tst,], s="lambda.min")[,1]
 MSEP_df$MSEP_PFR[p] = mean((y_tst-yp_tst)^2)

 #BGLR
 y_NA = y
 y_NA[Pos_tst] = NA
 A_BGLR = BGLR(y_NA,ETA= ETA,nIter=1e4, burnIn = 1e3,verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR[p] = mean((y_tst-yp_tst)^2)

 A_BGLR = BGLR(y_NA,ETA= ETA_NP,nIter=1e4, burnIn = 1e3,
verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR_NP[p] = mean((y_tst-yp_tst)^2)

 cat('Partition = ', p,'\n')

}
MSEP_df

#Mean and SD of MSEP across 100 RP
Tab = apply(MSEP_df[,-1],2,function(x)c(mean(x),sd(x)))
Tab
```

## Appendix 4 (Example 14.6)

It is the same as the R code used in Example 14.5 but now to the corresponding predictor need to be added the matrix design of environments:

```
rm(list=ls(all=TRUE))
library(fda)
#Example 14.6
load('dat_ls.RData')
#Phenotypic data
dat_F = dat_ls$dat_F
head(dat_F)
yv =  dat_F$y
n =  length(yv)

#Wavelengths data
dat_W = dat_ls$dat_WL
colnames(dat_W)[1:8]
head(dat_W)[,1:8]

#Wavelengths used
Wv = as.numeric(substring(colnames(dat_W)[-(1:2)],2))
#Reflectance in each individual
X_W = unique(dat_W[,-(1:2)])
X_W =  scale(X_W,scale=FALSE)
#Reflectance behavior as a function of wavelength for 10 individuals
W = matrix(Wv,nr=length(Wv),nc=60,byrow = FALSE)
matplot(W,t(X_W),xlab='Wavelengths',ylab='Reflectance',
    col=dat_W$Env,pch=1,cex=0.5)

#Number of Fourier and B-spline basis functions to represent the
covariable
#function
nbFo  = 29
nbBSo = 16

#----
#Functional regression with Fourier and B-splines
#---
#Computing X for Fourier representation with L1 =  21 Fourier basis
functions
#for the beta function
L1 = 21
P = diff(range(Wv))
Psi_F = create.fourier.basis(range(Wv), nbasis = nbFo,
              period = P)
SF_mat = smooth.basisPar(argvals = Wv,
            y =t(X_W), fdobj = Psi_F,lambda=0,
            Lfdobj = 0)
Phi_F = create.fourier.basis(range(Wv), nbasis = L1,
              period = P)
```

```
Q    = inprod(Phi_F,Psi_F)
c_mat = t(SF_mat$fd$coefs)
X_F   = c_mat%*%t(Q)
dim(X_F)

#Computing X for B-spline representation with L1 = 21 B-spline basis
functions
#for the beta function
L1 = 21
Psi_BS = create.bspline.basis(range(Wv), nbasis = nbBSo)
SBS_mat = smooth.basisPar(argvals = Wv,
            y =t(X_W), fdobj = Psi_BS,lambda=0,
            Lfdobj = 0)
Phi_BS = create.bspline.basis(range(Wv), nbasis = L1,
              norder= 4)
Q    = inprod(Phi_BS,Psi_BS)
c_mat = t(SBS_mat$fd$coefs)
X_BS   = c_mat%*%t(Q)
dim(X_BS)

#----
#Smoothing estimation using Fourier representation for the functional
#covariable and L1 =  21 Fourier basis functions for the beta function
#----
par(mfrow=c(1,2))
library(glmnet)
#Penalization with first derivate
P_mat_F = eval.penalty(Phi_F,Lfdobj=1)
ei_P  =  eigen(P_mat_F)
gamma = ei_P$vectors
dv = ei_P$values
dv = ifelse(dv<1e-10,0,dv)

X_Fa =  X_F%*%gamma

#Grid of lambda values obtained by varying the proportion of variance
explained
#by the functional predictor
lamb_FR_f<-function(Xa,dv,K=100,li=1e-1,ls=1-1e-12)
{
 Pos =  which(dv<1e-10)
 D_inv = diag(1/dv[-Pos])
 PEV = seq(li,ls,length=K)
 Xa  = Xa[,-Pos]
 lambv = (1-PEV)/PEV*mean(diag(Xa%*%D_inv%*%t(Xa)))
 lambv = exp(seq(min(log(lambv)),max(log(lambv)),length=K))
 sort(lambv,decreasing = TRUE)
}

lambda = lamb_FR_f(X_Fa,dv,K=1e2)
```

```
y=yv
library(BGLR)
#Linear predictor specification in BGLR for the PBFR model
Pos = which(dv>0)
#Matrix design for the non-penalized columns of X^a
X_Fa1 =  X_Fa[,-Pos]
#Matrix design for the penalized columns of X^a
X_Fa2 =  X_Fa[,Pos]%*%diag(1/sqrt(dv[Pos]))
#Matrix design of environment
X_E = model.matrix(~0+Env,data=dat_F)[,-1]
#Linear predictor to PBFR + Env effect
ETA = list(list(X=X_E,model='FIXED'),list(X=X_Fa1,model='FIXED'),
list(X=X_Fa2,model='BRR'))

# Linear predictor to BFR + Env effect
ETA_NP = list(list(X=X_E,model='FIXED'),list(X=X_F[,1],
                        model='FIXED'),list(X=X_F[,-1],model='BRR'))

#Random cross-validation
RP = 100
set.seed(1)
MSEP_df = data.frame(RP=1:RP,MSEP_PFR=NA,MSEP_FR=NA,lamb_o=NA)
for(p in 1:RP)
{
 Pos_tst =  sample(n,0.20*n)
 X_F_tr  = X_F[-Pos_tst,]; n_tr =  dim(X_F_tr)[1]
 y_tr   =  y[-Pos_tst]; y_tst = y[Pos_tst]

 #FR
 dat_df =  data.frame(y=y,X=X_F)
 A_F = lm(y~.,data=dat_df[-Pos_tst,])
 yp_tst  =  predict(A_F,newdata = dat_df[Pos_tst,])
 MSEP_df$MSEP_FR[p] = mean((y_tst-yp_tst)^2)

 #PFR with first derivative
 A_PFR = cv.glmnet(x =X_Fa[-Pos_tst,],y = y[-Pos_tst] ,alpha = 0,
           nfolds=5,lambda=lambda/n_tr,
           penalty.factor=dv,
           standardize=FALSE,maxit=1e6)
 MSEP_df$lamb_o[p] = A_PFR$lambda.min
 yp_tst =  predict(A_PFR,newx=X_Fa[Pos_tst,], s="lambda.min")[,1]
 MSEP_df$MSEP_PFR[p]  = mean((y_tst-yp_tst)^2)

 #BGLR
 y_NA =  y
 y_NA[Pos_tst] = NA
 A_BGLR =  BGLR(y_NA,ETA= ETA,nIter=1e4, burnIn = 1e3,verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR[p]  = mean((y_tst-yp_tst)^2)

 A_BGLR =  BGLR(y_NA,ETA= ETA_NP,nIter=1e4, burnIn = 1e3,
verbose=FALSE)
```

```
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR_NP[p] = mean((y_tst-yp_tst)^2)

 cat('Partition = ', p,'\n')

}
MSEP_df

#Mean and SD of MSEP across 10 RP
Tab = apply(MSEP_df[,-1],2,function(x)c(mean(x),sd(x)))
Tab

#---
#Smoothing estimation using B-spline representation for the functional
#covariable and L1 =  21 B-spline basis functions for the beta function
#---
library(glmnet)
#Penalization with first derivate
P_mat_F = eval.penalty(Phi_BS,Lfdobj=1)
ei_P  =  eigen(P_mat_F)
gamma = ei_P$vectors
dv = ei_P$values
dv = ifelse(dv<1e-10,0,dv)
X_Fa =  X_BS%*%gamma

#Grid of lambda values obtained by varying the proportion of variance
explained
#by the functional predictor
lamb_FR_f<-function(Xa,dv,K=100,li=1e-1,ls=1-1e-12)
{
 Pos =  which(dv<1e-10)
 D_inv = diag(1/dv[-Pos])
 PEV = seq(li,ls,length=K)
 Xa  = Xa[,-Pos]
 lambv = (1-PEV)/PEV*mean(diag(Xa%*%D_inv%*%t(Xa)))
 lambv = exp(seq(min(log(lambv)),max(log(lambv)),length=K))
 sort(lambv,decreasing = TRUE)
}

lambda = lamb_FR_f(X_Fa,dv,K=1e2)

#Random cross-validation
y=yv
library(BGLR)
Pos = which(dv>0)
X_Fa1 =  X_Fa[,-Pos]
X_Fa2 =  X_Fa[,Pos]%*%diag(1/sqrt(dv[Pos]))
#Matrix design of environment
X_E = model.matrix(~0+Env,data=dat_F)[,-1]
#Linear predictor to PBFR + Env effect
ETA = list(list(X=X_E,model='FIXED'),list(X=X_Fa1,model='FIXED'),
      list(X=X_Fa2,model='BRR'))
```

```
#Linear predictor to BFR + Env effect
ETA_NP = list(list(X=X_E,model='FIXED'),list(X=X_BS,model='BRR'))
RP = 100
set.seed(1)
MSEP_df = data.frame(RP=1:RP,MSEP_PFR=NA,MSEP_FR=NA,lamb_o=NA)
for(p in 1:100)
{
 Pos_tst = sample(n,0.20*n)
 X_F_tr = X_BS[-Pos_tst,]; n_tr = dim(X_F_tr)[1]
 y_tr  = y[-Pos_tst]; y_tst = y[Pos_tst]

 #FR
 dat_df = data.frame(y=y,X=X_BS)
 A_F = lm(y~.,data=dat_df[-Pos_tst,])
 yp_tst = predict(A_F,newdata = dat_df[Pos_tst,])
 MSEP_df$MSEP_FR[p] = mean((y_tst-yp_tst)^2)

 #PFR with first derivative
 A_PFR = cv.glmnet(x =X_Fa[-Pos_tst,],y = y[-Pos_tst] ,alpha = 0,
         nfolds=5,lambda=lambda/n_tr,penalty.factor=dv,
         standardize=FALSE,maxit=1e6)
 #plot(A_PFR)
 MSEP_df$lamb_o[p] = A_PFR$lambda.min
 yp_tst = predict(A_PFR,newx=X_Fa[Pos_tst,], s="lambda.min")[,1]
 MSEP_df$MSEP_PFR[p] = mean((y_tst-yp_tst)^2)

 #BGLR
 y_NA = y
 y_NA[Pos_tst] = NA
 A_BGLR = BGLR(y_NA,ETA= ETA,nIter=1e4, burnIn = 1e3,verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR[p] = mean((y_tst-yp_tst)^2)

 A_BGLR = BGLR(y_NA,ETA= ETA_NP,nIter=1e4, burnIn = 1e3,
verbose=FALSE)
 yp_tst = A_BGLR$yHat[Pos_tst]
 MSEP_df$MSEP_BGLR_NP[p] = mean((y_tst-yp_tst)^2)

 cat('Partition = ', p,'\n')

}
MSEP_df

#Mean and SD of MSEP across 100 RP
Tab = apply(MSEP_df[,-1],2,function(x)c(mean(x),sd(x)))
Tab
```

# References

Cardot H, Sarda P (2006) Linear regression models for functional data. In: Sperlich S, Härdle W, Aydınlı G (eds) The art of semiparametrics. Contributions to statistics. Physica-Verlag HD

Górecki T, Krzyśko M, Waszak Ł, Wołyński W (2018) Selected statistical methods of data analysis for multivariate functional data. Stat Pap 59(1):153–182

Hastie T, Tibshirani R, Friedman J (2009) The elements of statistical learning: data mining, inference, and prediction. Springer, USA

Montesinos-López OA, Montesinos-López A, Crossa J, de Los Campos G, Alvarado G, Mondal S, Rutkoski J, González-Pérez L, Burgueño J (2017a) Predicting grain yield using canopy hyperspectral reflectance in wheat breeding data. Plant Methods 13(4). https://doi.org/10.1186/s13007-016-0154-2

Montesinos-López A, Montesinos-López OA, Cuevas J, Mata-López WA, Burgueño J, Mondal S, Huerta J, Singh R, Autrique E, González-Pérez L, Crossa J (2017b) Genomic Bayesian functional regression models with interactions for predicting wheat grain yield using hyperspectral image data. Plant Methods 13(62). https://doi.org/10.1186/s13007-017-0212-4

Perperoglou A, Sauerbrei W, Abrahamowicz M, Schmid M (2019) A review of spline function procedures in R. BMC Med Res Methodol 19(46):1–16

Quarteroni A, Sacco R, Saleri F (2000) Numerical mathematics. Springer

Ramsay J, Hooker G, Graves S (2009) Functional data analysis with R and MATLAB. Springer

Ruppert D, Wand MP, Carroll RJ (2003) Semiparametric regression. Cambridge University Press, Cambridge

Shizgal BD, Jung JH (2003) Towards the resolution of the Gibbs phenomena. J Comput Appl Math 161(1):41–65

Usset J, Staicu AM, Maity A (2016) Interaction models for functional regression. Comput Stat Data Anal 94:317–329. https://doi.org/10.1016/j.csda.2015.08.020

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Chapter 15
# Random Forest for Genomic Prediction

## 15.1 Motivation of Random Forest

The complexity and high dimensionality of genomic data require using flexible and powerful statistical machine learning tools for effective statistical analysis. Random forest (RF) has proven to be an effective tool for such settings, already having produced numerous successful applications (Chen and Ishwaran 2012). RF is a supervised machine learning algorithm that is very flexible, easy to use, and that without a lot of effort produces very competitive predictions of continuous, binary, categorical, and count outcomes. Also, RF allows measuring the relative importance of each predictor (independent variable) for the prediction. For these reasons, RF is one of the most popular and powerful machine learning algorithms that has been successfully applied in fields such as banking, medicine, electronic commerce, stock market, and finance, among others.

Due to the fact that there is no universal model that works in all circumstances, many statistical machine learning models have been adopted for genomic prediction. RF is one of the models adopted for genomic prediction with many successful applications (Sarkar et al. 2015; Stephan et al. 2015; Waldmann 2016; Naderi et al. 2016; Li et al. 2018). For example, García-Magariños et al. (2009) found that RF performs better than other methods for binary traits when the sample size is large and the percentage of missing data is low (García-Magariños et al. 2009). Naderi et al. (2016) found that, for binary traits, RF outperformed the GBLUP method only in a scenario combining the highest heritability, the largest dense marker panel (50K SNP chip), and the largest number of QTL. González-Recio and Forni (2011) found that RF performed better than Bayesian regression when detecting resistant and susceptible animals based on genetic markers. They also reported that RF produced the most consistent results with very good predictive ability and outperformed other methods in terms of correct classification.

Some of the reasons for the increased popularity of RF models are (a) they require very simple input preparation and can handle binary, categorical, count, and

© The Author(s) 2022

O. A. Montesinos López et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0_15

continuous dependent variables without the need for any preprocessing also of independent variables like scaling, (b) they perform implicit variable selection and provide a ranking of predictor (feature) importance, (c) they are inexpensive in terms of computational resources needed for their training since there are few hyperparameters that commonly need to be tuned (number of trees, number of features sampled, and number of samples in the final nodes) and because instead of working directly with all independent variables simultaneously each time, they use only a fraction of the independent variables, (d) some algorithms can beat random forests, but it is never by much, and other algorithms many times take much longer to build and tune than an RF model, (e) contrary to deep neural networks that are really hard to build, it is really hard to build a bad random forest, since it depends on very few hyperparameters and some of them are not very sensitive, which means that a lot of tweaking and fiddling is not required to get a decent random forest model, (f) they have a very simple learning algorithm, (g) they are easy to implement since there are many free and open-source implementations, and (h) RF parallelization is possible because each decision tree is grown independently.

For this reason, this chapter provides the fundamentals for building RF models as well as many illustrative examples for continuous, binary, categorical, and count response variables in the context of genomic prediction. All examples are provided in the context of genomic selection with the goal of facilitating the learning process of users that do not have a strong background in statistics and computer science.

## 15.2 Decision Trees

A decision tree is a prediction model used in various fields ranging from social science to astrophysics. Given a set of data, logic construction diagrams are manufactured that are very similar to rule-based prediction systems, which serve to represent and categorize a series of conditions that occur in succession, to solve a problem. Nodes in a decision tree involve testing a particular independent variable (attribute). Often, an attribute value is compared with a constant.

As can be seen in Fig. 15.1, decision trees use a conquer-and-divide approach for regression or classification. They work top-down, at each stage seeking to split an attribute that best separates the classes in classification problems, and then recursively processing the subproblems that result from the split. Under this strategy, a decision tree is built that can be converted into decision rules. Leaf nodes give a classification that applies to all instances that reach the leaf, or a set of classifications or a probability distribution over all possible classifications (Fig. 15.1b). In a decision tree, each internal node represents a "test" on an independent variable, each branch represents the outcome of the test, and each leaf node represents a class label in a classification problem. To classify a new individual, it is routed down the tree according to the values of the independent variables tested in successive nodes, and when a leaf is reached, the new individual is classified according to the class assigned to the leaf.

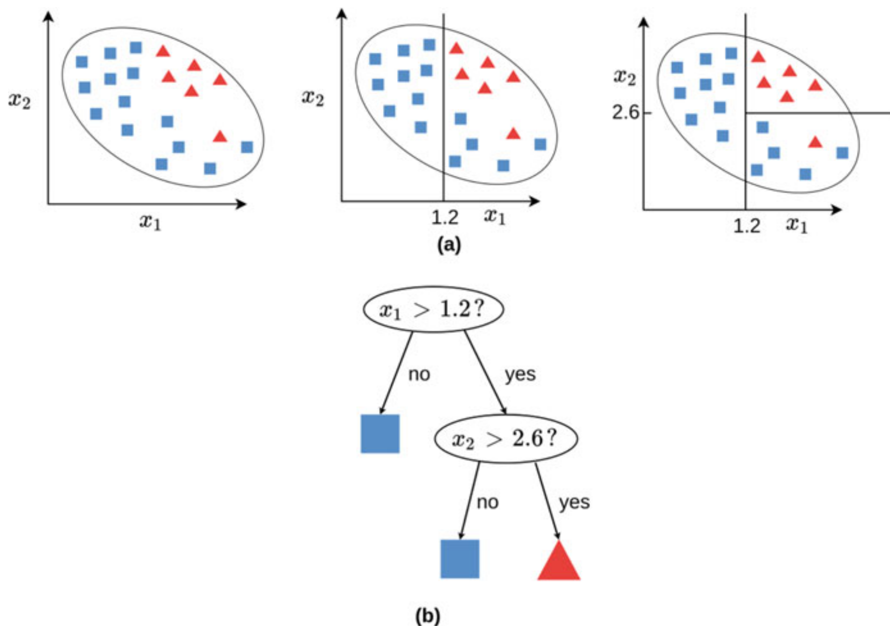



**Fig. 15.1** Illustration of a decision tree for a classification problem

Next, the following synthetic data set illustrates how to use the library party with the function ctree().

