

Ankit Dixit

Ensemble Machine Learning

A beginner's guide that combines powerful machine learning algorithms to build optimized models



Packt>

Ensemble Machine Learning

A beginner's guide that combines
powerful machine learning algorithms
to build optimized models

Ankit Dixit



BIRMINGHAM - MUMBAI

Ensemble Machine Learning

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the

information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2017

Production reference: 1191217

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78829-775-2

www.packtpub.com

Credits

Author

Ankit Dixit

Reviewers

Apeksha Jain

Radovan Kavicky

Commissioning Editor

Sunith Shetty

Acquisition Editor

Viraj Madhav

Content Development Editor

Aishwarya Pandere

Technical Editor

Suwarna Patil

About the Author

Ankit Dixit is a data scientist and computer vision engineer from Mumbai. Ankit has studied BTech in biomedical engineering and has a master's degree in computer vision specialization. He has worked in the field of computer vision and machine learning for the past 6 years. He has worked with various software and hardware platforms for the design and development of machine vision algorithms. Ankit has experience with a wide variety of machine learning algorithms. Currently, he is focusing on designing computer vision and machine learning algorithms for medical imaging data, with the use of various advanced technologies such as ensemble methods and deep learning-based models.

About the Reviewers

Apeksha Jain is a data scientist and computer vision engineer from Mumbai, India. She holds a BTech in biomedical engineering and has a master's degree in computer vision specialization. She has been working in the field of computer vision and machine learning for more than 6 years. She has used various software and hardware platforms for the design and development of machine vision algorithms, and has experience on various machine learning algorithms, including deep learning. Currently, she is working on designing computer vision and machine learning algorithms for medical imaging data for Aditya Imaging and Information Technologies (part of the Sun Pharmaceutical advanced research center), Mumbai. She does this with the use of various advanced technologies such as ensemble methods and

deep learning-based models.

Radovan Kavicky is the principal data scientist and president at GapData Institute, based in Bratislava, Slovakia, where he harnesses the power of data and wisdom of economics for public good. He is a macroeconomist by education, and consultant and analyst by profession (8+ years of experience in consulting for clients from the public and private sectors), with strong mathematical and analytical skills. He is able to deliver top-level research and analytical work. From MATLAB, SAS, and Stata, he switched to Python, R, and Tableau.

Radovan is an evangelist of open data and a member of the Slovak Economic Association (SEA), Open Budget Initiative, Open Government Partnership, and the global Tableau #DataLeader network (2017). He is the founder of PyData Bratislava, R <- Slovakia, and the SK/CZ Tableau User Group (skczTUG). He has been a speaker at @TechSummit (Bratislava, 2017) and

@PyData (Berlin, 2017).

You can follow him on Twitter at @radovankavicky, @GapDataInst, or @PyDataBA. His full profile and experience are available at <https://www.linkedin.com/in/radovankavicky/> and <https://github.com/radovankavicky>.

GapData Institute: <https://www.gapdata.org>.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/178829775X>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

What this book covers

What you need for this book

Who this book is for

Conventions

Reader feedback

Customer support

Downloading the example code

Errata

Piracy

Questions

1. Introduction to Ensemble Learning

What is ensemble machine learning?

The purpose of ensemble machine learning

How to create an ensemble system

Quantification of performance

Bias and variance errors

Methods to create ensemble systems

Bagging

Boosting

Stacking

Summary

2. Decision Trees

How do decision trees work?

ID3 algorithm for decision tree building

Root node

Salary

The Sex attribute

Marital status

Parent node

Choosing between the Sex and Marital attributes for the low salary group

Choosing between the Sex and Marital attributes for the Med salary group

Marital status

Case study – car evaluation problem

Summary

3. Random Forest

Classification and regression trees

Gini index for impurity check

Node selection

Creating a split

Tree building

At depth – 1 (root node)

At depth – 2 (left branch)

At depth – 2 (right branch)

Case study – breast cancer type prediction

Decision tree bagging

From bagging to random forest

Summary

4. Random Subspace and KNN Bagging

Subspace bagging

Case study – subspace bagging

More information about the dataset

KNN classification

KNN for spam filtering

Dataset

Dataset information

Attribute information

KNN bagging with random subspaces

Summary

5. AdaBoost Classifier

Boosting

AdaBoost in a nutshell

Weak classifier

AdaBoost in action

Application of the AdaBoost classifier in face detection

Face detection using Haar cascades

Integral image

Implementation using OpenCV

Summary

6. Gradient Boosting Machines

Gradient Boosting Machines

What is the difference?

Create split

Node selection

Build tree

Regression tree as a classifier

GBM implementation

Algorithm

Improvements to basic gradient boosting

Tree constraints

Weighted updates

Stochastic gradient boosting

Penalized gradient boosting

Summary

7. XGBoost – eXtreme Gradient Boosting

XGBoost – supervised learning

Models and parameters

Objective function – training loss + regul

arization

Why introduce the general principle?

XGBoost features

Model features

System features

Algorithm features

Why use XGBoost?

- XGBoost execution speed
- Model performance
- How to install
 - Building the shared library
 - Building on Ubuntu/Debian
 - Building on Windows
 - A trick for easy installation on a Windows machine
- XGBoost in action
 - Dataset information
 - Attribute information
- XGBoost parameters
 - General parameters
 - Booster parameters
 - Learning task parameters
 - Parameter tuning – number and size of decision trees
 - Problem description – Otto dataset
 - Tune the number of decision trees in XGBoost
 - Tuning the size of decision trees in XGBoost
 - Tuning the number of trees and max depth in XGBoost
- Summary

8. Stacked Generalization

Stacked generalization

Submodel training

KNN classification

Distance calculation (Euclidean)

Estimating the neighbors

Making predictions using voting

Perceptron

Training the perceptron

Gradient descent

Stochastic gradient descent

Implementation of perceptron

Logistic regression

The logistic function

Representation of logistic regression

Modeling probability using logistic regression

Learning the model

Prediction using logistic regression

Implementation of algorithm

Stacked generalization implementation

Practical application – Sonar dataset (Mine and Rock prediction)

More information about the dataset

Summary

9. Stacked Generalization – Part 2

Feature selection

Why feature selection?

- Simplification of models

- Dataset information

- Predicted attribute

- Attribute information

- Shorter training time

- To avoid the curse of dimensionality

- Enhanced generalization by reducing overfitting

Feature selection for machine learning

- Univariate selection

- Recursive Feature Elimination

- Principle Component Analysis

- Choosing important features (feature importance)

Understanding the SVM

How does SVM work?

- Hyperplane – separation between the data points

Implementation of an SVM

- Objective function

- Function optimization

- Handling a nonlinear dataset

Stacking of nonlinear algorithms

Spam classification with stacking

Dataset information

Attribute information

How to choose classifiers?

Summary

10. Modern Day Machine Learning

Artificial Neural Networks (feed-forward)

How does ANN work?

Training of ANNs

Learning by backpropagation

ANN implementation using Keras and TensorFlow

TensorFlow for machine learning

Keras for machine learning

Digit classification using Keras and TensorFlow

Deep learning

Convolutional Neural Networks

Local receptive fields

Shared weights and biases

Pooling layers

Combining all the layers

Implementation of CNN in Python

Recurrent Neural Networks

How RNN works (unrolling RNN)

Unrolling the forward pass

Unrolling the backward pass

Backpropagation Through Time

Backpropagation training algorithm

Backpropagation Through Time

Long Short-Term Memory networks

The idea behind LSTMs

Step-by-step LSTM walkthrough

Text generation using LSTM

Problem description – project Gutenberg

LSTM model

Generating text with an LSTM Network

Summary

11. Troubleshooting

Full code of the implemented algorithm ID3

Code of the CART algorithm

Code for random forest

Code for KNN and subspace bagging

KNN subspace bagging code

Code of the AdaBoost classifier

Code of GBMs

Full code of implementation

Full code of LSTM implementation

Preface

Science has given us its biggest gift: *Computers*. This invention is as significant as Fire. It has changed the history of mankind. Tell me any field of work where computers are not being used; I bet you cannot. Computers are special kind of species that only eats electricity and one precious thing in which all of the world is interested, information a.k.a. *DATA*. Yes, without data, there is no use of a computer; it is just a television-like screen and nothing more. So the next question arises: What to do with this data? Believe me, every chapter of this book will give you a perspective to utilize your data and extract useful results from it.

What this book covers

[Chapter 1](#), *Introduction to Ensemble Learning*, is our introductory chapter to the world of ensembles. So we will see how ensembles can be useful for getting high accuracy from classifiers, and how to quantify the performance of a classifier by analyzing variance and bias errors. We will discuss three important aspects of ensemble algorithms: bagging, boosting, and stacking. We will see decision tree bagging in this chapter. We will also see how boosting works and how to use it. At the end, we will discuss what stacking is and how to implement stacked generalization.

[Chapter 2](#), *Decision Trees*, teaches us about the creation of decision trees for making predictions on our dataset and how to code it in Python and then use this algorithm to make

predictions on Car dataset.

[Chapter 3](#), *Random Forest*, shows how can we make decisions on real-world numerical data using a decision tree. We will learn how a simple binary tree can be converted into a decent classifier; then we will use multiple such decision trees to create a forest of trees, which is known as the random forest algorithm. We will code this from scratch to apply it and a real-world dataset.

[Chapter 4](#), *Random Subspace and KNN Bagging*, covers random subspaces and how they can improve the classification accuracy of a simple classifier. We will see how this method can improve the results when we use ensemble methods; we will also work with the KNN algorithm and its practical applications, and improve our classification accuracy using subspace bagging with k-NN for spam classification.

[Chapter 5](#), *AdaBoost Classifier*, is the starting point for the boosting algorithm. We will

learn boosting itself in detail; then we will learn to create a decision stump to make a simple boosting solution. We will see how to put many simple classifiers in a series to solve complex problem and we will write our own code for the AdaBoost algorithm. Later in the chapter, we will see how we can use AdaBoost in a face detection task.

[Chapter 6](#), *Gradient Boosting Machines*, covers the basics of gradient boosting. We'll start from the basic concepts of gradient boosting. We will also see what regression is and how a regression tree works. Then we will implement a working regression tree ourselves and use it to fit a sinusoidal function; afterwards, we will see how to use a regression tree as a classifier. Then we will implement the theoretical concept of the gradient boosting machine in practical code and see how it can reduce prediction.

[Chapter 7](#), *XGBoost – eXtreme Gradient Boosting*, talks about the extreme gradient boosting library XGBoost. We will start with

a general description of XGBoost and then discuss the advantages of this third-party library. We will discuss the various parameters of the library in detail; we can tune them to obtain a good prediction accuracy from the classifier. We will work with two practical applications, which can help us to apply the algorithm to complex datasets.

[Chapter 8](#), *Stacked Generalization*, discusses the stacking of different classifiers. We will start with a simple introduction to the stacking process and cover linear classifiers in combination. We will create a stack of three classifiers and, during the process, learn two very useful classification models: perceptrons and logistic regression. We will see how a gradient descent algorithm can help us train a single perceptron for prediction purposes and cover the core concepts of logistic regression. We will train our logistic regression model using the same gradient descent algorithm; finally, we will create a stack and apply it to a practical

dataset.

[Chapter 9](#), *Stacked Generalization – Part 2*, teaches the stacking of linear and nonlinear algorithms for prediction. We will start with feature selection methods, where we will discuss the importance of feature selection and its benefits by implementing various feature selection algorithms using the sklearn library. We will see how feature selection can reduce the curse of dimensionality and improve the performance of the classifier by reducing the variance-bias trade-off. We will discuss support vector machines in detail by implementing one from scratch and see how to optimize its loss function using gradient descent algorithm. We will also see how to use the kernel trick to address the problem of non-separable datasets. In the end, we will come back to stacking, this time with a bang! We will use six different classifiers to create a stack and use the bagging strategy to predict the output.

[Chapter 10](#), *Modern Day Machine Learning*,

this chapter will talk about machine learning algorithms being used in current trend. We will start with simple definitions of artificial neural networks and how to train them using stochastic gradient descent and the backpropagation algorithm. Then we will implement a network for digit recognition from the MNIST dataset, and see how convolutional neural networks work and why they have an advantage over normal neural network. Afterwards, we will implement a CNN for a digit recognition problem. Then we will turn our learning train towards networks used in the natural language processing domain. We will see the motivation behind RNNs, how they work, and a very popular and successful type of RNN: LSTM networks. These networks are widely used in text generation tasks. We will see a detailed description of how these networks work. After completing the theoretical part, we will implement a small RNN for a text generation task and see how LSTM can be used for such tasks.

[Appendix](#), *Troubleshooting*, in this chapter, the author has mentioned detailed code of a few chapters so that the readers don't struggle while implementing the code.

What you need for this book

This book is a practical walkthrough of the machine learning technologies that require implementation of algorithms by you to understand the concepts in a more concrete way. I have used Python as the language to implement the algorithms in the form of code. You need not be a Python expert to code these algorithms; a simple understanding of Python is enough to get started with the implementation.

The code included in this book can run on Python 2.7 and 3, but you will need the NumPy and scikit-learn packages to implement most of the code discussed in this book.

For the implementation of ANNs, I have used Keras and TensorFlow libraries; again, basic

a understanding of these libraries is enough
for the code implementation.

Who this book is for

This book is written for those readers who are interested in understanding machine learning concepts in a do-it-yourself style. As this book implements almost all machine learning algorithms it present from scratch, without using any machine learning libraries, it will give you a clear understanding of the working of each algorithm.

This book is suitable for beginner, intermediate, and skillful users. As I have started each topic with very basic concepts, beginners will not feel any discomfort in understanding the concepts. Intermediates will learn the code implementation of crucial algorithms, such as random forests and SVMs. Skillful readers will learn how simple algorithms such as AdaBoost and gradient boosting machine can become very powerful and solve real-world problems.

This book is written as a journey from basic to advanced concepts. Although you can pick any chapter to start with, the code blocks require reference to different chapters.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We will use the `iris` dataset for this implementation."

A block of code is set as follows:

```
# Import All the required packages from sklearn
import numpy as np
from sklearn import model_selection
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris

#Load data
iris = load_iris()
X = iris.data
```

```
| Y = iris.target
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
| #We will define a method to calculate accuracy  
| of predicted output with known labels  
| def CalculateAccuracy(y_test, pred_label):  
|     nnz = np.shape(y_test)[0] -  
|     np.count_nonzero(pred_label - y_test)  
|     acc = 100*nnz/float(np.shape(y_test)[0])  
|     return acc
```

New terms and **important words** are shown in bold.



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your email address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.

7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Ensemble-Machine-Learning>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required

information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Introduction to Ensemble Learning

Ensemble learning is the divide-and-conquer technique of the machine learning world. As the name ensemble or grouping itself suggests, it is an ensemble of multiple models. There are many cases where a single machine learning model lacks in performance, then the perfect solution would be using more than one model. This figure shows the basic architecture of an ensemble framework:

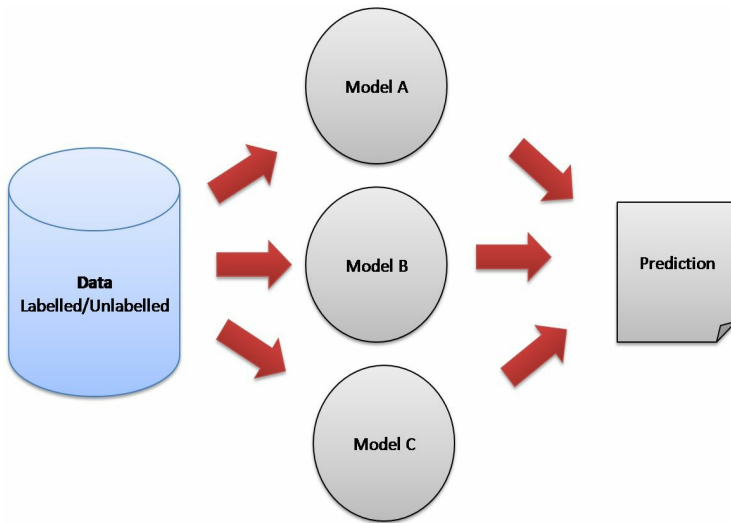


Figure 1.1: Ensemble learning architecture

In everyday life, we use ensemble learning for our daily decision making, let's see an example of this. Suppose you have to take admission in a university for an ensemble machine learning course. How will you decide whether it is the right choice or not? Here is the general approach:

- **Other students' reviews:** Students may provide information such as whether this course is useful for improving skill sets, information about the course curriculum,

the practical sessions, and so on. But as the students are not fully aware of the course's details (obviously, that's why they are taking that course) and also because they cannot suggest any other competitive course, you cannot rely on their suggestion solely. However, you know that their suggestions helped you in the past to choose previous courses. Let's say they are correct 60% of the time.

