

Ameet V. Joshi

# Machine Learning and Artificial Intelligence

*2nd Edition*

 Springer

# Machine Learning and Artificial Intelligence

Ameet V. Joshi

# Machine Learning and Artificial Intelligence

Second Edition

 Springer

Ameet V. Joshi  
Microsoft (United States)  
Redmond, WA, USA

ISBN 978-3-031-12281-1      ISBN 978-3-031-12282-8 (eBook)  
<https://doi.org/10.1007/978-3-031-12282-8>

1<sup>st</sup> edition: © Springer Nature Switzerland AG 2020

2<sup>nd</sup> edition: © The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

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

*To everyone who believes in human  
intelligence to be mature enough to coexist  
with machine intelligence and be benefitted  
and entertained by it...*

# Preface to the Second Edition

The first edition of the book received a resounding response from the readers and I was extremely thrilled to have the opportunity to have a follow up second edition of the book.

I have been collecting feedback from multiple readers and reviewers over the last couple of years and also keeping close with the new developments happening the field. There was certainly more content that was ready to be added to the book, as well as some content needed edits. Also, this time Springer suggested that I should be creating a textbook version rather than reference book version as was the case with first edition. This meant making the content more tuned for collegiate curriculum. It was a challenge that I was happy to accept.

So, this second edition of the book is primarily targeted to serve as a textbook for an undergrad or graduate level course on machine learning and artificial intelligence. The book is now organized into five parts including a conclusion. The core of the book borrows the concepts from first edition but then and adds implementations to all the algorithms as they are explained in the same chapter. This way, students can now complete the understanding of the concepts with implementation and application to real world problems.

Part **I** of the book sets the stage for learning the vast array of machine learning techniques and their applications with foundationary concepts, and implementation platform details.

Part **II** of the book dives deep into the theory of machine learning techniques coupled with implementing them using cloud based open source resources.

Part **III** Integrates all these concepts to study how to build end to end machine learning pipelines and performance measurement of the implemented ML models.

Part **IV** focuses on artificial intelligence, which is essentially application of all the techniques learned so far to create intelligent experiences in practical situations.

I wish good luck to all the student embarking on the awesome journey to learn a true twentieth century science of machine learning and its application to create artificial intelligence. Hope that this book will give them a solid foundation to build their future careers.

Redmond, WA, USA  
March 2022

Ameet V. Joshi

# Preface to the First Edition

One of the greatest physicists of all time and Nobel Laureate Dr. Richard Feynman was once asked by his peer to explain a property of Fermi Dirac statistics, which was then a very recent discovery. Feynman quickly said,

Not only I will explain it to you, but I will prepare a lecture on it for freshman level.

However, quite unusually, after few days, he came back and admitted,

I could not do it. I just could not reduce the explanation to freshman level. That means we really don't understand it.

It was quite a bold remark coming from even Dr. Feynman. However, apart from the topic of Fermi Dirac statistics itself, it alludes to a very deep thought about our understanding of things in general. Freshman level here essentially meant something that can be derived directly using the first principles in mathematics or physics. This thought has always made me conscious to try and explain everything that I claim to understand using first principles, try to explain everything conceptually and not only using elaborate set of equations.

The area of *artificial intelligence* and *machine learning* has exploded over last decade. With the widespread popularity, the core concepts in the field have been sometimes diluted and sometimes reinterpreted. With such exponential growth in the area, the scope of the field has also grown in proportion. A newcomer in the area can quickly find the topic daunting and confusing. One can always start with searching the relevant topics on the Web, or just start with Wikipedia, but more often than not every single topic opens a rabbit-hole with more and more new and unknown concepts, and one can get lost very easily. Also, most of the concepts in machine learning are deeply rooted in mathematics and statistics. Without solid background in theoretical mathematics and statistics, the sophisticated derivations of the theorems and lemmas can make one feel confused and disinterested in the area.

I have made an attempt here to introduce most fundamental topics in machine learning and their applications to build artificially intelligent solutions with an intuitive and conceptual approach. There would be some mathematical guidance



used from time to time, without which the concepts would not be sufficiently clear, but I have tried to avoid complex derivations and proofs to make the content more accessible to readers that are not from strong mathematical background. In the process, as per Dr. Feynman, also making sure I have understood them myself. As far as general mathematical and statistical requirements go, I would say that typical undergraduate level should suffice. Also, with proliferation and standardization of the machine learning libraries in open source domain, one does not need to go that deep into mathematical understanding of the theory to be able to implement the state-of-the-art machine learning models leading to state-of-the-art intelligent solutions.

One of the main sources of confusion that arises when trying to solve a problem in the given application is the choice of algorithm. Typically each algorithm presented here has originated from some specific problem, but the algorithm is typically not restricted to solving only that problem. However, choosing the right algorithm for given problem is not trivial even for a doctoral fellow with strong mathematical background. In order to separate these two areas, I have divided these two areas into separate parts altogether. This will make the topics much easier to access for the reader.

I would recommend the reader to start with Part **I**, and then choose Parts **II** or **III** depending on the needs. It will be ideal for a student to go sequentially through the book, while a newcomer to the area from professional background would be better suited to start with Part **III** to understand or focus on the precise application at hand and then delve into the details of the theory for the algorithms as needed in Part **II**. Parts **IV** and **V** should follow afterwards. I have added sufficient references between the two parts to make this transition smooth.

In my mind, unless one can see the models in action on real data that one can see and plot, the understanding is not complete. Hence, following the details of algorithms and applications, I have added another part to cover the basic implementation of the models using free and open source options. Completion of this part will enable the reader to tackle the real-world problems in AI with state-of-the-art ML techniques!

Redmond, WA, USA  
March 2019

Ameet V. Joshi

# Acknowledgments

I would like to use this opportunity to acknowledge the people who made significant contributions towards creation of the second edition of this book. It was certainly a daunting task to update on the first edition of the book that already was a huge success.

However, continuous support and encouragement from my wife, Meghana, and sons, Dhroov and Sushaan, really helped me continue with the efforts and ultimately complete the book. I would also like to thank my mother Madhuri, father Vijay, and brother Mandar for their continuous encouragement and support.

I would like to thank Mary James of Springer for the encouragement and support as I worked through the completion of the book. I would also like to thank Amrita, Zoe, Brian, and the whole team of Springer for their timely help with updates.

The present book is a furtherance of the efforts from the publication of the first edition towards unification of the disparate areas in the field of machine learning to produce artificially intelligent experiences. The book essentially stands on the pillars of this knowledge that was created by the numerous extraordinary scientists and brilliant mathematicians over several decades. So, I would like to thank them all as well.

Last but not the least, I would like to thank all the readers who provided me with valuable feedback and thousands of readers who used this book towards understanding the concepts in this area. Their critic and encouragement certainly helped me improve on the first edition of this book.

# Contents

## Part I Introduction

<b>1</b>	<b>Introduction to AI and ML</b> .....	3
1.1	Introduction .....	3
1.2	What Is AI.....	4
1.3	What Is ML .....	4
1.4	Organization of the Book .....	5
1.4.1	Introduction.....	5
1.4.2	Machine Learning .....	5
1.4.3	Building End-to-End Pipelines .....	6
1.4.4	Artificial Intelligence.....	6
1.4.5	Conclusion .....	6
<b>2</b>	<b>Essential Concepts in Artificial Intelligence and Machine Learning</b> .	7
2.1	Introduction .....	7
2.2	Big Data and Not-So-Big Data .....	7
2.2.1	What Is Big Data .....	7
2.2.2	Why Should We Treat Big Data Differently? .....	8
2.3	Machine Learning Algorithms, Models, and Types of Learning .	8
2.3.1	Supervised Learning .....	9
2.3.2	Unsupervised Learning .....	9
2.3.3	Reinforcement Learning .....	10
2.4	Machine Learning Methods Based on Time .....	10
2.4.1	Static Learning .....	10
2.4.2	Dynamic Learning .....	10
2.5	Dimensionality .....	11
2.5.1	Curse of Dimensionality .....	12
2.6	Linearity and Nonlinearity .....	13
2.7	Occam’s Razor .....	13
2.8	No Free Lunch Theorem.....	18
2.9	Law of Diminishing Returns .....	18
2.10	Early Trends in Machine Learning.....	18

- 2.10.1 Expert Systems ..... 19
- 2.11 Conclusion ..... 19
- 2.12 Exercise ..... 19
- 3 Data Understanding, Representation, and Visualization..... 21**
  - 3.1 Introduction ..... 21
  - 3.2 Understanding the Data..... 21
    - 3.2.1 Understanding Entities ..... 23
    - 3.2.2 Understanding Attributes ..... 23
    - 3.2.3 Understanding Data Types ..... 24
  - 3.3 Representation and Visualization of the Data ..... 24
    - 3.3.1 Principal Component Analysis ..... 25
    - 3.3.2 Linear Discriminant Analysis..... 27
  - 3.4 Conclusion ..... 29
- 4 Implementing Machine Learning Algorithms ..... 31**
  - 4.1 Introduction ..... 31
  - 4.2 Use of *Google Colab* ..... 31
    - 4.2.1 scikit-learn Python Library ..... 32
  - 4.3 Azure Machine Learning (AML) ..... 33
    - 4.3.1 How to Start with AML? ..... 33
  - 4.4 Conclusion ..... 40
  - 4.5 Exercises ..... 41

**Part II Machine Learning**

- 5 Linear Methods ..... 45**
  - 5.1 Introduction ..... 45
  - 5.2 Linear and Generalized Linear Models ..... 46
  - 5.3 Linear Regression ..... 46
    - 5.3.1 Defining the Problem ..... 46
    - 5.3.2 Solving the Problem ..... 47
  - 5.4 Example of Linear Regression ..... 47
  - 5.5 Regularized Linear Regression ..... 48
    - 5.5.1 Regularization ..... 48
    - 5.5.2 Ridge Regression ..... 49
    - 5.5.3 Lasso Regression ..... 50
  - 5.6 Generalized Linear Models (GLM) ..... 50
    - 5.6.1 Logistic Regression ..... 51
  - 5.7 K-Nearest Neighbor (KNN) Algorithm..... 52
    - 5.7.1 Definition of KNN ..... 52
    - 5.7.2 Classification and Regression..... 54
    - 5.7.3 Other Variations of KNN ..... 54
  - 5.8 Conclusion ..... 55
  - 5.9 Exercises ..... 55

<b>6</b>	<b>Perceptron and Neural Networks</b> .....	57
6.1	Introduction .....	57
6.1.1	Biological Neuron .....	57
6.2	Perceptron .....	58
6.2.1	Implementing Perceptron .....	59
6.3	Multilayered Perceptron or Artificial Neural Network .....	61
6.3.1	Feedforward Operation .....	61
6.3.2	Nonlinear MLP or Nonlinear ANN .....	62
6.3.3	Training MLP .....	64
6.3.4	Hidden Layers .....	65
6.3.5	Implementing MLP .....	65
6.4	Radial Basis Function Networks .....	66
6.4.1	Interpretation of RBF Networks .....	67
6.4.2	Implementing RBF Networks .....	68
6.5	Overfitting .....	69
6.5.1	Concept of Regularization .....	70
6.5.2	L1 and L2 Regularization .....	70
6.5.3	Dropout Regularization .....	71
6.6	Conclusion .....	71
6.7	Exercises .....	71
<b>7</b>	<b>Decision Trees</b> .....	73
7.1	Introduction .....	73
7.2	Why Decision Trees? .....	74
7.2.1	Types of Decision Trees .....	75
7.3	Algorithms for Building Decision Trees .....	75
7.4	Regression Tree .....	76
7.4.1	Implementing Regression Tree .....	77
7.5	Classification Tree .....	77
7.5.1	Defining the Terms .....	77
7.5.2	Misclassification Error .....	78
7.5.3	Gini Index .....	78
7.5.4	Cross-Entropy or Deviance .....	80
7.6	CHAID .....	81
7.6.1	CHAID Algorithm .....	81
7.7	Training Decision Tree .....	82
7.7.1	Depth of Decision Tree .....	82
7.8	Ensemble Decision Trees .....	83
7.8.1	Bagging Ensemble Trees .....	83
7.8.2	Random Forest Trees .....	84
7.8.3	Boosted Ensemble Trees .....	85
7.9	Implementing a Classification Tree .....	86
7.10	Conclusion .....	87
7.11	Exercises .....	87

- 8 Support Vector Machines** ..... 89
  - 8.1 Introduction ..... 89
  - 8.2 Motivation and Scope ..... 89
    - 8.2.1 Extension to Multi-class Classification ..... 90
  - 8.3 Theory of SVM ..... 91
  - 8.4 Separability and Margins ..... 93
    - 8.4.1 Use of Slack Variables ..... 93
  - 8.5 Implementing Linear SVMs ..... 94
  - 8.6 Nonlinearity and Use of Kernels ..... 96
    - 8.6.1 Radial Basis Function ..... 96
    - 8.6.2 Polynomial ..... 97
    - 8.6.3 Sigmoid ..... 97
  - 8.7 Implementing Nonlinear SVMs with Kernels ..... 97
  - 8.8 Risk Minimization ..... 98
  - 8.9 Conclusion ..... 99
  - 8.10 Exercises ..... 99
- 9 Probabilistic Models** ..... 101
  - 9.1 Introduction ..... 101
  - 9.2 Discriminative Models ..... 102
    - 9.2.1 Maximum Likelihood Estimation ..... 102
    - 9.2.2 Bayesian Approach ..... 103
    - 9.2.3 Comparison of MLE and Bayesian Approach ..... 104
  - 9.3 Implementing Probabilistic Models ..... 107
  - 9.4 Generative Models ..... 108
    - 9.4.1 Mixture Models ..... 108
    - 9.4.2 Bayesian Networks ..... 108
  - 9.5 Some Useful Probability Distributions ..... 109
    - 9.5.1 Normal or Gaussian Distribution ..... 110
    - 9.5.2 Bernoulli Distribution ..... 112
    - 9.5.3 Binomial Distribution ..... 113
    - 9.5.4 Gamma Distribution ..... 114
    - 9.5.5 Poisson Distribution ..... 116
  - 9.6 Conclusion ..... 117
  - 9.7 Exercises ..... 118
- 10 Dynamic Programming and Reinforcement Learning** ..... 119
  - 10.1 Introduction ..... 119
  - 10.2 Fundamental Equation of Dynamic Programming ..... 119
  - 10.3 Classes of Problems Under Dynamic Programming ..... 121
  - 10.4 Reinforcement Learning ..... 121
    - 10.4.1 Characteristics of Reinforcement Learning ..... 121
    - 10.4.2 Framework and Algorithm ..... 122
  - 10.5 Exploration and Exploitation ..... 122
  - 10.6 Applications of Reinforcement Learning ..... 124

- 10.7 Theory of Reinforcement Learning ..... 125
  - 10.7.1 Variations in Learning ..... 126
- 10.8 Implementing Reinforcement Learning ..... 126
- 10.9 Conclusion ..... 127
- 10.10 Exercises ..... 127
- 11 Evolutionary Algorithms ..... 129**
  - 11.1 Introduction ..... 129
  - 11.2 Bottleneck with Traditional Methods ..... 129
  - 11.3 Darwin’s Theory of Evolution ..... 130
  - 11.4 Genetic Programming ..... 132
    - 11.4.1 Implementing Genetic Programming ..... 134
  - 11.5 Swarm Intelligence ..... 134
  - 11.6 Ant Colony Optimization ..... 136
  - 11.7 Simulated Annealing ..... 137
  - 11.8 Conclusion ..... 138
  - 11.9 Exercises ..... 138
- 12 Time Series Models ..... 139**
  - 12.1 Introduction ..... 139
  - 12.2 Stationarity ..... 140
  - 12.3 Autoregressive Moving Average Models ..... 141
    - 12.3.1 Autoregressive (AR) Process ..... 142
    - 12.3.2 Moving Average (MA) Process ..... 142
    - 12.3.3 Autoregressive Moving Average (ARMA) Process ..... 143
  - 12.4 Autoregressive Integrated Moving Average (ARIMA) Models .. 143
  - 12.5 Implementing AR, MA, ARMA, and ARIMA in Python ..... 144
  - 12.6 Hidden Markov Models (HMMs) ..... 145
    - 12.6.1 Applications ..... 146
  - 12.7 Conditional Random Fields (CRFs) ..... 146
  - 12.8 Conclusion ..... 147
  - 12.9 Exercises ..... 148
- 13 Deep Learning ..... 149**
  - 13.1 Introduction ..... 149
  - 13.2 Why Deep Neural Networks? ..... 151
  - 13.3 Types of Deep Neural Networks ..... 152
  - 13.4 Convolutional Neural Networks (CNNs) ..... 152
    - 13.4.1 One-Dimensional Convolution ..... 152
    - 13.4.2 Two-Dimensional Convolution ..... 153
    - 13.4.3 Architecture of CNN ..... 154
    - 13.4.4 Training CNN ..... 156
    - 13.4.5 Applications of CNN ..... 157
  - 13.5 Recurrent Neural Networks (RNNs) ..... 157
    - 13.5.1 Limitation of RNN ..... 158
    - 13.5.2 Long Short-Term Memory RNN ..... 158

- 13.5.3 Advantages of LSTM ..... 160
- 13.5.4 Applications of LSTM-RNN ..... 160
- 13.6 Attention-Based Networks ..... 160
- 13.7 Generative Adversarial Networks (GANs) ..... 161
- 13.8 Implementing Deep Learning Models ..... 162
- 13.9 Conclusion ..... 169
- 13.10 Exercises ..... 169
- 14 Unsupervised Learning ..... 171**
  - 14.1 Introduction ..... 171
  - 14.2 Clustering ..... 172
    - 14.2.1 k-Means Clustering ..... 172
    - 14.2.2 Improvements to k-Means Clustering ..... 174
    - 14.2.3 Implementing k-Means Clustering ..... 176
  - 14.3 Component Analysis ..... 177
    - 14.3.1 Independent Component Analysis (ICA) ..... 177
  - 14.4 Self-Organizing maps (SOMs) ..... 178
  - 14.5 Autoencoding Neural Networks ..... 178
  - 14.6 Conclusion ..... 180
  - 14.7 Exercises ..... 180
- Part III Building End-to-End Pipelines**
- 15 Featurization ..... 183**
  - 15.1 Introduction ..... 183
  - 15.2 UCI: Adult Salary Predictor ..... 183
    - 15.2.1 Feature Details ..... 184
  - 15.3 Identifying the Raw Data: Separating Information from Noise .. 185
    - 15.3.1 Correlation and Causality ..... 185
  - 15.4 Building Feature Set ..... 186
    - 15.4.1 Standard Options of Feature Building ..... 187
    - 15.4.2 Custom Options of Feature Building ..... 192
  - 15.5 Handling Missing Values ..... 193
  - 15.6 Visualizing the Features ..... 193
    - 15.6.1 Numeric Features ..... 194
    - 15.6.2 Categorical Features ..... 194
  - 15.7 Conclusion ..... 200
  - 15.8 Exercises ..... 200
- 16 Designing and Tuning Model Pipelines ..... 201**
  - 16.1 Introduction ..... 201
  - 16.2 Choosing the Technique or Algorithm ..... 201
    - 16.2.1 Choosing Technique for Adult Salary Classification .... 202
  - 16.3 Splitting the Data ..... 202
    - 16.3.1 Stratified Sampling ..... 204
  - 16.4 Training ..... 205



- 16.4.1 Tuning the Hyperparameters..... 205
- 16.4.2 Cross-Validation..... 206
- 16.5 Implementing Machine Learning Pipeline..... 206
- 16.6 Accuracy Measurement..... 209
- 16.7 Explainability of Features ..... 209
- 16.8 Practical Considerations ..... 210
  - 16.8.1 Data Leakage ..... 210
  - 16.8.2 Coincidence and Causality ..... 211
- 16.9 Conclusion ..... 212
- 16.10 Exercises ..... 212
- 17 Performance Measurement..... 213**
  - 17.1 Introduction ..... 213
  - 17.2 Sample Size ..... 213
  - 17.3 Metrics Based on Numerical Error..... 214
    - 17.3.1 Mean Absolute Error..... 214
    - 17.3.2 Mean Squared Error..... 215
    - 17.3.3 Root Mean Squared Error..... 215
    - 17.3.4 Normalized Error ..... 215
  - 17.4 Metrics Based on Categorical Error..... 215
    - 17.4.1 Accuracy..... 216
    - 17.4.2 Precision and Recall..... 216
    - 17.4.3 Receiver Operating Characteristic (ROC) Curve Analysis ..... 218
    - 17.4.4 Precision-Recall Curve..... 219
  - 17.5 Hypothesis Testing ..... 219
    - 17.5.1 Background..... 220
    - 17.5.2 Steps in Hypothesis Testing ..... 221
    - 17.5.3 A/B Testing..... 221
  - 17.6 Conclusion ..... 221
  - 17.7 Exercises ..... 222

**Part IV Artificial Intelligence**

- 18 Classification..... 225**
  - 18.1 Introduction ..... 225
  - 18.2 Examples of Real-World Problems in Classification ..... 225
  - 18.3 Spam Email Detection ..... 226
    - 18.3.1 Defining Scope ..... 226
    - 18.3.2 Assumptions ..... 227
    - 18.3.3 Skew in the Data ..... 228
    - 18.3.4 Supervised Learning ..... 229
    - 18.3.5 Feature Engineering..... 229
  - 18.4 Implementing Spam Filter Classifier..... 229
    - 18.4.1 Using Azure ML ..... 230

18.5	Conclusion .....	233
18.6	Exercises .....	234
<b>19</b>	<b>Regression .....</b>	<b>235</b>
19.1	Introduction .....	235
19.2	Examples of Real-World Problems .....	235
19.3	Predicting Real Estate Prices .....	236
	19.3.1 Defining Regression-Specific Problem .....	236
	19.3.2 Aspects in Real Estate Value Prediction .....	236
	19.3.3 Gather Labelled Data .....	237
19.4	Implementing Regression .....	237
	19.4.1 Model Performance .....	239
19.5	Other Applications of Regression .....	242
19.6	Conclusion .....	242
19.7	Exercises .....	242
<b>20</b>	<b>Ranking .....</b>	<b>243</b>
20.1	Introduction .....	243
20.2	Measuring Ranking Performance .....	244
20.3	Ranking Search Results and Google's PageRank .....	246
20.4	Techniques Used in Text-Based Ranking Systems .....	246
	20.4.1 Keyword Identification/Extraction .....	246
20.5	Implementing Keyword Extraction .....	248
20.6	Implementing Ranking System .....	249
20.7	Conclusion .....	249
20.8	Exercises .....	250
<b>21</b>	<b>Recommendation Systems .....</b>	<b>251</b>
21.1	Introduction .....	251
21.2	Collaborative Filtering .....	252
	21.2.1 Solution Approaches .....	252
	21.2.2 Information Types .....	252
	21.2.3 Algorithm Types .....	253
21.3	Amazon's Personal Shopping Experience .....	254
	21.3.1 Context-Based Recommendation .....	254
	21.3.2 Aspects Guiding the Context-Based Recommendations .....	255
	21.3.3 Personalization-Based Recommendation .....	255
21.4	Netflix's Streaming Video Recommendations .....	256
21.5	Implementing Recommendation System .....	256
	21.5.1 MovieLens Data .....	256
	21.5.2 Planning the Recommendation Approach .....	258
	21.5.3 Implementing Recommendation System .....	259
21.6	Conclusion .....	260
21.7	Exercises .....	260

**Part V Conclusion**

<b>22 Conclusion and Next Steps</b> .....	263
22.1 Overview .....	263
<b>References</b> .....	265
<b>Index</b> .....	269

# Part I

## Introduction

This is your last chance. After this, there is no turning back. You take the blue pill - the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill - you stay in Wonderland and I show you how deep the rabbit-hole goes. The question is, which pill would you choose?

—Morpheus, “The Matrix”

### **Part Synopsis**

This part introduces the concepts of artificial intelligence (AI) and machine learning (ML) in the modern context. Various scenarios where these concepts are used will also be discussed. Going further, this part also discusses the understanding, representation, and visualization aspects of data that form the foundation to the next topics. This will serve as a good starting point to delve into the details of the techniques, applications, and implementations.

# Chapter 1

## Introduction to AI and ML



### 1.1 Introduction

It is an understatement to say that the field of artificial intelligence and machine learning has exploded in the last decade. There are some rather valid reasons for the exponential growth and some not so much. As a result of this growth and popularity, they are also some of the most wrongly used words. Along with the enormous growth of the field, it has also become one of the most confusing fields to understand the scope of. The very definitions of the terms AI and ML have also become diluted and vague.

We are not going to try and define these entities with word play, but rather are going to learn the context around the origins and scope of these entities. This will make their meaning apparent. The roots of these words originate from multiple disciplines and not just computer science. These disciplines include pure mathematics, electrical engineering, statistics, signal processing, and communications along with computer science to name the top few. I cannot imagine any other area that emerges as a conflation of such wide variety of disciplines. Along with the wide variety of origins, the field also finds applications in even greater number of industries ranging from high-tech applications like image processing and natural language processing to online shopping to nondestructive testing of nuclear power plants and so on.

In spite of being such multidisciplinary, there are several things that we can conceptually learn collectively and obtain clarity of its scope. That is the primary objective behind this book. I want to make the field accessible to newcomers to the field in the form of graduate-level students to engineers and professionals who are entering the field or even actively working in this field.

## 1.2 What Is AI

Alan Turing defined artificial intelligence as follows: “If there is a machine behind a curtain and a human is interacting with it (by whatever means, e.g. audio or via typing etc.) and if the human feels like he /she is interacting with another human, then the machine is artificially intelligent.” This is quite a unique way to define AI. It does not directly aim at the notion of intelligence, but rather focusses on humanlike behavior. As a matter of fact, this objective is even broader in scope than mere intelligence. From this perspective, AI does not mean building an extraordinarily intelligent machine that can solve any problem in no time, but rather it means to build a machine that is capable of humanlike behavior. However, just building machines that mimic humans does not sound very interesting. As per modern perspective, whenever we speak of AI, we mean machines that are capable of performing one or more of these tasks: understanding human language, performing mechanical tasks involving complex maneuvering, solving computer-based complex problems possibly involving large data in a very short time and reverting back with answers in humanlike manner, etc.

The supercomputer depicted in movie 2001: A Space Odyssey, called HAL, represents very closely the modern view of AI. It is a machine that is capable of processing a large amount of data coming from various sources and generating insights and summary of it at extremely fast speed and is capable of conveying these results to humans in humanlike interaction, e.g., voice conversation.

There are two aspects to AI as viewed from humanlike behavior standpoint. One is where the machine is intelligent and is capable of communication with humans, but does not have any locomotive aspects. HAL is example of such AI. The other aspect involves having physical interactions with humanlike locomotion capabilities, which refers to the field of robotics. For the purpose of this book, we are only going to deal with the first kind of AI.

## 1.3 What Is ML

The term “machine learning” or ML in short was coined in 1959 by Arthur Samuel in the context of solving game of checkers by machine. The term refers to a computer program that can learn to produce a behavior that is not explicitly programmed by the author of the program. Rather it is capable of showing behavior that the author may be completely unaware of. This behavior is learned based on three factors: (1) data that is consumed by the program, (2) a metric that quantifies the error or some form of distance between the current behavior and ideal behavior, and (3) a feedback mechanism that uses the quantified error to guide the program to produce better behavior in the subsequent events. As can be seen, the second and third factors quickly make the concept abstract and emphasize deep mathematical roots of it. The methods in machine learning theory are essential in building artificially intelligent systems.

## 1.4 Organization of the Book

I have divided this book into six parts, as described below.

### Parts of the Book

1. Introduction
2. Machine learning
3. Building end-to-end pipelines
4. Artificial intelligence
5. Conclusion

### 1.4.1 Introduction

The first part of the book introduces the key concepts in the field as well as outlines the scope of the field. There are multiple aspects discussed in these chapters that may appear disconnected, but put together they are all extremely important in holistic understanding of the field and inspiring the confidence. This part also discusses the preliminary understanding and processing of the raw data as is typically observed. This sets the foundation for the reader to understand the subsequent parts.

### 1.4.2 Machine Learning

The second part of the book deals with the theoretical foundation of machine learning. In this part we will study various algorithms, as well as their origins and their applications. Machine learning has united a vast array of algorithms that find their origins in fields ranging from electrical engineering, signal processing, statistics, financial analysis, genetic sciences, and so on. All these algorithms are primarily developed from the principles of pure mathematics and statistics, in spite of being originated in fundamentally different areas. Along with the roots, they also have one more thing in common: use of computers to automate complex computations. These computations ultimately lead to solving problems that seem so hard that one would believe that they are solved by some intelligent entity or *artificial intelligence*.

We are also going to learn how to implement the concepts learnt in each of the chapter using open-source Python libraries. Python has become de facto standard for implementing the machine learning algorithms due to support from the community as well as from commercial platforms from companies like Amazon, Google, and Microsoft. Also most of the cutting-edge research happening in the field is readily accessible in the form of open-source libraries in Python. We will use Google's AI platform to implement the code, due to its simple setup and easy access to all the

libraries. However, I will outline the setup process for Microsoft's platform and use that for some implementations as well.

### ***1.4.3 Building End-to-End Pipelines***

The third part is dedicated to building end-to-end pipelines for solving real-life problems using the techniques learned in the previous section. Real-life problems always pose unique challenges that need to be addressed in custom manner. However, there are certain broader steps one can take to tackle most of the problems, and this section focusses on these aspects of ML and AI. This section really ties together theory and practice to give readers a glimpse into the real world.

### ***1.4.4 Artificial Intelligence***

This is likely the most interesting part of the book that directly deals with problems that machine learning has solved and we experience in our day-to-day life. These applications represent real problems that have been solved in a variety of different industries, e.g., face recognition requirements for national security, detecting spam emails for e-commerce providers, deploying drones to carry out tactical jobs in areas where situations are hazardous for humans, etc. These applications need help from the rich collection of machine learning techniques that are described in the previous part. It is important to separate the applications and techniques, as there is a nontrivial overlap between those areas. There are some techniques specifically developed for certain types of applications, but there are some applications that can be benefitted from a variety of different techniques, and the relationship between them is many to many. Only when we start looking at the field from these angles separately that a lot of confusion starts to melt away and things start to make sense. The seamless solution of these problems and their delivery to millions of people are what create the feel of artificial intelligence.

### ***1.4.5 Conclusion***

In this last part, we summarize the learnings and discuss the next steps for the reader. After completion of this book, the reader is well equipped to embark on solving real-world problems that are still unsolved and make the world a better place.



# Chapter 2

## Essential Concepts in Artificial Intelligence and Machine Learning



### 2.1 Introduction

There are various concepts, mostly originated in mathematics and optimization theory, that are spread over disparate topics and are rather difficult to categorize with respect to techniques or applications or implementations. However, they are at the heart of the machine learning theory and its applications and need special attention. We will cover these topics in this chapter. We will keep revising them multiple times later in the book, but aggregating them here will make the reader more equipped to study the subsequent chapters.

### 2.2 Big Data and Not-So-Big Data

#### 2.2.1 What Is Big Data

The area of *big data* has become a very interesting and exciting charter in the field of AI and ML since the exponential growth in computation hardware, network bandwidth, and cloud computing. Although there is no specific threshold above which it is called as big data, the generally accepted definition is as follows: When the size of data is large enough such that it cannot be processed on a single machine within reasonable time frame, it is called as big data. Based on the current (2021 AD) generation of computers, this equates to something roughly more than 10–50 GB. This size is primarily linked with the volatile memory or RAM of the computer. From there it can go to hundreds of petabytes (1 petabyte is 1000 terabytes and 1 terabyte is 1000 gigabytes) and more. When the data size is more than what can fit in the RAM, then processing becomes exponentially slow as the computer has to start caching the RAM to hard drive, and the process is extremely inefficient. Hence, in such cases a cluster of machined (computers) is used, and the data is spread over

the RAM of all the machines in the cluster. By adding arbitrarily more machines to a cluster, required data sizes can be supported.

### 2.2.2 *Why Should We Treat Big Data Differently?*

Although the big data is separated from the not-so-big-data by simple size restrictions, the handling of the data changes drastically when we move from not-so-big data to big data. For processing not-so-big data, one need not worry about the location where each bit of data is stored. All the data can be loaded into the memory of a single machine and can be accessed without additional overhead, and all the typical numeric or string operations on any chunk of data can be executed in predictable time. However, once we enter the realm of big data, all such bets are off. Instead of a single machine, we are dealing with a cluster of machines. Each machine has its own storage and computational resources. For all the numeric and string operations, one must carefully split the data into different machines, such that all the operands in individual operation are available on the same machine. If that is not the case, there is additional overhead of moving the data across the machines, and it is typically a very expensive and inefficient operation. Iterative operations especially do not scale well on big data. Thus, when dealing with big data, one must also carefully design the storage along with the operations.

Hadoop [22] or Hadoop Distributed File System (HDFS) (which finds its roots in Google File System [90]) was one of the first architecture that addressed most of the concepts of handling big data. Most principles from this architecture are still in use today in the form of various different offerings from Amazon, Microsoft, Google, etc.

## 2.3 Machine Learning Algorithms, Models, and Types of Learning

The machine learning algorithms are essentially abstract mathematical constructs. In order to understand the concept of algorithm, let's take a simple example of a mathematical function,  $f(x, a) = ax^2$ . Here  $x$  is an input and  $f(x, a)$  is the output. It is expected that input will be available for the algorithm to predict the output, but the value of  $a$  is still unknown and is specific to the given algorithm. The way in which the value of  $a$  is calculated differs for different types of algorithms. Also, the above example is given to illustrate the concept only. Typically an algorithm can contain tens or hundreds or even thousands of parameters involved in a series of complex operations. Based on the differences in learning the parameters of algorithms, they are broadly classified into three types:

1. Supervised learning algorithms
2. Unsupervised learning algorithms
3. Reinforcement learning algorithms

The learning algorithms typically go through two steps, training and scoring, although in some cases the training step can be missing. In training step, the algorithms learn the parameters, and in scoring step, it generates the output or scores. An algorithm that has gone through the training step is called as model. For simplicity, let's consider a machined learnt model as a black box that produces some output when given some input.

### ***2.3.1 Supervised Learning***

If we have a set of data, called as training data, that contains a set of expected outputs (also called as labels) for a set of inputs. The objective is to train the ML system to predict the labels when a similar input is presented to it. The learning based on such data is called as supervised learning. A common example of supervised learning is a classification system. Let's say we have a set of measurements of four different properties (sepal length, sepal width, petal length, and petal width) for hundreds of different flowers. All these flowers come from three fundamental types: Setosa, Versicolor, and Virginica [3]. We know the type of flower for sample measurement in our data. This data would then serve as training data that can be used to train the model. Once the model is trained, we can use the black box of the model to classify any flower (between the three known types) based on the sepal and petal measurements. We will study supervised classification in Chap. 18. However, there are many different types of supervised learning methods that we will study in multiple chapters.

### ***2.3.2 Unsupervised Learning***

In unsupervised learning paradigm, the labelled data is not available. A classic example of unsupervised learning is "clustering." Consider the same example as described in the previous subsection, where we have measurements of the sepal and petal dimensions of three types of flowers. However, in this case, we don't have exact names of the flowers for each set of measurements. All we have is set of measurements. However, we are told that these measurements belong to flowers of three different types. In such cases, one can use unsupervised learning techniques to automatically identify three clusters from the measurements. However, as the labels are not known, all we can do is call each cluster as *flower-type-1*, *flower-type-2*, and *flower-type-3*. If a new set of measurement is given, we can find the cluster to which they belong and classify them into one of them. We will study unsupervised learning in Chap. 14.

### ***2.3.3 Reinforcement Learning***

Reinforcement learning is a special type of learning method that needs to be treated separately from supervised and unsupervised methods. Reinforcement learning involves feedback from the environment; hence, it is not exactly unsupervised. However, it does not have a set of labelled samples available for training as well, and hence it cannot be treated as supervised in traditional sense. In reinforcement learning methods, the system is continuously interacting with the environment in search of producing desired behavior and is getting feedback from the environment. The learning can be stopped at any point to observe the learned behavior. This type of learning resembles the most to human learning. From the moment a human baby is born, it is continuously going through reinforcement learning by interacting with the environment. We will study this technique in greater detail in Chap. 10.

## **2.4 Machine Learning Methods Based on Time**

Another way to slice the machine learning methods is to classify them based on the type of data that they deal with. The systems that take in static labelled data are called static learning methods. The systems that deal with data that is continuously changing with time are called dynamic methods. Each type of methods can be supervised or unsupervised; however, reinforcement learning methods are always dynamic.

### ***2.4.1 Static Learning***

Static learning refers to learning on the data that is taken as a single snapshot, and the properties of the data remain constant over time. Once the model is trained on the data (either using supervised or unsupervised learning), the trained model can be applied to similar data anytime in the future, and the model will still be valid and will perform as expected. Typical examples of this would be image classification of different cats and dogs.

### ***2.4.2 Dynamic Learning***

This is also called as time series-based learning. The data in this type of problems is time sensitive and keeps changing over time. Hence, the model training is not a static process, but the model needs to be trained continuously (or after every reasonable time window) to remain effective. A typical example of such problems

is weather forecasting or stock market predictions. A model trained a year back will be completely useless to predict the weather for tomorrow or predict the price of any stock for tomorrow. The fundamental difference in the static and dynamic learning types is the notion of state. In static models, the state of the model never changes, while in dynamic models the state of the model is a function of time and keeps changing.

## 2.5 Dimensionality

The word dimension arises from physics, where there are only three spatial dimensions in the universe that we live in. If you factor in time, it becomes four dimensional, but the time dimension has different characteristics and is not the same as spatial dimension. The fundamental rule that dictates the maximum number of dimensions is orthogonality. Orthogonality also means independence. When we say that our known universe is three dimensional, there are three dimensions that are independent of each other. If you travel in one dimension, then your coordinates from the other dimensions do not change. If we consider two-dimensional space, we can draw two lines that are perpendicular or orthogonal to each other. If we draw a third line, it can either be parallel to the previous two lines, or it will intersect both of them at an angle less than 90 degrees. Such line can be represented as linear combination of the previous two lines we learn in basic algebra and graph theory. Extending this concept to three-dimensional space, we can have only three lines that are mutually orthogonal. Any fourth line or onwards can either be parallel to the earlier lines or can be expressed as a linear combination of them. As discussed earlier, in our known universe, we have only three dimensions, and as a result it is very difficult to image the fourth or higher dimensions. However, extending the mathematical properties of three-dimensional space, one can easily create equations with higher dimensions. The primary rule being each new dimension is orthogonal to all the previous dimensions at the same time. This rule has another important outcome: each point in the space can be uniquely represented with a combination of three coordinates. There can be no ambiguity. No two points apart from each other can have the same set of coordinates.

This concept is widely applicable in the theory of machine learning. We can obtain data with different attributes. If each attribute is independent of the others, then they can be considered as orthogonal dimensions. For example, consider the properties of trees: height, thickness of stem, width, length of leaves, number of branches, and so on. All these properties are independent of each other. If we measure these attributes for hundreds of trees, we will get a data that will look similar to what is shown in Table 2.1.

As, each column is a new dimension, this is essentially a five-dimensional data. It is quite easy to express in tabular form, and process mathematically using equations, but it is extremely difficult to visualize it. Our brain typically has a very hard time imagining anything more than three-dimensional. We cannot even plot it in any

**Table 2.1** Sample measurements of trees in five dimensions

Height (m)	Thickness of stem (m)	Width (m)	Length of leaves (in)	Number of branches
10	2.0	0.8	8	45
12	1.8	0.6	9	48
15	1.9	0.7	8	38
8	2.3	0.8	6	54
9	2.2	1.1	6	48
12	1.3	1.0	6	44

practical way that will make sense. It is not uncommon to have tens if not hundreds and more dimensions when we deal with data for machine learning.

### 2.5.1 *Curse of Dimensionality*

Adding arbitrarily a large number of dimensions is perfectly fine from mathematical standpoint, and as long as the machine can handle the size of the data, it is also fine from processing standpoint. However, the thing that is not trivial is: with increase in dimensions, the density of the data gets diminished exponentially. Let's consider a concrete example to illustrate this: if we have 1000 data points in training data, and data has 3 unique features. Each data maps to a single value as label. Thus, dimensionality of input is 3, and dimensionality of output is 1. Let's say the values of all the features are within 1–10. Thus all these 1000 data points lie in a cube of size 10 X 10 X 10. Thus, the density is 1000/1000 or 1 sample per unit cube. If instead we had 5 unique features instead of 3, then the density of the same data quickly drops to 0.01 sample per unit 5-dimensional cube. If we add more and more dimensions, the density will reduce exponentially, and we will be essentially left with mostly empty space. It is important to remember that the density of the output space is still 1. The density of the data is rather important from the processing perspective. With high-density data, most of the feature space is occupied by the data, we know what is the expected outcome in output space for the corresponding point in the input space. In other words, we can have a high confidence in the model built using such data. When the density drops, there are lots of empty spaces in the input space for which we don't know the expected output. As a result, the confidence in the accuracy of the model drops exponentially. Hence, although high dimensions are acceptable mathematically, unless we have proportionate number of training samples, it is not good for building a model. Hence, one needs to be careful with the dimensionality of the data.

## 2.6 Linearity and Nonlinearity

The concept of linearity and nonlinearity is applicable to both the data and the model built on top of it. However, the concept of linearity differs in each context. Data is called as linear if the relationship between the input and output is linear. To put this simply, when the value of input increases, the value of output also increases and vice versa. Pure inverse relation is also called as linear and would follow the rule of reversal of sign for either input or output. Figure 2.1 shows various possible linear relationships between input and output.

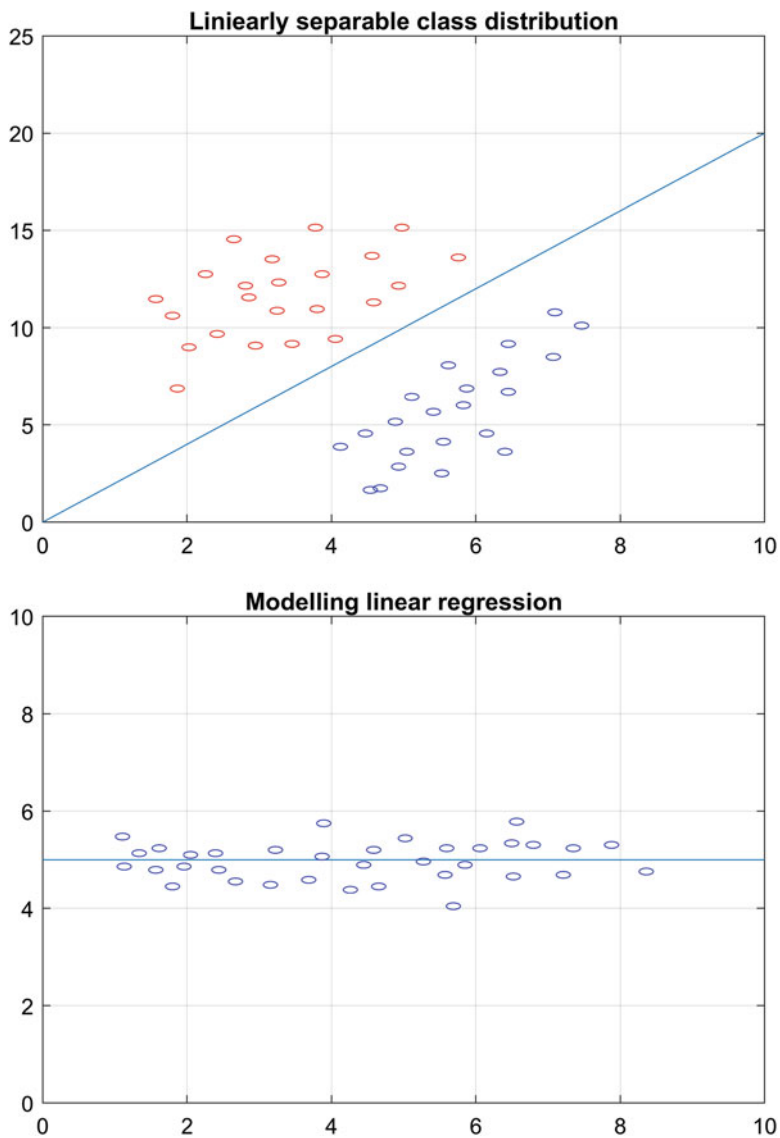
Linear models have a slightly more involved definition. All the models that use linear equations to model the relationship between input and output are called as linear models. However, sometimes, by preconditioning the input or output, a nonlinear relationship between the data can be converted into linear relationship, and then the linear model can be applied on it, e.g., if input and output are related with exponential relationship as  $y = 5e^x$ . The data is clearly nonlinear. However, instead of building the model on original data, we can build a model after applying a *log* operation. This operation transforms the original nonlinear relationship into a linear one as  $\log y = \log(5) + x$ . Then we build the linear model to predict  $\log y$  instead of  $y$ , which can then be converted to  $y$  by taking exponent. There can also be cases where a problem can be broken down into multiple parts and linear model can be applied to each part to ultimately solve a nonlinear problem. Figures 2.2, 2.3 and 2.4 show examples of converted linear, piecewise linear and nonlinear relationships respectively. In some cases the relationship is purely nonlinear and needs a proper nonlinear model to map it. Figure 2.4 shows examples of pure nonlinear relationships.

Linear models are the simplest to understand, build, and interpret. Our brain is highly tuned for linear models, as most of our experiences tend to have linear trends. Most times, what we call as intuitive behavior is mathematically a linear behavior. All the models in the theory of machine learning can handle linear data. Examples of purely linear models are linear regression, support vector machines without nonlinear kernels, etc. Nonlinear models inherently use some nonlinear functions to approximate the nonlinear characteristics of the data. Examples of nonlinear models include neural networks, decision trees, probabilistic models based on nonlinear distributions, etc.

In analyzing data for building the artificially intelligent system, determining the type of model to use is a critical starting step, and knowledge of linearity of the relationship is a crucial component of this analysis.

## 2.7 Occam's Razor

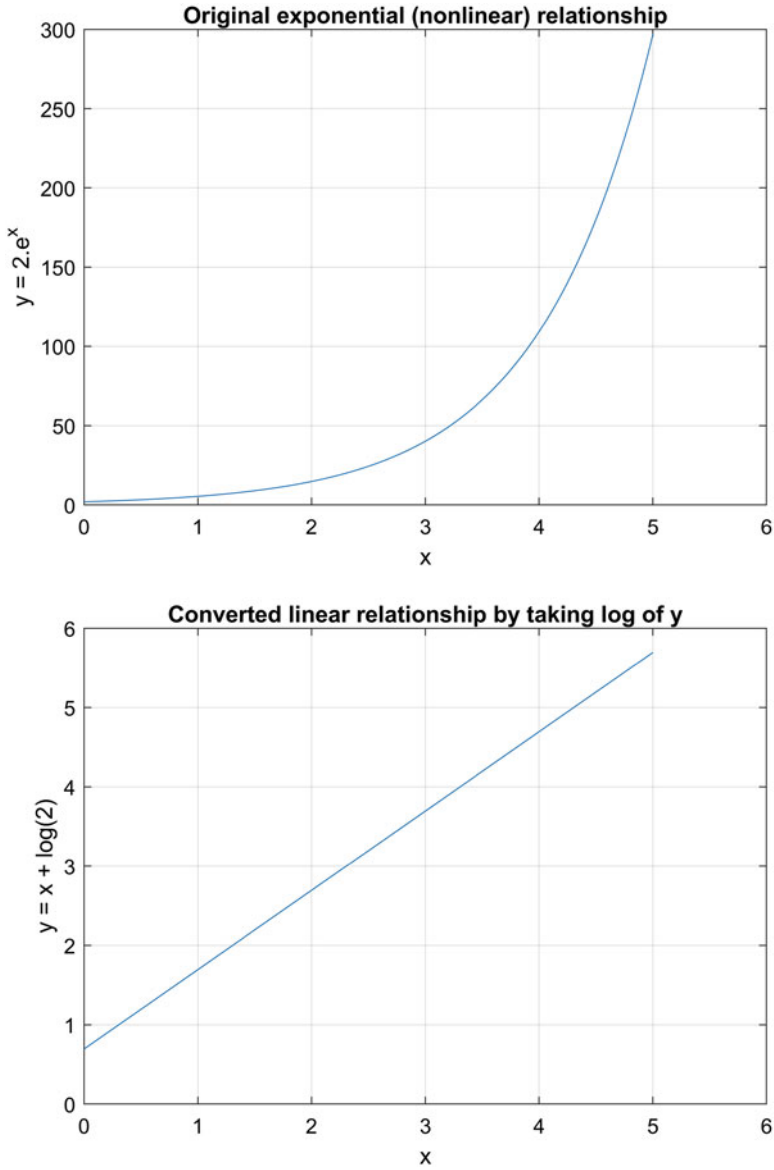
In developing and applying machine learning models, one always comes across multiple possible solutions and multiple possible approaches to get the answer.



**Fig. 2.1** Examples of linear relationships between input and output

Many times there is no theoretical guideline as to which solution or which approach is better than the rest. In such cases the concept of *Occam's razor*, which is also sometimes called as *principle of parsimony*, can be effectively applied. The principle states:





**Fig. 2.2** Example of nonlinear relationship between input and output being converted into linear relationship by applying logarithm

**Definition 2.1** Occam's Razor One should not make more assumptions than the minimum needed; or in other words, when there are multiple solutions for a given problem, the simplest approach is the best.

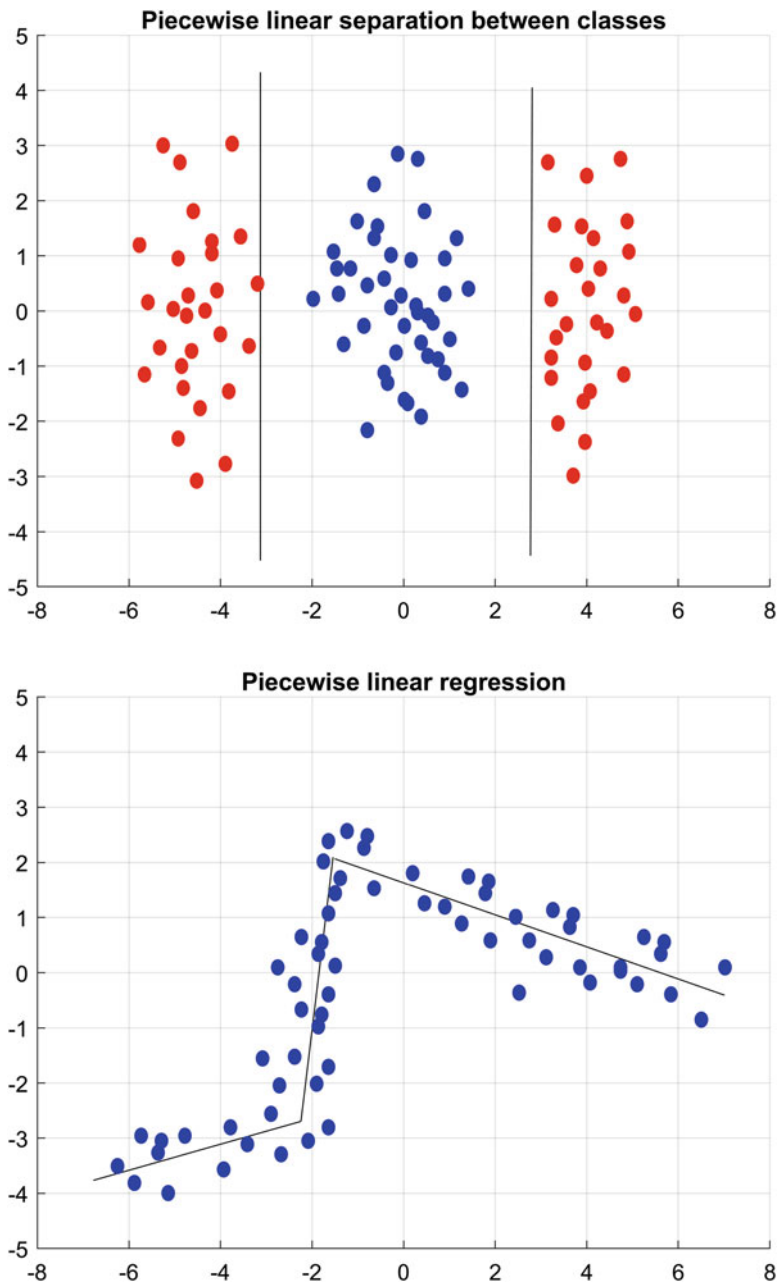


Fig. 2.3 Examples of piecewise linear relationships between input and output

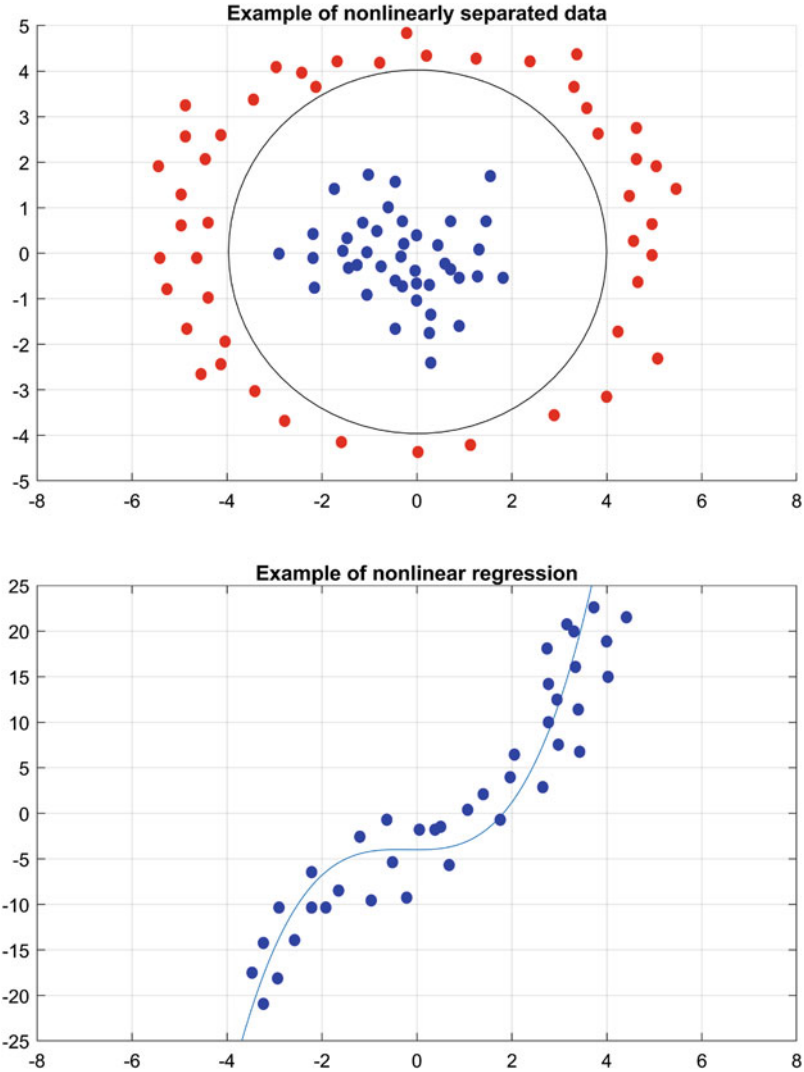


Fig. 2.4 Examples of pure nonlinear relationships between input and output

This principle is not quite a theorem and cannot be applied as a quantitative rule or equation. It is also stated without proof and more of a rule of thumb and may not apply in all the possible cases. However, it stands as a strong effective conceptual guideline when making such decisions in real life. It is also important to note that this rule creates a form of tradeoff, where on one side we have more information in the form of more complexity and on the other hand less information in the form of simplicity. One should not oversimplify the problem such that some of the core

information is lost. Another derived aspect of Occam's razor is: simpler solutions tend to have more generalization capabilities.

## 2.8 No Free Lunch Theorem

Another interesting concept that is good to be aware of when designing a machine learning system comes from a paper by Wolpert and Macready [79], in the form of no free lunch theorem or *NFL* theorem in optimization. The theorem essentially states that:

**Definition 2.2** NFL Theorem If an algorithm performs better on certain class of problems, then it pays for it in the form of degraded performance in other classes of problems. In other words: you cannot have a single optimal solution for all classes of problems.

Although this is stated as a theorem, it needs to be taken more of a guideline than a law, as it is quite possible to have one well-designed algorithm outperform some other not so well-designed algorithms in all the possible classes of problems. However, in practical situations, what we can infer from this theorem is that: there is no silver bullet solution that solves all the problems. Depending on the problem at hand, we need to design a custom solution if we want an optimal solution.

## 2.9 Law of Diminishing Returns

The law of diminishing returns is typically encountered in economics and business scenarios, where it states that adding more headcounts to complete a job starts to yield less and less returns with increase in existing number of headcount [9]. From the machine learning perspective, this law can be applied with respect to feature engineering. From a given set of data, one can only extract a certain number of features, after which the gains in performance start to diminish and the efforts are not worth the effect. In some ways it aligns with Occam's razor and adds more details.

## 2.10 Early Trends in Machine Learning

Before the machine learning started off commercially in the true sense, there were few implementations that were already pushing the boundary of routine computation. One such notable application was *expert systems*. It is useful to know what it meant at the time.

### 2.10.1 Expert Systems

The definition by Alan Turin marks the beginning of the era where machine intelligence was recognized, and with that the field of AI was born. However, in the early days (all the way till 1980s), the field of machine intelligence or machine learning was limited to what were called as *expert systems* or *knowledge-based systems*. One of the leading experts in the field of expert systems, Dr. Edward Feigenbaum, once defined expert system as,

**Definition 2.3** Expert Systems An intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for the solution [58].

Such systems were capable of replacing experts in certain areas. These machines were programmed to perform complex heuristic tasks based on elaborate logical operations. In spite of being able to replace the humans who are experts in the specific areas, these systems were not “intelligent” in the true sense, if we compare them with human intelligence. The reason being the systems were “hard-coded” to solve only a specific type of problem and if there is need to solve a simpler but completely different problem, these systems would quickly become completely useless. Nonetheless, these systems were quite popular and successful specifically in areas where repeated but highly accurate performance was needed, e.g., diagnosis, inspection, monitoring, and control [58].

## 2.11 Conclusion

In this chapter we studied a variety of different concepts that are used in the theory of machine learning and building artificially intelligent systems. These concepts arise from different contexts and different applications, but in this chapter we aggregated them at one place for reference. All these concepts need to be always at the back of mind when solving some real-life problems and building artificially intelligent systems.

## 2.12 Exercise

1. Find the amount of memory of your computer from settings. Use a platform of choice (e.g., Python, Matlab, R, etc.), and randomly create a numerical data matrix that would roughly use about half of the memory of the computer or less. Remember that a single floating point number takes 4–8 bytes of memory depending on what type of variable is used. Perform a simple matrix addition or subtraction operation. Don't bother with anything more complex. Note the time

of execution. Then randomly initialize a matrix with about two to four times the memory of the computer. Try the same operation, and note the time of execution. Make sure you have a hard drive caching enabled. Compare the two times. It should give you the idea of difference in processing big data (later case) against not-so-big data.

2. Use sample data given in [3]. Compute the density of the data for single dimension, and then compute the density of the data for two dimensions and so on till four dimensions. Plot the densities as function of dimension. Which curve does the plot resemble?

# Chapter 3

## Data Understanding, Representation, and Visualization



### 3.1 Introduction

Before starting with the theory of machine learning in the next part of the book, this chapter focusses on the understanding, representing, and visualizing the data. We will use the techniques described here multiple times throughout the book. These steps together can be called as data preprocessing.

With recent explosion of small devices that are connected to internet,<sup>1</sup> the amount of data that is being generated has increased exponentially. This data can be quite useful for generating a variety of insights if handled properly; else, it can only burden the systems handling it and slow down everything. The science that deals with general handling and organizing and then interpreting the data is called as data science. This is a fairly generic term, and concepts of machine learning and artificial intelligence are part of it.

### 3.2 Understanding the Data

The first step in building an application of artificial intelligence is to understand the data. The data in raw form can come from different sources, in different formats. Some data can be missing, some data can be mal-formatted, etc. It is a first task to get familiar with the data. Clean up the data as necessary.

The step of understanding the data can be broken down into three parts:

1. Understanding entities
2. Understanding attributes
3. Understanding data types

---

<sup>1</sup> Sometimes the network consisting of such devices is referred to as *internet of things* or *IoT*.

In order to understand these concepts, let's consider a dataset called *Iris data* [3]. Iris data is one of the most widely used dataset in the field of machine learning for its simplicity and its ability to illustrate many different aspects of machine learning at one place. Specifically, Iris data states a problem of multi-class classification of three different types of flowers, *Setosa*, *Versicolor*, and *Virginica*. The dataset is ideal for learning basic machine learning application as it does not contain any missing values and all the data is numerical. There are 4 features per sample, and there are 50 samples for each class totalling 150 samples. Here is a sample taken from the data (Table 3.1).

**Table 3.1** Sample from Iris dataset containing three classes and four attributes

Sepal-length	Sepal-width	Petal-length	Petal-width	Class label
5.1	3.5	1.4	0.2	Iris-setosa
4.9	3.0	1.4	0.2	Iris-setosa
4.7	3.2	1.3	0.2	Iris-setosa
4.6	3.1	1.5	0.2	Iris-setosa
5.0	3.6	1.4	0.2	Iris-setosa
4.8	3.4	1.9	0.2	Iris-setosa
5.0	3.0	1.6	0.2	Iris-setosa
5.0	3.4	1.6	0.4	Iris-setosa
5.2	3.5	1.5	0.2	Iris-setosa
5.2	3.4	1.4	0.2	Iris-setosa
7.0	3.2	4.7	1.4	Iris-versicolor
6.4	3.2	4.5	1.5	Iris-versicolor
6.9	3.1	4.9	1.5	Iris-versicolor
5.5	2.3	4.0	1.3	Iris-versicolor
6.5	2.8	4.6	1.5	Iris-versicolor
6.7	3.1	4.7	1.5	Iris-versicolor
6.3	2.3	4.4	1.3	Iris-versicolor
5.6	3.0	4.1	1.3	Iris-versicolor
5.5	2.5	4.0	1.3	Iris-versicolor
5.5	2.6	4.4	1.2	Iris-versicolor
6.3	3.3	6.0	2.5	Iris-virginica
5.8	2.7	5.1	1.9	Iris-virginica
7.1	3.0	5.9	2.1	Iris-virginica
6.3	2.9	5.6	1.8	Iris-virginica
6.5	3.0	5.8	2.2	Iris-virginica
6.7	3.0	5.2	2.3	Iris-virginica
6.3	2.5	5.0	1.9	Iris-virginica
6.5	3.0	5.2	2.0	Iris-virginica
6.2	3.4	5.4	2.3	Iris-virginica
5.9	3.0	5.1	1.8	Iris-virginica



### 3.2.1 Understanding Entities

In the field of data science or machine learning and artificial intelligence, entities represent groups of data separated based on conceptual themes and/or data acquisition methods. An entity typically represents a table in a database or a flat file, e.g., comma-separated variable (csv) file or tab-separated variable (tsv) file. Sometimes it is more efficient to represent the entities using a more structured format like *svmlight*.<sup>2</sup> Each entity can contain multiple attributes. The raw data for each application can contain multiple such entities.

In case of Iris data, we have only one such entity in the form of dimensions of sepals and petals of the flowers. However, if one is trying to solve this classification problem and finds that the data about sepals and petals alone is not sufficient, then he/she can add more information in the form of additional entities. For example, more information about the flowers in the form of their colors or smells or the longevity of the trees that produce them can be added to improve the classification performance.

### 3.2.2 Understanding Attributes

Each attribute can be thought of as a column in the file or table. In case of Iris data, the attributes from the single given entity are *sepal length in cm*, *sepal width in cm*, *petal length in cm*, and *petal width in cm*. If we had added additional entities like color, smell, etc., each of those entities would have their own attributes. It is important to note that in the current data, all the columns are all features, and there is no *ID* column. As there is only one entity, *ID* column is optional, as we can assign arbitrary unique *ID* to each row. However, when we have multiple entities, we need to have an *ID* column for each entity along with the relationship between *IDs* of different entities. These *IDs* can then be used to join the entities to form the feature space.

---

<sup>2</sup> The structured formats like *svmlight* are more useful in case of sparse data, as they add significant overhead when the data is fully populated. A sparse data is data with high dimensionality (typically in hundreds or more) but with many samples missing values for multiple attributes. In such cases, if the data is given as a fully populated matrix, it will take up a huge space in memory. The formats like *svmlight* employ a name-value pair approach to specify the name of attribute and its value in pair. The name-value pairs are given for only the attributes where value is present. Thus, each sample can now have different number of pairs. The model needs to assume that for all the missing name-value pairs, the data is missing. In spite of the added names in each sample, due to the sparsity of the data, the file is much smaller.

### 3.2.3 Understanding Data Types

Attributes in each entity can be of various different types from the storage and processing perspective, e.g., string, integer valued, datetime, binary (“true”/“false,” or “1”/“0”), etc. Sometimes the attributes can originate from completely different domains like an image, a sound file, etc. Each type needs to be handled separately for generating a feature vector that will be consumed by the machine learning algorithm. We will discuss the details of this processing in Chap. 15. As discussed before, we can also come across sparse data, in which case, some attributes will have missing values. This missing data is typically replaced with special characters, which should not be confused with any of the real values. In order to process data with missing values, one can either fill those missing values with some default values or use an algorithm that can work with missing data.

In case of Iris data, all the attributes are real valued numerical, and there is no missing data. However, if we add additional entities like color, it would have enumerative-type string features like *green*, *orange*, etc.