```
> Data_GY
     GY    X1    X2    X3    X4
1   9.98  7.56  8.45  10.0  8.99
2   9.48  7.39  8.23  9.92  9.04
3  10.0   8.32  8.84  9.25  10.0
4   7.11  6.96  7.13  9.24  8.49
5   9.07  7.07  9.07  9.41  8.63
6  10.0   9.02  9.65  10.0  9.93
7   8.79  7.91  8.27  8.89  8.48
8   8.61  7.24  8.22  8.38  9.86
9  10.0   8.34  8.55  9.50  8.95
10 10.0   8.30  9.72  9.66  10.0
11  7.84  6.60  8.49  9.00  8.67
12  6.41  7.11  7.34  8.83  8.29
```

The basic R code to build decision trees in the library is given next:

```
Control_GY=ctree_control(teststat = c("quad", "max"),
                    testtype = c("Univariate"),
        mincriterion = 0.05, minsplit =2, minbucket = 1,
        stump = FALSE, nresample = 101, maxsurrogate =2,
```

```
    mtry =2, savesplitstats = TRUE, maxdepth = 30, remove_weights =
    FALSE)

Grades_tree=ctree(GY~X1+X2+X3+X4, controls= Control_GY,
data=Data_GY)
```

The output of the decision tree built with the ctree() function is given next:

```
> Grades_tree
```

Conditional inference tree with five terminal nodes
Response: GY
Inputs: X1, X2, X3, X4
Number of observations: 12

```
1) X2 <= 7.34; criterion = 0.991, statistic = 6.812
 2)* weights = 2
1) X2 > 7.34
 3) X1 <= 7.24; criterion = 0.982, statistic = 5.561
  4)* weights = 3
 3) X1 > 7.24
  5) X3 <= 9.25; criterion = 0.85, statistic = 2.07
   6)* weights = 2
  5) X3 > 9.25
   7) X4 <= 9.04; criterion = 0.55, statistic = 0.57
    8)* weights = 3
   7) X4 > 9.04
    9)* weights = 2
```

And the corresponding plot of this decision tree is given next (Fig. 15.2):

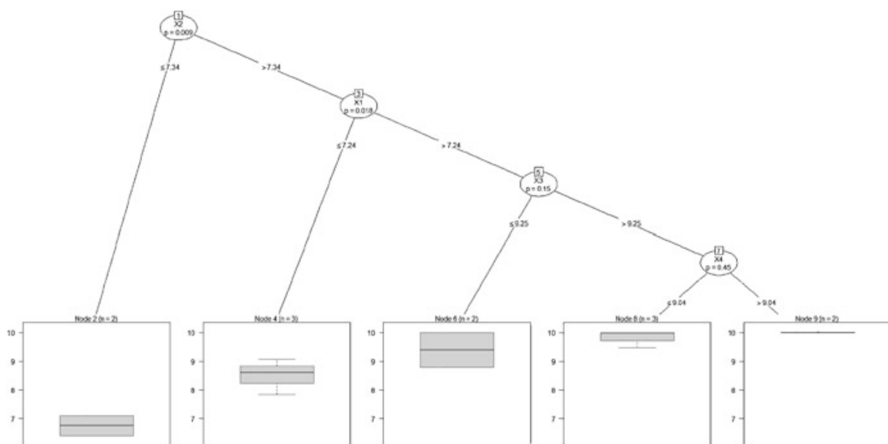


Fig. 15.2 Decision tree for a regression problem with five terminal nodes

The ctree () function of the party package allows you to fit conditional decision trees. The choice between a regression tree or a classification tree is made automatically depending on whether the response variable is of a continuous type or factor. It is important to note that only these two types of response variables are allowed; if one type of character is passed, an error is returned.

The predictive capacity of models based on a single tree is considerably lower than that achieved with other models. This is due to the tendency of those models to overfitting and high variance. One way to improve the generalizability of decision trees is to use regularization or to combine multiple trees, and one of these approaches is called random forest. Decision trees are also sensitive to unbalanced training data (one class dominates over the others). When dealing with continuous predictors, decision trees lose some of their information by categorizing them at the time of node splitting. As described before, the creation of tree branches is achieved by the recursive binary splitting algorithm. This algorithm identifies and evaluates the possible divisions of each predictor according to a certain measure (RSS, Gini, entropy, etc.). Continuous predictors are more likely to contain, just by chance, some optimal cutoff point, which is why they tend to be favored in the creation of trees. For these reasons, decision trees are not able to extrapolate outside the range of the predictors observed in the training data.

## 15.3 Random Forest

Random forest (RF) is a decision tree-based supervised statistical machine learning technique. Its main advantage is that you get better generalization performance for similar training performance than with decision trees. This improvement in generalization is achieved by compensating for the errors in the predictions of the different decision trees. To ensure that the trees are different, what we do is that each one is trained with a random sample of the training data. This strategy is called bagging.

As mentioned before, RF is a set (ensemble) of decision trees combined with bootstrapping (bagging). When using bootstrapping, what actually happens is that different trees see different portions of the data. No tree sees all the training data. This entails training each tree of the forest with quite different data samples for the same problem. In this way, by combining their results, some errors are compensated for by others and we have a prediction that generalizes better. The RF adds additional randomness to the model while growing the trees. Instead of searching for the most important independent variable while splitting a node, it searches for the best independent variable among a random subset of independent variables. This generates a wide heterogeneity that generally improves the model performance. A good binary split partitions data from the parent tree node into two daughter nodes so that the ensuing homogeneity of the daughter nodes is improved by the parent node.

In Fig. 15.3 we can see that a collection of *ntree* >1 trees is grown in which each tree is grown independently using a bootstrap sample of the original data. The terminal nodes of the tree contain the predicted values which are tree-aggregated

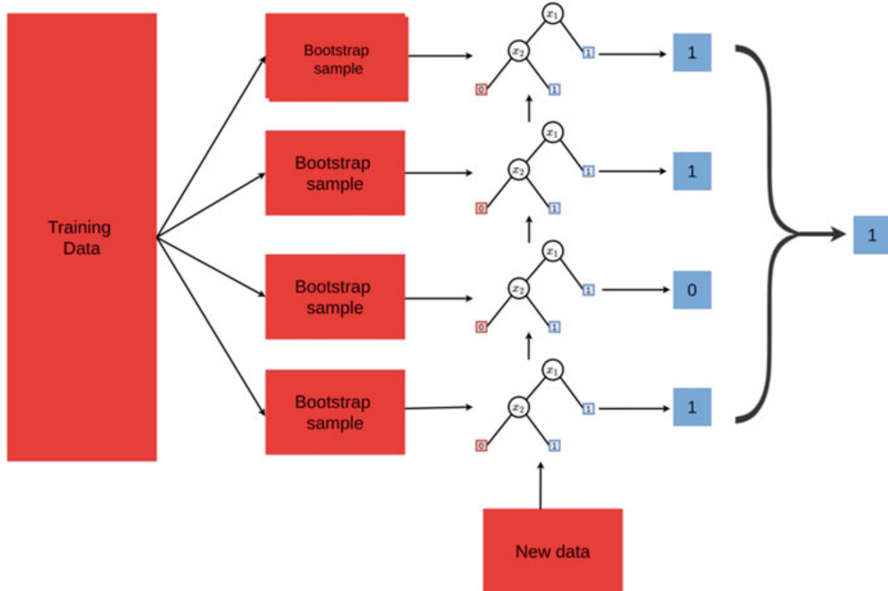



**Fig. 15.3** Illustration of how the random forest model works and combines multiple trees

to obtain the forest predictions. For example, in classification, each tree casts a vote for the class and the majority vote determines the predicted class label, while in regression problems the predicted value is the average of the observations in the terminal nodes. Rather than splitting a tree node using all $p$ independent variables (features), RF, as mentioned above, selects at each node of each tree, a random subset of $1 \leq mtry \leq p$ independent variables that is used to split the node where typically $mtry$ is substantially smaller than $p$. The purpose of this two-step randomization is to decorrelate trees and reduce variance. RF trees are grown deeply (with many ramifications) which reduces bias. In general, a RF tree is grown as deeply as possible under the constraint that each terminal node must contain no fewer than nodesize $\geq$1 cases. The nodesize is the minimum size of terminal nodes.

Figure 15.4 provides more details of how both randomizations take place. We can see that the bootstrap samples have the same number of observations as the original training data, but with the difference that not all observations are present since some rows are repeated. In general, 2/3 (63.2%) of the original observations are maintained in a bootstrap sample. Also, we can see in Fig. 15.4 that not all independent variables are present in each bootstrap sample, and that only three randomly selected samples (observations) are present in each bootstrapped sample. It is important to point out that the bootstrapping phase is independent of the feature subsampling.

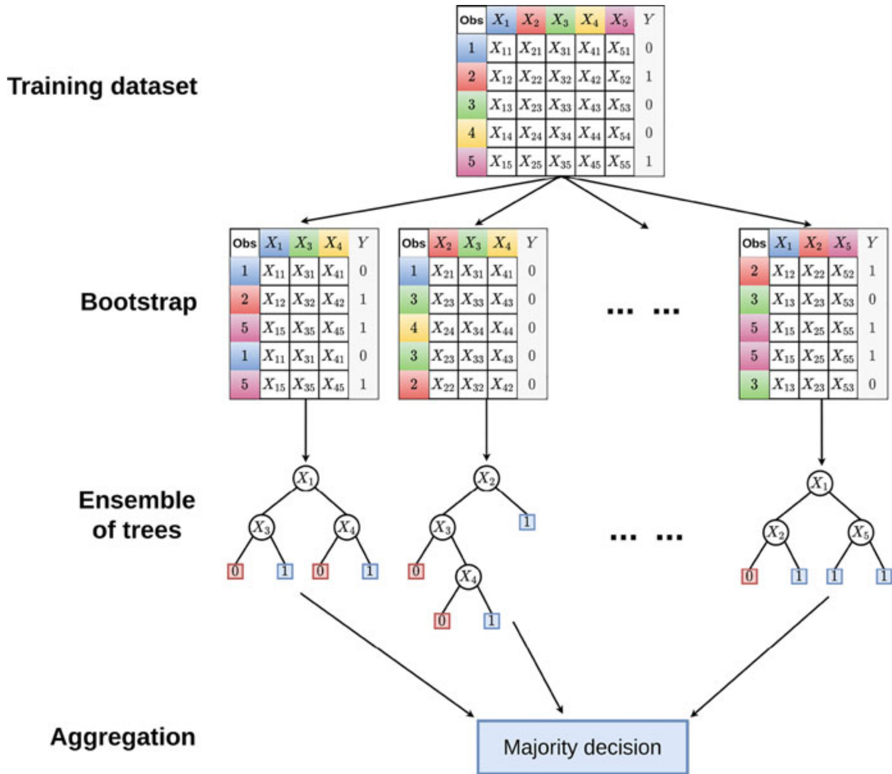



**Fig. 15.4** Illustration of how bootstrap samples and samples of predictors are selected under the random forest model

## 15.4 RF Algorithm for Continuous, Binary, and Categorical Response Variables

RF is an interchange of bootstrap aggregating that builds a large collection of trees, and then averages out the results. Each tree is built using a splitting criterion (loss function), that should be appropriate for each type of response variables (continuous, binary, categorical, and count). For training data (Breiman 2001), RF takes $B$ bootstrap samples and randomly selects subsets of independent variables as candidate predictors for splitting tree nodes. Each decision tree will minimize the average loss function in the bootstrapped (resampled) data and is built up using the following algorithm:

$$\text{For } b = 1, \ldots, B \text{ bootstrap samples } \{\boldsymbol{y}_b, \boldsymbol{X}_b\}$$

Step 1. From the training data set, draw bootstrap samples of size $N_{\text{train}}$.

Step 2. With the bootstrapped data, grow a random forest tree $T_b$ with the specific splitting criterion (appropriate for each response variable), by recursively repeating

the following steps for each terminal node of the tree, until the minimum node size (minimum size of terminal nodes) is reached.

(a) Randomly draw *mtry* out of the $p$ independent variables (IVs); *mtry* is a user-specified parameter and should be less or equal to $p$ (total number of IVs).
(b) Pick the best independent variable among the *mtry* IVs.
(c) Split the node into two child nodes. The split ends when a stopping criterion is reached, for instance, when a node has less than a predetermined number of observations. No pruning is performed.

Step 3. The ensemble of trees is obtained $\{T_b\}_1^B$.

The predicted value of testing set $(\widehat{y}_i)$ individuals with input $\boldsymbol{x}_i$ is calculated as $\widehat{y}_i = \frac{1}{B}\sum_{b=1}^{B} T_b(\boldsymbol{x}_i)$ when the response variable is continuous, but when the response variable is binary or categorical, the predicted values are calculated as $\widehat{y}_{ic} =$ majority vote $\left\{\widehat{C}_b(\boldsymbol{x}_i)\right\}_1^B$, where $\widehat{C}_b(\boldsymbol{x}_i)$ is the class prediction of the $b$th RF tree. Readers are referred to Breiman (2001) and Waldmann (2016) for details on the theory of RF. As explained above, the choice of splitting function is very important since we need to use different splitting rules for each type of response variable.

Also, the adequate selection of these hyperparameters could significantly improve the prediction performance of the models, but the choice of the optimal values is case study-dependent. Next are given the importance and influence of these tuning parameters in the prediction performance considering that input and response variables correspond to the same time point or individual.

(a) Number of trees in the forest (*ntree*). The number of decision trees used to build up the ensemble has an explicit influence on prediction performance. The higher the number of trees, the smaller the error. But this trend is asymptotic: if the number of trees is large enough, increasing the number of trees does not result in significant improvement of the prediction accuracy. Besides, using more trees requires longer computing times. For this reason, the number of trees is set based on a trade-off solution between computing time and predictive performance. Each tree makes use of around two-thirds (63.2%) of the observations to build the tree. The remaining observations are referred to as Out-Of-Bag (OOB). One may predict the response for the ith observation using all of the trees in which these observations are OOB.
(b) Number of independent variables randomly selected to be contemplated at each split (*mtry*) in RF. To choose the value of the *mtry* parameter, it is necessary to consider the correlation between the input variables. With highly correlated input variables, it is preferable to use a small value. Generally, $mtry = p/3$ for regression forests (continuous outputs) and $mtry = \sqrt{p}$ for classification (binary or categorical outputs) forests (where $p$ is the total number of input variables). On the other hand, if there are many irrelevant input variables, a larger value of *mtry* would be needed in order to obtain better predictions. In any problem, there

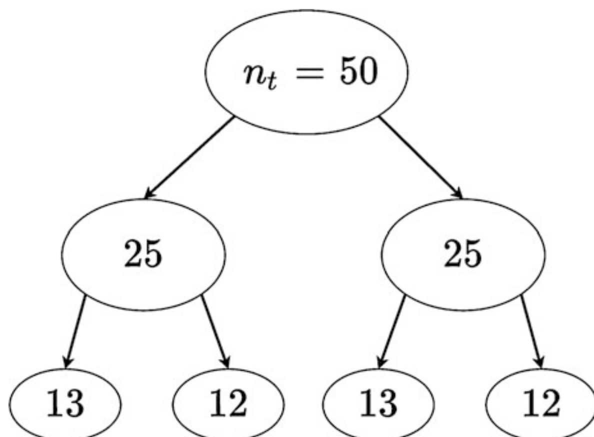



**Fig. 15.5** Illustration of the minimum node size in a decision tree

are independent variables that may be of relatively minor importance (irrelevant). An input variable highly uncorrelated with the rest of the input could be very important due to its unique role in the analysis, or not important at all in the prediction if not linked to the response.

(c) Node size. A decision tree works by recursive partitioning of the training set. Every node t of a decision tree is related to a set of $n_t$ observations from the training set:

By node size we understand the **minimum node size**; in Fig. 15.5 the minimum node size is 12. This parameter essentially sets the depth of your decision trees. The depth of each particular decision tree can be either fine-tuned manually or have the algorithm select it automatically. When decision trees grow too deep, there is the risk of overfitting. For this reason, this parameter also needs to be tuned. The terminal nodes of the tree contain the predicted values which are tree-aggregated (by average for continuous variables and majority vote for categorical variables) to obtain the forest predictions. This means that in classification, each tree casts a vote for the class and the majority vote determines the predicted class label, while in continuous response variables the average is reported as prediction performance.

## *15.4.1 Splitting Rules*

The splitting rule defined as the rule on how to decide whether to split it is a central component of RF models and crucial for the performance of each tree of a RF model. For continuous response variables, there are many criteria to decide where to split, and the most common splitting criterion is the least squares criterion. Suppose the proposed split for the root node is of the form $X \leq$ c and $X >$ c for a continuous variable $X$, and a split threshold of c. The best split is the one that minimizes the weighted sum of square errors (SSE):

$$\mathrm{SSE} = \mathrm{SSE}_L \Omega_L + \mathrm{SSE}_R \Omega_R,$$

where $\mathrm{SSE}_L = \sum_{i=1}^{L} (y_i - \bar{y_L})^2$ represents the sum of square errors for the left node, $L$ denotes the number of elements that contain the left partition, $\bar{y_L}$ is the mean of the response variables of elements in the left partition and $\Omega_L = \frac{n_L}{n}$ is the proportion of observations of the left node. $\mathrm{SSE}_R = \sum_{i=1}^{R} (y_i - \bar{y_R})^2$ denotes the sum of square errors for the right node, $R$ is the number of elements that contain the right partition, and $\bar{y_R}$ is the mean of the *response* variables of elements in the right partition. $\Omega_R = \frac{n_R}{n}$ is the proportion of observations of the right node and $n = n_L + n_R$. When $\Omega_L = \Omega_R = 1$, the *SSE* is reduced to its unweighted version.

For binary and categorical response variables, there are some options for splitting criteria. Next, we provide the weighted Gini index criterion that should minimize

$$\mathrm{GI} = \left[\sum_{i=1}^{C} p_{Li}(1 - p_{Li})\right]\Omega_L + \left[\sum_{i=1}^{C} p_{Ri}(1 - p_{Ri})\right]\Omega_R,$$

where $C$ is the number of classes in the response variable, $p_{Li} = \frac{n_{Li}}{n_L}$ is the probability of occurrence of class $i$ in the left node, $\Omega_L = \frac{n_L}{n}$ is the proportion of observations in the left node, $p_{Ri} = \frac{n_{Ri}}{n_R}$ is the probability of occurrence of class $i$ in the right node, and $\Omega_R = \frac{n_R}{n}$ is the proportion of observations in the right node and $n = n_L + n_R$. The GI making $\Omega_L = \Omega_R = 1$ is reduced to the unweighted Gini index.

Another splitting criterion for binary and categorical response variables is the weighted binary (or categorical) cross-entropy, which is defined as

$$\mathrm{CE} = -\left[\sum_{i=1}^{C} p_{Li} \log\left(p_{\mathrm{Li}}\right)\right]\Omega_L - \left[\sum_{i=1}^{C} p_{Ri} \log\left(p_{\mathrm{Ri}}\right)\right]\Omega_R$$

Also, the weighted binary (or categorical) cross-entropy is reduced to its unweighted version by making $\Omega_L = \Omega_R = 1$. For most of the examples, we will use the fast unified random forests for survival, regression, and classification of (randomForestSRC) R package. This package performs parallel computing of Breiman's random forests (Breiman 2001) for a variety of data settings including regression, classification, and right-censored survival and competing risks (Ishwaran and Kogalur 2008). Other important applications cover multivariate classification/ regression, quantile regression (see quantreg), unsupervised forests, and novel solutions for class imbalanced data. However, in this chapter, we will only use the randomForestSCR library for illustrating the implementation of RF for univariate and multivariate outcomes for binary, categorical, and continuous response variables. Different splitting rules can be invoked under RF applications, for which variable importance measures (VIM) can also be computed for each predictor. There are many measures of variable importance; one common approach for regression trees is to calculate the decrease in prediction accuracy from the testing data set. For each tree, the testing set portion of the data is passed through the tree and the prediction error (PE) is recorded. Each predictor variable is then randomly permuted

and $j$ new PE is computed. The differences between the two are then averaged over all the trees, and normalized by the standard deviation of the differences. The variable showing the largest decrease in prediction accuracy is the most important variable. These VIM can be displayed in a variable importance plot of the top-ranked variables.

To implement RF models in the randomForestSRC package, we used the function rfsrc(), for which we show the main elements of its usage:

```
rfsrc(formula, data, ntree = 1000, mtry = NULL,  nodesize= NULL,
splitrule = NULL,  importance =TRUE),
```

where formula is a symbolic description of the model to be fitted. If missing, unsupervised splitting is implemented, data is a data frame containing the $y$-outcome and $x$-variables, *ntree* represents the number of trees, and *mtry* denotes the number of variables randomly selected as candidates for splitting a node. The default is $mtry = p/3$ for regression, where $p$ equals the number of independent variables. For all other families (including unsupervised settings), the default is $mtry = \sqrt{p}$. Values are always rounded up. Nodesize denotes the forest average number of unique cases (data points) in a terminal node. The defaults are classification (1), regression (5), competing risk (15), survival (15), mixed outcomes (3), and unsupervised (3). It is good practice to explore with different nodesize values. Splitrule denotes the splitting rule, for regression analysis (continuous response variables) can be used the mean squared error (mse) also known as the least square criterion. The mse splitting rule implements the weighted mean squared error splitting criterion (Breiman et al. 1984, Chapter 8.4), while for classification analysis (binary and categorical response variables), there are three splitting rules available in this package: (a) Gini: default splitrule implements Gini index splitting (Breiman et al. 1984, Chapter 4.3), (b) auc: AUC (area under the ROC curve) splitting for both two-class and multi-class settings. AUC splitting is appropriate for imbalanced data, and (c) entropy: entropy splitting (Breiman et al. 1984, Chapter 2.5, 4.3) and *importance* = TRUE compute VIM for each predictor. Default action is code importance = “none”.

Next, we provide some examples for implementing RF models for continuous, binary, and categorical response variables.

**Example 15.1**

For this example, we also used the Data_Toy_EYT.RData data set composed of 40 lines, four environments (Bed5IR, EHT, Flat5IR, and LHT), and four response variables: DTHD, DTMT, GY, and Height. G_Toy_EYT is the genomic relationship matrix of dimension $40 \times 40$. The first two variables are ordinal with three categories, the third is continuous (GY = Grain yield), and the last one (Height) is binary.

First, using the continuous response variable (GY), we illustrate how to implement the RF for continuous outcomes. We build the design matrices that jointly will form the input for the RF model. The design matrices for this example were built as

```
### Design matrix of genotypes ###
ZG <- model.matrix(~0 + GID, data=Pheno)
### Compute the Cholesky factorization of the genomic relationship
matrix
ZL <- chol(Geno)
###Incorporating the information of the GRM to the design matrix of
genotypes
ZGL <- ZG %*% ZL
####Design matrix of environments
ZE <- model.matrix(~0 + Env, data=Pheno)
### Design matrix of the interaction between genotype and environment
(GE)
ZGE <- model.matrix(~0 + ZGL:Env, data=Pheno)
```

Joining the design matrices of environments and genotypes

```
X <- cbind(ZGL, ZE)
```

First, using all the data set at hand in the Data_Toy_EYT.RData, we show how to train an RF model with a continuous outcome and how to extract and plot variable importance. It is important to point out that as input we not used directly the information of markers but the square root of the genomic relationship matrix.

```
# Data frame with the response variable and all predictors (environments
+ genotypes)
Data <- data.frame(y=Pheno$GY, X)
# Fit the model with importance=TRUE for computing the variable
importance
model <- rfsrc(y ~ ., data=Data, importance=TRUE)
```

Get the variable importance

```
> head(model$importance)
  EnvBed5IR     EnvEHT       EnvFlat5IR    EnvLHT
 0.1471930788  0.3345026471 0.2117008212  1.7597458640
   GID6569128    GID6688880
-0.0001849502 -0.0003675786
```

Plot the ten most important variables with their own functions

```
Plot <- plot_importances(model$importance, how_many=10)
Plot
```

As mentioned above, with the rfsrc(), the RF model is fitted for continuous response variables that use the mse splitting rule by default; then with model$importance we extract the variable importance values for each of the independent variables of matrix X; however, only ten of them are printed. When the response variables are categorical, the values of VIM are probabilities; for this

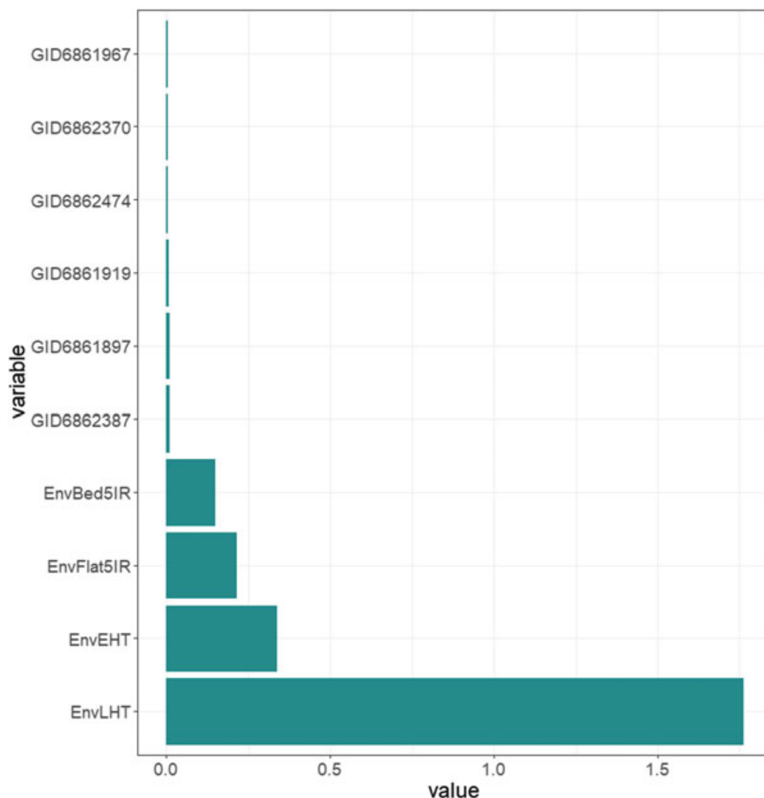



**Fig. 15.6** Variable importance measures (VIM) for the ten most important predictors of the data set Data_Toy_EYT.RData

reason, the output of the VIM is a probability distribution, that is, a probability for each categorical response that sums up to 1 in each row (individual). For this reason, extracting the VIM should be done for each category; for example, model$importance[, 2] and model$importance[, C] extract the VIM for categories 2 and C, respectively, of the response variable. Then with the plot_importance() that is available in Appendix 1, a plot is generated for only the ten most important independent variables. This plot is given in Fig. 15.6 for the example given above for the continuous response variable (GY). The complete code for generating Fig. 15.6 is given in Appendix 2.

Figure 15.6 shows that the most important predictors are the dummy variables of environments and that each of the lines has a small effect. However, it is important to point out that in this plot, genetic and environmental effects are together, but what we see in Fig. 15.6 is not unusual, since in plant breeding trials the environmental main effect is always larger than the genetic effects.

The way of extracting and building the VIM is similar for continuous, binary, and categorical outcomes; as mentioned above, for this reason, it is not necessary to give examples for other types of response variables.

Next, we will illustrate how to make predictions for new data with RF, but now using not only the information of environments and genotypes in the predictor but also taking into account the genotype–environment interaction. For this reason, again first we stack the information on environment, lines, and genotype–environment interaction.

Joining the design matrices of environments, genotype, and interaction GE term

```
X <- cbind(ZGL, ZE, ZGE)
```

It is important to point out that since the response variable is continuous (GY), the mse, also known as the least squares criterion, is used as the splitting rule. Since the RF model has hyperparameters to tune in these examples, the following grid of values was used for the following three hyperparameters:

```
ntrees=c(100, 200, 300)
mtry=c(80, 100, 120)
nodesize=c(3, 6, 9)
```

This means that the grid contains $3 \times 3 \times 3 = 27$ combinations that need to be evaluated, with part of the training set and then will be selected the best combination with which the RF model will be refitted but using the whole training (outer training) set. With this final fitted model, the predictions for the testing set are performed. It is important to point out that the larger the grid, the greater the possibility that you can improve the prediction performance in the testing set. For the evaluation of the prediction performance, in this example, ten random partitions were used and in each partition 80% of the information was used for outer training and 20% for testing, but the outer training set (80% of the original data set) in each random partition was also split and 80% of the data was for inner training and the remaining 20% for tuning the set. See Chap. 4 for a review of the outer and inner training concepts. The 27 hyperparameter combinations of the grid set given above were evaluated with the inner training set. The basic code used for the tuning process is given next:

```
tuning_model <- rfsrc(y ~ ., data=DataInnerTraining, ntree=flags$ntree,
           mtry=flags$mtry, nodesize=flags$nodesize, splitrule ="mse" )
predictions <- predict(tuning_model, newdata=DataInnerTesting)$
predicted
```

This code is used for the tuning process, and for this reason, the RF with continuous response variable (splitrule = “mse”) is used for each of the 27 combinations of the grid and the predicted values of each of the 27 combinations are computed with the predict() function which requires as inputs the fitted model (tuning_model); the newdata for which the predictions should be obtained in this

case is the DataInnerTesting (tuning set). Finally, from this function, only the predicted values were extracted for computing the prediction performance of each combination in the grid. Then the best hyperparameter combination was selected and with this best combination, the RF was refitted but using all the outer training sets (inner training+tuning set). The R code for doing this training process is the same, but using the whole outer training set:

```
model <- rfsrc(y ~ ., data=DataTraining, ntree=best_params$ntree,
        mtry=best_params$mtry, nodesize=best_params$nodesize,
        splitrule ="mse")
predicted <- predict(model, newdata=DataTesting)$predicted
```

Finally, with this fitted model, the prediction performance was computed for each of the ten testing sets. The average of the ten partitions [MAAPE, average Pearson's correlation (PC), coefficient of determination (R2), and mean square error (MSE) of prediction] is reported as the prediction performance. The code for computing these metrics and the plots of variable important measures are given in Appendix 1. The whole code for reproducing the results given in Table 15.1 is provided in Appendix 3. As pointed out before, if the splitrule is ignored, the mse splitrule is implemented by default.

Table 15.1 indicates that the best predictions under the MAAPE were observed in environments Bed5IR and EHT, while under the PC and R2, the best predictions were observed in environments Bed5IR and Flat5IR. However, under the MSE, the best predictions were in environments Bed5IR and LHT. These results show that the selection is in part affected by the metric used for the evaluation of prediction performance; for this reason, using more than one is suggested.

Next, we show how to use RF for predicting a binary response variable (Height); the independent variables (X) used for this example are exactly the same as the ones used in the previous example for continuous response variables that contain information of environments, genotypes, and genotype×environment interaction. However, since now we want to train RF for a binary outcome, first we need to convert the response variable as factor:

```
Pheno$y <- as.factor(Pheno$y)
```

Pheno$y is the height trait that is binary. Also, the grid used for the tuning process was exactly the same, with 27 combinations resulting from three values of ntrees, three values of mtry, and three values of nodesize. However, now instead of using random partitions for evaluating the prediction performance, we used five-fold cross-validation, and each time four of them were used for training (outer training) and one for testing. We used another strategy of cross-validation only with the purpose of illustrating that in some circumstances one strategy should be preferred over others. However, the k-fold cross-validation guarantees orthogonal folds. For selecting the hyperparameters, we also used five-fold cross-validation with four for training (inner training) and one for tuning, but now this five-fold cross-validation was performed in

**Table 15.1** Prediction performance for the continuous trait (GY) in terms of MAAPE, average Pearson's correlation (PC), coefficient of determination (R2), and mean square error (MSE). Ten random partitions with 80% of data for training and 20% for testing were used

| Env | MAAPE | SE_MAAPE | PC | SE_PC | R2 | SE_R2 | MSE | SE_MSE |
|---|---|---|---|---|---|---|---|---|
| Bed5IR | 0.0421 | 0.0042 | 0.4805 | 0.0747 | 0.2811 | 0.0754 | 0.1087 | 0.0178 |
| EHT | 0.067 | 0.0019 | 0.3004 | 0.1244 | 0.2294 | 0.0889 | 0.3086 | 0.0165 |
| Flat5IR | 0.0933 | 0.0094 | 0.4728 | 0.0951 | 0.305 | 0.0956 | 0.4782 | 0.0717 |
| LHT | 0.1124 | 0.0129 | 0.0193 | 0.1395 | 0.1755 | 0.0783 | 0.1708 | 0.0295 |

each of the outer training sets. The key code for implementing RF for tuning is given next:

```
tuning_model <- rfsrc(y ~ ., data=DataInnerTraining, ntree=flags$ntree,
           mtry=flags$mtry, nodesize=flags$nodesize, splitrule="gini")
predictions <- predict(tuning_model, newdata=DataInnerTesting)$
predicted
```

Now instead of using MSE as splitrule, we used the Gini, which is appropriate for binary and categorical response variables. However, if you do not specify the splitrule, but the response variable is used as factor, the Gini split rule is used by default, but when the response variable is continuous, the MSE split rule is used by default. Again, once the best hyperparameter combination in each outer training set was selected, the RF was refitted with the whole outer training set (inner training +tuning set), and with this final refitted model, the predictions for each testing set are performed. The R code using the rfsrc() for the final fitting process using the whole training set is given next, and it is exactly the same as for the tuning process.