- **Study counselors:** You can get information regarding other competitive courses. They also know which universities have experts for the domain, so they can help you out to choose one course over other ML courses. Let's assume that these counselors are correct 60% of the time.
- **Career counselors:** Why do you want to take this course? Of course, for a better job, so a career counselor can tell you about the current requirements related to this skill set. They can tell you whether this course will help you enhance your

career. Keep in mind that career counselors are right most of the time, say 40%.

- **Social media:** Yes it helps! Here, you can join many discussion forums, find many suggestions, and pros and cons of the course. You can get suggestions from a large audience and they can perform a very critical role in your decision. This can show you in which region of the country there is more demand for a certain course, or where it is not considered an important skill. It may be correct, say, 40% of the time.
- **Placement officer:** A placement officer is a person who takes care of job placements; so they better know which specific companies need employees for this domain. Also they will give you assurance of getting a decent job after completion of this course. And trust me, 70% of the time you will go with their review, because they will help you in getting a job!

So have you noted down all the suggestions or reviews? I think yes. As you can see, no one is able to give you a clearly correct decision. Can we combine all the suggestions to get to the correct decision? Let's take a look at how you can make a decision on whether you should go with this course or not. Let's quickly analyze the scenario. As all the experts are from an independent system, we can get a very high accuracy rate as follows:

$$\begin{array}{|l} 1 - 60\% * 60\% * 40\% * 40\% * 70\% \\ 1 - 0.040 \\ 96\% \end{array}$$

Can you see what we have got by the combined decision? I think it is more than you think. Can we further improve it? Yes we can; for that, we have to take suggestions from more sources, such as course faculties, a company's workers, and so on.

The preceding example is based on an assumption that the suggestions from all the sources are independent. Well, in a practical

scenario, this is not possible. If we are talking about the same domain, more or less there will be a correlation between the suggestions. Suppose we choose six sources but all are students of that course. Then we cannot reach the correct decision with high confidence; this is where the power of ensembles comes into the picture, where you have multiple predictions and you combine all of them to get a high-confidence prediction. Let's enter the world of ensembles.

What is ensemble machine learning?

I think most of the part of this question is answered in the introduction itself. Ensemble learning is a very diversified field of machine learning where we combine multiple learners to increase the prediction power of our system. It's an art rather than a science. Combining the output of these learners is known as ensemble machine learning.

This chapter is mainly focused on introducing a few ensemble learning techniques (which we will elaborate on in later chapters). These techniques are being used widely in machine learning communities, so let's get into the details.

The purpose of ensemble machine learning

There are many reasons to go for ensembles, as each model of the group is usually based on algorithms, some of which are very simple and less computation intensive, but some may be quite complex and more computation intensive. For any production environment, accuracy and computation time are equally important. A system with higher accuracy but one that is unimplementable for real-time applications is of no use. However, a simple algorithm may lack in accuracy and may not fit onto the data properly; in those cases, we have to make a compromise between accuracy and computation time. This compromise can be minimized if we use many weak learners to get a combined

confidence index out of them, which may help us to implement such a system for real-time applications with very high accuracy.

These are the reasons to use ensembles:

- **The dataset is too large or too small:** When a dataset is too large and so it cannot be trained by a single model, we can create a small subset of data to train different models. At the end, we can choose the average of all as the final prediction. Similarly, when a dataset is too small to train a single model, we can use bootstrap methods to create random subsamples of data to train the models.
- **Complex (nonlinear) data:** Most of the time, a real-world dataset is a nonlinear dataset, where a single model cannot define the class boundary clearly. This is known as underfitting of the model. In such cases, we can use more than one model to train different subsets of the data and average out the result at the end to predict distinct boundaries.

- **High confidence:** When we train multiple classifiers on the training dataset and get mostly correlated output, it ensures a high prediction rate. Consider a case of classification where most of our classifiers predict the same class for an instance; in such cases, interprets ensemble system having high confidence on its decision.

How to create an ensemble system

We need to consider the following points to create an ensemble system:

- All models should have a difference of population. We should divide our dataset in such a way that subsets should have less correlation between each other. This will help create different classification models, which can give independent predictions.
- Models should have different hypotheses; that is, our expected outcomes should be different for each model. This will help us get a more generalized ensemble system.
- Each model should be dependent on different algorithms. It may be a combination of linear and nonlinear algorithms, or a combination of

unsupervised and supervised algorithms.
This will help us visualize data from
different perspectives.

Quantification of performance

The most important aspect of any statistical model is quantification of its performance. We can check the performance by calculating the difference between input and output, that is, error:

$$Err(x) = (E[\bar{f}(x)] - f(x))^2 + E[\bar{f}(x) - E[\bar{f}(x)]]^2 + \sigma_\epsilon^2$$

$$Err(x) = Bias^2 + Variance + IrreducibleError$$

Equation 1.1

The preceding equation for error shows that there are three main components: *Bias*, *Variance*, and *Irreducible Error*. We cannot do much about the last one, but by reducing the *Bias* and *Variance*, we can improve our classification results. How? Let's see.

Bias and variance errors

Bias error: As you can see in the preceding equation, bias is the average difference between predicted and actual values; if our system shows high bias error, that means we are getting a low-performing or under fitting model:

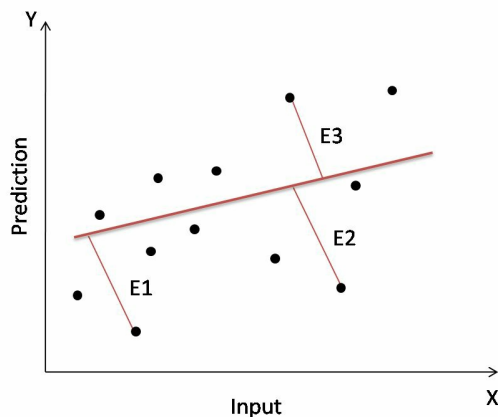


Figure 1. 2: Bias error

The preceding figure shows a representation of bias error. A linear model is trained over complex data and you can clearly see the error between the prediction (red line) and actual output (black dots). This shows that our model is not fitting properly on the dataset and thus underperforming. We can avoid such cases by using a complex (polynomial) model rather than a simple linear model.

Variance error: This quantifies the difference of the predicted value in the same observation. This error shows overfitting of our model. When we train a model that shows high variance, it gives near 100% accuracy of the training data; but when we present any test case beyond our training data, this model fails to predict the correct output. This condition can occur in two cases: either if we have less training data or if we train a complex model on a simple data:

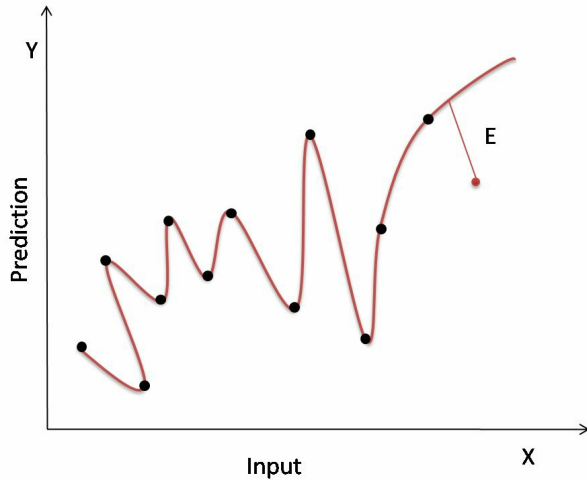


Figure 1.3: Variance error

As you can clearly see (*Figure 1.3*), our model is well trained (red curve) over the given training set but when we test an instance (red dot) outside the training data, we get a wrong prediction from the model. The problem of overfitting can be solved by increasing the number of training instances or by choosing the correct classifier for prediction.

We can summarize the preceding two errors as follows:

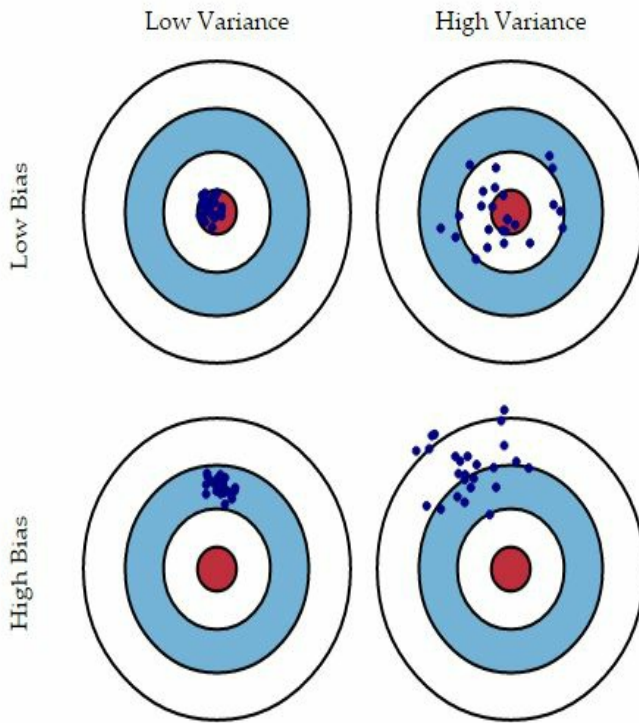


Figure 1.4: Bias and variance errors (Credits: Scott Fortmann)

In the preceding figure, consider the red dots as real values and blue dots as predicted outputs. This shows that when we observe a high bias in our model, we start increasing its complexity, which results in low bias. But if

we keep increasing the complexity (and our dataset is small), we can end up with an overfit model.

Management of the **Bias** and **Variance** error is one way to keep a balance between **Bias** and **Variance** errors, and ensemble methods come into the picture to do a trade-off analysis:

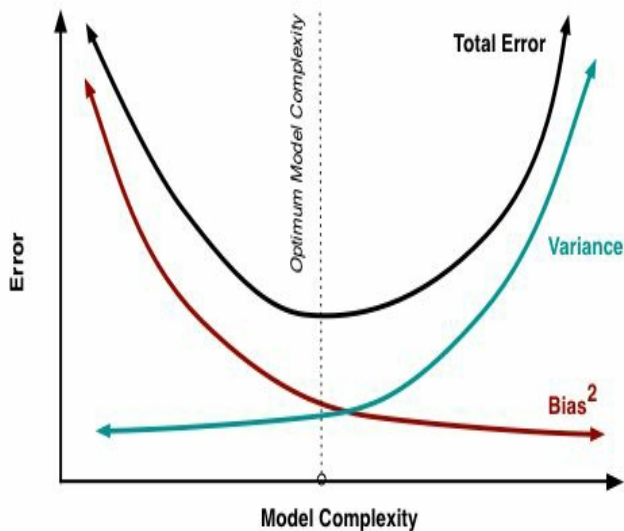


Figure 1.5: Trade-off analysis (Credit: Scott Fortmann)

Methods to create ensemble systems

There are three most common methods used to create ensemble systems and these methods have different optimized versions to solve different problems. They are as follows:

- Bagging
- Boosting
- Stacked generalization

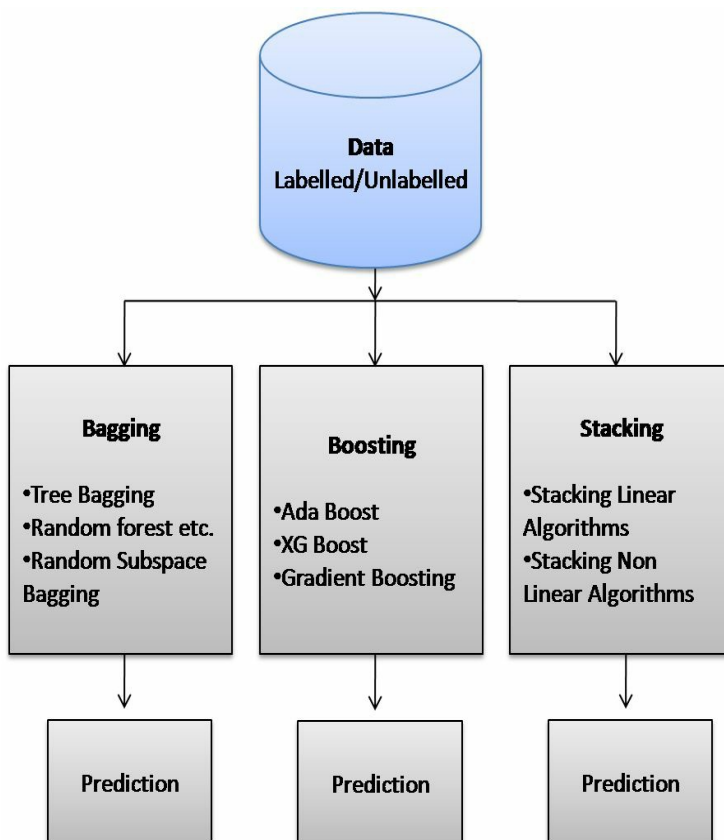


Figure 1.6: Ensemble algorithms

The preceding three methods are the core structure of this book. You will get an introduction to these techniques in this chapter. We will explain each of the algorithms later in different chapters.

Bagging

Bagging, an acronym of **bootstrap aggregation**, involves creating samples from the dataset with replacement; that is, any instance we have selected may repeat in the same sample many times. Apparently, we are increasing our training data by bootstrap, each created and then used to create a classifier model. The final prediction is the average of all of the prediction models.

The most popular bagging algorithm, frequently used by data scientists, is random forest, which is based on the decision tree algorithm. Another useful algorithm is **K-nearest neighbor (KNN)** subspace bagging, where the base learners are based on the k nearest neighbor algorithm. We will discuss these algorithms in detail in future.

We can understand bagging via the following example. Suppose we want to fit a complex

dataset as shown in this figure:

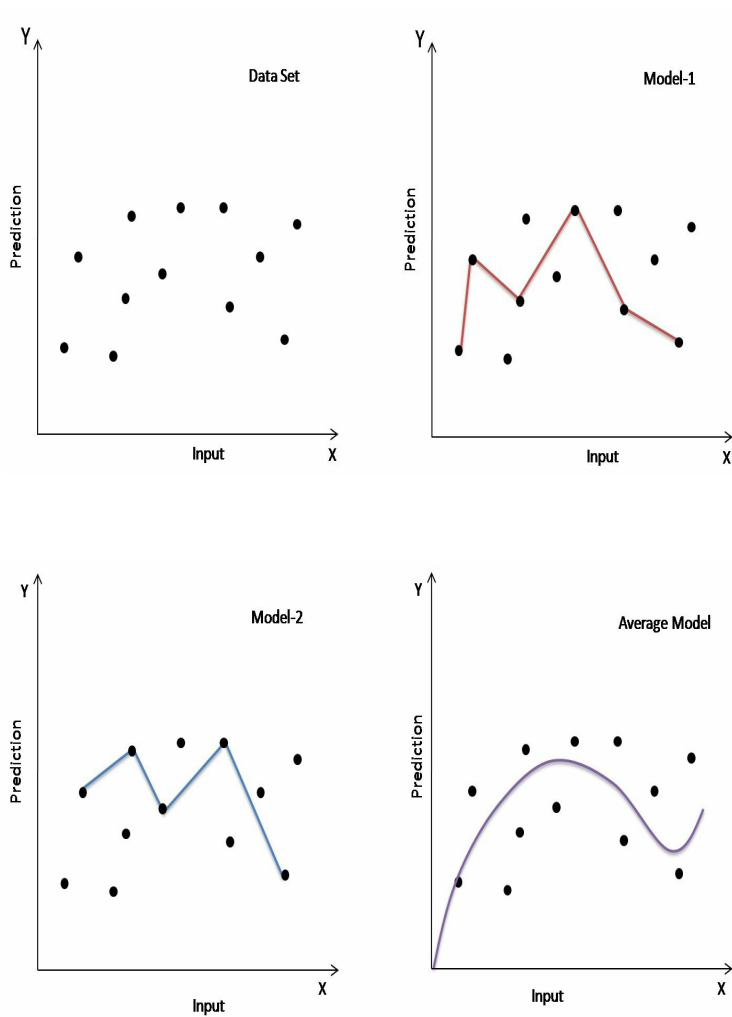


Figure 1.7: How bagging works

As you can see in *Figure 1.7*, there are four different subplots shown; subplot 1 (dataset) shows the distribution of our training data, where the x axis is for input and y axis is for the predicted output.