## 3.3 Representation and Visualization of the Data

Even after we have understood the data with the three hierarchical levels, we still do not know how the data is distributed and how it related to the output or class label. This marks the last step in preprocessing the data. We live in a three-dimensional world; so any data that is up to three dimensions, we can plot it and visualize it. However, when there are more than three dimensions, it gets tricky. The Iris data, for example, also has four dimensions. There is no way we can plot the full information in each sample in a single plot that we can visualize. There are a couple of options in such cases:

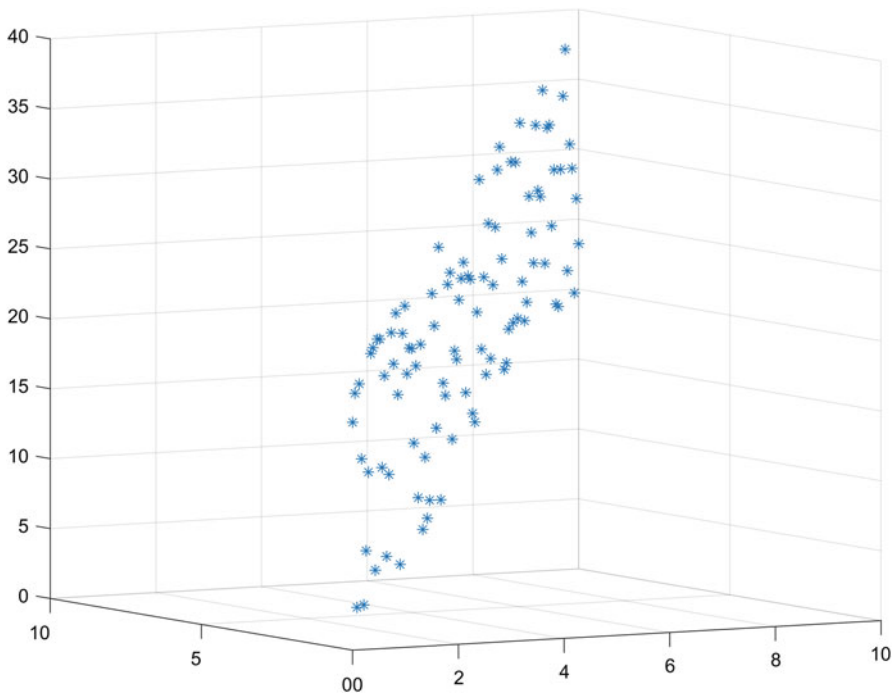
1. Draw multiple plots taking two or three dimensions at a time.
2. Reduce the dimensionality of the data and plot up to three dimensions

Drawing multiple plots is easy, but it splits the information, and it becomes harder to understand how different dimensions interact with each other. Reducing dimensionality is typically the preferred method. Most common methods used to reduce the dimensionality are:

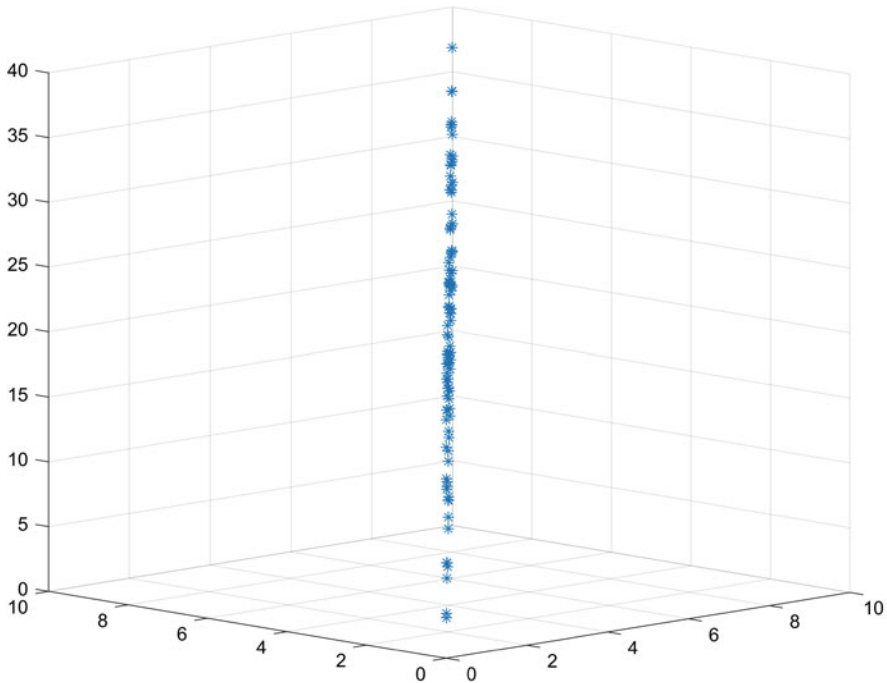
1. Principal component analysis or PCA
2. Linear discriminant analysis or LDA

### 3.3.1 *Principal Component Analysis*

We can only visualize the data in a maximum of two or three dimensions. However, it is common practice to have the dimensionality of the data in tens or even hundreds. In such cases, we can employ the algorithms just fine, as the mathematics on which they are based scales perfectly fine for higher dimensions. However, if we want to actually have a look at the data to see the trends in the distribution or to see the classification or regression in action, it becomes impossible. One way to do this is to plot the data in pairs of dimensions. However, in such case, we only see partial information in any one plot, and it is not always intuitive to understand the overall trends by looking at multiple separate plots. In many situations, the real dimensionality of data is much less than what the dimensionality in which the data is presented. For example, check the three-dimensional data plotted in Fig. 3.1 and the same data plotted with different perspectives in Fig. 3.2. As can be seen in the second perspective, the data is actually only two dimensional, if we can adjust the coordinates suitably. In other words, one can imagine all the data to be plotted on a single piece paper, which is only two dimensional and then holding it in skewed manner in a three-dimensional space. If we can find the exact



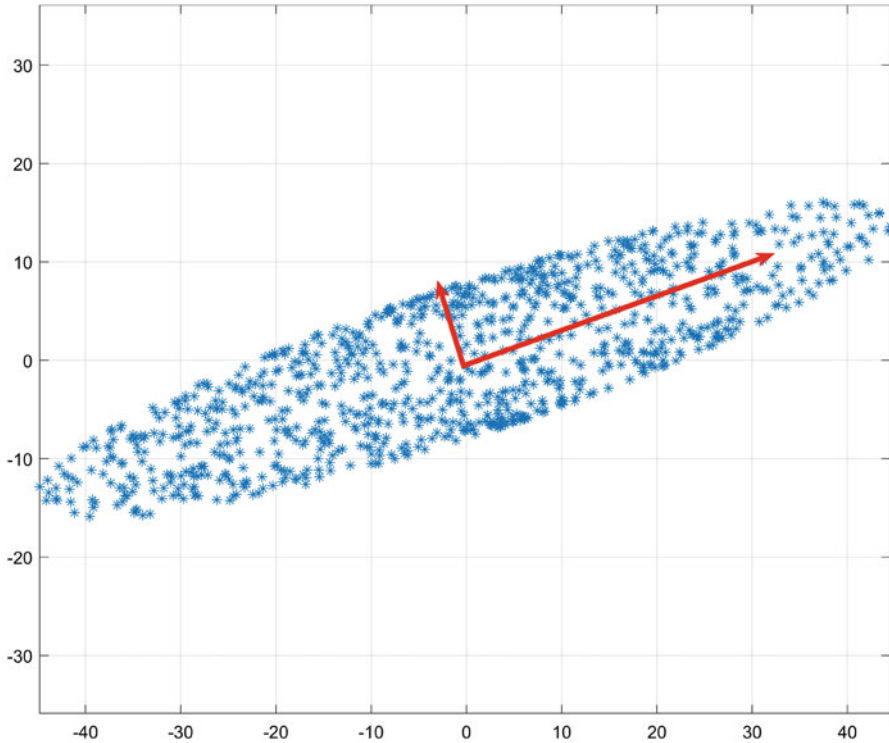
**Fig. 3.1** Three-dimensional data containing only two-dimensional information with first perspective



**Fig. 3.2** Three-dimensional data containing only two-dimensional information with alternate perspective

coordinates  $(X', Y')$  of the paper's orientation as linear combination of  $X$ ,  $Y$ , and  $Z$  coordinates of the three-dimensional space, we can reduce the dimensionality of the data from three to two. The above example is for illustrations only, and in most real situations, the lower dimensionality is not reflected in such obvious manner. Typically the data does have some information in all the dimensions, but the extent of the information in some dimensions can be very small compared to the information present in the other dimensions. For visualization purposes and also in real application purposes, it is quite acceptable to lose the information in those dimensions, without sacrificing any noticeable performance. As discussed in the previous chapter, the higher dimensionality increases the complexity of the problem exponentially, and such reduction in dimensionality for loss of some information is acceptable in most situations.

The mathematical process for finding the information spread across all the dimensions and then ranking the dimensions based on the information content is done using the theory of principal component analysis or PCA. This theory is based on properties of matrices, specifically the process of *singular value decomposition* (SVD). This process starts with first finding the dimension along which there is maximum spread or information content. If we have started with  $n$ -dimensions, then after the first principal component is found, the algorithm tries to find the

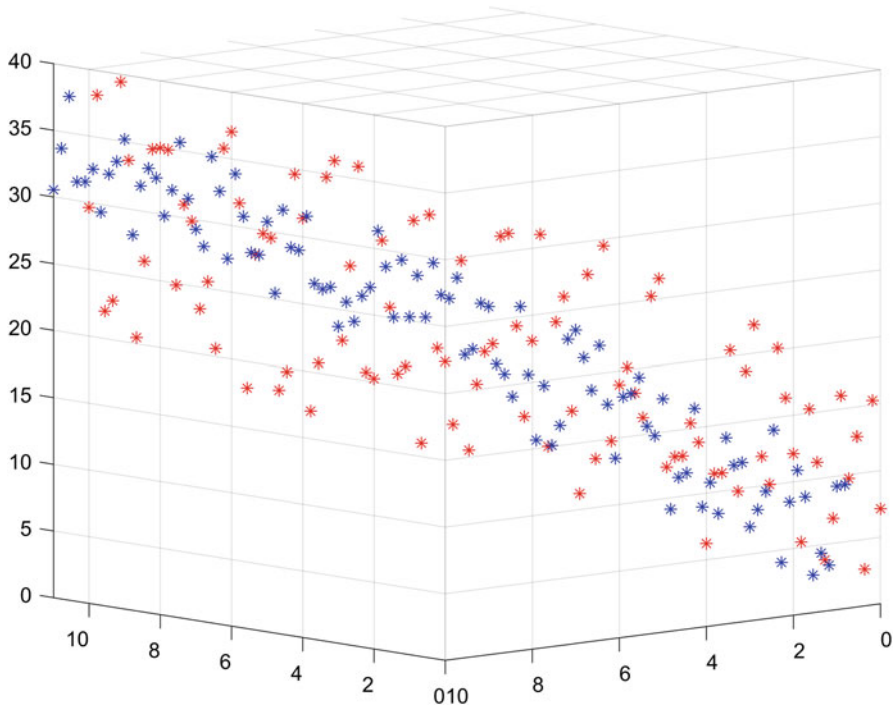


**Fig. 3.3** Two-dimensional data where PCA dimensionality is same but along different axes

next component of maximum spread in the remaining  $n-1$  dimensional space that is orthogonal to the first component. The process continues till we reach the last dimension. The process also gives coefficient for each of the principal component that represents the relative spread along that dimension. As all the components are found with deterministic and monotonically decreasing coefficient of spread, this representation is useful for any further analysis of the data. Thus, this process is not restricted to only reducing the dimensions, but to find the optimal dimensions that represent the data. As can be seen in Fig. 3.3 (principal components are shown in red arrows), the original data and principal components are both two-dimensional, but the representation of the data along principal components is different and preferable compared to original coordinates.

### 3.3.2 *Linear Discriminant Analysis*

The way PCA tries to find the dimensions that maximize variance of the data, linear discriminant analysis or LDA tries to maximize the separation between the



**Fig. 3.4** Three-dimensional data with two classes

classes of data. Thus, LDA can only effectively work when we are dealing with classification type of problem, and the data intrinsically contains multiple classes. Conceptually LDA tries to find the hyperplane in  $c - 1$  dimensions to achieve: maximize the separation of means of the two classes, and minimize the variance of each class. Here  $c$  is number of classes. Thus, while doing this classification, it finds a representation of the data in  $c - 1$  dimensions. For full mathematical details of theory, one can refer to [5].

Figure 3.4 shows three-dimensional data in one perspective, where the classes are not quite separable. Figure 3.5 shows another perspective of the data where classes are separable. LDA finds precisely this perspective as linear combination of the features and creates a one-dimensional representation of the data on a straight line where the classes are best separated. As there are only two classes, the LDA representation is one-dimensional. The LDA representation is independent of the original dimensionality of the data.

As can be seen, sometimes this representation is bit extreme and not quite useful in understanding the structure of the data, and then PCA would be a preferred choice.

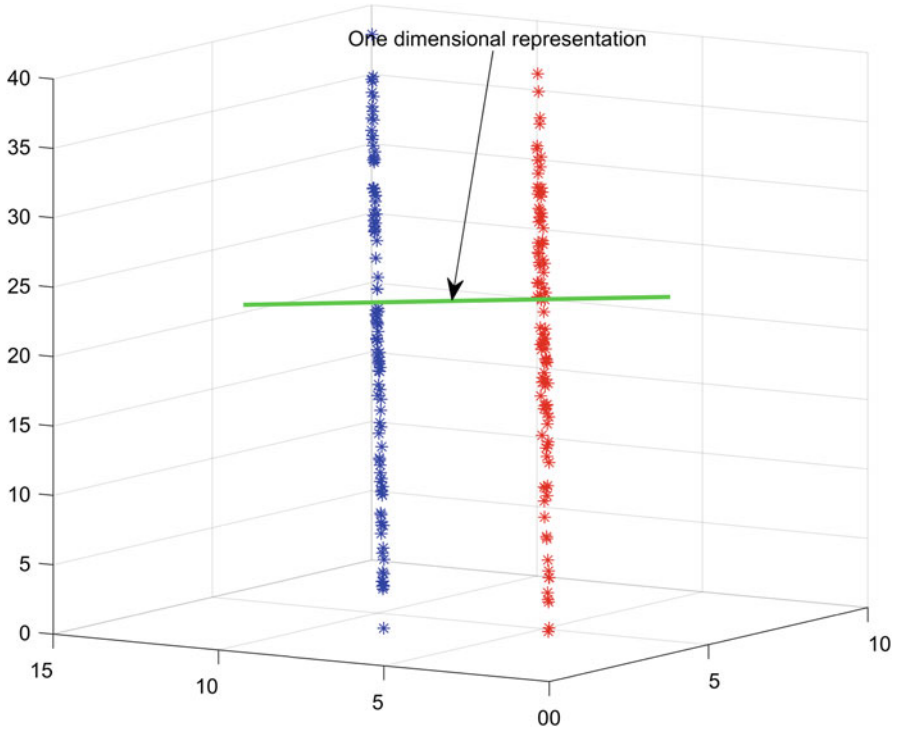


Fig. 3.5 Three-dimensional data with two classes in another perspective, with LDA representation. LDA reduces the effective dimensionality of data into one dimension where the two classes can be best separated

### 3.4 Conclusion

In this chapter, we studied the different aspects of preparing the data for solving the machine learning problem. Each problem comes with a unique set of properties and quirks in the training data. All these custom abnormalities in the data need to be ironed out before it can be used to train a model. In this chapter we looked at different techniques to understand and visualize the data, clean the data, reduce the dimensionality as possible, and make the data easier to model in the subsequent steps in the machine learning pipeline.

# Chapter 4

## Implementing Machine Learning Algorithms



### 4.1 Introduction

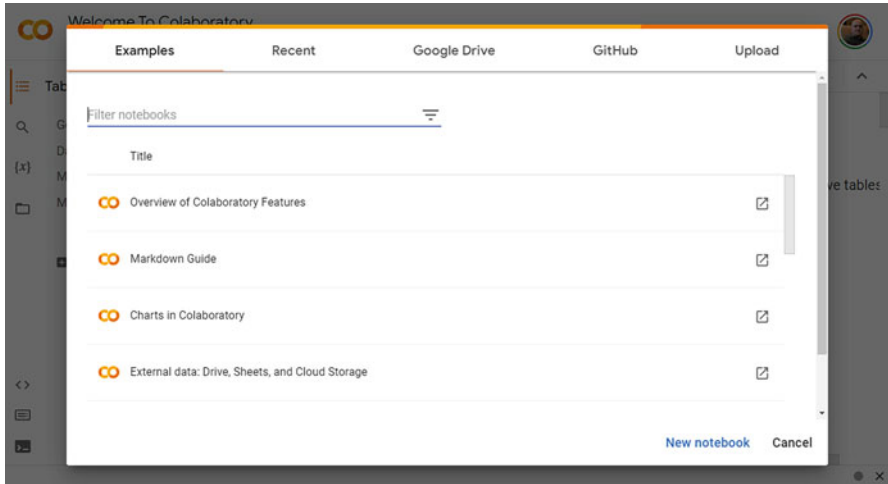
Studying the theory of machine learning supplemented with its implementation can really help with better understanding of the algorithms and concepts. It was also one of the common feedbacks received with the first edition of the book. One can set up Python environment on PC/Mac; however, the setup required for even starting with a single line of code can be quite daunting. Anaconda makes the process a bit streamlined, but it is highly recommended to use the online resources provided by Google, Microsoft, or Amazon to get started quicker. We will look at the systems provided by Google and Microsoft in this chapter, and we will use those throughout the book. It is highly recommended to not skip this chapter.

### 4.2 Use of *Google Colab*

We are going to use Google's *Colaboratory* or *Colab* platform for implementing the concepts in Python throughout the book. The primary reason for choosing this platform is its simplicity. The other offerings from Microsoft (Azure Machine Learning) and Amazon (Sagemaker) need a bit more involved setup to get started. We will cover the details of Azure ML in the latter section of the book that is dedicated for implementations of the machine learning solutions.

In order to use Google's platform, all one needs is a Google or a Gmail account. Once signed with that, you can visit the URL: <https://colab.research.google.com/>.





**Fig. 4.1** Welcome screen after logging in to Google account for using Google Colab

You will be shown the following welcome screen as shown in Fig. 4.1, where you can get started with your first Python Notebook.<sup>1</sup>

### 4.2.1 *scikit-learn Python Library*

As shown in the code displayed in Fig. 4.12 we will use the Python library called as *scikit-learn* or *sklearn* for implementing most of the classical machine learning algorithms in this book. This library has evolved and is still evolving and has become one of the most reference libraries that support most common algorithms with necessary customizations. *scikit-learn* is one of the oldest (started around 2013 [23]) and is relatively easy to use. However, it is important to note that it may not produce the optimal results in many situations. This is not to say that this is the best library out there. Many other libraries support only a subset of the algorithm, but they have optimized versions of those algorithms and can provide better accuracy. I have referenced such libraries, whenever relevant, throughout the book. It is recommended to start with *sklearn* for initial experimentation and then try out other libraries when optimizing the code for production.

<sup>1</sup> Python Notebook, also sometimes called as Jupyter Notebook, is an interactive interface where you can write the Python code and execute it and see the results. The code in Notebook is divided into a sequence of *cells*, and each cell is executed at once. One can also add cells with plain/formatted text for making the notebook self-explanatory.

### 4.3 Azure Machine Learning (AML)

In the previous version of this book, I used the first version of Azure ML, called as Azure ML Studio. However, that version is now discontinued, and we will look at the new and currently available version in this book. The current version is called as just *Azure Machine Learning* or *AML*, and we will call it just *AML* henceforth. *AML* has an interface similar to Google’s with some differences. Just like Google’s version is linked with Google account, you will need a Microsoft account in the form of *hotmail.com* or *outlook.com* to start using this feature.

#### 4.3.1 How to Start with AML?

The first place one needs to visit is: <https://azure.microsoft.com/en-us/free/>. Here, the user is greeted with the screen shown in Fig. 4.2.

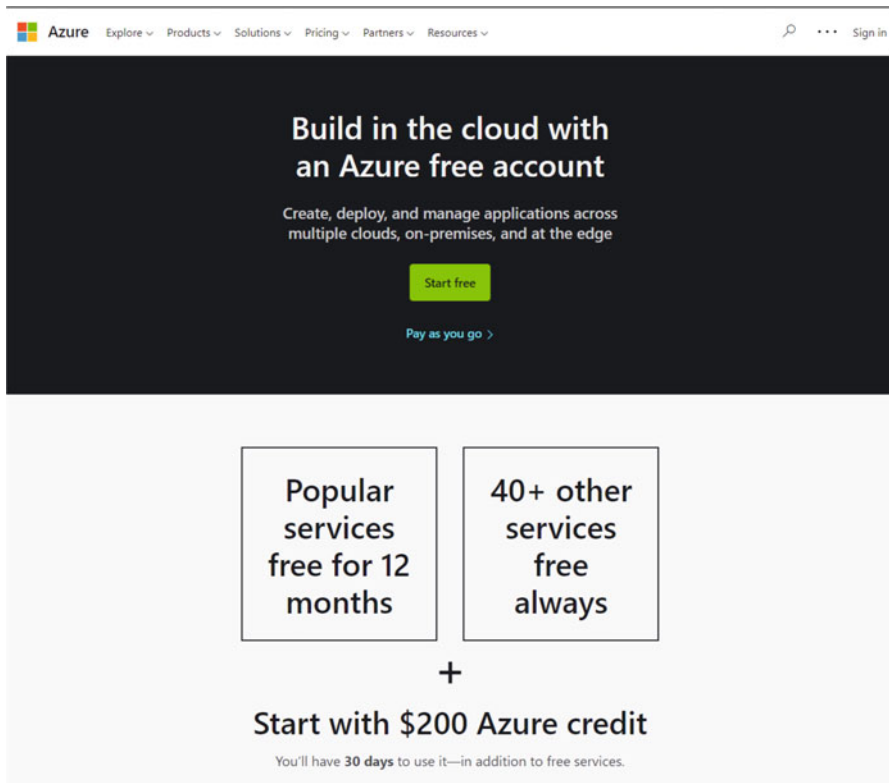
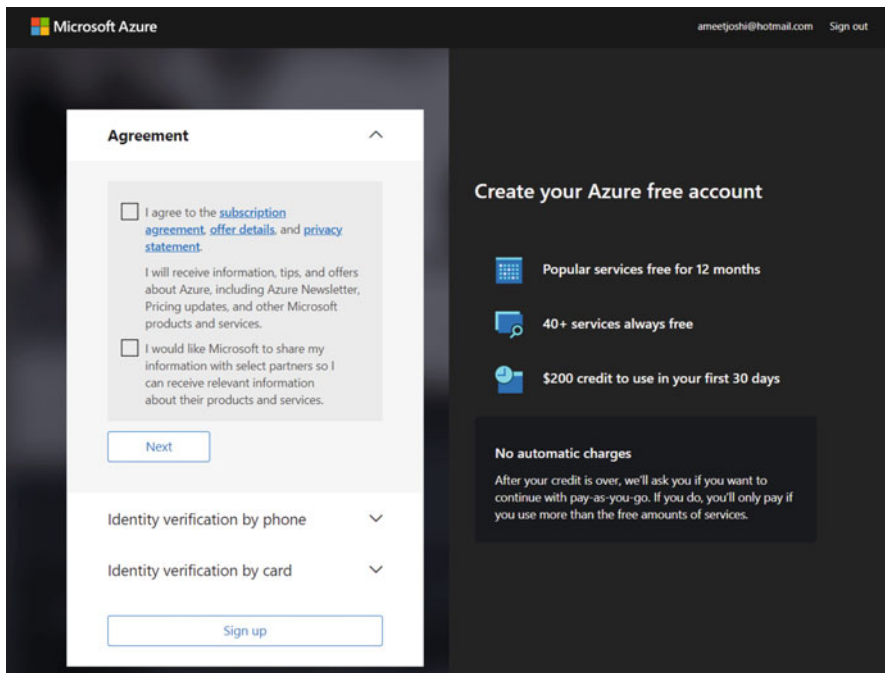


Fig. 4.2 Start screen for Azure Machine Learning



**Fig. 4.3** Screen after logging in to Azure Machine Learning for the first time

Here, one can click on *Start* and follow the standard sign-in process. It is important to note that as of writing this book in 2022, the AML free access is limited to only the first year, after which you will have to get a paid subscription. After logging in you are presented with the screen as shown in Fig. 4.3.

You will have to select the first item, the second is optional, and click *Next*. You will also have to set up a credit card to enable the service, but the first year is free. Once you set up the account, you are taken to central Azure portal. Here, you can click on *Products* and then choose *Azure Machine Learning* as shown in Fig. 4.4.

You need to further upgrade your account to pay as you go with \$200 credit to boot. Figure 4.5 shows the screen. You can select *Basic* to skip making any payment at the moment and click *Upgrade*. You also get to name your subscription. I have chosen *AML subscription*.

Now, you are ready to start using AML services, and you are presented with the options list as shown in Fig. 4.6. You see the option of *Machine Learning Studio (Classic)* here, which is deprecated service, and we will not focus on that in this book. I have given the full details of how to use this service in the previous version of this book, and it can be referred to as needed. We will use the option of *Machine Learning* and then click on *create*.

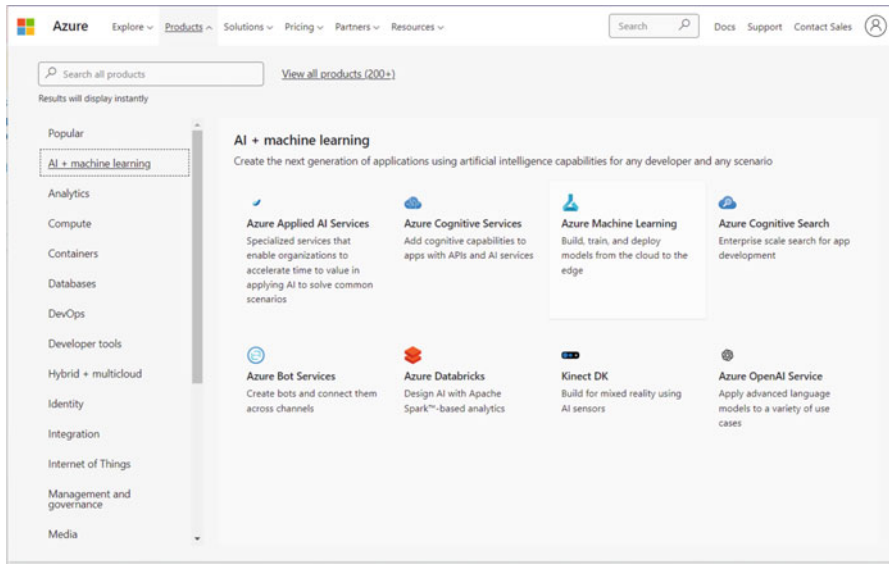


Fig. 4.4 Start page for Selecting Azure ML

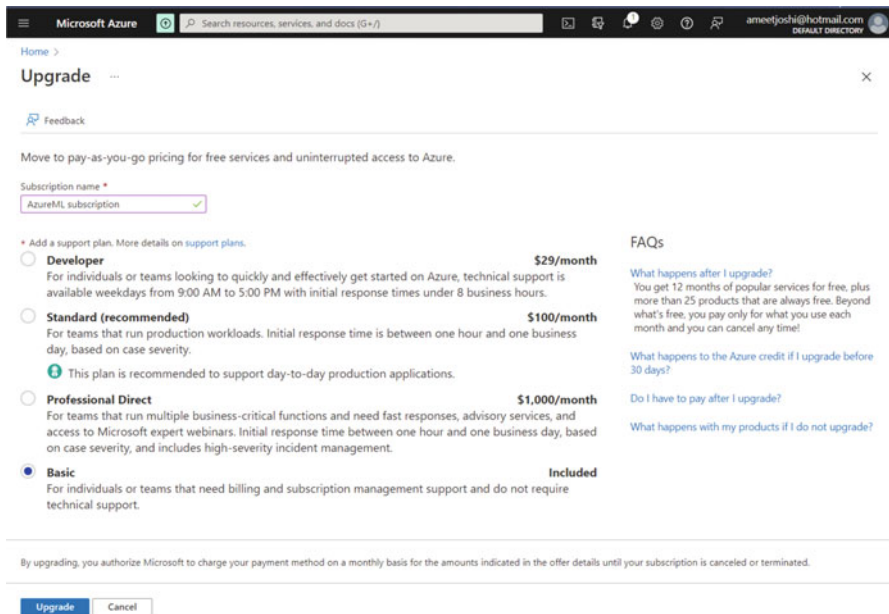


Fig. 4.5 Upgrade screen to choose the type of subscription you want

You will then see a form that you need to fill to create your first Azure Machine Learning workspace. Figure 4.7 shows the basic options you can fill in to get started. Click on *Review+Create*, and then after the validation passes, click *Create*.

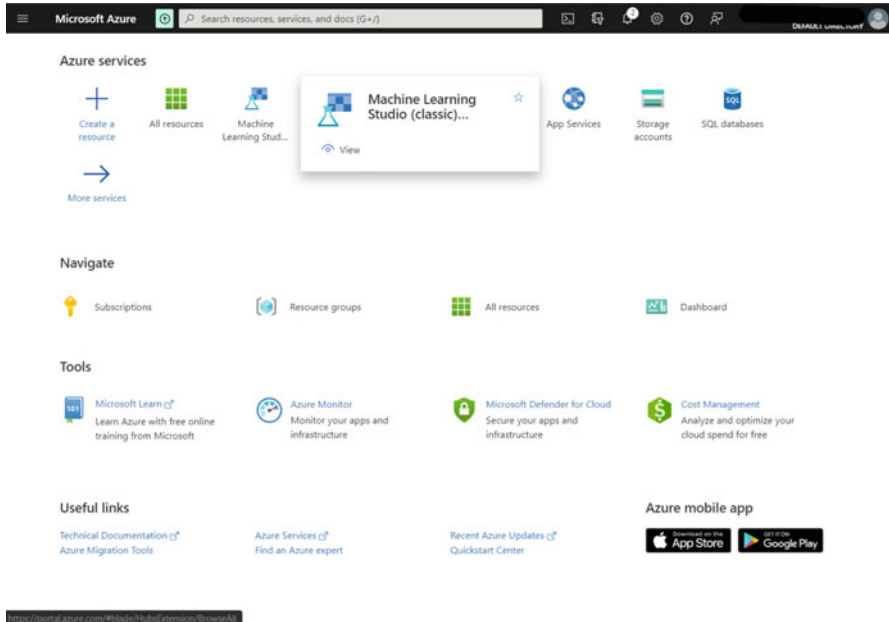


Fig. 4.6 Azure ML Options

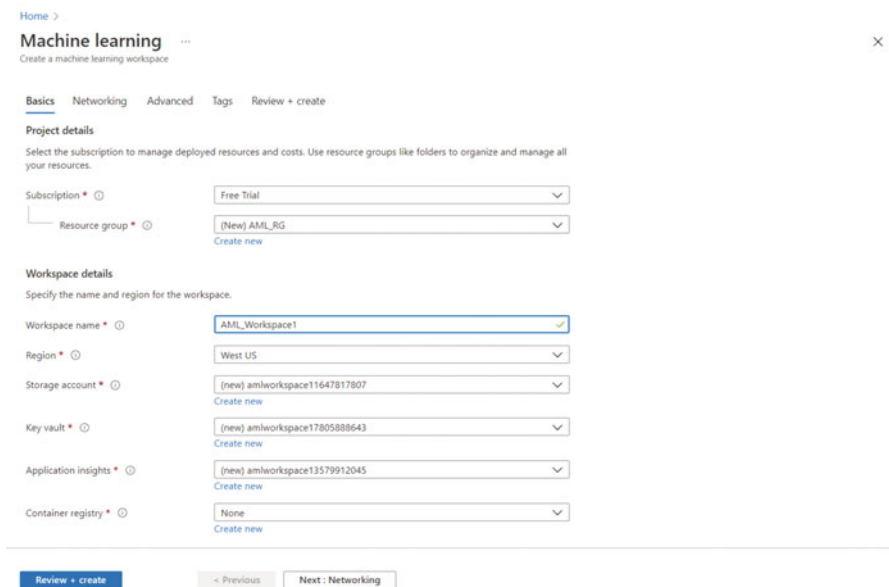


Fig. 4.7 Create your first Azure ML workspace

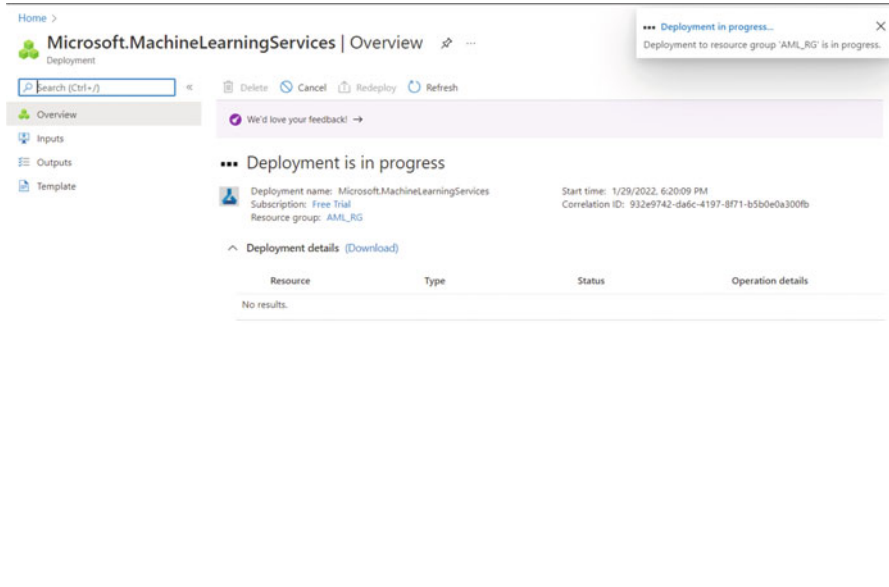


Fig. 4.8 Deploying the newly created workspace

Then the workspace deployment will start as shown in Fig. 4.8. After few minutes the deployment will be complete, and you will see the update on the same screen.

You can click on *Go to resource* and launch the AML studio environment. After launching the studio for the first time, you will see the welcome screen as shown in Fig. 4.9. You can click cross to skip the tour, or optionally you can go through it.

This is where AML differs from Google Colab, in the sense that AML is oriented towards more seasoned data scientist who is planning to build an end to end system that can be deployed in production scenarios. All the steps outlined so far essentially take you to the place where you can access the *Jupyter Notebook* like interface to start writing your first Python code. Figure 4.10 shows the primary options you have to start coding your first ML program. You have three options: (1) Notebooks: this will take you to the Google Colab like Jupyter Notebook interface. (2) Automated ML: this is a fully automated ML experience more catered to people who want to avoid coding and directly try to solve a problem at hand. (3) Designer: This is similar to Azure ML classic drag-and-drop interface.

We will only focus on the Notebooks option in this chapter.

The notebook creation and editing are essentially free, but the compute that is required to run the code is not free in AML. However, you have a \$200 credit to start with, and that can be sufficient to run multiple experiments. At the start, there is no default compute like the one provided in Google Colab, and you will have to go through the creation of the compute. Figure 4.11 shows the options for

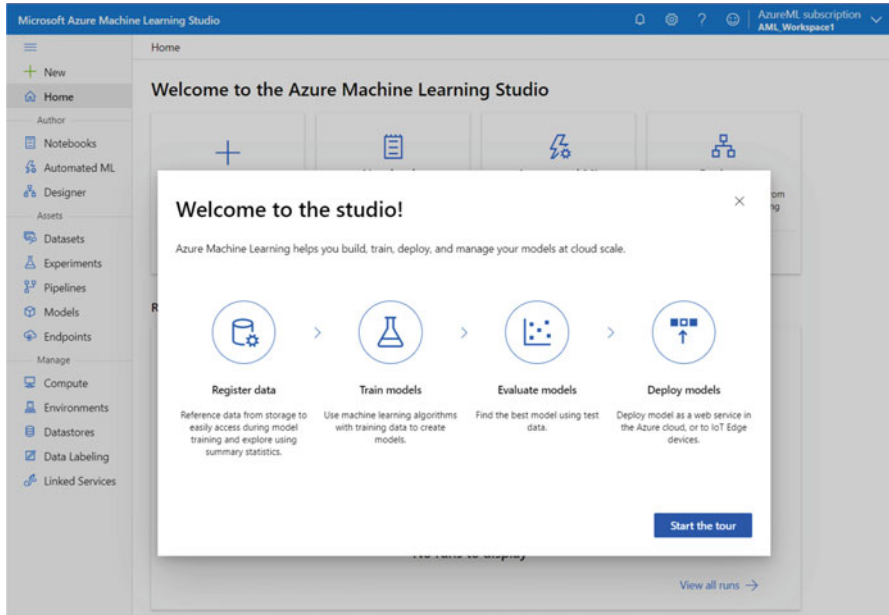


Fig. 4.9 Welcome screen showing the components of machine learning pipeline that you can build

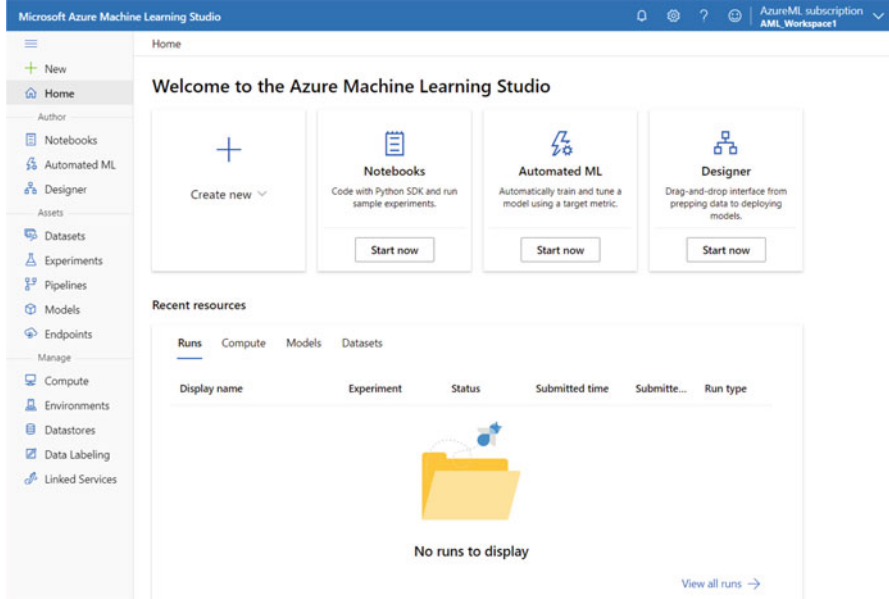


Fig. 4.10 Primary studio interface in Azure ML. You have three options to start creating your experiments: Notebooks, Automated ML, and Designer

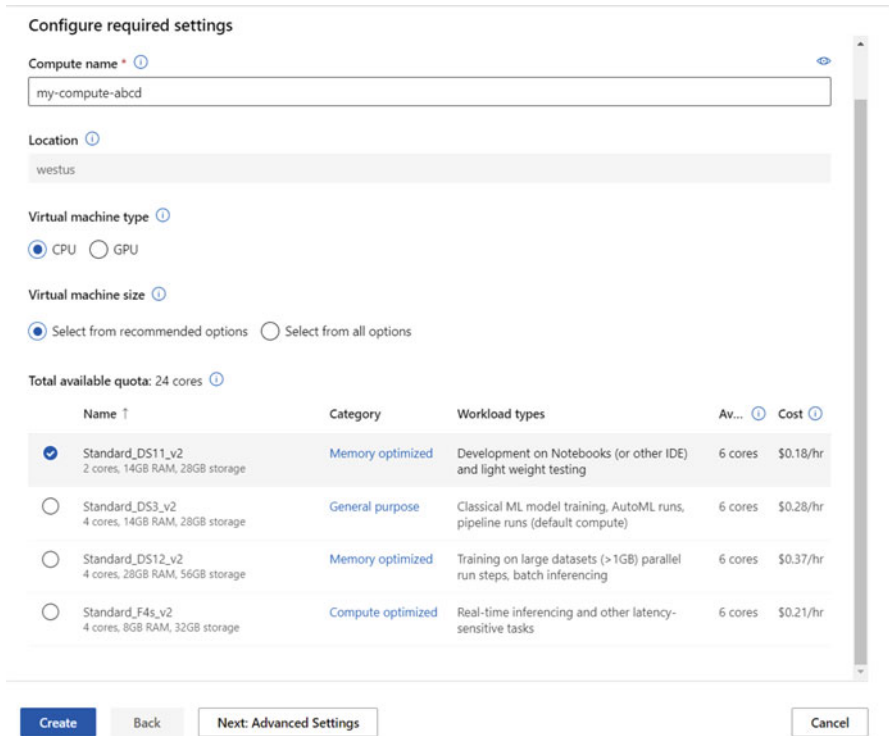


Fig. 4.11 Creating first compute to run the Python scripts

creating a compute. You can choose the cheapest option and CPU-only compute to optimize the cost. Later, depending on the size of experiment you want to run, you can carefully choose the number of cores and memory sizes. Google Colab provides a basic compute free of cost to ease this step. It is important to choose the compute name to be unique from the perspective of the entire AML ecosystem, so it can be a good idea to use your username as basis for this.

After creating the compute, we can get started with the first experiment. AML also provides some tutorials and getting-started guides, if you want to explore, but we will directly start with a new experiment. We will create a new notebook within an experiment and move over our first classifier program that we used earlier in Google Colab to get started. The interface in AML is similar to Jupyter Notebook, but it has its own features. Figures 4.12 and 4.13 show the script running the AML environment. You need to make sure that the compute is running in order to run the script.

From this point, the working in AML is essentially the same as working in Google Colab. You can install some of the missing libraries in the same way. AML also provides additional support for creating pipelines and models and managing their life cycles.



```

1 from sklearn import datasets
2 iris = datasets.load_iris()
3 X = iris.data
4 y = iris.target
[1] ✓ 5 sec

1 from sklearn.tree import DecisionTreeClassifier
2 dtc = DecisionTreeClassifier(max_depth=2)
3 dtc.fit(X,y)
4 y_pred = dtc.predict(X)
5 dtc.score(X,y)
[2] ✓ 1 sec
0.96

```

Fig. 4.12 Running Iris classifier script in AML, Part I

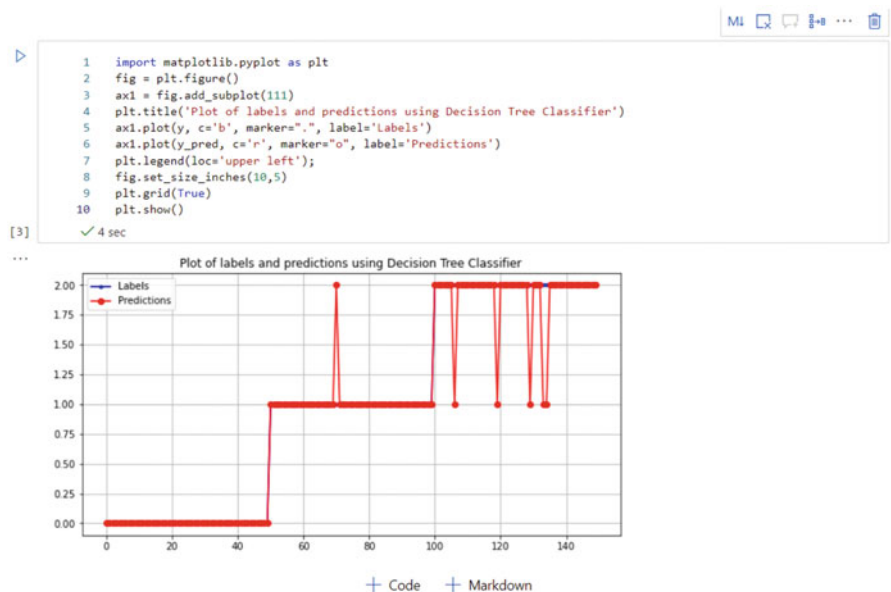


Fig. 4.13 Running Iris classifier script in AML, Part II

## 4.4 Conclusion

We looked at using the implementation platforms from Google and Microsoft in this chapter. Google Colab is simpler and *completely free* to use as of writing of this book, and we will use it most frequently in this book. However, having knowledge of other options is good, especially if it suits better with ecosystem that it comes with.

It is highly recommended to go through both these setups and make yourself comfortable with the environment as we will use them extensively in this book.

## 4.5 Exercises

1. Create an account in Google Colab using the instructions given in this chapter. Look at the sample notebooks to get acclimated with the coding setup.
2. Create an account in Microsoft AML using instructions given in the chapter. Explore the other options in the AML ecosystem. Also get familiarized with the different compute options available. It would be important to choose the right compute for the task to optimize the \$200 budget available for free.

# Part II

## Machine Learning

The unknown future rolls toward us. I face it, for the first time, with a sense of hope. Because if a machine, a Terminator, can learn the value of human life, maybe we can too.

—Sarah Connor, “Terminator 2: Judgement Day”

### **Part Synopsis**

In this part, we will learn theoretical foundations for various machine learning techniques. Each chapter contains theory as well as Python implementations of the concepts to complete the understanding of the topic.

# Chapter 5

## Linear Methods



### 5.1 Introduction

In general machine learning algorithms are divided into two types:

1. Supervised learning algorithms
2. Unsupervised learning algorithms

Supervised learning algorithms deal with problems that involve learning with guidance. In other words, the training data in supervised learning methods need labelled samples. For example, for a classification problem, we need samples with class label, or for a regression problem, we need samples with desired output value for each sample, etc. The underlying mathematical model then learns its parameters using the labelled samples, and then it is ready to make predictions on samples that the model has not seen, also called as test samples. Most of the applications in machine learning involve some form of supervision, and hence most of the chapters in this part of the book will focus on the different supervised learning methods.

Unsupervised learning deals with problems that involve data without labels. In some sense one can argue that this is not really a problem in *machine learning* as there is no knowledge to be learned from the past experiences. Unsupervised approaches try to find some structure or some form of trends in the training data. Some unsupervised algorithms try to understand the origin of the data itself. A common example of unsupervised learning is clustering.

In Chap. 2 we briefly talked about the linearity of the data and models. Linear models are the machine learning models that deal with linear data or nonlinear data that can be somehow transformed into linear data using suitable transformations. Although these linear models are relatively simple, they illustrate fundamental concepts in machine learning theory and pave the way for more complex models. These linear models are the focus of this chapter.

## 5.2 Linear and Generalized Linear Models

The models that operate on strictly linear data are called linear models, and the models that use some nonlinear transformation to map original nonlinear data to linear data and then process it are called as generalized linear models. The concept of linearity in case of supervised learning implies that the relationship between the input and output can be described using linear equations. For unsupervised learning, the concept of linearity implies that the distributions that we can impose on the given data are defined using linear equations. It is important to note that the notion of linearity does not imply any constraints on the dimensions. Hence, we can have multivariate data that is strictly linear. In case of one-dimensional input and output, the equation of the relationship would define a straight line in two-dimensional space. In case of two-dimensional data with one-dimensional output, the equation would describe a two-dimensional plane in three-dimensional space and so on. In this chapter we will study all these variations of linear models.

## 5.3 Linear Regression

Linear regression is a classic example of strictly linear models. It is also called as polynomial fitting which is one of the simplest linear methods in machine learning. Let us consider a problem of linear regression where training data contains  $p$  samples. Input is  $n$ -dimensional as  $(\mathbf{x}_i, i = 1, \dots, p)$  and  $\mathbf{x}_i \in \mathfrak{R}^n$ . Output is single dimensional as  $(y_i, i = 1, \dots, p)$  and  $y_i \in \mathfrak{R}$ .

### 5.3.1 Defining the Problem

The method of linear regression defines the following relationship between input  $\mathbf{x}_i$  and predicted output  $\hat{y}_i$  in the form of a linear equation as

$$\hat{y}_i = \sum_{j=1}^n x_{ij}.w_j + w_0 \quad (5.1)$$

where  $\hat{y}_i$  is the predicted output when the actual output is  $y_i$ .  $w_i, i = 1, \dots, p$  are called as the weight parameters, and  $w_0$  is called as the bias. Evaluating these parameters is the objective of training. The same equation can also be written in matrix form as

$$\hat{\mathbf{y}} = \mathbf{X}^T.\mathbf{w} + w_0 \quad (5.2)$$

where  $\mathbf{X} = [\mathbf{x}_i^T], i = 1, \dots, p$  and  $\mathbf{w} = [w_i], i = 1, \dots, n$ . The problem is to find the values of all weight parameters using the training data.

### 5.3.2 Solving the Problem

The most commonly used method to find the weight parameters is to minimize the mean square error between the predicted values and actual values. It is called as least squares method. When the error is distributed as Gaussian, this method yields an estimate called as maximum likelihood estimate or MLE. This is the best unbiased estimate one can find given the training data. The optimization problem can be defined as

$$\min \|y_i - \hat{y}_i\|^2 \quad (5.3)$$

Expanding the predicted value term, the full minimization problem to find the optimal weight vector  $\mathbf{w}^{\text{lr}}$  can be written as

$$\mathbf{w}^{\text{lr}} = \arg \min_w \left\{ \sum_{i=1}^p \left( y_i - \sum_{j=1}^n x_{ij} \cdot w_j - w_0 \right)^2 \right\} \quad (5.4)$$

This is a standard quadratic optimization problem and is widely studied in the literature. As the entire formulation is defined using linear equations, only linear relationships between input and output can be modelled. Figure 5.1 shows an example.

## 5.4 Example of Linear Regression

To illustrate the concept, let's construct a sample single-dimensional data where  $X = \{1, 2, \dots, 20\}$  and  $y$  is directly proportional to  $X$  with added random noise. The code as well as the plot of data is shown in Fig. 5.1.

Now let's use this data to train a linear regression model and predict the outcomes for  $X = \{21, 22, \dots, 40\}$ . The code and outcome are shown in Fig. 5.2.

As you can see, the model has learnt the linear trend from the sample data in spite of the added noise, and we can see a strictly linear prediction from the model for the new data.

```

✓ [16] import numpy as np
import matplotlib.pyplot as plt
X = np.array(range(1,21)).reshape(20,1)
y = X + 2*np.random.rand(20,1)
plt.plot(X,y,'*')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Sample data to train the regression model')
plt.grid(True)
plt.show()

```

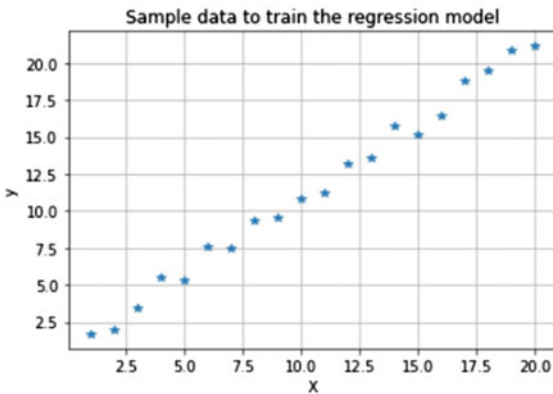


Fig. 5.1 Python code and plot for sample one-dimensional data to illustrate linear regression

## 5.5 Regularized Linear Regression

Although, in general, the solution obtained by solving Eq. 5.4 gives the best unbiased estimate, in some specific cases, where it is known that the error distribution is not Gaussian or the optimization problem is highly sensitive to the noise in the data, the above procedure can result in what is called as overfitting. In such cases, a mathematical technique called regularization is used.

### 5.5.1 Regularization

Regularization is a formal mathematical trickery that modifies the problem statement with additional constraints. The main idea behind the concept of regularization is to simplify the solution. The theory of regularization is typically attributed to Russian mathematician *Andrey Tikhonov*. In many cases the problems are what is referred to as *ill posed*. What it means is that the training data if used to

```

from sklearn.linear_model import LinearRegression
reg = LinearRegression().fit(X, y)
X_pred = np.array(range(21,41)).reshape(20,1)
y_pred = reg.predict(X_pred)
plt.plot(X_pred,y_pred, '*')
plt.xlabel('X_pred')
plt.ylabel('y_pred')
plt.title('Predictions of the regression model')
plt.xticks([20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41])
plt.grid(True)
plt.show()

```

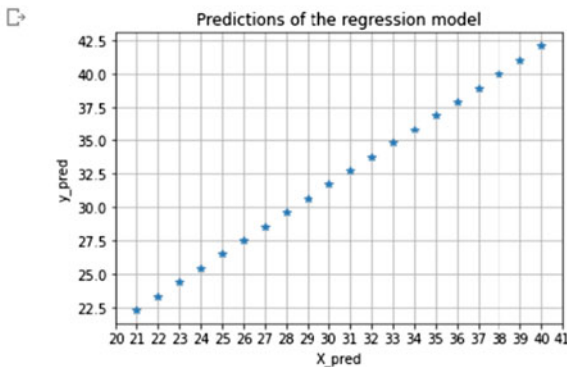


Fig. 5.2 Python code and plot to show the predictions of the linear regression model

the full extent can produce a solution that is highly overfitted and possesses less generalization capabilities. Regularization tried to add additional constraints on the solution, thereby making sure the overfitting is avoided and the solution is more generalizable. The full mathematical theory of regularization is quite involved, and the interested reader can refer to [57].

Multiple regularization approaches are proposed in the literature, and one can experiment with many. However, we will discuss two most commonly used ones. The approaches discussed below are also sometimes referred to as *shrinkage methods*, as they try to shrink the weight parameters close to zero.

### 5.5.2 Ridge Regression

In *ridge regression* approach, the minimization problem defined in Eq. 5.4 is constrained with



$$\sum_{j=1}^n (w_j)^2 \leq t \quad (5.5)$$

where  $t$  is a constraint parameter. Using Lagrangian approach,<sup>1</sup> the joint optimization problem can be written as

$$\mathbf{w}^{Ridge} = \arg \min_w \left\{ \sum_{i=1}^p \left( y_i - \sum_{j=1}^n x_{ij} \cdot w_j - w_0 \right)^2 + \lambda \sum_{j=1}^n (w_j)^2 \right\} \quad (5.6)$$

where  $\lambda$  is the standard Lagrangian multiplier.

### 5.5.3 Lasso Regression

In *Lasso regression* approach, the minimization problem defined in Eq. 5.4 is constrained with

$$\sum_{j=1}^n |w_j| \leq t \quad (5.7)$$

where  $t$  is a constraint parameter. Using Lagrangian approach, the joint optimization problem can be written as

$$\mathbf{w}^{Lasso} = \arg \min_w \left\{ \sum_{i=1}^p \left( y_i - \sum_{j=1}^n x_{ij} \cdot w_j - w_0 \right)^2 + \lambda \sum_{j=1}^n |w_j| \right\} \quad (5.8)$$

## 5.6 Generalized Linear Models (GLM)

The *generalized linear models* or *GLMs* represent generalization of linear models by expanding their scope to handle nonlinear data that can be converted into linear form using suitable transformations. The obvious drawback or limitation of linear regression is the assumption of linear relationship between input and output. In quite a few cases, the nonlinear relationship between input and output can be converted into linear relationship by adding an additional step of transforming one

---

<sup>1</sup> Lagrangian method is a commonly used method of integrating the regularization constraints into the optimization problem, thereby creating a single optimization problem.

of the data (input or output) into another domain. The function that performs such transformation is called as basis function or link function. For example, logistic regression uses logistic function as basis function to transform the nonlinearity into linearity. Logistic function is a special case where it also maps the output between the range of [0 and 1], which is equivalent to a probability density function. Also, sometimes the response between input and output is monotonic, but not necessarily linear due to discontinuities. Such cases can also be converted into linear space with the use of specially constructed basis functions. We will discuss logistic regression to illustrate the concept of GLM.

### 5.6.1 Logistic Regression

The building block of logistic regression is the logistic sigmoid function  $\sigma(x)$  and is defined as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (5.9)$$

Figure 5.3 shows the plot of Eq. 5.1. Logistic regression adds an exponential functional on top of linear regression to constrain the output  $y_i \in [0, 1]$  rather than  $y_i \in \mathfrak{R}$  as in linear regression. The relationship between input and predicted output for logistic regression can be given as

$$\hat{y}_i = \sigma \left( \sum_{j=1}^n x_{ij} \cdot w_j + w_0 \right) \quad (5.10)$$

As the output is constrained between [0 and 1], it can be treated as a probabilistic measure. Also, due to symmetrical distribution of the output of logistic function between  $-\infty - \infty$ , it is also better suited for classification problems. Other than these differences, there is no fundamental difference between linear and logistic regressions. Although there is nonlinear sigmoid function present in the equation, it should not be mistaken for a nonlinear method of regression. The sigmoid function is applied after the linear mapping between input and output, and at heart this is still a variation of linear regression. The minimization problem that needs to be solved for logistic regression is a trivial update from the one defined in Eq. 5.1. Due to its validity in regression as well as classification problems, unlike the linear regression, logistic regression is the most commonly used approach in the field of machine learning as default first alternative.

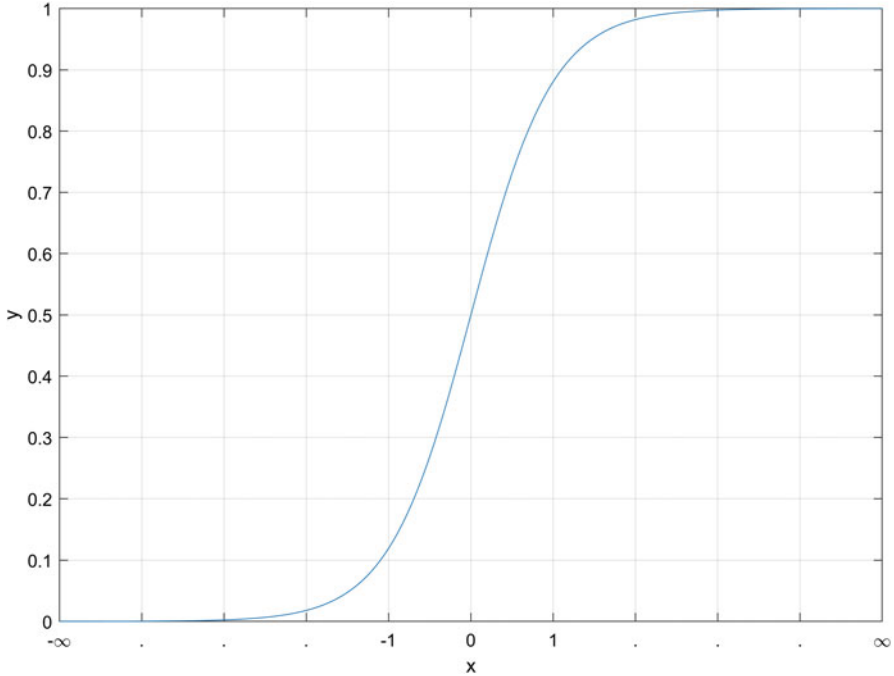


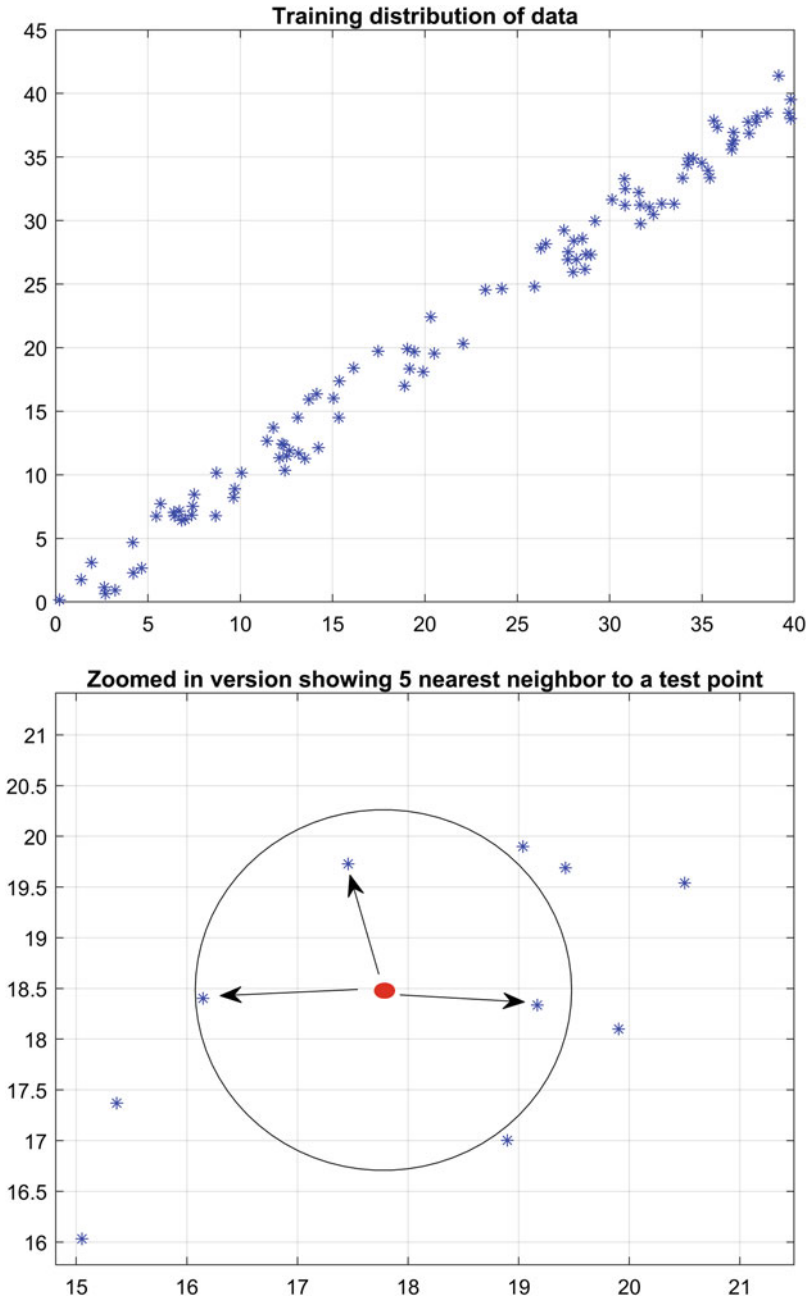
Fig. 5.3 Plot of logistic sigmoid function

## 5.7 K-Nearest Neighbor (KNN) Algorithm

The KNN algorithm is not exactly an example of a linear method, but it is one of the simplest algorithms in the field of machine learning, and it is apt to discuss it here in the first chapter of this part. KNN is also a generic method that can be used as a classifier or regressor. Unlike linear methods described before in this chapter, this algorithm does not assume any type of functional relationship between input and output.

### 5.7.1 Definition of KNN

In order to illustrate the concept of k-nearest neighbor algorithm, consider a case of two-dimensional input data as shown in Fig. 5.4. The top plot in the figure shows the distribution of the data. Let there be some relationship between this input data and output data (not shown here). For the time being, we can ignore that relationship. Let us consider that we are using the value of  $k$  as 3. As shown in the bottom plot in the figure, let there be a test sample located as shown by red dot. Then we find the 3 nearest neighbors of the test point from the training distribution as shown. Now, in



**Fig. 5.4** Figure showing a distribution of input data and showing the concept of finding nearest neighbors

order to predict the output value for the test point, all we need to do is find the value of the output for the 3 nearest neighbors and average that value. This can be written in equation form as

$$\hat{y} = \left( \sum_{i=1}^k y_i \right) / k \quad (5.11)$$

where  $y_i$  is the output value of the  $i$ th nearest neighbor. As can be seen, this is one of the simplest ways to define the input-to-output mapping. There is no need to assume any priory knowledge or any need to perform any type of optimization. All you need to do is to keep all the training data in memory and find the nearest neighbors for each test point and predict the output.

This simplicity does come at a cost though. This lazy execution of the algorithm requires heavy memory footprint along with a high computation to find the nearest neighbors for each test point. However, when the data is fairly densely populated, and computation requirements can be handled by the hardware, KNN produces good results in spite of being expressed with overly simplistic logic.

### 5.7.2 *Classification and Regression*

As the formula expressed in Eq. 5.11 can be applied to classification as well as regression problems, KNN can be applied to both types of problems without the need to change anything in the architecture. Figure 5.4 showed an example of regression. Also, as KNN is a local method as opposed to global method, it can easily handle nonlinear relationships unlike the linear methods described above. Consider the two-class nonlinear distribution as shown in Fig. 5.5. KNN can easily separate the two classes by creating the circular boundaries as shown based on the local neighborhood information expressed by Eq. 5.11.

### 5.7.3 *Other Variations of KNN*

As such, the KNN algorithm is completely described by Eq. 5.11. However, there exist some variations of the algorithm in the form of weighted KNN, where the value of each neighbors output is inversely weighted by its distance from the test point. In other variation, instead of using Euclidean distance, one can use Mahalanobis distance [16] to accommodate the variable variance of the data along different dimensions.

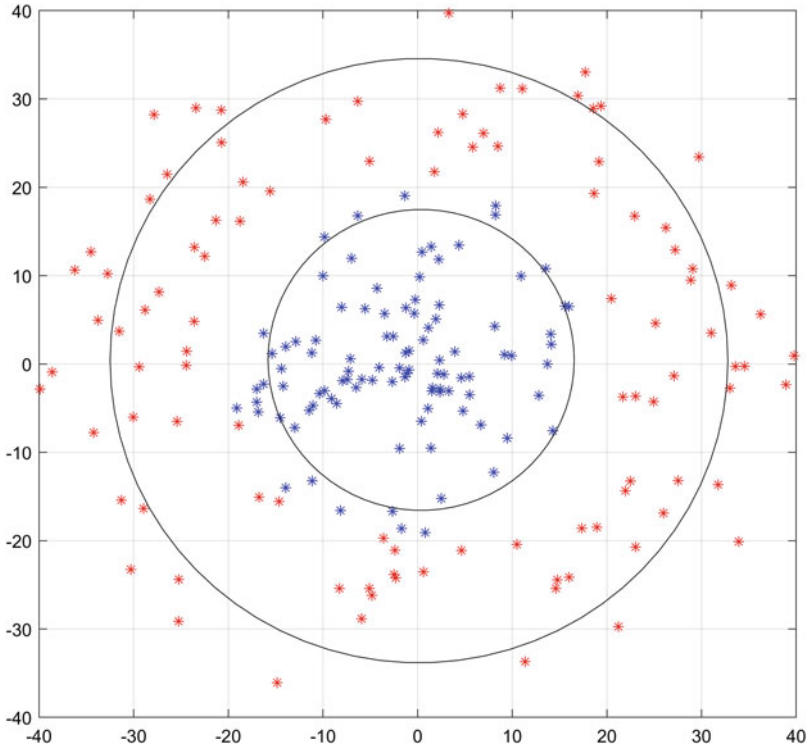


Fig. 5.5 Figure showing nonlinear distribution of the data

## 5.8 Conclusion

In this chapter we looked at some of the simple techniques to introduce the topic of machine learning algorithms. Linear methods form the basis of all the subsequent algorithms that we will study throughout the book. Generalized linear methods extend the scope of the linear methods to apply for some simple nonlinear cases as well as probabilistic methods. KNN is another simple technique that can be used to solve the most basic problems in machine learning and also illustrates the use of local methods as opposed to global methods.

## 5.9 Exercises

1. Create a Google Colab account, and try to execute the code that is illustrated in the chapter.

2. Think of examples of supervised and unsupervised learning during our day-to-day life, and compare and contrast them.
3. Use scikit-learn package, and import KNN module [24], and apply it to a sample data of your choice to carry out classification using command: *from sklearn.neighbors import KNeighborsClassifier*.
4. Repeat the above experiment to carry out a regression problem. Use *from sklearn.neighbors import KNeighborsRegressor*.

# Chapter 6

## Perceptron and Neural Networks



### 6.1 Introduction

For most people, the area of machine learning and artificial intelligence starts and ends with neural networks. From the name itself, it suggests an implicit resemblance with human brain and the neurons inside it. Common sense dictates: as the biological neurons are responsible for the human intelligence, these artificial neurons must be the building blocks of artificial intelligence. Incidentally, in this case, the common sense is not far from the truth. The invention of such an artificial neuron (technically called as perceptron in the literature) indeed marked the beginning of modern neural networks.

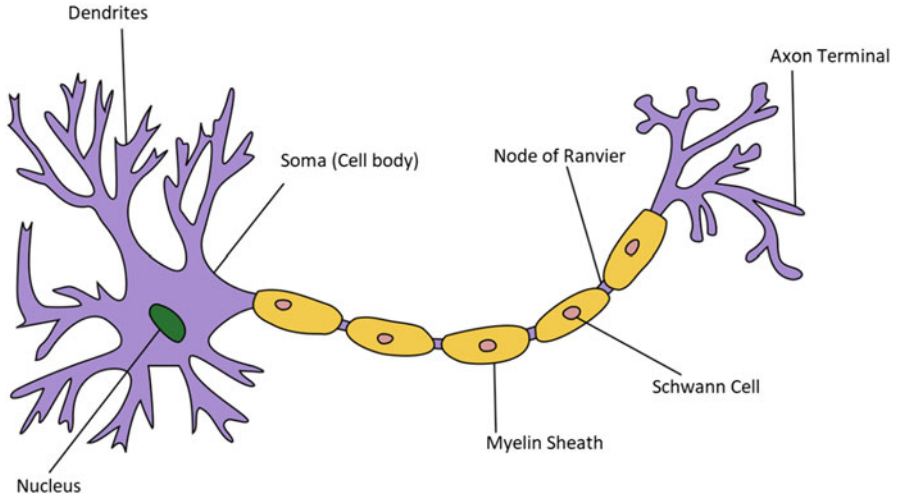
#### 6.1.1 Biological Neuron

A biological neuron looks like the illustration shown in Fig. 6.1. It is composed of three main parts that are unique to it along with other typical cellular objects:

1. Soma: This is the nucleus of the neuron.
2. Dendrites: They exhibit a complex treelike structure with branches ranging up to tens of thousands. These are responsible for receiving messages from other neurons.
3. Axon: This is a long stemlike structure that carries the incoming signals from the dendrites to send it to other neurons that may be physically farther away. It ends in axon terminals that look similar to dendrites.

The activation of neurons and the transmission of the messages to and from neurons are controlled by electrochemical processes. Each neuron is continuously getting activation signals from the other connecting neurons. The information is





**Fig. 6.1** Biological neuron [25]

gathered in the neuron's cell body where it gets processed and then sent out to axon terminals.

## 6.2 Perceptron

Perceptron was introduced by Rosenblatt [64] in 1957 with an architecture that was strongly influenced by that of a biological neuron. It was used to craft a generalized computation framework for solving linear problems. It was quite effective, one of a kind machine at the time and seemingly had unlimited potential. In the early days after its successful implementation to solve certain types of problems, some critical flaws were detected in the theory of perceptron that limited their scope significantly (e.g., problem of implementing XOR operation). However, these issues were overcome in time with addition of multiple layers in the architecture of the perceptron giving rise to the concept of artificial neural networks (ANNs). The addition of nonlinear kernel functions like sigmoid further extended the scope of ANNs. We will study the concept of perceptron and its evolution into modern ANNs in this chapter. However, we will restrict the scope to traditional neural networks and will not delve into the deep networks. We will study those in Chap. 13.