```
 model <- rfsrc(y ~ ., data=DataTraining, ntree=best_params$ntree,
         mtry=best_params$mtry, nodesize=best_params$nodesize,
        splitrule="gini")
predicted <- predict(model, newdata=DataTesting)$predicted
```

The complete R code for implementing RF for the binary response variable is given in Appendix 4. Finally, the proportion of cases correctly classified (PCCC) and the Kappa coefficient (Kappa) in each of the five testing sets are computed with all the predicted values of each testing set, and the average of five-fold is reported as the prediction performance for each environment.

Table 15.2 indicates that the best prediction performance was observed in environments Bed5IR (PCCC = 0.7331 and Kappa = 0.3746) and EHT (PCCC = 0.680079 and Kappa = 0.2964).

Next, we provide the results of implementing RF for the same data set, but now for the categorical response variable DTHD. The implementation was also done with five-fold cross-validation (outer and inner) and all inputs and the grid used for the tuning process were the same as in the previous binary example, except that now the response variable was categorical. The R code given in Appendix 4 can also be used for categorical response variables but by replacing the binary response variable

**Table 15.2** Prediction performance for the binary trait (Height) in terms of the proportion of cases correctly classified (PCCC) and the Kappa coefficient. Five-fold cross-validation was used

| Env | PCCC | SE_PCCC | Kappa | SE_Kappa |
|---|---|---|---|---|
| Bed5IR | 0.7331 | 0.0616 | 0.3746 | 0.183 |
| EHT | 0.68 | 0.107 | 0.2964 | 0.2436 |
| Flat5IR | 0.591 | 0.0679 | 0.0552 | 0.1519 |
| LHT | 0.611 | 0.1056 | 0.2397 | 0.1919 |

**Table 15.3** Prediction performance for the categorical trait (DTHD) in terms of the proportion of cases correctly classified (PCCC) and the Kappa coefficient (Kappa). Five-fold cross-validation was used

| Env | PCCC | SE_PCCC | Kappa | SE_Kappa |
|---|---|---|---|---|
| Bed5IR | 0.7623 | 0.0562 | 0.5693 | 0.0707 |
| EHT | 0.8057 | 0.069 | 0.678 | 0.0938 |
| Flat5IR | 0.7371 | 0.0948 | 0.5921 | 0.1426 |
| LHT | 0.7783 | 0.1095 | 0.5856 | 0.1933 |

(Height) with the categorical response variable (DTHD). The results of the prediction performance in terms of PCCC and Kappa coefficient are given in Table 15.3; the best prediction performance was observed in environments Bed5IR (PCCC = 0.7623 and Kappa = 0.5693) and EHT (PCCC = 0.8057 and Kappa = 0.678).

## 15.5 RF Algorithm for Count Response Variables

The popular RF models presented before were originally developed for continuous, binary, and categorical data. There are also RF models for count data (Chaudhuri et al. 1995; Loh 2002) that can be implanted in R using the package part (Therneau and Atkinson 2019). However, these RF models for count data are not appropriate for counts with an excess of zeros. For this reason, Lee and Jin (2006) proposed an RF method for counts with an excess of zeros, by building the splitting criterion with the zero-inflated Poisson distribution, but the proposed method models both the excess zero part and the Poisson part jointly, which is unlike the basic hurdle and zero-inflated regression models that use two models, thus allowing different covariate effects for each part. The excess zeros are generated by a disconnect process from the count values and the excess zeros can be modeled independently. For this reason, classic regression models for counts with an excess of zeros use a logistic model for predicting excess zeros and a truncated Poisson model for counts larger than zero.

For this reason, next are described two algorithms for count data with an excess of zeros proposed by Mathlouthi et al. (2019) and applied by Montesinos-López et al. (2021) for genomic prediction. The first algorithm is called zero-altered Poisson random forests (ZAP_RF) and the other is called zero-altered Poisson custom random forest (ZAPC_RF), both algorithms [zero-altered Poisson (ZAP) regression models], assumed that $Y = 0$ with probability $\theta$ ($0 \leq \theta < 1$), and that $Y$ follows a zero-truncated Poisson distribution with parameter $\mu$ ($\mu > 0$), given that $Y > 0$ (Mathlouthi et al. 2019). That is, they are based on the ZAP random variable

$$P(Y=y)=\begin{cases}\theta & y=0\\ \dfrac{(1-\theta)\exp(-\mu)\mu^{y}}{(1-\exp(-\mu))y!} & y>0\end{cases}$$

The mean and variance for ZAP are

$$E(Y)=\frac{(1-\theta)\exp(-\mu)}{(1-\exp(-\mu))}\text{ and Var}(Y)$$
$$=\frac{(1-\theta)}{(1-\exp(-\mu))}\left(\mu+\mu^{2}\right)-\left(\frac{(1-\theta)}{(1-\exp(-\mu))}\mu\right)^{2}$$

In general, zero-altered models are two-part models, where the first part is a logistic model, and the second part is a truncated count model. However, under the ZAP_RF and ZAPC_RF, instead of assuming a linear predictor (like ZAP regression models), it is assumed that the links between the covariates and the responses (Mathlouthi et al. 2019) through $\mu$ and $\theta$ are given by nonparametric link functions like

$$\log(\mu)=f_{\mu}(\boldsymbol{x})\text{ and }\log\left(\frac{\theta}{1-\theta}\right)=f_{\theta}(\boldsymbol{x}),\tag{15.1}$$

where $f_{\mu}$ and $f_{\theta}$ are general unknown link functions. A general nonparametric and flexible procedure can be used to estimate $f_{\mu}$ and $f_{\theta}$ in (15.1). However, here we used a random forest in two steps instead of a parametric model:

Step 1. Zero model. Fit a binary RF to the response $I(Y=0)$, that is, the binary variable takes a value of 1 if $Y=0$ and a value of 0 if $Y>0$. This model produces estimates of $\widehat{\theta}$.

Step 2. Truncated model. Fit an RF using only the positive ($Y>0$) observations. Assume there are $N^{+}$ such observations denoted by $Y_{1}^{+},\ldots,Y_{N^{+}}^{+}$. This model produces estimates of $\widehat{\mu}$. However, to exploit the Poisson assumption, the splitting criteria used in the RF with the truncated part was derived from the zero-truncated Poisson likelihood that is equal to:

$$LL^{+}=-N^{+}\log(1-\exp(-\mu))+\log(\mu)\sum_{i}^{N^{+}}Y_{i}^{+}-N^{+}\mu-\sum_{i}^{N^{+}}\log\left(Y_{i}^{+}!\right),\tag{15.2}$$

where $LL^{+}$ is the log-likelihood function of a sample of a zero-truncated Poisson distribution. The estimate of $\mu$ is obtained by solving $\frac{\partial LL^{+}}{\partial\mu}=0$, which reduces to

$$\frac{\sum_{i}^{N^{+}} Y_i^{+}}{N^{+}} = \frac{\mu}{1 - \exp(-\mu)}$$

For a given candidate split, the log-likelihood function given in (15.2) is computed separately in the two children nodes and the best split is the one that maximizes

$$\widehat{LL^{+}}\ (\text{left node}) + \widehat{LL^{+}}\ (\text{right node}),$$

where $\widehat{LL^{+}}$ (left node) and $\widehat{LL^{+}}$ (right node) are the log-likelihood for each node.

Once we have the estimates of $\mu$ and $\theta$, the predicted values of $Y$ under the ZAP_RF are obtained with

$$\widehat{Y} = \frac{\left(1 - \widehat{\theta}\right) \exp(-\widehat{\mu})}{(1 - \exp(-\widehat{\mu}))} \tag{15.3}$$

It is important to point out that in the prediction formula given above (15.3), ($\widehat{Y}$) is equal to the mean of the ZAP model, while under the ZAPC_RF, the predictions are obtained as

$$\widehat{Y} = \begin{cases} 0, & \widehat{\theta} > 0.5 \\ \widehat{\mu}, & \widehat{\theta} \leq 0.5 \end{cases} \tag{15.4}$$

The ZAPC_RF as conventional logistic regression, the predicted values are probabilities and those probabilities are converted to a binary outcome if the probability is larger (or smaller) than some probability threshold (most of the time this threshold is 0.5). However, under the ZAPC_RF, instead of converting the probabilities to 0 and 1, we convert to zero if $\widehat{\theta} > 0.5$ (15.4) and to the estimated expected count value ($\widehat{\mu}$) if $\widehat{\theta} \leq 0.5$ (15.4). One limitation of the ZAPC_RF shared with logistic regression is that the probability threshold is not unique, since many other values between zero and one can be used. However, the threshold value of 0.5 is used most of the time since it assumes no prior information, and for this reason, both categories have the same probability of occurring (Montesinos-López et al. 2021).

The implementation of the ZAP_RF and ZAPC_RF can be done using the following function:

```
tuning_model<-zap.rfsrc(X_training, y_training,
ntree_theta=ntree_1, mtry_theta=mtry_1, nodesize_theta=nodesize_1,
ntree_lambda=ntree_2, mtry_lambda=mtry_2,
nodesize_lambda=nodesize_2)
```

```
predictions <- predict(tuning_model, X_testing, type="original")$
predicted
```

It is important to point out that for the zap.rfsrc() function we need to provide a training set with predictors (X_training) and the count response variable (y_training). Also, like the rfsrc() function, we need to provide the ntree, mtry, and nodesize; however, we need to specify these parameters for the two parts (zero model and truncated model) of the ZAP_RF model and ZAPC_RF. The parameters that end with _theta are for the zero model and those that end with _lambda are for the truncated model. It is important to point out that the specification given above in the zap.rfsrc() is valid for the ZAP_RF and ZAPC_RF since the only difference between the two models is in how the predictions are performed. For this reason, only by changing in the predict() function the parameter type = "original" that implements the ZAP_RF to type = "custom", it is possible to implement the ZAPC_RF model.

Next, we show how to implement the ZAP_RF model using the same data set used in the previous examples. First, it is important to point out that all the input information used for implementing this example are the same as in the previous examples (the same design matrices, the same grid for the tuning process, etc.), except that now for the illustrating process, the response variable is assumed as count. But since we are using the same data set (Data_Toy_EYT.RData) as in the previous example, we assume trait DTHD as count. Next is given the basic R code for the tuning process:

```
tuning_model=zap.rfsrc(X_inner_training, y_inner_training,
ntree_theta=flags$ntree,                   mtry_theta=flags$mtry,
nodesize_theta=flags$nodesize, ntree_lambda=flags$ntree,
mtry_lambda=flags$mtry,                nodesize_lambda=flags$nodesize)
   predictions <- predict(tuning_model, X_inner_testing,
type="original")$predicted
```

The R code for refitting the model with the best hyperparameter combination is the same as above, with the only difference that instead of using the inner information, the whole training set (inner_training+tuning set) is used. The whole code for implementing the ZAP_RF is provided in Appendix 5.

Table 15.4 indicates that since the categorical trait (DTHD) was assumed as count (only for illustration purposes), the prediction performance was reported in terms of those metrics (MAAPE, PC, R2, MSE) used for continuous traits. Table 15.4 shows that in terms of MAAPE, PC, and R2, the best predictions were observed in environments Bed5IR and EHT, while in terms of MSE, the best predictions were observed in environments EHT and LHT.

Next, we obtain the predictions under the ZAPC_RF that only differ from the ZAP_RF model in specifying inside the predict() function in type = "custom" as is shown below.

```
predictions <- predict(tuning_model, X_inner_testing,
type="custom")$predicte
```

**Table 15.4** Prediction performance for the categorical trait (DTHD) assumed as a count response variable under the ZAP_RF model with five-fold cross-validation

| Env | MAAPE | SE_MAAPE | PC | SE_PC | R2 | SE_R2 | MSE | SE_MSE |
|---|---|---|---|---|---|---|---|---|
| Bed5IR | 0.3328 | 0.0527 | 0.3503 | 0.1150 | 0.1756 | 0.0716 | 0.7286 | 0.1614 |
| EHT | 0.2895 | 0.0403 | 0.2779 | 0.2712 | 0.3715 | 0.0984 | 0.7111 | 0.1585 |
| Flat5IR | 0.3196 | 0.0342 | 0.5332 | 0.0970 | 0.3196 | 0.1059 | 0.9044 | 0.1663 |
| LHT | 0.3332 | 0.0586 | 0.3700 | 0.1782 | 0.2639 | 0.1003 | 0.7222 | 0.1681 |

**Table 15.5** Prediction performance for the categorical trait (DTHD) assumed as a count response variable under the ZAPC_RF model with five-fold cross-validation

| Env | MAAPE | SE_MAAPE | PC | SE_PC | R2 | SE_R2 | MSE | SE_MSE |
|---|---|---|---|---|---|---|---|---|
| Bed5IR | 0.4518 | 0.0486 | 0.2464 | 0.1546 | 0.1563 | 0.078 | 1.138 | 0.1837 |
| EHT | 0.3599 | 0.023 | 0.356 | 0.165 | 0.2356 | 0.048 | 1.0579 | 0.2311 |
| Flat5IR | 0.361 | 0.0276 | 0.3302 | 0.1637 | 0.2162 | 0.083 | 1.1595 | 0.3545 |
| LHT | 0.3729 | 0.0443 | 0.3428 | 0.1802 | 0.2473 | 0.0353 | 0.9504 | 0.2104 |

For this reason, the code given in Appendix 5 can also be used for implementing the ZAPC_RF, but by only changing type = "original" to type = "custom". The prediction performance of the ZAPC_RF is given in Table 15.5. Under MAAPE, the best predictions were observed in the Bed5IR and LHT environments, while under APC, R2, and MSE, the best predictions were observed in environments EHT and LHT, respectively.

## 15.6 RF Algorithm for Multivariate Response Variables

Multivariate RF models are a generalization of univariate RF models, whereas a typical univariate RF model involves a data set where each instance has a single (continuous, binary, categorical, or count response) response value. The instances in a multivariate (multi-trait) RF problem have two or more response values—i.e., the output value is a vector rather than a scalar. There are two general approaches for solving multivariate RF problems: either by transforming the problem into multiple univariate problems or by adapting the RF algorithm so that it directly handles multivariate response variables simultaneously. The naive approach of transforming a multivariate problem into several univariate response problems is applicable for regression (continuous response variable) just as for classification (binary and categorical), and the same caveats apply.

Training any statistical machine learning model for predicting a continuous, binary, categorical, and count response variable is a time-consuming task—particularly so when training data sets are very large. When multiple models need to be trained using the same predictors in the data, but with different response variables, time consumption can quickly get out of hand. Thus, for very large problems, transforming a multivariate approach to univariate analysis may prove to be unsuitable. Using the algorithm adaptation approach, it is possible to directly create a model that simultaneously predicts a set of two or more continuous, binary, categorical, or count responses, or even mixed response variables (continuous, binary, and categorical) from a single training iteration. More importantly, when the prediction tasks are related (i.e., there is a correlation or covariance between response values), training a coherent multivariate model can potentially bring

benefits in the form of increased predictive performance compared to training multiple disjoint models (Evgeniou and Pontil 2004).

RFs have already been adapted to handle multivariate response variables for continuous, binary, categorical, and mixed (continuous, binary, and categorical) outcomes (Segal 1992; Larsen and Speckman 2004; Zhang 1998; De'Ath 2002; Faddoul et al. 2012; Segal and Xiao 2011; Glocker et al. 2012). For multivariate regression analysis, Tang and Ishwaran (2017) suggest using an averaged standardized variance splitting rule. Assuming that there are measures of $q$ traits in each observation, that is, $\boldsymbol{y}_i = (y_{i,1}, \ldots, y_{i,q})^{\mathrm{T}}$, the goal is to minimize the multivariate sums of squares (MSS),

$$\mathrm{MSS} = \sum_{j=1}^{q} \left( \sum\nolimits_{i=1}^{L} \left( y_{ij} - \bar{y}_{Lj} \right)^2 + \sum\nolimits_{i=1}^{R} \left( y_{ij} - \bar{y}_{Rj} \right)^2 \right), \tag{15.5}$$

where $\bar{y}_{Lj}$ and $\bar{y}_{Rj}$ are the sample means of the jth response variable in the left and right daughter nodes. Note that such a splitting rule (15.5) can only be effective if each of the response variables is measured on the same scale, otherwise we could have a response variable $j$ with, say, very large values, and its contribution would dominate MSS. We therefore calibrate MSS by assuming that each response variable has been standardized (with mean zero and variance equal to one). The standardization is applied prior to splitting a node. To make this standardization clear, we denote the standardized responses by $y_{ij}^*$ (Tang and Ishwaran 2017). With some elementary manipulations, it can be verified that minimizing MSS is equivalent to minimizing

$$\mathrm{MSS} = \sum_{j=1}^{q} \left( \frac{1}{n_L} \left( \sum\nolimits_{i=1}^{L} y_{ij}^* \right)^2 + \frac{1}{n_R} \left( \sum\nolimits_{i=1}^{R} y_{ij}^* \right)^2 \right) \tag{15.6}$$

For multivariate classification, an averaged standardized Gini splitting rule is used. According to Tang and Ishwaran (2017), the best split $s$ for X is obtained by maximizing

$$\mathrm{MGI} = \sum_{j=1}^{r} \left( \frac{1}{C_j} \sum_{k=1}^{C_j} \left( \frac{1}{n_L} \left( \sum\nolimits_{i=1}^{L} z_{i(k)j} \right)^2 + \frac{1}{n_R} \left( \sum\nolimits_{i=1}^{R} z_{i(k)j} \right)^2 \right) \right), \tag{15.7}$$

where $r$ denotes the number of categorical traits, the response variable $(y_{ij})$ is a class label from $\{1, \ldots, C_j\}$ for $C_j \geq 2$, and $z_{i(k)j} = 1_{\{y_{ij}=k\}}$. Note that the normalization $1/C_j$ employed in (15.7) for response variable $j$ is required to standardize the contribution of the Gini split from that response variable. Observe that (15.6) and (15.7) are equivalent optimization problems, with optimization over $y_{ij}$ for regression and $z_{i(k)j}$ for classification. This leads to similar theoretical splitting properties in regression and classification settings. Given this similarity, it is feasible to combine

the two splitting rules ((15.6) and (15.7)) to form a composite splitting rule. The mixed outcome splitting rule MCI is a composite standardized split rule of mean squared error (15.6) and Gini index splitting (15.7), i.e.,

$$\mathrm{MCI} = \mathrm{MSS} + \mathrm{MGI}, \tag{15.8}$$

where $p = q + r$. The best split for $X$ is the value of $s$ maximizing MCI (15.8). This multivariate normalized composite splitting function of mean squared error and Gini index splitting can be invoked with splitrule = "mv.mix" inside the function rfscr() for implementing multivariate RF for mixed outcomes (binary, categorical, and continuous).

There are other splitting functions for continuous and categorical response variables. For continuous multivariate responses, Segal (1992) proposed the multivariate Mahalanobis splitting rule:

$$\mathrm{SSE} = \sum\nolimits_{i=1}^{L}(\boldsymbol{y}_i - \bar{\boldsymbol{y}_L})\widehat{\boldsymbol{\Sigma}}^{-1}(\theta, t)(\boldsymbol{y}_i - \bar{\boldsymbol{y}_L}) + \sum\nolimits_{i=1}^{R}(\boldsymbol{y}_i - \bar{\boldsymbol{y}_R})\widehat{\boldsymbol{\Sigma}}^{-1}(\theta, t)(\boldsymbol{y}_i - \bar{\boldsymbol{y}_R}),$$

where $\widehat{\boldsymbol{\Sigma}}(\theta, t)$ is an estimate of the covariance matrix obtained from the parent node t, before the split, which may be modeled through a vector of parameters $\theta$ in order to impose a specific structure, for example, exchangeable or autoregressive. Even when no structure is imposed—that is, when the sample covariance matrix is used—$\widehat{\boldsymbol{\Sigma}}(\theta, t)$ is still computed on the parent node before the split. The Mahalanobis splitting rule allows incorporation of a correlation between response variables; however, it is not available in the rfscr() function.

When the response variable contains only binary or categorical variables, the splitting criterion is

$$\mathrm{CE} = \sum\nolimits_{k=1}^{S}\left(n_k^L n\left(\widehat{\pi}_k^L\right) + n_k^R n\left(\widehat{\pi}_k^R\right)\right)$$

under the assumption that there are $r$ categorical responses, and letting $s_j$ be the number of different values of the categorical outcome $f_j$, for $j = 1, \ldots, r$. The whole vector $(f_1, \ldots, f_r)$ can be described through a single variable $S$, taking $S = \prod_{j=1}^{r} s_j$ possible values or states, that is, each state corresponds to one unique combination of the original variables. This way, the original vector of categorical responses is cast into a single multinomial outcome with $S$ possible values that we assume to be 1, 2, . . . , $S$ without loss of generality. This criterion amounts to casting all individual responses into a single categorical outcome with $S$ values and then using the usual entropy criterion. This is in accordance with our goal to remain as free as possible from assumptions, but obviously, this solution becomes impractical when $S$ is too large. In such a case, dimension reduction is required.

Also, for implementing multivariate RF, we will be using the rfsrc() function but instead of specifying only one response variable (before ~), we need to specify at least two response variables inside the Multivar(GY, DTHD, DTMT, Height)

function or cbind(GY, DTHD, DTMT, Height) specification, where GY, DTHD, DTMT, Height denote each of the four response variables. The specification of the remaining parameters is the same. For example, below we show the basic code for performing the tuning process:

```
   tuning_model <- rfsrc(Multivar(GY, DTHD, DTMT, Height) ~ .,
data=DataInnerTraining, ntree=flags$ntree, mtry=flags$mtry,
nodesize=flags$nodesize)
   predictions <- predict(tuning_model, DataInnerTesting)
```

The complete R code for implementing the multivariate RF for continuous traits is given in Appendix 6. The mv.mse splitting rule is used by default for all continuous response variables under a multivariate framework. Table 15.6 shows that the best prediction performance in terms of MAAPE was observed in the GY trait; however, in terms of PC and R2, the best predictions were observed in trait DTMT. The predictions under the MSE are not comparable between traits because the traits are on different scales.

Next we provide the basic R code for implementing multivariate RF for categorical response variables.