Suppose we want to fit a simple model (**Model-1**) over this dataset. You can clearly see that our model is underperforming or having a high bias error, but there are some data points that are well predicted by our model. Next, if we chose **Model-2** to fit our data, again our model faces underfitting. But now it is well fitted for those data instances that were not predicted correctly by **Model-1**. Now if we take an average prediction from both the models, we can compromise somewhere and get an optimal model with lower bias error than individual models.

The preceding example shows us the actual power of ensembles. We can combine results from several underperforming classifiers and then can create an ensemble model that can predict output with a very high confidence.

This figure shows such an ensemble system used for bagging:

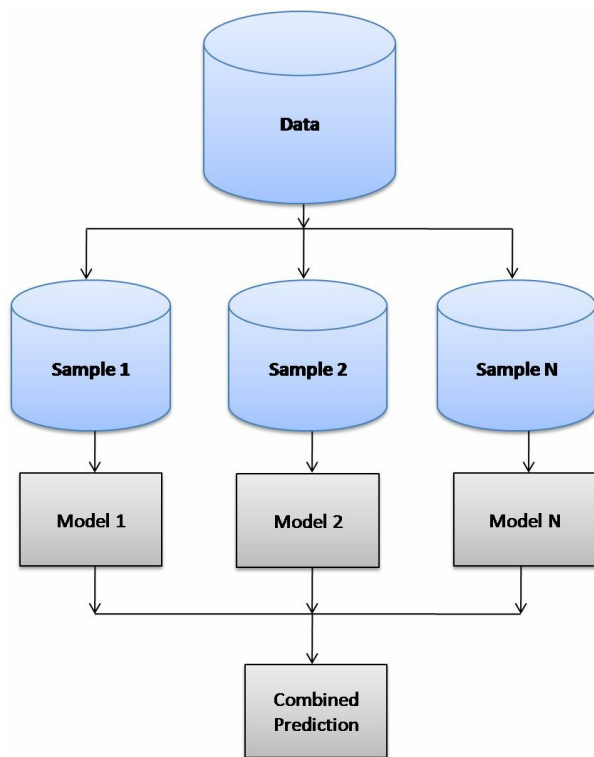


Figure 1.8: Bagging

As the preceding figure shows, bagging has three steps:

- Bootstrapping data

- Aggregation or model fit
- Combining the predictions from different models

Bootstrapping is a process in which we create multiple random samples out of our training dataset; those samples can be created by selecting random instances from data and grouping them together as shown in the following figure:

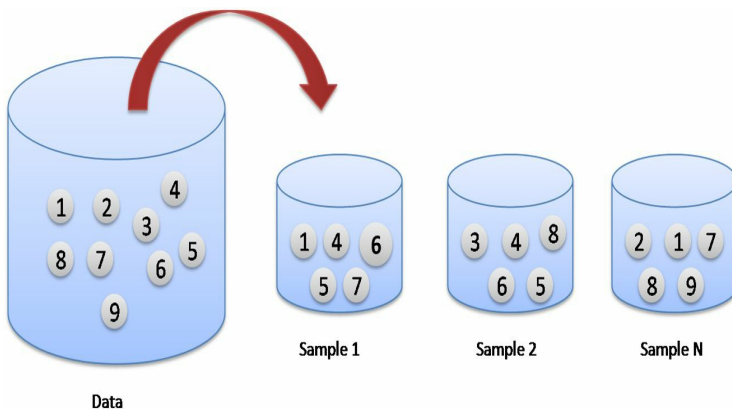


Figure 1.9 Bootstrapping samples

As you can see in the preceding figure, we have some data instances (numbered balls). We randomly choose some of the instances

(by replacing) and create a sample set. In **Sample 1**, you can see instance one is repeated two times. This means when we select any new instance, we are logically considering this as replacing the previous instance, but actually we are keeping both of them in our sub-sample. This is known as sampling by replacement.

Aggregation or model fit is explained in the following figure. Here, we select each bootstrap sample and try to train a classifier (such as **decision trees**) on each sample. We update the status of our classifier on the basis of the error between the actual value and predicted value:

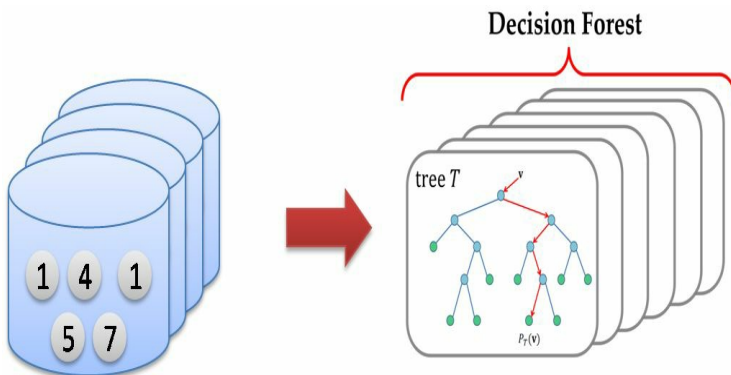


Figure 1.10 : Aggregation

The final step of bagging is combining the predictions from different classifiers. Averaging is the most common way to combine the predictions. If we use the decision tree as our base model, each model gives us the probability for each class. We choose the average probability across the classifiers as the predicted outcome.

Let's implement a Python code for the preceding steps. We will use `iris` dataset for this implementation. We're gonna use some Python libraries too, such as `sklearn` for machine learning algorithms and `numpy` for array-based operations. Let's jump to the code:

```
# Import All the required packages from sklearn
import numpy as np
from sklearn import model_selection
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris

#Load data
iris = load_iris()
X = iris.data
Y = iris.target
```



```

#Split data in training and testing set
X_fit, X_eval, y_fit, y_test= train_test_split(
X, Y, test_size=0.30, random_state=1 )

#Create random sub sample to train multiple models, This the step where number of Bootstrap samples will be define, we will create five samples.
seed = 7
kfold = model_selection.KFold(n_splits=5,
random_state=seed)

#Define a decision tree classifier: We will going to use tree based classification with 100 trees in each model.
cart = DecisionTreeClassifier()
num_trees = 100

#Create classification model for bagging: Here we will define that we want to create a bagging classifier in which decision tree will be trained.
model = BaggingClassifier(base_estimator=cart,
n_estimators=num_trees, random_state=seed)

#Train different models: Here we train our defined models with different samples, there will be 5 models gonna train for 5 samples.
results =
model_selection.cross_val_score(model, X_fit,
y_fit,cv=kfold)

#Print accuracy from all trained models.
for i in range(len(results)):
    print("Model: "+str(i)+" Accuracy is:
"+str(results[i]))

#Combine the result by averaging all the results.
print("Mean Accuracy is: "+str(results.mean()))

```

The execution result of the preceding code is:

```
Model: 0 Accuracy is: 1.0  
Model: 1 Accuracy is: 0.952380952381  
Model: 2 Accuracy is: 1.0  
Model: 3 Accuracy is: 0.904761904762  
Model: 4 Accuracy is: 0.857142857143  
Mean Accuracy is: 0.942857142857
```

As you can see, two out of five models are showing 100% accuracy. Models 4 and 5 are lacking in performance as they've got only 90.4% and 85.7 % accurate predictions. But when we average the results of all, we get a mean of 94.2% accurate predictions, which is quite decent.

Boosting

Unity is the power. Yes, the same concept can work in machine learning problems also. How? Oh yes it can, and the approach is known as **boosting**. It is a process in which we train multiple weak classifiers and combine their results to create a strong classifier. In theory, boosting algorithms are primarily used to prevent underfitting (high bias) and, of course, overfitting (high variance) of the classification model.

There are many boosting algorithms used by the data science community, and in future chapters, we will discuss some of them in detail. These algorithms are AdaBoost, XGBoost, gradient boosting machines, and so on.

So how does boosting work? Well, you can see in *Figure 1.4*. It starts with bootstrapping of data, which is the same as bagging. Then

we start training the different models here. These models are known as weak learners. Now the first question that comes to mind is: what is a weak learner?

Let's try to understand the concept of weak learners. As these classifiers are not fully responsible for the final prediction, they contribute their small part to the final decision. So, with respect to making the final decision, these classifiers are known as weak learners. For example, let's say we have to make a prediction model for Peter, our friend. He is studying in fifth standard. So we want to know how he will get an A+ in his term exams. For the answers, we ask the same question to his friends and they give the following answers:

- **Friend 1:** Peter will get A+ if he reads for two hours
- **Friend 2:** Peter will get an A+ if he watches less TV
- **Friend 3:** He can get A+ if he take classes regularly

- **Friend 4:** He can only succeed if he gets good tuition

What do you think? I think all of the preceding answers will be required for good term results, but none of his friends have a complete answer. Why? The answer is quite simple. They are not experts; they are not mature enough. Each child has his/her own opinion, but if you combine their answers, you will get a complete answer. Give it a try.

So here, Peter's friends are our weak learners. Each one can provide us partial information, and then we can combine all of them and get the conclusion. Weak learners work in the same fashion; they use simple rules (in the preceding case, the if statement) to predict classes. Finally, we combine all of their results to get our prediction.

Boosting is a supervised learning algorithm, mainly a framework that consists of many weak learners in cascade. It trains iteratively on the training data and updates their weights

according to the difference between predicted and actual values. The weight update depends on the algorithm we used for the update. In case the of boost by a majority algorithm, the misclassified learner gets a weight gain and a learner with true classification loses weight. So the classifier gives less attention to the true classification, which helps in faster convergence of the classifier.

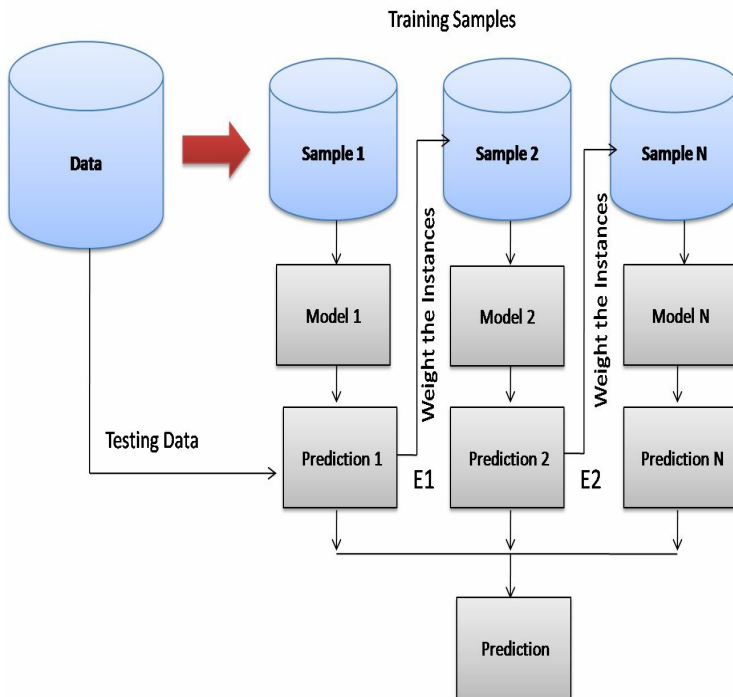


Figure 1.11: Boosting algorithm (cascade classifiers)

We can understand the boosting process via the following steps:

1. First, create random samples from your training data.
2. Now, train a classifier (**Model 1**) for this sample and test the whole training data (yes whole, not the sample).
3. Calculate the error for each instance prediction. If the instance is classified wrongly, increase the weight for that instance and create another sample (by replacement).
4. Repeat this procedure until you get high accuracy from the system.

Let's write some Python code for a well-known boosting algorithm, AdaBoost:

```
# Import All the required packages from sklearn
from sklearn import model_selection
from sklearn.datasets import load_iris
from sklearn.ensemble import AdaBoostClassifier
# Boosting Algorithm
from sklearn.tree import DecisionTreeClassifier
import numpy as np
```

```

#Load IRIS data
iris = load_iris()
X = iris.data
Y = iris.target

#Split data in training and testing set (80% training data & 20 % testing)
X_fit, X_eval, y_fit, y_test=
model_selection.train_test_split( X, Y,
test_size=0.20, random_state=1 )

#Define a decision tree classifier as WEAK learner
cart = DecisionTreeClassifier()
num_trees = 25

#Create classification model for boosting
model = AdaBoostClassifier(base_estimator=cart,
n_estimators=num_trees, learning_rate = 0.1)

#Train Classification model
model.fit(X_fit, y_fit)

#Test trained model over test set
pred_label = model.predict(X_eval)
nnz = np.float(np.shape(y_test)[0] -
np.count_nonzero(pred_label - y_test))
acc = 100*nnz/np.shape(y_test)[0]

#Print accuracy of the model
print('accuracy is: '+str(acc))

```

Output of the preceding code snippet:

```
| accuracy is: 96.6666666667
```

After training the preceding model, we get an accuracy of 96.6% on our test data, which

shows that our classifier is well trained over the training samples.

Stacking

In the previous two methods, we got an idea of how multiple models of the same kind can help us improve the accuracy of our classification. What if we combine models of two different kinds? Can it help us? The answer is yes! In some cases, it is very helpful to use different kinds of prediction models to get higher prediction accuracy. Well, the next question is: how?

As we have seen, a single decision tree in a weak learner (boosting) or in bagging can help us only to make partial predictions. To reduce the bias error from our model, we need to increase the number of classifiers in our ensemble framework. In the same way, for very complex datasets, a single solution might not give us a higher prediction rate. For such situations we need to combine different kinds of classifiers, where the output of one classifier can be the input of another. The

interesting part in the stacking process is that the new model is trained to combine the predictions from previously trained models on the same dataset.

Combining these classifiers' results may be done by simply averaging their predictions, or we can use the weighted sum also. This weighted sum can be calculated by using linear or logistic regression algorithms. An important thing to note down: we train the initial level of classifiers separately and then combine their results by training another classifier. So the combining classifier does not see the actual training set.

One more important thing is that the submodels (initial classifier) should produce uncorrelated results; that is, their results should not match. So a prediction from submodels will be like uncorrelated features for the final classifier:

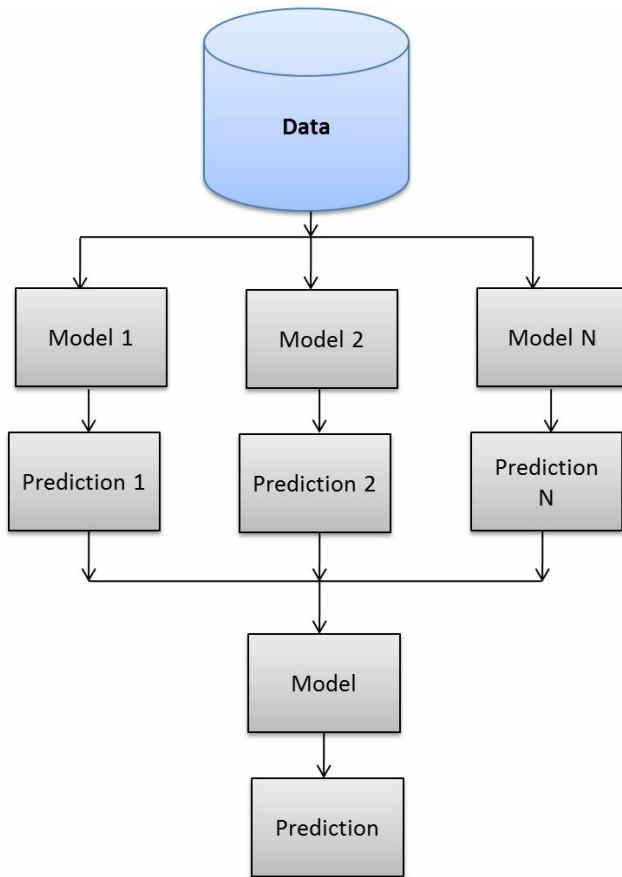


Figure 1.12: stacked generalization

As you can see in the preceding figure, we are not creating different samples out of training data to train classifiers. Instead, we are training each classifier with the whole

training data; that is, each classifier is independent of the other, which allows us to use classifiers with different hypotheses as well as algorithms. For example, we can use a linear regression classifier and a random forest for training and then we can combine their predictions using a support vector machine.