Geometrically a single-layer perceptron with linear mapping represents a linear plane in  $n$ -dimensions. In  $n$ -dimensional space, the input vector is represented as  $(x_1, x_2, \dots, x_n)$  or  $\mathbf{x}$ . The coefficients or weights in  $n$ -dimensions are represented as  $(w_1, w_2, \dots, w_n)$  or  $\mathbf{w}$ . The equation of perceptron in the  $n$ -dimensions is then written in vector form as

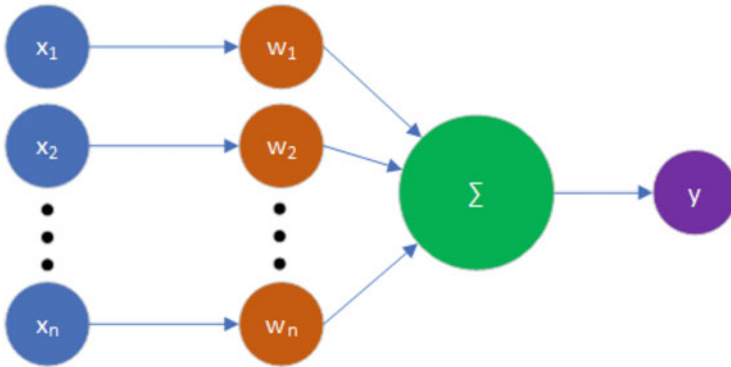


Fig. 6.2 Perceptron

$$\mathbf{x} \cdot \mathbf{w} = y \quad (6.1)$$

Figure 6.2 shows an example of a  $n$ -dimensional perceptron. This equation looks a lot like the linear regression equation that we studied in Chap. 5, which is essentially true, as perceptron represents a similar computational architecture to solve this problem. However, the differences lie in the way the weights are computed or learned.

### 6.2.1 Implementing Perceptron

Let's use the *sklearn* library and *Google Colab* as explained in the earlier chapter to implement the perceptron. As perceptron architecture is primarily used for classification, let's use the *Iris* dataset to illustrate the use of perceptron. *Iris* dataset contains 150 samples corresponding to three types of flowers. Each type of flower has 50 samples. There are *four* features for each samples making the data four dimensional.

Figure 6.3 shows how we can import the perceptron model from *sklearn* and train it on *Iris* data. In Fig. 6.4 we apply the trained model on the same data to see how well it is trained. As can be seen, the overall prediction accuracy of this classifier is only at 0.48. As the original data is not linearly separable, the accuracy of simple perceptron is not great.

Ideally, we should use separate datasets for training and predicting, but we are using the same dataset here for the purpose of illustrating how well the model is learning the patterns and also to emphasize the fact that the model is not just remembering the training patterns showing 100% accuracy on them. We will also use these scores as reference to compare with other models that we will learn in the chapter.

```
from sklearn import datasets
from sklearn.linear_model import Perceptron
iris = datasets.load_iris()
X = iris.data
y = iris.target

[2] iris.feature_names, iris.target_names

(['sepal length (cm)',
 'sepal width (cm)',
 'petal length (cm)',
 'petal width (cm)'],
 array(['setosa', 'versicolor', 'virginica'], dtype='<U10'))
```

Fig. 6.3 Implementing perceptron using Google Colab and sklearn

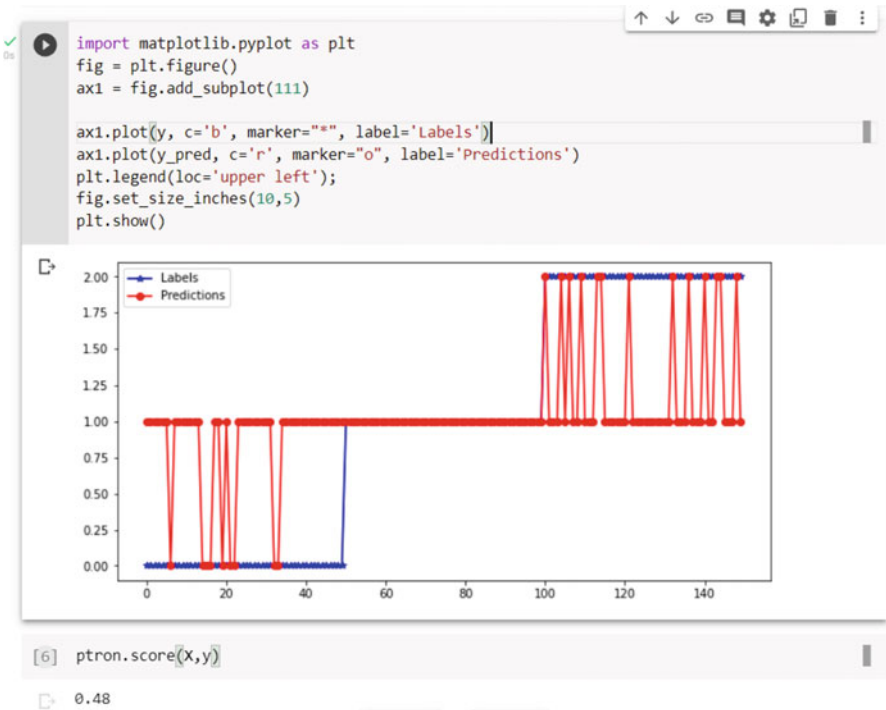


Fig. 6.4 Visualizing the training performance with perceptron on Iris data

If there are multiple variables to be predicted using the same set of inputs, one can have a series of perceptrons in parallel to generate those outputs. This architecture is called as single-layer perceptron.

### 6.3 Multilayered Perceptron or Artificial Neural Network

Multilayered perceptron or (*MLP*) is a logical extension of the single-layer architecture, again following the functioning of biological neural network, where there are multiple layers of perceptrons chained in series. Each layer can contain an arbitrary number of nodes or perceptrons as needed. Figure 6.1 shows an illustration of a generic MLP with 3 layers. Let  $n_1$  be the number of nodes in layer 1, which is the same as the input dimensionality. The subsequent layers have  $n_2$  and  $n_3$  number of nodes. The layers of nodes between the input and output layers are called as hidden layers, as their size is independent of the input and output, and are not directly visible from that perspective. The number of nodes in hidden layers can have any arbitrary value. Typically the more complex the relation between input and output, the more and bigger hidden layers are used. Also, all the consecutive layers in MLP are fully connected, meaning each of the internal layers needs to be fully connected to all the nodes in the previous and next layer. It is important to note that as long as we are using linear mapping (also called as activation function) at each node, single-layer perceptron and multilayered perceptron are mathematically equivalent. In other words, having multiple layers does not really improve the capabilities of the model, and it can be proved mathematically using rigorous calculations. Thus, the real benefits of MLP architecture start to surface with the use of nonlinear activation functions.

#### 6.3.1 Feedforward Operation

The network shown in Fig. 6.5 also emphasizes another important aspect of MLP called as feedforward operation. The information that is entered from the input propagates through each layer towards the output. There is no feedback of

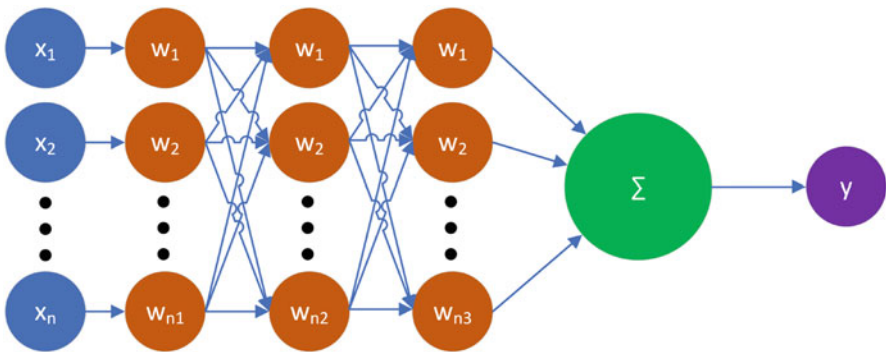


Fig. 6.5 Multilayered perceptron

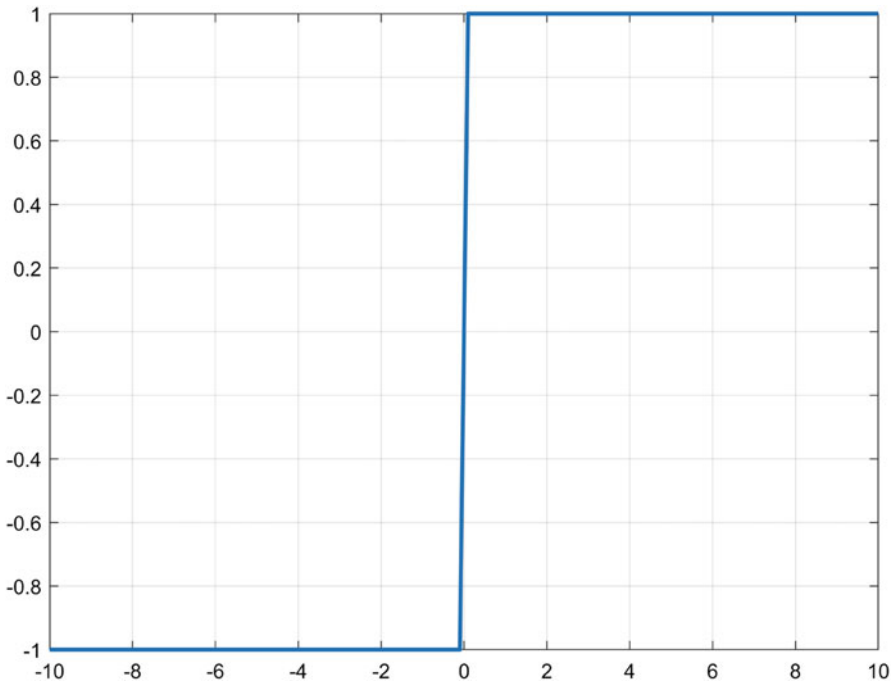
information from any layer backwards when the network is used for predicting the output in the form of regression or classification. This process also closely resembles the operation of human brain.

### 6.3.2 *Nonlinear MLP or Nonlinear ANN*

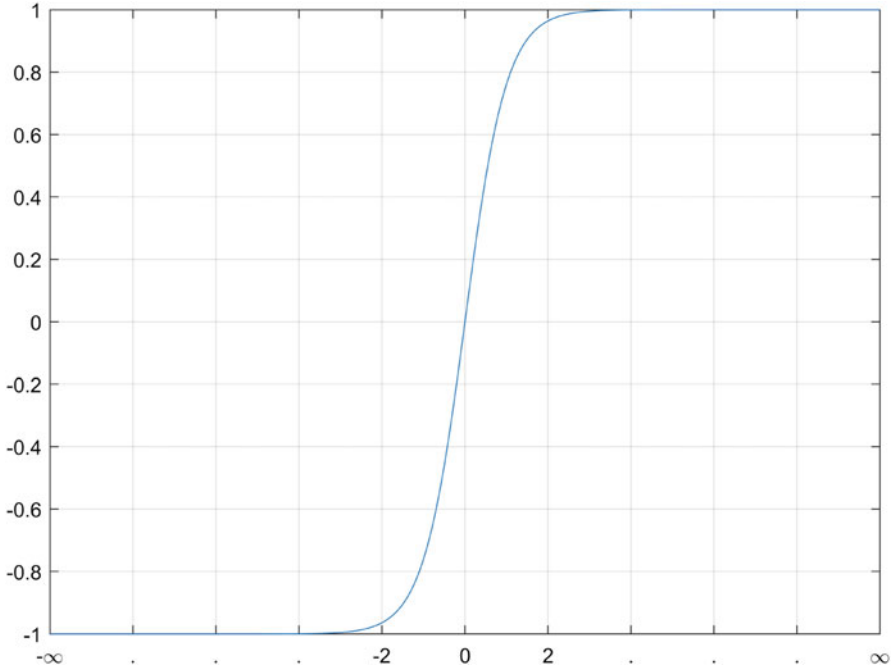
The major improvement in the MLP architecture comes in the way of using nonlinear mapping. Instead of using simple dot product of the input and weights, a nonlinear function, called as activation function, is used.

#### 6.3.2.1 **Activation Functions**

The most simple activation function is a step function, also called as a *sign* function, as shown in Fig. 6.6. This activation function is suited for applications like binary classification. However, as this is not a continuous function, it is not suitable for most training algorithms as we will see in the next section.



**Fig. 6.6** Activation function *sign*



**Fig. 6.7** Activation function *tanh*

The continuous version of step function is called as a hyperbolic tan or *tanh* function as shown in Fig. 6.7. Sometimes a variation of *tanh* function called as sigmoid function is used. Sigmoid function has exactly the same shape, but its value ranges from 0 to 1, instead of from  $-1$  to 1 in the case of *tanh* function.

The relationship between input  $\mathbf{x}$  and output  $Y$  at a given node that uses sigmoid activation function  $S(x)$  can be written as

$$S(\mathbf{x} \cdot \mathbf{w}) = y \quad (6.2)$$

where  $w$  represents weights for each input coming to the given node. This mapping ensures that the value of  $y$  will always be bounded between 0 and 1 irrespective of the values of inputs and the weights.

With the use of such nonlinear activation functions, MLP architecture is no more equivalent to the single layer and can now deal with more complex and nonlinear input-output mappings.

### 6.3.3 Training MLP

During the training process, the weights of the network are learned from the labelled training data using the process of backpropagation. Conceptually the process can be described as

1. Present the input to the neural network.
2. All the weights of the network are assigned some default value.
3. The input is transformed into output by passing through each node or neuron in each layer.
4. The output generated by the network is then compared with the expected output or label.
5. The error between the prediction and label is then used to update the weights of each node.
6. The error is then propagated in backward direction through every layer, to update the weights in each layer such that they minimize the error.

Reference [65] summarizes various backpropagation training algorithms commonly used in the literature along with their relative performances. We are not going to go into the mathematical details of these algorithms here, as the theory quickly becomes quite advanced and can make the topic very hard to understand. Also, as we look at the implementation of this algorithm using sklearn, we will see that with conceptual understanding of the training framework, one is sufficiently armed to apply these concepts to solve real problems.

Thus, backpropagation algorithm for training and feedforward operation for prediction mark the two phases in the operation of neural network. Backpropagation-based training can be done in two different methods.

1. Online or stochastic method
2. Batch method

#### 6.3.3.1 Online or Stochastic Learning

In this method a single sample or a small subset of an entire training set (drawn randomly) is sent as input to the network, and based on the output error, the weights are updated. The optimization method most commonly used to update the weights in this setup is called stochastic gradient descent or *SGD* method. The use of stochastic here implies that the samples are drawn randomly from the whole dataset rather than using them sequentially. The process can converge to the desired accuracy level even before all the samples are used. It is important to understand that in stochastic learning process, a small set of samples is used in each iteration, and the learning path is more noisy. The set of samples used in each iteration is called a mini-batch. SGD is beneficial when the expected learning path can contain multiple local minima and/or the size of training data is too large that using all the data in each iteration is not feasible.

### 6.3.3.2 Batch Learning

In batch method the entire training dataset is used and divided into small and deterministic set of batches (unlike stochastic method where samples are drawn randomly). The entire batch of samples is sent to the network before computing the error and updating the weights. After an entire batch is processed, the weights are updated. The training process using each batch is called as one iteration. When all the samples are used once, it is considered as one epoch in the training process. Typically multiple epochs are used before the algorithm fully converges. As the batch learning uses a batch of samples in each iteration, it reduces the overall noise, and the learning path is cleaner. However, the process is a lot more computation heavy and needs more memory and computation resources. Batch learning is preferred when computer hardware permits, and the learning path is expected to be relatively smooth.

### 6.3.4 Hidden Layers

The concept of hidden layers needs a little more explanation. As such they are not directly connected with inputs and outputs, and there is no theory around how many such layers are optimal in a given application. Each layer in MLP transforms the input to a new dimensional space. The hidden layers can have a higher dimensionality than the actual input, and thus they can transform the input into an even higher dimensional space. Sometimes, if the distribution of input in its original space has some nonlinearities and is ill conditioned, the higher dimensional space can help improve the distribution and as a result improve the overall performance. These transformations also depend on the activation function used. Increasing the dimensionality of hidden layer also makes the training process much more complicated, and one needs to carefully trade between the added complexity and performance improvement. Also, how many such hidden layers should be used is another variable where there are no theoretical guidelines. Both these parameters are called as hyperparameters, and one needs to do an open-ended exploration using a grid of possible values for them and then choose the combination that gives the best possible results within the constraints of the training resources.

### 6.3.5 Implementing MLP

We revisit the problem of classifying the Iris data, now with MLP architecture with logistic activation function and a single hidden layer with 50 nodes (the number 50 is selected as fairly arbitrary) (Fig. 6.8). We use all the other default parameters to train the model using sklearn library. We then apply on the same training data as before to see how the algorithm is able to model the data (Fig. 6.9).



```
from sklearn.neural_network import MLPClassifier
mlp = MLPClassifier(activation='logistic', hidden_layer_sizes=(50,))
mlp.fit(X,y)
y_pred = mlp.predict(X)
```

/usr/local/lib/python3.7/dist-packages/sklearn/neural\_network/\_multilayer\_perceptron.py:69: ConvergenceWarning,

Fig. 6.8 Implementing MLP using Google Colab and sklearn

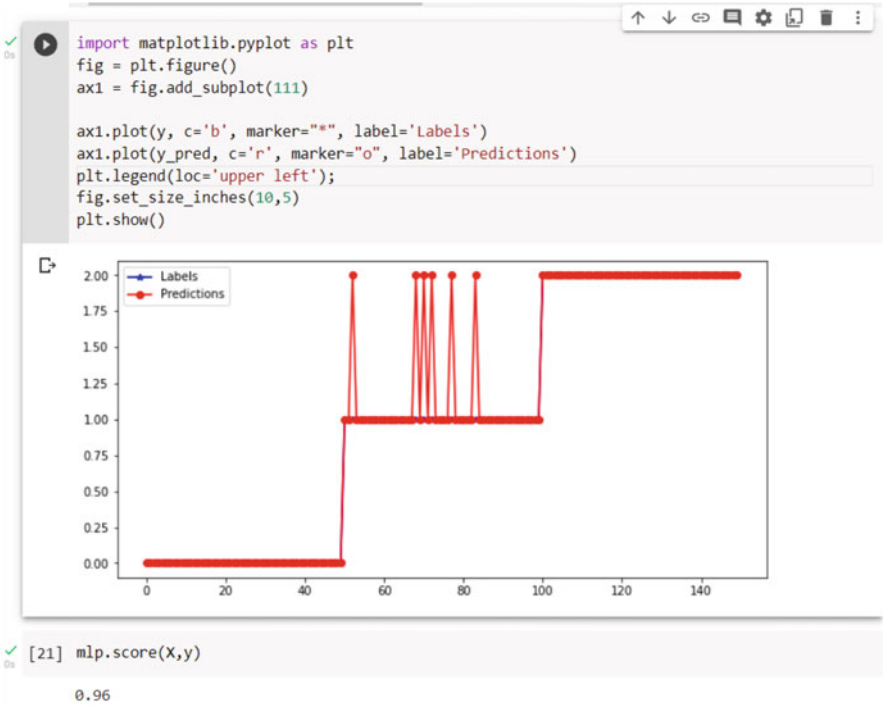
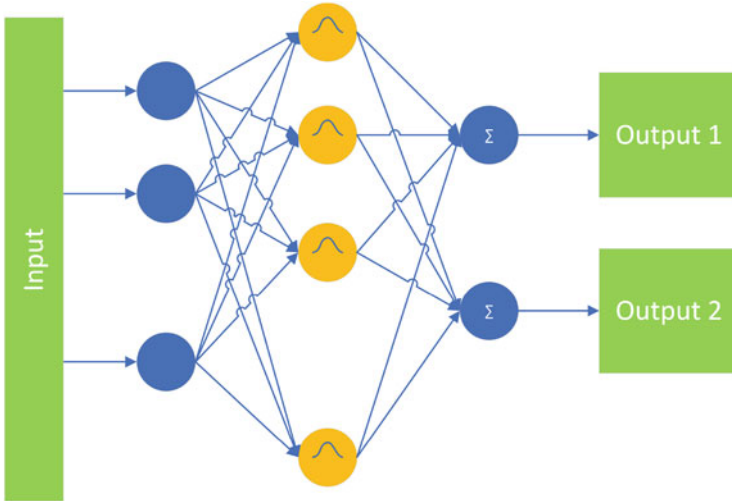


Fig. 6.9 Visualizing the training performance with MLP on Iris data

As you can see, the classification accuracy is order of magnitude better compared to a simple perceptron using linear mappings.

### 6.4 Radial Basis Function Networks

Radial basis function networks *RBFN* or radial basis function neural networks *RBFNN* are a variation of the feedforward neural networks (we will call them as RBF networks to avoid confusion). Although their architecture as shown in



**Fig. 6.10** Architecture of radial basis function neural network

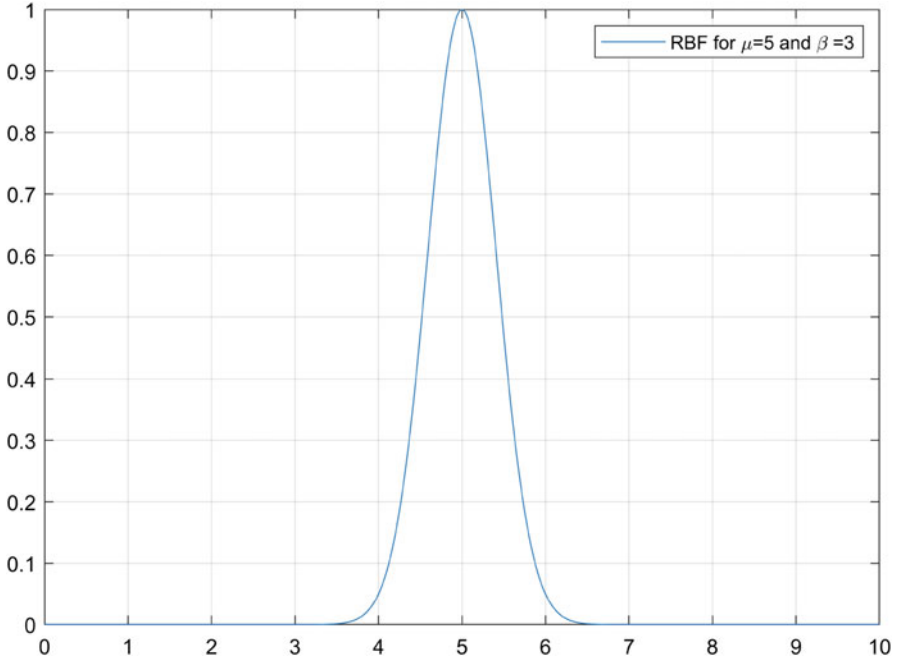
Fig. 6.10 looks similar to MLP as described above, functionally they are more close to the support vector machines with radial kernel functions. The RBF networks are characterized by three layers: an input layer, a single hidden layer, and an output layer. The input and output layers are linear weighing functions, and the hidden layer has a radial basis activation function instead of sigmoid-type activation function that is used in traditional MLP. The basis function is defined as

$$f_{RBF}(x) = e^{-\beta\|x-\mu\|^2} \quad (6.3)$$

The above equation is defined for a scalar input, but without lack of generality, it can be extended for multivariate inputs.  $\mu$  is called as center, and  $\beta$  represents the spread or variance of the radial basis function. It lies in the input space. Figure 6.11 shows the plot of the basis function. This plot is similar to Gaussian distribution.

### 6.4.1 Interpretation of RBF Networks

Aside from the mathematical definition, RBF networks have a very interesting interpretability that regular MLP does not have. Consider that the desired values of output form  $n$  number of clusters for the corresponding clusters in the input space. Each node in the hidden layer can be thought of as a representative of each transformation from the input cluster to the output cluster. As can be seen from Fig. 6.11, the value of radial basis function reduces to 0 rather quickly as the distance between the input and the center of the radial basis function  $\mu$  increases with respect



**Fig. 6.11** Plot of radial basis function

to the spread  $\beta$ . Thus, RBF network as a whole maps the input space to the output space by linear combination of outputs generated by each hidden RBF node. It is important to choose these cluster centers carefully to make sure the input space is mapped uniformly and there are no gaps. The training algorithm is capable of finding the optimal centers, but the number of clusters to use is a hyperparameter (in other words it needs to be tuned by exploration). If an input is presented to RBF network that is significantly different than the one used in training, the output of the network can be quite arbitrary. In other words the generalization performance of RBF networks in extrapolation situations is not good. However, if requirements for the RBF network are followed, it produces accurate predictions.

## 6.4.2 Implementing RBF Networks

We can implement RBF network with sklearn using *GaussianProcessClassifier* and selecting the kernel as *RBF*. Figure 6.12 shows the code for implementation. The following figures show the code for implementation. We will use all the default parameters for the model to focus on the concept. As can be seen, the accuracy is better than MLP, but marginally, and with some tuning both models can essentially produce similar results (Fig. 6.13).

```
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF

kernel = 1.0 * RBF(1.0)
rbfn = GaussianProcessClassifier(kernel=kernel, random_state=0)
rbfn.fit(X, y)
y_pred = rbfn.predict(X)
```

GaussianProcessClassifier(kernel=1\*\*2 \* RBF(length\_scale=1), random\_state=0)

Fig. 6.12 Implementing RBF network using Google Colab and sklearn

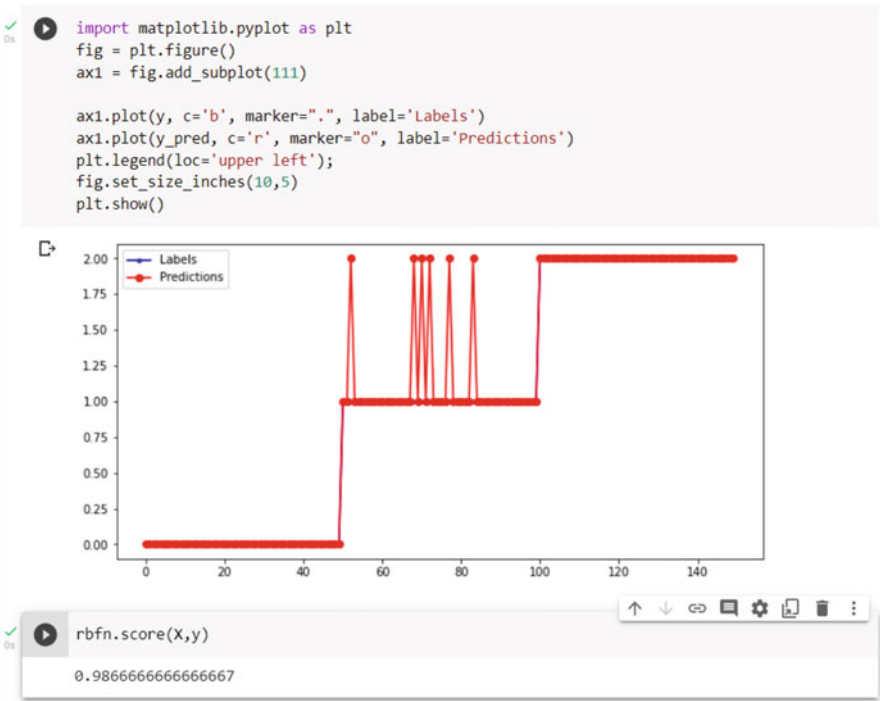


Fig. 6.13 Visualizing the training performance with RBF network on Iris data

## 6.5 Overfitting

Neural networks open up a feature-rich framework with practically unlimited scope to improve the performance for the given training data by increasing the complexity of the network. Complexity can be increased by manipulating various factors as follows:

1. Increasing the number of hidden layers
2. Increasing the nodes in hidden layers
3. Using complex activation functions

#### 4. Increasing the training epochs

Such improvements in training performance with arbitrary increase in complexity typically lead to overfitting. Overfitting is a phenomenon where we try to model the training data so accurately that in essence we just memorize the training data rather than identifying the generic patterns and structure in it. Such memorization leads to significantly worse performance on unseen data. However, determining the optimal threshold where the optimization should be stopped to keep the model generic enough but also accurate is not trivial. Multiple approaches are proposed in the literature, e.g., *Optimal Brain Damage* [67] or *Optimal Brain Surgeon* [66]. Before delving into the regularization techniques specifically tailored for neural networks, let's look at the concept of regularization in general.

### 6.5.1 Concept of Regularization

When we are dealing with any prediction problem, we have a loss function that we want to optimize using labelled training data. This optimization process leads to finding the values of the parameters of the equation and concludes the training. If we start with overly complex functional, the process can lead to what we just defined as memorization. In order to restrict the memorization, we can use a mathematical trick, by altering the optimization functional with some added terms. These added factors limit the range of values that the parameters of the algorithm can take and ultimately ensure that the memorization is avoided.

### 6.5.2 L1 and L2 Regularization

$$C(x) = L(x) + \lambda \sum \|\mathbf{W}\| \quad (6.4)$$

$$C(x) = L(x) + \lambda \sum \|\mathbf{W}\|^2 \quad (6.5)$$

The process of regularization that alters the optimization function (or loss function), also called as technique of Lagrangian multipliers, typically adds another term in the optimization problem that restricts the complexity of the network and is weighted by Lagrangian weighing factor  $\lambda$ . Equations 6.4 and 6.5 show the updated cost function  $C(x)$  use of L1 and L2 types of regularizations to reduce the overfitting.

$L(x)$  is the loss function that is dependent on the error in prediction, while  $\mathbf{W}$  stands for the vector of weights in the neural network. The L1 norm tries to minimize the sum of absolute values of the weights, while the L2 norm tries to minimize the sum of squared values of the weights. Each type has some pros and cons. The L1 regularization requires less computation but is less sensitive

to strong outliers, as well as prone to making the weights zero. In other words, L1 regularization tends to reduce the overall dimensionality of the problem by dropping some weights altogether. L2 regularization is typically overall a better metric and provides slow weight decay towards zero, but is more computation intensive. However, depending on the problem at hand, either one can be a better choice. Linear methods (regression/classification) that use L1 regularization are also called as Lasso methods, and the ones that use L2 regularization are also called as Ridge methods.

### 6.5.3 Dropout Regularization

This is an interesting method and is only applicable to the case of neural networks, while the L1 and L2 regularization can be applied to any algorithm. In dropout regularization, the neural network is considered as an ensemble of neurons in sequence, and instead of using a fully populated neural network, some neurons are randomly dropped from the path. The effect of each dropout on overall accuracy is considered, and after some iterations, an optimal set of neurons is selected in the final models. As this technique actually makes the model simpler rather than adding more complexity like L1 and L2 regularization techniques, this method is quite popular, specifically in the case of more complex and deep neural networks that we will study in later chapters.

## 6.6 Conclusion

In this chapter, we studied the machine learning model based on simple neural network. We studied the concept of single perceptron and its evolution into full-fledged neural network. We also studied the variation of the neural networks using radial basis function kernels. In the end we studied the effect of overfitting and how to reduce it using regularization techniques.

## 6.7 Exercises

1. Play with the parameters of MLP, e.g., the number of hidden layers, the number of nodes in each hidden layer, and activation function, and see the effect of the changes on the accuracy.
2. Separate the training and test data using `sklearn.model_selection.train_test_split` function. Use 70% training and 30% test, and repeat MLP training to see how the test accuracy compares with the earlier method of using the same data for training and test for MLP.

3. Play with all the parameters for RBF Network, and see the impact on the accuracy. Split the training and test data as before, and see the effect of that.
4. After fine-tuning all the models, see which model gives the absolute best accuracy, and also compare which model is easier to tune.
5. Ridge and Lasso regression models are offered in sklearn in *sklearn.linear\_model*. Try these two alternatives for perceptron in classification of Iris data, and compare the results.

# Chapter 7

## Decision Trees



### 7.1 Introduction

Decision trees represent conceptually as well as mathematically a different approach towards machine learning. While other approaches deal with the data that is strictly numerical, decision trees can deal with data in a more humanlike manner. The equations that define algorithms like logistic regression or neural networks cannot directly process a data that is not numerical, e.g., categorical or string type (albeit, there exist methods to convert non-numeric data into numerical form). However, the theory of decision trees does not rely on the data being numerical. While other approaches start by writing equations about the properties of data, decision trees start with constructing a tree-type structure such that at each node in the tree, there is a decision to be made. At heart, decision trees are heuristic structures that can be built by a sequence of choices or comparisons that are made in certain order.

To understand the process underlying decision trees, let's take an example of classifying different species on earth. We can start with asking questions like: "Can they fly?." Based on the answer, we can split the whole gamut of species into two parts: ones that can fly and ones that can't. Then, we go to the next step and consider species of each category to see if we can classify them further. Let's go to the branch of species that cannot fly. We ask another question: "How many legs do they have?." Based on this answer, we create multiple branches with answers like species with two legs, species with four legs, species with six legs, and so on. There is no need to limit on how many such categories there can be. But for better tractability, we can restrict the number of categories by creating a category as species with more than six legs and group all the species with more than six legs into a single category. We can either ask the same question for the flying species or ask a different question. As we are at a different decision point in the tree, there is no restriction as to which question to ask. We can then continue splitting the species at each level till we reach the leaf nodes such that there is only one single species there. This approach essentially summarizes the conceptual process of building a decision tree.



Although the above process describes the high-level operation underlying a decision tree, the actual building process for a decision tree in a generalized setting is much more complex. The reason for complexity lies in answering the following: “How to choose which questions to ask and in which order?” One can always start asking random questions and ultimately still converge on the full solution, but when the data is large and highly dimensional, this random or brute force approach invariably becomes suboptimal. One also has to remember that in almost all the real-life situations where decision tree or any other classifier is applied, the data is not necessarily completely classifiable. There exists noise in the data that limits the accuracy of the classifier. We need to build the decision trees within the following constraints:

1. Have the minimum number of levels in the tree to limit the complexity of the tree.
2. Have the maximum accuracy in classifying the labelled test data.
3. Have the most generalization capability when applied to the unseen data.

There is never a single perfect approach that tackles all these constraints in best possible manner. There are multiple variations of the implementations of this concept that are widely used in the machine learning applications.

## 7.2 Why Decision Trees?

Before going into the details of decision tree theory, let’s understand why decision trees are important. Here are the unique advantages of using decision tree algorithms for reference:

1. More humanlike and intuitive behavior. It is also one of the most easily explainable or interpretable machine learning models available, meaning it is very easy to explain why a certain sample is classified to the predicted class in a humanlike manner. Most other methods (e.g., neural networks, support vector machines) are more of black box making such humanlike explanations rather difficult or even impossible.
2. Can work directly on non-numeric data, e.g., categorical, without having the additional processing to convert the data into numerical format, which tends to add some bias to the data.
3. Can work directly with missing data. As a result missing data prediction step is not required.
4. Decision tree algorithms scale easily from linear data to nonlinear data without any change in core logic.

### 7.2.1 Types of Decision Trees

Based on the application (classification or regression), there are some differences in how the trees are built, and consequently they are called classification decision trees and regression decision trees. However, we are going to treat the machine learning techniques based on applications in the next part of the book, and in this chapter, we will focus on the fundamental concepts underlying the decision trees that are common between both applications.

## 7.3 Algorithms for Building Decision Trees

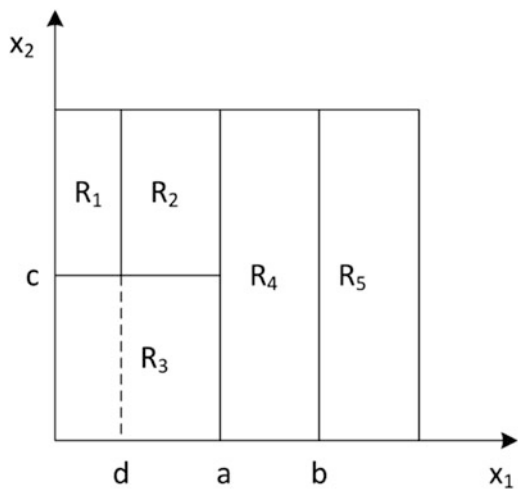
The most commonly used algorithms for building decision trees are as follows:

- CART or Classification and Regression Tree
- ID3 or Iterative Dichotomiser 3
- CHAID or Chi-Squared Automatic Interaction Detector

CART or classification and regression tree is a generic term used for describing the process of building decision trees as described by Breiman-Friedman [57]. ID3 [62] is a variation of CART methodology with slightly different use of optimization method. CHAID uses a significantly different procedure, and we will study it separately.

The development of classification trees is slightly different but follows similar arguments as a regression tree. Let's consider a two-dimensional space defined by axes  $(x_1, x_2)$ . The space is divided into five regions ( $R_1, R_2, R_3, R_4, R_5$ ) as shown in the figure, using a set of rules as defined in Fig. 7.1.

**Fig. 7.1** Rectangular regions created by decision tree



### 7.4 Regression Tree

Regression trees are the trees that are designed to predict the value of a function at given coordinates. Let us consider a set of N-dimensional input data  $\{\mathbf{x}_i, i = 1, \dots, p$  and  $\mathbf{x}_i \in \mathfrak{R}^n\}$ . The corresponding outputs are  $\{y_i, i = 1, \dots, p$  and  $y_i \in \mathfrak{R}\}$ . In case of regression trees, it is required that the input and output data is strictly numerical and not categorical. Once the labelled training data is available, the algorithm builds a set of rules. How many rules should be used, what dimensions to use, and when to terminate the tree are all the parameters that the algorithm needs to optimize based on the desired error rate.

Based on the example shown in Figs. 7.2 and 7.1, let the classes be regions  $R_1 - R_5$  and the input data two dimensional. In such case, the desired response of the decision tree is defined as

$$t(x) = r_k \forall x_i \in R_k \tag{7.1}$$

where  $r_k \in \mathfrak{R}$  is a constant value of output in region  $R_k$ . If we define the optimization problem as minimizing the mean square error,

$$\sum_{i=1}^{i=p} (y_i - t(x_i))^2 \tag{7.2}$$

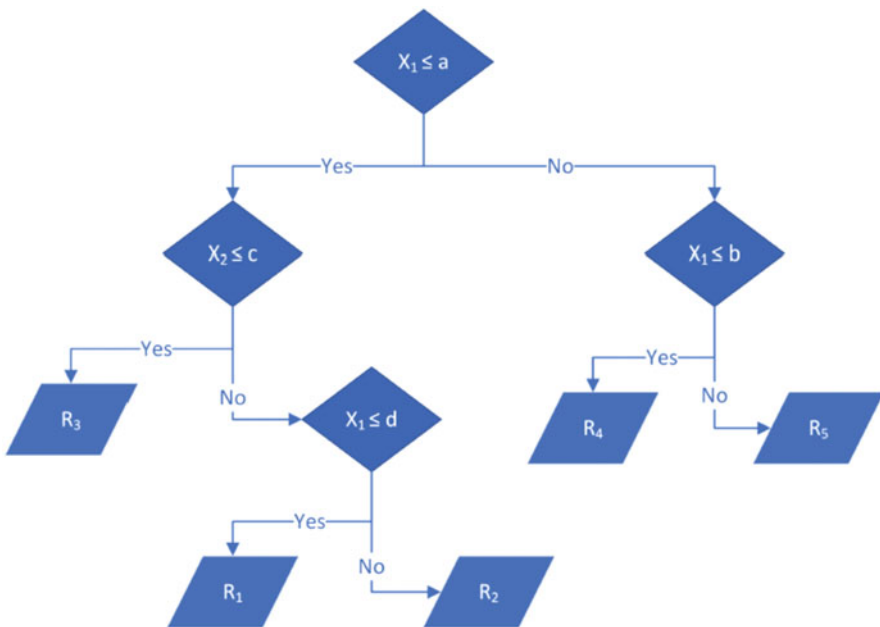


Fig. 7.2 Hierarchical rules defining the decision tree

then a simple calculation would show that the estimate for  $r_k$  is given by

$$r_k = \text{ave}(y_i | x_i \in R_k) \quad (7.3)$$

Solving the problem to find the globally optimum regions to minimize the mean square error is an NP-hard problem and cannot be solved in general in finite time. Hence, greedy methods resulting in local minimum are employed. Such greedy methods typically result in a large tree that overfits the data. Let us denote such large tree as  $T_0$ . Then, the algorithm must apply a pruning technique to reduce the tree size to find the optimal tradeoff that captures the most of the structure in the data without overfitting it. This is achieved by using squared error node impurity measure optimization as described in [57].

### 7.4.1 Implementing Regression Tree

We will use `DecisionTreeRegressor` from `sklearn.tree` to illustrate the use of regression trees. We will use `sklearn`'s built-in function to generate random dataset to test the regression model (Fig. 7.3).

In the first attempt, we limit the maximum depth of the tree to 4. As we can see, the mapping is not great, and we can see the predictions lying in multiple bands defined by the rules up to a depth of 4 (Fig. 7.4).

Now we increase the depth to 10 and see the improvement in the predictions (Fig. 7.5).

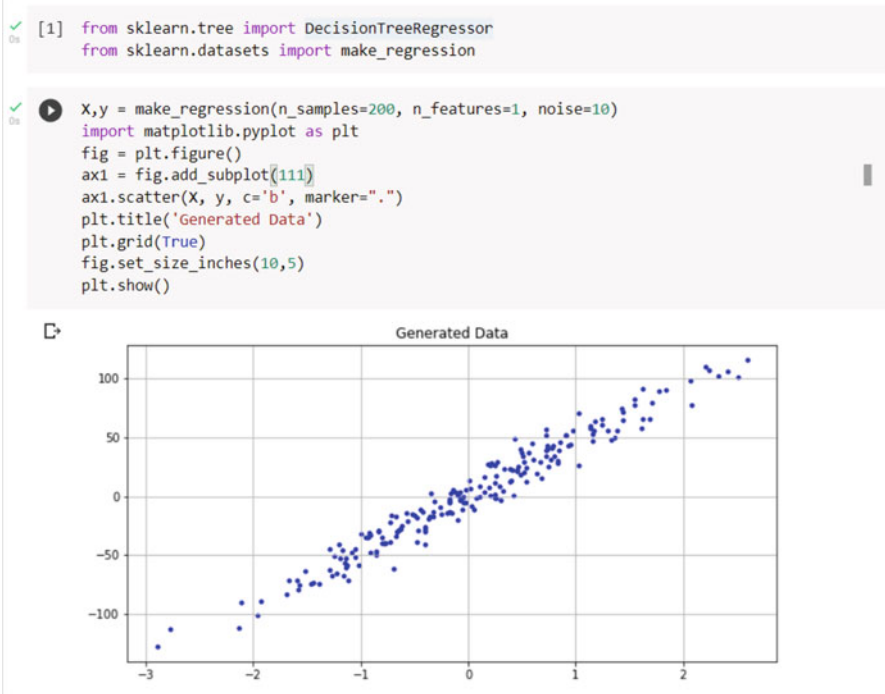
## 7.5 Classification Tree

In case of classification, the output is not a continuous numerical value, but a discrete class label. The development of the large tree follows the same steps as described in the regression tree subsection, but the pruning methods need to be updated as the squared error method is not suitable for classification. Three different types of measures are popular in the literature:

- Misclassification error
- Gini index or Gini impurity
- Cross-entropy or deviance

### 7.5.1 Defining the Terms

Let there be “ $k$ ” classes and “ $n$ ” nodes. Let the frequency that class- $m$  is predicted at node- $i$  be denoted as  $f_{mi}$ . The fraction of the classes predicted as  $m$  at node  $i$  be



**Fig. 7.3** Creating synthetic data with noise for testing regression tree. We are creating 200 single-dimensional samples for better visualization. The noise level is selected at 10

denoted as  $p_{mi}$ . Let the majority class at node  $m$  be  $c_m$ . Let us denote the fraction of classes  $c_m$  at node  $m$  would be  $p_{mc_m}$ .

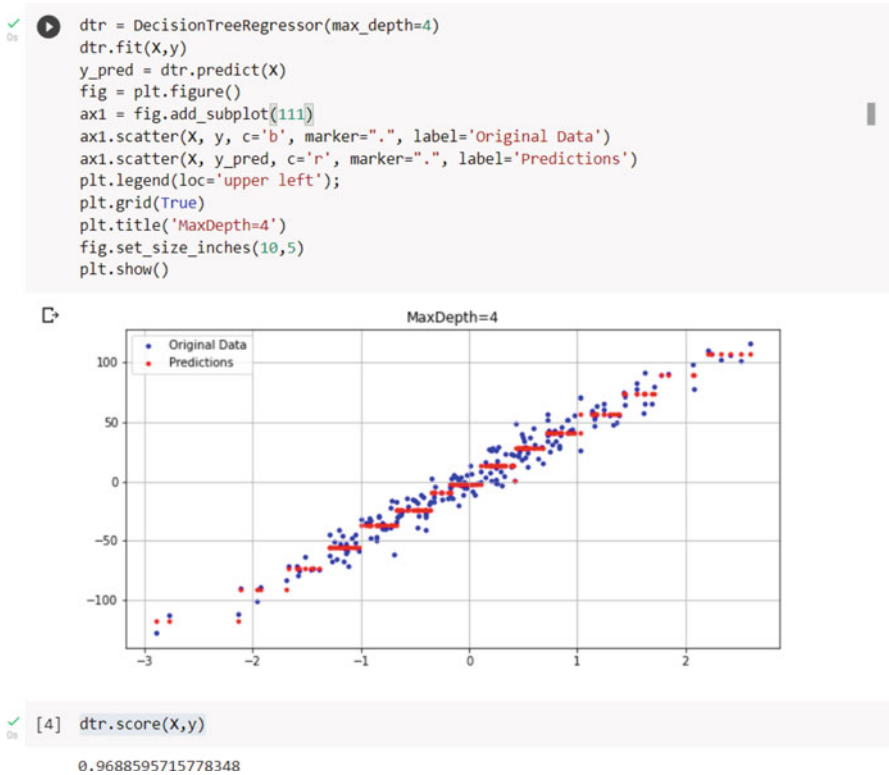
Now we are ready to define the metrics used for making the decision at each node. Differences in the metric definition separate the different decision tree algorithms.

### 7.5.2 Misclassification Error

Based on the variables defined above, the misclassification rate is defined as  $1 - p_{mc_m}$ . As can be seen from Fig. 7.6, this rate is not a continuous function and hence cannot be differentiated. However, this is one of the most intuitive formulations and hence is fairly popular.

### 7.5.3 Gini Index

Gini index is the measure of choice in CART algorithm. Concept of the Gini index can be defined as the probability of misclassification of a randomly selected input



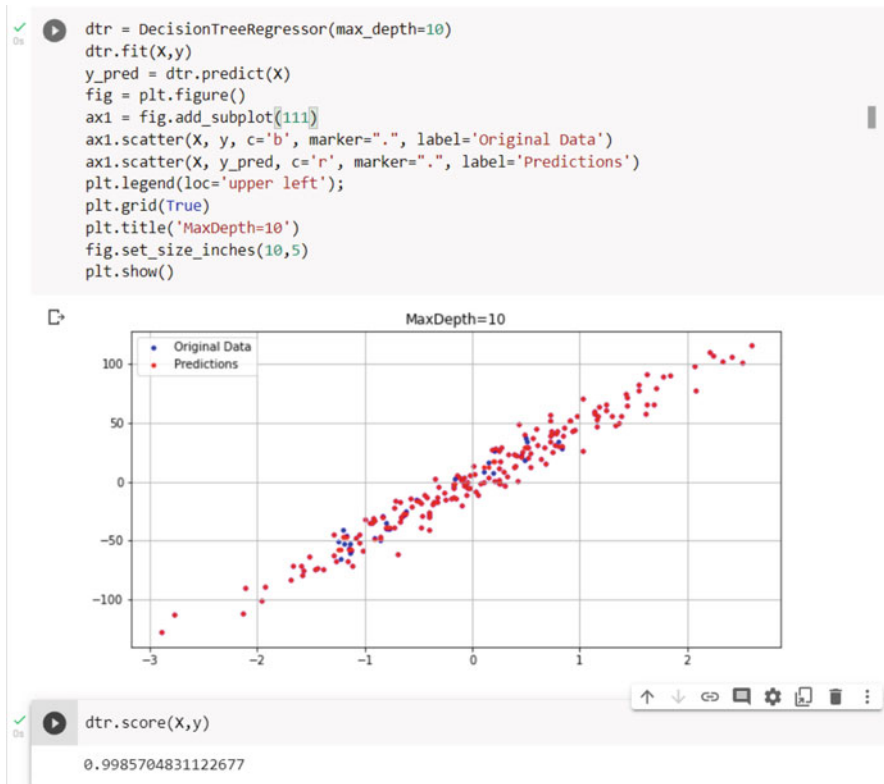
**Fig. 7.4** Regression tree with a maximum depth of 4. The predictions shown in red are not great and are clearly separated into multiple bands showing that the max depth is not sufficient

sample if it was labelled based on the distribution of the classes in the given node. Mathematically it is defined as

$$G = \sum_{m=1}^{m=k} p_{mi}(1 - p_{mi}) \tag{7.4}$$

It can be further simplified as

$$\begin{aligned}
 G &= \sum_{m=1}^{m=k} p_{mi}(1 - p_{mi}) \\
 G &= \sum_{m=1}^{m=k} p_{mi} - \sum_{m=1}^{m=k} p_{mi}^2 \\
 G &= 1 - \sum_{m=1}^{m=k} p_{mi}^2
 \end{aligned}$$



**Fig. 7.5** Regression tree with a maximum depth of 10. With this depth the regression tree is able to map the data quite well

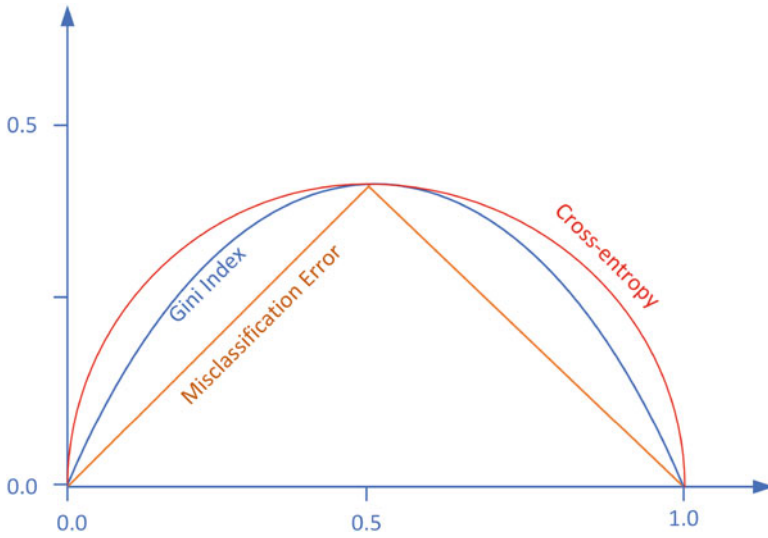
As the plot in Fig. 7.6 shows, this is a smooth function of the proportion and is continuously differentiable and can be safely used in optimization.

### 7.5.4 Cross-Entropy or Deviance

Cross-entropy is an information theoretic metric defined as

$$\mathcal{E} = - \sum_{m=1}^{m=k} p_{mi} \log(p_{mi}) \tag{7.5}$$

This definition resembles classical entropy of a single random variable. However, as the random variable here is already a combination of the class prediction and nodes of the tree, it is called as cross-entropy. ID3 models use cross-entropy as the



**Fig. 7.6** The plot of decision metrics for a case of two-class problem. X-axis shows the proportion in class 1. Curves are scaled to fit, without loss of generality

measure of choice. As the plot in figure shows, this is also a smooth function of the proportion and is continuously differentiable and can be safely used in optimization.

## 7.6 CHAID

Chi-square automatic interaction detector or *CHAID* is a decision tree technique that finds its origin in statistical chi-square test for goodness of fit. It was first published by G. V. Kass in 1980, but some parts of the technique were already in use in the 1950s. This test uses the chi-square distribution to compare a sample with the population and predict whether the sample belongs to the population at desired statistical significance. CHAID technique uses this theory to build a decision tree. Due to the use of chi-square technique in building a decision tree, this method is quite different compared to any other types of decision trees discussed so far. The following subsection discusses the details of the algorithm briefly.

### 7.6.1 CHAID Algorithm

The first task in building the CHAID tree is to find the most dependent variable. This is in a way directly related to what is the final application of the tree. The algorithm works best if a single desired variable can be identified. Once such variable is



identified, it is considered as root node. Then the algorithm tries to split the node into two or more nodes, called as initial or parent nodes. All the subsequent nodes are called as child nodes, till we reach the final set of nodes that cannot be split any further. These nodes are called as terminal nodes. Splitting at each node is entirely based on statistical dependence as dictated by chi-square distribution in case of categorical data and by F-test in case of continuous data. As each split is based on dependency of variables, unlike a more complex expression like Gini impurity or cross-entropy in case of CART- or ID3-based trees, the tree structure developed using CHAID is more interpretable and human readable in most cases.

## 7.7 Training Decision Tree

We are not going into full mathematical details of building a decision tree using CART or ID3, but the following steps will explain the methodology with sufficient details and clarity.

### Steps

1. Start with the training data.
2. Choose the metric of choice (Gini index or cross-entropy).
3. Choose the root node, such that it splits the data with optimal values of metrics into two branches.
4. Split the data into two parts by applying the decision rule of root node.
5. Repeat the steps 3 and 4 for each branch.
6. Continue the splitting process till leaf nodes are reached in all the branches with predefined stop rule.

### 7.7.1 *Depth of Decision Tree*

Typically a decision tree is characterized by its depth. Depth of the tree refers to the maximum number of nodes one needs to traverse to reach the final leaf node starting at the root node. To give an example: in a given tree, one can reach a leaf node with only say 3 nodes, while another leaf node needs 7 nodes to be traversed. In such case the depth of the tree is 7 and not 3 or any intermediate number. The deeper the tree, the more complex patterns it can represent in the data, but arbitrarily increasing the depth of the tree can lead to unnecessary complexity, and it starts to lose its generalization capability. In the optimization steps in constructing decision trees, reducing the depth of the tree while maintaining the accuracy in test data is one of the important steps.

## 7.8 Ensemble Decision Trees

In the previous sections, we learned ways to develop a single decision tree based on different techniques. In many situations such trees work very well, but there are ways to extract more performance out of the similar architecture if we create multiple such trees and aggregate them. Such techniques are called as ensemble methods, and they typically deliver superior performance at the cost of computation and algorithmic complexity. In ensemble methods, a single decision tree is called as a single learner or weak learner, and the ensemble methods deal with a group of such weak learners.

There are various approaches proposed in the literature that can successfully combine multiple weak learners to create a strong overall model.<sup>1</sup> Each weak learner in the ensemble of learners captures certain aspects of the information contained in the data that is used to train it. The job of ensemble tree is to optimally unite the weak learners to have a better overall accuracy. The primary advantage of ensemble methods is reduction in overfitting.

There are three main types of ensembles:

1. Bagging
2. Random forest
3. Boosting

### 7.8.1 Bagging Ensemble Trees

The term bagging finds its origins in *bootstrap aggregation*. Coincidentally, the literal meaning of bagging, which means putting multiple decision trees in a *bag*, is not too far from the way the bagging techniques work. Bagging technique can be described using the following steps:

1. Split the total training data into a predetermined number of sets with random sampling with replacement. The term *with replacement* means that the same sample can appear in multiple sets. Each sample is called as *bootstrap* sample.
2. Train the decision tree using CART or ID3 method using each of the dataset.
3. Each learned tree is called as a weak learner.
4. Aggregate all the weak learners by averaging the outputs of individual learners for the case of regression, and aggregate all the individual weak learners by

---

<sup>1</sup> The words *weak* and *strong* have a different meaning in this context. A weak learner is a decision tree that is trained using only a fraction of the total data and is not capable or even expected of giving metrics that are close to the desired ones. Theoretical definition of a weak learner is one whose performance is only slightly better than pure random chance. A strong learner is a single decision tree that uses all the data and is capable of producing reasonably good metrics. In ensemble methods individual tree is always a weak learner as it is not exposed to the full dataset.

voting for the case of classification. The aggregation step involves optimization, such that prediction error is minimized.

5. The output of the aggregate or ensemble of the weak learners is considered as the final output.

The steps described above seem quite straightforward and does not really involve any complex mathematics or calculus. However, the method is quite effective. If the data has some outliers,<sup>2</sup> a single decision tree can get affected by it more than an ensemble can be. This is one of the inherent advantages of bagging methods.

## 7.8.2 *Random Forest Trees*

The bagging process described above improves the resilience of the decision trees with respect to outliers. Random forest methods go one step forward to make the ensemble even more resilient in case of varying feature importances. Even after using a carefully crafted feature space, not all features are equally influential on the outcome. Also certain features can have some interdependencies that can affect their influence on the outcome in a counterproductive manner. Random forest tree architecture improves model performance in such situations over previously discussed methods by partitioning feature space as well as data for individual weak learner. Thus, each weak learner sees only a fraction of samples and a fraction of features. The features are also sampled randomly with replacement [68], as data is sampled with replacement in bagging methods. The process is also called as random subspace method, as each weak learner works in a subspace of features. In practice, this sampling improves the diversity among the individual trees and overall makes the model more robust and resilient to noisy data. The original algorithm proposed by *Tin Ho* was then extended by *Leo Breiman* et al. [69] to merge the multiple existing approaches in the literature to what is now commonly known as random forest method.

### 7.8.2.1 **Decision Jungles**

Recently a modification to the method of random forests was proposed in the form of *decision jungles* [81]. One of the drawbacks of random forests is that they can grow exponentially with data size and if the compute platform is limited by memory, the

---

<sup>2</sup> Outliers represent an important concept in the theory of machine learning. Although, its meaning is obvious, its impact on learning is not quite trivial. An outlier is a sample in training data that does not represent the generic trends in the data. Also, from mathematical standpoint, the distance of an outlier from other samples in the data is typically large. Such large distances can throw a machine learning model significantly away from the desired behavior. In other words, a small set of outliers can affect the learning of a machine learning model adversely and can reduce the metrics significantly. It is thus an important property of a machine learning model to be resilient of a reasonable number of outliers.

depth of the trees needs to be restricted. This can result in suboptimal performance. Decision jungles propose to improve on this by representing each weak learner in random forest method by a directed acyclic graph DAG instead of an open-ended tree. The DAG has capability to fuse some of the nodes, thereby creating multiple paths to a leaf from a root node. As a result decision jungles can represent the same logic as random forest trees but in a significantly compact manner.

### 7.8.3 *Boosted Ensemble Trees*

A fundamental difference between boosted and bagging (or random forest for that matter) is the sequential training of the trees against the parallel training. In bagging or random forest methods, all the individual weak learners are generated using random sampling and random subspaces. As all the individual weak learners are independent of each other, they all can be trained in parallel. Only after they are completely trained that their results are aggregated. Boosting technique employs a very different approach, where first the tree is trained based on a random sample of data. However, the data used by the second tree depends on the outcome of training of the first tree. The second tree is used to focus on the specific samples where the first decision tree is not performing well. Thus, the training of the second tree is dependent on the training of the first tree, and they cannot be trained in parallel. The training continues in this fashion to the third and fourth tree and so on. Due to the unavailability of parallel computation, the training of boosted trees is significantly slower than training trees using bagging and random forest. Once all the trees are trained, then the output of all individual trees is combined with necessary weights to generate final output. In spite of the computational disadvantage exhibited by the boosted trees, they are often preferred over other techniques due to their superior performance in most cases.

#### 7.8.3.1 **AdaBoost**

AdaBoost was one of the first boosting algorithms proposed by Yoav Freund and Robert Schapire [71]. The algorithm was primarily developed for the case of binary classification, and it was quite effective in improving the performance of a decision tree in a systematic iterative manner. The algorithm was then extended to support multi-class classification as well as regression.

#### 7.8.3.2 **Gradient Boosting**

Breiman proposed an algorithm called as *ARCing* [70] meaning *Adaptive Reweighting and Combining* algorithms. This algorithm marked the next step in improving the capabilities of boosting type methods using statistical framework. Gradient

boosting in a way is a generalization of AdaBoost algorithm using statistical framework developed by Breiman and Friedman [57]. In gradient-boosted trees, the boosting problem is stated as numerical optimization problem with the objective to minimize the error by sequentially adding weak learners using gradient descent algorithm. Gradient descent being a greedy method, gradient boosting algorithm is susceptible to overfitting the training data. Hence, regularization techniques are always used with gradient boosting to limit the overfitting.

## 7.9 Implementing a Classification Tree

We will use the same example of Iris data to illustrate the implementation of decision tree classifier. Figures 7.7 and 7.8 show the code along with the accuracy of the algorithm. As the Iris data contains only 150 samples between 3 classes, both the algorithms are providing similar performance. However, when the data gets larger and more complex, ensemble methods tend to train faster and provide better accuracy. Restating that we are using the same data for training and finding the accuracy only to compare how different algorithms model the patterns in the data. In most real-life situations, we use different sets for training and testing.

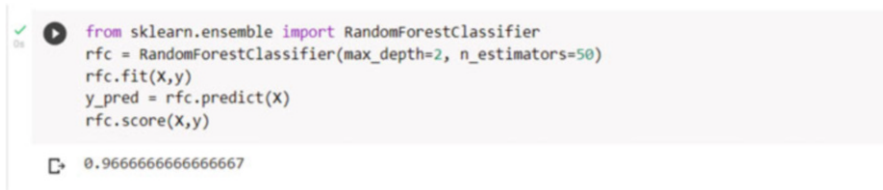


```
[1] from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target

from sklearn.tree import DecisionTreeClassifier
dtc = DecisionTreeClassifier(max_depth=2)
dtc.fit(X,y)
y_pred = dtc.predict(X)
dtc.score(X,y)

0.96
```

Fig. 7.7 Implementing decision tree classifier using Iris dataset with a max depth of 2



```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(max_depth=2, n_estimators=50)
rfc.fit(X,y)
y_pred = rfc.predict(X)
rfc.score(X,y)

0.9666666666666667
```

Fig. 7.8 Implementing decision tree classifier using Iris dataset with a max depth of 2 and 50 weak learners in the ensemble

## 7.10 Conclusion

In this chapter we studied the concept of decision trees. These methods are extremely important and useful in many applications encountered in practice. They are directly motivated by the hierarchical decision-making process very similar to the human behavior in tackling real-life problems and hence are more intuitive than the other methods. Also, the results obtained using decision trees are easier to interpret, and these insights can be used to determine action to be taken after knowing the results. We also looked at ensemble methods that use aggregate of multiple decision trees to optimize the overall performance and make the models more robust and generic.

## 7.11 Exercises

1. Create a manual decision tree to classify languages of the world. You can limit the total number of languages to a reasonable number, say 50 or 70.
2. Try to solve the same problem using either different way: either different set of questions or in different sequence. Compare the complexity of the two trees in the form of depth of the tree and number of nodes.
3. Try to play with different parameters in the regression tree as illustrated in the chapter and compare the performance.
4. Split the total data into training and testing, apply the models, and compare the performance differences.
5. Play with tree depth and other parameters in the decision tree and ensemble decision tree options including random forest, AdaBoost, `baggingclassifier`, `gradientboostingclassifier`, and so on. Compare the performances of these models, and see which one gives the best results.
6. There are a couple other machine learning libraries, e.g., LightGBM and XGBoost, that are well known for their improved decision tree implementations. Try to use these libraries and compare the performance with sklearn options. For importing LightGBM in Google Colab, use the code `import lightgbm`, and for XGBoost, use `import xgboost`.

# Chapter 8

## Support Vector Machines



### 8.1 Introduction

The development of the theory of support vector machines, commonly known as *SVMs*, is typically attributed to Vladimir Vapnik. Vapnik was born in the Soviet Union or present-day Russia but later moved to the United States. His primary research happened during his tenure in AT&T Bell Labs. He is also known for statistical learning theory that he co-developed with Chevronekns. In the early 1960s, he was working in the field of optimal pattern recognition using statistical methods. His paper with Alexander Lerner [72] on generalized portrait algorithm marks the origin of the concept of support vector machines. In the early days, the method was primarily used to solve the problem of classification using construction of optimal hyperplane that separates the given classes with a maximum separation.

### 8.2 Motivation and Scope

The original SVM algorithm targeted the solution of simple binary classification problem. Figure 8.1 shows the concept of SVM in case of classes that are linearly separable. The algorithm of SVM tries to separate the two classes with maximal separation using select number of data points, also called as support vectors, as shown in Fig. 8.1 as well as Fig. 8.2. Figure 8.1 shows the case of linearly separable classes where the result is trivial. The solid line represents the hyperplane<sup>1</sup> that optimally separates the two classes. The dotted lines represent the boundaries of

---

<sup>1</sup> A hyperplane is a generic term used to represent a linear plane in  $n$  dimensions. In one-dimensional space, it is a dot; in two-dimensional space, it is a line; and in three-dimensional space, it is a regular plane. The same concept can be extended to higher dimensions where the geometric manifold cannot be directly visualized, but can be mathematically modelled.

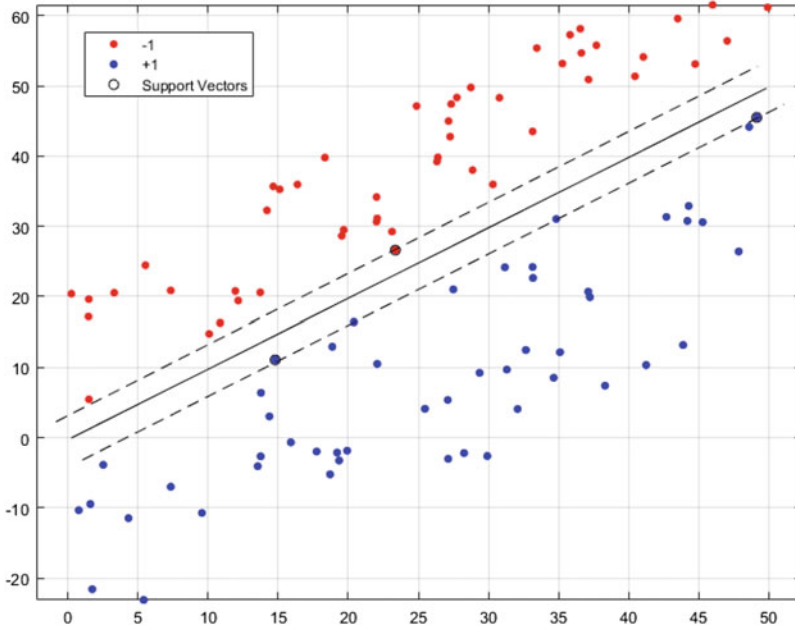


Fig. 8.1 Linear binary SVM applied on separable data

the classes as defined by the support vectors. The class separating hyperplane tries to maximize the distance between the class boundaries. However, as can be seen in Fig. 8.2, where the classes are not entirely linearly separable, the algorithm still finds the optimal support vectors and the classification hyperplane. Once the support vectors are identified, the hyperplane can be constructed, and for the rest of the classification process, one does not need the rest of the samples for predicting the class. Typically the number of support vectors is a very small fraction of the entire set of samples. The beauty of the algorithm lies in the drastic reduction in the amount of information that needs to be preserved for classification, compared to other algorithms like k-nearest neighbors. It is important to note that the process of regularization is fundamentally based on the computation of hyperplane, and it makes the algorithm intrinsically more generalizable.

### 8.2.1 Extension to Multi-class Classification

As per the conceptual setup of SVM, it is meant to solve only binary classification problems and cannot be directly applied to solving the problem of multi-class classification. However there are few approaches that are commonly used to extend the framework for such cases. One approach is to use SVM as binary classifier to



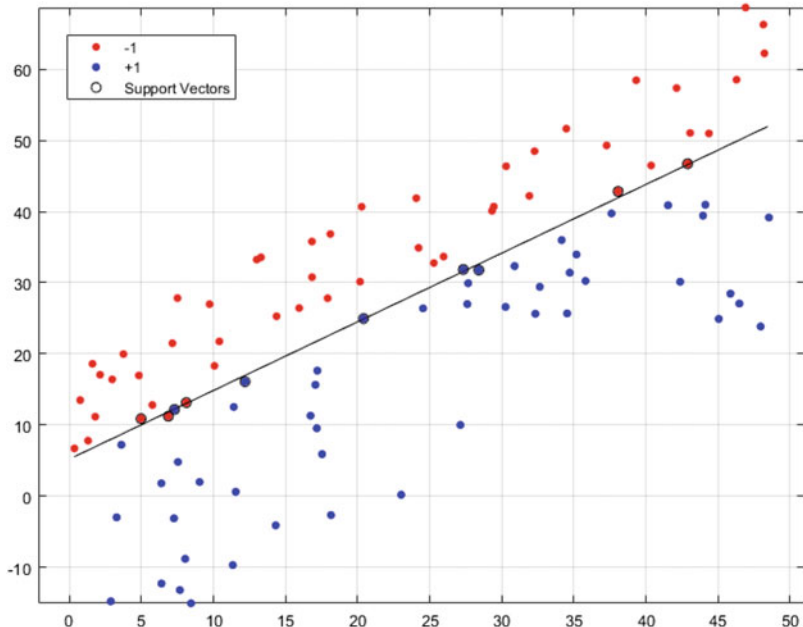


Fig. 8.2 Linear binary SVM applied on non-separable data

separate each pair of classes from the given multi-class problem and then apply some heuristic to predict the class for each sample. This is a suboptimal and time-consuming method and not typically preferred. To illustrate the inefficiency of this method, let's consider a case of three-class problem: one has to train a SVM for separating classes 1-2, 1-3, and 2-3, thereby training three separate SVMs. However, the complexity will increase in polynomial rate with more classes. Specifically, using the notation of combinations, we can state that  ${}^nC_2$  number of SVMs are required for the case of  $n$  classes.