```
model <- rfsrc(cbind(DTHD, DTMT, Height) ~ .,
        data=DataTraining, ntree=best_params$ntree,
        mtry=best_params$mtry, nodesize=best_params$nodesize
predicted <- predict(model, DataTesting)
```

We can see that there are no differences between the specification of a multivariate RF model for continuous and categorical response variables; however, when implementing the multivariate RF for categorical outcomes, all the response variables should be converted into factors. This is important because if this conversion is ignored, the RF will be implemented assuming that all response variables are continuous using the mv.mse splitting function. The multivariate Gini splitting rule (splitrule = "mv.gini") is used by default when all response variables are categorical. Only this splitting function is available in the rfsrc() for multivariate categorical outcomes.

Next, we give the results in terms of PCCC and the Kappa coefficient for one binary trait (Height) and two categorical traits (DTHD and DTMT) (Table 15.7). Across environments, the best predictions were observed for trait DTHD (PCCC = 0.7348 and Kappa = 0.5799) and the worst for the Height trait (PCCC = 0.6988 and Kappa = 0.3959). For trait DTHD, the best predictions were observed in the DHT environment, while for trait DTMT, the best predictions were observed in environment Bed5IR; finally, for trait Height, the best predictions were observed in environment EHT.

Finally, the specification in function rfsrc() of the multivariate RF model with mixed outcomes (continuous, binary, and categorical) is equal to the specification given above for all continuous outcomes or all categorical outcomes, but you need to put as factors those categorical response variables and as numeric those continuous

**Table 15.6** Prediction performance under multivariate RF assuming all traits are continuous with five-fold cross-validation

| Env | Trait | MAAPE | SE_MAAPE | PC | SE_PC | R2 | SE_R2 | MSE | SE_MSE |
|---|---|---|---|---|---|---|---|---|---|
| Bed5IR | GY | 0.0552 | 0.0054 | 0.3816 | 0.1730 | 0.2654 | 0.1159 | 0.1544 | 0.0223 |
| EHT | GY | 0.0696 | 0.0045 | 0.5570 | 0.1411 | 0.3900 | 0.1600 | 0.3257 | 0.0548 |
| Flat5IR | GY | 0.0826 | 0.0105 | 0.4762 | 0.1356 | 0.3003 | 0.1354 | 0.4528 | 0.0882 |
| LHT | GY | 0.1182 | 0.0198 | 0.2989 | 0.2116 | 0.2685 | 0.1350 | 0.1864 | 0.0452 |
| Bed5IR | DTHD | 0.3491 | 0.0451 | 0.6757 | 0.0448 | 0.4646 | 0.0619 | 0.5074 | 0.1008 |
| EHT | DTHD | 0.2862 | 0.0158 | 0.7457 | 0.0338 | 0.5606 | 0.0491 | 0.4571 | 0.0499 |
| Flat5IR | DTHD | 0.3041 | 0.0470 | 0.6552 | 0.1854 | 0.5667 | 0.1569 | 0.4145 | 0.1183 |
| LHT | DTHD | 0.4290 | 0.0538 | 0.4422 | 0.1769 | 0.3207 | 0.1573 | 0.7926 | 0.1456 |
| Bed5IR | DTMT | 0.3000 | 0.0561 | 0.7519 | 0.0358 | 0.5704 | 0.0515 | 0.4197 | 0.1294 |
| EHT | DTMT | 0.2583 | 0.0189 | 0.7589 | 0.0454 | 0.5842 | 0.0673 | 0.3382 | 0.0494 |
| Flat5IR | DTMT | 0.2469 | 0.0352 | 0.7238 | 0.0966 | 0.5611 | 0.1217 | 0.3005 | 0.0889 |
| LHT | DTMT | 0.3889 | 0.0611 | 0.4973 | 0.0995 | 0.2869 | 0.1013 | 0.6722 | 0.0967 |
| Bed5IR | Height | 0.9263 | 0.1213 | 0.6199 | 0.0930 | 0.4189 | 0.0977 | 0.1694 | 0.0151 |
| EHT | Height | 0.7842 | 0.0771 | 0.6243 | 0.1354 | 0.4630 | 0.1297 | 0.1804 | 0.0369 |
| Flat5IR | Height | 1.1624 | 0.0326 | 0.4456 | 0.1230 | 0.2591 | 0.1138 | 0.1924 | 0.0234 |
| LHT | Height | 1.0032 | 0.0785 | −0.1675 | 0.2371 | 0.2529 | 0.1632 | 0.2795 | 0.0231 |

**Table 15.7** Prediction performance under multivariate RF assuming all traits are categorical with five-fold cross-validation

| Env | Trait | PCCC | SE_PCCC | Kappa | SE_Kappa |
|---|---|---|---|---|---|
| Bed5IR | DTHD | 0.6929 | 0.1381 | 0.5888 | 0.156 |
| EHT | DTHD | 0.7839 | 0.0516 | 0.6052 | 0.0943 |
| Flat5IR | DTHD | 0.7486 | 0.0734 | 0.5697 | 0.1184 |
| LHT | DTHD | 0.7138 | 0.092 | 0.556 | 0.1349 |
| Bed5IR | DTMT | 0.7429 | 0.0863 | 0.6133 | 0.1203 |
| EHT | DTMT | 0.7367 | 0.0493 | 0.5679 | 0.0656 |
| Flat5IR | DTMT | 0.662 | 0.0409 | 0.4342 | 0.0494 |
| LHT | DTMT | 0.5968 | 0.0822 | 0.3876 | 0.1019 |
| Bed5IR | Height | 0.6857 | 0.1126 | 0.3743 | 0.2244 |
| EHT | Height | 0.7911 | 0.1039 | 0.5515 | 0.2386 |
| Flat5IR | Height | 0.7024 | 0.1107 | 0.3323 | 0.2521 |
| LHT | Height | 0.6161 | 0.0864 | 0.3258 | 0.1287 |

outcomes; in this way, the splitting function to be used to perform the training process will be the splitrule = "mv.mix" and the analysis should be performed in the right way. In Table 15.8, we can see that for the continuous trait GY, the best predictions were observed in the Bed5IR environment for all traits. For the binary and categorical traits, the best predictions across environments were observed in trait Height under both metrics.

## 15.7 Final Comments

Part of the power of RF is due to the fact that RF introduces two kinds of randomization. First, a bootstrap sample randomly drawn from the training data is used to grow a tree. Second, at each node of the tree, a randomly selected subset of variables (covariates) is chosen as candidate variables for splitting. This means that rather than splitting a tree node using all $p$ variables (features), RF selects, at each node of each tree, a random subset of $1 \leq mtry \leq p$ variables that is used to split the node where typically $mtry$ is substantially smaller than $p$. The purpose of this two-step randomization is to decorrelate trees and reduce variance. RF trees are grown deeply, which reduces bias. Averaging across trees, in combination with the subsampling process used in growing a tree, enables RF to approximate rich types of functions while maintaining low generalization error. Considerable empirical evidence has shown RF to be highly accurate, comparable to state-of-the-art methods such as bagging [Breiman 1996], boosting [Schapire et al. 1998], and support vector machines [Cortes and Vapnik 1995].

As a tree-based ensemble statistical machine learning tool that is highly data adaptive, RF can be applied successfully to "large $p$, small $n$" problems, and it is also able to capture correlation as well as interactions among independent variables

**Table 15.8** Prediction performance under multivariate RF with mixed traits (Height = binary, DTHD and DTMT = categorical, and GY = continuous) with five-fold cross-validation

| Env | Trait | MAAPE | SE_MAAPE | PC | SE_PC | R2 | SE_R2 | MSE | SE_MSE |
|---|---|---|---|---|---|---|---|---|---|
| Bed5IR | GY | 0.049 | 0.009 | 0.607 | 0.145 | 0.453 | 0.149 | 0.139 | 0.042 |
| EHT | GY | 0.078 | 0.010 | 0.033 | 0.170 | 0.117 | 0.072 | 0.415 | 0.119 |
| Flat5IR | GY | 0.080 | 0.010 | 0.650 | 0.063 | 0.439 | 0.075 | 0.371 | 0.082 |
| LHT | GY | 0.121 | 0.021 | 0.280 | 0.215 | 0.263 | 0.139 | 0.182 | 0.040 |
| Env | Trait | PCCC | SE_PCCC | Kappa | SE_Kappa | | | | |
| Bed5IR | DTHD | 0.545 | 0.101 | 0.260 | 0.141 | | | | |
| EHT | DTHD | 0.657 | 0.067 | 0.079 | 0.146 | | | | |
| Flat5IR | DTHD | 0.665 | 0.059 | 0.476 | 0.082 | | | | |
| LHT | DTHD | 0.425 | 0.074 | 0.056 | 0.056 | | | | |
| Bed5IR | DTMT | 0.673 | 0.118 | 0.296 | 0.267 | | | | |
| EHT | DTMT | 0.690 | 0.041 | 0.286 | 0.113 | | | | |
| Flat5IR | DTMT | 0.655 | 0.095 | 0.440 | 0.152 | | | | |
| LHT | DTMT | 0.294 | 0.055 | 0.085 | 0.153 | | | | |
| Bed5IR | Height | 0.712 | 0.071 | 0.417 | 0.144 | | | | |
| EHT | Height | 0.677 | 0.090 | 0.277 | 0.118 | | | | |
| Flat5IR | Height | 0.798 | 0.075 | 0.575 | 0.112 | | | | |
| LHT | Height | 0.493 | 0.083 | 0.134 | 0.068 | | | | |

(Chen and Ishwaran 2012). RF is becoming increasingly used in genomic selection because, unlike traditional methods, it can efficiently analyze thousands of loci simultaneously and account for nonadditive interactions. For these reasons, RF is very appealing for high-dimensional genomic data analysis. In this chapter, we provide the motivation, fundamentals, and some applications of RF for genomic prediction with genomic data with continuous, binary, ordinal, and count response variables that are very often found in genomic selection.

Another advantage of RF over alternative machine learning methods is variable importance measures, which can be used to identify relevant independent variables (input) or perform variable selection. In general, RF provides a very powerful algorithm that often has great predictive accuracy and has become one of the benchmarks in the predictive field due to the good results it generates in very diverse problems. RF comes with all the benefits of decision trees (with the exception of surrogate splits) and bagging, but greatly reduces instability and between-tree correlation. And due to the added split variable selection attribute, RF models are also faster than bagging (not explained in this book) as they have a smaller feature search space at each tree split. However, RF will still suffer from slow computational speed as the data sets get larger but, similar to bagging, the algorithm is built upon independent steps, and most modern implementations allow for parallelization to improve training time.

## Appendix 1

Metrics for computing prediction performance and variable important plots.

```
library(dplyr)
library(caret)

# Mean Square Error of prediction
mse <- function(actual, predicted) {
 return(mean((actual - predicted)^2, na.rm=TRUE))
}

# Proportion of cases correctly classified
pccc <- function(actual, predicted) mean(actual == predicted, na.
rm=TRUE)

# Mean arctangent absolute percentage error
maape <- function(actual, predicted) {
 return(mean(atan(abs(actual - predicted) / abs(actual)),
     na.rm=TRUE))
}

# Kappa coefficient
kappa <- function(actual, predicted) {
```

```
 confusion_matrix <- confusionMatrix(table(actual, predicted))
 return(confusion_matrix$overall[2])
}

# Generates folds for K-fold cross-validation.
# From a data set with "n_records" returns a list with "k"
# lists containing the elements to be used as training and testing in each
fold.
CV.Kfold <- function(n_records, k=5) {
 folds_vector <- findInterval(cut(sample(n_records, n_records),
                 breaks=k), 1:n_records)

 folds <- list()

 for (fold_num in 1:k) {
  current_fold <- list()
  current_fold$testing <- which(folds_vector == fold_num)
  current_fold$training <- setdiff(1:n_records,
current_fold$testing)

  folds[[fold_num]] <- current_fold
 }
 return(folds)
}

# Generates folds for random cross-validation.
# From a data set with "n_records" it is returned a list with "n_folds"
# lists containing "testing_proportion" elements records for testing
and the
# complement for training.
CV.Random <- function(n_records, n_folds=10, testing_proportion=0.2)
{
 folds <- list()

 for (fold_num in 1:n_folds) {
  current_fold <- list()
  current_fold$testing <- sample(n_records, n_records *
testing_proportion)
  current_fold$training <- setdiff(1:n_records,
current_fold$testing)
  folds[[fold_num]] <- current_fold
 }
 return(folds)
}
apply_white_theme <- function(Plot) {
 Plot <- Plot + theme(axis.text=element_text(size=14),
          axis.title=element_text(size=14, face="bold")) +
        theme_bw() +
        theme(text=element_text(size=18))
```

```
 return(Plot)
}

plot_importances <- function(importances, how_many=10,
color="#248a8a") {
 importances <- importances[!is.na(importances)]
 importances <- importances[importances > 0]
 n_indices <- min(length(importances), how_many)
 indices <- order(importances, decreasing = TRUE)[1:n_indices]
 best_importances <- importances[indices]

 X_names <- names(best_importances)
 if (is.null(X_names)) {
  X_names <- 1:length(best_importances)
 }

 X_names <- factor(X_names, levels=X_names)
 PlotData <- data.frame(value=best_importances, variable=X_names)

 Plot <- ggplot(PlotData, aes(x=variable, y=value)) +
     geom_bar(stat="identity", color=color, fill=color) +
     coord_flip()
 Plot <- apply_white_theme(Plot)

 return(Plot)
}

save_plot <- function(Plot, file="plot.png", width=700, height=450,
res=110) {
 png(file=file, width=width, height=height, res=res)
 print(Plot)
 dev.off()
}
```

## Appendix 2

R code for implementing RF for continuous response variables and how to extract variable importance predictors.

```
# Remove all variables from our workspace
rm(list=ls(all=TRUE))
library(randomForestSRC)
library(dplyr)
library(ggplot2)

# Import the own function for plotting variable importances
source("utils.R")
```

```
# Import the data set
load("Data_Toy_EYT.RData", verbose=TRUE)
Pheno <- Pheno_Toy_EYT
Pheno$Env <- as.factor(Pheno$Env)
Geno <- G_Toy_EYT

# Sorting data
Pheno <- Pheno[order(Pheno$Env, Pheno$GID), ]
geno_sort_lines <- sort(rownames(Geno))
Geno <- Geno[geno_sort_lines, geno_sort_lines]

### Design matrices definition ###
ZG <- model.matrix(~0 + GID, data=Pheno)

# Compute the Choleski factorization
ZL <- chol(Geno)
ZGL <- ZG %*% ZL
ZE <- model.matrix(~0 + Env, data=Pheno)

# Bind all design matrices in a single matrix to be used as predictor
X <- cbind(ZE,ZGL)
dim(X)

# Create a data frame with the information of response variable and all
# predictors
Data <- data.frame(y=Pheno$GY, X)
head(Data[, 1:5])

# Fit the model with importance=TRUE for also computing the variable
importance
model <- rfsrc(y ~ ., data=Data, importance=TRUE)

# Get the variable importance
head(model$importance,10)

# Plot the 30 most important variables with own function
Plot <- plot_importances(model$importance, how_many=10)
Plot
save_plot(Plot, "plots/1.continuous.png")
```

# Appendix 3

R code for implementing RF for continuous response variables with ten random partitions.