Let us understand stacking with the help of the following example with Python implementation.

We will choose the fisher iris data for this example again. We will train a KNN classifier, random forest, and naive Bayes classifier for the first level of classification. Then we will use the predictions of these classifiers as feature instances for a linear regression classifier and try to see whether it helps us to increase our prediction accuracy or not.

The following is the code listing for our problem:

```

#Import IRIS dataset from sklearn
from sklearn import datasets

#Import Random forest Logistic regression,
naive bayes and knn classifier classes for
creating stacking.
from sklearn.ensemble import
RandomForestClassifier
from sklearn.linear_model import
LogisticRegression
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import
KNeighborsClassifier

#Import numpy for array based operations
import numpy as np

#Load the dataset
iris = datasets.load_iris()

#Extract data and target out of dataset
X, y = iris.data[:, 1:3], iris.target

#We will define a method to calculate accuracy
of predicted output with known labels
def CalculateAccuracy(y_test, pred_label):
    nnz = np.shape(y_test)[0] -
np.count_nonzero(pred_label - y_test)
    acc = 100*nnz/float(np.shape(y_test)[0])
    return acc

#Create a KNN classifier with 2 nearest
neighbors
clf1 = KNeighborsClassifier(n_neighbors=2)

#We will create a random forest classifier with
2 decision trees
clf2 = RandomForestClassifier(n_estimators =
2, random_state=1)

#Create a Naive bayes classifier
clf3 = GaussianNB()

```

```

#Finally create a logistic regression classifier to combine prediction from above classifiers.
lr = LogisticRegression()

#Now we will Train all first level classifiers
clf1.fit(X, y)
clf2.fit(X, y)
clf3.fit(X, y)

#Predict the labels for input data by all the classifier; print their accuracy and store the prediction into an array (f1,f2,f3)
f1 = clf1.predict(X)
acc1 = CalculateAccuracy(y, f1)
print("accuracy from KNN: "+str(acc1) )

f2 = clf2.predict(X)
acc2 = CalculateAccuracy(y, f2)
print("accuracy from Random Forest: "+str(acc2) )

f3 = clf3.predict(X)
acc3 = CalculateAccuracy(y, f3)
print("accuracy from Naive Bayes: "+str(acc3) )

#Combine the predictions into a single array and transpose the array to match input shape of or classifier.
f = [f1,f2,f3]
f = np.transpose(f)

#Now train the classifier
lr.fit(f, y)
final = lr.predict(f)

#Calculate and print the accuracy of final classifier
acc4 = CalculateAccuracy(y, final)
print("accuracy from Stacking: "+str(acc4) )

```

Output of the preceding code snippet:

```
accuracy from KNN: 96.66666666666667  
accuracy from Random Forest: 94.66666666666667  
accuracy from Naive Bayes: 92.0  
accuracy from Stacking: 97.33333333333333
```

We can see an improvement in the prediction accuracy when we stack all the classifiers. This example shows how we can use predictions from other classifiers to train a new classifier over them to get a high-performance prediction framework. In the following chapters, we will have detailed discussions on all of the techniques we have discussed here.

Summary

As this was our introductory chapter to the world of ensembles, we saw how an ensemble can be useful for getting high accuracy from classifiers. We saw how to quantify the performance of a classifier by analyzing variance and bias errors. We discussed the three important aspects of ensemble algorithms, that is, bagging, boosting, and stacking. We learned how to do bagging using the same kind of classifiers, in this case, decision trees. We also saw how boosting works and how to use it. At the end, we discussed what stacking is and how to implement stacked generalization.

We learned to implement the preceding classifier algorithms in Python for practical analysis. You can choose different datasets to try these codes. In future chapters, we will see detailed discussions on all the classifiers and algorithms we have used here,

of course with their Python implementation.

Decision Trees

We have seen a small application of decision trees in the previous chapter, where we created a bagging classifier with a decision tree as the base learner. In this chapter, we will discuss decision trees in depth and see how to create decision trees, how they work, and where we can use them to solve ML problems.

Let's start with a simple example. Suppose your fund advisory company wants to open its new branch in a small city. You want to select people who are willing to invest through your company. How can you choose them? One way to do this is to go *brute force*; pick up your phone and dial some numbers. Ask them whether they are willing to invest money in the market. Some of them may not be able to earn enough to invest, some may not want to, and, more dramatically, some may get irritated by your phone calls and may

file a complaint against your firm. So what would you do?

Your problem can be solved by analyzing the people's mindset in that city, and for that, you have to reach out them and create a dataset by choosing a sample of them; it will represent the whole population. Suppose you have gone through this step and now you have a dataset with you that looks like this:

SN	Name	Sex	Age	Employment
1	Person 1	M	20	Unemployed
2	Person 2	M	25	Employed
3	Person 3	M	30	Employed
4	Person 4	F	30	Employed

5	Person 5	M	25	Employed
---	----------	---	----	----------

Table 2.1: Sample dataset

This table is just a representation of our big dataset. Suppose that after analyzing this table, we get some basic statistics regarding their investment. The summary of the dataset is:

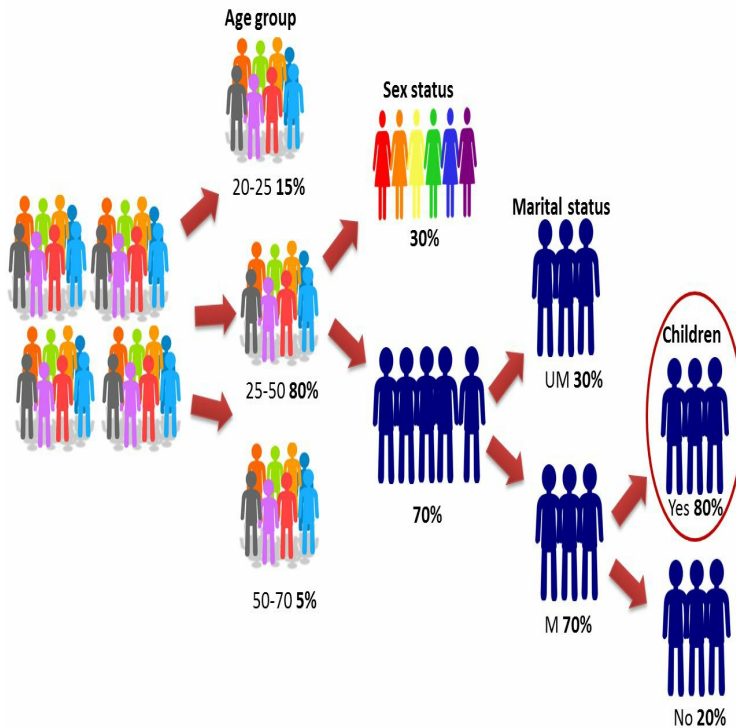


Figure 2.1: decision tree;

This figure shows us a in-depth analysis of our dataset; we can analyze it as follows:

- From our total sample population, there are three age groups. You can see that only **15%** of people from the **Age group** of **20-25** years are interested. Only **5%**

are from the **Age group 50-70** years, and **80%** of the **25-50** years are persons who invest in the market. So, for the time being, we will concentrate on this **80%** of the population only.

- Only **30%** of females from this population invest whereas the male investor population is **70%**. So we will move ahead with the male population.
- As you can see, males who are unmarried are less interested in investing (**30%**); on the other hand **70%** married males are interested. So our criteria further narrows down to them.
- Next, we see that married persons who have children are most likely to invest (**80%**) and only **20%** of the people without children are investing.

What we have achieved from this analysis?
Yes, folks, we got our target people (married young males with children) who are most likely to invest through our firm. But before getting excited, wait a second and think: what we have done? Congratulations! We have just

created a decision tree from our dataset. Now, whenever you enter a person's details into your tree, it can tell you whether he/she may be your investor or not. But how? We will try to find the answer to this question in the next section.

How do decision trees work?

The previous solution left us with questions about how exactly decision trees work, how we have selected the root node feature, and how we can prove mathematically that the root we are choosing will converge for a solution. In this section, we will try to understand how we can write an algorithm to create decision trees from scratch.

First, we will try to understand the basic terms used frequently in tree-based algorithms:

- **Root** is the first node of the tree. Every tree must have a root node. It is actually the origin of the tree; in any kind of tree, there will be only one root node.
- **Edge** is the connection between two nodes. A tree with N number of nodes

will have a maximum of $N-1$ edges.

- **Parent** is a node with branches toward other nodes.
- **Child** is a descendant of any node. In a tree, any parent node can have any number of child nodes. All nodes except the root node are known as child nodes.
- **Siblings** are nodes that are children of the same nodes; in other words siblings are children of the same parent node.
- **Leaf** is a node that does not have any child node; in our case, it will be the node where we find predictions on input data.
- **Degree** of a node is the total number of children of that node.
- **Level** of a node is each step from root (top) to leaf (bottom); this count starts from the root node, which is always at level 0.
- **Height** is the longest path from a leaf to the root node; that is, the height of the leaf node is zero and height of the root node is the height of the tree.
- **Depth** of a tree is the number of edges

from root to leaf on the longest path.

- **Sub tree** can be assumed like this: any parent in the tree is a root of a sub tree. Each child from a node forms a sub tree recursively.

We will understand and implement one of the most used algorithms to create a decision tree, that is, the **Iterative Dichotomiser 3 (ID3)** algorithm. In future chapters, we will see **Classification And Regression Trees (CART)**. ID3 is the classical decision tree implementation proposed by Ross Quinlan, and CART is proposed by Leo Breiman et. al. Both the algorithms are widely used in the data science community.

ID3 algorithm for decision tree building

The ID3 algorithm is used for classification of categorical datasets, that is, datasets that have non-numerical values, such as the sample dataset we discussed in the example in the introduction. We will create a decision tree on the same data with some modifications to understand that it's working.

SN	Name	Salary	Gender
1	Person 1	Low	Male
2	Person 2	Med	Female

3	Person 3	Med	
4	Person 4	Med	
5	Person 5	Med	
6	Person 6	High	
7	Person 7	Low	
8	Person 8	High	
9	Person 9	Med	
10	Person 10	Low	

Table 2.2: Training dataset

We have selected some attributes from our investor dataset and converted all numerical data fields into categorical data, such as salary divided into three category: `Low`, `Med`, and `High`. Now we will implement the ID3 algorithm step wise. And, as you know, you can learn an algorithm much faster if you write a code for it, so let's also implement a Python code for our decision tree.

The first step is to feed data into a suitable data structure from where we can easily perform the required mathematical calculations for storing our data in a tabular form. We are going to use the `pandas` library, which has a specific a data structure known as `DataFrame`; let's write it out:

```
#Lets create a python dictionary to store our
attributes and their respected values as
follows
dataset = {'Name':['Person 1', 'Person
2', 'Person 3', 'Person 4', 'Person 5', 'Person
6', 'Person 7', 'Person 8', 'Person 9', 'Person
10'],
'Salary':
['Low', 'Med', 'Med', 'Med', 'Med', 'High', 'Low', 'Hig
'Sex':
```

```

['Male', 'Male', 'Male', 'Female', 'Male', 'Female', '
'Marital':
['Unmarried', 'Unmarried', 'Married', 'Married', 'Ma
'Class':
['No', 'No', 'Yes', 'No', 'Yes', 'Yes', 'No', 'Yes', 'Ye
#Now we will create a Data frame out of the
preceding dataset(dictionary)
import pandas as pd
df = pd.DataFrame(dataset)
print(df)

```

When we execute the preceding code block, the `DataFrame` creates a tabular representation of our dictionary dataset, where we can visualize it in much better way, as follows:

	Salary	Sex	Marital	Class
0	Low	Male	Unmarried	No
1	Med	Male	Unmarried	No
2	Med	Male	Married	Yes
3	Med	Female	Married	No
4	Med	Male	Married	Yes
5	High	Female	Unmarried	Yes
6	Low	Female	Unmarried	No
7	High	Male	Unmarried	Yes
8	Med	Female	Unmarried	Yes
9	Low	Male	Married	Yes

Root node

First of all, to create a tree, we need to define its root node. For that, we have to select an attribute (feature) that has the most informative data. This can lead us to a true classification of the instance. For this, we will require three basic calculations, that is, entropy of classes, information gain, and entropy of attributes.

Remember the definition of entropy from your high-school chemistry? It is the degree of randomness or uncertainty, in our case the degree of variance, or in simple terms the class variance. So our main target is to select such nodes by which we can reduce the entropy or attribute variance. In other words, entropy is also a measure of impurity; a dataset is called a pure dataset if all its instances have the same class attributes (in our case, if all persons are investors, our dataset will be a pure dataset). For a pure

dataset, entropy will be zero or near zero. Information gain is also known as **mutual information**; it is nothing but a measure to select the most informative attribute that can help us to reduce entropy. We can calculate all three of them for our problem as follows:

Entropy:

$$E = -F_{c1} \log_2(F_{c1}) - F_{c2} \log_2(F_{c2}) \quad (2.1)$$

where F_{c1} and F_{c2} are the fractions of different classes (for example, _{Yes} and _{No}) in class attributes.

The entropy of a dataset for different attribute values (for example, for _{Low}, _{Med}, and _{High}):

$$Ev = -F_{c1i} \log_2(F_{c1i}) - F_{c2i} \log_2(F_{c2i}) \quad (2.2)$$

where F_{c1i} and F_{c2i} are the fractions of _{Yes} and _{No} for the selected attribute.

Total entropy of the attribute:

$$E_a = \text{Sum}[(Y_i + N_i) * E_v]$$

Information gain for the attribute:

$$IG = E - E_a \quad (2.4)$$

where E is the entropy of the class and E_a is the entropy of the attribute.

Let's write some Python definitions to calculate the class entropy using (2.1):

```
import numpy as np
def getClassEntropy(classAttributes):
    #Get distinct classes and how many time
    they occure
    _, counts =
np.unique(classAttributes, return_counts=True)
    denom = len(classAttributes)
    entropy = 0 #Initialize entropy variable
    #Run a loop to calculate entropy of dataset
    for count in counts:
        fraction = count/denom
        entropy+= -fraction*np.log2(fraction)
    #Equation 2.1
    return entropy
```

The attribute with maximum information gain as per (2.3) will be our root node. Now we will calculate the root node for our tree in the following way:

First, we have to calculate the entropy of our classes using (2.1), so the parameters we need are:

- Number of _{yes} in the dataset: 6
- Number of _{no} in the dataset: 4
- Total number of instances: 10

So, by (2.1), the entropy of our class attribute is:

$$\begin{aligned} E &= -0.6 * \log_2(0.6) - \\ &\quad 0.4 * \log_2(0.4) \\ E &= 0.9709 \end{aligned}$$

Next we will have to calculate the entropy of individual attributes; our first attribute is the salary of participants. Let's prepare a small tabular representation for the attribute.