In an alternative approach, a binary SVM is used to separate each class from the rest of the classes. With this approach, for three-class problem, one still has to train three SVMs, 1-(2,3), 2-(1,3), and 3-(1,2). However, with further increase in the number of classes, the complexity increases linearly instead of polynomially.

### 8.3 Theory of SVM

In order to understand how the SVMs are trained, it is important to understand the theory behind SVM algorithm. This can get highly mathematical and complex; however, I am going to try to avoid the gory details of derivations. The reader is encouraged to read [55] for detailed theoretical overview. I will state the

assumptions on which the derivations are based and then move to final equations that can be used to train the SVM without missing on the essence of SVM.

Let us consider a binary classification problem with  $n$ -dimensional training dataset consisting of  $p$  pairs  $(\mathbf{x}_i, y_i)$ , such that  $\mathbf{x}_i \in \mathfrak{R}^n$  and  $y_i \in \{-1, +1\}$ . Let the equation of the hyperplane that separates the two classes with maximal separation be given as

$$(\mathbf{w} \cdot \mathbf{x}) - w_0 = 0 \quad (8.1)$$

Here  $\mathbf{w} \in \mathfrak{R}^n$ , same as  $\mathbf{x}$ . For the samples belonging to each class we can write

$$(\mathbf{w} \cdot \mathbf{x}_i) - w_0 \begin{cases} > 1, & \text{if } y_i = 1 \\ \leq 1, & \text{if } y_i = -1 \end{cases} \quad (8.2)$$

The two equations in 8.2 can be combined into a single equation as

$$y_i[(\mathbf{w} \cdot \mathbf{x}) - w_0] \geq 1, \quad i = 1, \dots, p \quad (8.3)$$

The above equation can be solved to get infinite solutions for weight vector. Here we apply the concepts from regularization to get the solution that also minimizes a Lagrange multiplier in the form of  $\Phi(\mathbf{w})$ , where  $\Phi(\mathbf{w})$  is given as

$$\Phi(\mathbf{w}) = \|\mathbf{w}'\|^2 \quad (8.4)$$

where  $\mathbf{w}'$  is an  $(n + 1)$  dimensional vector that is a combination of  $\mathbf{w}$  and  $w_0$ .

Thus, we have arrived at a point where we can define the optimization problem that needs to be solved. To obtain the precise equation of the hyperplane, we need to minimize the functional  $\Phi(\mathbf{w}')$  with constraints defined in Eq. 8.3. We can now combine the two equations using *Lagrangian* multiplier technique into a single functional that is to be minimized:

$$\min_{\mathbf{w}} \left\{ \left( \frac{1}{n} \sum_{i=1}^n 1 - y_i[(\mathbf{w} \cdot \mathbf{x}) - w_0] \right) + \lambda \Phi(\mathbf{w}) \right\} \quad (8.5)$$

With some manipulations, the Lagrangian in Eq. 8.5 reduces to a subset that contains only a very small number of training samples that are called as support vectors. As can be seen from Fig. 8.1, these support vectors are the vectors that represent the boundary of each class. After some more mathematical trickery performed using well-known Kühn-Tucker conditions, one arrives at a convex quadratic optimization problem, which is relatively straightforward to solve. The equation to compute the optimal weight vector  $\hat{\mathbf{w}}$  can then be given in terms of Lagrange multipliers  $\alpha_i \geq 0$  as

$$\hat{\mathbf{w}} = \sum_{\text{support vectors only}} y_i \alpha_i \mathbf{x}_i \quad (8.6)$$

Once these parameters are computed as part of training process, the classifying function can be given as (for a given sample  $\mathbf{x}$ )

$$f_c(x) = \text{sign} \left( \sum_{\text{support vectors only}} y_i \alpha_i (\mathbf{x}_i \cdot \mathbf{x}) - \alpha_0 \right) \quad (8.7)$$

## 8.4 Separability and Margins

The SVM algorithm described above is designed to separate the classes that are in fact completely separable. In other words, when the separating hyperplane is constructed between the two classes, the entirety of the one class lies on one side of the hyperplane, and the entirety of the other class lies on the other side of the hyperplane with 100% accuracy in separation. The margins defined in Eqs. 8.2 are called as hard margins that impose complete separability between the classes. However, in practice such cases are seldom found. There is always some degree of overlap between the participating classes. As a result, an ideal separating hyperplane cannot be constructed. In order to account for such cases, *soft SVMs* were introduced.

In order to construct the equations for SVM with soft margins, let us rewrite Eq. 8.3 in a slightly different manner as

$$0 \geq 1 - y_i[(\mathbf{w} \cdot \mathbf{x}) - w_0], i = 1, \dots, p \quad (8.8)$$

As discussed before, for all the cases when the samples are not separable, the above inequality will not be satisfied. To accommodate such cases, the optimization problem is reformulated using additional regularization techniques.

### 8.4.1 Use of Slack Variables

One commonly used regularization technique involves the use of slack variables denoted as  $\xi_i$ , where  $\xi_i \geq 0, i = 1, \dots, p$ . Equation 8.8 is updated with the use of slack variables as

$$\xi_i \geq 1 - y_i[(\mathbf{w} \cdot \mathbf{x}) - w_0], i = 1, \dots, p \quad (8.9)$$

The cost functional is also updated with slack variables as

$$\Phi(\mathbf{w}) = \|\mathbf{w}\|^2 + C \sum_{i=1}^p \xi_i \quad (8.10)$$

The additional parameter  $C$  needs to be added along with the slack variables such that  $C > 0$ . Another implication of this added variable is that it puts an upper bound on all the Lagrange multipliers as

$$0 \geq \alpha_i \geq C \quad (8.11)$$

The parameter  $C$  limits the influence of individual sample and reduces the impact of noisy data.

With this setup, the SVMs can now be used in more realistic cases of overlapping distributions of classes effectively. The optimization operation needs to compute the values of all the slack variables along with weights of support vectors. SVMs built using this approach are called as  $C$ -SVMs.

There is an alternate way to apply the regularization, called as  $\nu$ -SVM where the parameter  $C$  is replaced with two additional parameters  $\nu$  and  $\rho$ . The value of  $\nu$  is limited as  $0 \leq \nu \leq 1$ , and the value of  $\rho$  is limited as  $\rho \geq 0$ . The cost function is now updated as

$$\Phi(\mathbf{w}) = \|\mathbf{w}\|^2 - \nu\rho + \frac{1}{2} \sum_{i=1}^p \xi_i \quad (8.12)$$

This difference in the formulation changes the bounds on Lagrange variables as

$$\sum_{i=1}^{i=p} \alpha_i \geq \nu \quad (8.13)$$

Overall the differences between the two implementations are minor, and it can be shown that the two implementations are in fact equivalent with certain constraints on the participating variables. However, the  $\nu$ -SVM is the preferred implementation in most open-source machine learning libraries.

## 8.5 Implementing Linear SVMs

We revisit the example of Iris data classification to illustrate the use of SVM as classifier. The following code shows how to import the SVM model in sklearn library using Google Colab platform and use it to train and predict as classifier. We are using all the default parameters, except the choice of linear kernel (default option

```
▶ from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

```
[ ] from sklearn import svm
svmc = svm.SVC(kernel="linear")
svmc.fit(X,y)
y_pred = svmc.predict(X)
svmc.score(X,y)
```

0.9933333333333333

```
[ ] import matplotlib.pyplot as plt
fig = plt.figure()
ax1 = fig.add_subplot(111)
plt.title('Plot of labels and predictions using Decision Tree Classifier')
ax1.plot(y, c='b', marker=".", label='Labels')
ax1.plot(y_pred, c='r', marker="o", label='Predictions')
plt.legend(loc='upper left');
fig.set_size_inches(10,5)
plt.grid(True)
plt.show()
```

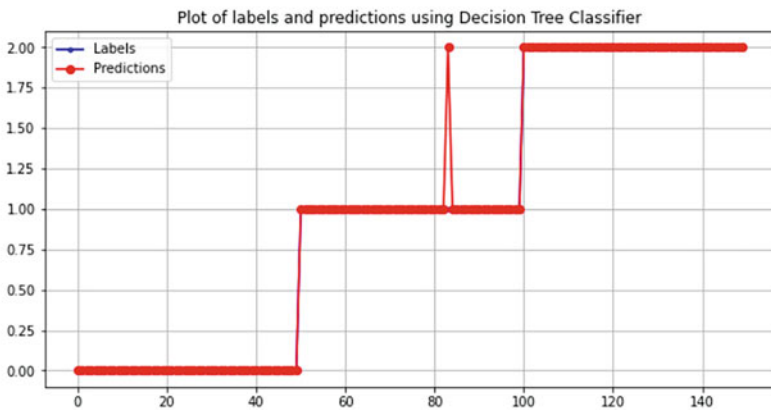


Fig. 8.3 Implementing linear SVM classifier using Google Colab and sklearn

is RBF, which is a nonlinear kernel), in the initialization, to not complicate things to begin with. Right off the bat, there is accuracy of 0.99 showing the superiority of SVM as classifier compared to other algorithms (Fig. 8.3).

## 8.6 Nonlinearity and Use of Kernels

Use of kernels is one of the groundbreaking discoveries in the field of machine learning. With the help of this method, one can elegantly transform a nonlinear problem into a linear problem. These kernel functions are different from the *link functions* that we discussed in Chap. 5. In order to understand the use of kernels in case of support vector machines, let's look at Eq. 8.7, specifically the term  $(\mathbf{x} \cdot \mathbf{x})$ . Here, we are taking a dot product of input vector with itself and as a result generating a real number. Use of kernel function<sup>2</sup> states that we can replace the dot product operation with a function that accepts two parameters (in this case both will be input vector) and outputs a real valued number. Mathematically, this kernel function is written as

$$k : (\mathcal{X} \cdot \mathcal{X}) \rightarrow \Re \quad (8.14)$$

Although this representation allows for using any arbitrary kernel function to transform the original data, in order to have deterministic answers in all the situations, the kernel function needs to be a *positive definite* function. A positive definite function needs to satisfy a property defined by Mercer's theorem. Mercer's theorem states that for all finite sequence of points  $x_1, x_2, \dots, x_n$  and all real numbers  $c_1, c_2, \dots, c_n$ , the kernel function should satisfy

$$\sum_{i=1}^n \sum_{j=1}^n k(x_i, x_j) c_i c_j \geq 0 \quad (8.15)$$

With the appropriate choice of such positive definite kernel function, one can map the n-dimensional input to a real valued output in a deterministic linear manner. If we know certain nonlinearity trends in the data, we can build a custom kernel function that will transform the problem suitable to be solved by the linear support vector machine. Some of the commonly used kernel functions are as follows:

### 8.6.1 Radial Basis Function

Radial basis function kernel with variance  $\sigma$  is given as

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|^2}{2\sigma^2}\right) \quad (8.16)$$

---

<sup>2</sup> Sometimes this is also called as kernel trick, although this is far more than a simple trick. A function needs to satisfy certain properties in order to be able called as kernel function. For more details on kernel functions, refer to [55].

This representation of SVM resembles closely with radial basis function neural networks that we learnt in Chap. 6. In some cases, use of squared distance between the two inputs can lead to vanishingly low values. In such cases a variation of the above function, called as Laplacian radial basis function, is used. It is defined as

$$k(x_i, x_j) = \exp\left(-\frac{\|x_i - x_j\|}{\sigma}\right) \quad (8.17)$$

### 8.6.2 Polynomial

For polynomial with degree  $d$ , the kernel function is given as

$$k(x_i, x_j) = (x_i \cdot x_j + 1)^d \quad (8.18)$$

### 8.6.3 Sigmoid

Sigmoid kernel can also be used which resembles a traditional neural network. It is defined as

$$k(x_i, x_j) = \tanh(Ax_i'x_j + B) \quad (8.19)$$

## 8.7 Implementing Nonlinear SVMs with Kernels

In order to illustrate the use of nonlinear kernels, let's use the same example as before. See the code for this in Fig. 8.4; we only change the definition of SVM classifier with choice of nonlinear kernel as RBF. As we can see, the accuracy actually drops compared to the linear version. It is important to note that nonlinear kernels are not always the best choice. When the data is actually linearly separable, linear kernels work best. However, when the data is distributed in nonlinear fashion, the nonlinear kernels can perform better. As the SVM algorithm is primarily developed with linear distribution in mind, it works quite well in linear mode. The nonlinear kernels need to be carefully chosen to get better performance in case of nonlinear data.



Fig. 8.4 Implementing nonlinear SVM classifier using Google Colab and sklearn

## 8.8 Risk Minimization

Methods based on risk minimization, sometimes called as structural risk minimization [83], essentially aim at learning to optimize the given system with constraints on parameters imposed by regularization as well as problem definition itself. Support vector machines solve this problem of risk minimization in elegant fashion. These methods strike the balance between performance optimization and reduction in overfitting in a programmatic manner. Vapnik further extended the theory of structural risk minimization for the cases of generative models using vicinal risk minimization [82]. These methods can be applied to the cases that do not fit in the traditional SVM architecture, such as problems with missing data or unlabelled data.



## 8.9 Conclusion

In this chapter we studied an important pillar of machine learning theory, the support vector machines. SVM represents a mathematically elegant architecture to build an optimal classification or regression model. The training process is a bit complex and needs tuning of some hyperparameters, but once properly tuned, SVM models tend to provide a very high accuracy and generalization capabilities.

## 8.10 Exercises

1. Use the Iris data set for multi-class classification using SVM. Use the two different approaches outlined in the chapter to perform the classification, and compare the outcomes from each using classification accuracy, training time, memory footprint, etc.
2. Repeat the above experiment with linear kernels and nonlinear kernels. You can choose your choice of nonlinear function. Observe and compare the difference in performance.
3. Compare the results of the different SVMs with the results from the previous chapter using decision trees. What do you conclude?
4. Here is a list of additional real work datasets from sklearn [26]. Try to use them with linear and nonlinear SVMs and see the difference.

## Appendix

New Lagrangian to be minimized is given as

$$\min_{\mathbf{w}} \left\{ \left( \frac{1}{n} \sum_{i=1}^n \max(0, 1 - y_i[(\mathbf{w} \cdot \mathbf{x}) - w_0]) \right) + \lambda \Phi(\mathbf{w}) \right\} \quad (8.20)$$

Here, with the max function, we are essentially ignoring the cases when there is error in the classification.

# Chapter 9

## Probabilistic Models



### 9.1 Introduction

Most algorithms studied thus far are based on algebraic, graphical, and/or calculus-based methods. In this chapter we are going to focus on the probabilistic methods. Probabilistic methods try to assign some form of uncertainty to the unknown variables and some form of belief probability to known variables and try to find the unknown values using the extensive library of probabilistic models. The probabilistic models are mainly classified into two types:

1. Generative
2. Discriminative

Generative models take a more holistic approach towards understanding the data compared to discriminative models. Commonly, the difference between the two types is given in terms of the probabilities that they deal with. If we have an observable input  $X$  and observable output  $Y$ , then the generative models try to model the joint probability  $P(X; Y)$ , while the discriminative models try to model the conditional probability  $P(Y|X)$ . Most non-probabilistic approaches discussed so far conceptually belong to the class of discriminative models as well, though they don't explicitly assign probabilities. It is important to understand the difference between the approaches to begin with, but this distinction between the two approaches can be vague and confusing. Hence, we will try to define the two approaches on their own merit.

It is important to define one more concept before we actually define the two approaches. Let there be a hidden entity called *state* denoted as  $S$  along with the input  $X$  and output  $Y$ . The input actually makes some changes into the state of the system, and that change along with the input dictates the output. Discriminative models try to predict the changes in the output based on only changes in the input and do not explicitly try to understand changes in the state of the system. Generative models try to model the changes in the output based on changes in input as well as

the changes in the state. This inclusion and modeling of the state gives a deeper insight into the systemic aspects, and the generative models are typically harder to build and need more information and assumptions to start with. However, there are some inherent advantages that come with this added complexity as we will see in later sections of this chapter.

The probabilistic approaches (discriminative as well as generative) are also sliced based on two universities of thought groups:

1. Maximum likelihood estimation
2. Bayesian approach

## 9.2 Discriminative Models

We will first discuss the distinction between these two classes from the perspective of discriminative models, and then we will turn to generative models.

### 9.2.1 Maximum Likelihood Estimation

The maximum likelihood estimation or *MLE* approach deals with the problems at the face value and parameterizes the information into variables. The values of the variables that maximize the probability of the observed variables lead to the solution of the problem. Let us define the problem using formal notations. Let there be a function  $f(\mathbf{x}; \theta)$  that produces the observed output  $y$ .  $x \in \mathfrak{R}^n$  represents the input on which we don't have any control over, and  $\theta \in \Theta$  represents a parameter vector that can be single dimensional or multidimensional. The MLE method defines a likelihood function denoted as  $L(y|\theta)$ . Typically the likelihood function is the joint probability of the parameters and observed variables as  $L(y|\theta) = P(y; \theta)$ . The objective is to find the optimal values for  $\theta$  that maximizes the likelihood function as given by

$$\theta^{\text{MLE}} = \arg \max_{\theta \in \Theta} \{L(y|\theta)\} \quad (9.1)$$

or

$$\theta^{\text{MLE}} = \arg \max_{\theta \in \Theta} \{P(y; \theta)\} \quad (9.2)$$

This is a purely frequentist approach that is strictly data dependent.

### 9.2.2 Bayesian Approach

Bayesian approach looks at the problem in a different manner. All the unknowns are modelled as random variables with known prior probability distributions. Let us denote the conditional prior probability of observing the output  $y$  for parameter vector  $\theta$  as  $P(y|\theta)$ . The marginal probabilities of these variables are denoted as  $P(y)$  and  $P(\theta)$ . The joint probability of the variables can be written in terms conditional and marginal probabilities as

$$P(y; \theta) = P(y|\theta).P(\theta) \quad (9.3)$$

The same joint probability can also be given as

$$P(y; \theta) = P(\theta|y).P(y) \quad (9.4)$$

Here, the probability  $P(\theta|y)$  is called as posterior probability. Combining Eqs. 9.3 and 9.4,

$$P(\theta|y).P(y) = P(y|\theta).P(\theta) \quad (9.5)$$

Rearranging the terms we get

$$P(\theta|y) = \frac{P(y|\theta).P(\theta)}{P(y)} \quad (9.6)$$

Equation 9.6 is called as Bayes' theorem. This theorem gives relationship between the posterior probability (sometimes called as *a posteriori*) and prior probability (sometimes called as *a priori*) in a simple and elegant manner. As Bayes' method tries to optimize the a posteriori probability, it is also sometimes referred to as Maximum A Posteriori estimation or MAP estimation.

This equation is the foundation of the entire Bayesian framework of analysis. Each term in the above equation is given a name:  $P(\theta)$  is called as *prior*,  $P(y|\theta)$  is called as *likelihood*,  $P(y)$  is called as *evidence*, and  $P(\theta|y)$  is called as *posterior*. Thus, using these terms, the Bayes' theorem can be stated as

$$(\text{posterior}) = \frac{(\text{prior}).(\text{likelihood})}{(\text{evidence})} \quad (9.7)$$

The Bayes' estimate is based on maximizing the posterior. Hence, the optimization problem based on Bayes' theorem can now be stated as

$$\theta^{\text{Bayes}} = \arg \max_{\theta \in \Theta} \{P(\theta|y)\} \quad (9.8)$$

expanding the term

$$\theta^{\text{Bayes}} = \arg \max_{\theta \in \Theta} \left\{ \frac{P(y|\theta) \cdot P(\theta)}{P(y)} \right\} \quad (9.9)$$

Comparing this equation with 9.2, we can see that Bayesian approach adds more information in the mix in the form of prior probability. Sometimes, this information is available, and then Bayesian approach clearly provides an edge over MLP, while in cases where this information is not explicitly available, one can still assume certain default distribution and proceed.

### 9.2.3 Comparison of MLE and Bayesian Approach

These formulations are relatively abstract and in general can be quite hard to comprehend. In order to understand them such that we can apply the principles in real life, let us consider a simple numerical example. Let there be an experiment to toss a coin for *five* times. Let's say the two possible outcomes of each toss are *H, T*, or *Head* or *Tail*. The outcome of our experiment is *H, H, T, H, H*. The objective is to find the outcome of the sixth toss. Let's work out this problem using MLE and Bayes' approach.

#### 9.2.3.1 Solution Using MLE

The likelihood function is defined as,  $L(y|\theta) = P(y; \theta)$ , where  $y$  denotes the outcome of the trial and  $\theta$  denotes the property of the coin in the form of probability of getting given outcome. Let the probability of getting a *Head* be  $h$  and the probability of getting a *Tail* be  $1 - h$ . Now, the outcome of each toss is independent of the outcome of the other tosses. Hence, the total likelihood of the experiment can be given as

$$P(y; \theta) = P(y = H|\theta = h)^4 \cdot P(y = T|\theta = (1 - h)) \quad (9.10)$$

Now, let us solve this equation:

$$P(y; \theta) = h.h.(1 - h).h.h$$

$$P(y; \theta) = h^4 - h^5$$

In order to maximize the likelihood, we need to use the fundamental principle from differential calculus, that is, at any maximum or minimum of a continuous function, the first order derivative is 0. In order to maximize the likelihood, we will differentiate the above equation with respect to  $h$  and equate it to 0. Solving the equation (assuming  $h \neq 0$ ), we get

$$\begin{aligned}
\frac{\partial}{\partial h} P(y; \theta) &= 0 \\
\frac{\partial}{\partial h} (h^4 - h^5) &= 0 \\
4.h^3 - 5.h^4 &= 0 \\
4.h^3 &= 5.h^4 \\
4 &= 5.h \\
h &= \frac{4}{5}
\end{aligned}$$

This probability of getting *Head* in the next toss would be  $\frac{4}{5}$ .

### 9.2.3.2 Solution using Bayes' Approach

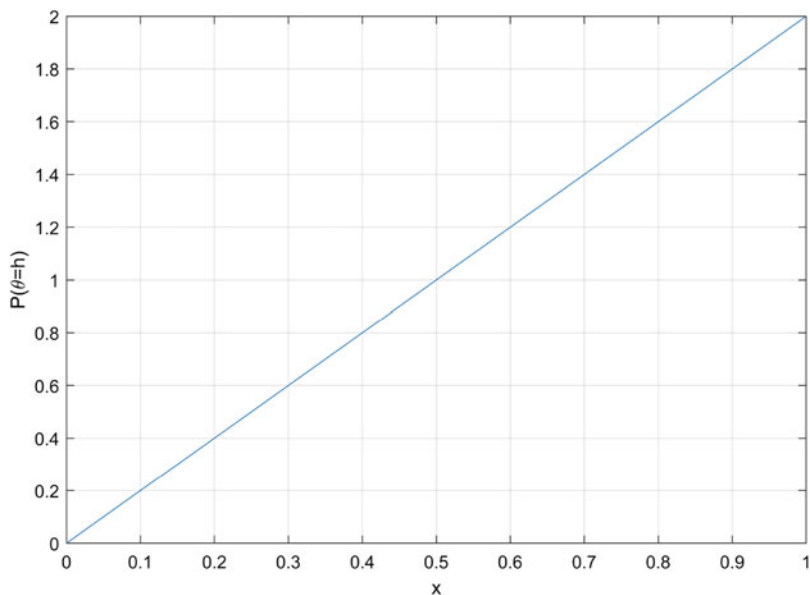
Writing the posterior as per Bayes' theorem,

$$P(\theta|y) = \frac{P(y|\theta).P(\theta)}{P(y)} \quad (9.11)$$

Comparing this equation with Eq. 9.10, we can see that the likelihood function is the same as the term  $P(y|\theta)$  in the current equation. However, we need value for another entity  $P(\theta)$ , and that is the *prior*. This is something that we are going to assume as it is not explicitly given to us. If we assume the prior probability to be uniform, then it is independent of  $\theta$ , and the outcome of Bayes' approach will be the same as the outcome of MLE. However, in order to showcase the differences between the two approaches, let us use a different and nonintuitive prior. Let  $P(\theta = h) = 2h$ . Consequently  $P(\theta = T)$ . While defining this prior, we need to make sure that it is a valid probability density function. The easiest way to make sure that is to confirm  $\int_{h=0}^{h=1} P(\theta = h) = 1$ . As can be seen from Fig. 9.1, it is indeed true. There is one more factor in the equation in the form of *evidence*,  $P(y)$ . However, this value is the probability of occurrence of the output without any dependency on the con bias and is constant with respect to  $h$ . When we differentiate with respect to  $h$ , the effect of this parameter is going to vanish. Hence, we can safely ignore this term for the purpose of optimization.

So we can now proceed with the optimization problem as before. In order to maximize the posterior, let's differentiate it with respect to  $h$  as before:

$$\frac{\partial}{\partial h} P(\theta|y) = \frac{\partial}{\partial h} \frac{P(y|\theta).P(\theta)}{P(y)} = 0 \quad (9.12)$$



**Fig. 9.1** Probability density function (pdf) for the prior

Substituting the values and solving (assuming  $h \neq 0$ ),

$$\begin{aligned} \frac{\partial}{\partial h} \frac{P(y|\theta) \cdot P(\theta)}{P(y)} &= 0 \\ \frac{\partial}{\partial h} ((2 \cdot h)^5 \cdot P(y = H|\theta = h)^4 \cdot P(y = T|\theta = (1 - h))) &= 0 \\ \frac{\partial}{\partial h} (2^5 \cdot h^5 \cdot (h^4 - h^5)) &= 0 \\ \frac{\partial}{\partial h} (2^5 \cdot (h^9 - h^{10})) &= 0 \\ 9 \cdot h^8 - 10 \cdot h^9 &= 0 \\ 9 \cdot h^8 &= 10 \cdot h^9 \\ h &= \frac{9}{10} \end{aligned}$$

With Bayes' approach, the probability of getting *Head* in the next toss would be  $\frac{9}{10}$ . Thus, assumption of a nontrivial prior with Bayes' approach leads to a different answer compared to MLE.

### 9.3 Implementing Probabilistic Models

When Bayes' approach is used in multidimensional problems, for simplicity it is assumed that the different dimensions are independent of each other. This approach is commonly called as Naive Bayes approach. We will try to use this approach with the Iris dataset as we have used in illustrating most other methods as shown in Figs. 9.2 and 9.3. With Naive Bayes classifier, one can choose different types of distributions. We will use the most commonly used choice of Gaussian distribution. During the training, the algorithm is trying to identify the parameters of Gaussian distribution for each class. I would like to re-emphasize here that accuracy numbers shown here are not accuracy numbers in the true sense as we are applying the prediction on the same data used for training. The objective here is to observe how good the model can learn the patterns in the input data. In the later chapter

```
from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn import naive_bayes
nbc = naive_bayes.GaussianNB()
nbc.fit(X,y)
y_pred = nbc.predict(X)
nbc.score(X,y)
```

Fig. 9.2 Implementing Naive Bayes classifier using Google Colab and sklearn



Fig. 9.3 Visualizing the training performance with Naive Bayes classifier on Iris data



on building machine learning pipelines, we will look at real-life application of the models and accuracy calculations by splitting the data into training-test-validation.

## 9.4 Generative Models

As discussed earlier, generative models try to understand how the data that is being analyzed came to be in the first place. They find applications in multiple different fields, such as speech synthesis and generating images or 3D environments that resemble the real life but are not directly copied from any of the real examples. The generative models can be broadly classified into two types: (1) classical models and (2) deep learning-based models. We will look at a few examples of classical generative models briefly.

### 9.4.1 Mixture Models

One of the fundamental aspects of generative models is to understand the composition of the input to understand how the input data came into existence in the first place. The most simplistic case would be to have all the input data as outcome of a single process. If we can identify the parameters describing the process, we can understand the input to its fullest extent. However, typically any input data is far from such ideal case, and it needs to be modelled as an outcome of multiple processes. This gives rise to the concept of mixture models.

When modeling given data as a mixture of multiple processes, one has to assume certain parametric distributions for the models and then use the labelled samples to optimally predict the parameters of the underlying distributions. Expectation maximization or EM algorithm is typically used along with many others to predict the parameters [61].

### 9.4.2 Bayesian Networks

Bayesian networks represent directed acyclic graphs as shown in Fig. 9.4. Each node represents an observable variable or a state. The edges represent the conditional dependencies between the nodes. Training of Bayesian network involves identifying the nodes and predicting the conditional probabilities that best represent the given data.

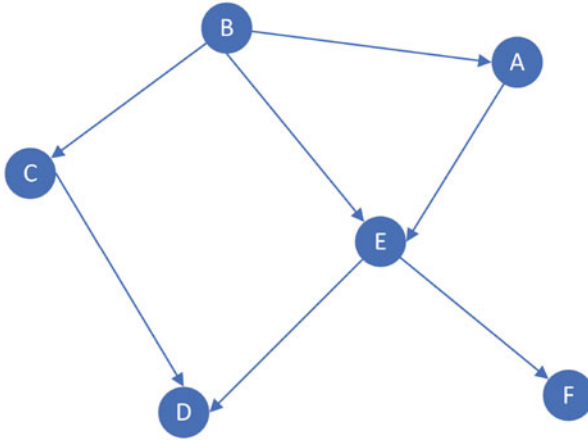


Fig. 9.4 Sample Bayesian network

## 9.5 Some Useful Probability Distributions

We will conclude this chapter with detailing some of the commonly used probability distributions. There are likely hundreds of different distributions studied in the literature of probability and statistics. We don't want to study them all, but in my experience with machine learning projects so far, I have realized that knowledge of few key distributions goes a long way. Hence, I am going to describe these distributions here without going into theoretical details of their origins, etc. We will look at the probability density functions or *pdf*'s and the cumulative density functions or *cdf*'s of these distributions and take a look at the parameters that define these distributions. Here are the definitions of these quantities for reference:

**Definition 9.1** pdf A probability density function or pdf is a function  $P(X = x)$  that provides probability of occurrence of value  $x$  for a given variable  $X$ . The plot of  $P(X = x)$  is bounded between  $[0, 1]$  on  $y$ -axis and can spread between  $[-\infty, \infty]$  on  $x$ -axis and integrates to 1.

**Definition 9.2** cdf A cumulative density function or cdf is a function  $C(X = x)$  that provides the sum of probabilities of occurrences of values of  $X$  between  $[-\infty, x]$ . This plot is also bounded between  $[0, 1]$ . Unlike pdf, this plot starts at 0 on the left and ends into 1 at the right.

I would strongly advise the reader to go through these distributions and see the trends in the probabilities as the parameters are varied. We come across distributions like these in many situations, and if we can match a given distribution to a known distribution, the problem can be solved in far more elegant manner.

### 9.5.1 Normal or Gaussian Distribution

Normal distribution is one of the most widely used probability distributions. It is also called as bell-shaped distribution due to the shape of its pdf. The distribution has a vast array of applications including error analysis. It also approximates a multitude of other distributions with more complex formulations. Another reason the normal distribution is popular is due to central limit theorem.

**Definition 9.3** Central Limit Theorem Central limit theorem states that if a sufficiently large number of samples are taken from a population from any distribution with finite variance, then the mean of the samples asymptotically approaches the mean of the population. In other words, sampling distribution of mean taken from the population of any distribution asymptotically approaches normal distribution.

Hence, sometimes the normal distribution is also called as distribution of distributions.

Normal distribution is also an example of continuous and unbounded distribution, where the value of  $x$  can span  $[-\infty, \infty]$ . Mathematically, the pdf of normal distribution is given as

$$P_{\text{normal}}(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] \quad (9.13)$$

where  $\mu$  is the mean and  $\sigma$  is the standard deviation of the distribution. Variance is  $\sigma^2$ . cdf of normal distribution is given as

$$C_{\text{normal}}(x|\mu, \sigma) = \frac{1}{2} \left[ 1 + \operatorname{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right) \right] \quad (9.14)$$

where erf is a standard error function, defined as

$$\operatorname{erf}(x) = \frac{1}{\sqrt{\pi}} \int_{-x}^x e^{-t^2} \quad (9.15)$$

The functional that is being integrated is symmetric; hence, it can also be written as

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \quad (9.16)$$

The plots of the pdf and cdf are shown in the figure below (Figs. 9.5 and 9.6).

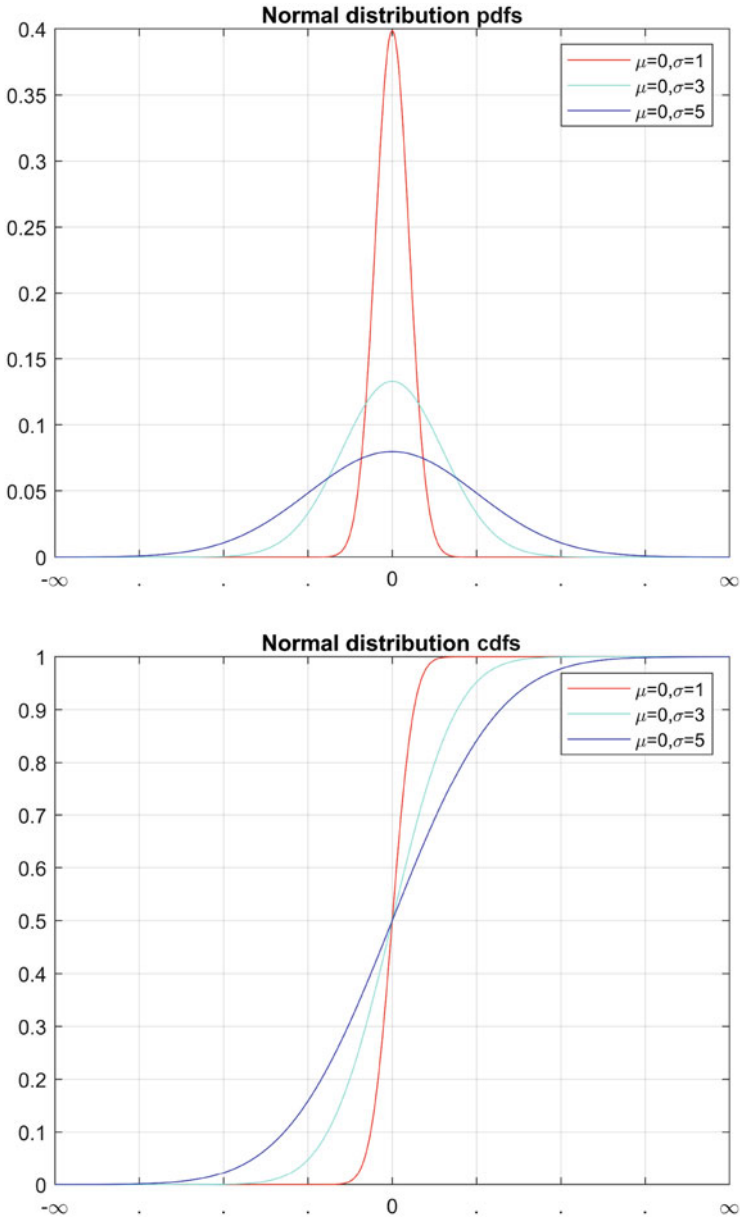


Fig. 9.5 Plot of normal pdfs for 0 mean and different variances

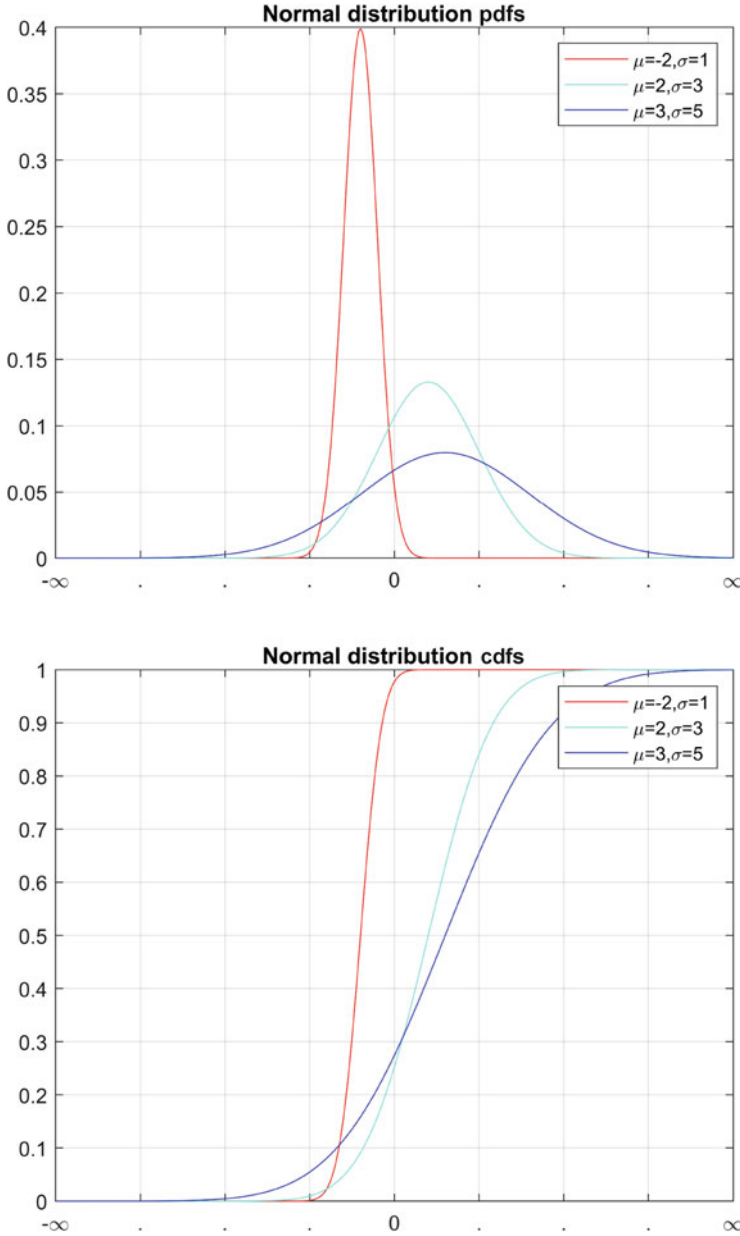


Fig. 9.6 Plot of normal pdfs for different means and different variances

### 9.5.2 Bernoulli Distribution

Bernoulli distribution is an example of discrete distribution, and its most common application is the probability of coin toss. The distribution owes its name to a great

mathematician of the seventeenth century, *Jacob Bernoulli*. The distribution is based on two parameters  $p$  and  $q$ , which are related as  $p = 1 - q$ . Typically  $p$  is called the probability of success (or in case of coin toss, it can be called as probability of getting a *Head*), and  $q$  is called the probability of failure (or in case of coin toss, probability of getting a *Tail*). Based on these parameters, the pdf (sometimes, in case of discrete variables, it is called as *probability mass function* or *pmf*, but for the sake of consistency, we will call it pdf) of Bernoulli distribution is given as

$$P_{\text{bernoulli}}(k|p, q) = \begin{cases} p, & \text{if } k = 1, \\ q = 1 - p, & \text{if } k = 0. \end{cases} \quad (9.17)$$

Here, we use the discrete variable  $k$  instead of the continuous variable  $x$ . The cdf is given as

$$C_{\text{bernoulli}}(k|p, q) = \begin{cases} 0, & \text{if } k < 0, \\ q = 1 - p, & \text{if } 0 \leq k < 1, \\ 1, & \text{if } k \geq 1. \end{cases} \quad (9.18)$$

### 9.5.3 Binomial Distribution

Binomial distribution generalizes Bernoulli distribution for multiple trials. Binomial distribution has two parameters  $n$  and  $p$ .  $n$  is the number of trials of the experiment, where the probability of success is  $p$ . The probability of failure is  $q = 1 - p$  just like Bernoulli distribution, but it is not considered as a separate third parameter. The pdf for binomial distribution is given as

$$P_{\text{Binomial}}(k|n, p) = \binom{n}{k} p^k (1 - p)^{n-k} \quad (9.19)$$

where

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \quad (9.20)$$

is called as binomial coefficient in this context. It also represents the number of combinations of  $k$  in  $n$  from the permutation-combination theory where it is represented as

$${}^n C_k = \binom{n}{k} \quad (9.21)$$

The cdf of binomial distribution is given as

$$C_{\text{Binomial}}(k|n, p) = \sum_{i=0}^k \binom{n}{i} p^i (1-p)^{n-i} \quad (9.22)$$

### 9.5.4 Gamma Distribution

Gamma distribution is also one of the very highly studied distributions in the theory of statistics. It forms a basic distribution for other commonly used distributions like chi-squared distribution, exponential distribution, etc., which are special cases of gamma distribution. It is defined in terms of two parameters:  $\alpha$  and  $\beta$ . The pdf of gamma distribution is given as

$$P_{\text{gamma}}(x|\alpha, \beta) = \frac{\beta^\alpha x^{\alpha-1} e^{-\beta x}}{\Gamma(\alpha)} \quad (9.23)$$

where  $x > 0$  and  $\alpha, \beta > 0$ . The simple definition of  $\Gamma(\alpha)$  for integer parameter is given as a factorial function as

$$\Gamma(n) = (n-1)! \quad (9.24)$$

The same definition is generalized for complex numbers with positive real parts as

$$\Gamma(\alpha) = \int_0^\infty x^{\alpha-1} e^{-x} dx \quad (9.25)$$

This function also follows the recursive property of factorials as

$$\Gamma(\alpha) = (\alpha-1)\Gamma(\alpha-1) \quad (9.26)$$

Figure 9.7 shows plots of the pdf for variable  $\alpha$  and  $\beta$  values.

The cdf of gamma function cannot be stated easily as a single valued function, but rather is given as sum of an infinite series as

$$C_{\text{gamma}}(x|\alpha, \beta) = e^{-\beta x} \sum_{i=\alpha}^{\infty} \frac{(\beta x)^i}{i!} \quad (9.27)$$

Figure 9.8 shows plots of the cdf for variable  $\alpha$  and  $\beta$  values similar to the ones shown for pdf.

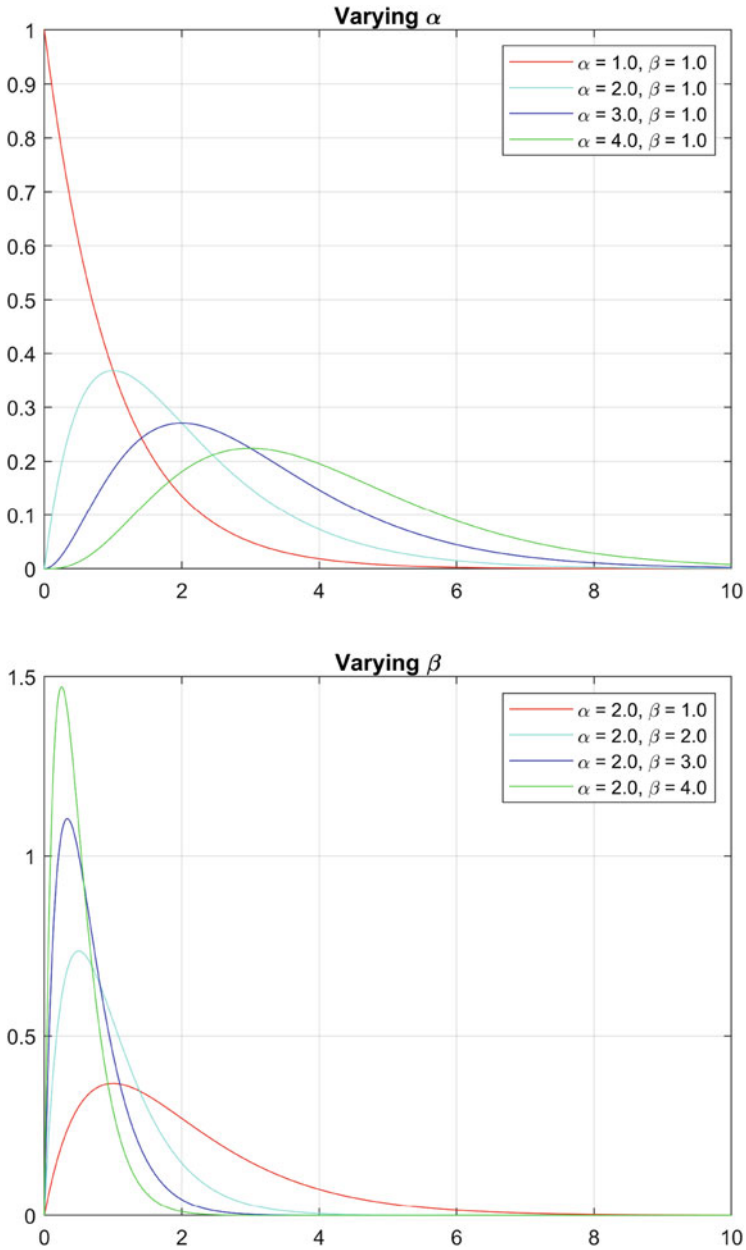


Fig. 9.7 Plot of Gamma pdfs for different values of  $\alpha$  and  $\beta$



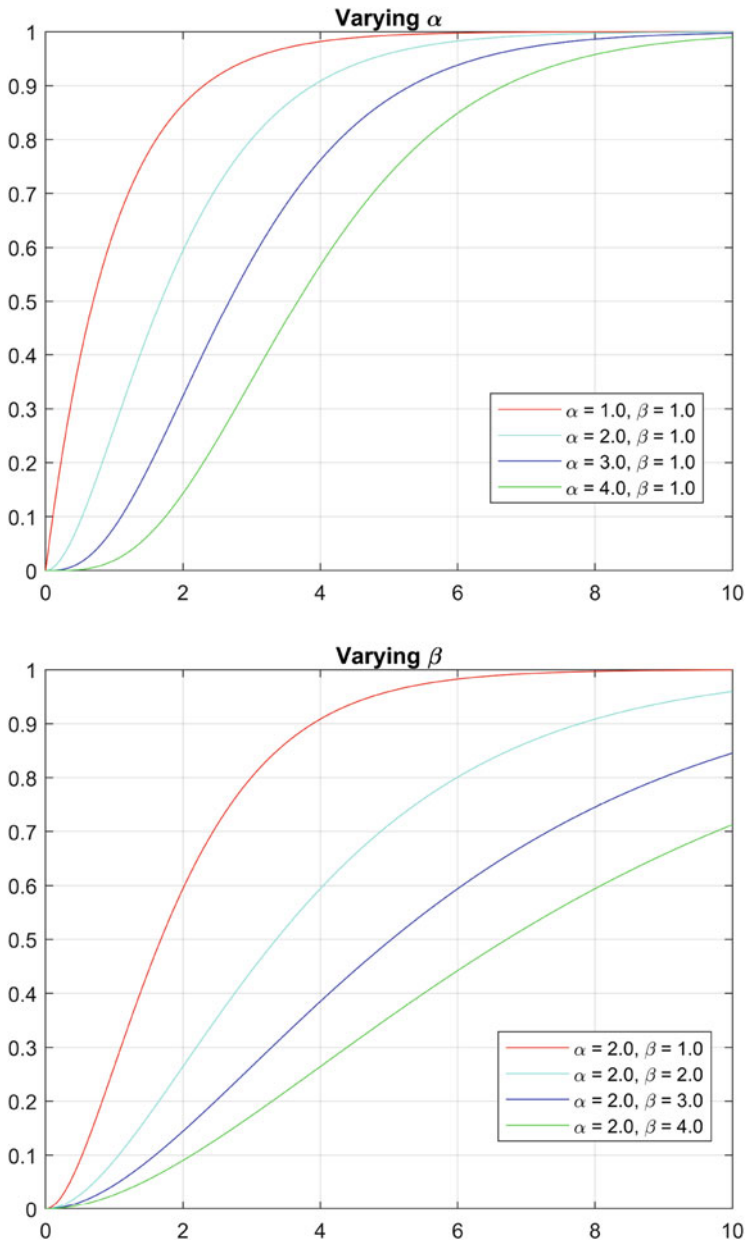


Fig. 9.8 Plot of Gamma cdfs for different values of  $\alpha$  and  $\beta$

### 9.5.5 Poisson Distribution

Poisson distribution is a discrete distribution loosely similar to binomial distribution. Poisson distribution is developed to model the number of occurrences of an outcome

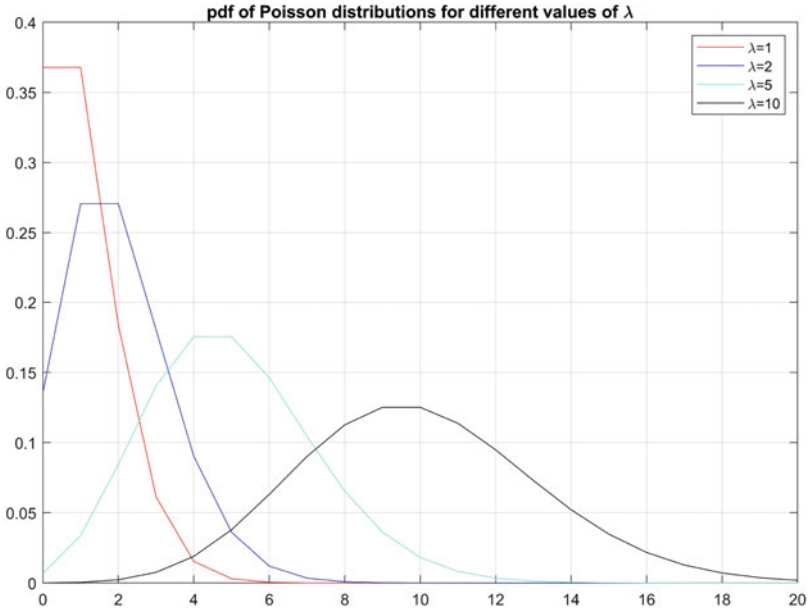


Fig. 9.9 Plot of Poisson pdfs for different values of  $\lambda$

in fixed interval of time. It is named after French mathematician *Siméon Poisson*. The pdf of Poisson distribution is given in terms of the number of events ( $k$ ) in the interval as

$$P_{\text{Poisson}}(k) = e^{-\lambda} \frac{\lambda^k}{k!} \tag{9.28}$$

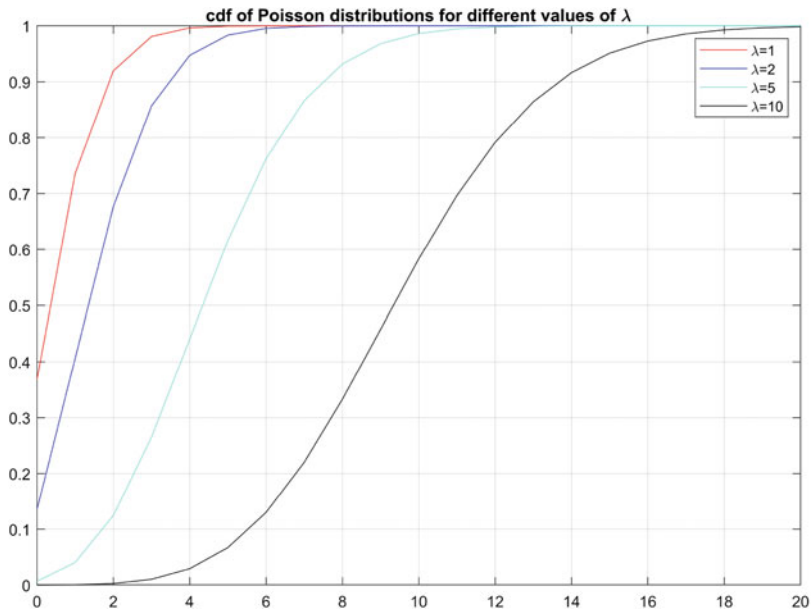
where the single parameter  $\lambda$  is the average number of events in the interval. The cdf of Poisson distribution is given as

$$C_{\text{Poisson}}(k) = e^{-\lambda} \sum_{i=0}^k \frac{\lambda^i}{i!} \tag{9.29}$$

Figures 9.9 and 9.10 show the pdf and cdf of Poisson distribution for various values of the parameter  $\lambda$ .

## 9.6 Conclusion

In this chapter, we studied various methods based on probabilistic approach. These methods start with some different fundamental assumptions compared to other methods, specifically the ones based on Bayesian theory. The knowledge of



**Fig. 9.10** Plot of Poisson cdfs for different values of  $\lambda$

priory knowledge separates them from all other methods. If available, this priory knowledge can improve the model performance significantly as we saw in fully worked example. Then, we concluded the chapter with learning a bunch of different probability distributions with their density and cumulative functions.

## 9.7 Exercises

1. Use different distributions, and repeat the experiment. Try to understand the results based on the visualization of the data and distributions. See if you can make some connection with why certain distributions work better in this case.

# Chapter 10

## Dynamic Programming and Reinforcement Learning



### 10.1 Introduction

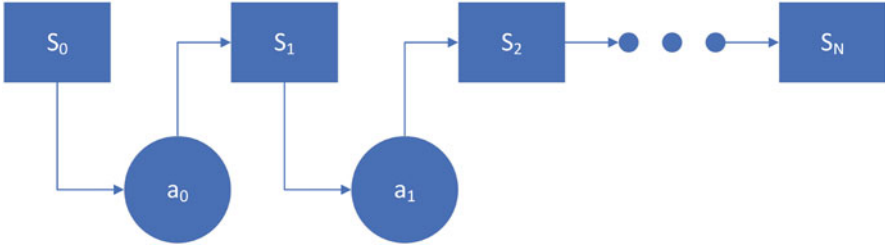
The theory of dynamic programming is typically attributed to Richard Bellman [56], who published the famed mathematical optimization method in the 1950s. In the preface of his iconic book, he defines dynamic programming as follows:

The purpose of this book is to provide introduction to the mathematical theory of multi-stage decision process. Since these constitute a somewhat complex set of terms we have coined the term *dynamic programming* to describe the subject matter.

This is very interesting and apt naming as the set of methods that come under the umbrella of dynamic programming is quite vast. These methods are deeply rooted in pure mathematics but are more favorably expressed so that they can be directly implemented as computer programs. In general such multistage decision problems appear in all sorts of industrial applications, and tackling them is always a daunting task. However, Bellman describes a structured and iterative manner in which the problem can be broken down and solved systematically. There exists a notion of state machine in the sequential solution of the subproblems, and the context of subsequent problems changes dynamically based on the solution of the previous problems. This non-static behavior of the methods imparts the name *dynamic* to the algorithm. These types of solutions also marked the early stages of *artificial intelligence* in the form of *expert systems*.

### 10.2 Fundamental Equation of Dynamic Programming

In general, the problem that dynamic programming tries to solve can be stated in the form of a single equation, and it is called as *Bellman equation*. Let us consider a process that goes through N steps as shown in Fig. 10.1. At each step there exist



**Fig. 10.1** The setup for Bellman equation

a state and a possible set of actions. Let the initial state be  $s_0$  and the first action taken be  $a_0$ . We also constrain the set of possible actions in step  $t$  as  $a_t \in \Gamma(s_t)$ . Depending on the action taken, the next state is reached. Let us call the function that combines the current state and action and produces the next state as  $T(s, a)$ . Hence,  $s_1 = T(s_0, a_0)$ . As the process is going through multiple states, let the problem we are trying to solve be to optimize a value function at step  $t$  as  $V(s_t)$ .

The optimality principle in iterative manner can be stated as: “In order to have the optimum value in the last step, one needs to have optimum value in the previous step that will lead to final optimum value.” To translate this into an equation, we can write

$$V(s_t) = \max_{a_t \in \Gamma(s_t)} V(T(s_t, a_t)) \quad (10.1)$$

This is a special case of Bellman equation when the payout after each state is not considered. To make the equation more generic, let us add the payout function at step  $t$  as  $F(s_t, a_t)$ . A more general form of Bellman equation can now be given as

$$V(s_t) = \max_{a_t \in \Gamma(s_t)} (F(s_t, a_t) + V(T(s_t, a_t))) \quad (10.2)$$

In some scenarios, the optimality of the future value cannot be assumed to be fully attainable, and a discount factor needs to be added as  $\beta$ , where  $0 < \beta < 1$ . Then, the Bellman equation can be written as

$$V(s_t) = \max_{a_t \in \Gamma(s_t)} (F(s_t, a_t) + \beta V(T(s_t, a_t))) \quad (10.3)$$

This generic equation is in fairly abstract form, where we have not defined any specific problem or specific constraints or even the specific value function that we want to maximize or minimize. However, once we have these defined, we can safely use this equation to solve the problem as long as the functions are continuous and differentiable.

### 10.3 Classes of Problems Under Dynamic Programming

Dynamic programming defines a generic class of problems that share the same assumptions as theory of machine learning. The exhaustive list of problems that can be solved using the theory of dynamic programming is quite large as can be seen here [4]. However, the most notable classes of problems that are studied and applied are as follows:

- Travelling salesman problem
- Quadratic optimization using recursive least squares (RLS) method
- Finding the shortest distance between two nodes in graph
- Viterbi algorithm for solving hidden Markov model (HMM)

Other than solving these specific problems, the area that is most relevant in the context of modern machine learning is *reinforcement learning* and its derivatives. We will study these concepts in the rest of the chapter.

### 10.4 Reinforcement Learning

Most of the machine learning techniques we have explored so far and will explore in later chapters primarily focus on two types of learning: (1) supervised and (2) unsupervised. These methods are classified based on the availability or unavailability of labelled data. However, both these methods can be called as *offline*, and they don't really focus on live interaction with the environment. It is assumed that the input and output data (if available) is already collected and is ready for consumption by the algorithms. Reinforcement learning takes a fundamentally different approach towards learning. It can be called as *online* method of learning and follows biological aspects of learning more closely. When a newborn baby starts interacting with the environment, his/her learning begins. In the initial times, the baby is making mostly random actions and is being greeted by the environment in some way. Over the course of time, the baby starts to learn based on all the actions generated and feedback received. This is the essence of reinforcement learning. It cannot be classified into either of the two types, although it is supervised in some sense. Let us look at some of the fundamental characteristics of the reinforcement learning to understand precisely how it differs from these methods. The reinforcement learning framework is based on interaction between two primary entities: (1) system and (2) environment.

#### 10.4.1 Characteristics of Reinforcement Learning

1. There is no preset labelled training data available at the start.
2. An action space is predefined that typically can contain a very large number of possible actions that the system can generate at any given instance.

3. The system chooses to make an action at every instance of time. The meaning of instance is different for each application.
4. At every instance of time, a feedback (also called as reward) from the environment is recorded. It can either be positive, negative, or neutral.
5. There can be delay in the feedback.
6. System learns while interacting with the environment.
7. The environment is not static, and every action made by the system can potentially change the environment itself.
8. Due to the dynamic nature of the environment, the total training space is practically infinite.
9. The training phase and application phase are not separate in case of reinforcement learning. The model is continuously learning as it is also reacting or predicting.

### ***10.4.2 Framework and Algorithm***

It is important to note that reinforcement learning is a framework and not an algorithm like most other methods discussed in this book; hence, it can only be compared with other learning frameworks. As stated earlier, reinforcement learning does involve supervision in some sense; Figs. 10.2 and 10.3 illustrate the comparison between the architectures of reinforcement learning and supervised learning frameworks. Unsupervised learning is completely different as it does not involve any type of feedback or labels and cannot be compared with reinforcement learning.

## **10.5 Exploration and Exploitation**

Reinforcement learning introduces two new concepts in the process of learning called exploration and exploitation. When the system starts its learning process, there is no prior knowledge learned thus far, and every action taken by the system is pure random. This process is called as exploration. During exploration, the system is just trying out different possible actions that it can make or exploring its capabilities and limits thereof and simultaneously registering the feedback from the system, which can be positive, negative, or neutral. After a while, in learning phase, when sufficient feedback is gathered, the system starts to associate the actions and corresponding feedback. Using these associations from the previous explorations, it can start producing actions that are not random but more deterministic in the sense that they are more likely to generate positive reward. This process is called as exploitation. Reinforcement learning needs to find a good tradeoff between exploration and exploitation. Exploration opens up more possible actions that can lead to better long-term rewards in the future at the cost of lower possible rewards in

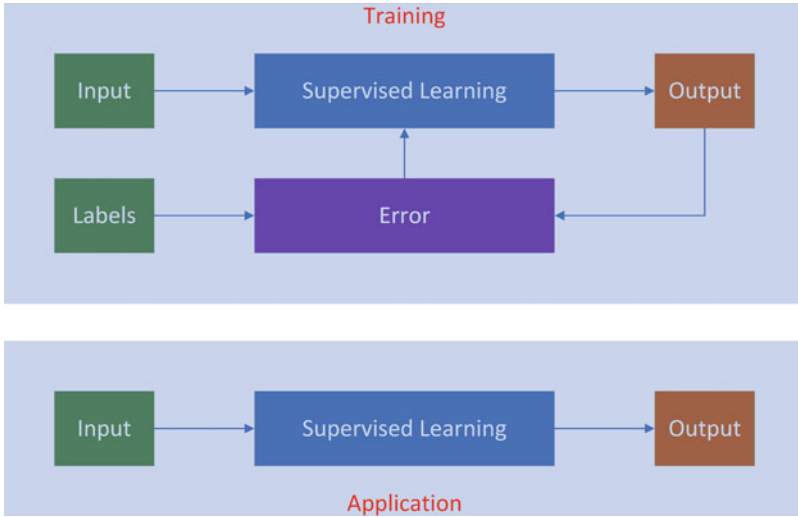


Fig. 10.2 Architecture of supervised learning

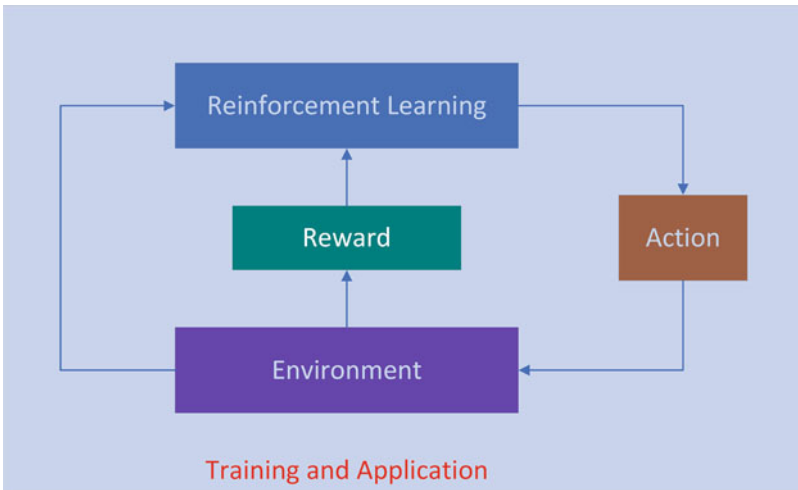


Fig. 10.3 Architecture of reinforcement learning

the short term, while exploitation tends to get better short-term rewards at the cost of possibly missing out on greater long-term rewards. Mathematically speaking, a system heavily biased towards exploitation is more likely to get stuck in local optimum, while a system heavily biased towards exploration can never reach any optimum. However, a carefully balanced system is likely to find the global optimum over time.



## 10.6 Applications of Reinforcement Learning

The theory of reinforcement learning is better understood after looking at some of the real-life applications as follows:

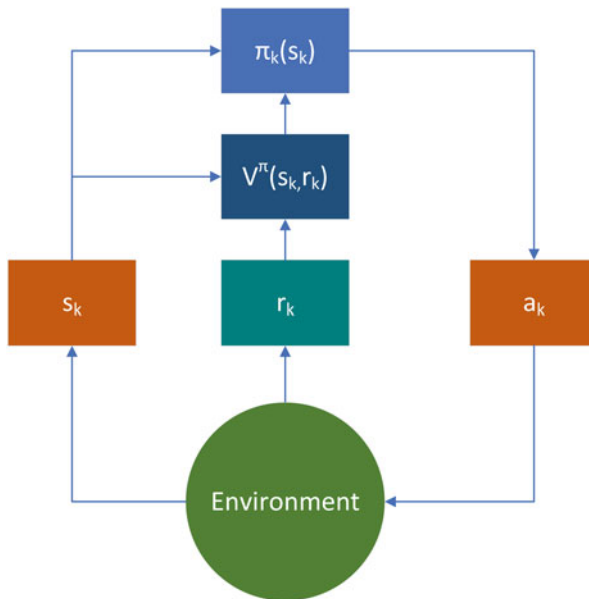
1. **Chess programs:** Solving the problem of winning a chess game by computers is one of the classic applications of reinforcement learning. Every move that is made by either side opens up a new position on the board. The ultimate objective is to capture the king of the opposite side, but the short-term goals can be to capture pieces of other side or gain control of the center, etc. The action space is practically infinite, as there are 32 pieces in total on 64 squares and each piece has different types of moves allowed. One of the conservative estimates on the number of possible moves in chess is calculated to be around  $10^{120}$ , which is also known as *Shannon number* [10]. When one talks about computers playing chess, IBM's Deep Blue supercomputer comes out as a top example. Although it was a computer system built to play chess, its primary objective was to specifically defeat the reigning world champion Garry Kasparov at the time (1997) [11]. Sometimes, this victory gets confused as a landmark in progress of machine learning. Although it did use some bits of reinforcement learning, it was heavily augmented with huge databases of past games played by Kasparov and other grand masters, and it resembled more to an expert system rather than a learning system. Since then the computers have become increasingly better at the game with the true application of reinforcement learning. The real victory of reinforcement learning-based chess program came with Google's AlphaZero system. This system was trained by playing against itself without using any of the historical game databases. The only input given to the system was the rules of chess and definition of a win. It was able to learn all the complex concepts in chess just by playing with itself. Just after about 9 h of training, it was able to defeat the other world champion chess program (which had already reached levels beyond the best of humans), called *Stockfish*, in 2015 [80]. Some of the moves displayed by AlphaZero in those games were nothing short of stunning and completely unorthodox compared to classical chess playing style. For more details on these games, one can refer to [27].
2. **Robotics:** Training a robot to maneuver in a complex real world is another classical reinforcement learning problem that closely resembles the biological learning. In this case, the action space is defined by the combination of scope of moving parts of the robot, and the environment is the area in which the robot needs to maneuver along with all the existing objects in the area. If we want to train the robot to lift one object from one place and drop it at another, then the rewards would be given accordingly.
3. **Video games:** Solving video games is another interesting application of reinforcement learning problems. A video game creates a simulated environment in which the user needs to navigate and achieve certain goals in the form of say winning a race, killing a monster, etc. Only certain combination of moves allows the user to pass through various challenging levels. The action space is also well

defined in the form of up, down, left, right, accelerate, brake, or attack with certain weapon, etc. OpenAI has created a platform for testing reinforcement learning models to solve video games in the form of *Gym* [13]. Here is one application where *Gym* is used to solve stages in the classical game *Super Mario* [12].

4. **Personalization:** Various e-commerce websites like Amazon and Netflix have most of their content personalized for each user. This can be achieved with the use of reinforcement learning as well. The action space here is the possible recommendations, and the reward is user engagement as a result of a certain recommendation.

## 10.7 Theory of Reinforcement Learning

Figure 10.4 shows the signal flow and value updates using reinforcement learning architecture.  $s_k$  denotes the state of the system that is a combination of the environment and the learning system itself at time instance  $k$ .  $a_k$  is the action taken by the system, and  $r_k$  is the reward given by the environment at the same time instance.  $\pi_k$  is the policy for determining the action at the same time instance and is function of current state.  $V^\pi$  denotes the value function that updates the policy using current state and reward.



**Fig. 10.4** Reinforcement learning model architecture

### 10.7.1 Variations in Learning

This depiction of reinforcement learning combines various different methods into a single generic representation. Here are the different methods typically used:

1. Q-learning
2. SARSA
3. Monte Carlo

#### 10.7.1.1 Q-Learning

In order to understand Q-learning, let's consider the most generic form of Bellman equation as Eq. 10.3. In Q-learning framework, the function  $T(s, a)$  is called as the value function. The technique of Q-learning focusses on learning the values of  $T(s, a)$  for all the given states and action combinations. The algorithm of Q-learning can be summarized as follows:

1. Initialize the Q-table, for all the possible state and action combinations.
2. Initialize the value of  $\beta$ .
3. Choose an action using a tradeoff between exploration and exploitation.
4. Perform the action and measure the reward.
5. Update the corresponding Q-value using Eq. 10.3.
6. Update the state to next state.
7. Continue the iterations (steps 3–6) till the target is reached.

#### 10.7.1.2 SARSA

SARSA stands for *state-action-reward-state-action* [86]. SARSA algorithm is an incremental update to Q-learning where it adds learning based on policy. Hence, it is also sometimes called as *on-policy Q-learning*. The traditional Q-learning is off-policy. The update equation for SARSA can be given as

$$V'(s_t, a_t) = (1 - \alpha\beta)V(s_t, a_t) + \alpha F(s_t, a_t) + \alpha\beta V(s_{t+1}, a_{t+1}) \quad (10.4)$$

where  $\beta$  is discount factor as before and  $\alpha$  is called as learning rate.

## 10.8 Implementing Reinforcement Learning

Unfortunately as of the time of writing this book, sklearn library does not support reinforcement learning algorithms. Reinforcement learning is typically studied with deep learning these days, and as a result many deep learning libraries like

TensorFlow (Google) and Keras support reinforcement learning as well. Also, this being an active area of research, new libraries are coming up each day. The setup of the environments with these libraries on Google Colab is not quite simple. One can always implement reinforcement learning from the first principles of exploration and exploitation and a tradeoff between them. It is important to have an optimized memorization data structure to accompany this learning to store all the prior actions and rewards. We will skip the implementation of reinforcement learning for the scope of this book. Interested reader can refer to Keras implementation of reinforcement learning [53].

## 10.9 Conclusion

In this chapter, we studied methods belonging to the class of dynamic programming as defined by Bellman. The specific case of reinforcement learning and its variations mark a topic of their own, and we devoted a section for studying these concepts and their applications. Reinforcement learning marks a whole new type of learning that resembles human learning more so than traditional supervised and unsupervised learning techniques. Reinforcement learning enables fully automated way of learning in a given environment. These techniques are becoming quite popular in the context of deep learning, and we will study those aspects in later chapters.

## 10.10 Exercises

1. To implement a reinforcement learning solution, one also needs a simulation of the environment. OpenAI Gym [54] provides a platform for such simulations. Choose a problem of your choice, and try to solve it using Keras-RL and Gym.
2. One can also convert a traditional supervised learning problem into a reinforcement learning problem but making the labelled training samples available to the learning algorithm in sequential manner. Cast the problem of Iris classification as reinforcement learning problem try to apply the Q-learning technique to solve it. Compare these results with the results obtained with other algorithms.

# Chapter 11

## Evolutionary Algorithms



### 11.1 Introduction

All the traditional algorithms including the new deep learning framework tackle the problem of optimization using calculus of gradients. The methods have evolved significantly to solve harder problems that were once considered impossible to solve. However, the horizon of the reach of these algorithms is linear and predictable. Evolutionary algorithms try to attack the optimization problems in a fundamentally different manner of massive exploration in a random but supervised manner. This approach opens up whole new types of solutions for the problems at hand. Also, these methods are inherently suitable for embarrassingly parallel computation, which is the *mantra* of modern computation based on GPUs.

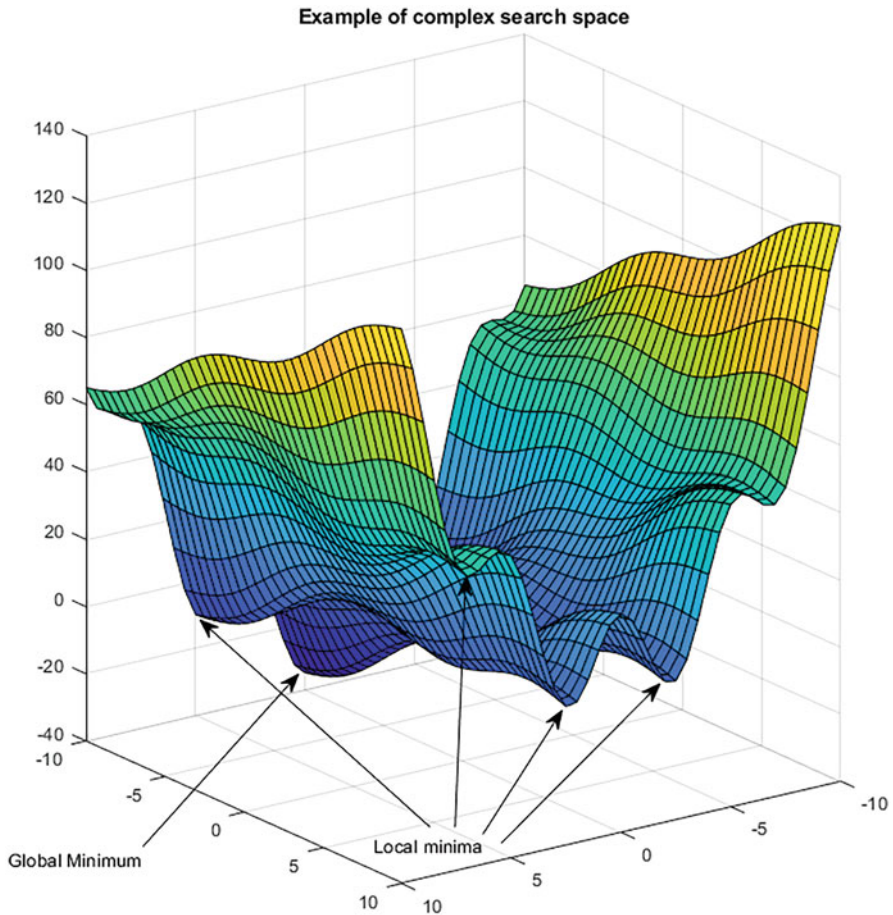
### 11.2 Bottleneck with Traditional Methods

In the applications of machine learning, one comes across many problems for which it is practically impossible to find universally optimal solution. In such cases one has to be satisfied with a solution that is optimal within some reasonable local neighborhood (the neighborhood is from the perspective of the hyperspace spanned by the feature values). Figure 11.1 shows an example of such space.

Most traditional methods employ some form of linear search in a greedy manner.<sup>1</sup> In order to see a greedy method in action, let us zoom into the previous

---

<sup>1</sup> In general all the algorithms that use gradient-based search are called as greedy algorithms. These algorithms use the fact from calculus that at any local optimum (minimum or maximum), the value of gradient is 0. In order to distinguish between whether the optimum is a minimum or a maximum, the second-order gradient is used. When the second-order gradient is positive, a minimum is reached; otherwise, it is a maximum.

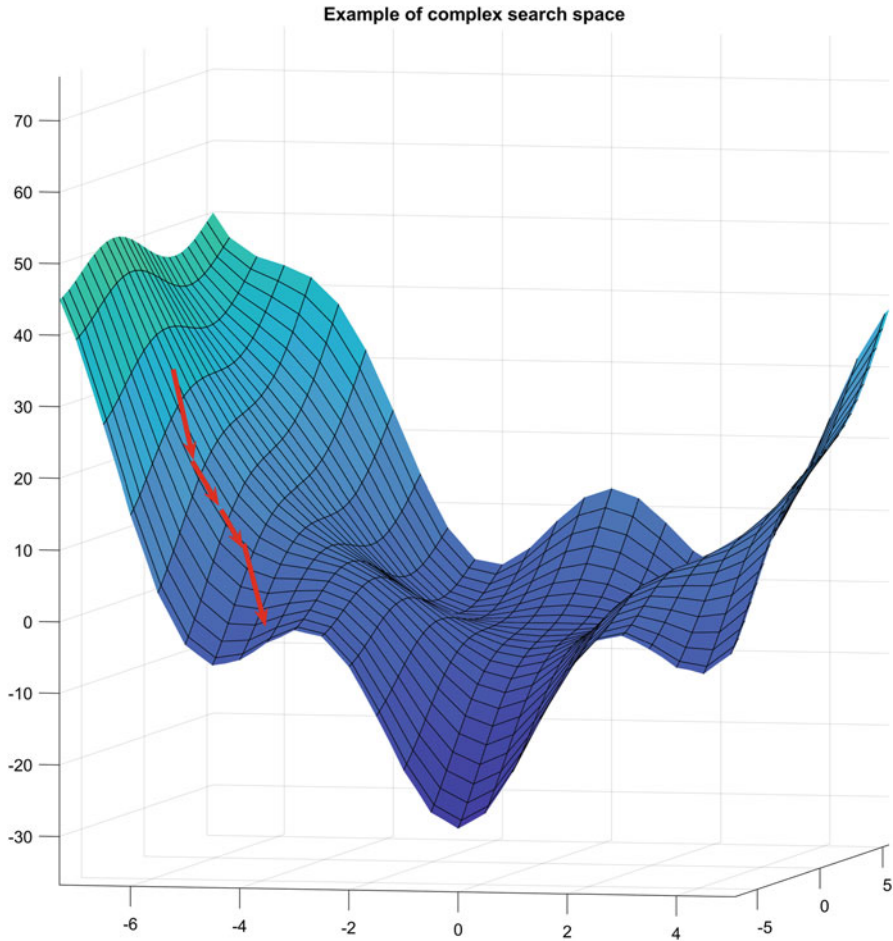


**Fig. 11.1** Example of complex search space with multiple local minima and a unique single global minimum

figure, as shown in Fig. 11.2. The red arrows show how the greedy algorithm progresses based on the gradient search and results into a local minimum.

### 11.3 Darwin's Theory of Evolution

There exists a very close parallel to this problem in the theory of natural evolution. In any given environment, there exists a complex set of constraints in which all the inhabitant animals and plants are fighting for survival and evolving in the process. The setup is quite dynamic, and the perfectly ideal species does not exist for any given environment at all times. All the species have some advantages and some



**Fig. 11.2** Example of greedy search in action resulting into a local minimum

limitations at any given time. The evolution of the species is governed by Darwin's theory of evolution by natural selection. The theory can be stated briefly as:

Over sufficiently long span of time, only those individual organisms survive in a given environment who are better suited for the environment.

This time span can extend over multiple generations, and its effects are typically not seen in a matter of few years or even a few hundred years. However, in a few thousand years and more, the effect of evolution by natural selection has been seen and verified beyond doubt. There is one more aspect to the theory of evolution without which it cannot work, and that is random variation in the species that typically happens by the process of mutation. If the process of reproduction kept on producing species that are identical to the parents, there would never be any

change in the setup, and natural selection cannot successfully happen. However, when we add a random variation in the offsprings during each reproduction, it changes everything. The new features that are created as a result of mutation are put to test in the environment. If the new features make the organism better cope with the environment, the organisms with those features thrive and tend to reproduce more in comparison to the organisms that do not have those features. Thus over time, the offsprings of the weaker organisms are extinct, and the species overall has evolved. As a result of continued evolution over time, the species get better and better with respect to surviving in the given environment. In the long run, the process of evolution leads the species along the direction of better adaptation, and on aggregate level it never retrogresses. These characteristics of evolution are rather well suited for the problems described at the beginning.

All the algorithms that come under the umbrella of evolutionary algorithms are inspired from this concept. Each such algorithm tries to interpret the concepts of random variation, environmental constraints, and natural selection in its own way to create a resulting evolution. Fortunately, as the processes of random variations and natural selection are implemented using computers running at GHz speed, they can happen in a matter of fraction of seconds compared to thousands or hundreds of thousands of years required in the biological setup.

Thus, evolutionary methods rely on creating an initial population of samples that is chosen randomly, instead of a single starting point used in greedy methods. Then, they let processes of mutation-based sample variation and natural selection to do their job to find which sample evolves into better estimate. Although evolutionary algorithms also do not guarantee a global optimum, they typically have a higher chance of finding one.

The following sections describe a few most common examples of evolutionary algorithms.

## 11.4 Genetic Programming

Genetic programming models try to implement Darwin's idea as closely as possible. It maps the concepts of genetic structure to solution spaces and implements the concepts of natural selection and reproduction with possibility of mutation in programmatic manner. Let us look at the steps in the algorithm.

### Steps in Genetic Programming

1. Set the process parameters as stopping criteria, mutation fraction, etc.
2. Initialize the population of solution candidates using random selection.
3. Create a fitness index based on the problem at hand.

(continued)



4. Apply the fitness index to all the population candidates and trip the number of candidates to a predetermined value by eliminating the lowest-scoring candidates.
5. Randomly select pairs of candidates from the population as parents, and carry out the process of reproduction. The process of reproduction can contain two alternatives:
  - a. **Crossover**: In crossover, the parent candidates are combined in a predefined structural manner to create the offspring.
  - b. **Mutation**: In mutation, the children created by the process of crossover are modified randomly. The mutation is applied only for a fraction of the offsprings as determined as one of the settings of the process.
6. Augment the original population with the newly created offsprings.
7. Repeat Steps 4, 5, and 6 till the desired stopping criteria are met.

Although the steps listed in the algorithm are fairly straightforward from a biological standpoint, they need customization based on the problem at hand for programmatic implementation. In order to illustrate the complexity in customization, let us take a real problem. A classic problem that is very hard to solve using traditional methods but is quite suitable for genetic programming is of travelling salesman. The problem is like this:

There is a salesman that wants to travel to  $n$  number of destinations in sequence. The distance between each pair of destinations is given as  $d_{ij}$ , where  $i, j \in \{1, 2, \dots, n\}$ . The problem is to select the sequence that connects all the destinations only once in shortest overall distance.

Although apparently a straightforward-looking one, this problem is actually considered as one of the hardest to solve,<sup>2</sup> and a universally optimal solution to this problem is not possible even when the number of destinations is as small as say 100.

Let us try to solve this problem using the above steps.

1. Let us define the stopping criteria as successive improvement in the distance to be less than some value  $\alpha$  or the maximum number of iterations.
2. We first need to create a random population of solutions containing say  $k$  number of distinct solutions. Each solution is a random sequence of destination from 1 to  $n$ .
3. The fitness test would be given as the sum of distances between successive destinations.

---

<sup>2</sup> This problem belongs to a class of problems called as NP-hard. It stands for nondeterministic polynomial time hard problems [15]. The worst-case solution time for this problem increases in near-exponential time and quickly becomes beyond the scope of current hardware.

4. We will keep  $k$  number of top candidates when sorted in a decreasing order of total distance.
5. Reproduction step is where things get a little tricky. First, we will choose two parents randomly. Now, let us consider the two cases one by one.
  - a. For crossover, we will select first  $k_1, k_1 < k$  destinations directly from parent-1 and then the remaining destinations from parent-2. However, this simple crossover can lead to duplicating some destinations and missing some destinations in the new sequence, called as the offspring sequence. These errors need to be fixed by appropriate adjustment.
  - b. For mutations, once the crossover-based offspring sequence is generated, randomly some destinations are swapped.
6. Once we have reproduced the full population, we will have a population of twice the size. Then we can repeat the above steps as described in the algorithm till stopping criteria are reached.

Unfortunately due to the random factor in the design of genetic programs, one cannot have a deterministic bound on how much time it would take to reach the acceptable solution, how much should be the size of the population, how much should be the percentage of mutations, etc. One has to experiment with multiple values of these parameters to find the optimal solution for each given case. In spite of this uncertainty, genetic programs are known to provide significant improvements in computation times for solutions of certain types of problems and are in general a strong tool to have in the repertoire of machine learning toolkit.

### 11.4.1 *Implementing Genetic Programming*

sklearn library does not support genetic programming as well. However, there is another library called *gplearn* that is compatible with sklearn and implements genetic programming. You can find more information about the library here: [28]. We will install the library in Google Colab environment and then use it to solve the Iris classification problem. However, currently the library does not support multi-class classification, so we will only use the first one hundred samples to classify two of the three classes. The following code shows installation of the library and its application (Figs. 11.3 and 11.4).

## 11.5 **Swarm Intelligence**

Swarm intelligence is a generic term used to denote algorithms that are influenced by the biological aspects of groups of primitive organisms. The origin of swarm intelligence techniques can be traced back to 1987, when Craig Reynolds published