```
# Remove all variables from our workspace
rm(list=ls(all=TRUE))
library(randomForestSRC)
```

```
library(dplyr)
library(caret)
library(purrr)

# Import some useful functions such as CV.Random, CV.Kfold, mse, maape,
etc.
source("utils.R")

# Import the data set
load("Data_Toy_EYT.RData", verbose=TRUE)
Pheno <- Pheno_Toy_EYT
Geno <- G_Toy_EYT

########## PREPARE DATA ##########
Pheno$Env <- as.factor(Pheno$Env)
Pheno$Height <- as.factor(Pheno$Height)

# Sorting data
Pheno <- Pheno[order(Pheno$Env, Pheno$GID), ]
geno_sort_lines <- sort(rownames(Geno))
Geno <- Geno[geno_sort_lines, geno_sort_lines]

### Design matrices definition ###
ZG <- model.matrix(~0 + GID, data=Pheno)
### Compute the Choleski factorization
ZL<-chol(Geno)
ZGL<-ZG %*% ZL

ZE<-model.matrix(~0 + Env, data=Pheno)
#Interaction design matrix
ZGE <- model.matrix(~0 + ZGL:Env, data=Pheno)

###Joining the three design matrices
X <- cbind(ZGL, ZE, ZGE)
dim(X)

# Create a data frame with the information of response variable and all
# predictors
Data <- data.frame(y=Pheno$GY, X)
head(Data[, 1:5])

n_records <- nrow(Pheno)
n_outer_folds <- 10
outer_testing_proportion <- 0.2
n_inner_folds <- 1
inner_testing_proportion <- 0.2

# Get the indices of the elements that are going to be used as training and
# testing in each fold
outer_folds <- CV.Random(n_records, n_folds=n_outer_folds,
            testing_proportion=outer_testing_proportion)
```

```
# Grid values for the tuning process
tuning_values <- list(ntrees=c(100, 200, 300),
          mtry=c(80, 100, 120),
          nodesize=c(3, 6, 9))

# Get all possible combinations of the defined tuning values - (3 * 3 * 3)
all_combinations <- cross(tuning_values)
n_combinations <- length(all_combinations)

########## RANDOM FOREST TUNING AND EVALUATION ##########
# Define the variable where the final results of each fold will be stored in
Predictions <- data.frame()

# Iterate over each generated fold
for (i in 1:n_outer_folds) {
 cat("Outer Fold:", i, "/", n_outer_folds, "\n")
 outer_fold <- outer_folds[[i]]

 # Divide our data into testing and training sets
 DataTraining <- Data[outer_fold$training, ]
 DataTesting <- Data[outer_fold$testing, ]

 ### Tuning only with training data ###
 n_tuning_records <- nrow(DataTraining)
 # Variable that will hold the best combination of hyperparameters and
the MSE
 # that was produced.
 best_params <- list(mse=Inf)

 inner_folds <- CV.Random(n_tuning_records, n_folds=n_inner_folds,
            testing_proportion=inner_testing_proportion)

 for (j in 1:n_combinations) {
  cat("\tCombination:", j, "/", n_combinations, "\n")

  flags <- all_combinations[[j]]

  cat("\t\tInner folds: ")
  for (m in 1:n_inner_folds) {
   cat(m, ", ")
   inner_fold <- inner_folds[[m]]

   DataInnerTraining <- DataTraining[inner_fold$training, ]
   DataInnerTesting <- DataTraining[inner_fold$testing, ]

   # Fit the model using the current combination of hyperparameters
   tuning_model <- rfsrc(y ~ ., data=DataInnerTraining,
ntree=flags$ntree,
          mtry=flags$mtry, nodesize=flags$nodesize, splitrule="mse")
   predictions <- predict(tuning_model, newdata=DataInnerTesting)$
predicted
```

```
    # Compute MSE for the current combination of hyperparameters
    current_mse <- mse(DataInnerTesting$y, predictions)

   # If the current combination gives a lower MSE set it as new best_params
    if (current_mse < best_params$mse) {
     best_params <- flags
     best_params$mse <- current_mse
    }
   }
  cat("\n")
 }

 # Using the best params combination retrain the model but using the
complete
 # training set
 model <- rfsrc(y ~ ., data=DataTraining, ntree=best_params$ntree,
         mtry=best_params$mtry, nodesize=best_params$nodesize,
splitrule="mse")
 predicted <- predict(model, newdata=DataTesting)$predicted

 # Save the information of the predictions in the current fold
 CurrentPredictions <- data.frame(Position=outer_fold$testing,
                   GID=Pheno$GID[outer_fold$testing],
                   Env=Pheno$Env[outer_fold$testing],
                   Partition=i,
                   Observed=DataTesting$y,
                   Predicted=predicted)
 Predictions <- rbind(Predictions, CurrentPredictions)
}

head(Predictions)
tail(Predictions)

# Summarize the results across environment computing four metrics
ByEnvSummary <- Predictions %>%
        # Calculate the metrics disaggregated by Partition and Env
        group_by(Partition, Env) %>%
        summarise(MSE=mse(Observed, Predicted),
             Cor=cor(Predicted, Observed, use="na.or.complete"),
             R2=cor(Predicted, Observed, use="na.or.complete")^2,
             MAAPE=maape(Observed, Predicted)) %>%
        select_all() %>%

        # Calculate the metrics disaggregated Env with standard errors
        # of each partition
        group_by(Env) %>%
        summarise(SE_MAAPE=sd(MAAPE, na.rm=TRUE) / sqrt(n()),
             MAAPE=mean(MAAPE, na.rm=TRUE),
             SE_Cor=sd(Cor, na.rm=TRUE) / sqrt(n()),
             Cor=mean(Cor, na.rm=TRUE),
             SE_R2=sd(R2, na.rm=TRUE) / sqrt(n()),
             R2=mean(R2, na.rm=TRUE),
```

```
            SE_MSE=sd(MSE, na.rm=TRUE) / sqrt(n()),
            MSE=mean(MSE, na.rm=TRUE)) %>%
       select_all() %>%

       mutate_if(is.numeric, ~round(., 4)) %>%
       as.data.frame()
ByEnvSummary

write.csv(Predictions, file="results/2.GY_random_all.csv", row.
names=FALSE)
write.csv(ByEnvSummary, file="results/2.GY_random_summary.csv", row.
names=FALSE)
```

## Appendix 4

R code for implementing RF for binary response variables with five-fold cross-validation.

```
# Remove all variables from our workspace
rm(list=ls(all=TRUE))
library(randomForestSRC)
library(dplyr)
library(caret)
library(purrr)

# Import some useful functions such as CV.Random, CV.Kfold, mse, maape,
etc.
source("utils.R")

# Import the data set
load("Data_Toy_EYT.RData", verbose=TRUE)
Pheno <- Pheno_Toy_EYT
Geno <- G_Toy_EYT

########## PREPARE DATA ##########
Pheno$Env <- as.factor(Pheno$Env)
Pheno$Height <- as.factor(Pheno$Height)

# Sorting data
Pheno <- Pheno[order(Pheno$Env, Pheno$GID), ]
geno_sort_lines <- sort(rownames(Geno))
Geno <- Geno[geno_sort_lines, geno_sort_lines]

### Design matrices definition ###
ZG <- model.matrix(~0 + GID, data=Pheno)
# Compute the Choleski factorization
ZL <- chol(Geno)
ZGL <- ZG %*% ZL
```

```
ZE <- model.matrix(~0 + Env, data=Pheno)
# Interaction design matrix
ZGE <- model.matrix(~0 + ZGL:Env, data=Pheno)

# Bind all design matrices in a single matrix to be used as predictor
X <- cbind(ZGL, ZE, ZGE)
dim(X)

# Create a data frame with the information of response variable and all
# predictors. As Height is binary response variable that is already
condired as factor variable automatically it will be trained a classifier
# random forest.
Data <- data.frame(y=Pheno$Height, X)
head(Data[, 1:5])

n_records <- nrow(Pheno)
n_outer_folds <- 5
n_inner_folds <- 5

# Get the indices of the elements that are going to be used as training and
# testing in each fold
outer_folds <- CV.Kfold(n_records, k=n_outer_folds)

# Define the values which are going to be evaluated in the tuning process
tuning_values <- list(ntrees=c(100, 200, 300),
          mtry=c(80, 100, 120),
          nodesize=c(3, 6, 9))

# Get all possible combinations of the defined tuning values - (3 * 3 * 3)
all_combinations <- cross(tuning_values)
n_combinations <- length(all_combinations)

########## RANDOM FOREST TUNING AND EVALUATION ##########
# Define the variable where the final results of each fold will be stored
Predictions <- data.frame()

# Iterate over each generated fold
for (i in 1:n_outer_folds) {
 cat("Outer Fold:", i, "/", n_outer_folds, "\n")
 outer_fold <- outer_folds[[i]]

 # Divide our data into testing and training sets
 DataTraining <- Data[outer_fold$training, ]
 DataTesting <- Data[outer_fold$testing, ]

 ### Tuning only with training data ###
 n_tuning_records <- nrow(DataTraining)
 # Variable that will hold the best combination of hyperparameters and
the PCCC
 # that was produced.
 best_params <- list(pccc=-Inf)
```

```
 inner_folds <- CV.Kfold(n_tuning_records, k=n_inner_folds)

 for (j in 1:n_combinations) {
  cat("\tCombination:", j, "/", n_combinations, "\n")

  flags <- all_combinations[[j]]

  cat("\t\tInner folds: ")
  for (m in 1:n_inner_folds) {
   cat(m, ", ")
   inner_fold <- inner_folds[[m]]

   DataInnerTraining <- DataTraining[inner_fold$training, ]
   DataInnerTesting <- DataTraining[inner_fold$testing, ]

   # Fit the model using the current combination of hyperparameters
   tuning_model <- rfsrc(y ~ ., data=DataInnerTraining,
ntree=flags$ntree,
              mtry=flags$mtry, nodesize=flags$nodesize)
   predictions <- predict(tuning_model, newdata=DataInnerTesting)$
class

   # Compute PCCC for the current combination of hyperparameters
   current_pccc <- pccc(DataInnerTesting$y, predictions)

   # If the current combination gives a greater PCCC, set it as new
best_params
   if (current_pccc > best_params$pccc) {
    best_params <- flags
    best_params$pccc <- current_pccc
   }
  }
  cat("\n")
 }

 # Using the best params combination, retrain the model but using the
complete
 # training set
 model <- rfsrc(y ~ ., data=DataTraining, ntree=best_params$ntree,
        mtry=best_params$mtry, nodesize=best_params$nodesize)
 predicted <- predict(model, newdata=DataTesting)$class

 # Save the information of the predictions in the current fold
 CurrentPredictions <- data.frame(Position=outer_fold$testing,
                  GID=Pheno$GID[outer_fold$testing],
                  Env=Pheno$Env[outer_fold$testing],
                  Partition=i,
                  Observed=DataTesting$y,
                  Predicted=predicted)
 Predictions <- rbind(Predictions, CurrentPredictions)
}
```

```
head(Predictions)
tail(Predictions)

# Summarize the results across environment computing two metrics
ByEnvSummary <- Predictions %>%
        # Calculate the metrics disaggregated by Partition and Env
        group_by(Partition, Env) %>%
        summarise(PCCC=pccc(Observed, Predicted),
             Kappa=kappa(Observed, Predicted)) %>%
        select_all() %>%

        # Calculate the metrics disaggregated Env with standard errors
        # of each partition
        group_by(Env) %>%
        summarize(SE_PCCC=sd(PCCC, na.rm=TRUE) / sqrt(n()),
             PCCC=mean(PCCC, na.rm=TRUE),
             SE_Kappa=sd(Kappa, na.rm=TRUE) / sqrt(n()),
             Kappa=mean(Kappa, na.rm=TRUE)) %>%
        select_all() %>%
        mutate_if(is.numeric, ~round(., 4)) %>%
        as.data.frame()
ByEnvSummary

write.csv(Predictions, file="results/3.Height_k_fold_all.csv", row.
names=FALSE)
write.csv(ByEnvSummary, file="results/3.Height_k_fold_summary.csv",
row.names=FALSE)
```

## Appendix 5

R code for implementing RF for count response variables with five-fold cross-validation.

```
# Remove all variables from our workspace
rm(list=ls(all=TRUE))

# Install the needed version of randomForestSRC library from this Github
repo
# that contains the zap.rfsrc function if not installed or if you have
another
# version of randomForestSRC
# devtools::install_github("brandon-mosqueda/randomForestSRC")
library(randomForestSRC)
library(dplyr)
library(caret)
library(purrr)
```

```
# Import some useful functions such as CV.Random, CV.Kfold, mse, maape,
etc.
source("utils.R")

# Import the data set
load("Data_Toy_EYT.RData", verbose=TRUE)
Pheno <- Pheno_Toy_EYT
Pheno$Env <- as.factor(Pheno$Env)
Geno <- G_Toy_EYT

# Sorting data
Pheno <- Pheno[order(Pheno$Env, Pheno$GID), ]
geno_sort_lines <- sort(rownames(Geno))
Geno <- Geno[geno_sort_lines, geno_sort_lines]

### Design matrices definition ###
ZG <- model.matrix(~0 + GID, data=Pheno)
# Compute the Choleski factorization
ZL <- chol(Geno)
ZGL <- ZG %*% ZL

ZE <- model.matrix(~0 + Env, data=Pheno)
# Interaction design matrix
ZGE <- model.matrix(~0 + ZGL:Env, data=Pheno)

# Bind all design matrices in a single matrix to be used as predictor
X <- data.frame(cbind(ZGL, ZE, ZGE))
dim(X)

# Response variable
y <- Pheno$DTHD

n_records <- nrow(Pheno)
n_outer_folds <- 5
n_inner_folds <- 5

# Get the indices of the elements that are going to be used as training and
# testing in each fold
outer_folds <- CV.Kfold(n_records, k=n_outer_folds)

# Define the values that are going to be evaluated in the tuning process
tuning_values <- list(ntrees=c(100, 200, 300),
          mtry=c(80, 100, 120),
          nodesize=c(3, 6, 9))

# Get all possible combinations of the defined tuning values - (3 * 3 * 3)
all_combinations <- cross(tuning_values)
n_combinations <- length(all_combinations)
```

```
########## RANDOM FOREST TUNING AND EVALUATION ##########
# Define the variable where the final results of each fold will be stored
Predictions <- data.frame()

# Iterate over each generated fold
for (i in 1:n_outer_folds) {
 cat("Outer Fold:", i, "/", n_outer_folds, "\n")
 outer_fold <- outer_folds[[i]]

 # Divide our data into testing and training sets
 X_training <- X[outer_fold$training, ]
 y_training <- y[outer_fold$training]

 X_testing <- X[outer_fold$testing, ]
 y_testing <- y[outer_fold$testing]

 ### Tuning only with training data ###
 n_tuning_records <- nrow(X_training)
 # Variable that will hold the best combination of hyperparameters and
the MSE
 # that was produced.
 best_params <- list(mse=Inf)

 inner_folds <- CV.Kfold(n_tuning_records, k=n_inner_folds)

 for (j in 1:n_combinations) {
  cat("\tCombination:", j, "/", n_combinations, "\n")

  flags <- all_combinations[[j]]

  cat("\t\tInner folds: ")
  for (m in 1:n_inner_folds) {
   cat(m, ", ")
   inner_fold <- inner_folds[[m]]

   X_inner_training <- X_training[inner_fold$training, ]
   y_inner_training <- y_training[inner_fold$training]

   X_inner_testing <- X_training[inner_fold$testing, ]
   y_inner_testing <- y_training[inner_fold$testing]

   # Fit the model using the current combination of hyperparameters
   tuning_model <- zap.rfsrc(X_inner_training, y_inner_training,
              ntree_theta=flags$ntree,
              mtry_theta=flags$mtry,
              nodesize_theta=flags$nodesize,
              ntree_lambda=flags$ntree,
              mtry_lambda=flags$mtry,
              nodesize_lambda=flags$nodesize)
   # You can also use custom as prediction type
   predictions <- predict(tuning_model, X_inner_testing)$predicted
```

```
    # Compute MSE for the current combination of hyperparameters
    current_mse <- mse(y_inner_testing, predictions)

    # If the current combination gives a lower MSE, set it as new
best_params
    if (current_mse < best_params$mse) {
     best_params <- flags
     best_params$mse <- current_mse
    }
   }
   cat("\n")
  }

  # Using the best params combination, retrain the model but using the
complete
  # training set
  model <- zap.rfsrc(X_training, y_training,
            ntree_theta=best_params$ntree,
            mtry_theta=best_params$mtry,
            nodesize_theta=best_params$nodesize,
            ntree_lambda=best_params$ntree,
            mtry_lambda=best_params$mtry,
            nodesize_lambda=best_params$nodesize)
  # You can also use custom as prediction type
  predicted <- predict(model, X_testing, type="original")$predicted

  # Save the information of the predictions in the current fold
  CurrentPredictions <- data.frame(Position=outer_fold$testing,
                   GID=Pheno$GID[outer_fold$testing],
                   Env=Pheno$Env[outer_fold$testing],
                   Partition=i,
                   Observed=y_testing,
                   Predicted=predicted)
  Predictions <- rbind(Predictions, CurrentPredictions)
}

head(Predictions)
tail(Predictions)

# Summarize the results across environment computing four metrics
ByEnvSummary <- Predictions %>%
        # Calculate the metrics disaggregated by Partition and Env
        group_by(Partition, Env) %>%
        summarise(MSE=mse(Observed, Predicted),
             Cor=cor(Predicted, Observed, use="na.or.complete"),
             R2=cor(Predicted, Observed, use="na.or.complete")^2,
             MAAPE=maape(Observed, Predicted)) %>%
        select_all() %>%

        # Calculate the metrics disaggregated Env with standard errors
        # of each partition
        group_by(Env) %>%
```

```
        summarise(SE_MAAPE=sd(MAAPE, na.rm=TRUE) / sqrt(n()),
             MAAPE=mean(MAAPE, na.rm=TRUE),
             SE_Cor=sd(Cor, na.rm=TRUE) / sqrt(n()),
             Cor=mean(Cor, na.rm=TRUE),
             SE_R2=sd(R2, na.rm=TRUE) / sqrt(n()),
             R2=mean(R2, na.rm=TRUE),
             SE_MSE=sd(MSE, na.rm=TRUE) / sqrt(n()),
             MSE=mean(MSE, na.rm=TRUE)) %>%
        select_all() %>%

        mutate_if(is.numeric, ~round(., 4)) %>%
        as.data.frame()
ByEnvSummary

write.csv(Predictions, file="results/7.DTHD_k_fold_all.csv", row.
names=FALSE)
write.csv(ByEnvSummary, file="results/7.DTHD_k_fold_summary.csv",
row.names=FALSE)
```

## Appendix 6

R code for implementing RF for multivariate continuous response variables with five-fold cross-validation.

