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

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.

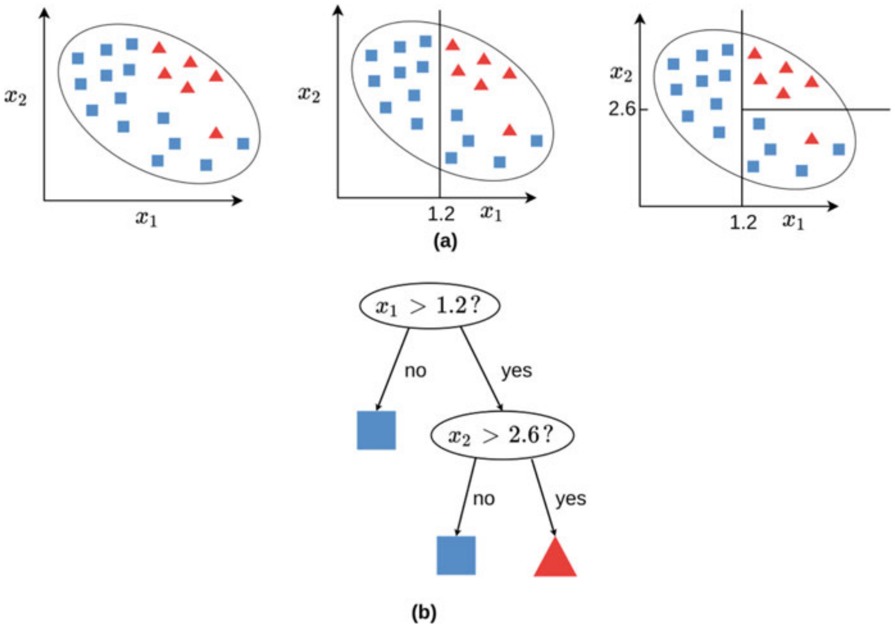

Fig. 15.1 Illustration of a decision tree for a classification problem

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):