```
[ ] pip install gplearn

Collecting gplearn
  Downloading gplearn-0.4.1-py3-none-any.whl (41 kB)
    [REDACTED] | 41 kB 267 kB/s
Requirement already satisfied: joblib>=0.13.0 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: scikit-learn>=0.20.0 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: numpy>=1.14.6 in /usr/local/lib/python3.7/dist-package
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.7/dist-packages
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.7/dist-
Installing collected packages: gplearn
Successfully installed gplearn-0.4.1

from sklearn import datasets
iris = datasets.load_iris()
X = iris.data
y = iris.target

[ ] from gplearn import genetic
gc = genetic.SymbolicClassifier()
gc.fit(X[1:100],y[1:100])

SymbolicClassifier()

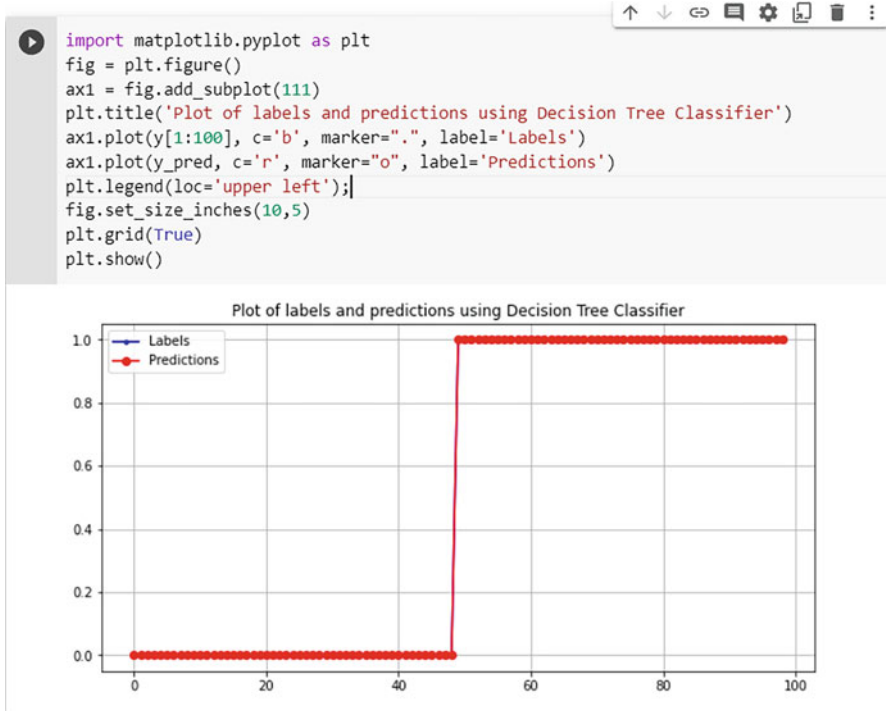
[ ] y_pred = gc.predict(X[1:100])
gc.score(X[1:100],y[1:100])

1.0
```

**Fig. 11.3** Implementing genetic programming-based binary classifier Google Colab and gplearn

his work on *boids* [88]. In his work Reynolds designed a system of flock of birds and assigned a set of rules governing the behavior of each of the bird in the flock. When we aggregate the behavior of the group over time, some completely startling and nontrivial trends emerge. This behavior can be attributed to the saying that sometimes,  $1 + 1 > 2$ . When a single bird is considered as a singular entity and is let loose in the same environment, it has no chance of survival. If all the birds in the flock act as single entities, they all are likely to perish. However, when one aggregates the birds to form a social group that communicates with each other without any specific governing body, the abilities of the group improve significantly. Some of the very long migrations of birds are classic examples of success of swarm intelligence.

In recent years, swarm intelligence is finding applications in computer graphics for simulating groups of animals or even humans in movies and video games. As a matter of fact, the encyclopedia of the twenty-first century, *Wikipedia*, can also be attributed to swarm intelligence. The techniques of swarm intelligence are also finding applications in controlling groups of autonomously flying drones. In general the steps in designing an algorithm that is based on swarm intelligence can be outlined as follows:



**Fig. 11.4** Visualizing the training performance with genetic programming-based classifier on two-class Iris data

1. Initialize the system by introducing a suitable environment by defining constraints.
2. Initialize an individual organism by defining the rules of possible actions and possible ways of communicating with others.
3. Establish the number of organisms and period of evolution.
4. Define the individual goals for each organism and group goals for the whole flock as well as stopping criteria.
5. Define the randomness factor that will affect the decisions made by individual organisms by trading between exploration and exploitation.
6. Start and repeat the process till the finishing criteria are met.

## 11.6 Ant Colony Optimization

Although ant colony optimization can be considered as a subset of swarm intelligence, there are some unique aspects to this method, and it needs separate consideration. Ant colony optimization algorithms, as the name suggests, are based

on the behavior of a large groups of ants in a colony. The individual ant possesses a very limited set of skills, e.g., they have a very limited vision; in most cases they can be completely blind, they have a very small brain with very little intellect, and their auditory and olfactory senses are also not quite advanced. In spite of these limitations, the ant colonies as such are known to have some extraordinary capabilities like building a complex nest and finding the shortest path towards food sources that can be at large distances from the nest. Another important aspect of the ant colonies is that they are a completely decentralized system. There is no central decision-maker, a king or queen ant that orders the group to follow certain actions. All the decisions and actions are decided and executed by individual ant based on its own method of functioning. For the ant colony optimization algorithm, we are specifically going to focus on the capabilities of the ants to find the shortest path from the food source to the nest.

At the heart of this technique lies the concept of *pheromones*. A pheromone is a chemical substance that is dropped by each ant as it passes along any route. These dropped pheromones are sensed by the ants that come to follow the same path. When an ant reaches at a junction, it chooses the path with the higher level of pheromones with a higher probability. This probabilistic behavior combines the random exploration with exploitation of the paths travelled by other ants. The path that connects the food source with the nest in least distance is likely going to be used more often than the other paths. This creates a form of positive feedback, and the path with the shortest distance keeps getting more and more chosen over time. In biological terms, the shortest path evolves over time. This is quite a different way of interpreting the process of evolution, but it conforms to the fundamentals nonetheless. All the different paths connecting the nest with the food source mark the initial population. The subsequent choices of different paths, similar to the process of reproduction, are governed in a probabilistic manner using pheromones. Then, the positive feedback created by aggregation of pheromones acts as a fitness test and controls the evolution in general.

These biological concepts related to emission of pheromones and their decay and aggregation can be modeled using mathematical functions to implement this algorithm programmatically. The problem of the travelling salesman is also a good candidate to use ant colony optimization algorithm. It is left as an exercise for the reader to experiment with this implementation. It should be noted that as the ant colony optimization algorithm has graphical nature of the solution at heart, it has relatively limited scope compared to genetic programs.

## 11.7 Simulated Annealing

Simulated annealing [87] is an odd man out in this group of evolutionary algorithms as it finds its origins in metallurgy and not biology. The process of annealing involves heating the metal above a certain temperature called as recrystallization temperature and then slowly cooling it down. When the metal is heated above the

recrystallization temperature, the atoms and molecules involved in the crystallization process can move. Typically this movement occurs such that the defects in the crystallization are repaired. After annealing process is complete, the metal typically improves its ductility and machinability as well as electrical conductivity.

Simulated annealing process is applied to solve the problem of finding the global minimum (or maximum) in a solution space that contains multiple local minima (or maxima). The idea can be described using Fig. 11.1. Let's say with some initial starting point, the gradient descent algorithm converges to the nearest local minimum. Then, the simulated annealing program generates a disturbance into the solution by essentially throwing the algorithm's current state to a random point in a predefined neighborhood. It is expected that the new starting point leads to another local minimum. If the new local minimum is smaller than the previous one, then it is accepted as a solution; otherwise, the previous solution is preserved. The algorithm is repeated again till the stopping criteria are reached. By adjusting the neighborhood radius corresponding to the higher temperature in the metallurgical annealing, the algorithm can be fine-tuned to get a better performance.

## 11.8 Conclusion

In this chapter, we studied the different algorithms inspired by the biological aspects of evolution and adaptation. In general the entire machine learning theory is inspired by the human intelligence, but the various algorithms used to achieve that goal may not directly be applicable to humans or even other organisms for that matter. However, the evolutionary algorithms are specifically designed to solve some very hard problems using methods that are used by different organisms individually or as a group.

## 11.9 Exercises

1. Use the technique of one class against the rest of the classes, and use the `gplearn` classifier to solve the full three-class problem of Iris data. Compare the accuracy with the accuracy obtained with other algorithms. Also note the time of execution for genetic programming classifier training.
2. Play with the different parameters of the model, and see its impact on the resulting accuracy and training time.
3. There is a library called `PySwarms` that implements what is called particle swarm optimization (PSO). Try to import this library into the Google Colab environment and implement classifier model with it [29].

# Chapter 12

## Time Series Models



### 12.1 Introduction

All the algorithms (except for reinforcement learning) discussed so far are based on static analysis of the data. By static it is meant that the data that is used for training purposes is constant and does not change over time. For example, Iris data: the measurements of the sepals and petals do not change over time. However, there are many situations where the data is not static, e.g., analysis of stock trends, weather patterns, analysis of audio or video signals, etc. The static models can still be used up to a certain extent to solve these problems, by converting the dynamic data into static by taking snapshots of the time series data at intermediate times. These snapshots can then be used as static data to train the models. However, this approach is seldom optimal and always results in less-than-ideal results.

Time series analysis has been studied quite extensively for over centuries as part of statistics and signal processing, and the theory has now matured. Typical applications of time series analysis involve trend analysis, forecasting, etc. In signal processing theory, the time series analysis also deals with frequency domain which leads to spectral analysis. These techniques are extremely powerful in handling dynamic data. We are going to look at this problem from the perspective of machine learning, and we will not delve too much into the signal processing aspects of the topic that essentially represent fixed mode analysis. The essence of machine learning is in feedback. When a certain computation is performed on the training data and a result is obtained, the error in the result must somehow be fed back into the computation to improve the performance. If this feedback is not present, then in most cases it does not classify as a machine learning application (unsupervised methods do contradict this statement, and in some sense they do not belong in machine learning). We will use this concept as yardstick to separate the pure signal processing or statistical algorithms from machine learning algorithms and only focus on the latter.

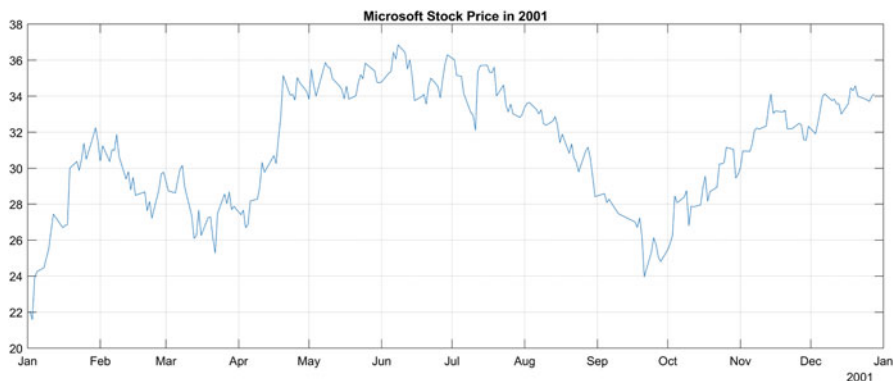


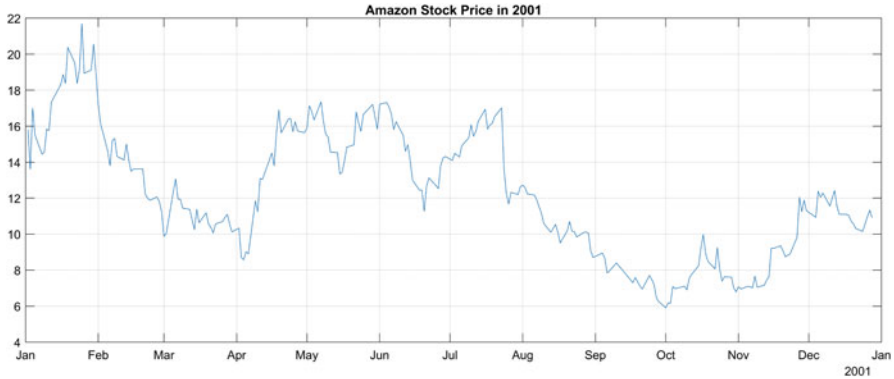
Fig. 12.1 Plot showing Microsoft stock price on a daily basis in calendar year 2001

## 12.2 Stationarity

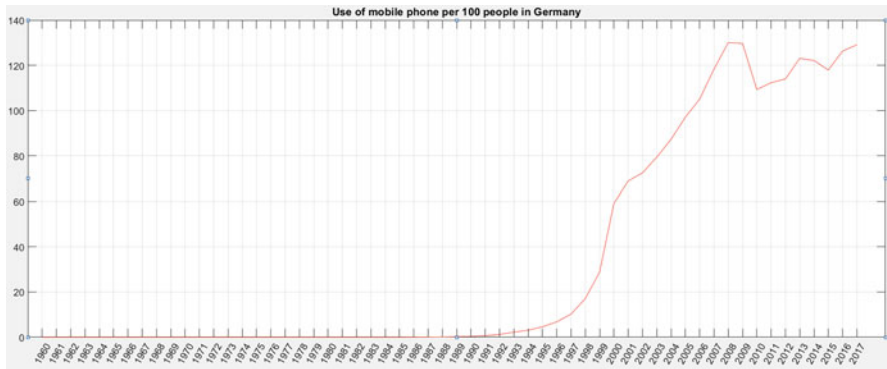
Stationarity is a core concept in the theory of time series, and it is important to understand some implications of this before going into modeling the processes. Stationarity or a stationary process is defined as a process for which the unconditional joint probability distribution of its parameters does not change over time. Sometimes this definition is also referred to as *strict stationarity*. A more practical definition based on normality assumption would be the mean and variance of the process remain constant over time. These conditions make the process strictly stationary only when the normality condition is satisfied. Sometimes a weaker form of stationarity is used where only mean is constant over time and second-order moment if finite. These types of processes are called as *weak stationary* or *wide sense stationary*. In general, when the process is non-stationary, the joint probability distribution of its parameters changes over time, or the mean and variance of its parameters are not constant. It becomes very hard to model such process. Although most processes encountered in real life are non-stationary, we always make the assumptions of stationarity or wide-sense stationary to simplify the modeling process. Then, we add concepts of trends and seasonality to address the effects of non-stationarity partially. Seasonality means the mean and variance of the process can change periodically with changing *seasons*. Trends essentially model the slow changes in mean and variance with time. We will see simple models that are built on the assumptions of stationarity, and then we will look at some of their extensions to take into consideration the seasonality.

To understand the nuances of trends and seasonality, let's look at the plots shown in Figs. 12.1 and 12.2. The plot of Microsoft stock price almost seems periodical with a period of roughly 6 months with upward trend, while Amazon stock plot shows irregular changes with overall downward trend. On top of these macro trends, there is additional periodicity on a daily basis.





**Fig. 12.2** Plot showing Amazon stock price on a daily basis in calendar year 2001



**Fig. 12.3** Plot mobile phone use in Germany per 100 people from 1960 to 2017. The data is courtesy of [7]

Figure 12.3 shows the use of mobile phones per 100 people in Germany from 1960 to 2017. This plot does not show any periodicity, but there is a clear upward trend in the values. The trend is not linear and not uniform. Thus, it represents a good example of non-stationary time series.

## 12.3 Autoregressive Moving Average Models

Autoregressive moving average or ARMA analysis is one of the simplest techniques of univariate time series analysis. As the name suggests, this technique is based on two separate concepts: *autoregression* and *moving average*. In order to define the two processes mathematically, let's start with defining a system. Let there be a discrete time system that takes white noise inputs denoted as  $\epsilon_i, i = 1, \dots, n$ ,

where  $i$  denotes the instance of time. Let the output of the system be denoted as  $x_i, i = 1, \dots, n$ . For ease of definition and without loss of generality, let's assume all these variables as univariate and numerical.

### 12.3.1 Autoregressive (AR) Process

An autoregressive or *AR* process is a process in which the current output of the system is a function of the weighted sum of a certain number of previous outputs. We can define an autoregressive process of order  $p$ ,  $AR(p)$  using the established notation as

$$x_i = \sum_{j=i-p}^{i-1} \alpha_j \cdot x_j + \epsilon_i \quad (12.1)$$

$\alpha_i$  are the coefficients or parameters of the AR process along with the error or residual term  $\epsilon_i$  at instance  $i$ . It is important to note that by design, AR process is not necessarily stationary. The AR process can be more efficiently represented with the help of time lag operator  $L$ . The operator is defined as

$$Lx_i = x_{i-1} \forall i \quad (12.2)$$

and for  $k$ th order lag

$$L^k x_i = x_{i-k} \forall i \quad (12.3)$$

Using this operator the AR model can now be defined as

$$\left( 1 - \sum_{j=1}^p \alpha_j L^j \right) x_i = \epsilon_i \quad (12.4)$$

### 12.3.2 Moving Average (MA) Process

A moving average process is always stationary by design. A moving average or *MA* process is a process in which the current output is a moving average of a certain number of past states of the default white noise process. We can define a moving average process of order  $q$ ,  $MA(q)$  as

$$x_i = \epsilon_i + \beta_1 \epsilon_{i-1} + \dots + \beta_q \epsilon_{i-q} \quad (12.5)$$

which can be written using the lag operator as

$$x_i = \left( 1 + \sum_{k=1}^q \beta_k \cdot L^k \right) \epsilon_i \quad (12.6)$$

$\beta_i$  are the coefficients or parameters of the MA process.

### 12.3.3 Autoregressive Moving Average (ARMA) Process

Now, we combine the two processes into a single ARMA( $p, q$ ) process with parameters  $p$  and  $q$  to model a more generic process as

$$x_i = \alpha_1 x_{i-1} + \dots + \alpha_p x_{i-p} + \epsilon_i + \beta_1 \epsilon_{i-1} + \dots + \beta_q \epsilon_{i-q} \quad (12.7)$$

or using the lag operator as

$$\left( 1 - \sum_{j=1}^p \alpha_j L^j \right) x_i = \left( 1 + \sum_{j=1}^q \beta_j L^j \right) \epsilon_i \quad (12.8)$$

## 12.4 Autoregressive Integrated Moving Average (ARIMA) Models

Although ARMA( $p, q$ ) process in general can be non-stationary, it cannot explicitly model a non-stationary process well. This is why the ARIMA process is developed. The added term *integrated* adds differencing terms to the equation. The differencing process explicitly tries to remove the trends or seasonalities (essentially all the non-stationary components) from the data to make the residual process stationary. Differencing operation as the name suggests computes the deltas between consecutive values of the outputs as

$$x_d(i)^1 = x(i) - x(i-1) \quad (12.9)$$

Equation 12.9 shows the first-order differences. Differencing operation in discrete time is similar to differentiation or derivative operation in continuous time. First-order differencing can make polynomial-based second-order non-stationary processes into stationary ones, just like differentiating a second-order polynomial equation leads to a linear equation. Processes with a higher polynomial order non-stationarity need a higher-order differencing to convert into stationary processes. For example, second-order differencing can be defined as

$$x_d(i)^2 = x_d(i)^1 - x_d(i-1)^1 \quad (12.10)$$

$$x_d(i)^2 = (x(i) - x(i-1)) - (x(i-1) - x(i-2)) \quad (12.11)$$

$$x_d(i)^2 = x(i) - 2x(i-1) + x(i-2) \quad (12.12)$$

and so on. Using the lag operator, the same differencing operations can be written as

$$x_d(i)^1 = (1 - L)x_i \quad (12.13)$$

and

$$x_d(i)^2 = (1 - L)^2 x_i \quad (12.14)$$

This can be quickly generalized into any arbitrary order  $r$  as

$$x_d(i)^r = (1 - L)^r x_i \quad (12.15)$$

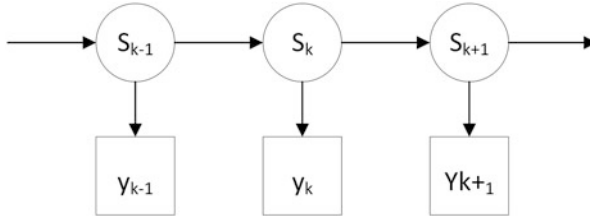
Now, we are ready to give the full equation of the ARIMA process. Let the parameter associated with the differencing operations be  $r$ . Thus ARIMA( $p, q, r$ ) process can be defined as

$$\left(1 - \sum_{j=1}^p \alpha_j L^j\right) (1 - L)^r x_i = \left(1 + \sum_{j=1}^q \beta_j L^j\right) \epsilon_i \quad (12.16)$$

Thus, ARIMA process generalizes the ARMA process for the cases with non-stationarity. When the value of  $r$  is 0, the ARIMA process reduces to ARMA process. Similarly, when  $r$  and  $q$  are 0, the ARIMA process reduces to AR process, and when  $r$  and  $p$  are 0, it reduces to MA process and so on.

## 12.5 Implementing AR, MA, ARMA, and ARIMA in Python

sklearn library does not offer tools for implementing the time series processes directly. However, there are numerous other libraries in open source that can be used, e.g., Darts [30] and Prophet [31]. Darts library offers explicit functions to use different models like ARIMA, and many more all the way into deep learning models, while Prophet abstracts the models and offers functions like fit and predict. The use of these libraries is fairly straightforward, and we will skip the details of implementations and leave it as an exercise.



**Fig. 12.4** A sequence of states and outcomes that can be modelled using hidden Markov models technique

## 12.6 Hidden Markov Models (HMMs)

Hidden Markov Models or HMMs represent a popular generative modeling tool in time series analysis. The HMMs have evolved from Markov processes in statistical signal processing. Consider a statistical process generating a series of observations represented as  $y_1, y_2, \dots, y_k$ . The process is called as a Markov process if the current observation depends only on the previous observation and is independent of all the observation before that. Mathematically it can be stated as

$$y_{k+1} = F(y_k) \quad (12.17)$$

where  $F$  is the probabilistic Markov function.

In HMM, there is an additional notion of state. Consider Fig. 12.4. The states are shown as  $s_{k-1}$ ,  $s_k$  and  $s_{k+1}$ , and the corresponding outcomes or observations are shown as  $y_{k-1}$ ,  $y_k$ , and  $y_{k+1}$ . The states follow Markov property such that each state is dependent on the previous state. The outcomes are probabilistic functions of only the corresponding states. The HMMs further assume that the states, although they exist, are invisible to the observer. The observer can only see the series of outcomes, but cannot know or see the actual states that are generating these outcomes. Mathematically it can be stated using two equations as

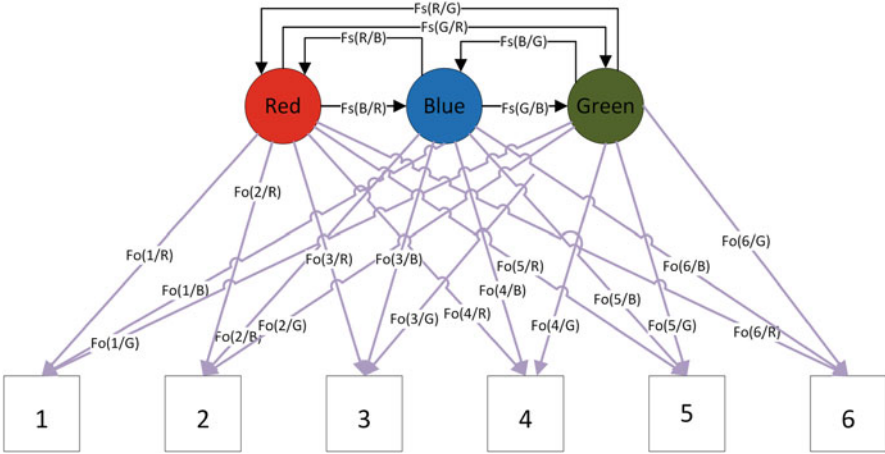
$$s_{k+1} = F_s(s_k) \quad (12.18)$$

where  $F_s$  is the probabilistic function of state transitions.

$$y_{k+1} = F_o(s_{k+1}) \quad (12.19)$$

where  $F_o$  is the probabilistic function of observations.

Consider a real-life example with three different states represented by three dies: red, blue, and green. Each die is biased differently to produce the outcomes of (1, 2, 3, 4, 5, 6). The state transition probabilities are given by  $F_s(s_i/s_j)$ , and outcome probabilities are given as  $F_o(s_i/o_j)$ . Figure 12.5 shows the details of the model.



**Fig. 12.5** Showing a full HMM with three states as three different dies—red, blue, and green—and six outcomes as 1, 2, 3, 4, 5, and 6

Once a given problem is modelled using HMM, there exist various techniques to solve the optimization problem using training data and predict all the transition and outcome probabilities [59].

### 12.6.1 Applications

HMMs have been widely used to solve the problems in natural language processing with notable success, e.g., part of speech (POS) tagging, speech recognition, machine translation, etc. As generic time series analysis problems, they are also used in financial analysis, genetic sequencing, etc. They are also used with some modifications in image processing applications like handwriting recognition.

## 12.7 Conditional Random Fields (CRFs)

Conditional random fields or CRFs represent a discriminative modeling tool as opposed to HMM which is a generative tool. CRFs were introduced by Lafferty et al. [89] in 2001. In spite of having a fundamental different perspective, CRFs share a significant architecture with HMM. In some ways CRFs can be considered as generalization of HMMs and logistic regression. As generative models typically try to model the structure and distribution of each participating class, discriminative models try to model the discriminative properties between the classes or the boundaries between the classes. As HMMs try to model the state transition

probabilities first and then the outcome or observation probabilities based on states, CRFs directly try to model the conditional probabilities of the observations based on the assumptions of similar hidden states. The fundamental function for CRF can be stated as

$$\hat{y} = \arg \max_y P(y/\mathbf{X}) \quad (12.20)$$

In order to model the sequential input and states, CRF introduces feature functions. The feature function is defined based on four entities.

### Entities in Feature Function

1. Input vectors  $\mathbf{X}$ .
2. Instance  $i$  of the data point being predicted.
3. Label for data point at  $(i - 1)$ th instance,  $l_{i-1}$ .
4. Label for data point at  $(i)$ th instance,  $l_i$ .

The function is then given as

$$f(\mathbf{X}, i, l_{i-1}, l_i) \quad (12.21)$$

Using this feature function, the conditional probability is written as

$$P(y/\mathbf{X}, \lambda) = \frac{1}{Z(\mathbf{X})} \exp \left( \sum_{i=1}^n \sum_j \lambda_j f_i(\mathbf{X}, i, y_{i-1}, y_i) \right) \quad (12.22)$$

where the normalization constant  $Z(\mathbf{X})$  is defined as

$$Z(\mathbf{X}) = \sum_{\hat{y} \in \mathcal{Y}} \sum_{i=1}^n \sum_j \lambda_j f_i(\mathbf{X}, i, y_{i-1}, \hat{y}_i) \quad (12.23)$$

## 12.8 Conclusion

Time series analysis is an interesting area in the field of machine learning that deals with data that is changing with time. The entire thought process of designing a time series pipeline is fundamentally different than the one used in all the static models that we studied in the previous chapters. In this chapter, we studied the concept of stationarity followed by multiple different techniques to analyze and model the time series data to generate insights.

## 12.9 Exercises

1. Use Darts library to model stock trends in a stock of mutual fund of your choice. Obtain the stock ticks from Yahoo finance or source of your choice for past 1 year. Keep the present date as 6 months in the past. Use the data before the set present, and try to predict the future values till today. Play with the duration of historical data and the length of future prediction window, and see the effect of increasing or decreasing the past duration on the accuracy of predictions.
2. Use Prophet library and repeat the experiment and see if you can find any differences and what parameters you can attribute them to.
3. Use NFL or any other sport of your choice, and try to predict the game outcomes using historical data. Try to gather as much features as possible, including but not limited to players, weather, past games by each team current win/loss states, etc.



# Chapter 13

## Deep Learning



### 13.1 Introduction

In his famous book, “*The Nature of Statistical Learning Theory*” [55], Vladimir Vapnik stated (in 1995) the four historical periods in the area of learning theory as:

1. Constructing first learning machine
2. Constructing fundamentals of the theory
3. Constructing the neural network
4. Constructing alternatives to the neural network

This list was meant as a subtle way of undermining the neural networks by pointing at their failures. Neural networks came with an implicit promise of replicating human brain. As our brain is capable of learning anything (humanly possible), great things were expected from neural networks. However, in large part, they were ineffective in delivering on that promise. Neural networks possess a generic architecture as described in Chap. 6. With appropriate labelled training data, they should be able to solve any problem in theory. The size and complexity of neural networks would have to increase with the complexity of the problem. If these neural networks were to be expanded into arbitrarily large size augmented with multiple hidden layers, they should be able to solve any desired problem. However, those architectures were simply inconceivable with the hardware at the time. Also, they were falling short of achieving convergence rates for desired performance levels. To set the context, an average human brain has roughly 100 billion neurons and over 1000 trillion synaptic connections [1]!! This would compare to a processor with trillion calculations/second processing power backed by over 1000 terabytes of hard drive. Even as of 2022 A.D., this configuration for a single computer

is far-fetched.<sup>1</sup> Typical hard drives are in the range of few terabytes, and high-end desktop CPUs are reaching about half trillion instructions per second [32] with multiple cores. The generic learning of neural network can only converge to sufficient accuracy if it is trained with sufficient labelled data followed by corresponding level of computation. The fact of the matter remained in the 1990s, where such computation was just beyond the scope of hardware and neural network as a technology was not able to realize its full potential.

To make matter worse for neural networks, at the same time, Vapnik and others came up with various alternatives to neural networks that were far more technologically feasible and efficient in solving the problems at hand at the time. Neural networks slowly became history as a result. However, something quite dramatic happened in the twenty-first century, and neural networks were brought back. Now, we are in the fifth historical period in the realm of learning theory, and we need to augment Vapnik's list with "Constructing re-birth of neural network" or "Constructing deep neural networks."

The dramatic shift that happened at the dawn of the twenty-first century that led to emergence of deep neural networks can be attributed to aggregation of these technological innovations:

1. Evolution of GPUs as general-purpose computing units or GPGPUs [2]. Specifically made popular by Nvidia as CUDA library. Even though CPUs in the computers were getting faster with Moore's law, the amount of processing needed for training extremely large and deep neural networks with equally large training sets could not be supported with CPUs. CPUs typically have handful of cores that are extremely powerful. But GPUs have a very different architecture with hundreds or even thousands of small processing units, where each unit is not nearly as powerful as a core in a CPU, but collectively they can offer an order of magnitude more computation power. There is one fundamental difference though. As the cores in GPU as independent entities, they can process large amount of information in parallel. However, that processing is disconnected, and if processing needed from one core has dependency on outcome of the processing happening in other cores, the whole architecture will be completely ineffective. Hence, if there are ways to create distributed chunks of compute blocks that can be later integrated effectively, GPUs can provide almost supercomputing level of capability from a simple gaming desktop machine.
2. Invention of modern optimization algorithms that addressed some of the crucial problems that had limited the learning capabilities of earlier algorithms. Some examples of such problems included:
  - a. Problem of vanishing gradient
  - b. Limitation to sequential processing

---

<sup>1</sup> The fastest supercomputer in the world is rated at 200 petaFLOPS or 200,000 trillion calculations/second. It is backed by storage of 250 petabytes or 250,000 terabytes. So it does have significantly better hardware than a single human brain, albeit at the cost of consuming 13 MW of power and occupying the entire floor of a huge office building [14].

3. Confluence of multiple techniques like concept of convolution from the theory of signal and image processing, recurrent networks from the early concept of Hopfield networks [91], backpropagation of errors, etc.
4. Availability of large amounts of data with labels coupled with increased storage capacity.

The current state of deep learning is made possible by the advances in computation hardware as well as the optimization algorithms that are needed to successfully use these resources and converge to the optimal solution. It is hard to name one person as inventor of deep learning, but there are few names that are certainly at the top of likely candidates. Geoffrey Hinton with his seminal paper titled *A fast learning algorithm for deep belief networks* [74] definitely sits at the top. Some of the early applications of deep learning were in the field of acoustic and speech modeling and image recognition. Reference [73] summarizes the steps in the evolution of deep learning as we see today quite nicely as well as concisely. Typically, when one refers to deep learning or deep networks, he/she is either referring to convolutional networks or recurrent networks or their variations. Not surprisingly, both of these concepts were invented well before the emergence of deep learning. Fukushima introduced convolutional networks back in 1980 [75], while Michael Jordan introduced recurrent neural networks in 1986 [76].

## 13.2 Why Deep Neural Networks?

Although technological and theoretical innovations were able to catch up with the hefty requirements of deep learning, it still remained an unanswered question as to why use them at the exorbitant cost of such hardware, when traditional machine learning techniques can do the same job with much less. The simple answer to this question was deep neural networks were able to produce results that could not be matched by the traditional algorithms, at least in certain niche problem spaces. Some of early success stories of deep neural networks were:<sup>2</sup>

1. In 2012, deep learning-based AlexNet algorithm achieved a top 5 error of 15.3% in the image classification problem based on famous ImageNet database. This error rate was whopping 10.8% points lower than the runner-up that was using traditional machine learning techniques. This groundbreaking victory really solidified the supremacy of deep learning techniques over traditional learning techniques.

---

<sup>2</sup> One of the most popular chess victories of IBM's Deep Blue supercomputer against the reigning world champion Gary Kasparov in 1997 is not a success story of deep learning. As a matter of fact, it was not even an example of artificial intelligence in true sense. It was more like an expert system developed specifically to excel in a single task: chess. Any artificial intelligence system (whether it is traditional ML or DL) is not restricted to solving a single problem.

2. Google's Deep Mind represents epitome of current state of the art in deep learning technology. In 2015, Google's AlphaGo program based on Deep Mind defeated European Go champion Fan Hui in the game of Go. In 2017, it beat the world's highest rated chess programs Stockfish and Elmo with only few hours of training. By this time, even these two programs had reached the level in chess unmatched by any human.
3. Generative Pre-Trained Transformer 3 or GPT3 [33] model developed by OpenAI [13] can produce not just one or two sentences but paragraphs and pages of text that even linguistics would have hard time believing to be produced by machine. It is one of the most complex deep neural networks in operation as of 2022 with about 175 billion parameters.

### 13.3 Types of Deep Neural Networks

Although the notion of deep neural network does not restrict them to be of certain types, there are about three to four different types of networks that are primarily studied and used under the umbrella of deep neural networks (DNN) or just deep networks (DN). They are:

1. Convolutional neural networks (CNNs)
2. Recurrent neural networks (RNNs)
3. Attention-based networks
4. Generative adversarial networks (GANs)

In the following sections, we will discuss architectures of these networks.

### 13.4 Convolutional Neural Networks (CNNs)

Convolutional neural networks or *CNNs* involve blocks based on the convolution operation. This operation has its roots in signal processing, and before going into details of CNN, let us look at the convolution process. The process of convolution can be operated in single dimension or any arbitrary number of dimensions.

#### 13.4.1 One-Dimensional Convolution

Mathematically, convolution defines a process by which one real valued function operates on another real valued function to produce new real valued function. Let one real valued continuous function be  $f(t)$  and the other be  $g(t)$ , where  $t$  denotes continuous time. Let the convolution of two be denoted as  $s(t)$ . Convolution operation is typically denoted as  $*$ .

$$f(t) * g(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau \quad (13.1)$$

Also, convolution is a commutative operation meaning  $(f * g)$  is the same as  $(g * f)$ ,

$$(f * g)(t) = (g * f)(t) = \int_{-\infty}^{\infty} g(\tau)f(t - \tau)d\tau \quad (13.2)$$

The same equations can also be written for discrete processes that we typically deal with in machine learning applications with computers. The discrete counterparts of the two equations can be written as  $f(k)$  and  $g(k)$ , where  $k$  denoted a discrete instance of time.

$$f(k) * g(k) = (f * g)(k) = \sum_{\delta=-\infty}^{\infty} f(\delta)g(k - \delta) \quad (13.3)$$

and

$$(f * g)(k) = (g * f)(k) = \sum_{\delta=-\infty}^{\infty} g(\delta)f(k - \delta) \quad (13.4)$$

These definitions can appear quite similar to the definition of correlation between two functions. The key difference here is the sign of  $\delta$  in case of discrete convolution and  $\tau$  in case of continuous convolution, which is opposite of  $f$  and  $g$ . If the sign is flipped, the same equations would represent correlation operation. The sign reversal makes one function reversed in time before being multiplied (point-wise) with the other function. This time reversal has far-reaching impact, and the result of convolution is completely different than the result of correlation. Convolution of one function with another is also called as filtering in the theory of signal processing. The convolution process has interesting implications from the perspective of Fourier transform [8] when the time domain functions are converted into frequency domain and is heavily used in signal processing applications.

### 13.4.2 Two-Dimensional Convolution

Just as convolution can be applied in case of one-dimensional functions, the concept can be extended to apply in higher dimensions. One-dimensional convolution is typically used in speech applications, where the signal is single dimensional. Concept of convolution can also be applied in two dimensions (or higher dimensions for that matter) as well, which makes it suitable for applications on images. In order to define two-dimensional convolution, let us consider two images  $A$  and  $B$ . The two-dimensional convolution can now be defined as

$$(A * B)(i, j) = \sum_m \sum_n A(m, n)B(i - m, j - n) \quad (13.5)$$

Typically the second image  $B$  is a small two-dimensional kernel compared to the first image  $A$ . The convolution operation transforms the original image into another one to transform the image in some desired way.

With the understanding of the process of convolution, we can now proceed to learn the architecture of convolutional neural networks.

### 13.4.3 Architecture of CNN

Figure 13.1 shows the architecture of CNN. The building block of CNN is composed of three units:

1. Convolutional layer
2. Rectified linear unit, also called as ReLU
3. Pooling layer

#### 13.4.3.1 Convolutional Layer

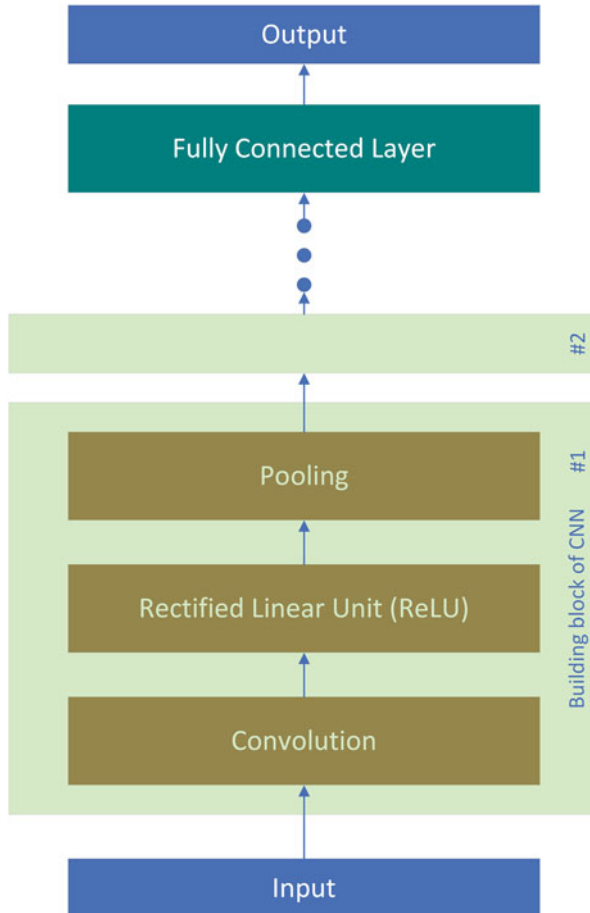
The convolutional layer consists of a series of 2D kernels. Each of this kernel is applied to original figure using 2D convolution defined in Eq. 13.5. This generates a 3D output.

#### 13.4.3.2 Rectified Linear Unit (ReLU)

Rectified linear unit or ReLU, as the name suggests, rectifies the output of convolutional layer to convert all the negative values to 0. The function is defined as

$$f(x) = \max(0, x) \quad (13.6)$$

The ReLU function is shown in Fig. 13.2. This layer also introduces nonlinearity into the network, which is otherwise linear. This layer does not change the dimensionality of the data. *Sigmoid* or *tanh* functions were used earlier to model the nonlinearity in a more continuous way, but it was observed that use of simpler ReLU function is just as good and it also improves the computation speed of the model and also fixes the vanishing gradient problem [77].

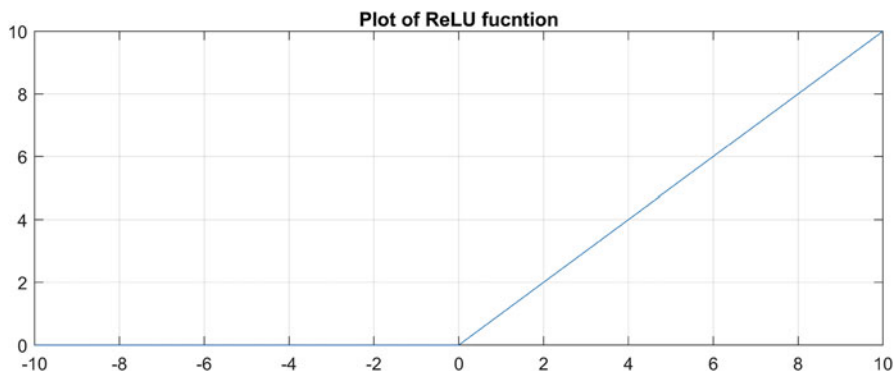


**Fig. 13.1** Architecture of convolutional neural network. The building block of the network contains the convolution and ReLU followed by pooling layer. A CNN can have multiple such layers in series. Then there is a fully connected layer that generates the output

### 13.4.3.3 Pooling

Pooling layer performs down-sampling by replacing larger sized blocks with single value. The most commonly used pooling method is called *max pooling*. In this method, simply the maximum value of the block is used to replace the entire block. This layer reduces the dimensionality of the data flowing through the network drastically while still maintaining the important information captured as a result of convolution operations. This layer also reduces overfitting.

These three layers together form the basic building block of a CNN. Multiple such blocks can be employed in single CNN.



**Fig. 13.2** Plot of the rectified linear unit (ReLU)

#### 13.4.3.4 Fully Connected Layer

The previously described layers essentially target certain spatial parts of the input and transform them using the convolutional kernels. The fully connected layer brings this information together in traditional MLP manner to generate the desired output. It typically uses softmax activation function as an additional step to normalize the output. Let the input to the softmax be a vector  $y$  of dimensions  $n$ . Softmax function is defined as

$$\sigma(y)_j = \frac{e^{y_j}}{\sum_{i=1}^n e^{y_i}} \quad (13.7)$$

Softmax function normalizes the output so that it sums to 1.

#### 13.4.4 Training CNN

CNNs are typically trained using stochastic gradient descent algorithm. Here are the main steps:

1. Gather the sufficient labelled training data.
2. Initialize the convolution filter coefficients weights.
3. Select a single random sample or a mini-batch of samples and pass it through the network and generate the output (class label in case of classification or real value in case of regression).
4. Compare the network output with the expected output, and then use the error along with backpropagation to update filter coefficients and weights at each layer.
5. Repeat steps 3 and 4 till algorithm converges to desired error levels.



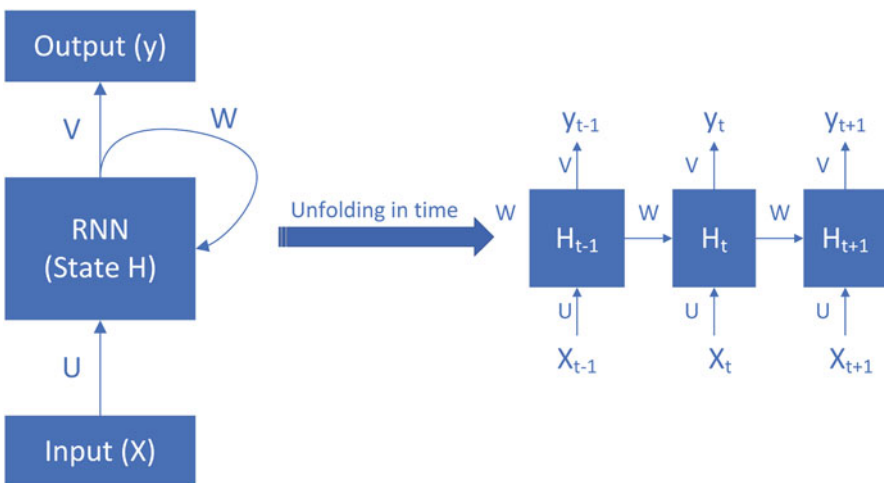
- Repeat the entire training process multiple times for different set of hyperparameters (which can be the size of convolution kernels, number of convolutional blocks, etc.) to find the optimal hyperparameters. This process is called as hyperparameter tuning.

### 13.4.5 Applications of CNN

Deep neural networks based on CNN architecture are typically found to be effective in image and video processing applications like image classification, object detection, and medical image analysis. They are also used effectively in recommender systems and time series analysis. Recently, they are also proving to be an effective tool in NLP applications. Authors have used the CNN architecture in a novel application of conflation of textual entities as detailed in [85].

## 13.5 Recurrent Neural Networks (RNNs)

All the traditional neural networks as well as CNNs are static models as defined in Chap. 2. They work with data that is already gathered and that does not change with time. Recurrent neural networks or *RNNs* propose a framework for dealing with dynamic or sequential data that changes with time. RNNs exhibit a concept of *state* that is a function of time. Classical RNNs, sometimes called as fully recurrent networks, have architecture very similar to MLP, but with addition of a feedback of current state as shown in Fig. 13.3. Thus, the output of the RNN can be given as



**Fig. 13.3** Architecture of classic or fully recurrent RNN. The time unfolded schematic shows how the state of RNN is updated based on sequential inputs

$$y_t = f_y(V.H_t) \quad (13.8)$$

The state of RNN is updated at every time instance using input and previous state as

$$H_t = f_H(U.X_t + W.H_{t-1}) \quad (13.9)$$

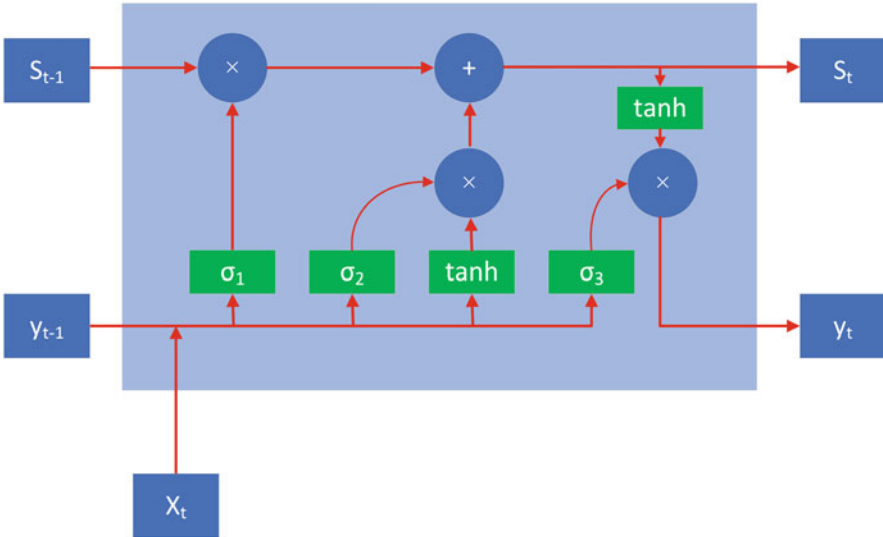
Training of the RNN is carried out using backpropagation method in a similar fashion using a labelled input sequence for which the output sequence is known. The operation of RNNs is typically described as encoder-decoder setup, where training phase operates as encoder by computing the parameters of the network and prediction step operates as decoder using the estimated parameters.

### 13.5.1 Limitation of RNN

RNNs were successful in modeling time series data that was not possible to model using traditional neural networks that dealt with static data. However, when the sequence of data to be analyzed was long with varying trends and seasonalities, the RNNs were not able to adapt to these conditions due to problems of exploding and vanishing gradients, oscillating weights, etc. [78]. The learning algorithms appeared to reach a limit after a certain number of samples were consumed, and any updates after that point were minuscule and not really changing the behavior of the network. Alternatively, in some cases, due to high volatility of the changes in training samples, the weights of the network kept on oscillating and would not converge.

### 13.5.2 Long Short-Term Memory RNN

Hochreiter and Schmidhuber proposed an improvement to the standard RNN architecture in the form of long short-term memory RNN or *LSTM-RNN*. This architecture improved the RNNs to overcome most of their limitations that are discussed in previous section. The fundamental change brought by this architecture was the use of forget gate. The LSTM-RNN had all the advantages of RNN and much reduced limitations. As a result, most of modern RNN applications are based on LSTM architecture. Figure 13.4 shows the components of LSTM-RNN. The architecture is quite complex with multiple gated operations. In order to understand the full operation, let's us dive deeper into the signal flow.  $S_t$  denotes the state of the LSTM at time  $t$ .  $y_t$  denotes the output of the LSTM at time  $t$ .  $X_t$  denotes the input vector at time  $t$ .  $\sigma_1$ ,  $\sigma_2$ , and  $\sigma_3$  denote the forget gate, input gate, and output gate, respectively. Let us look at the operations at each gate.



**Fig. 13.4** Architecture of LSTM-RNN.  $\sigma_1$  denotes the forget gate,  $\sigma_2$  denotes the input gate, and  $\sigma_3$  denotes the output gate

### 13.5.2.1 Forget Gate

At forget gate, the input is combined with the previous output to generate a fraction between 0 and 1, which determines how much of the previous state needs to be preserved (or in other words, how much of the state should be forgotten). This output is then multiplied with the previous state.

### 13.5.2.2 Input Gate

Input gate operates on the same signals as the forget gate, but here the objective is to decide which new information is going to enter the state of LSTM. The output of the input gate (again a fraction between 0 and 1) is multiplied with the output of  $\tanh$  block that produces the new values that must be added to previous state. This gated vector is then added to previous state to generate current state.

### 13.5.2.3 Output Gate

At output gate, the input and previous state are gated as before to generate another scaling fraction that is combined with the output of  $\tanh$  block that brings the current state. This output is then given out. The output and state are fed back into the LSTM block as shown.

### 13.5.3 Advantages of LSTM

The specific gated architecture of LSTM is designed to improve all the following shortcomings of the classical RNN:

1. Avoid the exploding and vanishing gradients, specifically with the use of forget gate at the beginning.
2. Long-term memories can be preserved along with learning new trends in the data. This is achieved through combination of gating and maintaining state as separate signal.
3. A priori information on states is not required, and the model is capable of learning from default values.
4. Unlike other deep learning architectures, there are not many hyperparameters needed to be tuned for model optimization.

### 13.5.4 Applications of LSTM-RNN

LSTM is one of the important areas in cutting edge of machine learning, and many new variants of the model are proposed including bidirectional LSTM, continuous-time LSTM, hierarchical LSTM, etc. The main applications of RNNs are in problems having time-varying data, e.g., language understanding, machine translation, speech recognition/classification, video tagging, etc. However, just as CNNs have been effectively used in applications where RNNs typically excel, RNNs have also been used in image processing applications like face detection [92], OCR applications, etc. where CNNs are default choice.

## 13.6 Attention-Based Networks

LSTM-RNN improved on the concept of recurrent neural networks by adding multiple gates and focusing on the specific information that has optimal impact over the subsequent outcomes. The RNNs were further improved with adding bidirectional operation to make them agnostic of the strictly causal implementation, where it is not needed. Even, then in case of very long sequence of data, LSTM architecture was falling short. Especially applications in natural language processing (NLP) were exposing the limitations of the architecture. A well-known paper called “Attention is All You Need” by Google Brain [93] introduced a new concept to further advance the concept with notion of *Attention*.

In many problems in the NLP area, even if the input can be very long, the entire output is not uniformly dependent on the entire input. There can be parts of the output that depend on parts of the input. A classic example of such problem is machine translation. The input is a long paragraph in input language, and

the expected output is its translation. Each and every word in the output is not necessarily dependent on all the words in the input. However, translation of the first word in the first sentence will only have reasonable dependence on few words after it and so on. Attention mechanism specifically trains the network to focus on the local context to improve the accuracy as well as reduce the complexity of the network overall. The implementation of the algorithm involves addition of additional weights that define the context. The weights are also trained along with other parameters of the network. [93] explains the details of the mathematical implementation.

The deep learning models using the concept of attention are typically called as transformer models, and one of the most successful implementations of transformer model is called BERT [34] (Bidirectional Encoder Representations from Transformers). GPT3 also uses the transformer architecture.

## 13.7 Generative Adversarial Networks (GANs)

We discussed generative models in Chap. 9. Generative models typically take a more holistic approach toward understanding and solving the problem, by trying to model the structure and distribution of the data. *Generative adversarial networks* or *GANs* are a combination of two neural networks as suggested in the name: generative network and adversarial network. The generative network is designed to imitate the process that generates the sample data, while adversarial network is a type of discriminative network that checks if the data generated by the generative network belongs to the desired class. Thus, these two networks working together can generate any arbitrary type of synthetic data. In order to better understand the architecture of GANs, let's take an example of image synthesis. We would like to generate images of human faces that look like faces of real people, but are completely computer generated. Figure 13.5 shows the architecture of GAN to solve this problem.

The generative network is given input as random noise to generate a synthetic face. The generated faces are compared with a database of real human face images by the adversarial discriminative network to generate a binary classification output: *real* or *fake*. This output is fed back to train both the generative networks. With availability of sufficiently large labelled database of human face images, the generative network can be trained to generate synthetic but realistic human face images. In theory, the concept of GANs need not be limited to deep learning, but the typical complexity involved in the generative models can only be addressed with deep neural networks. GANs represent one of the novel applications in the field of machine learning, specifically in the field of deep learning that was introduced in 2014 by *Ian Goodfellow et al.* [63].

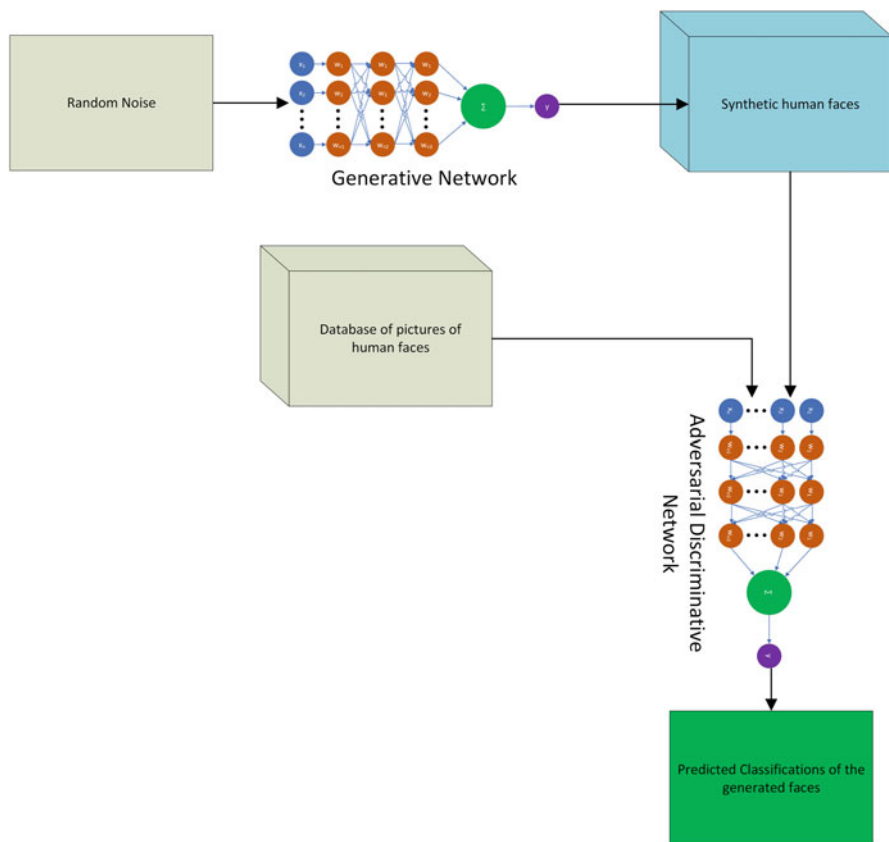


Fig. 13.5 Architecture of generative adversarial network

## 13.8 Implementing Deep Learning Models

The sklearn library that we have been using for most implementations does not support deep learning. Rather most libraries that support traditional machine learning algorithms do not support deep learning. As a result, we need to explore new set of libraries that are specifically designed for deep learning. There are multiple reasons for this inherent separation:

1. Use of GPUs as compute units: Most deep learning computations are so complex that it would take weeks if not more to train using just the CPUs.
2. Complex training algorithms: Deep neural networks need more sophisticated and customized optimization algorithms, as traditional optimization algorithms fall short when dealing with the scale of data required for deep learning.

3. There are handful of architectures that are commonly used in deep learning, and they need fairly long list of repetitive steps. These deep learning libraries implement and abstract these steps to make the training process much simpler.

Even until only a few years ago, setting up your local machine for deep learning was a very long and tedious process, and not enough documentation existed online, and almost no free online resource like Google Colab or AzureML supported deep learning. However, fortunately, things have improved a lot, and setting up your machine has become easier, and online tools have also started supporting deep learning to some extent for free.

We will use the familiar Google Colab online tool where we can use Google's own TensorFlow [35] deep learning library. We will try to solve the well-known problem of image classification for CIFAR-10 dataset [45]. This dataset contains over 60,000 color images from 10 classes. The data is uniformly distributed with 6000 images of each class. The classes are:

1. Airplane
2. Automobile
3. Bird
4. Cat
5. Deer
6. Dog
7. Frog
8. Horse
9. Ship
10. Truck

All the images are normalized to standard dimensions of  $32 \times 32$ . Also the data comes pre-separated into training and test sets with 50,000 images in training set and 10,000 images in test set. We will use convolutional neural networks (CNNs) to solve this problem. Figure 13.6 shows how we import the TensorFlow library in Colab and load the CIFAR-10 data.