```
# Remove all variables from our workspace
rm(list=ls(all=TRUE))
library(randomForestSRC)
library(dplyr)
library(caret)
library(purrr)

# Import some useful functions such as CV.Random, CV.Kfold, mse, maape,
etc.
source("utils.R")

# Import the data set
load("Data_Toy_EYT.RData", verbose=TRUE)
Pheno <- Pheno_Toy_EYT
Pheno$Env <- as.factor(Pheno$Env)

# Verify all variables are numeric
str(Pheno)
Geno <- G_Toy_EYT

# Sorting data
Pheno <- Pheno[order(Pheno$Env, Pheno$GID), ]
geno_sorted_lines <- sort(rownames(Geno))
Geno <- Geno[geno_sorted_lines, geno_sorted_lines]
```

```
### Design matrices definition ###
ZG <- model.matrix(~0 + GID, data=Pheno)
# Compute the Choleski factorization
ZL <- chol(Geno)
ZGL <- ZG %*% ZL

ZE <- model.matrix(~0 + Env, data=Pheno)
# Interaction design matrix
ZGE <- model.matrix(~0 + ZGL:Env, data=Pheno)

# Bind all design matrices in a single matrix to be used as predictor
X <- cbind(ZGL, ZE, ZGE)
dim(X)

# Create a data frame with the information of the four response variables
and all
# predictors
Data <- data.frame(GY=Pheno$GY, DTHD=Pheno$DTHD,
         DTMT=Pheno$DTMT, Height=Pheno$Height, X)
head(Data[, 1:8])

responses <- c("GY", "DTHD", "DTMT", "Height")
n_records <- nrow(Pheno)
n_outer_folds <- 5
n_inner_folds <- 5

# Get the indices of the elements that are going to be used as training and
# testing in each fold
outer_folds <- CV.Kfold(n_records, k=n_outer_folds)

# Define the values which are going to be evaluated in the tuning process
tuning_values <- list(ntrees=c(100, 200, 300),
          mtry=c(80, 100, 120),
          nodesize=c(3, 6, 9))

# Get all possible combinations of the defined tuning values (3 * 3 * 3)
all_combinations <- cross(tuning_values)
n_combinations <- length(all_combinations)

########## RANDOM FOREST TUNING AND EVALUATION ##########
# Define the variable where the final results of each fold will be stored
Predictions <- data.frame()

# Iterate over each generated fold
for (i in 1:n_outer_folds) {
 cat("Outer Fold:", i, "/", n_outer_folds, "\n")
 outer_fold <- outer_folds[[i]]

 # Divide our data into testing and training sets
 DataTraining <- Data[outer_fold$training, ]
 DataTesting <- Data[outer_fold$testing, ]
```

```
 ### Tuning only with training data ###
 n_tuning_records <- nrow(DataTraining)
 # Variable that will hold the best combination of hyperparameters and
the
 # MAAPE that was produced.
 best_params <- list(maape=Inf)
 inner_folds <- CV.Kfold(n_tuning_records, k=n_inner_folds)
 for (j in 1:n_combinations) {
  cat("\tCombination:", j, "/", n_combinations, "\n")
  flags <- all_combinations[[j]]
  cat("\t\tInner folds: ")
  for (m in 1:n_inner_folds) {
   cat(m, ", ")
   inner_fold <- inner_folds[[m]]
   DataInnerTraining <- DataTraining[inner_fold$training, ]
   DataInnerTesting <- DataTraining[inner_fold$testing, ]

   # Fit the multivariate model using the current combination of
   # hyperparameters
   tuning_model <- rfsrc(Multivar(GY, DTHD, DTMT, Height) ~ .,
              data=DataInnerTraining, ntree=flags$ntree,
              mtry=flags$mtry, nodesize=flags$nodesize, splitrule="mv.
mse")
   predictions <- predict(tuning_model, DataInnerTesting)

   # Compute MAAPE for all response variables with the current
combination of
   # hyperparameters
   gy_maape <- maape(DataInnerTesting$GY,
            predictions$regrOutput$GY$predicted)
   dthd_maape <- maape(DataInnerTesting$DTHD,
             predictions$regrOutput$DTHD$predicted)
   dtmt_maape <- maape(DataInnerTesting$DTMT,
             predictions$regrOutput$DTMT$predicted)
   height_maape <- maape(DataInnerTesting$Height,
              predictions$regrOutput$Height$predicted)

   current_maape <- mean(c(gy_maape, dthd_maape, dtmt_maape,
height_maape))
   # If the current combination gives a lower MAAPE set it as new
best_params
   if (current_maape < best_params$maape) {
    best_params <- flags
    best_params$maape <- current_maape
   }
  }
  cat("\n")
 }

 # Using the best hyper-params combination, retrain the model but using
the complete
 # training set
 model <- rfsrc(Multivar(GY, DTHD, DTMT, Height) ~ .,
```

```
        data=DataTraining, ntree=best_params$ntree,
        mtry=best_params$mtry, nodesize=best_params$nodesize,
splitrule="mv.mse")
 predicted <- predict(model, DataTesting)

 CurrentPredictions <- data.frame()
 # Bind the predictions of each response variable in the current fold
 for (response_name in responses) {
  CurrentPredictions <- rbind(
   CurrentPredictions,
   data.frame(
    Position=outer_fold$testing,
    GID=Pheno$GID[outer_fold$testing],
    Env=Pheno$Env[outer_fold$testing],
    Partition=i,
    Trait=response_name,
    Observed=DataTesting[[response_name]],
    Predicted=predicted$regrOutput[[response_name]]$predicted
   )
  )
 }

 Predictions <- rbind(Predictions, CurrentPredictions)
}

head(Predictions)
tail(Predictions)

# Summarize the results across environment computing four metrics per
response
ByEnvSummary <- Predictions %>%
        # Calculate the metrics disaggregated by Partition and Env
        group_by(Trait, Partition, Env) %>%
        summarise(MSE=mse(Observed, Predicted),
             Cor=cor(Predicted, Observed, use="na.or.complete"),
             R2=cor(Predicted, Observed, use="na.or.complete")^2,
             MAAPE=maape(Observed, Predicted)) %>%
        select_all() %>%

        # Calculate the metrics disaggregated Env with standard errors
        # of each partition
        group_by(Env, Trait) %>%
        summarise(SE_MAAPE=sd(MAAPE, na.rm=TRUE) / sqrt(n()),
             MAAPE=mean(MAAPE, na.rm=TRUE),
             SE_Cor=sd(Cor, na.rm=TRUE) / sqrt(n()),
             Cor=mean(Cor, na.rm=TRUE),
             SE_R2=sd(R2, na.rm=TRUE) / sqrt(n()),
             R2=mean(R2, na.rm=TRUE),
             SE_MSE=sd(MSE, na.rm=TRUE) / sqrt(n()),
             MSE=mean(MSE, na.rm=TRUE)) %>%
        select_all() %>%
```