Salary

The following table is a subtable extracted out of *Table 2.2* for analyzing the salary attribute, as salary has three types of values: *Low*, *Med*, and *High*. You can see in *Table 2.2* that we've got three persons with a low-salary attribute. One of them is an investor and the other two are not investing. Similarly 3 out 5 in the medium-salary group are investors and 2 are not. Whereas for the high-salary group, 2 out of 2 are investors:

Values	Y	N	
<i>Low</i>	1	2	0.92
<i>Med</i>	3	2	0.97

<i>High</i>	2	0	0
$E_{salary} = 0.7624$			
$I_{Gsalary} = 0.2085$			

Table 2.3: Summary of the Salary attribute

Now let's put in the code for getting a summary table as before. We will call it a histogram table; it shows us the class distribution. The Python code for it will be:

```
def getHistTable(df,attribute):
    #This function create a subtable for the
    given attribute
    #Get values for the attribute
    value = df[attribute]

    #Extract class
    classes = df['Class']

    #Get distinct classes
    classunique = df['Class'].unique()

    #Get distinct values from attribute for
    example, Low, High and Med for Salary
    valunique = df[attribute].unique()

    #Create an empty table to store attribute
```

```

value and their respective class occurrence
    temp =
np.zeros((len(classunique),len(valunique)),dtype

    histTable =
pd.DataFrame(temp,index=classunique,columns=valu

    #Calculate class occurrence for each value
for Med salary how many time class attribute is
Yes
    for i in range(len(classes)):
        histTable[value[i]][classes[i]]+= 1

    return histTable

```

When we call the preceding function, the input argument will be the data frame and attribute value; the calling will look like:

```

| histTable = getHistTable(df,"Salary")
| print(histTable)

```

Following are the results of the execution of the previous lines of code:

	Low	Med	High	
No	2	2	0	
Yes	1	3	2	

Now we will calculate the entropy of individual attributes with the use of (2.2); for the low-salary group, the equation will look like this:

$$E_{low} = -0.33 * \log_2(0.33) - 0.66 * \log_2(0.66)$$

$$E_{low} = 0.9234$$

For the medium-salary group:

$$E_{med} = -0.6 * \log_2(0.6) - 0.4 * \log_2(0.4)$$

$$E_{med} = 0.9709$$

Similarly, we will find the information gain for the high-salary group:

$$E_{high} = 0$$

Once we have got the individual information gain for *low*, *med*, and *high* salary, we can use it to calculate the entropy of the attribute using (2.3):

$$E_{low} = E_{low} * (F_{c_{1low}} + F_{c_{2low}}) / (\text{no. of instances})$$

$$E_{med} = E_{med} * (F_{c_{1med}} + F_{c_{2med}}) / (\text{no. of instances})$$

$$E_{high} = E_{high} * (F_{c_{1high}} + F_{c_{2high}}) / (\text{no. of instances})$$

$$E_{salary} = E_{low} + E_{med} + E_{high}$$

$$E_{salary} = 0.7624$$

Once we get the entropy of the *salary* attribute, we use (2.3) to calculate its information gain, which will be:

$$IG_{salary} = E - E_{salary}$$

$$IG_{salary} = 0.2085$$

Similarly we can calculate the information gains for `sex` and `Marital` status.

Now let's add a `getInformationGain` function to our code to get the information gain of these attributes:

```
def getInformationGain(histTable, classEntropy):

    #Initialize a variable for storing
    probability of Classes
    fraction = 0
    #Calculate total number of instances
    denom = np.sum(np.sum(histTable))

    #Initialize variable for storing total
    entropies of attribute values
    EntropyAtt = 0

    #Now we will run a loop to access each
    attribute and its information gain
    for key in histTable.keys():
```



```

        #Extract Attribute
        attribute = histTable[key]
        entropy = 0
    #Initialize variable for entropy
    calculation
        coeff = 0
    #Initialize variable to store
    coefficient

        #Find out sum of class attributes(in our
    case Yes and No)
        denom2 = np.sum(attribute)

        #Run a loop to get entropy of distinct
    values of attribute
        for value in attribute:
            #Calculate coeff
            coeff+= value/denom

            #Calculate probability of the
    attribute value
            fraction = value/denom2

            #Calculate Entropy
            eps = np.finfo(float).eps
            entropy+= -
    fraction*np.log2(fraction+eps)
            EntropyAtt+= coeff*entropy

        #Calculate Information Gain using class
    entropy
        InfGain = classEntropy - EntropyAtt
    return InfGain,EntropyAtt

```

The preceding function will take `histTable` as an input argument along with the `classEntropy` of the dataset to calculate the information gain. Let's run the preceding script for the `hist` table of the `salary` attribute and see what we

get:

```
| Ec = getClassEntropy(df["Class"])  
| histTable = getHistTable(df, "Salary")  
| Ig, Ea = getInformationGain(histTable, Ec)  
  
| print("Information Gain for %s: %.2f and  
| Entropy: %.2f"%("Salary", Ig, Ea))
```

After execution we will get:

```
| Information Gain for Salary: 0.20 and Entropy:  
| 0.76
```

The Sex attribute

The following table shows a summary of the calculations. In this case, you can see that 4 out of 6 males are investors while 2 are not investing anywhere. There are 50% female investors:

Values	Y	N	At
M	4	2	0.5
F	2	2	1
$E_{sex} = 0.9540$			
$IG_{sex} = 0.0169$			

Table 2.4: Summary of the Sex attribute

Let's see the information gain and entropy for the `sex` attribute:

```
Ec = getClassEntropy(df["Class"])
histTable = getHistTable(df, "Sex")
Ig, Ea = getInformationGain(histTable, Ec)

print("Information Gain for %s: %.2f and
Entropy: %.2f"%("Salary", Ig, Ea))
```

After execution we will get:

```
Information Gain for Sex: 0.02 and Entropy:
0.95
```

Marital status

The following table shows the summary of the calculations; here, 3 out of 4 married persons are investing somewhere and 50% of unmarried people are investing:

Values	Y	N
M	3	1
UM	3	3
$E_{Marital} = 0.9244$		
$IG_{Marital} = 0.0465$		

Table 2.5: Summary of the Marital status attribute

Let's see the information gain and entropy calculated by our implemented function;

```
Ec = getClassEntropy(df["Class"])
histTable = getHistTable(df, "Marital")
Ig, Ea = getInformationGain(histTable, Ec)

print("Information Gain for %s: %.2f and
Entropy: %.2f" % ("Salary", Ig, Ea))
```

Information gain and Entropy for Marital attribute will be:

```
Information Gain for Marital: 0.05 and
Entropy: 0.92
```

After performing the preceding calculations, we can summarize the information gain for all the attributes in this table:

Attribute	Information Gain
Salary	0.2085
Sex	0.0169
Marital	0.0465

Table 2.6: A summary of the information gain calculated

Now, as we have all the building blocks needed to get `winnerNode` using `InformationGain`, we can write a method that directly gives us the `winnerNode` whenever we pass a data frame to it. Let's write it and try to find our winner attribute:

```
def getNode(df):  
    #This function is written for getting  
    winner attribute to assign node  
  
    #Get Classes  
    classAttributes = df['Class']  
  
    #Create empty list to store Information  
    gain for respected attributes  
    InformationGain = []  
    AttributeName = []  
  
    #Extract each attribute  
    for attribute in df.keys():  
        if attribute is not 'Class':  
            #Get class occurrence for each  
            attribute value  
            subtable = getHistTable(df,attribute)  
  
            #Get class entropy of the data  
            Ec = getClassEntropy(classAttributes)  
  
            #Calculate Information Gain for each  
            attribute  
            InfoGain,EntropyAtt =  
            getInformationGain(subtable, Ec)  
  
            #Append the value into the list
```

```

        InformationGain.append(InfoGain)
        AttributeName.append(attribute)
        print("Information Gain for %s: %.2f
and Entropy: %.2f"%
(attribute, InfoGain, EntropyAtt))

    #Find out attribute with maximum
information gain
    indx = np.argmax(InformationGain)
    winnerNode = AttributeName[indx]
    print("\nWinner attribute is: %s"%
(winnerNode))

    return winnerNode

```

Let's call the preceding function for our data frame and try to find out the root node for our tree:

```
| node = getNode(df)
```

Following is the output for the winner attribute;

```

| Information Gain for Salary: 0.21 and Entropy:
| 0.76
| Information Gain for Sex: 0.02 and Entropy:
| 0.95
| Information Gain for Marital: 0.05 and Entropy:
| 0.92
| Winner attribute is: Salary

```

As you can see, the preceding entropy of the salary attribute is much lower than that of sex

and `Marital` status, and its information gain is quite higher than that of the other two, which clearly indicates our winner attribute.

Branches: It is clear from *Table 2.6* that the `Salary` attribute has the highest information gain and will be the root node for our tree. Once we have got our winner as the root node, the next thing is to decide its edges or branches. So the branches of a decision tree are nothing but attribute values of the parent node (in the current case, root). As the **Salary** attribute has become the root node here, its attribute values of **Low**, **Med**, and **High** will be the branches of this node. So our tree will look like this:

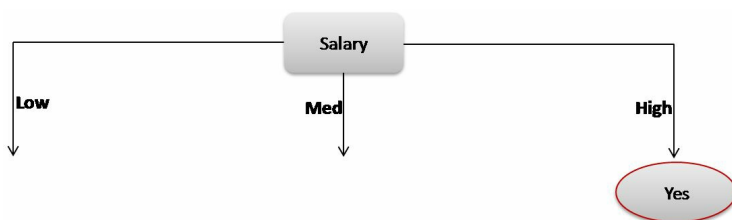


Figure 2.2: Root node and its branches

You can see an interesting thing in *Figure*

2.2. That **High** branch has already got its decision as **Yes**. How? Well, if you see the data table, you will understand that the person with the high salary is a investor no matter what his/her marital status or sex is. It shows that we are heading towards the right status:

SN	Name	Salary	St
6	Person 6	High	F
8	Person 8	High	M

Table 2.7: A condition of a pure subset

The preceding table shows a case of a pure subset. Wait! What? Pure subset? Yes, a subset of our dataset that has all class values of the same class is known as a pure subset. Well, our whole task here is to find nodes that create pure subsets, as you saw before.

Let's create `subtable` in our code too and let's try to find whether it is a pure subset (with zero entropy) or not:

```
def getSubtable(df,node,atValues):  
    #This function is written to get subtable  
    for given attribute values(such as table for  
    those persons  
    #whose salary is Medium)  
    subtable = []  
  
    #run a loop through the dataset and create  
    subtable  
    for i in range(len(df[node])):  
        if df[node][i]==atValues:  
            row = df.loc[i,df.keys()]  
            subtable.append(row)  
  
    for c in range(len(df.keys())):  
        if df.keys()[c]==node:  
            break;  
    #Create a new dataframe  
    subtable =  
    pd.DataFrame(subtable,index=range(len(subtable)))  
  
    return subtable
```

Let's execute the preceding code block for the High attribute value for salary:

```
| subtable = getSubtable(df,"Salary","High")  
| print(subtable)
```

After executing previous lines we will get the following information:

	Salary	Sex	Marital	Class
0	High	Female	Unmarried	Yes
1	High	Male	Unmarried	Yes

So what's next from here? Now we want to know which node should come under the Low and Med salary groups in the same way as we have done for the root node.

Parent node

For the calculation of the next node under the `Low` salary branch, we will create a subset out of our main data, which will contain values regarding the `Low` salary attribute only:

SN	Name	Salary	Sex
1	Person 1	Low	M
2	Person 7	Low	F
3	Person 10	Low	M

Table 2.8: A subdataset for Low salary values

For the preceding data, we need to calculate

the information gain for `Sex` and `Marital` status, and their winner will become the parent node under this branch.

As you can observe, we repeat all of the procedure again to find out the next node that will come underneath the low salary attribute, and then we again check whether we have a pure subset or not. Then we repeat our quest for the next node until we reach the last attribute. Oh god! What can we do to reduce our workload? Of course, we have sufficient codes to get nodes from different subsets, so we can use them recursively to build the whole tree out of our dataset. Let's see how to write the code for that.

So now we have a code for creating subtables out of our dataset for a specific node (attribute) and its branches. If we put these methods all together, we can actually build a tree! Yup, we can. Let's add a method to our code to build a tree using the preceding methods:

```
| def buildTree(df, tree=None):
```

```

#Here we build our decision tree
#Get attribute with maximum information gain
node = getNode(df)

#Get distinct value of that attribute e.g
Salary is node and Low,Med and High are values
attValue = np.unique(df[node])

#Create an empty dictionary to create tree
if tree is None:
    tree={}
    tree[node] = {}

#Loop below is written for building tree
using recursion of the function,We will create
subtable of
#each attribute value and try to find
whether it have a pure subset or not, if it is
a pure subset we
#will stop tree growing for that node. if it
is not a pure set then we will. again call the
same
#function.
for value in attValue:
    print("Value: %s"%value)
    subtable = getSubtable(df,node,value)
#Get subtable for the attribute and value
    clValue,counts =
np.unique(subtable['Class'],return_counts=True)

    if len(counts)==1:
#Checking purity of subset
        print("Class: %s\n"%clValue)
        tree[node][value] = clValue
    else:
        tree[node][value] = buildTree(subtable)
#Recursion of the function

return tree

```

As you can see, we are using a dictionary

data structure to create our tree, which gives us the advantage of accessing values using keys. So if we want to know about the branches of any node, we can simply find out using the node value.

Now we will use the preceding function to go through the next steps, and see how theory and practice get a strong bond together:

```
| tree = buildTree(df)
```

In the following discussion, I will use the intermediate results from the preceding execution, which will explain to you both the working of the algorithm and the code. We have reached up to:

	Value: Low			
	Salary	Sex	Marital	Class
0	Low	Male	Unmarried	No
1	Low	Female	Unmarried	No
2	Low	Male	Married	Yes

Choosing between the Sex and Marital attributes for the low salary group

As you can see in *Table 2.8*, there are two males in this salary group. One of them is an investor. There is only one female and she is not investing anywhere. *Table 2.9* shows a summary of the attribute. We will first calculate the entropy for the subset from (2.1), which is 0.9234:

Values	Y
<i>M</i>	<i>1</i>

F	0
$E_{sex} = 0.6666$	
$IG_{sex} = 0.2568$	

Table 2.9: Statistics for the Sex attribute

Next we will calculate individual information gains for the values `Male` and `Female` to calculate the entropy of the attribute using (2.2) and (2.3); the value of entropy will be 0.6666 and Information gain will be 0.2568 .

Similarly we will calculate these values for `Marital` status as follows.

Again in the reference of *Table 2.8*, there are one married and two unmarried persons in this salary group; only one of them is an investor and that is the married one. We can summarize our values in the form of the following table:

Values	Y	N	Information gain
M	1	0	0
UM	0	2	0
$E_{marital} = 0$			
$IG_{marital} = 0.9234$			

Table 2.10: Statistics for the Marital attribute

After calculating information gain for both the attributes, we will get the following values out of it:

Attribute

Sex
Marital

Table 2.11: A summary of information gain calculation

Information Gain for Salary: 0.00 and Entropy: 0.92
Information Gain for Sex: 0.25 and Entropy: 0.67
Information Gain for Marital: 0.92 and Entropy: -0.00

You see our winner is **Marital** status, and it will be the next node of our tree under the **Low** salary branch. So, now our tree will look like this:

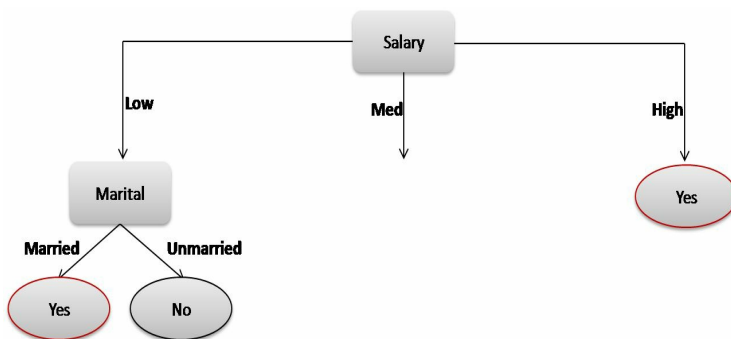


Figure 2.3 Marital Node

Following condition will happen when we will choose `Marital` status as the node, it will have two branches as `Married` and `Unmarried`, which will hold the class values:

```
Winner attribute is: Marital
Value: Married
    Salary Sex  Marital  Class
0  Low    Male  Married   Yes
Class: ['Yes']

Value: Unmarried
    Salary Sex  Marital  Class
0  Low    Male  Unmarried No
1  Low    Female Unmarried No
Class: ['No']
```

As you can see in our tree, `Marital` status will have two branches (`Married` and `Unmarried`), creating a pure subset. It directly shows that if a low-salaried person is married, then he will be an investor, whereas there will be a lesser probability of investing when he/she is unmarried.

Next, we will inspect the `Med` salary group for its next node. Again, we have two options for that: `Sex` and `Marital` status. Let's first create a

subset for the Med salary group:

SN	Name	Salary	Sex
1	Person 2	Med	M
2	Person 3	Med	M
3	Person 4	Med	F
4	Person 5	Med	M
5	Person 9	Med	F

Table 2.12: Subset for the Med salary group

Choosing between the Sex and Marital attributes for the Med salary group

Table 2.12 shows us that there are three males and two females belonging to the medium salary structure, and three out of them are investors. Let's calculate the entropy and the information gain for this attribute:

Values	Y	N	Attribute entropy
<i>M</i>	2	1	0.9234
<i>F</i>	1	1	1

$E_{sex} = 0.9693$
$IG_{sex} = 0.0169$

Table 2.13: A summary of the Sex attribute for the Med salary group

The entropy of the subset (*Table 2.12*) is *0.9709*, the entropy of the attribute is *0.9693*, and information gain is *0.0169*.