```
# Import TensorFlow and print the version
import tensorflow as tf
print(tf.__version__)

2.9.2


# Load the CIFAR-10 image dataset
cifar10 = tf.keras.datasets.cifar10
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [=====] - 3s 0us/step
```

Fig. 13.6 Import TensorFlow library and load the CIFAR-10 dataset

```
[3] # Define the labels
labels = ['airplane', 'automobile', 'bird', 'cat', 'deer',
         'dog', 'frog', 'horse', 'ship', 'truck']

# Plot sample images from training set
import matplotlib.pyplot as plt
fig, axes = plt.subplots(ncols=5, nrows=3, figsize=(15,6))
ctr = 0
for r in range(3):
    for c in range(5):
        axes[r,c].set_title(labels[train_labels[ctr][0])
        axes[r,c].imshow(train_images[ctr])
        axes[r,c].get_xaxis().set_visible(False)
        axes[r,c].get_yaxis().set_visible(False)
        ctr += 1
plt.show()
```



**Fig. 13.7** Define the 10 class labels and plot first 15 images to understand what we are dealing with

After loading the data, we first display some images to understand the problem we are dealing with, such as the quality of images, their size, etc. We also define all the class labels as shown in Fig. 13.7.

Dealing with color images means tripling the dimensionality of the problem. It is not impossible, but it increases the complexity of the problem significantly. In most image processing situations, the gray scale images (where each pixel only possesses information about the brightness) are sufficient. In order to make the educated decision, we convert all the images into gray scale and plot them again. As we can see in Fig. 13.8, we can still identify the images as belonging to their class. Thus, as far as classification problem is concerned, we can work with gray scale images.

The original gray scale pixels typically contain 8-bit resolution with range from 0 to 255. To use this data with neural networks, we convert these values into floating point numbers and normalize them to stay within the range of 0.0–1.0. Also, we need to encode the labels which are from 1 to 10 into one ten-dimensional binary number



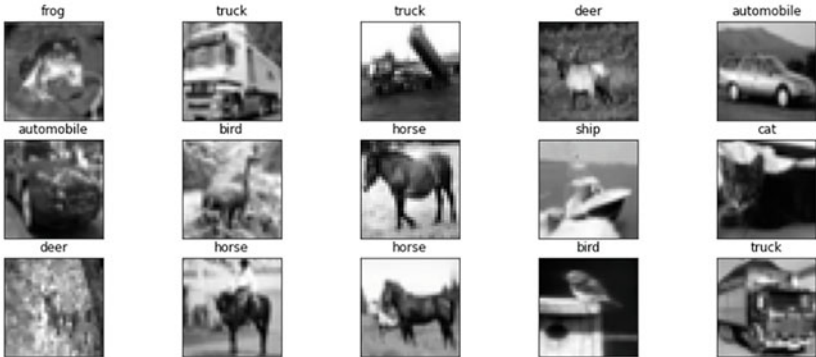
```

▶ # Import necessary libraries
import numpy as np
import cv2

# Convert color images to grayscale
train_images_gray = np.array([cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
                              for image in train_images])
test_images_gray = np.array([cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
                              for image in test_images])

▶ fig, axes = plt.subplots(ncols=5, nrows=3, figsize=(15, 6))
ctr = 0
for r in range(3):
    for c in range(5):
        axes[r,c].set_title(labels[train_labels[ctr][0]])
        axes[r,c].imshow(train_images_gray[ctr], cmap='gray')
        axes[r,c].get_xaxis().set_visible(False)
        axes[r,c].get_yaxis().set_visible(False)
        ctr += 1
plt.show()

```



**Fig. 13.8** Convert the color images to gray scale and view them again to compare

using one-hot encoding technique as shown in Fig. 13.9. This is an important step. It ensures that the neural network generates a ten-dimensional output with each dimension identifying the corresponding class.

Now, we are ready to build our CNN model. We still need to import all the necessary libraries. Then as described in the chapter earlier, we start with building each layer of the model. First, we add multiple 2D convolutional layers followed by *MaxPool* layers. These layers will extract positional information from the images. The first parameter defines the number of filters the convolutional layer will learn. We start with 32 and then increase it to 64. This number is generally kept close to the size of the image; however, it is not mandatory. There is no strict rule in what kernel size convolution to apply (we have used  $3 \times 3$  in this case), and one can experiment with different options. Then we use the ReLU activation function as defined earlier and use the padding to maintain the same size as shown in Fig. 13.10.

```
[ ] # Check the shape of images
train_images_gray.shape, test_images_gray.shape

((50000, 32, 32), (10000, 32, 32))

[ ] # Normalizing the grayscale images to 0.0 to 1.0
train_images_gray = train_images_gray/255.0
test_images_gray = test_images_gray/255.0

[ ] # Encoding the labels using One Hot Encoding
from sklearn.preprocessing import OneHotEncoder
ohe = OneHotEncoder(sparse=False).fit(train_labels)
train_labels_enc = ohe.transform(train_labels)
test_labels_enc = ohe.transform(test_labels)

[ ] # Define input shape
in_shape = (train_images_gray.shape[1], train_images_gray.shape[2], 1)
```

**Fig. 13.9** Normalize the image data to have all the values between 0.0 and 1.0 and encode the labels

```
# Import necessary modules from Keras
from keras.layers import Conv2D, MaxPool2D, Flatten, Dense, Dropout
from keras.models import Sequential, load_model
from keras.callbacks import EarlyStopping

# Build Model
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', strides=(1, 1), padding='same',
                input_shape=in_shape))
model.add(MaxPool2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu', strides=(1, 1), padding='same'))
model.add(MaxPool2D((2, 2)))
model.add(Conv2D(32, (3, 3), activation='relu', strides=(1, 1), padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu', strides=(1, 1), padding='same'))
model.add(MaxPool2D((2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))

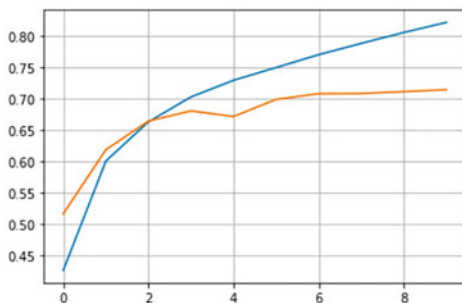
[ ] model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['acc'])
```

**Fig. 13.10** Import CNN libraries from TensorFlow and build the model architecture

After multiple convolutional and max pooling layers, we add the dense fully connected layers to bring the positional information together to finally predict the ten classes. For the last prediction layer, we use the softmax activation function. We use *Adam* optimizer and compile the model with *cross entropy* as loss function and *accuracy* as the metric.

```
[ ] history = model.fit(train_images_gray, train_labels_enc, epochs=10, validation_data=
Epoch 1/10
1563/1563 [=====] - 121s 77ms/step - loss: 1.5893 - acc: 0.4
Epoch 2/10
1563/1563 [=====] - 109s 70ms/step - loss: 1.1345 - acc: 0.6
Epoch 3/10
1563/1563 [=====] - 111s 71ms/step - loss: 0.9651 - acc: 0.6
Epoch 4/10
1563/1563 [=====] - 111s 71ms/step - loss: 0.8550 - acc: 0.7
Epoch 5/10
1563/1563 [=====] - 111s 71ms/step - loss: 0.7767 - acc: 0.7
Epoch 6/10
1563/1563 [=====] - 108s 69ms/step - loss: 0.7114 - acc: 0.7
Epoch 7/10
1563/1563 [=====] - 107s 68ms/step - loss: 0.6575 - acc: 0.7
Epoch 8/10
1563/1563 [=====] - 110s 70ms/step - loss: 0.6035 - acc: 0.7
Epoch 9/10
1563/1563 [=====] - 109s 70ms/step - loss: 0.5545 - acc: 0.8
Epoch 10/10
1563/1563 [=====] - 109s 70ms/step - loss: 0.5117 - acc: 0.8
```

```
[ ] plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.grid()
plt.show()
```



**Fig. 13.11** Train the model and see how the model converges after each epoch

The next step is training. We use ten epochs and start the training process. One can experiment with more or less number of epochs. Then we plot the training process to visualize how the accuracy improved at every epoch. The blue line shows the improvement in the accuracy on training data, which is fairly consistent. The orange line shows the accuracy on the test data. It is important to note that we are not taking the feedback from the accuracy of test data after each epoch, and as a result, this improvement may not be uniform. The accuracy on test set actually drops a bit in fourth epoch, but ultimately it gets better. Figure 13.11 shows the process.

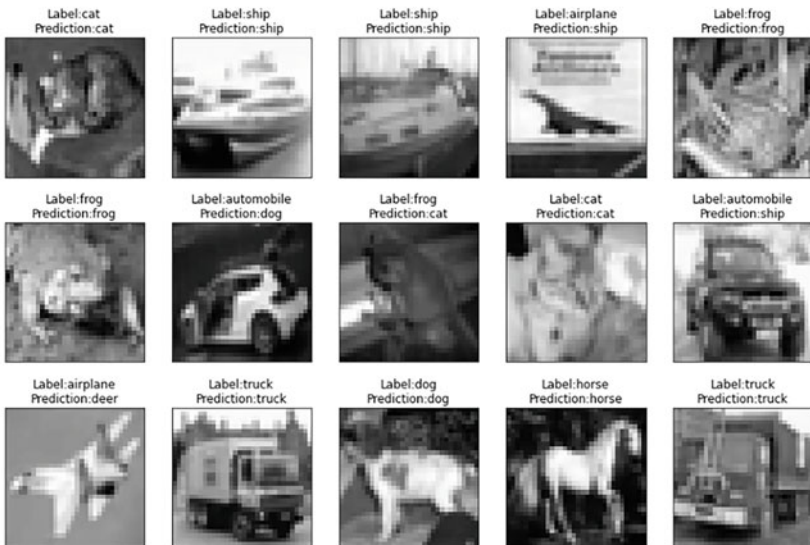
```

▶ predictions = ohe.inverse_transform(model.predict(test_images_gray))
test_labels_int = test_labels.astype(int)
predictions_int = predictions.astype(int)

[ ] fig, axes = plt.subplots(ncols=5, nrows=3, sharex=False, sharey=True,
                             figsize=(15, 10))

index = 0
for i in range(3):
    for j in range(5):
        axes[i,j].set_title('Label:' + labels[test_labels_int[index][0]] +
                            '\n' + 'Prediction:' + labels[predictions_int[index][0]])
        axes[i,j].imshow(test_images_gray[index], cmap='gray')
        axes[i,j].get_xaxis().set_visible(False)
        axes[i,j].get_yaxis().set_visible(False)
        index += 1
plt.show()

```



**Fig. 13.12** Apply the trained model on test set and compare predictions with labels

```

▶ test_loss, test_acc = model.evaluate(test_images_gray, test_labels_enc, verbose=2)
313/313 - 5s - loss: 0.8966 - acc: 0.7139 - 5s/epoch - 16ms/step

```

**Fig. 13.13** Evaluate the model accuracy

After the training is complete, we apply the trained model to make the predictions. Figure 13.12 shows the labels and predictions for the first 15 images from test set. We are accurate in predicting most images, but there are some errors.

In the end, we can look at the overall accuracy on the test set as 0.7139 as shown in Fig. 13.13, which is a great start for such a small piece of code.

## 13.9 Conclusion

We studied deep learning techniques in this chapter. The theory builds on the basis of classical perceptron or single-layer neural network and adds more complexity to solve different types of problems given sufficient training data. We studied two specific applications in the form of convolutional neural networks and recurrent neural networks.

### 13.10 Exercises

1. Re-create the experiment with CIFAR-10 data with making the following changes:
  - a. Add or remove convolutional layers.
  - b. Change the kernel size.
  - c. Add or remove max pooling layers.
  - d. Change filter size.
  - e. Change padding options.

Observe the effect of each change on the final accuracy as well as the accuracy improvement after each epoch. Draw and record the conclusions. Also record how the accuracy on the test set varies.

2. Try changing the parameters of training and repeat the experiment.
3. Use the color images instead of gray scale. Use the same experimental setup except the dimensionality handling aspects, and observe the changes in accuracy/time for training. Draw conclusion if keeping the color information is worth the efforts.
4. Reduce the gray scale image size from  $32 \times 32$  to  $16 \times 16$  and rerun the experiment. See how much accuracy is lost due to reduction in the data.
5. Repeat above experiment with color images and compare the results.

# Chapter 14

## Unsupervised Learning



### 14.1 Introduction

Unsupervised learning methods deal with problems where the labelled data is not available. There is no form of supervised feedback to be gained about the processing that happens. Absence of labelled samples marks a fundamental shift of thought from the other supervised methods discussed so far in the book. In some sense, one can also think that these methods do not belong to the area of machine learning, as there no learning based on feedback. Nonetheless, unsupervised learning is very much an integral part of machine learning. It is important to understand that even humans are continuously learning patterns from the data that is obtained from the sensory organs without any feedback.

It might also appear that such processing might not yield any useful information, and one could not be more wrong. There are many situations where unsupervised methods are extremely valuable. The first and foremost is cost of labelling. In many situations, having all the training data fully labelled can be rather expensive and practically impossible, and one needs to work with only small set of labelled data. In such cases, one can start with supervised methods using small set of labelled data and then grow the model in unsupervised manner on larger set of data. In other cases, the labels may not be available at all, and one needs to understand the structure and composition of data by doing exploratory unsupervised analysis. Understanding the structure of the data can also help with choosing the right algorithm or better set of initialized parameters for the algorithm in cases where supervised methods will be ultimately used. In general, unsupervised learning marks an important pillar of modern machine learning.

In this chapter, we will learn about the following aspects of unsupervised learning:

1. Clustering
2. Component analysis
3. Self-organizing maps (SOMs)
4. Autoencoding neural networks

## 14.2 Clustering

Clustering is one of the most fundamental and also simplest forms of unsupervised learning. When there is a sample population, clustering methods try to aggregate the samples into two or more groups. The criteria used for deciding the membership to a group are determined by using some form of metric or distance. Clustering being one of the oldest methods in the machine learning repertoire, there are numerous methods described in the literature. We will focus on one of the simple yet quite effective methods which with the help of certain modifications can tackle broad range of clustering problems. It is called as *k-means clustering*. The variable  $k$  denotes the number of clusters. The method expects the value of  $k$  to be determined before starting to apply the algorithm, although it can automatically increase or decrease  $k$  during the process.

### 14.2.1 *k*-Means Clustering

Figure 14.1 shows two examples of extreme cases encountered in case of clustering. In top figure, the data is naturally distributed into separate non-overlapping clusters, while the bottom figure shows the case where there is no natural separation between the data. Most cases encountered in practice are somewhere in between.

The steps in  $k$ -means clustering algorithm are:

1. Start with a default value of  $k$ , which is the number of clusters to find in the given data.
2. Randomly initialize the  $k$  cluster centers as  $k$  samples in training data, such that there are no duplicates.
3. Assign each of the training samples to one of the  $k$  cluster centers based on a chosen distance metric.
4. Once the classes are created, update the centers of each class as mean of all the samples in that class.
5. Repeat steps 2–4 until there is no change in the cluster centers.

The distance metric used in the algorithm is typically the Euclidean distance. However, in some cases, different metrics like Mahalanobis [16] distance,

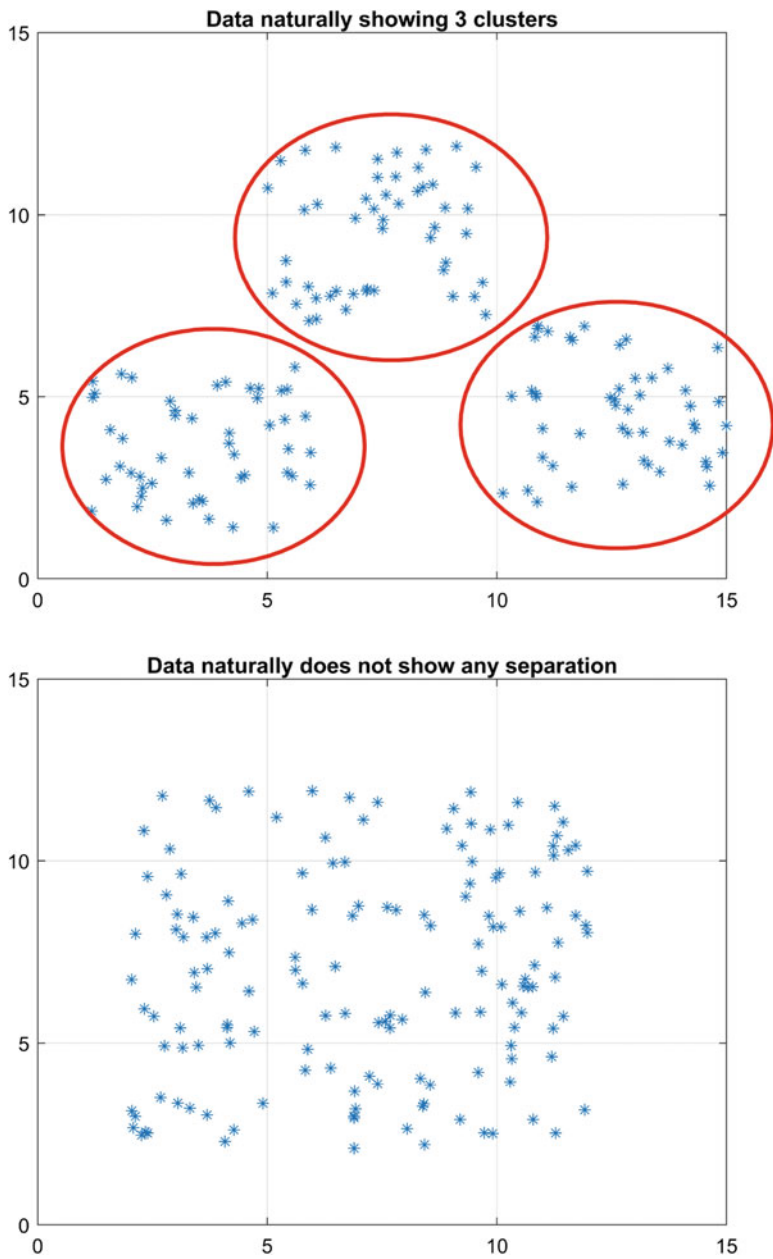


Fig. 14.1 Figures showing the extreme cases of data distribution for clustering



Minkowski [36] distance, or Manhattan [37] distance can be used depending on the problem at hand.<sup>1</sup>

Figure 14.2 shows the algorithm in intermediate stages as it converges to desired clusters. This is somewhat an ideal case. In most practical situations, where the clusters are not well separated, or the number of naturally occurring clusters is different than the initialized value of  $k$ , the algorithm may not converge. The cluster centers can keep oscillating between two different values in subsequent iterations, or they can just keep shifting from one set of clusters to another. In such cases, multiple optimizations of the algorithms are proposed in the literature [60]. Some of the optimizations that are commonly used are as follows:

### Optimizations for k-Means Clustering

1. Change the stopping criterion from absolute “no change” to cluster centers to allow for a small change in the clusters.
2. Restrict the number of iterations to a maximum number of iterations.
3. Find the number of samples in each cluster, and if the number of samples are less than a certain threshold, delete that cluster and repeat the process.
4. Find the intra-cluster distance versus inter-cluster distance, and if two clusters are too close to each other relative to other clusters, merge them and repeat the process.
5. If some clusters are getting too big, apply a threshold of maximum number of samples in a cluster, and split the cluster into two or more clusters and repeat the process.

## 14.2.2 Improvements to k-Means Clustering

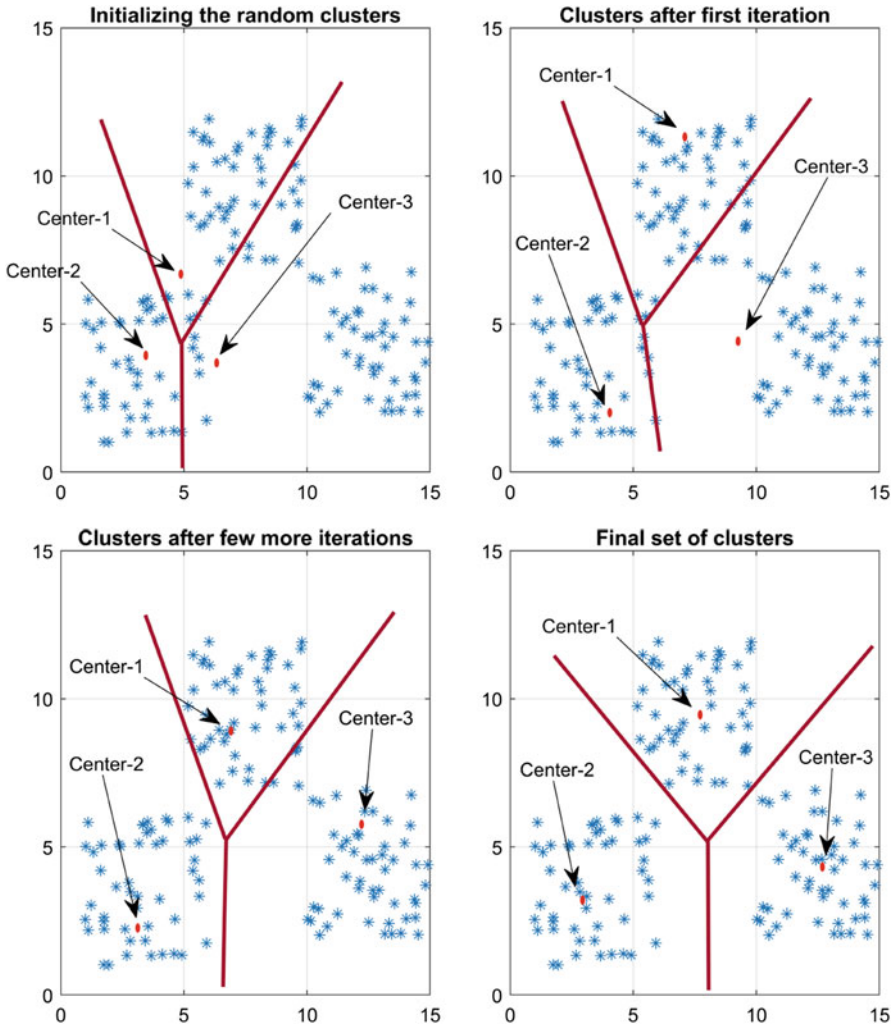
Even after the multiple optimizations are described in previous subsection, there are cases when the results are still suboptimal and some further improvements can be applied.

### 14.2.2.1 Hierarchical k-Means Clustering

In some cases, using the same k-means clustering algorithm recursively can be helpful. After each successful completion of the clustering, new set of random clusters are initialized inside of each cluster, and the same algorithm is repeated

---

<sup>1</sup> There exists another clustering algorithm called as *k-nearest neighbor* or *KNN*, which might appear to be related to K-means clustering but in fact is a completely different concept. It is actually a supervised clustering algorithm that looks at  $k$  number of nearest neighbors to an unclassified sample to classify it as one of the other classes based on the distribution of classes in the  $k$  neighbors.



**Fig. 14.2** Figures showing the progress of k-means clustering algorithm iterating through steps to converge on the desired clusters

in the subset of each cluster to find the sub-clusters. This is called as hierarchical k-means clustering method.

### 14.2.2.2 Fuzzy k-Means Clustering

In traditional k-means clustering algorithm after the cluster centers are chosen or updated, all the training samples are grouped into nearest cluster. Instead of using

such absolute grouping, fuzzy k-means algorithm suggests a use of probabilistic grouping. In this case, each sample has non-zero probability of belonging to multiple clusters at the same time. The nearer the cluster, the higher the probability and so on.

### 14.2.3 Implementing k-Means Clustering

Sklearn library does provide algorithms for k-means clustering and its variations, and we will use the same for illustrating the implementation of k-means clustering algorithm. It is important to remember that k-means clustering being an unsupervised method, we will not use the labels in the process. Figure 14.3 shows the code, and Fig. 14.4 shows the outcome of the clustering. In order to view the class distribution easily, we will only use the first two dimensions from the data and will ignore the remaining two dimensions.

As we can see from the figure, the k-means clustering does a perfect job in discriminating the third class, but there is some overlap with the first and second class. Also, note that the class names given by clustering between 1 and 2 are flipped, and it can be quite random. With different random initializations, the class names

```
[13] from sklearn import datasets
iris = datasets.load_iris()
X = iris.data[:,[1,2]]
y = iris.target

from sklearn.cluster import KMeans
km = KMeans(n_clusters=3, random_state=0).fit(X)
y_pred = km.predict(X)
```

Fig. 14.3 Implementing k-means clustering using sklearn on Iris data (only first two dimensions)

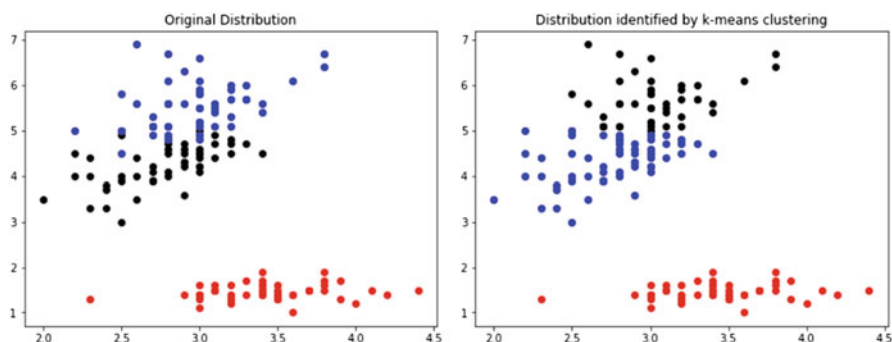


Fig. 14.4 Comparing the classification of the Iris data by k-means clustering

can be arbitrary. This example illustrates the power of unsupervised learning when there exists a real separation between two or more classes of samples. Even without the knowledge of class labels, the clustering method is able to detect and classify the data.

## 14.3 Component Analysis

Another important aspect of unsupervised machine learning is dimensionality reduction. Component analysis methods are quite effective in this regard. Principal component analysis or PCA is one of the most popular techniques in dimensionality reduction in the theory of machine learning as we saw in Chap. 3. In this chapter, we will look at another important technique similar to PCA called independent component analysis or ICA.

### 14.3.1 Independent Component Analysis (ICA)

Although PCA and ICA are both generative methods of extracting the core dimensionality of the data, they both differ in the underlying assumptions. PCA uses variation in the data and tries to model it to find the dimensions in ranked order where the variation is maximized. Matrix algebra and singular value decomposition (SVD) tools are used to find these dimensions. ICA takes a very different and more probabilistic approach toward finding the core dimensions in the data by making the assumption that the given data is generated as a result of combination of a finite set of independent components. These independent components are not directly observable and hence sometimes referred to as latent components. Mathematically, the ICA can be defined for given data  $(x_i), i = 1, 2, \dots, n$  as

$$x_i = \sum_{j=1}^k a_j s_j, \forall i \quad (14.1)$$

where  $a_j$  represent weights for corresponding  $k$  number of  $s_j$  independent components. The cost function to find the values of  $a_j$  is typically based on mutual information. The fundamental assumption in choosing the components is that they should be statistically independent with non-Gaussian distributions. Conceptually, the process of finding ICA is quite similar to the topic of blind source separation in statistics. Typically, in order to make the predictions more robust, a noise term is added into Eq. 14.1.

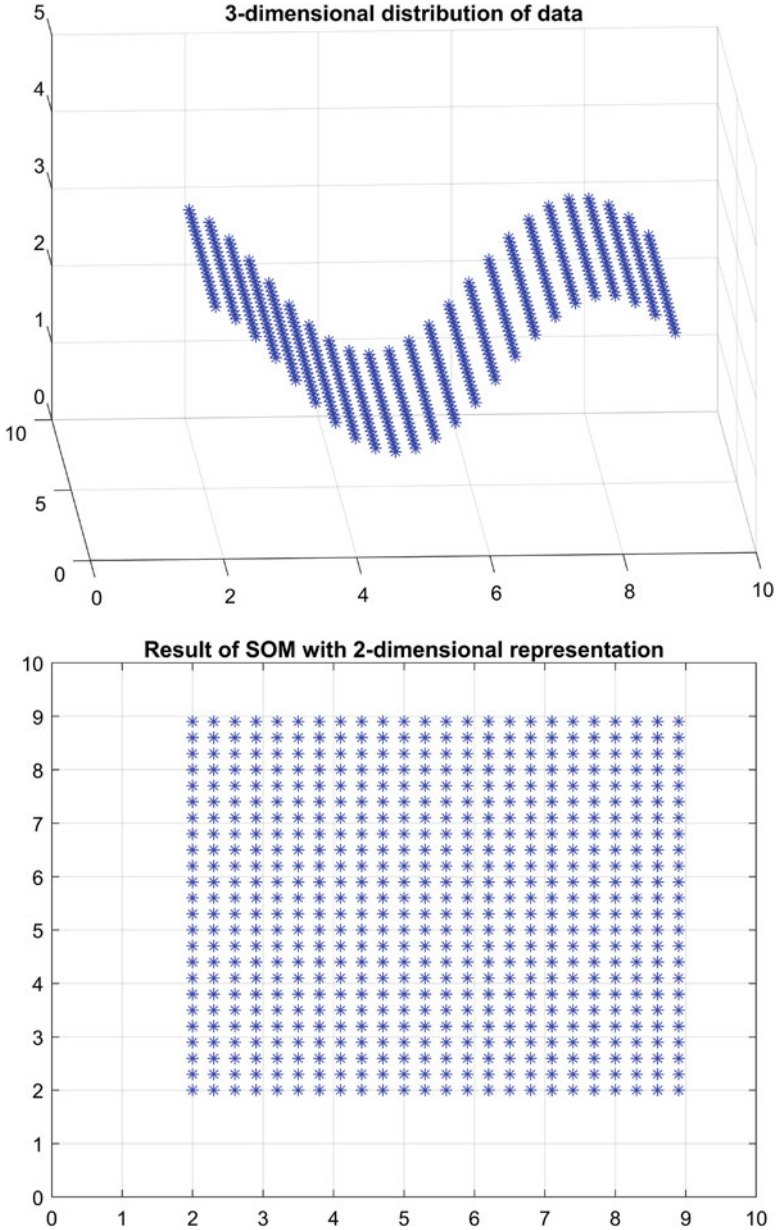
## 14.4 Self-Organizing maps (SOMs)

Self-organizing maps, also called as self-organizing feature maps, present a neural network-based unsupervised learning system, unlike other options discussed before. The neural networks are inherently supervised methods of learning, so their use in unsupervised class of methods is quite novel in that sense. SOMs define a different type of cost function that is based on similarity in the neighborhood. The idea here is to maintain the topological distribution of the data while expressing it in smaller dimensions efficiently. In order to illustrate the functioning of SOM, it will be useful to take an actual example. Top plot in Fig. 14.5 shows data distributed in three dimensions. This is a synthetically generated data with ideal distribution to illustrate the concept. The data is essentially a two-dimensional plane folded into three dimensions. SOM unfolds the plane back into two dimensions as shown in the bottom figure. With this unfolding, the topological behavior of the original distribution is still preserved. All the samples that are neighbors in original distribution are still neighbors. Also, the relative distances of different points from one another are also preserved in that order.

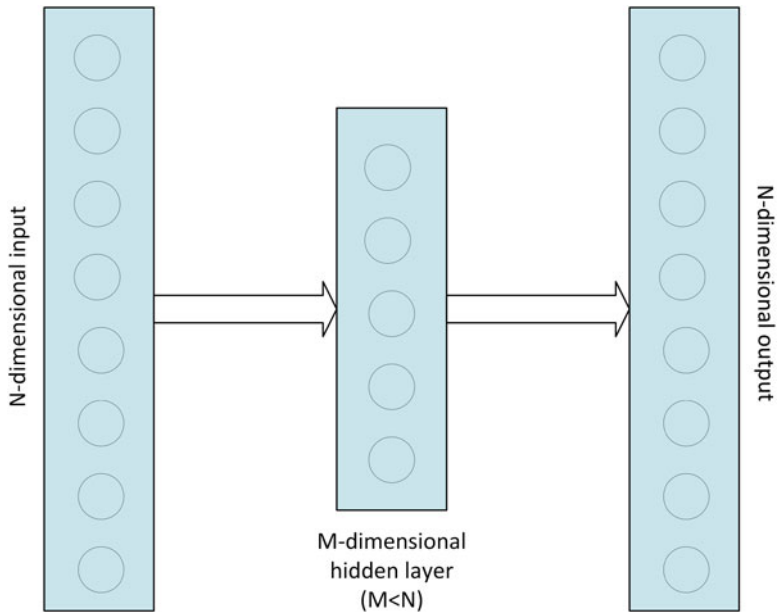
The mathematical details of the cost function optimization for SOM can be found here [20]. The representation generated by SOM is quite useful and is generally better than the first principal component as predicted by PCA. However, PCA also provides multiple subsequent components to fully describe the variation in the data as needed, and SOM lacks this capability. However, if one is only interested in efficient representation of the data, SOM provides a powerful tool.

## 14.5 Autoencoding Neural Networks

Autoencoding neural networks or just autoencoders are a type of neural networks that work without any labels and belong to the class of unsupervised learning. Figure 14.6 shows architecture of autoencoding neural network. There is input layer matching the dimensionality of the input and hidden layer with reduced dimensionality followed by an output layer with the same dimensionality as input. The target here is to regenerate the input at the output stage. The network is trained to regenerate the input at the output layer. Thus, the labels are essentially the same as input. The unique aspect of autoencoding networks is reduced dimensionality of the hidden layer. If an autoencoding network is successfully trained within the required error margins, then in essence we are representing the input in lesser dimensional space in the form of coefficients of the nodes at hidden layer. Furthermore, the dimensionality of the hidden layer is programmable. Typically, with the use of linear activation functions, the lower dimensional representation generated by the autoencoding networks resembles the dimensionality reduction obtained from PCA.



**Fig. 14.5** Figures showing original three-dimensional distribution of the data and its two-dimensional representation generated by SOM. The SOM essentially unfolds two-dimensional plane folded into three-dimensional space



**Fig. 14.6** Figure showing architecture of autoencoding neural network

## 14.6 Conclusion

In this chapter, we discussed the new type of algorithms in machine learning literature known as unsupervised learning. These algorithms are characterized by learning without availability of labelled data. These algorithms belong to generative class of models and are used widely in many applications.

## 14.7 Exercises

1. In the example illustrated in the code snippet, we used only first two dimensions of the data. Repeat the experiment with remaining dimensions and see how the classification changes.
2. Sklearn library also includes an improved version of k-means clustering method called as k-means++. Use this method for classifying the Iris data and compare and contrast the results.
3. Use PCA on the Iris data to reduce the dimensionality from four to three, two, and one, and apply the supervised and unsupervised classification techniques on the reduced dimensional data, and see the differences in the classification.
4. There exists a library called *sklearn-som* [38] that implements self-organizing maps. Import and use this library in Google Colab environment and apply it to Iris data. See how it clusters the data and compare the differences with k-means clusters.

# Part III

## Building End-to-End Pipelines

**David:** Is this real or is it a game? **Joshua aka WOPR:** What's the difference?

—Dialog between David and Joshua aka WOPR, the supercomputer,  
“WarGames”

### Part Synopsis

Building a full solution of a problem using machine learning techniques needs multiple components to be tied together in a sequential chain that operate on the data one after another to finally produce the desired results. In this part, we will discuss the details of building an end-to-end machine learning pipeline using the algorithms described in the previous section.



# Chapter 15

## Featurization



### 15.1 Introduction

So far in this book, we have looked at different machine learning techniques in silo and tried to see how they identify and discriminate the patterns in the data. We did not explicitly consider training and prediction as two separate steps. In real life, the model development goes through different steps including training on labelled data and then prediction on unseen data. Building such end-to-end machine learning system is the topic of this part. Once a problem is defined that is to be solved using machine learning techniques, one needs to understand all the input data that is available. The raw input data needs to undergo some preprocessing to make it consumable through machine learning algorithms. Preprocessing is typically a custom step that can be different for different problems and is hard to generalize. After preprocessing, we need to identify the key aspects of the data that would have some impact on the desired outcome. This step is called as featurization or feature engineering. Generic preprocessing and featurization are the main focus of this chapter.

Although the human brain typically deals with non-numeric information just as comfortably if not more so than the pure numeric information, computers can only deal with numeric information. One of the fundamental aspects of feature engineering is to convert all the different features into some form of numeric features. However, as one can imagine, the process is far from trivial. In order to illustrate the process, let us consider a sample problem.

### 15.2 UCI: Adult Salary Predictor

*UCI Machine Learning Repository* is one of the well-known resources for finding sample problems in machine learning. It hosts multiple datasets targeting a variety

of different problems. We will use a problem listed there called as *Adult Salary Dataset* [6]. This dataset contains a multivariate data with features that present multiple challenges from the perspective of building end-to-end solutions. The data presented here contains information collected by census about the salaries of people from different work classes, age groups, locations, etc. The objective is to predict the salary, specifically predict a binary classification between salary greater than \$50K/year and less than \$50K/year. This is a good representative set for our needs.

In order to understand the details of each step in the featurization pipeline, here are the details of the features given in this dataset copied for reference.

### 15.2.1 Feature Details

1. *age*: continuous.
2. *workclass*: Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked.
3. *fnlwgt*: continuous.
4. *education*: Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th–8th, 12th, Masters, 1st–4th, 10th, Doctorate, 5th–6th, Preschool.
5. *education-num*: continuous.
6. *marital-status*: Married-civ-spouse, Divorced, Never-married, Separated, Widowed, Married-spouse-absent, Married-AF-spouse.
7. *occupation*: Tech-support, Craft-repair, Other-service, Sales, Exec-managerial, Prof-specialty, Handlers-cleaners, Machine-op-inspect, Adm-clerical, Farming-fishing, Transport-moving, Priv-house-serv, Protective-serv, Armed-Forces.
8. *relationship*: Wife, Own-child, Husband, Not-in-family, Other-relative, Unmarried.
9. *race*: White, Asian-Pac-Islander, Amer-Indian-Eskimo, Other, Black.
10. *sex*: Female, Male.
11. *capital-gain*: continuous.
12. *capital-loss*: continuous.
13. *hours-per-week*: continuous.
14. *native-country*: United-States, Cambodia, England, Puerto-Rico, Canada, Germany, Outlying-US(Guam-USVI-etc), India, Japan, Greece, South, China, Cuba, Iran, Honduras, Philippines, Italy, Poland, Jamaica, Vietnam, Mexico, Portugal, Ireland, France, Dominican-Republic, Laos, Ecuador, Taiwan, Haiti, Columbia, Hungary, Guatemala, Nicaragua, Scotland, Thailand, Yugoslavia, El-Salvador, Trinidad and Tobago, Peru, Hong, Holland-Netherlands.

Most of the feature names are descriptive of their meaning, but for the rest, the reader is advised to check [6]. The expected outcome is a binary classification as

1. salary > \$50K
2. salary <= \$50K

## 15.3 Identifying the Raw Data: Separating Information from Noise

In present, the problem we are given is already curated and in tabular form. Hence, we already know that we will need to use all the data that is available. This may not be true in some situations. In such cases, the rule of thumb is to use all the data that may even remotely have any influence on the outcome. However, all the data that is not related to the outcome in any way should be removed. Adding noise in the data dilutes it and affects the performance of the algorithm. Most classification algorithms can easily handle hundreds of features, but one must be careful in curating these features to have a good balance between what is needed and what is pure noise. This is not a one-time decision, and one can always experiment with different sets of features and study how they are affecting the outcomes.

### 15.3.1 Correlation and Causality

There are some techniques that can also be used to understand the relation between individual feature and the outcome. One simple technique is called as *correlation*. Quantitatively, it is computed using *correlation coefficient* or *Pearson correlation coefficient*  $\rho$  from the mathematician and statistician *Karl Pearson*. Let there be two sets of distributions  $X$  and  $Y$ . From the current context, let us say  $X$  corresponds to one of the features and  $Y$  corresponds to the outcome. Probabilistically, the coefficient is defined as

$$\rho_{X,Y} = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} \quad (15.1)$$

where  $\sigma_X$  is the standard deviation of  $X$  and  $\sigma_Y$  is the standard deviation of  $Y$  and the covariance between  $X$  and  $Y$ ,  $\text{cov}(X, Y)$ , is defined as

$$\text{cov}(X, Y) = E[(X - \mu_X)(Y - \mu_Y)] \quad (15.2)$$

$E(z)$  represents the expected value of variable  $z$  also known as population average.  $\mu_X$  and  $\mu_Y$  represent the sample means of  $X$  and  $Y$ . Using Eqs. 15.1 and 15.2, we can write

$$\rho_{X,Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y} \quad (15.3)$$

In order to numerically compute these values, let us assume there are  $n$  samples and denote individual samples from feature and outcome as  $x_i, i = 1, \dots, n$  and  $y_i, i = 1, \dots, n$ . We can expand the definitions of these expressions to arrive at

$$\rho_{X,Y} = \frac{\sum_{i=1}^n (x_i - \mu_X)(y_i - \mu_Y)}{\sqrt{\sum_{i=1}^n (x_i - \mu_X)^2} \sqrt{\sum_{i=1}^n (y_i - \mu_Y)^2}} \quad (15.4)$$

The value of correlation coefficient lies between  $[-1, 1]$ . Positive values indicate there is direct correlation between the two vectors. In other words, if the value of the feature increases, the value of the outcome also increases. Value of 0 means there is no correlation between the feature and outcome and the feature should not be used in the model. Correlation of 1 shows there is very direct correlation between feature and outcome, and one must be suspicious if we are using a feature that is part of the outcome. Typically very high correlation of 1 or a value close to it is not possible in real applications. Value of  $-1$  means very high but inverse correlation. Typically, negative correlations can be quickly converted to positive by changing the sign of the feature values. But, algorithms are smart enough to do that automatically as well.

This correlation analysis can quickly give us some insight into what features we are using and which features are likely to have more influence on the outcome. There is another caveat that must be recognized as well. This is the confusion between correlation and causality. Sometimes a completely noisy feature can show high correlation with outcome. For example, in the current context, say we have gathered the data about postal addresses of the people. Let's say one comes up with a binary feature defined as

- feature value = 1, when the street number is even
- feature value = 0, when the street number is odd

It so happens that this feature has a high correlation (0.7) with the outcome. Does that mean it is a good feature? From pure numbers, the answer would be yes, but from the *domain knowledge* of the origin of the feature, one can easily confirm that this is pure coincidence. Such coincidental features need to be carefully identified and removed from the model, as they may have adverse effect on the model performance.

## 15.4 Building Feature Set

Once we have identified the set of raw data that contains *good information* as discussed in the previous section, the next step is to convert this information into machine-usable features. The process of converting the raw data into features involves some standard options and some domain-specific options. Let us first look at the standard options.

### ***15.4.1 Standard Options of Feature Building***

The raw information that is curated for model building can be one of the following types from the perspective of computers:

1. **Numerical:** This can involve positive or negative integers, fractions, real numbers, etc. This does not include dates or some form of IDs (sometimes IDs can appear to be numeric, but they should not be considered numeric from the model standpoint, as the increment and decrement in the values of IDs have no real meaning from the perspective of the outcome). If a numeric entity is known to have only a small set of unique values, it is better to treat them as categorical rather than numeric. The rule of thumb is if the number of unique values is in few tens, it could be treated as categorical, but if that number is over a hundred, then it is better to treat them as numerical. Ideally, a feature should be treated as numerical, if the increase and decrease in the value of the feature have corresponding (direct or inverse) impact on the outcome.
2. **Categorical:** These can be string features or numeric features. String features (pure string or alphanumeric) should be treated as categorical using similar rule as stated above when the unique values it can take are in few tens. If there are more than hundred unique values, they should be treated as string features.
3. **String:** These are pure string features, where we expect to have some unique value in each sample. Typically these features would contain multiple words or even sentences in some cases.
4. **Datetime:** These would come as alphanumeric features and seem like string features, but they should be treated differently. They can have only date, only time, or datetime.

Now, let us dive deeper and see how each standard of feature should be dealt with.

#### **15.4.1.1 Numeric features**

Numeric features are the simplest to treat, as they are already in machine-understandable format. In most cases, we keep these features as is. In some cases, the following operations can be performed:

1. **Rounding:** The fractional or real-valued features can be rounded to the nearest or lower or upper integer if the added granularity of the fractional value is of no use.
2. **Quantization:** Rather than just rounding, the values can be quantized into a set of predefined buckets. This process can take the feature closer toward a categorical feature, but in many cases, this still provides a different information and can be desirable.

## Implementing numeric features

The rounding and quantization operations are basic mathematical operations, and implementation of these does not really need any machine learning-specific library.

### 15.4.1.2 Categorical Features

Treatment of categorical features is significantly different compared to the numeric features. As stated earlier, the feature is considered as categorical when the number of unique classes is in few tens. However, in some cases with large datasets, it may be appropriate to use categorical features even with hundreds of unique categories. At the heart of this treatment is a procedure called as *one-hot encoding*. One-hot encoding stands for conversion to a vector of 0s and 1s such that there is only single 1 in the entire vector. The length of the vector equals to the number of categories. Let's take the example of `[workclass]` from our example. There are *eight* different categories. Hence, each category would be encoded into a vector of length *eight*. The one-hot encoding of "Private" would be `[1, 0, 0, 0, 0, 0, 0, 0]`, of "Self-emp-not-inc" would be `[0, 1, 0, 0, 0, 0, 0, 0]`, and so on. Each value in this vector is a different feature, and the names of these features would be "workclass-Private," "workclass-Self-emp-not-inc," and so on. So now, we have expanded a single categorical feature into *eight* binary features. One can ask a question as why should we complicate this process into building so many more features, when we can just convert them into a single integer feature and assign the values from say 1–8 or 0–7 or even 00000001–10000000. The reason for not doing this way lies in the relation between two numbers. For example, number 1 is closer to number 2 than number 3 and so on. Does this relation hold for the values we are encoding? Does the value "Private" is closer to value "Self-emp-not-inc" than say "Without-pay"? The answer to these questions is *no*. All these values have a meaning of their own and cannot be compared to each other like we compare two numbers. This can only be expressed numerically by putting them in different dimensions of their own. One-hot encoding achieves precisely that.

## Implementing Categorical Features

Sklearn library provides a built-in function to implement one-hot encoding. Figure 15.1 shows how the one-hot encoding works in sklearn. Two examples are shown with two and three categories. The function can also handle missing values. However, it is important to note that the categories that are missing in the data provided in *fit* function should include all the categories one would expect in the data given to *transform* function.

```
[32] import numpy as np
      from sklearn.preprocessing import OneHotEncoder

[36] data = np.array([[ 'Red', 'Male' ],
                    [ 'Blue', 'Female' ],
                    [ 'Red', 'Male' ],
                    [ 'Yellow', 'Female' ],
                    [ 'Yellow', 'Female' ],
                    [ 'Blue', 'Male' ]])

ohenc = OneHotEncoder()
ohenc.fit(data[:,[1]])
ohenc.categories_

[array(['Female', 'Male'], dtype='<U6')]

[39] ohenc.transform(data[:,[1]]).toarray()

array([[0., 1.],
       [1., 0.],
       [0., 1.],
       [1., 0.],
       [1., 0.],
       [0., 1.]])

[40] ohenc = OneHotEncoder()
      ohenc.fit(data[:,[0]])
      ohenc.categories_

[array(['Blue', 'Red', 'Yellow'], dtype='<U6')]

ohenc.transform(data[:,[0]]).toarray()

array([[0., 1., 0.],
       [1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

Fig. 15.1 Implementation of one-hot encoder using sklearn library

### 15.4.1.3 String Features

Generic string features, also called as text features, can be processed in multiple ways. But in general before converting them into features, a set of preprocessing steps are recommended. These include:

1. Removing punctuation characters: In most cases, the punctuation characters do not carry any useful information and can be safely removed.
2. Conversion to lowercase: In most cases, converting all the text into lowercase is highly recommended. If the same word or set of words appears in multiple places with even a slight difference in casing, machine treats them as two completely separate entities. It is not desired in most cases and converting all the characters to lowercase makes the text processing more streamlined.
3. Removal of common words: Words like articles (a, an, the), conjunctions (for, but, yet, etc.), and prepositions (in, under, etc.) are used quite commonly and are typically not quite informative from the machine learning model perspective. These are also termed as *stop words* and can be safely removed. Due to high

frequency of these words, removal of these words reduces the complexity significantly. A pre-populated list of stop words is typically available in machine learning libraries, and it can be used as off-the-shelf component.

4. Spelling fixes: It is quite common to have spelling errors in the data, and misspelled words can give rise to a lot of noise. Standard spell check can be safely applied to improve the data quality.
5. Grammar: This is a complex topic in itself. In many cases, applying various grammar checks can improve the quality of data, but these modifications are not necessarily generic and should be applied on case-by-case basis. The common techniques include stemming and lemmatization (converting words like “walking” or “walked” to “walk,” etc.), tagging parts of speech (POS), etc.

The featurization of the text can be broadly classified into simple and advanced. Simple featurization can include counts and frequencies of different entities. Some examples are:

1. Number of words
2. Frequency of repeated words
3. Number of stop words
4. Number of special characters
5. Number of sentences

Advanced featurization includes:

1. N-gram analysis: N-gram analysis segments the given text into buckets of consecutive  $n$  words. Each unique sequence of  $n$  words is collected as a unique n-gram, and they are encoded based on the frequency of occurrence of each n-gram. For example, consider a sentence “Michael ran as fast as he can to catch the bus.” Assuming we applied the lowercase conversion as preprocessing step, unigram or 1-gram analysis would give “michael”, “ran”, “as”, “fast”, “he”, “can”, “to”, “catch”, “the”, “bus” unique 1-grams. The encoded vector will be [1, 1, 2, 1, 1, 1, 1, 1, 1, 1]. Bigram or 2-gram analysis of the same sentence will generate the following unique 2-grams “michael ran”, “ran as”, “as fast”, “fast as”, “as he”, “he can”, “can to”, “to catch”, “catch the”, “the bus”. The encoded vector will be [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]. As you can see the difference between 1-grams and 2-grams, the word “as” appears only once in the feature space of 1-grams, but in the bigrams, “as fast” and “fast as” appear separately in 2-gram feature space. In cases like this, elimination of stop words (such as “as”) can reduce the feature space drastically.

One more thing to notice here is that n-gram analysis with  $n > 1$  captures the sequential information of the words and not just their frequency. Especially when  $n$  gets larger, the uniqueness of features increases exponentially. Also, if  $n$  is too large, the space gets too sparse, so one needs to carefully do the trade-off between higher value of  $n$  and lower value.

2. Bag of words: Bag of words is typically described as the most popular method of featurizing string entities. It essentially represents the n-gram analysis with  $n = 1$ .



3. TF-IDF analysis: It stands for term frequency-inverse document frequency. This analysis is typically useful in case of long texts. For shorter texts, n-grams or bag of words is typically sufficient. Conceptually, it quantifies the importance of a word in a given text compared to the collection of texts. Let us look at the mathematical definition of this.

Let the term frequency (TF) of a word  $w_j$  in a text  $t_j$  be given as  $f_w(w_i, t_j)$ . This is calculated as the number of times the word  $w_i$  appears in text  $t_j$ . Let there be a total of  $n$  words over the entire corpus of texts. Let there be  $m$  number of texts. The inverse document frequency (IDF)  $f_{id}(w_i)$  of the word  $w_i$  is defined as

$$f_{id}(w_i) = \log \frac{n}{\sum_{j=1}^m \left( \begin{cases} 1, & \text{if } w_i \in t_j \\ 0, & \text{otherwise.} \end{cases} \right)} \quad (15.5)$$

The denominator essentially counts the number of texts that contain the word  $w_i$ . The joint TF-IDF expression is then the combination of TF and IDF as  $f_w(w_i, t_j) f_{id}(w_i)$ .

To illustrate the concept, the stop words will have a high TF in all the documents, but low IDF overall, and as a result, their TF score will be discounted accordingly. However, if a certain word appears only in one document, its TF in that document will be significantly boosted by IDF.

## Implementing Text Featurization

Most of the techniques discussed above are readily available from sklearn library on text featurization [39]. One can follow the steps outlined in Fig. 15.1 to try these methods on a sample text.

### 15.4.1.4 Datetime Features

Datetime features expose a very different type of information, and it needs a little bit of domain knowledge to make sense. That way, they belong more in the next section of custom features. However, they also allow for some standard processing and hence are discussed here. In string format, the datetime feature is quite meaningless, and at the same time, converting datetime directly to a numeric value also brings little to the table. Commonly extracted datetime features include

1. Day of week
2. Day of month
3. Day of year
4. Week of month
5. Week of year

6. Month of year
7. Quarter of year
8. Days from today
9. Hour of day
10. Minute of hour

### Implementing Datetime Features

The datetime library available in vanilla Python has most of the functions listed above readily available for use [40], and one can easily derive some more advanced features by using a combination of those.

#### ***15.4.2 Custom Options of Feature Building***

Most of the featurization options discussed in the previous section are standard in the sense that they are valid irrespective of the context or in absence of any domain knowledge. When we are building a custom machine model to solve a specific problem in artificial intelligence, we already have a domain knowledge that can be applied to extract features that may not make sense in generic settings. For example, let us consider the example of salary prediction. One of the numeric features is “age.” It is a numeric and continuous feature, and as per standard featurization guideline, we can use it as is or we can round it to integer or even bin it to multiples of 5, etc. However, for current problem of salary, this age is not just a number between say 0 and 100, but it has more to it. We know that the typical starting age for employment is say around 18 and typical retirement age is say around 65. Then we can add custom bins within that range to create a more meaningful feature in current context.

Another important aspect of custom feature building is in the form of interactions between two features. This is an entirely non-trivial way of extracting information from generic standpoint and can provide unique information that is otherwise not available. For example, in the case of adult salary problem, there are two features “Capital-gain” and “Capital-loss.” We can join these two features in the form of difference as “(Capital-gain) – (Capital-loss)” to see the big picture in single feature. Or we have some survey information that gives a specific relationship between the “age” and “education-num,” which we can use to join these two features in some proportion to create a more informative new feature. These features are non-trivial and would certainly add to the information that the model can use.

Datetime features are also strong candidates for creating custom features. Difference of days between two dates is always a useful feature. Another way to join features is applying statistical operators like mean, min, max, etc. on collection of numeric features. The options are practically unlimited. However, it must be noted that creating large numbers of custom features does not necessarily improve the

performance arbitrarily and there would soon be a point of diminishing returns. Nevertheless, this is always a good option to experiment.

## 15.5 Handling Missing Values

Missing values are another rather common occurrence in real-life situations, and they must be substituted with some values, or the models will fail to execute. If the missing values are present in the training data, trivial solution is to just ignore these samples with missing values or ignore the features that have missing values in some samples. However, in some cases, there are just too many samples or too many features with some missing information, and we cannot just ignore them. Also, if the values are missing in test data, we have no choice but to substitute the missing values with some reasonable estimates. The most common ways to handle the missing values are:

1. Use the mean or mode or median value in case of numeric values.
2. Use the mode in case of categorical values.
3. If we have some prior knowledge, we can always replace the missing value with a predetermined value.
4. In case of categorical features, we can create a new category as “unknown” and use that instead. However, there is one caveat here, and that is if the “unknown” category pops only in the test data and is not available in the training data, some models can fail. However, if there are samples in training data with missing values that are also substituted with “unknown,” then it is typically the most elegant solution.
5. Predicting the missing value. This can be achieved by a variety of ways, ranging from simple regression-based methods to more complicated *expectation-maximization* or *EM* algorithms [41]. If the feature that is being predicted has a very high variance, then this method can result in bad estimates, but in general, this is better suited than simple “mean/mode/median” substitution.
6. Use the algorithms that support missing values. There are certain algorithms that inherently use the features independently, e.g., naive Bayes or K-nearest neighbor clustering. Such algorithms can work with missing values without needing to substitute them with estimates.

## 15.6 Visualizing the Features

The last step in featurization is visualizing the features and their relationship with the labels. Looking at the visualizations, one can go back to some feature modifications as well and iterate over the process. To illustrate this, let us use the adult salary data.

### 15.6.1 Numeric Features

First let us look at the numeric features. The numeric features are:

1. *age*
2. *fnlwgt*
3. *education-num*
4. *capital-gain*
5. *capital-loss*
6. *hours-per-week*

Now, let us look at the plots of individual feature values compared to the label as shown in Figs. 15.2, 15.3, and 15.4.

Table 15.1 shows the correlation coefficients of each of the features with the label.

As we can see from the figures and the correlation coefficients, feature *fnlwgt* contains almost no information and most other features a relatively weak features on their own. As discussed before, the negative sign only shows the relationship between their values and label is inverse. Looking at this data, we can safely ignore the feature *fnlwgt* for model training purpose. However, considering that there are only 14 features to start with, we can keep it.

### 15.6.2 Categorical Features

Now, we will look at the categorical features one by one. As described before, each value of a categorical feature is treated as separate feature by using one-hot encoding technique.

#### 15.6.2.1 Feature: *Workclass*

Feature *workclass* is a categorical feature with possible values in {Private, Self-emp-not-inc, Self-emp-inc, Federal-gov, Local-gov, State-gov, Without-pay, Never-worked}. It would be much harder to visualize the effect of all the different values of the features in a single plot the way we did in case of numeric features. Also, we don't want to create a separate plot for each value, as that would lead to just too many plots. Instead, we will use a technique called pivot table or pivot chart. In this technique, we put all the values of one parameter on one axis and count or any other aggregate function (sum, max, min, median, mean, etc.) of the values of the other parameters on the other axis. Figures 15.5 and 15.6 show the pivot charts for all the values of this feature in case of salary  $>50K$  and salary  $\leq 50K$ .