```
        # Order by Trait
        arrange(Trait) %>%

        mutate_if(is.numeric, ~round(., 4)) %>%
        as.data.frame()
ByEnvSummary

write.csv(Predictions,
      file="multivariate/results/1.all_as_continuous_all.csv",
      row.names=FALSE)
write.csv(ByEnvSummary,
      file="multivariate/results/1.all_as_continuous_summary.csv",
      row.names=FALSE)
```

## References


Breiman L (1996) Bagging predictors. Mach Learn 26:123–140

Breiman L (2001) Random forests. Mach Learn 45:5–32

Breiman L, Friedman JH, Olshen RA, Stone CJ (1984) Classification and regression trees. Wadsworth, Belmont, California. MR0726392

Chaudhuri P, Lo WD, Loh WY, Yang C-C (1995) Generalized regression trees. Stat Sin 1995:641–666

Chen X, Ishwaran H (2012) Random forests for genomic data analysis. Genomics 99:323–329

Cortes C, Vapnik VN (1995) Support-vector networks. Mach Learn 20:273–297

De'Ath G (2002) Multivariate regression trees: a new technique for modeling species-environment relationships. Ecology 83(4):1105–1117

Evgeniou T, Pontil M (2004) Regularized multi–task learning. In: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp 109–117

Faddoul JB, Chidlovskii B, Gilleron R, Torre F (2012) Learning multiple tasks with boosted decision trees. In: Machine learning and knowledge discovery in databases. Springer, pp 681–696

García-Magariños M, Inaki LU, Cao R, Salas A (2009) Evaluating the ability of tree-based methods and logistic regression for the detection of SNP-SNP interaction. Ann Hum Genet 73:360–369

Glocker B, Pauly O, Konukoglu E, Criminisi A (2012) Joint classification-regression forests for spatially structured multi-object segmentation. In: Computer vision–ECCV 2012. Springer, pp 870–881

González-Recio O, Forni S (2011) Genome-wide prediction of discrete traits using Bayesian regressions and machine learning. Genet Sel Evol 43:7

Ishwaran H, Kogalur UB (2008) RandomSurvivalForest 3.2.2. R package. http://cran.r-project.org

Larsen DR, Speckman PL (2004) Multivariate regression trees for analysis of abundance data. Biometrics 60(2):543–549

Lee SK, Jin S (2006) Decision tree approaches for zero-inflated count data. J Appl Stat 33:853–865

Li B, Zhang N, Wang Y-G, George AW, Reverter A, Li Y (2018) Genomic prediction of breeding values using a subset of SNPs identified by three machine learning methods. Front Genet 9:237. https://doi.org/10.3389/fgene.2018.00237

Loh WY (2002) Regression trees with unbiased variable selection and interaction detection. Stat Sin 2002:361–386

Mathlouthi W, Larocque D, Fredette M (2019) Random forests for homogeneous and non-homogeneous Poisson processes with excess zeros. Stat Methods Med Res 29(8):2217–2237

Montesinos-López OA, Montesinos-López A, Mosqueda-Gonzalez BA, Montesinos-López JC, Crossa J, Lozano-Ramirez N, Singh P, Valladares-Anguiano FA (2021) A zero altered Poisson random forest model for genomic-enabled prediction. Genes, Genome and Genetics 11(2): jkaa057

Naderi S, Yin T, König S (2016) Random forest estimation of genomic breeding values for disease susceptibility over different disease incidences and genomic architectures in simulated cow calibration groups. J Dairy Sci 99:7261–7273. https://doi.org/10.3168/jds.2016-10887

Sarkar RK, Rao AR, Meher PK, Nepolean T, Mohapatra T (2015) Evaluation of random forest regression for prediction of breeding value from genomewide SNPs. J Genet 94(2):187–192. https://doi.org/10.1007/s12041-015-0501-5

Schapire R, Freund Y, Bartlett P, Lee W (1998) Boosting the margin: a new explanation for the effectiveness of voting methods. Ann Statist 26:1651–1686. MR1673273

Segal MR (1992) Tree-structured methods for longitudinal data. J Am Stat Assoc 87(418):407–418

Segal M, Xiao Y (2011) Multivariate random forests. WIREs Data Min Knowl Discov 1(1):80–87

Stephan J, Stegle O, Beyer A (2015) A random forest approach to capture genetic effects in the presence of population structure. Nat Commun 6:7432. https://doi.org/10.1038/ncomms8432

Tang F, Ishwaran H (2017) Random forest missing data algorithms. Stat Anal Data Min 10:363–377

Therneau T, Atkinson B (2019) rpart: recursive partitioning and regression trees. R Package Version 4:1–15. https://CRAN.R-project.org/package=rpart. Accessed Aug 2019

Waldmann P (2016) Genome-wide prediction using Bayesian additive regression trees. Genet Sel Evol 48:42. https://doi.org/10.1186/s12711-016-0219-8

Zhang H (1998) Classification trees for multiple binary responses. J Am Stat Assoc 93(441):180–193



**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

# Index

**A**
Activation functions, 382
  artificial neural networks, 387
    leaky ReLUs, 389
    linear, 388
    rectifier linear unit, 388, 389
    sigmoid, 390
    softmax, 390, 391
    Tanh, 391
Activation map, 540
Adadelta adaptive learning ability, 441
Adagrad, 441
Adaptive Moment Estimation (Adam), 437
Additive genetic values, 52
Algorithm adaptation approach, 655
ANN and deep learning (DL)
  activation functions, 428
  epoch, 433, 434
  hyperparameters, 427
  learning rate, 432, 433
  linearly and nonlinearly separable patterns, 429
  loss function, 428
  MaizeToy (*see* MaizeToy data set)
  network topology, 427
  normalization scheme for input data, 434
  number of batches, 433
  number of hidden layers, 428, 429
  number of neurons in each layer, 430, 431
  optimizer
    adaptive moment estimation, 437
    batch gradient descent, 436
    saddle point and vanishing gradient, 437
    stochastic gradient descent, 436
  popular DL frameworks, 435
  regularization type, 431
ANN topologies, 393–396
Appropriate fitting phenomenon, 110
Approximate kernel method, 329
  with five kernels, 331
Arc-cosine kernel (AK), 265
  R code, hidden layers in, 308
Area Under the receiver operating characteristic Curve (AUC–ROC), 133
Artificial intelligence (AI), 384
Artificial neural framework, 394
Artificial neural networks (ANN), 109, 379, 381
  activation functions, 387
    leaky ReLUs, 389
    linear, 388
    rectifier linear unit, 388, 389
    sigmoid, 390
    softmax, 390, 391
    Tanh, 391
  applications of, 396, 398, 399
  building blocks of, 382–387
  loss function, 399
    for binary and ordinal outcomes, 401, 402
    for continuous outcomes, 400, 401
    early stopping method of, 405–407
    regularized loss functions, 402–405
  topologies, 393–396
  training artificial neural networks, king algorithm for, 407–412
    backpropagation algorithm, 412, 413
    hand computation, 413–418, 420–424
  universal approximation theorem, 392, 393
Average Pearson's correlation (APC), 194, 197
Average pooling, 544

© The Author(s) 2022

O. A. Montesinos Lopez et al., *Multivariate Statistical Machine Learning Methods for Genomic Prediction*, https://doi.org/10.1007/978-3-030-89010-0

**B**
Backpropagation algorithm, 414
Backpropagation method, 407
Backward approach, 431
Bagging, 637
Balanced incomplete block (BIB) design, 121
Bandwidth parameter, 268
Banks, 2
Batch gradient descent (BGD), 436
BayesA model, 178, 179, 182
BayesA prior model, 218
BayesB model, 183
BayesC model, 180
Bayesian and classical models, 209
Bayesian and classical regression models, 235
Bayesian estimation, 604
Bayesian generalized linear regression (BGLR)
  Bayes theorem, 171
  Bayesian framework, 172
  Bayesian genome-based Ridge regression, 172, 176
  Bayesian Lasso linear regression model, 184
  BMTE, 195, 197
  extended predictor, 186, 187
  GBLUP genomic model, 176, 178
  genomic-enabled prediction BayesA model, 178, 179
  genomic-enabled prediction BayesB model, 183, 184
  genomic-enabled prediction BayesC model, 180, 183
  multi-trait linear regression model, 188, 190, 191, 194
Bayesian genomic multi-trait and multi-environment model (BMTME), 195–197
Bayesian information criterion (BIC), 582–584, 592, 596–598
Bayesian kernel BLUP, 282, 316
  extended predictor under, 283, 285, 286
  with only genotypic effects in predictor, 311
Bayesian kernel methods, 280, 282–284
  Bayesian kernel BLUP, extended predictor under, 283, 285, 286
    with binary response variable, 286, 287
    with categorical response variable, 287, 288
Bayesian linear regression model, 215
Bayesian ordinal regression model
  applications, 209
  BGLR, 213, 214, 216, 218
  chi-squared distribution, 213
  hyperparameters, 213
  linear regression models, 210, 212
  logistic ordinal model, 210
  multivariate normal distribution, 211
  parameters, 211
  probit ordinal regression model, 211
  probit regression model, 210
Bayesian Ridge Regression (BRR), 604
Bayes theorem, 171
Bernoulli random distribution, 182
Best linear unbiased estimates (BLUEs), 36–38, 40–43
Best linear unbiased predictors (BLUPs), 36–38, 40–43
Beta coefficient function, 579, 580, 596, 598, 600
BGLR package, 607, 609, 611
BGLR R package, 216
Bias, 110–114, 120, 129, 133
Big data, 1
Binary_crossentropy, 477, 478, 480, 496, 506
Binary response variable, 301
  Bayesian kernel BLUP, extended predictor under, 286, 287
  kernel methods for, 271, 273, 274
Biological neural networks, 381
Biological neuron, 380
BMTME library, 438
Bootstrap cross-validation, 119, 120
Breiman's random forests, 642
Brier score (BS), 136, 216, 217, 219–221, 229, 231
Broyden–Fletcher–Goldfarb–Shanno (BFGS), 128
B-spline basis, 584–593, 597, 598, 600, 602, 607–610

**C**
Cambridge Analytica, 3
Categorical_crossentropy, 477, 478, 500, 506
Categorical_crossentropy loss function, 500
Categorical encoding, 482
Categorical response variable
  Bayesian kernel BLUP, extended predictor under, 287
  kernel methods for, 274, 275
Categorical variables, 26
Centering, 57
Chi-squared distribution, 604
Cholesky decomposition, 177, 233
Cholesky decomposition of GRM, 439
Cholesky factorization, 194

Clustering, 32
Code_Tuning_With_Flags_00.R, 446
Code_Tuning_With_Flags_Bin.R, 479, 480
Code_Tuning_With_Flags_CNN_1D_1HL_2CHL.R, 557, 559
Code_Tuning_With_Flags_CNN_2D_2HL_1CHL.R, 563
Code_Tuning_With_Flags_Count_Lasso.R, 487
Code_Tuning_With_Flags_MT_Binary.R, 495, 496
Code_Tuning_With_Flags_MT_Count. R, 503
Code_Tuning_With_Flags_MT_Mixed.R, 504, 506
Code_Tuning_With_Flags_MT_normal.R, 491, 493
Code_Tuning_With_Flags_MT_Ordinal.R, 498, 500
Code_Tuning_With_Flags_Ordinal_4HL2.R, 483, 485
Cohen's Kappa, 133
Compression method, 63–67
Confusion matrix, 131
Continuous response variables, 270, 298
Convolution, 539
Convolutional neural networks (CNNs)
  0D tensor, 535
  1D tensor, 535, 545
  2D convolution, 560, 562, 564, 566
  2D tensor, 535
  3D tensor, 535, 536
  4D tensor, 537, 539
  convolution, 539, 540, 542
  critics of deep learning, 566, 567
  feedforward deep neural networks for processing images
    CNNs and deep feedforward networks relation, 552
    filter matching, 549
    local filters and feature maps, 550
    with multiple hidden layers, 553
    3D tensor converted to 1D tensor, 549
  importance, 533, 534
  maize, 554, 556, 558, 559
  motivation of CNN, 546–548
  pooling, 542–545
Covariance matrix adaptation evolution strategy (CMA-ES), 128
Cross-entropy, 401
Cross-validation (CV), 601
  bootstrapping, 119, 120
  definition, 115
  five partitions of training-validation-testing, 124
  incomplete block CV, 120, 121
  k-fold, 116, 117
  Leave-m-Out (LmO), 117
  Leave-One-Group-Out (LOGO), 118, 119
  Leave-One-Out (LOO), 117
  random cross-validation, 118
  random cross-validation with blocks, 121, 122
  single hold-out set approach, 115
  training, validation (tuning), and testing sets, 123
ctree () function, 637
Custom callback function, 443

**D**

Data_Count_Toy.RData, 501
Data_Toy_EYT.RData, 644, 653
Data_Toy_EYT.RData data set, 643
Decorrelate trees, 638
Deep learning, 383–386
  critics of, 566, 567
Deep neural networks (DNN)
  training DNN for binary outcomes, 477
    binary outcomes, 478
    configuring and compiling the model, 477
    fitting the model, 478
    in Keras, 477
    prediction performance, 478, 481
  training DNN for categorical (ordinal) outcomes
    configuring and compiling the model, 483
    fitting the model, 483
    in Keras, 482
    ordinal outcome, 483
    prediction performance, 483, 486
  training DNN with count outcomes, 486–489
  training DNN with multivariate outcomes
    multivariate binary outcomes, 493, 496, 497
    multivariate continuous outcomes, 490, 493
    multivariate count outcomes, 501, 503
    multivariate mixed outcomes, 504, 506
    multivariate ordinal outcomes, 498, 501
Diagonal matrix, 19
DL, applications of, 396, 398, 399

Dropout, 403
DTHD categorical variable, 368
Dying ReLU, 389

**E**
Early stopping approach, 443
Early stopping rule, 434
Elastic Net penalization, 431
Epoch, 433, 434
Estimability matters, 37
Estimable functions, 37, 42
Expected prediction error (EPE), 114
Exponential kernel, 264
Extended predictor, BGLR, 186, 187

**F**
Facebook, 3
Feedforward artificial neural network, 395
Feedforward networks, 395, 546–552, 556–558, 560, 562–564
Feedforward neural network, 404
Filter matching, 550
Fine-mapping approach, 4
Five-fold cross-validation, 439, 647, 650
FIXED, 175
Fixed effects, 13, 35, 36
flags() function, 446
Flexible models, 111
FLRSDS, 194
Forward approach, 431
4D tensors, 537
Fourier basis, 584, 585, 597–600, 602, 604, 605, 607, 609, 611
Fully connected deep neural network, 548
Functional regression
  basis function
    B-spline basis, 585–588, 591, 592
    Fourier basis, 584, 585, 592, 598
  Bayesian estimation, 604
  functional linear regression model principles, 579, 580, 582, 583
  with smoothed coefficient function, 598, 599, 601

**G**
Gaussian kernel, 263, 264
Gaussian noise, 592
Gaussian response variables, kernel methods for, 269–271
GBLUP Bayesian method, 56
GBLUP method, 53
Generalized kernel model, 253
  building kernel, two separate step process for, 269
  frequentist paradigm, parameter estimation under, 253, 254
  kernel, 255, 256
  kernel trick, 257–260
  popular kernel functions, 260–268
Genetic algorithms (GA), 128
GenoMaizeToy, 438, 491
Genomic BLUP (GBLUP), 277
Genomic breeding values and estimation, 52–56
Genomic prediction (GP) accuracy, 5
Genomic relationship matrix (GRM), 49–52, 176, 361, 438
Genomic selection (GS), 3, 4
  concepts of, 4–6
  matrix algebra, 17–24
  model building, two cultures of, 9
    inference, 10
    prediction, 10
    testing causal hypotheses, 10
    tobacco dust, 11
  statistical machine learning, 6–9, 11
    model effects, 13–16
    multivariate data types, 25–28
    nonparametric model, 12
    parametric model, 12
    semiparametric model, 12, 13
    semi-supervised learning, 33
    supervised learning, 29–31
    unsupervised learning, 32, 33
Gibbs phenomenon, 589
Gibbs sampler, 175
Gibbs sampler exploration, 214, 224
Gibbs sampler method, 173
Gibbs sampling method, 211
Gini index criterion, 642
Gini index splitting, 643
Gini split rule, 649
Gini splitting rule, 656
Glmnet R package, 234
Gradient-descent-based algorithm, 441
Gradient descent (GD) method
  linear multiple regression model via, 76–79
Grain yield (GY), 14
Gram matrix, 257
Grid search method, 127

**H**

Hand computation, 413–423
Hand-tuning kernel functions, 278, 280
Hess, Victor, 10
Hessian matrix, 100
Hidden layers, 386, 429
Hidden neuron, 386
High bias, 113
High-Level DL frameworks, 435
Hinge loss, 401
Hybrid kernels, 267, 268
Hyperparameters, 124, 355, 366
Hyperpar data frame, 450
Hyperplane, 338, 340, 342, 345, 348, 354
  mathematical point, 339

**I**

Identical by state (IBS), 45
Identity matrix, 19
Incomplete block (IB) CV, 120
Inflexible models, 111
Inner cross-validation, 124
Input layer, 385
Input neuron, 386
*install_git*, 193
Interlayer connection, 394
Internal covariate shift, 484
Interpretability, 111, 114
Intralayer connection, 394
Inverse Chi-square distribution, 173, 174

**K**

Kappa coefficient (Kappa), 133, 481, 497, 649, 650
Karush–Kuhn–Tucker conditions, 348
Keras, 435
Keras_model_sequential() function, 477
Kernel averaging, 267
Kernel compression methods, 289–292, 294, 295
  approximate kernel method, extended predictor under, 294, 296, 297
Kernel function, 359
Kernel regression, 12
Kernels, 255, 256
  Bayesian kernel BLUP
    with only genotypic effects in predictor, 311
  binary response variable, 301
  with continuous response variable, 298
  R code for, 306
  in rrBLUP, 305
Kernel trick, 252
K-fold cross-validation, 116, 118, 227

**L**

Lagrange multipliers, 352
Large number of neurons, 393
Lasso linear regression model (BL), 184
Lasso logistic regression, 102–104
Lasso penalization, 431
Lasso regression, 93–98
Latin hypercube sampling, 127
Layer_batch_normalization() function, 484
Layer_conv_2d(), 564
Layer_dense() function, 480
Layer_max_pooling_1d() function, 558
Layer_max_pooling_2d(), 562, 564
Leaky ReLUs, 389
Learning curves (LC), 117
Learning rate, 432
Leave-m-Out (LmO) CV, 117
Leave-One-Group-Out (LOGO) CV, 118, 119
Leave-One-Out (LOO) CV, 117
Leave-One-Out cross-validation (LOOCV), 583, 592
Library(devtools), 193
Linear kernel, 290
Linear mixed model with kernels, 274, 276–279
Linear multiple regression model (LMRM), 71
  advantages and disadvantages of, 80, 81
  via gradient descent method, 76–79
  via maximum likelihood, 75, 76
  via the ordinary least square (OLS) method, 71–74
Logistic loss, 401
Logistic ordinal model, 210
Logistic regression, 98–100
  Lasso logistic regression, 102–104
  logistic Ridge regression, 100–102
Logistic Ridge regression, 100–102
Loss function, 399, 428
  for binary and ordinal outcomes, 401, 402
  for continuous outcomes, 400, 401
  early stopping method of, 405–407
  regularized loss functions, 402–405
Low bias, 113
Lower triangular matrix, 20
Low-level high-level DL framework, 435
Low variance, 113

**M**


MAAPE, 503
Mahalanobis splitting rule, 657
MaizetToy data set, 490
  Code_Tuning_With_Flags_00.R, 446, 447
  custom callback function, 443
  early stopping approach, 444, 445
  five-fold cross-validation strategy, 439
  five-fold CV with dropout, 451–453
  five inner cross-validations, 450
  flags() function, 446
  genoMaizeToy, 438
  gradient-descent-based algorithm, 441
  Keras library, 440
  observed versus predicted values for fold 2 trained with grid search, 451
  phenoMaizeToy, 438
  prediction performance, 452–454
  sequential models, 441
  training and validation metrics, 443, 444
  TRN–TST partition, 449
  with more than one hidden layer with inner CV, 453, 454
  with more than one hidden layer without inner CV, 455
  with one hidden layer with inner CV
    a dropout rate of 5%, and different optimizers, 458
    and Ridge, Lasso, and elastic net (Ridge-Lasso) regularization, 455, 456
Major allele, 46
Manual tuning, 127
Marker-assisted selection (MAS), 4
Marker depuration, 43–49
Matrix algebra
  genomic selection, 17–24
Matrix arrays, 535
Matthews correlation coefficient (MCC), 136
Maximum likelihood (ML), 277
  linear multiple regression model via, 75, 76
Maximum-margin classification, 401
Maximum margin classifier, 341, 342
  classification model, 343
  hyperplane, 343
  optimization problem, 344
Max normalization, 58
Max pooling, 542–544
Mean absolute error (MAE), 130
Mean absolute percentage error (MAPE), 130
Mean arctangent absolute percentage error (MAAPE), 130
Mean square error (MSE), 129, 284, 598, 600, 602, 603, 610
Mean square error of prediction (MSEP), 187, 194, 197, 235, 609
Mini-batch gradient, 437
Minimax normalization, 58
Minor allele, 46
Minor allele frequency (MAF), 47
Mixed-effects models, 13
Mixed outcomes, 643, 657, 658
MIXTIM, 194
Model interpretability, 111–114
Model tuning
  grid search method, 127
  hyperparameter, 125
  importance of, 126
  optimization methods, 128
  PCCC, 125
  random search method, 127
  representation of, 125
mtry, 638
Multilayer neural networks, 386
Multinomial logistic regression model, 225
Multi-trait Bayesian kernel, 288, 289
Multi-trait Bayesian kernel BLUP
  with Gaussian response variable, environment + genotype + genotype×environment interaction, 325
Multi-trait linear regression model
  BGLR, 188, 190, 194
Multi-trait mixed outcome DNN model, 506
Multi-trait ordinal DNN model, 500
Multivariate regression analysis, 656
Multivariate RF model, 655
mv.mse splitting function, 658


**N**


Nadaraya–Watson (NW) kernel estimator, 12
Negative log-likelihood (MLL), 136
Newton–Raphson equation, 100
Nodesize, 643
No free lunch theorem, 566
Nonlinear problem, 358
Nonparametric model, 12
Normalization methods, 57, 58


**O**


One-dimensional tensors (1D tensors), 535
One-hot encoding, 482
One-versus-all approach, 361
One-versus-one classification approach, 361
Optimization problem, 349

optimizer_adagrad, 458
optimizer_adamax, 458
Ordinal ogistic regression
  Bayesian specification, 222
  conditional distribution, 223
Ordinal probit model, 210
Ordinal probit regression model, 211
Ordinary least square (OLS) method, 71–74
Output layer, 386
Output neuron, 386
Overfitting phenomenon, 109, 110

**P**
Parametric model, 12
Particle swarm (PS), 128
Pearson’s correlation, 129, 130, 493
Penalized Bayesian functional regression (PBFR), 607, 609, 610
Penalized functional regression (PFR), 602, 603, 607, 609
Penalized multinomial ogistic regression, 225
  design matrix and vector, 229
  GBLUP implementation, 230
  GMLRM-L3 model, 230
  likelihood, 227
  logistic Ridge regression, 226
  regularization parameter, 226
Penalized Poisson regression, 233
  parameter estimation, 232
  vector covariates, 232
Penalized Ridge multinomial logistic regression (PRMLR) model, 229
Percentage of cases correctly classified (PCCC), 125
Pheno$y, 647
PhenoMaizeToy, 438
Plant breeding, 3
Poisson distribution, 487
Poisson loss, 402, 489, 503
Poisson Ridge regression, 233
Pólya-Gamma distribution, 222
Pólya-Gamma latent random variable, 221
Pólya-Gamma random variables, 224
Polynomial kernel, 261
Pooling, 542–545
Positive definite symmetric (PDS), 360
predict_classes() function, 478, 483
Predictable functions, 41–43
Prediction accuracy, 111, 113, 114, 130
Prediction performance
  binary and ordinal measures, 131
    AUC–ROC, 133, 135
    confusion matrix, 131
    Kappa coefficient, 133
    MCC, 136
    proportion of cases correctly classified, 132
  MAAPE, 130
  MAE, 130
  MSE, 129
  Pearson’s correlation coefficient, 129
  RMSE, 129
  Spearman’s correlation, 137
Preprocessing tools, for data preparation
  BLUEs and BLUPs, 36, 37
    estimable function, 37–40, 42, 44
    predictable functions, 41–43
  fixed/random effects, 35, 36
  genomic breeding values and estimation, 52–56
  genomic relationship matrix, 49–51
  marker depuration, 43, 44
    genetic marker, 44–49
  normalization methods, 57, 58
  principal component analysis, 63–68
  removing/adding inputs, suggestions for, 58–63
Principal component analysis (PCA), 63–68
Proportion of cases correctly classified (PCCC), 132, 217, 219–221, 506, 649, 650
Python, 435

**Q**
QTL, 4

**R**
Random cross-validation, 118, 121
Random effects, 13, 35, 36
Random forest (RF), 128
  algorithm continuous, binary, and categorical response variables, 639
    minimum node size, 641
    node size, 641
    number of independent variables (mtry), 640
    number of trees in the forest (ntree), 640
    splitting rule, 643
  bagging, 637
  continuous, binary, and categorical response variables, 640
  decision tree, 634, 637
  decorrelate trees, 638

Random forest (RF) (*cont.*)
for multivariate response variables, 655, 658, 661
Gini split rule, 649
motivation, 633, 634
Pheno$y, 647
prediction performance
PCCC and Kappa, 650
random forest model works and combines multiple trees, 638
splitting rules, 641–643, 646
Breiman's random forests, 642
SSE, 641
weighted binary cross-entropy, 642
terminal node, 638
ZAP_RF and ZAPC_RF, 650, 652
randomForestSCR library, 642
randomForestSRC package, 643
Random search method, 127
Rectifier linear unit (ReLU) activation function, 388, 389
Recurrent networks, 396
Recurrent neural networks (RNNs), 396
Regularization, 431
Regularized linear multiple regression model
Lasso regression, 93, 94, 96–98
logistic regression, 98–104
Ridge regression, 81–93
RELU activation functions, 428, 477, 480, 481, 487
Reproducing Kernel Hilbert Spaces (RKHS) regression methods, 251, 252
Bayesian kernel methods, 280, 282–284
Bayesian kernel BLUP, extended predictor under, 283, 285, 286
with binary response variable, 286, 287
with categorical response variable, 287, 288
binary response variables, kernel methods for, 271, 273, 274
categorical response variables, kernel methods for, 274, 275
Gaussian response variables, kernel methods for, 269–271
generalized kernel model, 253
building kernel, two separate step process for, 269
frequentist paradigm, parameter estimation under, 253, 254
kernel, 255, 256
kernel trick, 257–260
popular kernel functions, 260–268
hand-tuning kernel functions, 278, 280
kernel compression methods, 289–292, 294, 295
approximate kernel method, extended predictor under, 294, 296, 297
linear mixed model with kernels, 274, 276–279
multi-trait Bayesian kernel, 288, 289
rfsrc() function, 657
Ridge and Lasso regression models, 601
Ridge penalization, 431
Ridge regression, 81–93, 127
Ridge regularization, 455
Root mean square error (RMSE), 129
rrBLUP, 305
running_run() function, 501

**S**

Scaling, 57
Self-connection, 394
Semiparametric model, 12, 13
Semi-supervised learning, 33
Sequential models, 441
Sigmoid, 390
Sigmoid activation, 390
Sigmoidal kernel, 263
Simple nucleotide polymorphism (SNP), 45
Single nucleotide polymorphism (SNP), 4
Slack variables, 354
Smoothed coefficient function, 598, 601–603
SNP-BLUP method, 55
Soft margin support vector machine, 355
Softmax activation function, 390, 391, 482, 484, 506
Spearman's correlation, 137
Splitrule, 643
Splitting rule, 641
Square matrix, 19, 20
Standardization, 57
Statistical analysis system (SAS) software, 128
Statistical machine learning, 109–112, 115–117, 119, 120, 122–127, 129–131, 133–136
genomic selection, 6–9, 11
model effects, 13–16
multivariate data types, 25–28
nonparametric model, 12
parametric model, 12
semiparametric model, 12, 13
semi-supervised learning, 33
supervised learning, 29–31
unsupervised learning, 32, 33
Stochastic gradient descent (SGD), 436

Stride, 540
Supervised learning, 29, 30
Supervised statistical machine learning models
  linear multiple regression model, 71
    advantages and disadvantages of, 80, 81
    gradient descent method, 76–79
    maximum likelihood, 75, 76
    OLA, 71–74
  regularized linear multiple regression model
    Lasso regression, 93, 94, 96–98
    logistic regression, 98–104
    Ridge regression, 81–93
Support vector machines (SVM), 251, 337, 360, 361, 364
  classification, 338
  cross-validation set, 367
  gamma parameter, 366
  gamma values, 368
  hyperparameters, 368
  mathematics, 338
  optimization problem, 360
  PCCC, 364, 365
  R package, 360
  type, 338
Support vector regression (SVR), 369
  genomic predictions, 371
  hyperplane, 369
  implementation, 369
  performance, 369
Supralayer connection, 394
Surrogate-based Bayesian optimization, 128
Synapse, 380

**T**
Tabu search (TS), 128
Tanh activation, 391
Ten-fold cross-validation, 271
TensorFlow, 435
Testing (TST) set, 4, 118
Three-dimensional tensors (3D tensors), 535
to_categorical() function, 482, 485
Training artificial neural networks, king algorithm for, 407–412
  backpropagation algorithm, 412, 413
  hand computation, 413–418, 420–424
Training (TRN) data, 5, 118
Training-validation-test, 124
TRN–TST partition, 449
True breeding values (TBVs), 52
Tuning_run () function, 450, 480, 485, 503, 559
2D convolution, 560, 562–564, 566
Two-dimensional tensors (2D tensors), 535
Two separate step process for building kernels, 269

**U**
Underfitted phenomenon, 109, 110
Univariate binary outcome, 477
Univariate DNN models, 492, 493
Univariate RF models, 655
Universal approximation theorem, 392, 393
Unsupervised learning, 32, 33
Upper triangular matrix, 20

**V**
Variable importance measures (VIM), 642, 645, 646
Variance, 110–114, 117, 119, 130
Variance–covariance matrix, 84

**W**
Weighted binary cross-entropy, 642
Wolfe dual, 346, 347

**Z**
Zero-altered Poisson custom random forest (ZAPC_RF), 650
Zero-altered Poisson random forests (ZAP_RF), 650
Zero-dimensional tensors (0D tensors), 535