Marital status

Table 2.12 shows us that there are three married and two unmarried persons in the medium-salaried subset. Two married persons are investors while only one unmarried investor is present. The summary of the calculation is:

Values	Y
M	2
UM	1
$E_{marital} = 0.9693$	
$IG_{marital} = 0.0169$	

Table 2.14: A Summary of the Sex attribute for the Med salary group

Following is a summary table for the information gains of both the attributes:

Attribute	IG
Sex	0.0169
Marital	0.0169

Table 2.15: Information gain summary

Wait!!! What! Both the attributes have the same information gain! How is that possible? Yes, it is possible. If you look closely at *Table 2.13* and *Table 2.14*, you will get to know that there are exactly the same number of investors for each of their respective attribute values. So what do we do now? Well, there's nothing to worry about. These

same information gains indicate to us that we may choose any one of them as our next parent node. We will choose the `sex` attribute as the next node (you can try `Marital` also; it will not affect our final outcome).

Let's see how:

```
Information Gain for Salary: 0.00 and Entropy: 0.97  
Information Gain for Sex: 0.02 and Entropy: 0.95  
Information Gain for Marital: 0.02 and Entropy: 0.95  
Winner attribute is: Sex
```

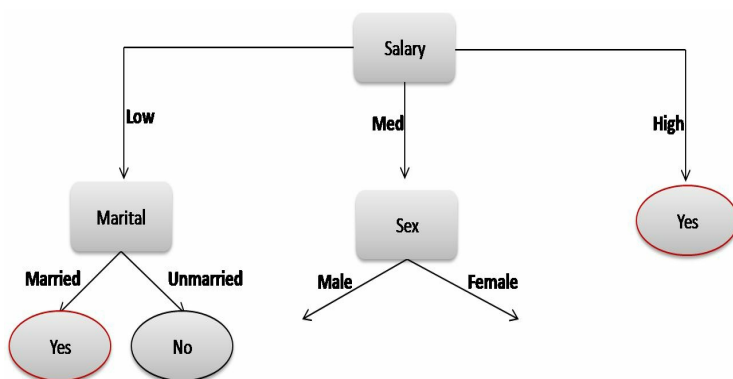


Figure 2.4: Adding the Sex attribute as a node to the Med salary group

The preceding figure shows that there will be

two branches out of the **Sex** attribute, that is, **Male** and **Female**. Now we need to choose the next attribute under these branches; let's do it. But wait!!! Do we really need to calculate anything from here? We are left with **Marital** status only, so, of course, it will be our next node, and this node (**Marital** status) will be common in both of the branches (**Male** and **Female**). Why? let's analyze the table for **Male** and **Female** values:

```
Value: Male
  Salary Sex  Marital  Class
0  Med   Male  Unmarried No
1  Med   Male  Married   Yes
2  Med   Male  Married   Yes

Information Gain for Salary: 0.00 and Entropy:
0.92
Information Gain for Sex: 0.00 and Entropy:
0.92
Information Gain for Marital: 0.92 and Entropy:
-0.00

Winner attribute is: Marital
Value: Married
  Salary Sex  Marital Class
0  Med   Male  Married Yes
1  Med   Male  Married Yes
Class: ['Yes']

Value: Unmarried
  Salary Sex  Marital  Class
0  Med   Male  Unmarried No
Class: ['No']
```

Sex	Marital status	Investor
<i>M</i>	<i>UM</i>	No
<i>M</i>	<i>M</i>	Yes
<i>M</i>	<i>M</i>	Yes

Table 2.16: Investor status of Male investores

For Female branch output will be:

```

Winner attribute is: Sex
Value: Female

    Salary  Sex    Marital    Class
0  Med    Female  Married    No
1  Med    Female  Unmarried  Yes

Information Gain for Salary: 0.00 and Entropy:
1.00
Information Gain for Sex: 0.00 and Entropy:
1.00
Information Gain for Marital: 1.00 and Entropy:
-0.00

```

For `Marital` node there will be two branches;
of course; `Married` and `Unmarried!!`

```
Winner attribute is: Marital
Value: Married

  Salary  Sex      Marital  Class
0 Med    Female  Married  No
Class: ['No']

Value: Unmarried

  Salary  Sex      Marital  Class
0 Med    Female  Unmarried  Yes
Class: ['Yes']
```

Sex	Marital status	Investor
<i>F</i>	<i>UM</i>	<p>Yes
<i>F</i>	<i>M</i>	No

Table 2.17: Investor status of female investors

If you will look at *Table 2.16*, you will notice that we've got three male persons from the medium salary group; two married males are

investors and the one unmarried male is not likely to be our investor. Similarly, *Table 2.17* shows that if a female is married, she is not interested in investing. Let's create a tree out of the preceding conclusion and see whether we are getting a convergence or not:

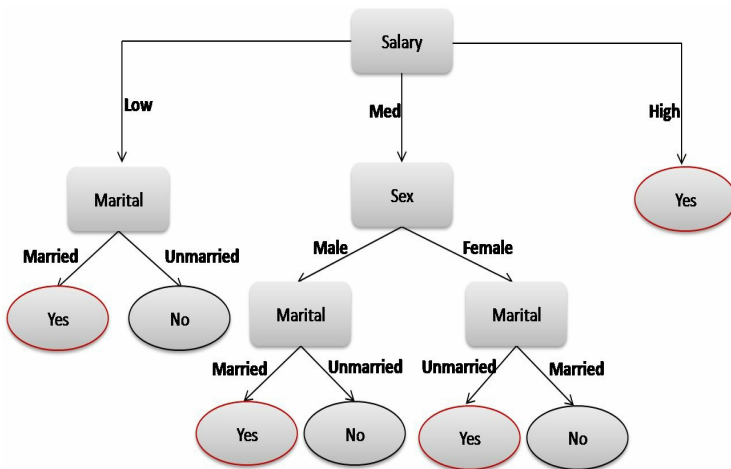


Figure 2.5: Final decision tree

Congratulations!!! We have got our first decision tree; now we can predict whether a person will invest through our firm or not. Let's look at the tree model generated by our code:

```
{
  'Salary': {
    'High': array(['Yes'],
                  'Low': {
    'Marital': {
      'Married': ['Yes'],
      'Unmarried': ['No']]},
    'Med': {
      'Sex': {
        'Female': {
          'Marital': {
            'Married': ['No'],
            'Unmarried': ['Yes']},
          'Male': {
            'Marital': {
              'Married': ['Yes'],
              'Unmarried': ['No']}}}}}}}]
```

You can see we've got exactly what we want. The root node is the first key of our dictionary, whose values are `High`, `Low`, and `Med`; the preceding structure is known as a **cascaded dictionary**.

Now, once we have created our model, we store it on disk, and then it can be used to find our investor. Let's see how to do that:

```
def predict(inst,tree):
    #This function will predict an input
    instance's class using given tree

    #We will use recursion to traverse through
    the tree same as we have done in case
    #of tree building
    for nodes in tree.keys():
        value = inst[nodes]
        tree = tree[nodes][value]
        prediction = 0
        if type(tree) is dict:
```



```

        prediction = predict(inst, tree)
    else:
        prediction = tree
        break;

    return prediction

```

Whenever we execute the preceding method, it will find the root node in our instance and then it will start traversing key by key into the tree recursively as we did at the time of building our tree.

Let's try to predict an instance from our input data:

```

inst = df.ix[2]
print(inst)

```

Instance will look like;

	Name	Salary	Sex	Marital	Class
2	Person 2	Med	Male	Married	Yes

As you can see, we have selected instance number 2. Before feeding this instance to our model, we have to apply a preprocessing step to our data to remove the `Name` and `Class` attributes from it; we can write a small method to do this:

```

def preProcess(dataset):
    #Create a dataframe out of our dataset with attribute names
    df = pd.DataFrame(dataset, columns=[
        'Name', 'Salary', 'Sex', 'Marital', 'Class'])

    #Remove name attribute as it is not required for the calculations
    df.pop('Name')

    #Make sure last attribute of our dataset must be Class attribute
    cols = list(df)
    cols.insert(len(cols),
        cols.pop(cols.index('Class')))
    df = df.ix[:,cols]

    return df

```

Whenever we feed our instance to the preceding method while creating a model or testing, the preceding method will ensure that there is no `Name` attribute in the instance and the `class` attribute is always in the last column of the dataset.

Now let's test the instance:

```

#Remove its class attribute
inst.pop('Class')

#Get prediction
prediction = predict(inst, tree)
print("Prediction: %s"%prediction[0])

```

After execution we will get:

| **Prediction: Yes**

As you can see, our prediction is correct as instance 2 has an investor.

Case study – car evaluation problem

So we have successfully built a decision tree, but what next? Is it useful in practical scenarios? Will it perform for a real-world dataset? We have put in so much effort to write and understand this code. Now is our exam time. Let's pick up a practical-world dataset and apply that code to build a tree to see whether it is useful stuff or not.

We will use the car evaluation dataset, which is available at <https://archive.ics.uci.edu/ml/datasets/Car+Evaluation>. This dataset was created by Marko Bohanec in June 1997. It was specifically developed for testing constructive induction and structure discovery methods, and it can also be used for multi-attribute decision making (which is what we are actually doing). You can find further information on the dataset web page; we will just discuss the

information that is significant to our purpose.

This is a multi-attribute dataset for evaluating a car: whether it is in an acceptable, unacceptable, good, or very good condition. There are six parameters (attributes) that are helpful to determine the aforementioned conditions.

The following is a little summary of the dataset:

The attributes in the dataset are:

The model evaluates cars according to the following concept structure:	
CAR	car acceptability
. PRICE	overall price
. . buying	buying price
. . maint	price of the
maintenance	
. TECH	technical
characteristics	
. . COMFORT	comfort
. . . doors	number of doors
. . . persons	capacity in terms
of persons to carry	
. . . lug_boot	the size of luggage
boot	
. . safety	estimated safety of
the car	

The number of samples in the dataset:

```
| Number of Instances: 1728  
| (instances completely cover the attribute  
| space)
```

The attribute values are:

```
| Number of Attributes: 6  
| Attribute Values:  
  
| buying      v-high, high, med, low  
| maint       v-high, high, med, low  
| doors       2, 3, 4, 5-more  
| persons     2, 4, more  
| lug_boot    small, med, big  
| safety      low, med, high
```

Following is the distribution of the classes:

```
| Class Distribution (number of instances per  
| class)  
  
| class      N      N[%]  
| -----  
| unacc      1210   (70.023 %)  
| acc        384    (22.222 %)  
| good       69     ( 3.993 %)  
| v-good     65     ( 3.762 %)
```

Before moving on to creating our tree, we will add two more functions to our code; one of them will help us create data for training and testing from the available dataset, and the

other one will be used to evaluate the model:

```
def split_data(df,percentage):  
  
    #First get the split index using percentage  
of data required for training  
    split_indx =  
    np.int32(np.floor(percentage*len(df.index)))  
  
    #We will shuffle the rows of data to mix  
out its well  
    df =  
    df.sample(frac=1).reset_index(drop=True)  
  
    #split training data for creating tree  
    train_data = df[:split_indx]  
    temp = df[split_indx:len(df.index)]  
    temp = temp.as_matrix()  
    test_data =  
    pd.DataFrame(temp,index=range(len(temp)),columns  
    [key for key in df.keys()])  
  
    return train_data,test_data
```

The preceding code block will divide our data into training and test sets. We will use the training set to build the tree and the test set to evaluate our model. Let's start with building the tree:

```
cardata = pd.read_csv("Path where dataset  
stored on the disk")  
  
#Convert data into matrix form  
mat = cardata.as_matrix()  
  
#Add column attributes to the data
```

```

df = pd.DataFrame(mat,columns=
['buying','maint','doors','persons','lug_boot','

#Create data split with the given percentage
trainData,testData = split_data(df, 0.991)

#Store training and test data into csv files
for future usage
trainData.to_csv(trainDataPath,columns=
['buying','maint','doors','persons','lug_boot','
testData.to_csv(testDataPath,columns=
['buying','maint','doors','persons','lug_boot','

#Lets Create the tree
tree = buildTree(trainData)
pprint.pprint(tree)

{'safety': {'high': {'persons': {'2': 'unacc',
                                '4':
{'buying': {'high': {'maint': {'high': 'acc',
                                'low': 'acc',
                                'med': 'acc',
                                'vhigh': 'unacc'}}},....
#Store the model on the disk as a json file
import json
with open(path2save,'w') as f:
    json.dump(tree,f)

```

The preceding block shows a very small part of our trained tree model.

Now we will test the model using our test data; we will add a small code for getting the predicted output for multiple instances:


```
def BatchTest(instances, tree):
    prediction = []
    instances.pop("Class")
    for i in range(len(instances.index)):
        inst = instances.ix[i]
        pred = predict(inst, tree)
        prediction.append(pred)
    return prediction
```

After getting the predictions for our test data, we will evaluate the accuracy of the model via this function:

```
def getAccuracy(testClass, predictedClass):
    match = 0
    for i in range(len(testClass)):
        if testClass[i]==predictedClass[i]:
            match+=1

    accuracy = 100*match/len(testClass)

    return accuracy, match
```

Now, as we have the model and evaluation code, we can test our model against the test data we have created, as follows:

```
#Lets load the model we have stored
with open(path2save) as f:
    model = json.load(f)
#Extract actual class values out of test data
actualClass = testData['Class']

#Get predictions for each instance in the training data
predictions = BatchTest(testData, model)
accuracy, match = getAccuracy(actualClass,
```

```
| predictions)
|
| print("Accuracy of the model is: %.2f and
| matched results are %i out of %i"%
| (accuracy, match, len(actualClass)))
|
| Accuracy of the model is: 93.75 and matched
| results are 15 out of 16
```

As you can see, we have predicted 15 out of 16 instances correctly from our simple decision tree!!

Summary

We learned how to create a decision tree out of our dataset, code it, and then use this algorithm for the practical usage. There are innumerable applications of decision trees, from weather condition prediction to spam alerts from e-mail data. Now, what next?

We worked with the categorical dataset in this chapter; in other words, the attributes of the dataset are mostly non-numeric or discrete values, and most real-world applications involve numerical datasets in which attributes may have fractional values. Whenever we have to deal with a numerical dataset, we cannot rely on the preceding algorithm for such a dataset. We have to work with some more advanced algorithms. In the next chapter, we will learn to create a decision tree for continuous or numerical datasets.

We have seen the power of a single decision tree, which is giving us pretty good accuracy. Just think of some cases where we have many more attribute values, where a single tree cannot fit very well and is not sufficient to predict correct outputs. In such cases, we can use more trees and then combine their predictions to get more accurate outputs out of our model.

Random Forest

What is a random forest? Forest, a dark and scary place with lots of predators who are always seeking their hunt! Don't worry, folks! We are not getting into that forest. We created a decision tree in the previous chapter; now, we will use them to create a forest of such decision trees. We have seen the power of decision trees and that they can well fit on the dataset and predict correct classes with good accuracy. However, the real world is not as easy as the conceptual world; you will not always get an ideal categorical dataset with only a few attribute values. Real-world data may be far bigger than you think. It may have millions of instances (such as a dataset of a census analysis) with many more attributes (in hundreds). In such cases, it is impossible to use a single decision tree to get predictions. So, what do you think is the solution? Yes!

We have to **divide and conquer**; you get it, right? Increase the number of trees and divide the dataset into different trees.

Creating multiple trees out of a single dataset and then combining the predictions of all of them is known as **tree bagging**, which is a widely used method by the machine learning community. This group of multiple trees is known as a decision tree forest, and when we create this forest using randomly selected samples (in combination with randomly selected features) out of our dataset, this forest is known as a **random forest**.