As can be seen from the charts, the value of *Private* has similar distribution in both cases, but the values *Federal-gov* and *self-emp-inc* show significantly different

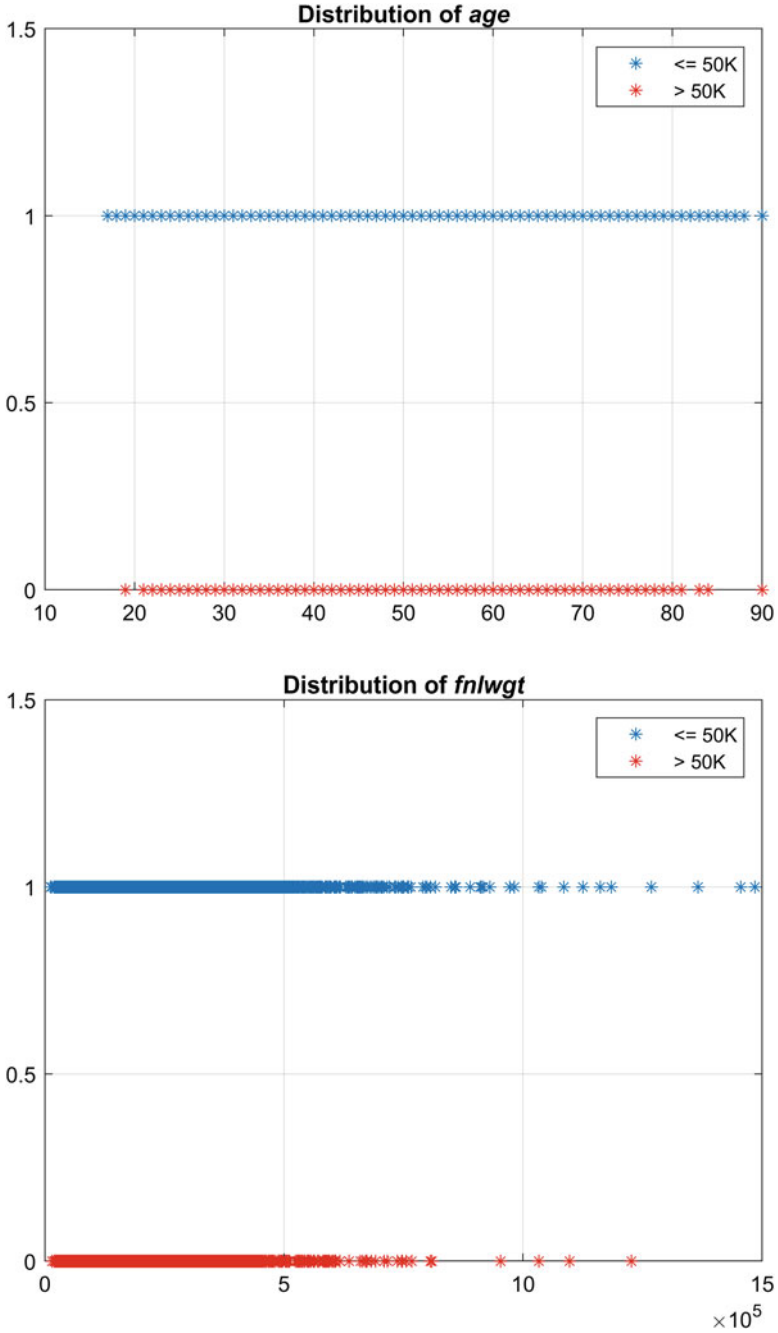


Fig. 15.2 Showing class separation using individual features *age* and *fnlwgt*

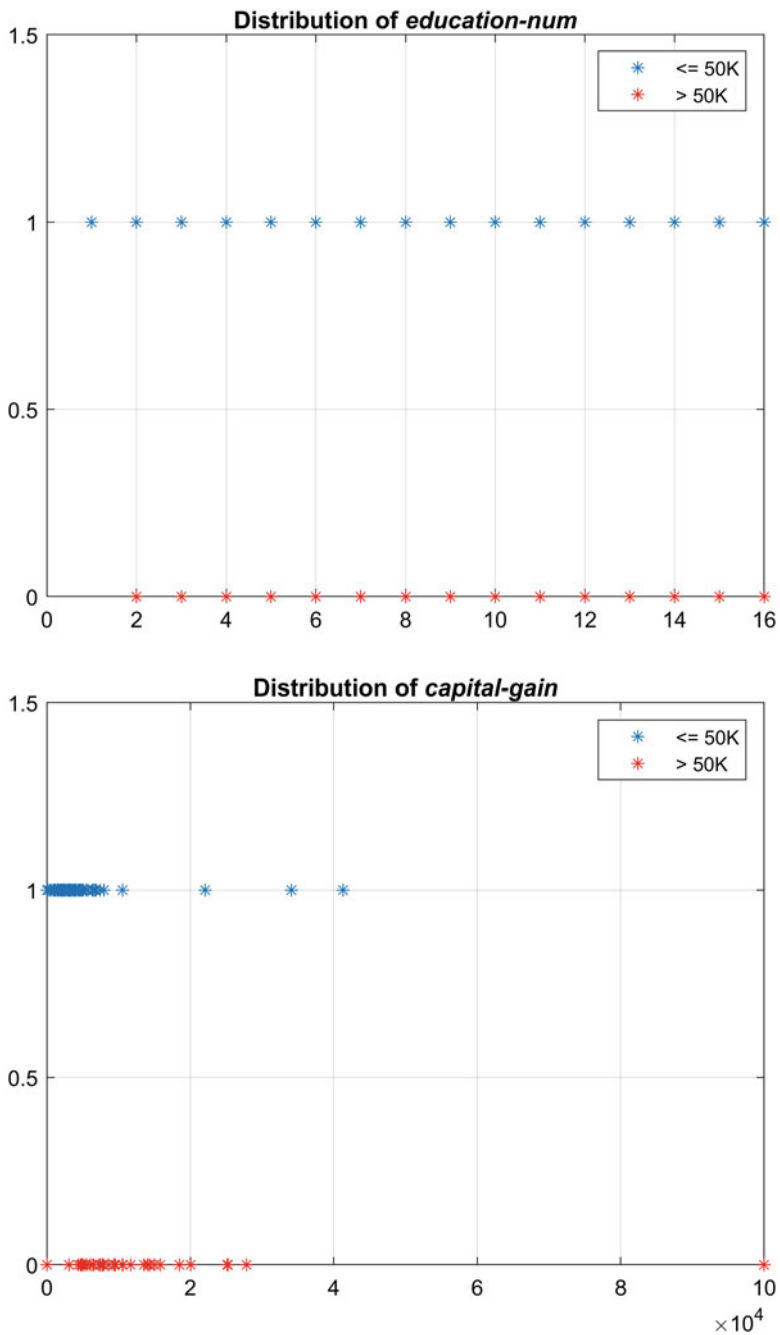


Fig. 15.3 Showing class separation using individual features *education-num* and *capital-gain*

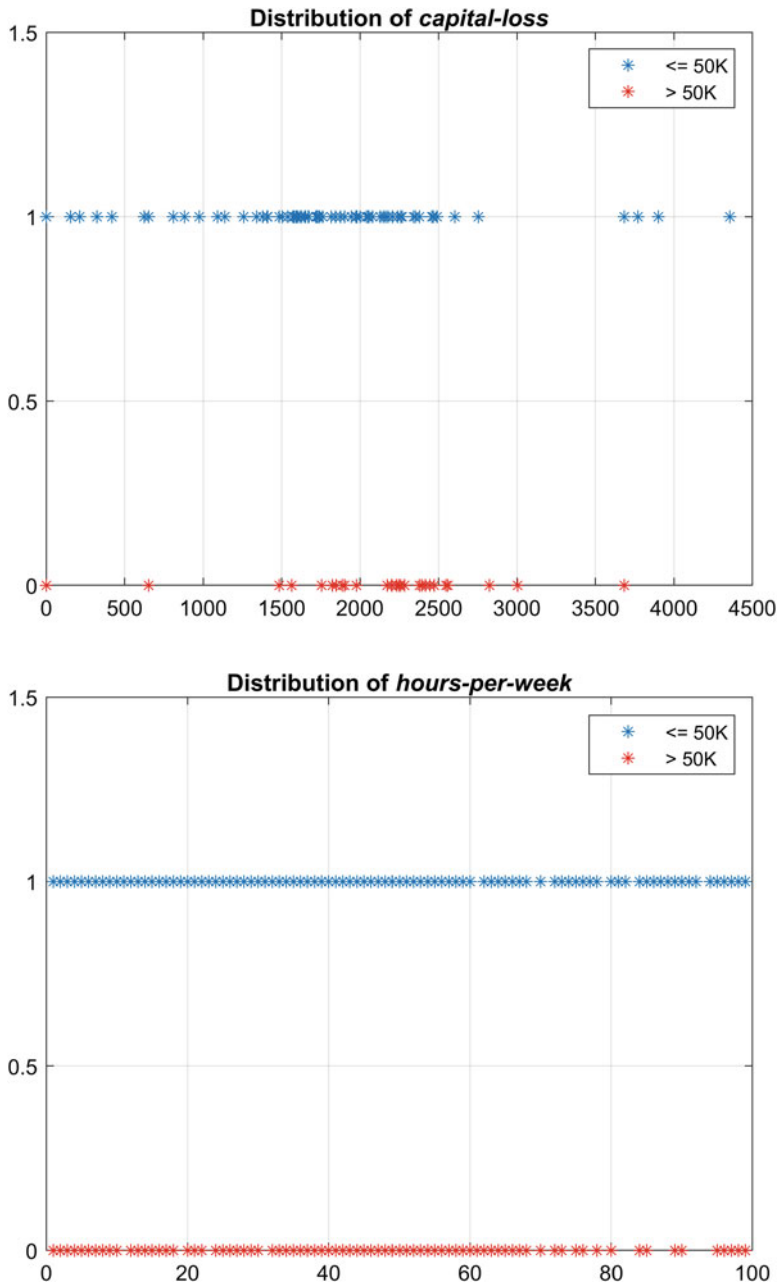
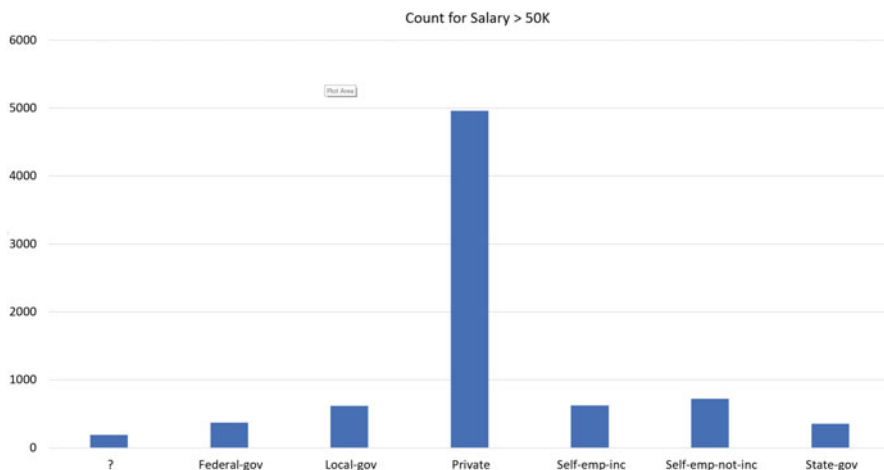


Fig. 15.4 Showing class separation using individual features *capital-loss* and *hours-per-week*

**Table 15.1** Correlation coefficients between numeric features and label

Feature name	Correlation coefficient
<i>age</i>	-0.23
<i>fnlwgt</i>	+0.01
<i>education-num</i>	-0.34
<i>capital-gain</i>	-0.22
<i>capital-loss</i>	-0.15
<i>hours-per-week</i>	-0.23



**Fig. 15.5** Pivot table showing distribution of the label (>50K) for each value of the feature. ? means missing value

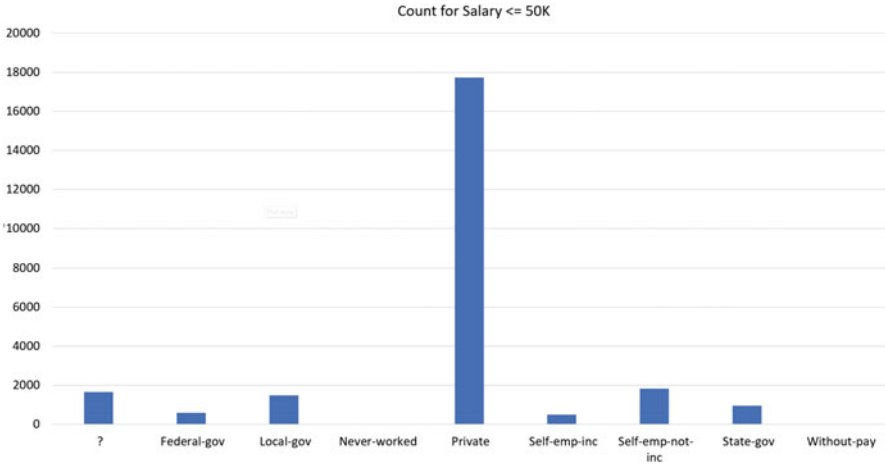
distribution in both cases. Thus, we expect that the separate features created using these values should be more influential in the model.

### 15.6.2.2 Feature: *Education*

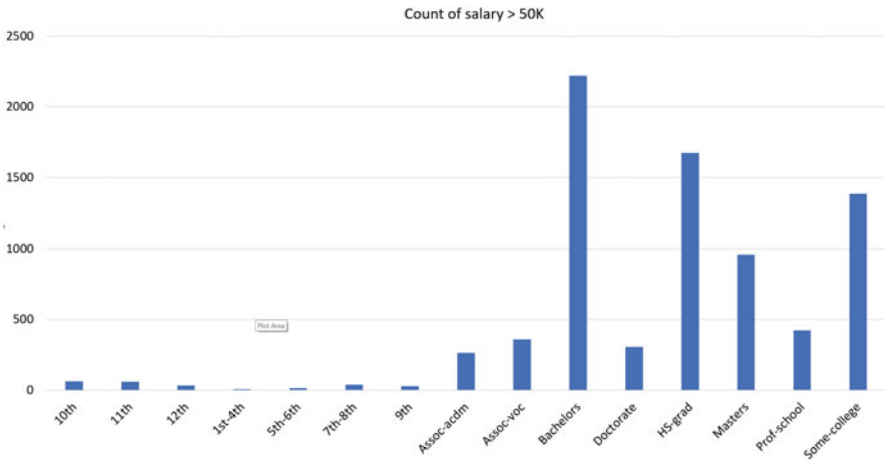
Feature *education* is a categorical feature with values in {Bachelors, Some-college, 11th, HS-grad, Prof-school, Assoc-acdm, Assoc-voc, 9th, 7th-8th, 12th, Masters, 1st-4th, 10th, Doctorate, 5th-6th, Preschool}. We will use the pivot chart as before to see the trends between the value of this feature and label. Figures 15.7 and 15.8 show the pivot charts.

Some features are altogether missing between the plots as a result of having zero count. These values would be expected to have a very high influence on the outcome. Also the values corresponding to higher education show strong separation between the two classes.





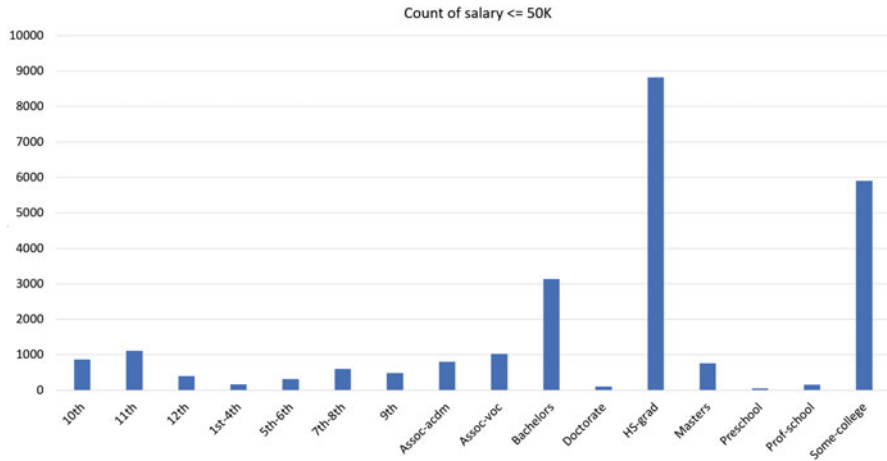
**Fig. 15.6** Pivot table showing distribution of the label ( $\leq 50K$ ) for each value of the feature. ? means missing value



**Fig. 15.7** Pivot table showing distribution of the label ( $>50K$ ) for each value of the feature. ? means missing value

### 15.6.2.3 Other Features

The remaining categorical features *marital-status*, *occupation*, *relationship*, *race*, *sex*, and *native-country* can be analyzed in a similar fashion. The insights gained by looking at the trends visually can be extremely effective in choosing the right mode, tuning it, selecting the constraints for regularization, etc.



**Fig. 15.8** Pivot table showing distribution of the label ( $\leq 50K$ ) for each value of the feature. ? means missing value

## 15.7 Conclusion

This chapter described the concept of featurization in a detailed manner. We started with generic description of the techniques, and to make them real, we took an example of the adult salary data and performed the featurization of it by taking each individual feature one at a time. We also used the help of visualization to draw some inferences on the influencing capabilities of each feature on the outcome.

## 15.8 Exercises

1. Apply all the featurization techniques learned in this chapter on the UCI adult salary prediction data in Google Colab.
2. Plot the features using suitable plots, and experiment with variations of the features, and see the impact of them in visual representations and correlation with the salary.
3. Try to build custom features from the raw features, and visualize them and study their correlation with salary.
4. Use suitable regression algorithm, and evaluate the performance on predicting salary with your featurization methods.

# Chapter 16

## Designing and Tuning Model Pipelines



### 16.1 Introduction

Once the features are prepared as we discussed in the previous chapter, the next step is to choose the technique or algorithm to use to solve the problem at hand. So far, we have been using all the data for training and using the same data to see fit. However, in real life, the data used in prediction is always unseen by the model. In order to simulate this behavior, we will learn the concept of splitting the data into two or three parts: training, validation (can be optional), and testing. The training set is used for training the model, optional validation set is used to tune the parameters, and test set is used to predict the performance metrics of the algorithm that we would expect when it would be applied to the real data. We will learn about these steps in details in the following sections.

### 16.2 Choosing the Technique or Algorithm

Typically the application dictates the broad category of method we should use. For example, if we are dealing with a problem of predicting continuously varying values like some form of score or rate, then we are restricted to all the regression-type techniques, which would eliminate the clustering or recommendation algorithms,<sup>1</sup> etc. Alternatively, if we are dealing with problem to identify some categories or

---

<sup>1</sup> The terms *model* and *algorithm* are used interchangeably in most machine learning literature, and it can be confusing. The precise definition of these two terms is algorithm is the underlying theoretical infrastructure and model is a binary structure that is obtained when an algorithm goes through the training process. In other words, model is a trained algorithm. With different training data, we get a different model, even with the same algorithm. The trained model can then be used to predict outcomes on the unlabelled data.

yes/no-type outcomes, then we have to use classification algorithms like clustering, decision trees, etc.

In order to illustrate the thought process in choosing the model, let's consider the problem of binary classification of the adult salary data as described in the previous chapter.

### ***16.2.1 Choosing Technique for Adult Salary Classification***

As the application dictates, we have to narrow our scope to classification algorithms. Regression algorithms can also be used in classification applications sometimes when there is some notion of continuous valued function. In such cases, a suitable threshold(s) can be applied in the end to separate the categories. One can always experiment with different algorithms to see which one gives better results. We already discussed about the types of field given to us in earlier chapter. Here is a snapshot of actual data for the first few samples (Table 16.1).

As can be seen here, the data types we are dealing with here are either continuous valued integers or string categorical. There are also some missing features represented with “?”.

Decision tree-based methods are typically better suited for problems that have categorical features as these algorithms can implicitly use the categorical information without the need for encoding them. Most other algorithms like logistic regression, support vector machines, neural networks, or probabilistic approaches work better with pure numerical features. This comment should be treated as a rule of thumb, and exceptions to this rule may exist from time to time. Another advantage of decision tree is the ability to understand its inner behavior. Most other algorithms appear as black box and are very difficult to interpret. After experimenting with decision tree, we can try with more complex algorithms and compare the performance with baseline established by the decision tree model. Complex algorithms are typically the ones that need elaborate setup and initialization of a large number of parameters, also called as hyperparameters. We will deal with this topic later in this chapter.

Single decision tree can be used as one of the simplest possible starting algorithms. However, ensemble algorithms like random forest are highly recommended as they offer much more robust and significantly better performance without adding too much complexity.

## **16.3 Splitting the Data**

Once an algorithm is selected for use, the next step is to separate the whole labelled training data into two or three sets, called as training, (optional) validation, and test. In this case, as we plan to do hyperparameter tuning, we will divide the data

**Table 16.1** Sample rows from UCI adult salary data

Age	Workclass	fnlwgt	Education	Education-num	Marital-status	Occupation	Relationship	Race	Sex	Capital-gain	Capital-loss	Hours-per-week	Native-country	Label
38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K
37	Private	284582	Masters	14	Married-civ-spouse	Exec-managerial	Wife	White	Female	0	0	40	United-States	<=50K
49	Private	160187	9th	5	Married-spouse-absent	Other-service	Not-in-family	Black	Female	0	0	16	Jamaica	<=50K
19	Private	168294	HS-grad	9	Never-married	Craft-repair	Own-child	White	Male	0	0	40	United-States	<=50K
54	?	180211	Some-college	10	Married-civ-spouse	?	Husband	Asian-Pac-Islander	Male	0	0	60	South	>50K
39	Private	367260	HS-grad	9	Divorced	Exec-managerial	Not-in-family	White	Male	0	0	80	United-States	<=50K
49	Private	193366	HS-grad	9	Married-civ-spouse	Craft-repair	Husband	White	Male	0	0	40	United-States	<=50K

into three sets. Typically, the division of the three sets in percentage is done as 60-20-20 or 70-15-15. The more samples available for training, the better, but we need to also keep sufficient samples in validation and test sets to have statistically significant metrics. This aspect is discussed in greater detail in Chap. 17. Typically, the validation and test sets are of the same size. But all these decisions are empirical, and one can choose to customize them as needed.

### 16.3.1 *Stratified Sampling*

Splitting data into two or three sets should always be done in random fashion and not sequential manner to have statistically uniform spread of data between all the three sets. There is one more aspect that needs careful attention, specifically in case of classification applications, and that is to ensure uniform distribution across different classes between the training, validation, and test sets. If pure random distribution puts all the samples from a class into training data and validation or test sets do not contain any samples from that class, we cannot accurately generate metrics for the classifier model. This issue is addressed using a technique called as *stratified sampling*. Stratified sampling ensures a roughly uniform distribution of classes in split sets. When we are dealing with say  $n$  number of classes, the number of samples per class is not always identical. Sometimes there can be a large skew in this distribution, meaning the number of samples of one class can be significantly larger than the others. In such cases, the uniform random split does not yield optimal separation, creating bias toward the class with more samples. In order to train the model without bias toward any specific class, we need to have roughly the same number of samples from each class in the training set. The distribution in validation and test set is less important, but it is always a good practice to have roughly balanced distribution across the classes in all the three sets. When the original distribution is unbalanced, there are two choices:

1. Ignore a certain number of samples from the classes that have more samples to match the classes that have less number of samples.
2. Repeat the samples from the classes that have less number of samples certain number of times to match with the number of samples from the classes that have larger number of samples.

Which option to choose depends on the original number of samples. If there are sufficient number of samples even after dropping samples from larger sets, then that is always a better option rather than repeated sampling.

Also, it is not mandatory to use stratified sampling, if the bias between classes is acceptable. In the current problem of adult salary classification, they have given the training and test data in separate sets already. There are 24,720 samples from the class  $\leq 50K$  and only 7841 samples from the class  $> 50K$  in training data that contains a total of 32,561 samples. The test set contains 12,435 samples from the class  $\leq 50K$  and only 3846 samples from the class  $> 50K$  with a total of 16,281

samples. Thus, there is significant skew in the sample distribution. However, as per the guidance given by the creators of the dataset, the bias between these two classes of salary ranges is known and needs to be preserved. Uniform random sampling typically gives similar distribution between the classes in different parts when data is large enough. If one finds that there is significant difference, either the sampling process can be repeated or one can use stratified sampling to enforce certain distribution of classes in each part. In the current problem, we just need to split the training set into two parts—training and validation. We will use 70–30 split for this purpose.

## 16.4 Training

Once the data is split into training and test sets, training set is used to train the algorithm and build the model. Each algorithm has its own specific method associated with it for training. The underlying concept common in all the different techniques is to find the right set of parameters for the model that can map the input to output as accurately as possible for the given labelled training data. In other words, the objective is to find the set of parameters that minimize the error (one can choose different types of errors as defined in Chap. 17) between the predictions and the expected values of the output in training set. Some methods are iterative, and they go through multiple loops of finding and improving the parameters. Some methods are one shot, where a one-time operation of optimization produces the optimal set of parameters.

The parameters of the model are also classified into two main types:

1. The parameters that can be computed using the training process to minimize the error in the prediction.
2. The parameters that need to be predetermined before starting the training process. These parameters are also called as hyperparameters. One can run the entire training multiple times to optimally select the hyperparameters through a process called as hyperparameter tuning or optimization. A third set of data called validation set is required for this process. The model with each unique set of hyperparameters is an entirely different model.

### 16.4.1 Tuning the Hyperparameters

The hyperparameters are typically a set of parameters that can be unbounded, e.g., the number of nodes in a hidden layer of a neural network or the number of trees used in random forest. One can theoretically choose any number between 1 and  $\infty$ . There is no direct formula that dictates the optimal number of nodes to be used in a neural network based on training data. Hence, the bounds on hyperparameters

are created by the data scientist, who is in charge of building the model. These bounds are created based on multiple constraints like computation requirements, dimensionality and size of data, etc. Typically one needs to create such bounds for more than one hyperparameter for each model. Thus, we end up with an  $n$ -dimensional grid of hyperparameters to choose.

As stated in the previous section, a single set of training set can be used to get results with one set of hyperparameters. Hence, in order to have multiple such sets to tune hyperparameters, one needs to split the training data further into two parts called as training and validation. There are multiple techniques described in the literature for generating these sets. However, the test set must be kept separate and never be used in the tuning or training processes. If one uses the test set in this process, then all the labelled data would be used for the training-tuning process, and no blind set of data would be left to predict the behavior of the trained-tuned model on unseen samples. Preserving the sanctity of blind test set is a fundamental requirement in machine learning theory that must never be broken.

For a given set of hyperparameters, the training set is the only data available for training, and the trained model is applied on the validation set to compute the accuracy metrics. Once all the different sets of hyperparameters are used, the set that provides the best performance on the validation set is used as the best model. Then the test data is used only once to predict the accuracy of the tuned and trained machine learning model.

## 16.4.2 Cross-Validation

## 16.5 Implementing Machine Learning Pipeline

Now, we will use the adult salary data to implement the entire pipeline for classification. Here we will integrate all the concepts that we have learned in this book for the first time.

Figure 16.1 shows the first step with importing all the libraries and reading the data in CSV format. Sklearn does not support this data natively, so we will have to download the data from [6] and upload it to our Google Colab workspace. All the names for the columns also need to be explicitly specified. We also add the names of columns containing numeric features and categorical features explicitly, as it will come handy in the subsequent steps.

The next step is featurization or preprocessing. We apply simple scaling on numeric features to normalize them. This is an optional step and can be skipped without much impact. However, applying uniform processing to all the feature types makes the overall pipeline design more streamlined. We also use the one-hot encoder as described earlier to encode the categorical features. We use the column transformer architecture in sklearn to combine all the preprocessing steps in to a single processing unit called *preprocessor* (Fig. 16.2).



### ▼ Import the libraries

```
[163] import pandas as pd
      from sklearn.model_selection import train_test_split
      from sklearn.pipeline import Pipeline
      from sklearn.compose import ColumnTransformer
      from sklearn.ensemble import RandomForestClassifier
      from sklearn.preprocessing import OneHotEncoder, StandardScaler
```

### ▼ Read the data and split into input - output

```
[179] df = pd.read_csv('adult.data', header=None, names=['age', 'workclass', 'fnlwgt', 'education', 'education-num',
      'marital-status', 'occupation', 'relationship', 'race', 'sex',
      'capital-gain', 'capital-loss', 'hours-per-week',
      'native-country', 'salary'])

all_features = ['age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week',
               'workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-country']
num_features = ['age', 'fnlwgt', 'education-num', 'capital-gain', 'capital-loss', 'hours-per-week']
cat_features = ['workclass', 'education', 'marital-status', 'occupation', 'relationship', 'race', 'sex', 'native-country']
X = df[all_features]
y = df['salary']
```

**Fig. 16.1** Importing all the necessary libraries and reading the data in CSV format

### ▼ Featurization

Categorical features need to be encoded before using into model training

```
[181] num_transformer = StandardScaler()
      cat_transformer = OneHotEncoder(handle_unknown="ignore")

      preprocessor = ColumnTransformer(transformers=[("num", num_transformer, num_features),
      ("cat", cat_transformer, cat_features),])
```

**Fig. 16.2** Creating the preprocessing or featurization pipeline using sklearn ColumnTransformer architecture

### ▼ Creating classification pipeline

```
[182] clf_pipeline = Pipeline(steps=[("prep", preprocessor),
      ("classify", RandomForestClassifier(n_estimators=100))])
```

**Fig. 16.3** Building the entire classification pipeline and integrating the preprocessing step into it

The next step is to create the classification component of the pipeline. This is the heart of the processing where we will use the random forest algorithm and train it with the training data. We also combine the preprocessing step with the classification step to build the end-to-end pipeline. This is a preferred method over doing these steps in silos. This setup makes it easy to apply the same preprocessing steps in training as well as scoring phases. We initialize the random forest classifier with 100 estimators or weak learners (Fig. 16.3).

The next step is to actually perform training. We first separate the data into training and test sets with 70–30 split ratio. Then we apply the entire classification

### ▾ Applying the pipeline to train the model

Splitting the data into training and test sets

Train the model on training data

```
[183] X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
271
clf_pipeline.fit(X_train, y_train)
Pipeline(steps=[('prep',
                 ColumnTransformer(transformers=[('num', StandardScaler(),
                                                ['age', 'fnlgt',
                                                 'education-num',
                                                 'capital-gain',
                                                 'capital-loss',
                                                 'hours-per-week']),
                                                ('cat',
                                                 OneHotEncoder(handle_unknown='ignore'),
                                                 ['workclass', 'education',
                                                  'marital-status',
                                                  'occupation', 'relationship',
                                                  'race', 'sex',
                                                  'native-country'])])),
                 ('classify', RandomForestClassifier())])
```

**Fig. 16.4** Splitting the data into training and test sets with 70–30 ratio and applying the classification pipeline on training set

### ▾ Scoring the model on test set

```
[184] print("Model score on test set: %.4f" % clf_pipeline.score(X_test, y_test))
271
Model score on test set: 0.8503
```

**Fig. 16.5** Apply the trained model on test set and compute the metrics

### ▾ Scoring the model on test set

```
[185] print("Model score on test set: %.4f" % clf_pipeline.score(X_test, y_test))
271
Model score on test set: 0.8495
```

**Fig. 16.6** Apply the trained model on test set and compute the metrics using a different random split. Notice the difference in the accuracy

pipeline built in the previous step on the training set. This gives a trained model (Fig. 16.4).

Now we apply the trained model on the test set that is blind to the trained model and generate accuracy metrics. It is important to note that current metrics are also dependent on the random split that has happened, and we may get different metrics if we run the same pipeline again (Fig. 16.5).

In order to illustrate the effect of random sampling on the trained model, we run the pipeline again with a different random split and see a slightly different accuracy (Fig. 16.6).

## 16.6 Accuracy Measurement

Measuring the accuracy of a model is the last step in the design of machine learning system. However, considering the scope of this topic, the entire next chapter is devoted for this step.

## 16.7 Explainability of Features

Typically, once a model is trained with sufficient accuracy, the job of the data scientist is done. The trained model is available as a black box that produces an output when an input is presented to it with statistical assurance that the output will be accurate within certain bounds. However, in recent times, an additional step of explaining *why a certain output is predicted for a given input* is getting more and more importance. This step does not find roots in theory of machine learning but has emerged more due to the collision of the traditional heuristic methods with machine-learned methods. When a domain expert builds a simpler heuristic model to predict certain outcomes, all the rules that are part of the heuristic model are human readable and have obvious interpretability. When the outcome is high or low, the reason for that specific outcome can quickly be seen from the rules and interpreted. These interpretations also lead to concrete actions that can be taken based on them.

Most of the machine-learned models, and especially the neural network-type models, lack this reasoning and interpretability completely. One must accept the outcome as it stands based on the accuracy metrics that are provided based on the test data. However, this lack of reasoning or explainability and interpretability came into strong criticism when the machine-learned models started to replace the older heuristic models. There are variety of different techniques proposed that can be applied after the model is the trained to add the explainability to the models. Another way to explain the model is based on importances of the features that are used. The core idea here is to vary the value of individual features within reasonable predefined ranges and see their effect on the outcomes. The features that have stronger impact on the outcome are more important and vice versa. However, this technique does not take into account interdependency of the features. This article [18] discusses some of the current advances in this field. Also, the explainability of the features needs to be built for aggregate level as well as case-by-case level. The models whose outcomes are explainable in such manner are called as interpretable models and are preferred over black box models even if they produce lesser accuracy in many cases.

## 16.8 Practical Considerations

All the machine learning algorithms discussed so far in the book are based on certain assumptions about the nature of the data. Either the data is static or it is in the form of time series. Either the data is strictly linear or it can be converted into linear with suitable link function or it is purely nonlinear. The number of classes defined in the training data cannot be different from the ones in test data and so on. However, when we deal with real data in practice most of the times, none of these assumptions are precisely applicable. There are always some gray areas, and it is rather crucial to identify such areas and address them explicitly. We will discuss some of the more commonly observed situations in this section, but this list should not be treated as comprehensive.

### 16.8.1 *Data Leakage*

In selecting the underlying algorithm for solving a given machine learning problem, one must understand the trends in the data. If the data is static, then there is a set of static algorithms that one can choose from. Examples of static problem could be classification of images into ones that contain cars and ones that don't. In this case, the images that are already gathered for the classification are not going to change over time, which is purely a case of static analysis. However, if the data is changing over time, it creates additional layer of complexity. If one is interested in modeling the trends of change in data over time, it becomes a time series type of problem, and one must choose appropriate model, e.g., ARIMA, as described in Chap. 12. Stock price prediction is one example of time series model. However, in some cases, we are dealing with data that is changing, but we are only interested in the behavior of the data for a given snapshot in time. This situation does not warrant a time series model, and once a snapshot of the data is taken, it becomes a strictly static problem. However, when such snapshot of the data is taken for building a machine learning model, one must be careful about the timeline of changes of all the individual features till the time of snapshot. If the causality of such data is not taken into consideration, one creates a problem of data leakage. This can lead to significant degradation of the model.

An example will help understand this phenomenon better. Consider a business of auto mechanic. We are trying to predict the customers who are likely to visit more than once in a period of say 6 months. We gathered the sales data for the past 6 months. Here are some of the columns in the data:

1. Number of visits
2. Year of manufacture of the car
3. Make of the car
4. Model of the car
5. Miles on the odometer

6. Amount of sale
7. Method of payment
8. Category of customer

The first column is actually the label of the data, where if the number of visits is greater than 2, we classify it as “True” and classify as “False” otherwise as per the definition of the problem. This column is then removed from the feature space, and all the remaining columns are used as features. However, the last column in the list is a categorical feature where we classify each customer based on the historical data. We put only the customers that are repeat customers into “Known” category, and the rest of them are put into “Unknown” category. Now, this column is not exactly the same as the label column; however, the “Known” customers are more likely to be classified as “True” as per our definition compared to the “Unknown” customers. Also, it must be noted that the value in this column is going to be updated after each visit. Thus, if we use this column as one of the features, it is going to leak partial information from the label back into the feature space. This will cause the model to perform much better than expected based on this single feature alone. Also, it must be noted that this feature will not be available for the customers when they are going to visit the first time. Hence, it would be best to not use this feature in training the model. This leakage of information is also called as *target leaking*.

Many times, these *leaky* features may not be readily identifiable. In such cases, one must try to do correlation analysis of each feature with the outcome. If one more features stand out with high levels of correlation (the level threshold is relative and one must use his/her judgment to find it) with output, they must be ignored in the training process.

### ***16.8.2 Coincidence and Causality***

In practice when dealing with data with a large number of features for predicting relatively small number of classes or predicting a relatively simple regression function, the noise levels in the training data can significantly affect the model quality. When there are too many features present in the data that have no causal effect on the outcome, there is a chance that some of these noisy features may show *lucky* correlation with the outcome. Identifying such noisy features is typically done with the domain knowledge of the problem space, but sometimes the scientist developing the model may lack this knowledge. In such cases, it becomes extremely difficult to identify the features that are coincidentally correlated with the outcome versus the features that actually have causal effect on the outcome. The coincidental features might help in getting high accuracy on training data, but will make the model quite weak in predicting the outcome on the unseen test data. In other words, the generalization performance of the model can be quite poor due to such features.

Unfortunately, there exists no theoretical method to identify and separate the coincidental features from the causal features purely from the data analysis. There

are some probabilistic methods proposed based on conditional dependence between features, but all such methods make some assumptions on the dependency and causality, and these assumptions may not hold true on all the cases of real data. This is a relatively novel aspect of machine learning and is under active study. Here is one article that discusses some aspects of this phenomenon [17]. Although this is a problem with no concrete theoretical solution, one can circumvent this situation by taking the following measures:

1. Using cross-validation even if hyperparameter tuning is not used. This splits the data in multiple different combinations, thereby reducing the chance of coincidental features getting more importance.
2. Using ensemble methods as opposed to single-model methods to improve the robustness and reduce coincidence.
3. Applying feature interpretation as an additional step after the model training to make sure all the important features have domain-specific explainability. Having such explainability makes the models more resilient of coincidences.

## 16.9 Conclusion

In this chapter, we integrated the concepts learnt so far with respect to various algorithms and data preprocessing and discussed the elements of designing an end-to-end machine learning pipeline. We also discussed about the lesser known aspects in this design like data leakage and coincidence against causality and how to address those to successfully build the system.

## 16.10 Exercises

1. Take another dataset with a regression problem, and build a full machine learning pipeline to build model and test accuracy.
2. Apply cross-validation and see how much improvement can be achieved.
3. Use a different algorithm and apply the same steps and compare the results.

# Chapter 17

## Performance Measurement



### 17.1 Introduction

Any discussion about machine learning techniques cannot be complete without the understanding of performance measurement. In general, performance can be measured qualitatively by subjective comparison between a set of results and reference or quantitatively by mathematically comparing the values of results with reference. Subjective inferences are always affected by the opinions, interpretations, and emotions of the person making the judgments and create a bias in the inference and as a result are not generalizable. In science and especially in case of machine learning, objective methods are the only way to go. There are various mathematical expressions, called as metrics, defined in the field for assessing the performance of different types of machine learning systems. In this chapter, we are going to focus on the theory of performance measurement and metrics.

### 17.2 Sample Size

Whenever a dataset is selected for training a machine learning model, one must keep a fraction of the set aside for testing the performance as we studied in the earlier chapter. This *test* data should not be used in any way during the training and validation process. This is a critical aspect, and anyone who wants to venture into this field should treat this as a sanctimonious and uber-principle. Typically, the *training* and *test* split is done as 70–30% or 75–25%. The split is governed by these two considerations:

1. The model should get as much as training data as possible.
2. The test set should contain sufficient samples to have statistical confidence in the metrics produced using it.

There is a whole area dedicated on computation of statistical significance and confidence intervals in the field of statistics. There is quite a wide variation in statistical community in choosing bare minimum sample, and it ranges anywhere from 30 to 100 for a single-variable and single-dimensional problem. When there are more than one variable and more than one dimension, the rule gets much more complex depending on the dependency between the dimensions and variables, etc. In such cases, a sufficient number of training samples need to be decided on a case-by-case basis. In general, the sample size also depends on how much margin of error you are comfortable with and at what statistical confidence you would like to have those error margins. If you want 5% error margin at 90% confidence, you would need certain number of samples, but if you want 95% confidence for the same error margin, you would need more samples and so on. As the error margin reduces, the required sample size increases, and as the confidence requirements decrease, the required sample size also reduces. There numerous publications on sample size selection based on the problem at hand [94, 95]. There are also multiple sample size calculators available online, such as [42] that can be used as needed.

The following sections define the most commonly used performance metrics. Some of them may appear trivial, but they are still provided for completeness. All the metrics are based on the assumption of discrete data. In case of continuous functions, the summation is replaced with integral, but the concepts remain the same.

## 17.3 Metrics Based on Numerical Error

These are the simplest form of metrics and are used in regression type of problems, where we are predicting a numerical valued outcome. When we have a list of expected values, say  $(y_i, i = 1, \dots, n)$ , and we have a list of predicted values, say  $(\hat{y}_i, i = 1, \dots, n)$ , the error in prediction can be given using the following metrics:

### 17.3.1 Mean Absolute Error

Mean absolute error is defined as

$$e_{\text{mae}} = \frac{\sum_{i=1}^{i=n} |\hat{y}_i - y_i|}{n} \quad (17.1)$$

Use of only *mean error* is typically avoided, as it can lead to unusually low values due to cancellation between the negative and positive errors.



### 17.3.2 Mean Squared Error

Mean squared error is defined as

$$e_{\text{mse}} = \frac{\sum_{i=1}^{i=n} (\hat{y}_i - y_i)^2}{n} \quad (17.2)$$

MSE typically penalizes larger errors (outliers) more heavily than smaller errors compared to MAE. One can choose to use either of them based on specific problem or use both.

### 17.3.3 Root Mean Squared Error

Root mean squared error is defined as

$$e_{\text{rmse}} = \sqrt{\frac{\sum_{i=1}^{i=n} |\hat{y}_i - y_i|}{n}} \quad (17.3)$$

RMSE reduces the sensitivity of the error to few outliers but still is more sensitive compared to the MAE.

### 17.3.4 Normalized Error

In many cases, all the above error metrics can produce some arbitrary number between  $-\infty$  and  $\infty$ . These numbers only make sense in a relative manner. For example, we can compare the performance of the two systems operating on the same data by comparing the errors produced by them. However, if we are looking at a given instance of the single system, then the one value of error computed using one of the above techniques can be quite arbitrary. This situation can be improved by using some form of normalization of the error, so that the error is bounded by lower and upper bounds. Typically the bounds used are  $(-1+1)$ ,  $(0-1)$ , or  $(0-100)$ . This way, even a single instance of normalized error can make sense on its own. All the above error definitions can have their own normalized counterpart.

## 17.4 Metrics Based on Categorical Error

Performance metrics based on categorical data are quite a bit different and are required in the case of classification problems. In order to quantitatively define the

metrics, we need to introduce certain terminology. Consider a problem of binary classification. Let there be total of  $n_1$  samples of class 1 and  $n_2$  samples of class 2. The total number of samples is  $n = n_1 + n_2$ . The classifier predicts  $\hat{n}_1$  number of samples for class 1 and  $\hat{n}_2$  number of samples for class 2, such that  $\hat{n}_1 + \hat{n}_2 = n$ . In such situations, the metrics can be calculated either from the perspective of only class 1 or only class 2 or with a joint perspective.

### 17.4.1 Accuracy

To quantitatively define the metrics, we need to define some more parameters. Let  $n_{ij}$  be the number of samples originally of class  $i$  that are classified as class  $j$ . Accuracy metric can be defined using this notation as

$$A = \frac{n_{11} + n_{22}}{n_{11} + n_{12} + n_{21} + n_{22}} \quad (17.4)$$

Accuracy is a metric from joint perspective and does not focus only on one or the other class.

### 17.4.2 Precision and Recall

Precision and recall metrics focus on one or the other class. We will need to define few more terms to come up with expressions. Let  $TP_1$  be the number of true positives from the perspective of class 1, meaning the predicted class is class 1 and it matches with actual class. Hence,  $TP_1 = n_{11}$ . Let  $FP_1$  be the false positives from the perspective of class 1. False positive means the sample is actually from class 2 but is classified as class 1. Hence,  $FP_1 = n_{21}$ . They are also called as *false calls*. Let  $TN_1$  be the number of true negatives from the perspective of class 1, meaning the predicted class is class 2 when the actual class is class 2. Hence,  $TN_1 = n_{22}$ . Let  $FN_1$  be the number of false negatives from the perspective of class 1. False negative means the sample is actually from class 1 but is classified as class 2. They are also alternatively called as *misses*. Hence,  $FN_1 = n_{12}$ . Now we can define the two metrics from perspective of class 1 as *precision*,  $P_1$ , and *recall*,  $R_1$ .

$$P_1 = \frac{TP_1}{TP_1 + FP_1} \quad (17.5)$$

$$= \frac{n_{11}}{n_{11} + n_{21}} \quad (17.6)$$

$$R_1 = \frac{TP_1}{TP_1 + FN_1} \quad (17.7)$$

$$= \frac{n_{11}}{n_{11} + n_{12}} \quad (17.8)$$

As can be seen, the numerator is the same in both equations, but the denominator is different. In order to subjectively understand these entities, one can follow these rules of thumbs.

### Rules of Thumb for Understanding Precision and Recall

1. Precision can be interpreted as how many samples actually belong to class 1 from the pool of samples that are classified as class 1.
2. Recall can be interpreted as how many samples actually belong to class 1 from the pool of samples that actually belong to class 1.

In the same manner, we can create the precision and recall metrics from the perspective of class 2. The true and false positives and negatives are defined as  $TP_2 = n_{22}$ ,  $FP_2 = n_{12}$ ,  $TN_2 = n_{11}$ , and  $FN_2 = n_{21}$ . The precision and recall are defined as

$$P_2 = \frac{TP_2}{TP_2 + FP_2} \quad (17.9)$$

$$= \frac{n_{22}}{n_{22} + n_{12}} \quad (17.10)$$

$$R_2 = \frac{TP_2}{TP_1 + FN_2} \quad (17.11)$$

$$= \frac{n_{22}}{n_{22} + n_{21}} \quad (17.12)$$

#### 17.4.2.1 F-Score

*F-score* metric combines the two metrics (precision and recall) into a single metric, as a harmonic mean of precision and recall. It is sometimes called as *F-measure* or *F1 score*, but they all mean the same quantity.

$$F = 2 \cdot \frac{P \cdot R}{P + R} \quad (17.13)$$

The F-score is also a single class-focused metric and can be computed for each class, using the corresponding precision and recall values.

### 17.4.2.2 Confusion Matrix

All the analysis described above can be further generalized for case of  $n$ -classes. Along with these metrics, oftentimes, the full matrix of misclassifications  $n_{ij}$ ,  $i = 1, \dots, n$  and  $j = 1, \dots, n$  is computed and is called as confusion matrix.

### 17.4.3 Receiver Operating Characteristic (ROC) Curve Analysis

When a binary classifier is designed, there is some form of threshold or a condition that is ultimately involved that separates the two classes. If we move the threshold in any direction, it affects precision and recall in opposite manner. In other words, if we try to improve the precision, the recall typically gets worse and vice versa. Hence, in order to understand the core performance of the classifier in separating the two classes, a plot called receiver operating characteristics (ROC) is generated. The name ROC can be quite confusing, as there is no receiver in any of the considerations here. The concept of ROC has originated from the theory of electronic communications. In an electronic communication system, there is a transmitter and receiver. In ideal situation, the receiver needs to decode the signals with no error as sent by the transmitter. However, due to noise on the transmission medium, there are always some errors. The ROC curve-based analysis was developed to provide quantitative measure on the performance of the receiver. ROC curve is typically plotted in terms of *true positive rate (TPR)* and *false positive rate (FPR)*. These terms are defined as

$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}} \quad (17.14)$$

$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}} \quad (17.15)$$

As can be seen from the equations, *TPR* has the same definition as *recall*, and it is also called as *sensitivity* in this context. *FPR* is different than precision and is called as *specificity*. However, in the current context, they are called as rate, and we are going to vary these parameters as a function of the threshold.

Figures 17.1 and 17.2 show two examples of ROC curve. The x-axis marks the FPR and the y-axis marks the TPR. At the origin, both the rates are 0, as threshold is such that we are classifying all samples into non-desirable class. Then as we move the threshold, ideally the TPR should increase much faster than FPR. At the opposite end, we are classifying all samples into desired class, and both rates are 1. In any given application, we need to choose a threshold that provides the optimal performance under given constraints. However, one aspect of the ROC curve provides a metric that is independent of the application and is considered as the most

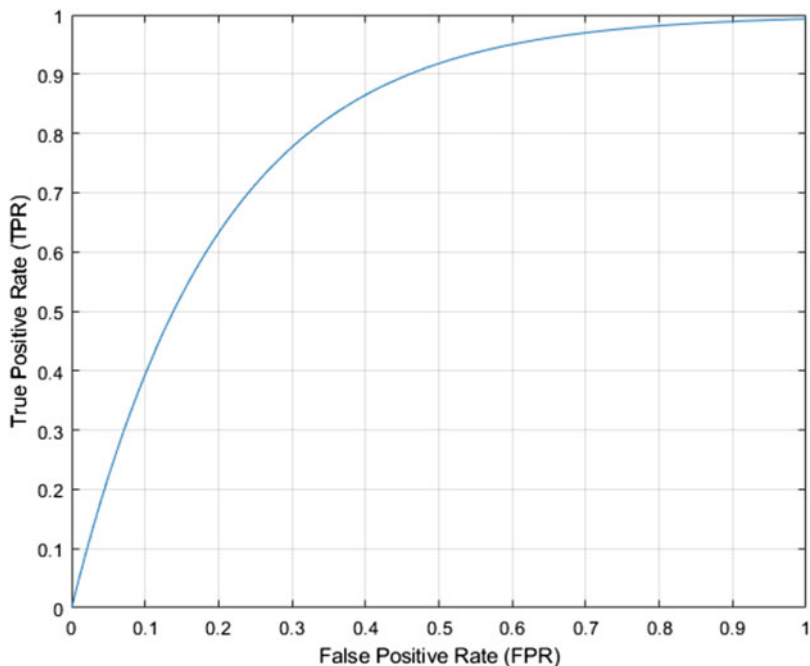


Fig. 17.1 Example of a relatively bad ROC curve

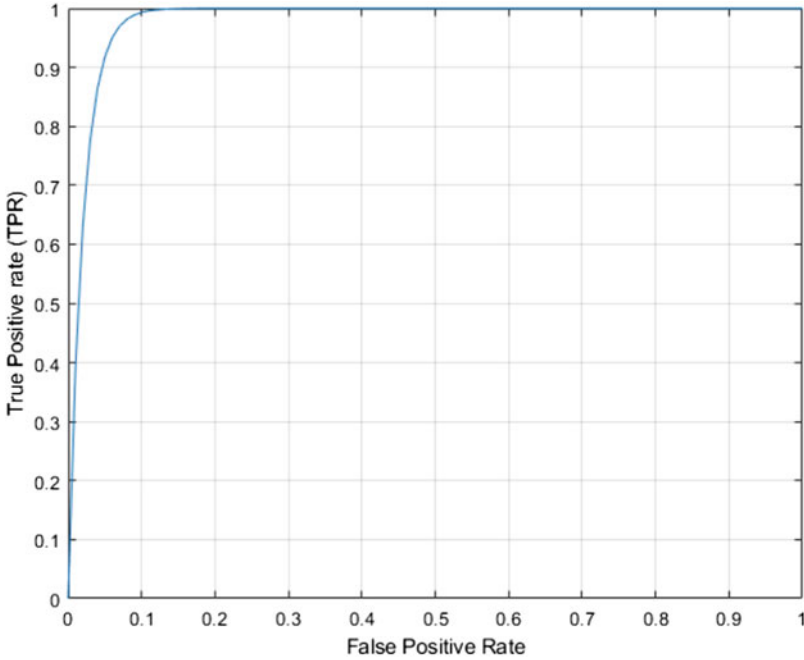
fundamental property of the classifier, and that is the area under the ROC curve or just *AUC*. The greater the area, the better the classifier.

#### 17.4.4 Precision-Recall Curve

Similar to ROC curve, we can also create precision-recall curve or PR curve, where we plot the precision against recall in similar fashion by varying the threshold. In some applications, PR curve can provide better insight into the classifier performance, but in general, using both the curves gives the complete picture.

### 17.5 Hypothesis Testing

Hypothesis testing is essentially a probabilistic process of determining whether a given hypothesis is true. Typical explanation of this process involves quite a few concepts that are deeply rooted in theory of statistics, and a person not conversant with these concepts can quickly get lost. However, the concept of hypothesis testing



**Fig. 17.2** Example of a relatively good ROC curve

is quite generic, and we will study this concept with sufficient clarity here without going into too much details.

### ***17.5.1 Background***

Before we dive deep into the process of hypothesis testing, let us first understand the background where we will apply this technique. We will take up a simple practical example to illustrate this. Consider daily weather predictions from the local news. Quite often we see that they are wrong, so we want to find out whether they actually have some scientific truth in them or they are just random predictions. Hypothesis testing framework can help in this analysis. Let us assume that there are four different types of weather conditions that are possible in a given region: sunny, cloudy, rainy, and snowy. Thus, if one starts to make simple random predictions, those predictions would have only 25% chance of becoming true each day. Then we note the weather predictions for each day for over a period of say 6 months or 180 days. Then we find out what was the accuracy of the weather predictions of the local news channel and compare them with the default random value of 25%

and determine whether the weather predictions from the local news channel are any better than pure chance using theory of probability.

### ***17.5.2 Steps in Hypothesis Testing***

The entire process of hypothesis testing is split into four broad parts.

#### **Steps in Hypothesis Testing**

1. The first step in the process involves defining what is called as null hypothesis  $H_0$ . Null hypothesis is essentially the default random behavior. In the example that we just described, null hypothesis would be “weather predictions by the local news channel are random and purely based on chance.”
2. The second step is to gather experimental or observational data. In the case of the current example, that would mean the difference between the weather predictions recorded every day from the local news channel and actual observed weather conditions. These observations constitute a random variable, called as *test statistic*.
3. The third step involves computation of the probability (also called as *P-value*) that the test statistic actually proves the null hypothesis.
4. The fourth step involves comparing the *P-value* with the predetermined level of significance, denoted as  $\alpha$  (typically, the value of  $\alpha$  lies between 5% and 1%), to either accept or reject the null hypothesis. The level of significance can be thought of as probability of test statistic lying outside of the expected region if null hypothesis is true.

### ***17.5.3 A/B Testing***

In many machine learning scenarios, we typically encounter comparison of more than one model for the solution of a problem. In such cases, we can extend the concept of hypothesis testing to what is called as *A/B testing*. In *A/B testing*, rather than using a default null hypothesis, we use hypothesis A, which is the outcome from the model A. Then, we compare this with the outcome of model B as the second hypothesis. Following similar steps, we can find out which model produces better results based on predetermined significance level.

## **17.6 Conclusion**

Performance measurement is the most critical aspect of building a machine learning system. Without having a proper definition of quantitative metrics, one cannot

effectively compare and contrast different approaches or models. In this chapter, we studied the different metrics used for measuring the performance of machine learning systems. Different metrics are needed based on the application as well as the type and size of data, and depending on the needs, one can choose which metrics to use.

## 17.7 Exercises

1. Take the example of Iris data, and use the pipeline that we studied in the previous chapter, and apply all the classification metrics that we learned in this chapter. Sklearn library provides computation of all these metrics out of the box [43].
2. Take an example of your choice for regression-type problem, and apply all the regression metrics we learned in this chapter. Compare how the different errors change with change in model.



# Part IV

## Artificial Intelligence

I know I've made some very poor decisions recently, but I can give you my complete assurance that my work will be back to normal. I've still got the greatest enthusiasm and confidence in the mission. And I want to help you.

—HAL 9000, “2001: A Space Odyssey”

### **Part Synopsis**

This part focuses on the implementation of the ML models to develop artificially intelligent applications.

# Chapter 18

## Classification



### 18.1 Introduction

We have discussed various algorithms that are designed for solving the classification problems in previous part. In this chapter, we are going to look at the topic in a slightly different way. We are going to look at some real-world problems that need underlying classification algorithms to solve. We will list a few problems that are well known and have made significant impact in the consumer space. We will try to build a machine learning pipeline to solve one of those problems in a rudimentary manner. Although we will not explicitly solve the problems, the process of tackling these classes of problems will give the reader a practical insight into the applications of machine learning in real world.

### 18.2 Examples of Real-World Problems in Classification

Although classification is one of the simple applications of machine learning theory, the real-world situations are not typically as simple as textbook problems like classification of flowers illustrated by Iris dataset [3]. Here are few examples that have classification as underlying machine learning model:

1. Spam email detection: Classifying emails as genuine or spam. This is one of the crucial components of most email application and services these days. If an application can successfully separate the spam emails from genuine emails, it can quickly become application of choice by millions of people. The spam can further be divided into categories like deal alerts, newsletters, fishing emails, etc. making it a multiclass problem.
2. Image classification: There are numerous applications of image classification. The images can be of faces, and the objective is to classify the faces into males

and females. Or the images can be of animals, and the problem can be to classify them into different species. Or the images can be of different vehicles, and the objective is to classify them into various types of cars, and the list can go on and on.

3. Medical diagnosis: There are different types of sensing technologies available in medical diagnosis, like ultrasound, X-ray, magnetic resonance imaging (MRI), etc. This diagnostic sensory data can then be given to a machine learning system which can then analyze the signals and classify the medical conditions into different predetermined types.
4. Musical genre identification: When a piece of music is played, the machine learning system can automatically detect the genre of the music.

In order to understand the nuances of real-life machine learning, we will look at the first problem in greater detail in the rest of the chapter.

## 18.3 Spam Email Detection

Detecting spam emails against genuine emails is a very important and real-world problem that most email services, e.g., *Gmail*, *Hotmail* or *Outlook*, *Yahoo Mail*, etc. as well as email clients, e.g., various email apps on mobile platforms including Microsoft's Outlook app, etc. face. However, the scope of the solution for each is very different. For example, email services can have access to all the emails from all the accounts in their platform, while the email clients can only access the accounts that are accessed through them. Again, as per the ever increasingly strict privacy rules, the extent to which these emails can be accessed by the services and apps is getting reduced. In order to illustrate the problem-solving methodology, we will try to solve the problem of spam detection from the perspective of email service. The problem at hand is a very good example of binary classification. Let's define the precise scope of the problem.

### 18.3.1 Defining Scope

Let us consider that we have our own email service called as *email.com*. Let us assume we have access to all the emails for all the accounts that are opened on our email service in *aggregate* and *anonymous* manner. This means only the automated algorithms can access the emails, and no human can look at any specific message. Also, when we are dealing with errors and metrics, only aggregate-level metrics can be accessed. There is no way to check why any specific message is being misclassified. This is a typical restriction that is faced by all email service providers from the legal standpoint, where direct access to personal emails is prohibited.

### ***18.3.2 Assumptions***

In real life, the problem definition is never crisp like the one that appears in a textbook (e.g., like Iris problem). The problem is typically identified in a broad sense, and micro details need to be figured out by the data scientist. The case at hand is no different. Hence, we need to start making some assumptions in order to narrow down the scope of the problem further. These assumptions are essentially an extension of the problem definition itself.

#### **18.3.2.1 Assumptions About the Spam Emails**

1. The spam emails can be originated from *email.com* as well as from other domains.
2. The sender of the spam emails must be sending large amount of emails from his/her account.
3. Most emails sent by spammer account would be almost character-wise identical to each other with slight differences possible if the emails have been customized for each user. In such cases, each email will have a different greeting name, but the other contents would still be the same.
4. Spam emails can fall into two categories: (1) marketing and advertising emails and (2) emails that are used to steal the identity of a person or to make him/her commit some fraudulent financial transaction. In the latter case, spam email would contain some URLs or fake email address(es), which will take the user to insecure destinations.
5. Spam emails are also created to look appealing and attractive, and hence, they are likely to be formatted as HTML rather than plain text.

#### **18.3.2.2 Assumptions About the Genuine Emails**

1. The genuine emails are sent from the people known to the person to whom they are sent.
2. The recipient likely reads those emails to get the new information.
3. The genuine emails can also originate in bulk, e.g., email reminder for credit card payments.
4. Most personal genuine emails are likely to be in plain text format.

#### **18.3.2.3 Assumptions About Precision and Recall Trade-Off**

The assumptions that we have made for defining the precision and recall would always have some overlap. As a result, the model would never achieve perfect

precision and recall for a typical training data set. This is one of the reasons we can almost never achieve an accuracy of 100%. The objective is to find an optimal trade-off between the prediction of two classes to maximize the overall experience.

An easy starting point can be to optimize for overall accuracy. In this case, we don't want to introduce any additional bias in the model. We use the default distribution that exists between the spam and genuine emails and try to optimize for highest accuracy in correct identification and reduce the number of misclassified emails. With this objective, if the number of spam emails is much higher than genuine emails (a typical case in most inboxes), the model will have a high bias toward correctly classifying spams, and a significant number of genuine emails would likely get pushed into spam folder. This is an unpleasant experience, when a user does not find an important email and potentially loses an opportunity or an appointment. Now, instead if we try to make the model unbiased, but taking equal number of emails from spam and genuine sets for training or adjusting the threshold, we are still likely to face a similar issue albeit with lesser severity. In this particular case to optimize for optimal user experience, we need to explicitly bias the model toward having very high recall for detecting the genuine email. With such bias, there is likely a loss in precision, where a larger number of spam emails would get classified as genuine. However, users of the email service will not lose any important emails. What we are doing here is called as a trade-off between precision and recall. In other words, we are trying to find the right point of operation on the *receiver operating characteristic* or *ROC* curve as described in Chap. 17.

### 18.3.3 Skew in the Data

We discussed the inherent skew in the data in earlier section and how it can affect our policy in designing the classifier. Let's take a concrete example to illustrate the concept. Let's consider different example distributions. In one case, the number of spam emails is 40%, and the remaining 60% are genuine emails. In this situation, if we design the model to be equally accurate (say 80%) for precision of both classes, then the resulting overall accuracy is also going to be 80%. However, consider a case when we design the model to be 90% accurate for genuine emails, and as a result, it is only 70% accurate for spam. Now let us see what is the total accuracy. The overall accuracy is going to be weights by the distribution of each class. Hence, the overall accuracy will be given as

$$A_{\text{overall}} = (0.6 \times 0.9) + (0.4 \times 0.7) = 0.82 \quad (18.1)$$

Thus, the overall accuracy is slightly higher than the accuracy achieved with previous approach of treating both accuracies equally important. Small amount of skew is typically not as impacting on the overall accuracy, but when there is large skew (e.g., 10–90%, etc.), then the design of the model needs to be carefully

evaluated. If we want to remove the effect of the skew in the data in training model, we can use the technique of stratified sampling as discussed in Chap. 16. It is useful to note that as stratified sampling can be used to remove the skew, it can also be used to add a predetermined skew in the training data.

### 18.3.4 *Supervised Learning*

We will treat this problem as strictly of supervised learning type, and we have access to a labelled training set of spam and genuine emails. A set of about 100,000 labelled emails where say 60,000 emails are spam and the remaining 40,000 emails are genuine should be sufficient to train the model and generate metrics on its performance. Unsupervised methods can be used, but the outcomes from those models can be arbitrary and may not align with our objective.

### 18.3.5 *Feature Engineering*

All the raw feature space we have access to is in the form of inboxes of users for the past several months/years. A lot of information is contained in the header of an email in the form of multiple fields such as *From*, *To*, *date* and *Subject*, *User-Agent*, *MIME-Version*, and so on. All these fields can be part of the feature space used to train the classifier model.

The emails can be written as simple text or they can be formatted as HTML. This formatting type itself can be one feature. Then, a lot of simple features can be mined from an actual content or body of the emails, for example, frequency of certain words, frequency of certain characters, number of words or characters in uppercase, etc. With some advanced linguistic processing, we can augment these features with features like number of misspelled words, occurrences of incorrect grammar, etc.

The features can be in different data types: numerical, string, binary datetime, and so on. Custom processing with different techniques we learned in earlier chapters will be needed to convert all these features into numerical features that can be readily consumed by the classifier model.

## 18.4 Implementing Spam Filter Classifier

We will use a publicly available dataset for spam detection from UCI repository [44]. This dataset contains 4601 instances with spam emails accounting for 1813 and genuine emails accounting for 2788. The rough distribution between spam emails and genuine emails is 40–60%. The work of feature engineering is already done for

us, and 57 numerical features are already extracted. Full details of the features are given here [44]. The features focus on simple frequency-based continuous valued functions such as:

1. Frequency for certain pre-identified words
2. Lengths of words in all caps
3. Number of characters in all caps

### 18.4.1 Using Azure ML

We will use Azure ML workspace to implement this classifier. We learned how to set this up in Chap. 4. Figure 18.1 shows how we can use the built-in libraries in Azure ML to directly import the CSV file from UCI repository into Python variable.

Once we have loaded the data, we can move on to the standard steps of splitting the data into training and test and then using the training set to train the random forest classifier as shown in Fig. 18.2.

Now we can apply the trained classifier on the test set and see the accuracy using the *score* command from sklearn as shown in Fig. 18.3. We get a respectable score of 0.9485.

We now move our attention to the concept of hyperparameter tuning. First, we identify all the parameters that can be tuned for random forest classifier as shown in Fig. 18.4.

Now, we can choose a subset of these parameters to be tuned. Here, we choose three parameters *max\_depth*, *n\_estimators*, and *max\_features* and assign suitable

```

1  from azureml.core import Dataset
2  from azureml.data.dataset_factory import DataType
3
4  # Create a TabularDataset from a delimited file behind a public URL
5  web_path = "https://archive.ics.uci.edu/ml/machine-learning-databases/spambase/spabase.data"
6  uci_spabase = Dataset.Tabular.from_delimited_files(path=web_path, header=False,
7  [infer_column_types=False])
8  uci_spabase.take(3).to_pandas_dataframe()

```

✓ 23 sec

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9	Column10	...	Column-
0	0	0.64	0.64	0	0.32	0	0	0	0	0	...	0
1	0.21	0.28	0.5	0	0.14	0.28	0.21	0.07	0	0.94	...	0
2	0.06	0	0.71	0	1.23	0.19	0.19	0.12	0.64	0.25	...	0

3 rows × 58 columns

Fig. 18.1 Importing the UCI spabase dataset formatted as CSV directly into Azure ML

```

1 (uci_spambase_train, uci_spambase_test) = uci_spambase.random_split(percentage=0.75, seed=11)
2 X_train = uci_spambase_train.drop_columns('Column58').to_pandas_dataframe()
3 y_train = uci_spambase_train.keep_columns('Column58').to_pandas_dataframe().values.ravel()
4 X_test = uci_spambase_test.drop_columns('Column58').to_pandas_dataframe()
5 y_test = uci_spambase_test.keep_columns('Column58').to_pandas_dataframe().values.ravel()
✓ 1 sec

```

+ Code + Markdown

```

1 from sklearn.ensemble import RandomForestClassifier
2 rfc = RandomForestClassifier()
3 rfc.fit(X_train, y_train)
✓ 14 sec

```

```

RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
criterion='gini', max_depth=None, max_features='auto',
max_leaf_nodes=None, max_samples=None,
min_impurity_decrease=0.0, min_impurity_split=None,
min_samples_leaf=1, min_samples_split=2,
min_weight_fraction_leaf=0.0, n_estimators=100,
n_jobs=None, oob_score=False, random_state=None,
verbose=0, warm_start=False)

```

Fig. 18.2 Splitting the data and training the random forest classifier with training set

```

1 rfc.score(X_test, y_test)
✓ <1 sec

```

0.9584055459272097

Fig. 18.3 Applying trained random forest classifier on test set and getting the accuracy

```

1 rfc.get_params().keys()
✓ <1 sec

```

```

dict_keys(['bootstrap', 'ccp_alpha', 'class_weight', 'criterion', 'max_depth', 'max_features',
'max_leaf_nodes', 'max_samples', 'min_impurity_decrease', 'min_impurity_split', 'min_samples_leaf',
'min_samples_split', 'min_weight_fraction_leaf', 'n_estimators', 'n_jobs', 'oob_score', 'random_state',
'verbose', 'warm_start'])

```

Fig. 18.4 The tunable hyperparameters for random forest classifier as supported by sklearn

range of values for each. *max\_depth* parameter decides the max depth of each tree, *n\_estimators* defines the max number of trees created during training, and *max\_features* defines the number of features used for identifying the best split. We choose two to four values for each parameter. Then we run the command *GridSearchCV* that stands for *grid search using cross-validation* as shown in Fig. 18.5. It is important to note that the more dimensions and more parameters you add in each dimension, the training time is going to increase exponentially. In the current case, the training process will run for  $4 * 4 * 2$  or 32 times. So, it is a good idea to experiment with only a handful of parameters at a time depending on the size of data. Cross-validation technique separates the training data into training and validation and runs each iteration on the smaller training set and validates



```

1  from sklearn.model_selection import GridSearchCV
2  √ param_grid_rfc = {
3      "max_depth": [3, 4, 5, 7],
4      "n_estimators": [50, 100, 150, 200],
5      "max_features": [3,5]
6  }
7
8  grid_cv_rfc = GridSearchCV(rfc, param_grid_rfc, n_jobs=1, cv=3, scoring="roc_auc")
9
10 grid_cv_rfc.fit(X_train, y_train)
11 grid_cv_rfc.best_score_
✓ 34 sec
0.9675378596717614

```

**Fig. 18.5** Running the grid search with three-dimensional parameter grid

```

1  grid_cv_rfc.best_params_
✓ <1 sec
{'max_depth': 7, 'max_features': 5, 'n_estimators': 200}

1  grid_cv_rfc.score(X_test, y_test)
✓ <1 sec
0.9812475319464846

```

**Fig. 18.6** Checking the parameters that gave the best results and score on test set

the metrics on the validation set. After training the algorithm on all possible combinations of the parameters, it finds the best set of parameters that yield the best performance on the validation set as seen in Fig. 18.6. Depending on the values of these parameters and where they lie in the ranges we had selected, we can update the original list of parameter ranges to get even better performance.

Then we apply this best model (which can be accessed from *grid\_cv\_rfc*) on the test set and get an even better accuracy of 0.9741.

We have been using *sklearn* library for most applications so far, and it is one of the most commonly used libraries that provides majority of the functions. However, there are many other libraries available to use from open-source domain, which in many cases outperform *sklearn*. XGBoost is a good example of decision tree classifier. Figure 18.7 shows how we can import it in Azure ML. This library is readily available here, but if a library is not available, we can quickly use *pip install* command to add more libraries. Then we check the list of parameters that can be tuned for XGBoost classifier. Although XGBoost also uses random forest technique at heart, it gives more parameters that can be tuned as you can see in Fig. 18.7.

Now we choose a list of four parameters with 4, 3, 3, 3 values for each, resulting in total of 108 training runs when we apply *GridSearchCV*. As we can see in Fig. 18.8, the process takes about 1 min and 42 s on the current data and gets the best set of parameters. When we apply this model on the test set, we get even better performance with 0.9843 accuracy.

```

1 from xgboost import XGBClassifier
2 xgbc = XGBClassifier()
✓ 28 sec

1 xgbc.get_params().keys()
✓ <1 sec

dict_keys(['objective', 'use_label_encoder', 'base_score', 'booster', 'colsample_bylevel',
'colsample_bynode', 'colsample_bytree', 'gamma', 'gpu_id', 'importance_type',
'interaction_constraints', 'learning_rate', 'max_delta_step', 'max_depth', 'min_child_weight',
'missing', 'monotone_constraints', 'n_estimators', 'n_jobs', 'num_parallel_tree', 'random_state',
'reg_alpha', 'reg_lambda', 'scale_pos_weight', 'subsample', 'tree_method', 'validate_parameters',
'verbosity'])

```

**Fig. 18.7** Using XGBoost library as an alternative to sklearn and checking its tunable hyperparameters

```

1  param_grid_xgbc = {
2      "max_depth": [3, 4, 5, 7],
3      "learning_rate": [0.1, 0.05, 0.01],
4      "gamma": [0, 0.25, 1],
5      "reg_lambda": [0, 1, 10]
6  }
7
8  grid_cv_xgbc = GridSearchCV(xgbc, param_grid_xgbc, n_jobs=1, cv=3, scoring="roc_auc")
9
10 grid_cv_xgbc.fit(Train_xgb)
11 grid_cv_xgbc.best_score_
    Press shift + enter to run
0.9771727272

1  grid_cv_xgbc.score(X_test, y_test)
    Press shift + enter to run
0.982628936289

```

**Fig. 18.8** Training XGBoost classifier and checking its accuracy on test set

## 18.5 Conclusion

In this chapter, we looked at various popular real-life problems that need underlying classification models to solve. We then looked at the specific example of email spam detection in detail by going from the assumptions that need to be made to constrain the problem so that it can be solved using a suitable training data and machine learning algorithm. Then we took a data that had already gone through the process of feature engineering and applied two different classifiers. We also learned the concept of cross-validation and grid search and used an alternative library XGBoost. We then compared the results from both.

## 18.6 Exercises

1. We learned the concept of hyperparameter tuning in this chapter and applied it to random forest classifier from `sklearn` and `XGBoost`. As an exercise, use more parameters in the tuning process with `sklearn` random forest classifier, and try to find the best possible accuracy you can get on test set. What conclusions can you draw from the exercise?
2. Use the same process with `XGBoost` classifier, and see what is the best performance you can extract. Compare the accuracy from `sklearn`. Also compare the set of best parameters obtained in each process and draw conclusions.
3. Try to vary the random seed in the data split a few times and rerun the above pipelines. See if there is a noticeable difference in the performance with change in training-test split.
4. There is another library called *LightGBM*. Use that library and rerun the experiment.
5. There is a well-known dataset for image classification known as *CIFAR-10* [45]. It contains 60,000 color images from 10 classes with uniform distribution of 6000 images per class. Use this dataset and apply the three models discussed above. With this dataset being much larger, you will need to select the parameters carefully for grid search. You can use the learnings from above.

# Chapter 19

## Regression



### 19.1 Introduction

Continuing from the last chapter where we looked at sample applications of classification in real life, in this chapter, we will look at real-life applications of regression technique. Regression problems are characterized by prediction of a real valued output in one or more dimensions. The output can be bounded in a variety of ways, but it is continuous valued as opposed to the case in classification. The input features need not be all numerical though. They can range from numerical to categorical to pure string valued. We will take up the problem in its raw form and derive a customized specific version of it suitable for a machine learning system to solve by making appropriate assumptions. Then, we will build a careful framework to analyze the results of the system in the form of suitable quantitative metrics.