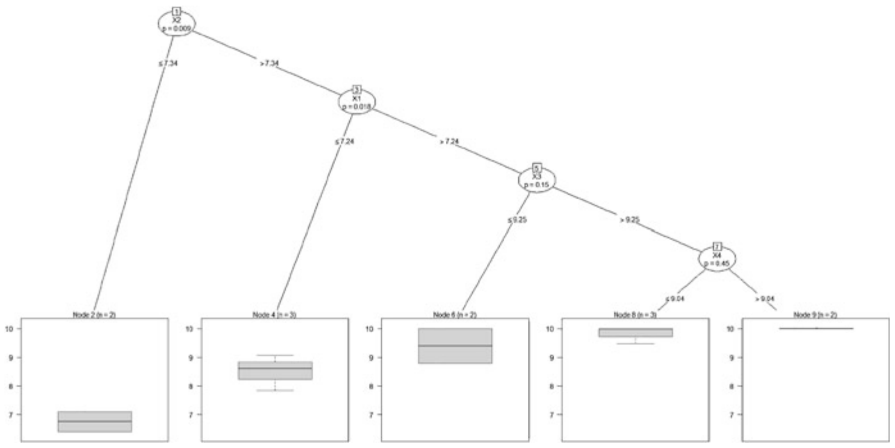

Fig. 15.2 Decision tree for a regression problem with five terminal nodes

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

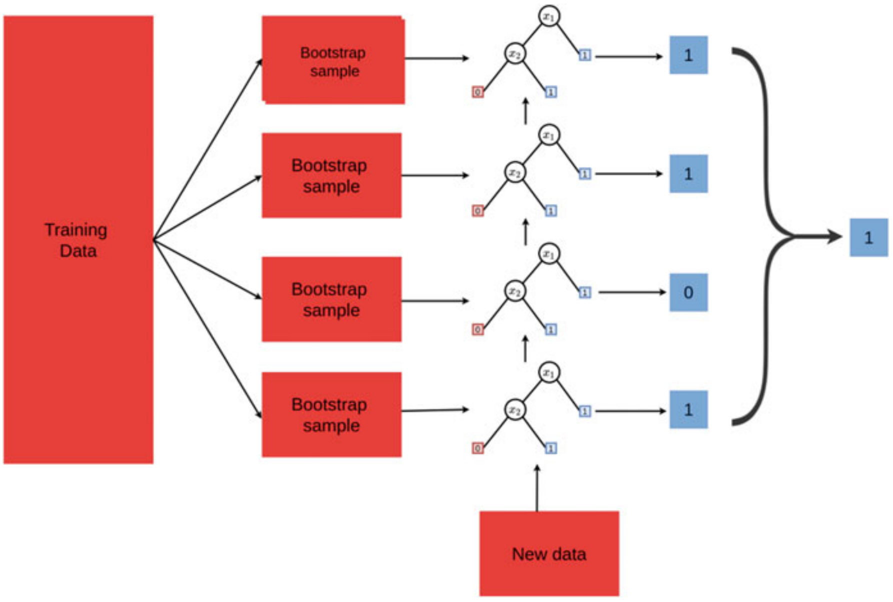

Fig. 15.3 Illustration of how the random forest model works and combines multiple trees

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.

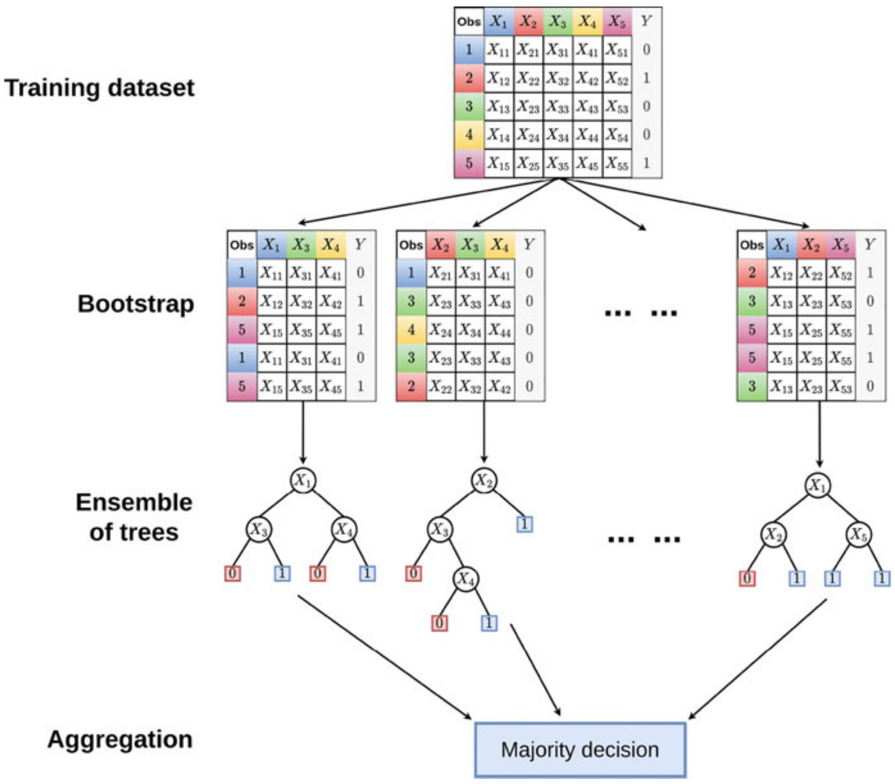

Fig. 15.4 Illustration of how bootstrap samples and samples of predictors are selected under the random forest model

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_{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 bth 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