This figure gives you an idea about bagged decision trees:

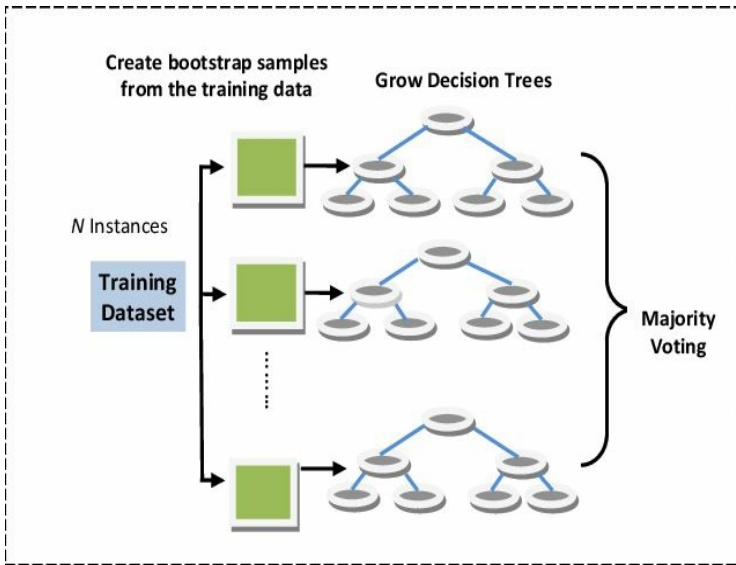


Figure 3.1: Random forest architecture

Let's try to understand it with the help of a problem. Do you remember we opened a fund adviser company in the last chapter and it can successfully choose its target customers using a decision tree? Now, after some months (or years) of effort, we are the topmost investment adviser group and want to expand our business throughout the country. So when we start expanding our business in different regions of our country, we will meet different kinds of customers. And there will be more

parameters to add to our dataset, such as the financial status of a state, city, or village, educational status of the targeted region (number of uneducated, graduate, and postgraduate investors, and so on), and many more factors such as urban and rural lifestyle. So, there are many different parameters possible to create a strong dataset through which we can reach our target investors.

When we increase the number of parameters in our dataset, it becomes more complex, not only due to the more number of parameters, but also due to the need to include numerical values. So this dataset will not suit the decision tree algorithm learned in [Chapter 2, *Decision Trees*](#). What! So, what we will do? One solution is to create a tree that can handle numerical values to create nodes and branches. How to do it? The answer is the **Classification and Regression Trees (CART)** algorithm, which is widely used to create decision trees. It is also the building block of our random forest algorithm.

Classification and regression trees

CART was proposed by Leo Breiman to the machine learning community. Classically, this algorithm is known as **decision trees** only, but in the modern-day community, some programming languages refer to it as CART. This algorithm is the cornerstone of the ensemble machine learning system, like bagging and boosting.

The representation of CART is the binary tree only, and this is the same binary tree that we all have learned in data structures. Let's have a brief review of the binary tree algorithm.

Binary trees are different from other trees in the sense that in a normal tree structure, there may be any number of children for a parent node (including root); but in a binary tree, as its name suggests, any parent node can have a

maximum of two branches (or nodes). This figure shows a representation of a simple binary tree:

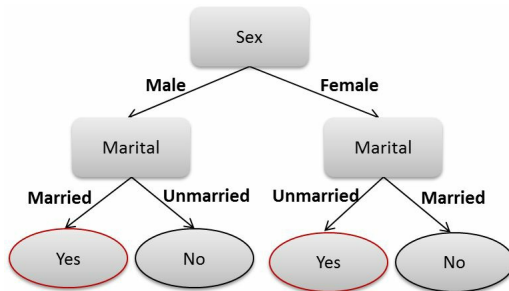


Figure 3.2: Decision tree with categorical data

This tree is a subtree that we created in the previous chapter. Here, you can see that each node has a maximum of two children. One more thing to add in here—when we work with numerical values as attribute values, our binary tree will look like this:

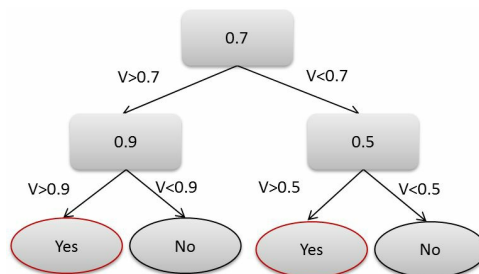


Figure 3.3: Decision tree with a numerical dataset

If you look closer at the preceding tree, you'll see that we have changed our attribute values from categorical to numerical. So, the criteria of node splitting is going to be based on numerical values only. There is an important property of binary trees: one branch will have values lesser than the parent node and the other will have them higher than the parent node. As you can see in the figure, all children on the left have values more than their parent nodes, while the children on the right have lower values than the parent nodes. This important criteria is used to divide data using an anchor point and create a subset out of it. Later on, we can check whether this subset can be divided further or not.

Now, let's try to create a simple binary tree out of a simple dataset to understand its concept.

Suppose we have the following array of values:

```
| data =  
| [0.7, 0.65, 0.83, 0.54, 0.9, 0.11, 0.44, 0.35, 0.75, 0.3,
```

After executing the preceding line, we will have a list of numbers like this:

```
| data = [0.7, 0.65, 0.83, 0.54, 0.9, 0.11, 0.44,  
| 0.35, 0.75, 0.3, 0.78, 0.15]
```

Now, as we know that a binary tree may have a maximum of two branches (`left` and `right`), we can write it in the form of a Python dictionary with key values as `left` and `right`. There will be one more field called `data`, where we can store our data. Each node in our binary tree will have these three fields. Let's write a method to create a node with the aforementioned key values:

```
| def getNewNode(data):  
|     node = {'data': [], 'left': [], 'right': []}  
|     node['data'] = data  
|     return node
```

As you can see, our dictionary node has three fields where `data` will hold the node value at a particular node, and `left` and `right` will hold the next nodes.

Let's call the preceding method to create a root node for our tree; we will choose `median` of the preceding array as our root node. Let's create it:

```
| med = np.median(data)
| print("Median of array is: %.2f"%med)
| tree = getNewNode(med)
|
| print(tree)
```

After executing the preceding lines, we will get the following dictionary:

```
| {'left': [], 'data': 0.59, 'right': []}
```

Now, as we have our root node, we can start building our tree out of it. We will take array values one by one using a `for` loop; we will put values less than the parent node in the `left` branch, while values greater than the parent node will go to the `right` branch. The following is the method to do this:

```
| def createBinaryTree(tree,data):
|
|     #Check whether we have any node in the tree
|     if not create one
|         if not tree:
|             tree = getNewNode(data)
```

```

    #Now if current value is less than parent
    node put it in left
    elif data<=tree['data']:
        tree['left'] =
createBinaryTree(tree['left'],data)

    #else put it in right
    else:
        tree['right'] =
createBinaryTree(tree['right'],data)
    return tree

```

We will use recursion to create our tree. Recursion is the way to call the same function again and again to complete repetitive tasks, As you can see in the preceding method, we are calling `createBinaryTree` inside the same function to build a tree under a tree; each call to `createBinaryTree` will add a subtree to the respective node.

Now that we have sufficient code to create a binary tree for our small array, let's create `tree` out of it:

```

for i in range(len(data)):
    value = data[i]
    tree = createBinaryTree(tree,value)
import pprint
pprint.pprint(tree)

```

As we have already created a root node of our

tree, we will now start extracting values from the array and use them to create `tree`. After the execution of the preceding block, we will have a Python dictionary; it will have our entire tree structure, as follows:

```
{'data': 0.59,
  'left': {'data': 0.54,
            'left': {'data': 0.11,
                      'left': [],
                      'right': {'data': 0.44,
                                'left': {'data':
0.35,
                                     'left':
{'data': 0.3,
  'left': {'data': 0.15,
  'left': [],
  'right': []},
  'right': []},
                                     'right':
[]},
                                'right': []}},
            'right': {'data': 0.7,
                      'left': {'data': 0.65, 'left': [],
                                'right': {'data': 0.83,
                                            'left': {'data': 0.75,
                                                        'left': [],
                                                        'right': {'data':
0.78, 'left': [], 'right': []}},
                                            'right': {'data': 0.9,
                                                        'left': [], 'right': []}}}},
                      'right': []}}}
```

So, this is the tree we have built out of our array.

The first question that can come to your mind is, now what? What to do with this tree?

Well, binary trees are extremely useful whenever you need to search for any element in them. However, binary search is out of our scope; we will see how a binary tree will help us make our decision tree.

So, to make a decision tree out of a binary tree algorithm, what should be required to add to our code? Let's think about it:

- We should have some metric on the basis of which we can decide what should be the value of our node (including the root node)
- We must know where to split a tree branch
- We must know some stopping criteria of our tree growing process, or it may grow for infinite time

What do I mean by some metric? Well, we used information gain (using the attribute entropy) to select nodes in the previous chapter and that technique showed us some promising results. But that criteria to check the impurity of a subset is well defined for categorical data rather than numerical. So what should we use for our test case? Well, we will be using Gini index as our metric to quantify purity and choose the value of the node.

Gini index for impurity check

Gini index is the cost function used to evaluate splits (nodes) in the dataset.

Whenever we use the word **split**, it belongs to the node, so each split has two important aspects; the first is attribute and the second is attribute value, which will divide our data into two groups.

Gini score can be used to know the impurity of our dataset just like the information gain of the attribute. But here, it works a bit differently; Gini score suggests how well the dataset can be separated. Let's understand this with an example.

Suppose we have a system that generates 0 at the output whenever the input is less than 0 and generates 1 at the output whenever the input is greater than or equal to 0. We want to

write it in the form of an equation:

```
if x>=0:  
    y= 1  
else:  
    y = 0
```

If we test a series of input, we can get the result as follows:

X	Y
-1.2	0
-3.2	0
2.1	1
1.5	1

Table 3.1

So, as you can see in the preceding table, whenever the input value is less than 0, the

output is 0. And whenever the input is greater than 0, the output is 1. This leads us to creating a perfect separation of the preceding data into two groups.

There may be one more case when there is not necessarily a perfect separation (or split); for example, suppose we modify the preceding system as follows:

```
if (x+2)>=0:  
    y= 1  
else:  
    y = 0
```

Our input-output table will change to this:

X	Y
-1.2	1
-3.2	0
2.1	1

1.5	1
-----	---

Table 3.2

This table does not have a perfect separation of values, unlike the previous case; so how can Gini score help us find the purity of split?

To find Gini score from a dataset, our input ingredient will be:

- The class proportion
- The number of instances belonging to a group

The class proportion can be calculated using the following formula:

$$\text{Proportion} = \frac{\text{class_value_count}}{\text{number_of_instances_in_the_group}}$$

So, from the preceding formula, the total class proportion for table 3.1 will be 0.5, and if we consider two groups of less than 0 and greater than 0, we will get a class proportion

of 1 for each group.

The formula for Gini score is:

```
| Gini_index = sum(proportion*(1.0-proportion))
```

Can we write code for calculating Gini index? Of course, yes!

```
import numpy as np

# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):

    #Initialize Gini variable
    gini = 0.0

    #Calculate proportion for each class
    for class_value in class_values:

        #Extract groups
        for group in groups:

            #Number of instance in the group
            size = len(group)
            if size == 0:
                continue

            #Initialize a list to store class
            index of the instances
            r = []

            #get class of each instance in the
            group
            for row in group:
                r.append(row[-1])

            #Count number of instances belongs
```

```

to current class
    class_count = r.count(class_value)

    #Calculate class proportion
    proportion =
class_count/float(size)

    #Calculate Gini index
    gini += (proportion * (1.0 -
proportion))

    return gini

```

The preceding function will take a group of instances and distinct class values at the input and return the Gini index for the dataset.

Let's test the preceding code for *Table 3.1* and *Table 3.2* and try to find out the Gini index for each.

We can divide our dataset into two groups, that is, negative input values and positive input values; we will put our data into an array as follows for *Table 3.1*:

```
| data1 = [[[-1.2,0],[-3.2,0]], [[2.1,1],[1.5,1]]]
```

As there are two class values:

```
| classes = [0, 1]
| print("Gini index for Table 1 dataset is:
```

```
|%.2f"% gini_index(data1, classes))
```

Let's execute the preceding lines and see what we get for *Table 3.1*:

```
| Gini index for Table 1 dataset is: 0.00
```

Similarly for *Table 3.2*:

```
| data2 = [[[-1.2,1],[-3.2,0]],[[2.1,1],[1.5,1]]]  
| classes = [0, 1]  
| print("Gini index for Table 2 dataset is:  
| %.2f"% gini_index(data2, classes))
```

```
| Gini index for Table 2 dataset is: 0.50
```

So, you see here that *Table 3.1*, as we know, has a perfect split set, where two groups are well separated. Whereas *Table 3.2* has an overlapping group and the Gini index for that is 0.5.

Node selection

We are ready with the performance evaluation metric, which will help us choose node values for our decision tree; we want to choose a node value that can split our data with the lowest possible Gini score. To do this, we will go through the following steps:

1. Choose an arbitrary value from the attribute
2. Use this value as a threshold, and create two groups from the attribute values such that one group will have values less than the threshold and the other group will have values greater than or equal to the threshold
3. Calculate the Gini index for the groups
4. Choose the value that gives the highest Gini score as the node

Now we will add the preceding steps in our code and see how it works.

Creating a split

We will write a function that will take the `threshold` value and `dataset` as input arguments to create two groups:

```
def createSplit(attribute,threshold,dataset):  
    #Initialize two lists to store the sub  
    sets  
        lesser, greater = list(),list()  
    #Loop through the attribute values and  
    create sub set out of it  
        for values in dataset:  
            #Apply threshold  
            if values[attribute]<threshold:  
                lesser.append(values)  
            else:  
                greater.append(values)  
    return lesser,greater
```

Let's test the preceding code block for separating negative and positive values from `dataset` we have used earlier:

```
| data = [[-1.2,0],[-3.2,0],[2.1,1],[1.5,1]]
```

Let's call our function to split the preceding data using threshold 0:

```
[lesser,greater] = createSplit(0, 0, data)
print('Group of negative values: ',lesser)
print('Group of positive values: ',greater)
```

After execution, we will have:

```
Group of negative values:  [[-1.2, 0], [-3.2,
0]]
Group of positive values:  [[2.1, 1], [1.5, 1]]
```

Congratulations! We are on the right track.

Now, to choose an attribute and its values as a node, we have all of the ingredients in our hands. We have a function that can create groups of attribute values. We have a metric that can tell us whether the selected attribute value can be used as the node or not. So, we have to combine them all to choose a node and its value; we will add the following function to our code, which will give us the node value out of our dataset:

```
def getNode(dataset):
    #Create a variable to store class values of
    instances
    class_values = []
    #Loop through each row and find out class
    of instance
    for row in dataset:
        class_values.append(row[-1])
    #initialize variables to store gini score,
```

```

attribute index and split groups
winnerAttribute = sys.maxsize
attributeValue = sys.maxsize
gScore = sys.maxsize
leftGroup = None
#Run loop to access each attribute and
attribute values
    for index in range(len(dataset[0])-1):
        for row in dataset:
            groups = createSplit(index,
row[index], dataset)
            gini = gini_index(groups,
class_values)
            print('A%d <- %.1f Gini=%.1f' %
((index+1),
                row[index], gini))
            if gini < gScore:
                winnerAttribute,
attributeValue, gScore,
                leftGroup = index, row[index],
gini, groups
            #Once done create a dictionary for node
            node =
{'attribute':winnerAttribute,'value':attributeVa

    return node

```

Let's test the preceding code block for a small toy dataset:

SR	A1	A2
1	3.2	1.5

2	1.3	1.2
3	3.7	2.8
4	2.9	2.4
5	3.9	1.9
6	7.5	3.5
7	9.0	3.2
8	7.4	0.9
9	9.5	4.2
10	7.3	3.5

Table 3.3

The preceding table shows a sample dataset for which we will calculate the root node; for that, we have to call our function, `getNode`, which will internally call `gini_index` and `createSplit` to get the best split attribute and its value. Let's try it:

```
dataset = [[3.2,1.5,0],
            [1.3,1.2,0],
            [3.7,2.8,0],
            [2.9,2.4,0],
            [3.9,1.9,0],
            [7.5,3.5,1],
            [9.0,3.2,1],
            [7.4,0.9,1],
            [9.5,4.2,1],
            [7.3,3.5,1]]
#Call the function here</strong>
node = getNode(dataset)
```

output of previous execution will be:

```
A1 <- 3.2 Gini=2.0
A1 <- 1.3 Gini=2.5
A1 <- 3.7 Gini=1.4
A1 <- 2.9 Gini=2.3
A1 <- 3.9 Gini=0.0
A1 <- 7.5 Gini=2.3
A1 <- 9.0 Gini=2.5
A1 <- 7.4 Gini=2.0
A1 <- 9.5 Gini=2.5
```

```
A1 <- 7.3 Gini=1.4  
A2 <- 1.5 Gini=4.7  
A2 <- 1.2 Gini=5.0  
A2 <- 2.8 Gini=1.4  
A2 <- 2.4 Gini=3.2  
A2 <- 1.9 Gini=4.1  
A2 <- 3.5 Gini=2.5  
A2 <- 3.2 Gini=2.0  
A2 <- 0.9 Gini=2.5  
A2 <- 4.2 Gini=2.5  
A2 <- 3.5 Gini=2.5
```

When we call the `getNode` function, it iterates for each attribute and its values. Each value is first considered as a threshold for creating groups, and then these groups will get evaluated using the Gini score; an attribute with the lowest Gini score will be chosen as the node in the decision tree.