### 19.2 Examples of Real-World Problems

Although predicting the value of a functional is a default mathematical application of regression, there are numerous real-life situations that can benefit from regression. Ranging from stock market to predict values of a particular stock or mutual fund or gold for that matter to predicting value of real estates, we can use regression techniques. In general business setup, regression techniques can help in predicting sales of a given product or service, risks, salaries at various positions in different locations, and so on. Regression has been used quite effectively in the field of medicine and healthcare for predicting side-effects of drugs, risks and dependencies between different drugs/supplements, rates and durations of hospitalization and so on.

## 19.3 Predicting Real Estate Prices

We will use the problem of real estate value prediction as an example in this chapter to illustrate the process similar to the previous chapter. Predicting values of real estate can be one of the hot topics in our lives at one or the other stage. The solution of the problem is useful to consumers as well as banks as well as real estate agents and so on. The prediction is going to be a real valued number of the dollar amount, and hence, it fits perfectly as a regression problem. It is important to identify the nuances of the problem further to narrow down the scope of it. When one talks about the real estate values, there are many aspects to it, and we must narrow down the definition to the specific problem we want to solve.

### 19.3.1 *Defining Regression-Specific Problem*

To define the specific problem that we want to solve, let us first list down all the different aspects when one talks about real estate pricing.

### 19.3.2 *Aspects in Real Estate Value Prediction*

1. Location, location, location (a typical remark you will hear if you ask a real estate agent!)
2. Trends in price of a certain specific property over time, specifically in different seasons
3. Price of certain type of property, e.g., a two-bed condo, in certain area
4. Average price of house in certain area
5. Range of prices of a type of property in various different areas
6. Price of a certain specific property at a given time
7. Proximity of the property from relevant businesses

Although the above list is far from comprehensive, it gives an idea about the possible different directions in which one can think when dealing with real estate prices. Each aspect defines a different problem from the machine learning perspective and will need a different set of features and possibly even a different type of model and a different type of training data. The property can be an empty land or a commercial property or a residential property with a single family. In each case, the factors that affect the price can be drastically different and ultimately would lead to a completely different machine learning problem.

### 19.3.3 Gather Labelled Data

The next step is to identify the labelled set of data. As regression is a supervised learning problem, without availability of appropriate labelled set of data, we cannot proceed. To illustrate the steps in modeling a regression problem, we will use another curated set from UCI machine learning repository [46]. This dataset has already gone through the feature engineering tasks and has identified six features. This dataset assumes personal housing property, so the features are created from that perspective.

1. Transaction date
2. Age of the property
3. Distance to the nearest MRT station
4. Number of convenience stores in the living circle on foot
5. Latitude
6. Longitude

The expected outcome is in the form of price (Taiwanese dollars) per unit area. The unit of area is a Taiwanese unit of *Ping*. One Ping is 3.3 square meter.

## 19.4 Implementing Regression

The data from UCI is available as an *Excel* spreadsheet, so we need to use the appropriate reader to read the data. You can also download it and convert the file to CSV format and use the standard reader. We will use a built-in function provided in Azure ML to read this data directly as shown in Fig. 19.1. The data contains only 414 rows or samples. The first column is sample number and the last column is price per unit area as described earlier. The middle six columns contain all the features as given by the column headings. The next step is to split the data into training and test sets. As the size of the data available is small, we will use 75-25 split. Some additional preprocessing needs to happen to make sure the imported data is numeric as shown in Fig. 19.2. Now, we will use linear (ridge) regression model to predict the trends in the real estate pricing. We start off with regularization parameter  $\alpha = 1$  as shown in Fig. 19.3. The accuracy score is  $-7.045$ , which is fairly arbitrary and cannot give us a good idea about how the predictions are tracking with expected outputs. In order to visualize this, we plot the predictions along with expected outcomes as shown in Fig. 19.4. As can be seen from the plot, there is a clear bias between the labels and predictions, where predictions are typically higher than the label values. However, there are few instances where the predictions are lower than the labels. In order to understand the reason for this bias, let us plot the training data and predictions of the model for training data. As we can see in Fig. 19.5, the predictions in training set are tracking well with the labels, but in

```

1  from azureml.core import Dataset
2  from azureml.data.dataset_factory import DataFactory
3
4  # Create a TabularDataset from a delimited file behind a public web URL
5  web_path = "https://archive.ics.uci.edu/ml/machine-learning-databases/00477/Real"
6  uci_real_estate = Dataset.from_excel_files(web_path, use_column_headers=True)
7  uci_real_estate.to_pandas_dataframe()

```

✓ 21 sec

"Dataset.from\_excel\_files" is deprecated after version 1.0.69. See Dataset API change notice at <https://aka.ms/dataset-deprecation>.

	No	X1 transaction date	X2 house age	X3 distance to the nearest MRT station	X4 number of convenience stores	X5 latitude	X6 longitude	Y house price of unit area
0	1.0	2012.916667	32.0	84.87882	10.0	24.98298	121.54024	37.9
1	2.0	2012.916667	19.5	306.59470	9.0	24.98034	121.53951	42.2
2	3.0	2013.583333	13.3	561.98450	5.0	24.98746	121.54391	47.3
3	4.0	2013.500000	13.3	561.98450	5.0	24.98746	121.54391	54.8
4	5.0	2012.833333	5.0	390.56840	5.0	24.97937	121.54245	43.1
...	...	...	...	...	...	...	...	...
409	410.0	2013.000000	13.7	4082.01500	0.0	24.94155	121.50381	15.4
410	411.0	2012.666667	5.6	90.45606	9.0	24.97433	121.54310	50.0
411	412.0	2013.250000	18.8	390.96960	7.0	24.97923	121.53986	40.6
412	413.0	2013.000000	8.1	104.81010	5.0	24.96674	121.54067	52.5
413	414.0	2013.500000	6.5	90.45606	9.0	24.97433	121.54310	63.9

414 rows × 8 columns

Fig. 19.1 Loading the UCI real estate pricing regression data as Excel spreadsheet

general, the predicted values are smaller in magnitude compared to labels. This case highlights an important aspect in machine learning called as bias-variance trade-off. The actual distribution of the labels is not quite linear compared to the input features, and as a result, the regression model has to make some approximations. These approximations are based on the trade-off between bias and variance. If the model tries to reduce the variance, as it is happening in this case, the bias is increased. If we somehow train the model to reduce the bias, the variance is likely to increase. Sklearn offers some examples of robust linear regressors that try to balance these trade-offs better. One such example is *TheilSenRegressor*. Figure 19.6 shows result after applying *TheilSenRegressor* model. Here, the bias is reduced a little, but the variance increases, and as a result, the overall error increases as well.

As the problem dictates here, linear models are not going to be successful beyond a certain point, and nonlinear models are required to improve the accuracy of predictions. Examples of nonlinear models that can be used here are support vector

```

1  from sklearn.model_selection import train_test_split
2  uci_real_estate_pd = uci_real_estate.to_pandas_dataframe()
3  uci_real_estate_train, uci_real_estate_test = train_test_split(uci_real_estate_pd,
4  test_size=0.25)
5
6  X_train = uci_real_estate_train.drop(['No', 'Y house price of unit area'], axis=1)
7  y_train = uci_real_estate_train['Y house price of unit area']
8  X_test = uci_real_estate_test.drop(['No', 'Y house price of unit area'], axis=1)
9  y_test = uci_real_estate_test['Y house price of unit area']
✓ 5 sec

```

```

1  import pandas as pd
2  X_train = X_train.apply(pd.to_numeric, errors='coerce')
3  X_test = X_test.apply(pd.to_numeric, errors='coerce')
✓ <1 sec

```

**Fig. 19.2** Splitting the data into training and test sets and preprocessing the data to ensure numerical input to model

```

1  from sklearn.linear_model import Ridge
2  rr1 = Ridge(alpha=1.0)
3
4  # Encode the output as required by the model
5  from sklearn.preprocessing import LabelEncoder
6  lab_enc = LabelEncoder()
7  y_train_enc = lab_enc.fit_transform(y_train)
8  y_test_enc = lab_enc.fit_transform(y_test)
9
10 # Fit the model
11 rr1.fit(X_train, y_train_enc)
✓ <1 sec

```

Ridge(alpha=1.0, copy\_X=True, fit\_intercept=True, max\_iter=None, normalize=False, random\_state=None, solver='auto', tol=0.001)

```

1  rr1.score(X_test, y_test_enc)
✓ <1 sec

```

-7.041430486341467

**Fig. 19.3** Training the model with training set and scoring it on the test set

regressor with nonlinear kernels or neural networks. It is left as an exercise for the reader to try these models and improve the performance.

### 19.4.1 Model Performance

It is quite important to decide the correct set of metrics to compute the model performance. With incorrect choice of metrics, the entire process can lead to a



```
1 import matplotlib.pyplot as plt
2
3 # Get the predictions for test set
4 y_pred_rr1 = rr1.predict(X_test)
5
6 # Plot the predictions and expected outcomes in the same plot
7 fig = plt.figure()
8 ax1 = fig.add_subplot(111)
9 plt.title('Plot of labels and predictions using Ridge Regression')
10 ax1.plot(y_test_enc, c='b', marker='.', label='Labels')
11 ax1.plot(y_pred_rr1, c='r', marker='o', label='Predictions')
12 plt.legend(loc='upper left')
13 fig.set_size_inches(20,5)
14 plt.grid(True)
15 plt.show()
✓ 8 sec
```

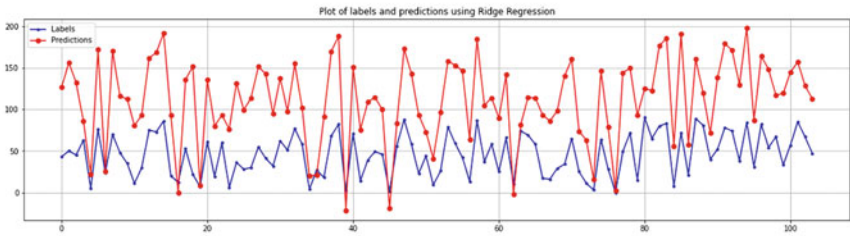


Fig. 19.4 Plotting the predictions from the ridge regression model with expected outcomes or labels

```
1 y_pred_train_rr1 = rr1.predict(X_train)
2 fig = plt.figure()
3 ax1 = fig.add_subplot(111)
4 plt.title('Plot of labels and prediction using Ridge Regression')
5 ax1.plot(y_train_enc, c='b', marker='.', label='Labels')
6 ax1.plot(y_pred_train_rr1, c='r', marker='o', label='Predictions')
7 plt.legend(loc='upper left')
8 fig.set_size_inches(20,5)
9 plt.grid(True)
10 plt.show()
✓ <1 sec
```

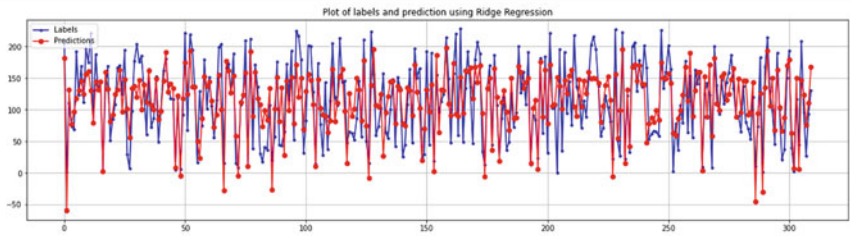
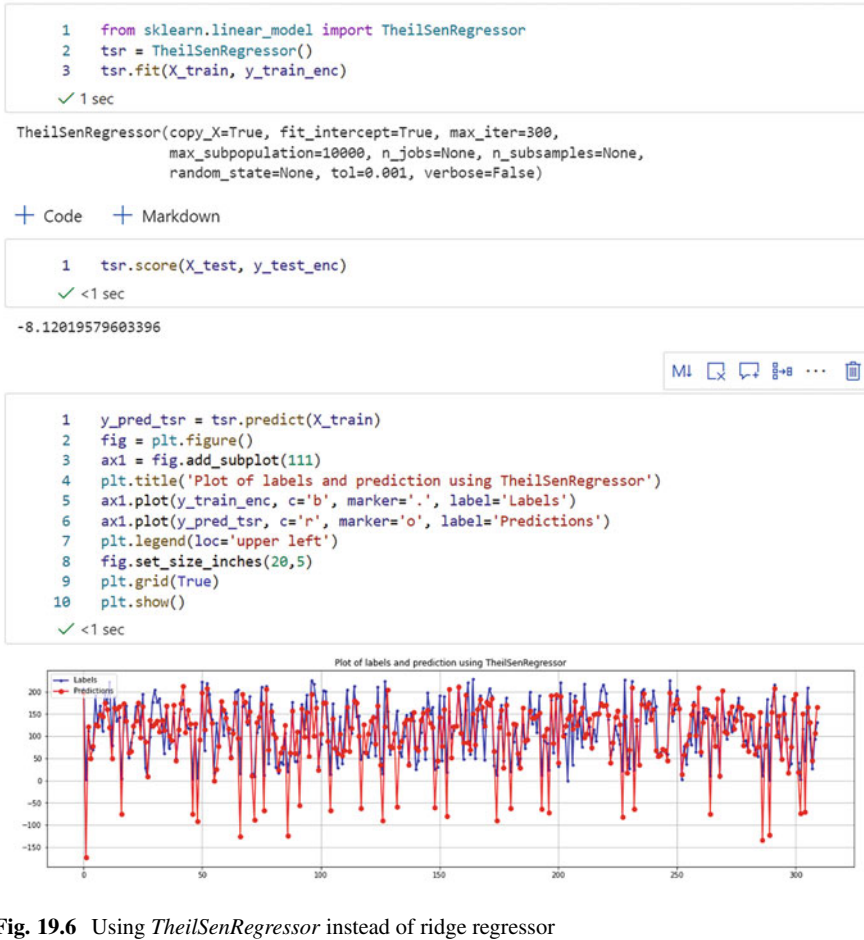


Fig. 19.5 Plotting the predictions and labels for training data



**Fig. 19.6** Using *TheilSenRegressor* instead of ridge regressor

very underperforming model providing suboptimal results. Given the current case of regression, the suitable metrics could be:

1. RMS error in predicted value
2. MA error in predicted value
3. Mean and standard deviation or variance of the absolute error
4. Max absolute error

It is important to decide the bounds on each of the metric that make the resulting model acceptable. If the metrics do not lie within the bounds, one needs to iterate the whole or partial process. Here are few possible ways to iterate:

1. Use different model.
2. Change the cross-validation policy.
3. Change, add, or remove the features.
4. Re-split the data.

## 19.5 Other Applications of Regression

The above example illustrated the details in building a regression-based application. There are many other areas where regression techniques are required. One such application of great importance marks an entire field of research, called as *nondestructive testing and evaluation* or *NDT/E*. NDT/E encapsulates all inspection-type applications of a system where the inspection needs to be performed without affecting or stopping the regular operation of the system, for example, inspection of aircrafts, inspection of various components in a nuclear power plants, or inspection of gas- or oil-carrying pipelines [84]. These inspections need to be performed without breaking open the aircraft, stopping the nuclear reactor, or stopping the pipeline from carrying any oil or gas. Various transducers are used to inspect the systems, and the inspection data is collected in digital form. Then the data is analyzed to predict any defects or flaws in the system. Most such problems pose good examples of application regression techniques.

## 19.6 Conclusion

In this chapter, we studied the real applications based on the theory of regression. This analysis gives a glimpse toward the various practical considerations one needs to go through in the form of quantitative problem definition and constrained optimization followed by certain assumptions.

## 19.7 Exercises

1. As discussed in the implementation section, use a nonlinear model to solve the regression problem, and try to improve the accuracy.
2. Use the grid search discussed in the earlier chapter to further improve the performance of linear as well as nonlinear models. Compare how much gain can be achieved in either case.
3. Here is a list of datasets available from UCI repository [49]. Choose any set of your choice, and try to solve the problems using linear and nonlinear regression algorithms. Use the learnings from the earlier examples to optimize the models using grid search. Validate the inferences drawn from the comparison between linear and nonlinear methods from earlier example.

# Chapter 20

## Ranking



### 20.1 Introduction

Ranking at heart is essentially sorting of information. Unconsciously, we are always ranking things around us based on some metrics. We rank the products based on their reviews. We rank players in various sports based on their performances in a season or over lifetime. We rank movies and music albums based on their earnings and popularity. We rank stocks based on their predicted growth or volatility, etc. In this regard, ranking should be a simple rule in basic mathematics rather than being an aspect of machine learning. However, based on their applications, ranking has become a quite popular and hot topic in machine learning and artificially intelligent systems. One of the glaring examples here would be the exponential growth of Google in the twenty-first century, which was truly based on their unique way to rank web pages.

The common use of ranking techniques is seen in a wide array of applications such as search and information retrieval problems, recommendation systems, machine translation, data mining, etc. When one queries for a string in a search engine or queries for a movie name in online streaming websites, typically there are tens, hundreds, or even thousands of results that match the query with certain margin. No user is likely going to go through all these results; hence, the order in which these results are presented to the user is important. To rank these results, one needs to have a set of quantitative measures that can be used to sort these hundreds and thousands of results. The user is then only going to look at top 5 or top 10 of them to gather the information he/she is looking for. This would make the user experience vastly superior than random presentation of the results. Ranking defines the underlying model that is capable of ordering the results in optimum manner before presenting to the user. The ranking problem fundamentally differs from the other machine learning problems, and it needs a separate set of metrics to analyze. In this chapter, we will look at different practical examples of ranking systems and also look at various techniques and metrics used in the process.

## 20.2 Measuring Ranking Performance

Subjectively, ideal ranking algorithm would be defined as:

**Definition 20.1** The algorithm that can rank the items in strictly non-increasing order of relevance

The relevance factor here is yet another non-trivial measure that needs to be defined for each ranking problem. There is no universal way to define relevance, and each problem may come with its own definition of relevance. Now, let's try to convert this subjective definition to mathematical expression. Let there be  $n$  number of items that need to be ranked and each has a relevance score of  $r_i, i = 1, 2, \dots, n$ . A simple measure, called as cumulative gain (CG), is defined based on these relevances as

$$CG = \sum_{i=1}^n r_i \quad (20.1)$$

CG essentially represents the overall quality of aggregate search results. As there is no positional factor in the expression, CG does not provide any information about ranking within the select group of results. Hence, a modified measure is defined as discounted cumulative gain or DCG. DCG is defined as

$$DCG = \sum_{i=1}^n \frac{r_i}{\log_b i + 1} \quad (20.2)$$

The base  $b$  of the logarithm is typically used as 2. The reason for using  $i + 1$  instead of  $i$  is to account for the case when  $i = 1$ . In this case, the denominator would be 0, and the metric would be meaningless. This expression penalizes the ranking if highly relevant item is ranked lower, thereby adding importance to the position of the item in the select group. In order to see the effect of this expression, let's take a real example. Table 20.1 shows a set of items with corresponding relevance scores ranging from 0.0 to 1.0. Score of 0.0 means no relevance at all, and 1.0 is the highest possible relevance; however, the maximum relevance in the given set is only 0.6. The third column in the table gives the corresponding discounted relevance score as defined in Eq. 20.2 (without summing the values).

If we sum the discounted scores, then we get DCG as 1.20. As we can see, this ranking is not ideal, as the most relevant item is at rank 7 and all the items are fairly arbitrarily ranked. Now, let's fix the rankings and recalculate the discounted relevance scores as shown in Table 20.2

Now, if we sum the discounted relevance scores, we get DCG value of 1.53, which is significantly higher than the previous score that was produced with random ordering of the items. Thus, we have a good metric in the form of DCG that can compare two different sets of rankings if we have corresponding relevance scores. However there is one slight drawback in this system and that is the presence and

**Table 20.1** Sample set of items to be ranked with relevance score. In this case, they are ranked arbitrarily

Item no.	Relevance score	Discounted relevance score
1	0.4	0.4
2	0.25	0.16
3	0.1	0.05
4	0.0	0.0
5	0.3	0.12
6	0.13	0.05
7	0.6	0.2
8	0.0	0.0
9	0.56	0.17
10	0.22	0.06

**Table 20.2** Sample set of items ideally ranked with non-increasing relevance score

Item no.	Relevance score	Discounted relevance score
1	0.6	0.6
2	0.56	0.35
3	0.4	0.2
4	0.3	0.13
5	0.25	0.1
6	0.22	0.08
7	0.13	0.04
8	0.1	0.032
9	0.0	0.0
10	0.0	0.0

position of the items with 0.0 score. If we remove the items with score 0.0, we will still have the same score. However, in reality, if we present the items to the user, the set without items with 0.0 score would be better than the one with them. However, penalizing the items with score of 0.0 without affecting the regular operation of DCG is a bit more complicated and typically not used. Also, it is important to note that this is more of a theoretical shortcoming, and results with no relevance should not be part of the select set anyway.

There is one more problem with DCG, and that is the actual value of a score is quite arbitrary based on the range of values of relevance scores. For example, if we recalibrate our relevance scores to range from 0 to 5 instead of 0–1 as before and recalculate the DCG, our ideal DCG would jump from 1.53 to 7.66, and the DCG value for the randomly ordered set would jump from 1.20 to 6.01. All these values are fairly arbitrary, and the value of 6.01 has not much meaning without knowing the ideal value of DCG, which is 7.66. Hence, another metric is introduced as normalized DCG or nDCG. nDCG is defined as

$$nDCG = \frac{DCG}{iDCG} \tag{20.3}$$

where iDCG is the value of ideal DCG.

## 20.3 Ranking Search Results and Google's PageRank

All the metrics discussed above hold true and provide a good measure of ranking quality, only if we have a single numeric measure of the relevance score for each item. However, coming up with such score is not always a trivial task. Rather the goodness of such measure is the true deciding factor in building a good ranking system.

Having better search results was the key aspect of exponential growth of Google in the first decade of this millennium. There were already many search engines available at that time, and they were not really bad. However, Google just had results that were somehow more relevant to what the users were searching for. All the search engines were crawling the same set of websites, and there was no secret database that Google was privy of. The single aspect where it performed better than the competition was the ranking of the search results. The concept of *PageRank* [19] was at the heart of Google's rankings. At the time before Google's PageRank came to existence, the commonly used technique for ranking the pages was the number of times a particular query appeared on the page. However, PageRank proposed an entirely new way of ranking the pages based on their importance. This importance was calculated based on how many other pages are referencing the given page and importance of those pages. This system created a new definition of relevance metric, and ranking based on this metric proved to be extremely effective in getting better search results, and what followed later is history!

## 20.4 Techniques Used in Text-Based Ranking Systems

Information retrieval and text mining are core concepts that underpin the ranking system that relate to searching websites or movies, etc. We will look specifically at a technique called as keyword identification/extraction and word cloud generation. This method is used in most of the text mining systems, and knowledge of these techniques is invaluable.

### 20.4.1 *Keyword Identification/Extraction*

Figure 20.1 shows a word cloud from the famous Gettysburg Address by Abraham Lincoln in 1863. A word cloud is a graphical representation of the keywords identified from a document or set of documents. The size of each keyword represents the relative importance of it. Let's look at the steps required to find these importances.





the documents become less important for a specific document compared to the words that only appear in that document. A new technique called as *TF-IDF* or *term frequency-inverse document frequency* is used to address this principle. Term frequency is calculated in similar way as described above. The inverse document frequency is the new concept, and it is based on the notion of how much important the given keyword is in the current document as opposed to other documents. Thus, a word that appears frequently across all the documents will have low value of IDF in any of these documents. However, when a keyword appears frequently in only one document, but is almost not present in the rest of the documents, then its importance in the given document is even higher. Mathematically, both the terms are defined as

$$tf = \frac{\text{frequency of the word in given document}}{\text{Maximum frequency of any word in the given document}} \quad (20.4)$$

$$idf = \frac{\text{frequency of the word in given document}}{\text{Total number of documents in which the word appears}} \quad (20.5)$$

Thus, combining *tf* and *idf*, we can have a more generic measure for keyword importance when dealing with multiple documents.

## 20.5 Implementing Keyword Extraction

Text analysis is a different field in the machine learning, and sklearn library does not support that functionality. There are many different open-source libraries available such as *spaCy* [48]. However, for the sake of demonstrating keyword extraction, we will use another library called as *yake* or *Yet Another Keyword Extractor* [47]. We will use the Azure ML workspace that we have created to demonstrate this. *Yake* does not come installed on it, so we will have to first install the library using the command `pip install yake`. Azure ML Notebook backend will take care of the installation with this command. Once the library is installed, we can use it to extract keywords.

Figure 20.2 shows how to use the *yake* library to extract keywords. For the sample text, we have chosen the first couple of paragraphs from the famous speech from Martin Luther King Jr., *I Have a Dream*. Now, we can list the ten extracted keywords as shown in Fig. 20.3. The list shows the extracted keywords along with their relative scores. The lesser the score, the higher the relevance of the keyword.

Now, let's create a word cloud for the same text. Word cloud is primarily a visualization technique and does not necessarily add to the keyword extraction process. However, it is a very powerful tool and can come handy when formulating the relevance metric. We will use another open-source library called *wordcloud* to create it. This library is installed by default in Azure ML and should be available readily. As shown in Fig. 20.4, we can create the word cloud and print it using *matplotlib*.

```

1 import yake
✓ 1 sec

1 text = """I am happy to join with you today in what will go down in history as the greatest demonstration for freedom in the
2 language = "en"
3 max_ngram_size = 3
4 numOfKeywords = 10
5 my_kwe = yake.KeywordExtractor(lan=language, n=max_ngram_size, top=numOfKeywords, features=None)
6 keywords = my_kwe.extract_keywords(text)
✓ <1 sec

```

Fig. 20.2 Demonstrating keyword extraction using yake

```

1 for kw in keywords:
2     print(kw)
✓ <1 sec

('Emancipation Proclamation', 0.0039075434928149055)
('score years ago', 0.00627328720650894)
('nation.Five score years', 0.007037662992507622)
('signed the Emancipation', 0.009684103954987788)
('hundred years', 0.015487067460567208)
('happy to join', 0.027818690528968064)
('greatest demonstration', 0.027818690528968064)
('demonstration for freedom', 0.027818690528968064)
('nation.Five score', 0.027818690528968064)
('symbolic shadow', 0.027818690528968064)

```

Fig. 20.3 Listing the extracted keywords

## 20.6 Implementing Ranking System

For the text-based rankers, the first steps would always involve some form of keyword extraction. For other rankers involving different types of media such as movies or music, different feature extraction techniques will have to be applied. Once these features are identified, we need to define the expression for relevance. The keyword scores that we computed with *yake* can form raw scores. However, the ranking metric needs to be a combination of the search query along with such raw scores. Once such metrics are identified, then all that remains is some form of sorting mechanism to generate the ranked list of results.

## 20.7 Conclusion

In this chapter, we looked at the problem of ranking and its evolution in the context of machine learning and artificial intelligence from simple ordering of items to information retrieval and resulting user experience. We studied the different measures for computing relevance and metrics for comparing different sets of ranking along with different techniques used in ranking documents.



# Chapter 21

## Recommendation Systems



### 21.1 Introduction

Recommendation system is a relatively new concept in the field of machine learning. In practical terms, a recommendation is a simple suggestion given by a friend, relative, or colleague to another friend, relative, or colleague to watch a movie or to eat in a certain restaurant. The most important thing about the recommendations is that they are very personal. What person recommends to one of his/her friends as a top choice for a restaurant may not be the same when the same person recommends it to another friend. The different recommendations can be based on the likings or preferences of the person getting the recommendations. Ideally, the person recommending should not have any ulterior motive behind the recommendations or that the recommendations must be unbiased, but in some cases, that may not be true as well. For example, if one of my good friends has started a new restaurant, then I might recommend it to my other friends irrespective of their likings. To summarize, the recommendations are dependent on the person recommending as well as the person to whom they are recommended.

In the machine learning context, a recommendation system is built for a business to promote its products or services to its customers. From this perspective, the bias for the business is intrinsically built into the system. When building a recommendation system on large scale, one must express these relationships quantitatively and formulate a theory to optimize a cost function to obtain a trained recommendation system. The core mathematical theory underlying modern recommendation systems is commonly called as collaborative filtering. Companies like Amazon and Netflix have influenced this genre of machine learning significantly for personalizing shopping and movie watching experiences, respectively. Both the systems started off with techniques almost as simple as table joins but soon evolved into sophisticated recommendation algorithms. Netflix had actually announced an open competition for developing the best performing collaborative filtering algorithm to predict user ratings for films [21]. A team called *BellKor* won the grand prize in 2009 by

achieving more than 10% better accuracy on predicted ratings than Netflix's existing algorithm.

We will first study the concepts in collaborative filtering, and then we will look at both the cases of Amazon and Netflix in the context of recommendation system and understand the nuances of each in this chapter along with learning various concepts and algorithms that are involved in building such systems.

## 21.2 Collaborative Filtering

Being quite new and cutting-edge technology, there are multiple different interpretations of the collaborative filtering based on the context and application. However, in broad sense, the collaborative filtering can be defined as a mathematical process of predicting the interests or preferences of a given user in the present or future based on a database of interests or preferences of other users and optionally historical interests and preferences of the given user. Very loosely speaking, these predictions are based on the nearest neighbor principle. Users with similar interests in one aspect tend to share similar interest in other aspects. To consider an example and oversimplifying the concept, if persons A, B, and C like the movie *Die Hard 1* and persons A and B also like the movie *Die Hard 2*, then person C is also probably going to like the movie *Die Hard 2*.

Collaborative filters always deal with two-dimensional data in the form of a matrix—one dimension being the list of users and the other dimension being the entity that is being liked or watched or purchased. Table 21.1 shows a representative table. The table is always partially filled, representing the training data. The goal is to predict all the missing values. Table 21.1 shows values that are binary, but they can as well be numeric or categorical with more than two categories. Also, you can see that the top 10 users have mostly known ratings, while bottom 10 users have mostly unknown ratings. This is a typical situation in practice, where substantial amount of data is missing, making the matrix quite sparse.

### 21.2.1 Solution Approaches

The solution of the problem defined in the previous subsection can be tackled in few different options. However, the core information that can be possibly available is of three types.

### 21.2.2 Information Types

1. Information about the users in the form of their profiles. The profiles can have key aspects like age, gender, location, employment type, number of kids, etc.

**Table 21.1** Sample training data for building a recommendation system

Persons	Electronics	Books	Travel	Household	Cars
Person1	Like	?	Dislike	Like	?
Person2	Dislike	Like	Dislike	Like	Like
Person3	Dislike	Like	Like	?	Dislike
Person4	Like	Like	Like	Like	Dislike
Person5	Like	Dislike	Dislike	Dislike	Like
Person6	?	Like	?	?	Dislike
Person7	Like	?	Dislike	Like	?
Person8	Like	Like	?	?	Like
Person9	Dislike	?	Dislike	Like	Like
Person10	Dislike	?	Like	Like	?
Person11	?	?	Dislike	?	?
Person12	?	?	?	?	?
Person13	?	?	Like	?	?
Person14	?	Like	?	?	?
Person15	?	?	?	?	Like
Person16	?	?	?	?	?
Person17	Like	?	?	?	?
Person18	?	Like	?	?	?
Person19	?	?	Dislike	?	?
Person20	Dislike	?	?	?	?

- Information about the interests. For movies, it can be in the form of languages, genres, lead actors and actresses, release dates, etc.
- Joint information of users' liking or rating their interests as shown in Table 21.1.

In some cases, one or more sets of information may not be available. So the algorithm needs to be robust enough to handle such situations. Based on the type of information used, the algorithms can be classified into three different types.

### 21.2.3 Algorithm Types

- Algorithms exploiting the topological or neighborhood information. These algorithms are primarily based on the joint historical information about the users' ratings. They use the nearest neighbor-type methods to predict the unknown ratings. These algorithms cannot work if the historical data is missing.
- Algorithms that exploit the structure of the relationships. Sometimes, they are also called as *content-based filtering*. These methods assume a structural relationship between users and their ratings on the interests. These relationships are modeled using probabilistic networks or latent variable methods like component

analysis (principal or independent) or singular value decomposition. These methods use the joint information as well as the separate information about the user profiles and interest details. These methods are better suited to tackle sparse data and ill-conditioned problems, but they miss out on neighborhood information.

3. Hybrid approach. Most recommendation systems go through different phases of operation, where in the beginning the ratings data is not available and only data that is available is user profiles and interest details. This is typically called as cold start. The neighborhood-based algorithms simply cannot operate in such situations. Hence, these hybrid systems start with algorithms that are more influenced by the structural models in the early stages and later on combine the advantages of neighborhood-based models.

## 21.3 Amazon's Personal Shopping Experience

In order to understand the problem from Amazon's perspective, we need to explain the problem in more detail. First and foremost, Amazon is a shopping platform, where Amazon itself sells products and services as well as it lets third-party sellers sell their products. Each shopper that comes to Amazon comes with some idea about the product that he/she wants to purchase along with some budget for the cost of the product that he/she is ready to spend on the product as well as some expectation of the date by which the product must be delivered. The first interaction in the experience would typically begin with searching the name or the category of the product, e.g., "digital camera." This would trigger a search experience, which would be similar to the search discussed in the previous chapter on ranking. This search would come up with a list of products that are ranked by relevance and importance of the different items that are present in Amazon's product catalog. Amazon then lets user alter the sorting by price, relevance, featured, or average customer review. These altered results are still just an extension of the ranking algorithm and not really the recommendations. The "featured" criteria typically is influenced by the advertising of the products by sellers (including Amazon). Once user clicks on the one item in the list that he/she finds interesting based on price, customer reviews, brand, etc., the user is shown the details of the product along with additional set of products gathered as *Frequently bought together* or *Continue your search* or *Sponsored products related to this item*. These sets represent the direct outcome of recommendation algorithm at work.

### 21.3.1 Context-Based Recommendation

Amazon likely has multi-million products in their catalog, but how many products should be recommended for a given user needs to have some limitation. If Amazon

starts putting a list of hundreds of products as recommended ones, the user is going to get lost in the list and may not even look at them or, even worse, move to another website. Also, if too few or no recommendations are shown, then Amazon is losing potential increase in sales of related items. Thus, an optimal balance between the two must be observed in order to maximize the sale and minimize the user distraction. The product recommendations shown here are context based and are related to the search query only. Once the number of recommendations that needs to be shown is finalized, then comes the question of which products to recommend. There can be multiple aspects guiding this decision.

### ***21.3.2 Aspects Guiding the Context-Based Recommendations***

1. Suggesting other similar products that are cheaper than the product selected. So, if the cost is the only aspect stopping the user from buying a selected item, the recommended item can solve the problem.
2. Suggesting a similar product from a more popular brand. This might attract user to buy a potentially more expensive product that is coming from more popular brand, so potentially more reliable or of better quality.
3. Suggesting a product that has better customer reviews.
4. Suggesting a set of products that are typically bundled with the selected product. These suggestions would right away increase the potential sale, e.g., suggesting carry bag or battery charger or a memory card when the selected item is a digital camera.

### ***21.3.3 Personalization-Based Recommendation***

The recommendations based on search results are not quite personal and would be similar for most users. However, the real personalization is seen when one opens the [www.amazon.com](http://www.amazon.com) the first time before querying for any specific product. The first screen that is shown to the user is heavily customized for the user based on his/her previous searches and purchases. If the user is not logged in, the historical purchases may not be available. In such cases, the browsing history or cookies can be used to suggest the recommended products. If nothing is available, then the products are shown that are in general popular among the whole set of Amazon buyers. These different situations decide which algorithm of the collaborative learning needs to be applied. For example, the last case represents the cold start situation and needs a model-based approach to be used, while in other cases, one can use a hybrid approach.



## 21.4 Netflix's Streaming Video Recommendations

Netflix was one of the pioneers of the modern-day recommendation systems, specifically through the open contest as well as being one of the first players in the online streaming genre. In early days, the ratings provided on Netflix were similar to the ones on [www.imdb.com](http://www.imdb.com) or other movie rating portals. These portals have different ways to generate these ratings. Some ratings are generated by the movie editors that specifically curate these ratings for a given site, and some ratings are created as aggregates from different newspapers or other websites, while some ratings are aggregated by user ratings. One of the most important differences here compared to Netflix is that the ratings given on websites are the same for all the people visiting those sites. These movie sites are not personalized for any user, and as a result, the ratings that are available on these websites are average ratings that might not mean much for any particular individual user other than finding some of the top rated movies of all time or from some specific genre.

Netflix's rating system differs here in the fundamental way. Each user sees a very different screen when he/she logs in to their Netflix account, in spite of Netflix having the same set of movies available for all the users to watch. Hence, the ratings or recommendations that Netflix shows to each user are personalized for that user based on their past viewing and/or search history. Netflix is not interested in having a generic set of ratings that produce top rated movies of all time or specific time in specific genre, but is more interested in catering to the specific user that is currently logged in and is interested in watching something. The recommendations have a one single motive, and that is to maximize the viewing time on Netflix.

A sample dataset that Netflix would have to use to build their recommendation system would be similar to the one shown in Table 21.2. For new users representing the bottom half of the table, there would be little to no ratings/viewings available, while for older users, there will be richer data.

## 21.5 Implementing Recommendation System

Here, we will use a popular dataset containing movie ratings by different people. The dataset is called as MovieLens [50]. Sklearn does not offer recommendation system algorithm off the shelf. There are multiple open-source libraries available, and we will use LensKit [51] to illustrate the concept.

### 21.5.1 *MovieLens Data*

Let's take a deeper look at what MovieLens data offers. The main entities in the data are as follows:

**Table 21.2** Sample training data for Netflix to build video recommendation system

Persons	Movie 1	Movie 2	Movie 3	Series 1	Series 2
	Rating, views	Rating, views	Rating, views	Rating, views	Rating, views
Person1	8 <sup>a</sup> , 2	2, 1	?, 1	7, 2	?, ?
Person2	1, 1	9, 2	4, 1	7, 1	8, 1
Person3	3, 1	6, 2	7, 1	?, 1	3, 1
Person4	9, 3	8, 2	5, 1	7, 2	1, 1
Person5	6, 1	2, 1	2, 1	4, 1	9, 3
Person6	?, 0	6, 0	?, 0	5, 1	4, 2
Person7	7, 1	?, 2	2, 3	9, 4	?, 1
Person8	10, 2	8, 2	7, 1	?, 0	8, 0
Person9	3, 1	?, 0	?, 0	2, 2	10, 0
Person10	1, 1	?, 0	8, 1	10, 5	4, 1
Person11	?, 0	?, 0	2, 1	?, 0	?, 0
Person12	?, 0	?, 0	?, 0	?, 0	?, 0
Person13	?, 0	?	8, 2	?, 0	?, 0
Person14	2, 1	5, 1	?, 0	?, 0	?, 0
Person15	?, 0	?, 0	?, 0	?, 0	7, 1
Person16	?, 0	?, 0	?, 0	?, 0	?, 0
Person17	6, 2	?, 0	?, 0	?, 0	?, 0
Person18	?, 0	9, 2	?, 0	?, 0	?, 0
Person19	?, 0	?, 0	1	1, 0	?, 0
Person20	1, 1	?, 0	?, 0	?, 0	?, 0

<sup>a</sup> Ratings are between 1 and 10. 10 being most liked and 1 being least liked

1. *Movies*: This entity contains movie-id, title, and genres. This provides more detailed information about each movie that is part of MovieLens.
2. *tags*: This entity contains user-id, movie-id, tag, and timestamp. The tag is a keyword (can contain more than one word) that is given by the corresponding user (user-id) for the given movie (movie-id). This tag is not a rating, but it provides another dimension or a feature associated with a movie.
3. *genome-scores*: This entity contains movie-id, tag-id, and relevance. This entity is not exactly a raw set of information. They have processed all the tags for all the movies and users to come up with a relevance of each tag for each movie.
4. *genome-tags*: This entity contains tag-id and tags. This entity essentially gives a numeric value to each tag that is used in the other tables.
5. *ratings*: This entity contains user-id, movie-id, rating, and timestamp. This is one of the most important pieces of information that connects all the users with all the movies that they have rated giving the actual rating value from 0.0 to 5.0.
6. *links*: This entity contains movie-id, imdb-id, and tmdb-id. This entity connects each movie in MovieLens database with external movie databases *IMDB* and *The Movie Database*.

## 21.5.2 *Planning the Recommendation Approach*

We can safely ignore the *links* data as we don't want to augment the MovieLens data with additional information from IMDB and The Movie Database. We can also ignore the *genome-scores* entity to stay focused on the raw information. That leaves us with *Movies*, *tags*, and *ratings*.

### 21.5.2.1 **Movies**

Movies entity provides information about its genres. So, we can potentially use this information to recommend a movie with a genre(s) to a user that likes movies with similar genre(s). We can start with creating a 2D matrix from this entity with the first dimension as movies and the second dimension as genres. When a movie is associated with a genre, the corresponding value in the table is 1; else, it is 0. Then we can use clustering algorithms like *KNN* to find similar movies. For a given user, we can find similar movies for each of the movie that he/she has liked and then combine them to create a final list of recommendations.

### 21.5.2.2 **Tags**

We can potentially proceed in a very similar manner described for the case of *Movies* entity by ignoring the user-id and creating a 2D matrix with the first dimension as movies and the second dimension as tags. Then we can use *KNN* in a similar manner to create a recommendation list. However, this table contains additional information in the form of user-ids and timestamps.

### 21.5.2.3 **Ratings**

The above two methods would provide a list of recommendations, but those are a little bit indirect ways of predicting them as we are not explicitly using the users' interaction with the movies. When we use the ratings entity, we are making the recommendations more personal. One way to use this table is following a similar approach described above—converting the ratings table into a 2D matrix with one dimension as users and the other dimension as movies. When we have a rating by a user for a movie, the corresponding value in the matrix is the value of rating. If the rating does not exist for the user-movie combination, the value is *N/A*. It is important to note that the value of missing ratings is not 0.0 but *N/A*, as 0.0 value means a very poor rating, and that is not the intention here. Once this matrix is created, we can use the same *KNN*-based approach to generate recommendations.

## 21.5.3 *Implementing Recommendation System*

### 21.5.3.1 K-Nearest Neighbors

The MovieLens data itself is not that big, but once we unfold it into 2D matrices as described above, the size of the matrices can grow quite big. For example, there are over 25 million ratings in the *ratings* entity, and there are 162K users, making the 2D matrix of size over 4 trillion. Allocating memory for this size of data is possible, but it goes beyond the resources available in the free version of Google Colab or even the initial tiers of compute available in Azure ML. Typically sparse matrices available through *SciPy* library are used in this situation to use the available memory in optimal manner. The Python implementation of this code is quite lengthy, and we will defer it in this book. With the approach outlined above, a reader should be able to implement the algorithm. To further improve the results from any one dataset, we can combine the rankings from multiple recommenders to generate a joint system of weighted recommendations.

### 21.5.3.2 Algebraic Least Squares (ALS) Algorithm

KNN is a valid approach and also gives reasonable results with the recommendations; however, there exists a much better option known as *algebraic least squares* or *ALS*. This algorithm came into limelight through the Netflix competition. ALS is a core mathematical algorithm that factorizes a matrix into two components. Let's consider an example of Movie rating type of matrix with  $N$  users and  $M$  movies. The rating matrix will have a shape of  $N \times M$ . ALS can factorize this matrix into two matrices (let's denote them as  $U$  and  $R$ ), such as  $U$  is of shape  $N \times C$  and  $R$  is of shape  $C \times M$ . We can think of this factorization as extracting  $C$  number of features for users in matrix  $U$  and extracting  $C$  number of features for movies in matrix  $R$ . These features are mutually dependent, such that when we choose a specific user and take a dot product of his/her features from the corresponding row with the features of a movie with its corresponding column, we are essentially predicting the rating for the movie from that user. Thus, the matrix factorization has now separated the information about the users from movies in the context of ratings. Once we have the matrix factors, we are ready to make any predictions. We can use the non-negative matrix factorization [52] functionality from sklearn for this purpose. We can choose and experiment with the number of components  $C$  to get desired results. The performance of the recommended list of movies can be measured with nDCG type of metrics for ranking.

## 21.6 Conclusion

In this chapter, we studied the collaborative filtering technique. We then looked at the specific cases of recommendation systems that are deployed by Amazon and Netflix. We also looked at a real-life example of MovieLens data to learn how we can implement recommendation systems using off-the-shelf libraries. Recommendation systems represent a relatively new entry into the field of machine learning, and it is a cutting edge of the area and is under strong development.

## 21.7 Exercises

1. Implement the KNN approach for recommendations using only Movies entity. You will have to split the different genres into different dimensions using one-hot encoding technique before applying KNN.
2. Implement the KNN approach using tags entity in similar manner described above.
3. Implement the KNN approach for recommendations using rankings entity. Use the approach described in the first problem.
4. Join the three recommendation systems using weights. Try to optimize the weights to improve the nDCG of the joint recommender.
5. Implement ALS-based recommender for all the three entities, and create a joint recommender. Compare the joint recommender accuracy with the one obtained with KNN model.

# Part V

## Conclusion

Do or do not, there is no try!!

—Yoda, “Star Wars: The Empire Strikes Back”

### **Part Synopsis**

This part concludes the book with key takeaways from the book and road ahead.

# Chapter 22

## Conclusion and Next Steps



### 22.1 Overview

The concepts of machine learning and artificial intelligence have become quite mainstream and the focus of a lot of cutting-edge research in science and technology. Although most of the concepts in machine learning are derived from the mathematical and statistical foundations that are quite hard to understand without having background in graduate-level courses in those topics, the artificially intelligent applications based on machine learning are becoming quite commonplace and accessible to anyone. This simplicity of applications has opened the doors to these topics for a huge number of engineers and analysts, which otherwise were only accessible to people doing their PhDs. This is good and bad in a way. This is *good* because these topics do have applications in technology that is used in every little aspect of our modern lives. The people who are building all this technology are not necessarily the people who are pursuing their PhDs in mathematics and statistics. Hence, when more people are armed with these powerful techniques, the faster is the pace of evolution of technology and revolution of lives of all the people. This is *bad* because oversimplification of the concepts can lead to misunderstanding and that can have catastrophically worse if not dangerous effects.

I have come across many colleagues and friends who had asked questions about different problems or applications of machine learning in day-to-day life. In spite of having a plethora of information on these topics on the Internet, in most cases, it creates more confusion than answering the questions. Many books, Wikipedia, and many other websites are dedicated on this topic. However, in my investigation, the material presented in most sources is focused on a small subset of topics and typically either goes too deep in theory or stays too superficial that leaves the readers searching for more clarity. Most of these sources are useful when you know exactly what you are looking for, but unfortunately, in most cases, that is not true. In such case, it is easy to get lost in the ocean of such information.

In recent years, the areas of machine learning and data science have emerged as stand-alone topics for graduate (Master's)-level courses. I have specifically organized this book to serve as a single reference textbook needed to create a course on the topic that can span one or two semesters.

Each individual aspect of this technology when looked in micro detail starts to seem obvious, trivial even. However, when all these individual components come together, the problem that is solved and the experience that is delivered are just magical. And that is what keeps me excited about this topic every single day. Hope to have passed this excitement through this book to every student!



# References

Here are the book wide references.

1. The Human Memory <http://www.human-memory.net/brainneurons.html>
2. Wikipedia - Deep Learning <https://en.wikipedia.org/wiki/Deeplearning>
3. Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository <https://archive.ics.uci.edu/ml/datasets/iris>. Irvine, CA: University of California, School of Information and Computer Science.
4. Wikipedia - Dynamic Programming Applications [https://en.wikipedia.org/wiki/Dynamicprogramming#Algorithms\\_that\\_use\\_dynamic\\_programming](https://en.wikipedia.org/wiki/Dynamicprogramming#Algorithms_that_use_dynamic_programming)
5. Wikipedia - Linear discriminant analysis <https://en.wikipedia.org/wiki/Lineardiscriminantanalysis>
6. UCI - Machine Learning Repository for Machine Learning and Intelligent Systems) - Adult Data Set <https://archive.ics.uci.edu/ml/datasets/Adult>
7. Mobile cellular subscriptions (per 100 people) <https://data.worldbank.org/indicator/IT.CEL.SETS.P2?end=2017&start=1991>
8. Convolution Theorem <https://en.wikipedia.org/wiki/Convolutiontheorem>
9. Diminishing Returns <https://en.wikipedia.org/wiki/Diminishingreturns>
10. Shannon number [https://en.wikipedia.org/wiki/Shannon\\_number](https://en.wikipedia.org/wiki/Shannon_number)
11. Deep Blue (chess computer) [https://en.wikipedia.org/wiki/Deep\\_Blue\(chess\\_computer\)](https://en.wikipedia.org/wiki/Deep_Blue(chess_computer))
12. Setting up Mario Bros. in OpenAI's gym <https://becominghuman.ai/getting-mario-back-into-the-gym-setting-up-super-mario-bros-in-openais-gym-8e39a96c1e41>
13. Open AI Gym <http://gym.openai.com/>
14. Summit Supercomputer <https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/>
15. Travelling Salesman Problem <https://en.wikipedia.org/wiki/Travellingsalesmanproblem>
16. Mahalanobis distance <https://en.wikipedia.org/wiki/Mahalanobisdistance>
17. Causality in machine learning <http://www.unofficialgoogledatascience.com/2017/01/causality-in-machine-learning.html>
18. A Brief History of Machine Learning Models Explainability <https://medium.com/@Zelros/a-brief-history-of-machine-learning-models-explainability-f1c3301be9dc>
19. PageRank <https://en.wikipedia.org/wiki/PageRank>
20. Self Organizing Maps <https://en.wikipedia.org/wiki/Self-organizingmap>
21. Netflix Prize <https://www.netflixprize.com/rules.html>
22. Apache Hadoop [https://en.wikipedia.org/wiki/Apache\\_Hadoop#cite\\_note-22](https://en.wikipedia.org/wiki/Apache_Hadoop#cite_note-22)
23. scikit-learn <https://en.wikipedia.org/wiki/Scikit-learn>
24. sklearn-knn <https://scikit-learn.org/stable/modules/classes.html#module-sklearn.neighbors>

25. Biological Neuron <https://en.wikipedia.org/wiki/Biologicalneuronmodel>
26. Real World Datasets from Sklearn <https://scikit-learn.org/stable/datasets/realworld.html>
27. Notable AlphaZero Games <https://www.chessgames.com/perl/chessplayer?pid=160016>
28. Genetic Programming API reference <https://gplearn.readthedocs.io/en/stable/reference.html>
29. Particle Swarm Optimization Library <https://pyswarms.readthedocs.io/en/latest/>
30. Time Series Made Easy in Python - Darts [https://unit8co.github.io/darts/generated\\_api/darts.html](https://unit8co.github.io/darts/generated_api/darts.html)
31. Time series forecasting with Prophet <https://facebook.github.io/prophet/docs/quickstart.html#python-api>
32. Instructions per second <https://en.wikipedia.org/wiki/Instructionspersecond>
33. Generative Pre-Trained Transformer 3 <https://en.wikipedia.org/wiki/GPT-3>
34. BERT (language model) [https://en.wikipedia.org/wiki/BERT\(language\\_model\)](https://en.wikipedia.org/wiki/BERT(language_model))
35. TensorFlow Overview <https://www.tensorflow.org/overview/>
36. Minkowski Distance <https://en.wikipedia.org/wiki/Minkowskidistance>
37. Manhattan Distance <https://en.wikipedia.org/wiki/Taxicabgeometry>
38. sklearn-som library <https://pypi.org/project/sklearn-som/>
39. Text Featurization in sklearn <https://scikit-learn.org/stable/modules/classes.html#module-sklearn.featureextraction.text>
40. DateTime Featurization <https://docs.python.org/3/library/datetime.html>
41. Expectation Maximization Algorithm <https://en.wikipedia.org/wiki/Expectation-maximizationalgorithm>
42. Sample Size Calculator <https://www.calculator.net/sample-size-calculator.html>
43. Metrics and Scoring <https://scikit-learn.org/stable/modules/modevaluation.html>
44. UCI Spam detection database <https://archive.ics.uci.edu/ml/datasets/spambase>
45. The CIFAR-10 dataset <https://www.cs.toronto.edu/~kriz/cifar.html>
46. Real Estate Valuation Dataset <https://archive.ics.uci.edu/ml/datasets/Real+estate+valuation+data+set>
47. Yet Another Keyword Extractor <https://github.com/LIAAD/yake>
48. Industrial Strength Natural Language Processing <https://spacy.io/>
49. Regression problems in UCI Repository <https://archive.ics.uci.edu/ml/datasets.php?format=&task=reg&att=&area=&numAtt=&numIns=&type=&sort=nameUp&view=table>
50. MovieLens <https://grouplens.org/datasets/movielens/>
51. LensKit Getting Started <https://lkipy.readthedocs.io/en/stable/GettingStarted.html>
52. Matrix Decomposition <https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.NMF.html>
53. Keras Reinforcement Learning <https://github.com/keras-rl/keras-rl>
54. Open AI Gym <https://gym.openai.com/>
55. Vladimir N. Vapnik, *The Nature of Statistical Learning Theory*, 2nd edn. (Springer, New York, 1995).
56. Richard Bellman, *Dynamic Programming*, (Dover Publications, Inc., New York, 2003).
57. Trevor Hastie, Robert Tibshirani, Jerome Friedman *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd edn. (Springer, New York, 2016).
58. Joseph Giarratano, Gary Riley, *Expert Systems: Principles and Programming*, PWS Publishing Company, 1994.
59. Olivier Cappé, Eric Moulines, Tobias Rydén, *Inference in Hidden Markov Models* (Springer, New York, 2005).
60. Richard O. Duda, Peter E. Hart, David G. Stork, *Pattern Classification* John Wiley and Sons, 2006.
61. Geof McLachlan, Kaye Basford *Mixture Models: Inference and Applications to Clustering* Vol. 38, M. Dekker, New York 1988.
62. Quinlan, J. R. "Induction of Decision Trees", *Mach. Learn.* 1, 1 (Mar. 1986), 81–106
63. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio *Generative Adversarial Nets*, NIPS, 2014.

64. Frank Rosenblatt, *The Perceptron - a perceiving and recognizing automation*, Report 85-460, Cornell Aeronautical Laboratory, 1957.
65. Zafer CÖMERT and Adnan Fatih KOCAMAZ, *A study of artificial neural network training algorithms for classification of cardiocography signals*, Journal of Science and Technology, Y 7(2)(2017) 93–103.
66. Babak Hassibi, David Stork, Gregory Wolff, Takahiro Watanabe *Optimal brain surgeon: extensions and performance comparisons*, NIPS, 1993.
67. Yann LeCun, John Denker, Sara Solla, *Optimal Brain Damage*, NIPS 1989.
68. Tin Kam Ho, *Random Decision Forests*, Proceedings of the 3rd International Conference on Document Analysis and Recognition, Montreal, QC, 14–16 August 1995. pp. 278–282.
69. Leo Breiman, *Random Forests*, Machine learning 45.1 (2001): 5–32.
70. Leo Breiman, *Prediction Games and ARCing Algorithms*, Technical Report 504, Statistics Department, University of California, Berkeley, CA, 1998.
71. Yoav Freund, Robert Schapire *A Short Introduction to Boosting*, Journal of Japanese Society for Artificial Intelligence, 14(5):771-780, September, 1999.
72. V. N. Vapnik and A. Y. Lerner *Pattern Recognition using Generalized Portraits* Automation and Remote Control, 24, 1963.
73. Haohan Wang, Bhiksha Raj *On the Origin of Deep Learning*, ArXiv e-prints, 2017.
74. Geoffrey Hinton, Simon Osidero, Yee-Whye Teh *A fast learning algorithm for deep belief networks*, Neural Computation, 2006.
75. Kunihiko Fukushima, *Neocognition: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position* Biological cybernetics, 36(4), 193-202, 1980.
76. Michael I Jordan, *Serial order: A parallel distributed processing approach*. Advances in psychology, 121:471-495, 1986.
77. Vinod Nair, Geoffrey Hinton *Rectified Linear Units Improve Restricted Boltzmann Machines*, 27th International Conference on Machine Learning, Haifa, Isreal, 2010.
78. Sepp Hochreiter, Jürgen Schmidhuber *Long Short-Term Memory* Neural Computation, vol-9, Issue 8, 1997.
79. David Wolpert, William Macready, *No Free Lunch Theorems for Optimization*, IEEE Transactions on Evolutionary Computation, Vol.1, No.1, April, 1997.
80. David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dhharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, Demis Hassabis, *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, ArXiv e-prints, Dec 2017.
81. Jamie Shotton, Toby Sharp, Pushmeet Kohli, Sebastian Nowozin, John Winn, Antonio Criminisi, *Decision Jungles: COmpact and Rich Models for Classification*, NIPS 2013.
82. Olivier Chapelle, Jason Weston, Leon Bottou and Vladimir Vapnik, *Vicinal Risk Minization*, NIPS, 2000.
83. Vladimir Vapnik, *Principles of Risk Minimization for Learning Theory*, NIPS 1991.
84. Ameet Joshi, Lalita Udpa, Satish Udpa, Antonello Tamburrino, *Adaptive Wavelets for Characterizing Magnetic Flux Leakage Signals from Pipeline Inspection*, IEEE Transactions on Magentics, Vol.42, No.10, October 2006.
85. Zhe Gan, P. D. Singh, Ameet Joshi, Xiaodong He, Jianshu Chen, Jianfeng Gao *Character-level deep conflation for business data analytics*, IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). IEEE, 2017.
86. G. A. Rummery, Mahesh Niranjan *On-Line Q-Learning using Connectionist Systems*, volume 37. University of Cambridge, Department of Engineering.
87. S. Kirkpatrick, C.D. Gelatt Jr., M.P. Vecchi, *Optimization by Simulated Annealing*, Science, New Series, Vol. 220, No. 4598, 1983.
88. Craig W. Reynolds *Flocks, Herd and Schools: A Distributed Behavioral Model*, Computer Graphics, 21(4), July 1987, pp 25-34.
89. Lafferty, J., McCallum, A., Pereira, F. *Conditional random fields: Probabilistic models for segmenting and labeling sequence data*. Proc. 18th International Conf. on Machine Learning, Morgan Kaufmann. pp. 282–289, 2001.

90. Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung, *The Google File System*, Proc. 19th ACM Symposium on Operating Systems Principles, pp. 20-43, 2003.
91. J J Hopfield, *Neural networks and physical systems with emergent collective computational abilities*, Proceedings of the National Academy of Sciences. pp. 79(8) 2554–2558, 1982.
92. Zhang, Changzheng, Xiang Xu, and Dandan Tu. *Face detection using improved faster rcnn*, arXiv preprint arXiv:1802.02142 (2018).
93. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan Gomez, Lukasz Kaiser, Illia Polosukhin, *Attention Is All You Need*, 31st Conference on Neural Information Processing Systems (NIPS), 2017.
94. Byrd, R.H., Chin, G.M., Nocedal, J. et al. *Sample size selection in optimization methods for machine learning* Mathematical Programming 134, 127–155 (2012)
95. Singh, Ajay S, Masuku, Micah B, *Sampling Techniques & Determination of Sample Size in Applied Statistics Research: An Overview*, International Journal of Economics, Commerce and Management, Oct 18, 2011

# Index

## A

A/B testing, 221  
Accuracy, 12, 32, 59, 64, 66, 68, 71, 72, 74, 83, 86, 93, 95, 97, 99, 107, 108, 138, 148, 150, 161, 166–169, 206, 208, 209, 211, 212, 216, 220, 228, 230–234, 237, 238, 242, 252, 260  
Activation function, 61–63, 65, 67, 69, 71, 156, 165, 166, 178  
Adaboost, 85–87  
Adult salary data, 193, 200, 202, 203, 206  
Algorithm, 5, 8, 24, 31, 45, 62, 73, 89, 101, 119, 129, 139, 150, 171, 183, 201, 225, 242, 244, 251  
Amazon, 5, 8, 31, 125, 140, 141, 251, 252, 254–255, 260  
Ant colony optimization, 136–137  
Applications, vii–x, 1, 3, 5–7, 18, 19, 21–23, 26, 45, 62, 65, 74, 75, 81, 87, 108, 110, 112, 119, 122, 124–125, 127, 129, 134, 135, 139, 146, 151, 153, 157, 158, 160, 161, 169, 180, 186, 201, 202, 204, 218, 219, 222, 223, 225, 232, 235, 242, 243, 252, 263  
Areas under curve, 219  
Artificial intelligence (AI), vii–x, 3–21, 23, 57, 151, 192, 249, 263  
AUC, 219  
Autoencoders, 178  
Autoregression, 141  
Autoregressive integrated moving average (ARIMA), 143–145, 210  
Autoregressive moving average (ARMA), 141, 143–145  
Azure Machine Learning (AML), 31, 33–41

## B

Bagging, 83–85c  
Bayesian approach, 102–106  
Bayesian networks, 108, 109  
Bellman equation, 119, 120  
Bernoulli distribution, 112–113  
Big data, 7–8, 20  
Binomial distribution, 113–114, 116  
Boosting, 83, 85, 86

## C

Categorical features, 188–189, 193–200, 202, 206, 211  
Causality, 185–186, 210–212  
Chi-Squared Automatic Interaction Detector (CHAID), 75, 81–82  
Classical neural network, 157, 160, 169  
Classification, 9, 10, 22, 23, 25, 28, 45, 51, 54, 56, 59, 62, 66, 71, 72, 75, 77–81, 85, 86, 89–92, 94, 99, 127, 134, 151, 156, 157, 160, 163, 164, 176, 180, 184, 185, 202, 204, 206–208, 210, 215, 216, 218, 222, 225–235  
Classification and regression tree (CART), 75, 78, 82, 83  
Clustering, 9, 45, 172–177, 180, 193, 201, 202, 258  
Coincidence and causality, 211–212  
Colaboratory/Colab, 32–33, 37, 39–41, 55, 59, 60, 66, 69, 87, 94, 95, 98, 107, 127, 134, 135, 138, 163, 180, 200, 206, 259  
Cold start, 254, 255  
Collaborative filtering, 251–254, 260

Conclusion, ix, 5, 6, 19, 29, 40–41, 55, 71, 87, 99, 117–118, 127, 138, 147, 169, 180, 200, 212, 221–222, 233, 234, 242, 249, 260, 261, 263–264

Conditional random fields (CRFs), 146–147

Confusion matrix, 218

Convolutional neural networks (CNNs), 152–157, 160, 163, 165, 166

Correlation, 153, 185–186, 194, 198, 200, 211

Cumulative gain (CG), 244

Curse of dimensionality, 12

## D

Data leakage, 210–212

Data preprocessing, 21, 212

Data science, 23, 264

Data skew, 228–229

Data understanding, 21–29

Decision tree, 13, 73–87, 99, 202, 232

Deep learning, 108, 126, 127, 129, 144, 149–169

Deep networks, 58, 151

Dimensionality reduction, 26, 177, 178

Discounted cumulative gain (DCG), 244, 245

Discriminative models, 101–106

Dynamic learning, 10–11

Dynamic programming, 119–127

## E

Early trends in machine learning, 18–19

EM algorithm, 108, 193

Ensemble decision trees, 83–86

Entropy, 77, 80–82, 166

Evolutionary algorithms, 129–138

Expectation maximization (EM), 108, 193

Expert systems, 18, 19, 119, 124, 151

Explainability, 209, 212

Exploration and exploitation, 122–123, 126, 127, 136, 137

## F

Feature engineering, 18, 183, 229, 233, 237

Featurization, 183–200, 206, 207

Feedforward operation, 61–62, 64

F-score, 217

## G

Gamma distribution, 114–116

Gaussian distribution, 67, 107, 110–112

Generalized linear models (GLMs), 46, 50–52

Generative models, 98, 101, 102, 108–109, 146, 161

Genetic algorithms, 137

Gini index, 77–80, 82

Google, 5, 8, 31–33, 39, 124, 127, 152, 163, 243, 246

Gradient-boosted trees, 86

## H

Hidden Markov models (HMMs), 121, 145–146

Hyperparameter tuning, 99, 160, 202, 205–206, 212, 230, 234

Hypothesis testing, 219–221

## I

Implementations, viii, ix, 6, 7, 18, 31, 40, 58, 64, 68, 74, 86, 87, 94, 127, 133, 137, 144, 160–162, 176, 188, 189, 242, 259

Independent Component Analysis (ICA), 177

Internet of things (IoT), 21

Iris data, 22–24, 59, 60, 65, 66, 69, 72, 86, 94, 99, 107, 136, 138, 139, 176, 225

Iterative Dichotomiser 3 (ID3), 75, 82, 83

Iterative optimization, 120

## J

Jupyter, 32, 37, 39

## K

Kernel methods, 95

*k*-means clustering, 172–177

K-nearest neighbor (KNN) algorithm, 52–55, 174, 258, 259

## L

Lasso, 50, 71

Linear, 11, 24, 45, 58, 74, 89, 129, 141, 154, 178, 210, 237

Linear discriminant analysis (LDA), 24, 27–29

Linearity, 13, 45, 46, 51

Link function, 51, 210

Long short term memory (LSTM), 158–160

## M

Machine learning (ML), 3, 7, 21, 31, 45, 57, 73, 94, 108, 121, 129, 139, 149, 171, 183, 201, 213, 225, 235, 243, 251, 263

Maximum likelihood estimation (MLE), 47, 102, 104–106

Mean squared error (MSE), 215

Metrics, 4, 71, 78, 80–84, 166, 172, 201, 204, 206, 208, 209, 213–219, 221, 222, 226,

- 229, 232, 235, 239, 241, 243–246, 248–250, 259
- Microsoft, 5, 6, 8, 31, 33, 40, 41, 140, 226
- Mixture methods, 108
- ML pipeline, ix, 29, 38, 108, 181, 206–208, 212, 225
- Moving average (MA), 141–145
- Multilayer perceptron (MLP), 61–68, 71, 104, 156, 157
- Mutation, 131–134
  
- N**
- Natural selection, 131, 132
- nDCG, 245, 259, 260
- No free lunch theorem, 18
- Nonlinearity, 13, 51, 65, 96–97, 154
- Notebook, 32, 37–39, 41, 248
- Numerical features, 183, 187–188, 194, 198, 202, 229
  
- O**
- Occam’s razor, 13–18
- Overfitting, 48, 49, 69–71, 77, 83, 86, 98, 155
  
- P**
- PageRank, 246
- Perceptron, 57–72, 169
- Performance measurement, ix, 213–222
- Poisson distribution, 116–117
- Precision, 216–219, 227, 228
- Precision recall trade-off, 227–228
- Principal component analysis (PCA), 24–28, 177, 178, 180
- Python, 5, 19, 31, 32, 37, 39, 48, 49, 144–145, 192, 230, 259
  
- Q**
- Q-learning, 126, 127
  
- R**
- Radial basis function networks, 67–69
- Random forest, 83–85, 87, 202, 205, 207, 230–232, 234
- Ranking, 26, 243–250, 254, 259, 260
- Ranking metrics, 249
- Real estate price prediction, 236
- Recall, 216–219, 228
- Recommendation systems, 243, 251–260
- Recurrent neural network (RNNs), 151, 152, 157–160
- Regression, 13, 25, 45–51, 54, 56, 59, 62, 71–73, 75–80, 83, 85, 87, 99, 146, 156, 200, 202, 211, 212, 214, 222, 235–242
- Regularization, 48–50, 70, 71, 86, 90, 92–94, 98, 199, 237
- Reinforcement learning, 9, 10, 119–127, 139
- Ridge, 49–50, 71, 72, 237, 240, 241
- Risk minimization, 98
- ROC curve, 218–220, 228
- Root mean squared error (RMSE), 215
  
- S**
- Sagemaker, 31
- scikit, 32
- Self organizing maps (SOMs), 172, 178, 179
- Simulated annealing, 137–138
- Sklearn, 32, 56, 59, 60, 64–66, 68, 69, 72, 77, 87, 94, 95, 98, 99, 107, 126, 134, 144, 162, 176, 180, 188, 189, 191, 206, 207, 222, 230–233, 238, 248, 250, 256, 259
- Spam detection, 225–229, 233
- State-action-reward-state-action (SARSA), 126
- Static learning, 10
- Stationarity, 140–141, 143, 144, 147
- Stochastic learning, 64
- Stratified sampling, 204–205, 229
- String features, 24, 187, 189–191
- Supervised, 9, 10, 121, 129, 171, 174, 178, 180
- Supervised learning, 9, 45, 46, 123, 127, 229
- Support vector machine (SVM), 13, 67, 74, 89–99, 202
- Swarm intelligence, 134–136
  
- T**
- Techniques, 6, 7, 21, 31, 48, 70, 75, 92, 121, 135, 139, 151, 177, 183, 201, 213, 229, 235, 243, 251, 263
- Textbook, ix, 225, 227, 264
- TF-IDF, 191, 248, 250
  
- U**
- Unknown categories, 211
- Unsupervised, 10, 45, 121, 139, 171, 177, 178, 180, 229
- Unsupervised learning, 9, 10, 45, 46, 56, 122, 171–180
  
- V**
- Visualization, 1, 21–29, 78, 118, 193, 200, 248
  
- W**
- What is AI, 4
- What is ML, 4