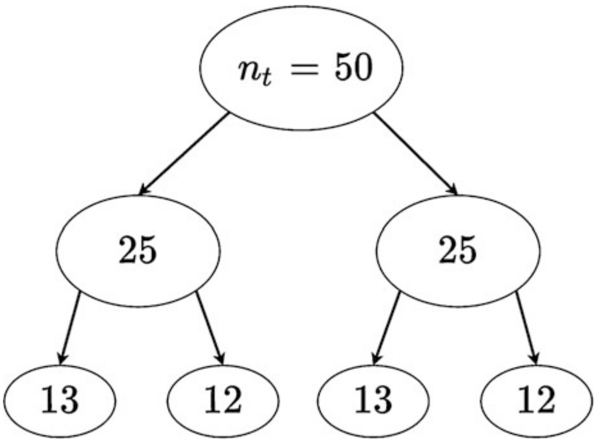

Fig. 15.5 Illustration of the minimum node size in a decision tree

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

$$\mathrm{GI} = \left[\sum\nolimits_{i=1}^{C} p_{Li}(1 - p_{Li})\right]\Omega_L + \left[\sum\nolimits_{i=1}^{C} p_{Ri}(1 - p_{Ri})\right]\Omega_R,$$

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

$$\mathrm{CE} = -\left[\sum\nolimits_{i=1}^{C} p_{Li} \log\left(p_{\mathrm{Li}}\right)\right]\Omega_L - \left[\sum\nolimits_{i=1}^{C} p_{Ri} \log\left(p_{\mathrm{Ri}}\right)\right]\Omega_R$$

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 × 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

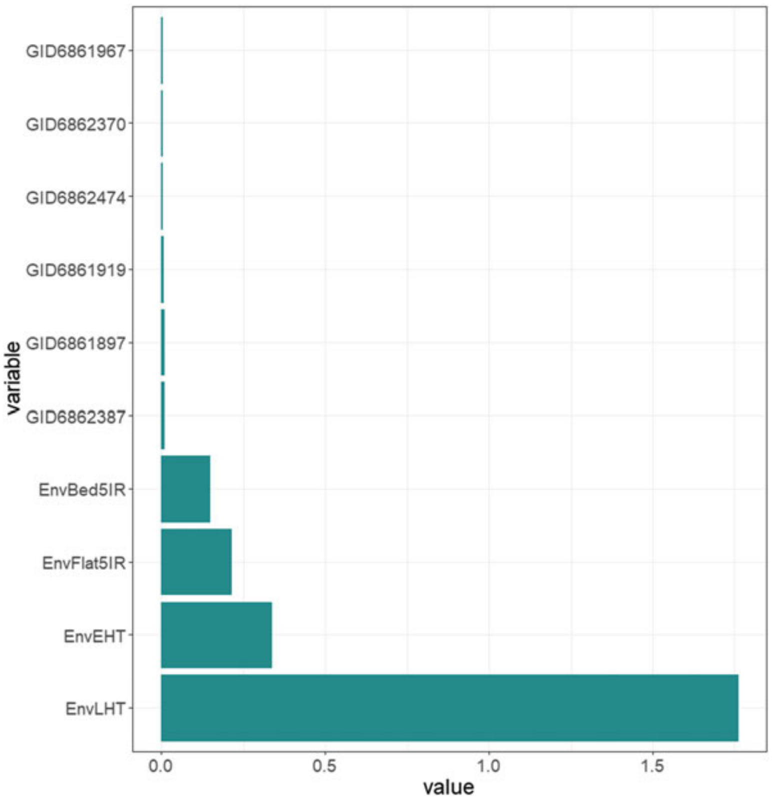

Fig. 15.6 Variable importance measures (VIM) for the ten most important predictors of the data set Data_Toy_EYT.RData

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$\times$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 θ ($0 \leq \theta < 1$), and that Y follows a zero-truncated Poisson distribution with parameter μ ($\mu > 0$), 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

$$E(Y) = \frac{(1-\theta)\exp(-\mu)}{(1-\exp(-\mu))} \text{and } \mathrm{Var}(Y)$$
$$= \frac{(1-\theta)}{(1-\exp(-\mu))}\left(\mu+\mu^2\right) - \left(\frac{(1-\theta)}{(1-\exp(-\mu))}\mu\right)^2$$

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 μ and θ 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_μ and f_θ are general unknown link functions. A general nonparametric and flexible procedure can be used to estimate f_μ and f_θ 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 μ 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 μ and θ, 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 θ 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 $S = \prod^{r}{}_{j=1} s_j$ 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