As you can see in the printed results, the value of 3.9 from the `A1` attribute has the lowest Gini score. If you look closely at our dataset, you'll realize that all instances in `A1` that are less than 3.9 have class attribute 0; others have 1. This gives us the best split; one can stop growing the tree here and can choose it as the classifier.

Tree building

We have all the required ingredients in our hand to create a decision tree, so what are we waiting for? Let's start our own decision tree to get an understanding of its working; we will create a toy dataset for RGB pixel values to build a tree. Our target is to segment out two different color pixel values:

SN	RED	GREEN
1	0.61	0.31
2	0.29	0.03
3	0.66	0.39
	0.32	0.07

4		
5	0.64	0.35
6	0.67	0.36
7	0.40	0.12
8	0.69	0.38
9	0.25	0.03
10	0.48	0.19

Table 3.4 RGB pixel values

However, before we go ahead and build our tree, we should know when to stop growing it. In many cases where we start building the

tree for larger datasets, we may include some redundant nodes, which unnecessarily increase the size of our tree. To overcome this problem, we will add two criteria to determine the stopping of tree building. One is the maximum depth (or number of nodes under the root) and the other is to determine the minimum number of instances to be inspected by any node. So, before adding a node to our tree, we will check these two criteria and decide whether to add them or not.

What if we reach the threshold of our depth or the number of minimum instances? Or what if we get a subset in which all instances belong to the same class? Then what should we do? Well, in that case, the current node will hold the value of the maximum occurring class in the subset. We can write a code block for it:

```
def terminalNode(dataset):  
    #Create a variable to store the class value  
    and count the class occurrence  
    classes = []  
    for row in dataset:
```

```
|         classes.append(row[-1])  
|     return max(set(classes), key=classes.count)
```

So, whenever we reach the thresholds of depth or minimum instances, we will call the preceding block to assign the class label to the node.

It's time to add the final page of the story to build our tree, but first let's check our ingredients of tree building. We will follow these steps:

1. Loop through each attribute and its value
2. Choose this value as the threshold and create two subsets out of your data
3. Evaluate these subsets by finding their Gini score
4. Choose the value with the lowest Gini score as the root node
5. Repeat the preceding steps until you reach the maximum depth or minimum instance count or the pure subset

Let's write a function for building our tree:

```
| def buildTree(node, max_depth, min_size,
```

```

depth):
    #Lets get groups information first.
    left, right = node['groups']
    del(node['groups'])
    # check if there are any element in the
    left and right group
    if not left or not right:
        #If there is no element in the groups
        call terminal Node
        combined = left+right
        node['left'] = terminalNode(combined)
        node['right']= terminalNode(combined)
        return
    # check if we have reached to maximum depth
    if depth >= max_depth:
        node['left']=terminalNode(left)
        node['right'] = terminalNode(right)
        return
    # if all okey lest start building tree for
    left side nodes
    # if minimum instances are done by the node
    stop further build
    if len(left) <= min_size:
        node['left'] = terminalNode(left)
    else:
        #Create new node under left side of the
        tree
        node['left'] = getNode(left)
        #append node under the tree and
        increase depth by one.
        buildTree(node['left'], max_depth,
        min_size, depth+1) #recursion will take place
        in here
        # Similar procedure for the right side
        nodes
        if len(right) <= min_size:
            node['right'] = terminalNode(right)
        else:
            node['right'] = getNode(right)
            buildTree(node['right'], max_depth,
            min_size, depth+1)

```

As you can see, here is the code of our tree building. Now it's time to execute the all functions to build our first CART-based tree and see how it works actually.

Let's start with loading our data table into a NumPy array:

```
dataset = [[0.61, 0.31, 0.54, 0],
            [0.29, 0.03, 0.24, 1],
            [0.66, 0.39, 0.61, 0],
            [0.32, 0.07, 0.29, 1],
            [0.64, 0.35, 0.58, 0],
            [0.67, 0.36, 0.59, 0],
            [0.40, 0.12, 0.35, 1],
            [0.69, 0.38, 0.60, 0],
            [0.25, 0.03, 0.24, 1],
            [0.48, 0.19, 0.41, 0]]
```

Before we start building our decision tree, we need to define its hyperparameters, that is, maximum depth and number of instances for a node. For our problem, we will go for a maximum depth of three nodes before defining the terminal node, and the maximum instances for a node will be set to one. Now, our first task is to get the root node for our decision tree. To do this, we will call the `getNode` function; let's see how it works:

```
| Node = getNode(dataset)
```

And the output after the execution of the `getNode` function:

```
A1 <- 0.61 Gini=2.2
A1 <- 0.29 Gini=1.9
A1 <- 0.66 Gini=2.5
A1 <- 0.32 Gini=1.2
A1 <- 0.64 Gini=2.4
A1 <- 0.67 Gini=2.5
A1 <- 0.40 Gini=0.0
A1 <- 0.69 Gini=2.4
A1 <- 0.25 Gini=2.2
A1 <- 0.48 Gini=1.6
A2 <- 0.31 Gini=2.2
A2 <- 0.03 Gini=1.9
A2 <- 0.39 Gini=2.4
A2 <- 0.07 Gini=1.2
A2 <- 0.35 Gini=2.4
A2 <- 0.36 Gini=2.5
A2 <- 0.12 Gini=0.0
A2 <- 0.38 Gini=2.5
A2 <- 0.03 Gini=1.9
A2 <- 0.19 Gini=1.6
A3 <- 0.54 Gini=2.2
A3 <- 0.24 Gini=1.9
A3 <- 0.61 Gini=2.4
A3 <- 0.29 Gini=1.2
A3 <- 0.58 Gini=2.4
A3 <- 0.59 Gini=2.5
A3 <- 0.35 Gini=0.0
A3 <- 0.60 Gini=2.5
A3 <- 0.24 Gini=1.9
A3 <- 0.41 Gini=1.6
```

As we know, `getNode` will treat each attribute

value as an anchor point, create two groups out of our data, and then calculate the Gini score for each anchor value. The value with the lowest Gini score will be chosen as the node value. So, let's see what we have got in our root node:

```
{'attribute': 0,  
  'groups': ([[0.29, 0.03, 0.24, 1],  
              [0.32, 0.07, 0.29, 1],  
              [0.4, 0.12, 0.35, 1],  
              [0.25, 0.03, 0.24, 1]],  
             [[0.61, 0.31, 0.54, 0],  
              [0.66, 0.39, 0.61, 0],  
              [0.64, 0.35, 0.58, 0],  
              [0.67, 0.36, 0.59, 0],  
              [0.69, 0.38, 0.6, 0],  
              [0.48, 0.19, 0.41, 0]]),  
  'value': 0.4}
```

You can see in the preceding node that we have three fields in it: `attribute`, `groups`, and `value`. The winner attribute in our case is the first attribute, that is, red; and the value that creates the best split is 0.4.

You can clearly see the segmentation of two different classes here as one group has a red value less than 0.4 and all instances have a class value of 1. The other group with red

more than 0.4 has a class value of 0.

What? We have classified our data successfully into two groups using the `root` node only, so should we go ahead from here? Well, this all depends on how complex our problem is. So if we create a tree with depth 1 (with only a root node), it will be sufficient to classify the preceding dataset into two classes. Let's print our function using `pprint` and see what we've got under the `root` node:

```
| root = getNode(dataset)
| buildTree(root, 1, 1, 1)
| pprint.pprint(root)
|
| {'attribute': 0, 'left': 1, 'right': 0,
|  'value': 0.4}
```

Let's see whether our tree is capable of predicting the correct classes for our training dataset; to get predictions out of a decision tree, we have to add a function to our main code:

```
| def predict(node, row):
|     #Get the node value and check whether the
|     attribute value is
|     less than or equal.
|     if row[node['attribute']] <= node['value']:
```



```

        #If yes enter into left branch and
        check whether it has
        another node or the class value.
        if isinstance(node['left'], dict):
            return predict(node['left'],
row)#Recursion
        else:
            #If there is no node in the branch
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

```

The preceding function is easy to interpret as it just matches the attribute values and does recursion to enter into more depth. Let's execute it and see what we get for our training set:

```

for row in dataset:
    prediction = predict(tree, row)
    print('Expected=%d, Got=%d' % (row[-1],
prediction))

```

Output of the preceding code execution is:

```

Expected=0, Got=0
Expected=1, Got=1
Expected=0, Got=0
Expected=1, Got=1
Expected=0, Got=0
Expected=0, Got=0
Expected=1, Got=1
Expected=0, Got=0
Expected=1, Got=1

```

| Expected=0, Got=0

So, as you can see, we have a tree with a `root` node (only) having two branches on `left` and `right`, which classify data into `0` and `1` using `root` value `0.4`. It classifies all of the instances correctly.

But sometimes, there are features in a dataset that cause various overlaps among instances, and because of this, the root node is not sufficient to segment out the data into different classes. Let's make some changes to the preceding RGB dataset and remake it with some complexity.

SN	RED	GREEN	BLUE	CLASS
1	0.95	0.30	0.63	0
2	0.83	0.40	0.61	0

3	0.75	0.25	0.59	0
4	0.63	0.19	0.39	0
5	0.65	0.45	0.45	1
6	0.53	0.30	0.19	1
7	0.32	0.50	0.35	1
8	0.77	0.55	0.41	1

Table 3.5

Let's feed this data into a NumPy array:

```
dataset = [[0.95,    0.30,    0.63,  0],
            [0.83,    0.40,    0.61,  0],
            [0.75,    0.25,    0.59,  0],
            [0.63,    0.19,    0.39,  0],
            [0.65,    0.45,    0.45,  1],
            [0.53,    0.30,    0.19,  1],
            [0.32,    0.50,    0.35,  1],
            [0.77,    0.55,    0.41,  1]]
```

Call the `buildTree` function in the same manner with the same hyperparameters:

```
| root = getNode(dataset)
| buildTree(root, 1, 1, 1)
| pprint.pprint(root)

| {'attribute': 2, 'left': 1, 'right': 0,
|  'value': 0.45}

| #Let's print the results for the training data;
| for row in dataset:
|     prediction = predict(tree, row)
|     print('Expected=%d, Got=%d' % (row[-1],
| prediction))
```

Following is the predicted outputs by the tree:

```
| Expected=0, Got=0
| Expected=0, Got=0
| Expected=0, Got=0
| Expected=0, Got=1
| Expected=1, Got=1
| Expected=1, Got=1
| Expected=1, Got=1
| Expected=1, Got=1
```

Oh! We have got one wrong prediction on the training data. Why? The answer is quite simple: we need more nodes to handle the data. What do you think? Can adding one more depth solve our problem? Let's see:

```
| root = getNode(dataset)
| buildTree(root, 2, 1, 1)
```

```
| pprint.pprint(root)
| {'attribute': 2,
|  'left': {'attribute': 1, 'left': 0, 'right':
|  1, 'value': 0.19},
|  'right': {'attribute': 0, 'left': 0, 'right':
|  0, 'value': 0.95},
|  'value': 0.45}
```

And the results are:

```
| for row in dataset:
|     prediction = predict(tree, row)
|     print('Expected=%d, Got=%d' % (row[-1],
| prediction))
```

Predictions for `depth =2` are:

```
| Expected=0, Got=0
| Expected=0, Got=0
| Expected=0, Got=0
| Expected=0, Got=0
| Expected=1, Got=1
| Expected=1, Got=1
| Expected=1, Got=1
| Expected=1, Got=1
```

Hell Yes! We got the correct predictions by increasing the depth of our tree. How did this happen? To understand this, we have to take a look at the dataset and tree.

At depth – 1 (root node)

Get the `root` node and its group:

```
| root = getNode(dataset)
```

We've got the lowest `Gini` score of 1.3 for attribute number three and value 0.45:

```
| A3 <- 0.45 Gini=1.3
```

Print the `root` node:

```
| pprint.pprint(root)
{'attribute': 2,
 'groups': ([[0.63, 0.19, 0.39, 0],
             [0.65, 0.45, 0.45, 1],
             [0.53, 0.3, 0.19, 1],
             [0.32, 0.5, 0.35, 1],
             [0.77, 0.55, 0.41, 1]],
            [[0.95, 0.3, 0.63, 0],
             [0.83, 0.4, 0.61, 0],
             [0.75, 0.25, 0.59, 0]]),
 'value': 0.45}
```

If you observe the `Gini` score, it suggests that

there is no pure subset found for the root node, and this is also reflected in the groups divided on the basis of the score. As you can see, group one has one wrong instance (the first one) while the second group consists of a pure subset. Thus, we have to move further to create more subsets for group 1 so that we can reach a pure subset.

At depth – 2 (left branch)

As we have got our root node, we can start building the tree using the `buildTree` function. As the `buildTree` function takes node and hyperparameters as the input, we will send the root node and other hyperparameters such as max depth and minimum: `split.maximum depth (2)`, node instances (1), and current depth (that is 1).

Now, `left, right = node['groups']` will extract groups created by the previous node and delete the group information from the dictionary as it is not further required.

In the next step, the program will check whether there is any element in the groups or not. As we have the data in both of the groups, the program will check the depth parameter.

We enter the `left` branch and again call the `getNode` function; now, we get a Gini score of `0.0` for attribute number two and value `0.19`:

```
| A2 <- 0.19 Gini=0.0
```

And the node data is:

```
| {'attribute': 1,  
|   'groups': ([[0.63, 0.19, 0.39, 0]],  
|               [[0.65, 0.45, 0.45, 1],  
|               [0.53, 0.3, 0.19, 1],  
|               [0.32, 0.5, 0.35, 1],  
|               [0.77, 0.55, 0.41, 1]]),  
|   'value': 0.19}
```

Here, you can see that groups created using value `0.19` are separating the data into two pure subsets; now, if we go towards the `left` branch, we will end up with class value `0`, and it's `1` for the `right` branch.

At depth – 2 (right branch)

We have seen earlier that the right-hand-side subset of the root node is a pure subset. Here we need to put one more node under the root node. You may have a question: why are we going to get one more node when we have a pure subset?

The answer is a binary tree is always a balanced binary tree, which means a tree from the left will always be equal to a tree from the right. In other words, each node of a binary tree will have exactly two nodes under it. So, we will call `getNode` on the pure subset too, where we will get:

```
| {'attribute': 0, 'left': 0, 'right': 0,  
| 'value': 0.95}
```

At the end of the exercise, we will get something like this:

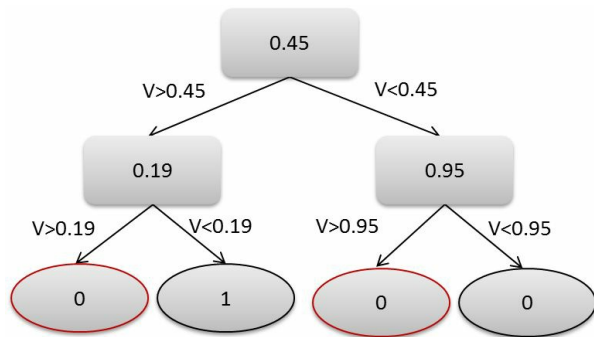


Figure 3.4: Decision tree

Case study – breast cancer type prediction

As you know, the best way to know the performance of an algorithm is to apply it for some practical usage. We will apply our decision tree algorithm to a practical scenario and see how it performs there. We will use data of a breast cancer study that is available online at <https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Original%29>; here is the summary of the dataset:

```
Data set Name: Breast Cancer Wisconsin  
(Original) Data Set  
Number of Instances: 699  
Attributes characteristics: Integer  
Number of attributes: 10  
Number of classes: 2  
Attribute Information:  
1. Sample code number: id number  
2. Clump Thickness: 1 - 10  
3. Uniformity of Cell Size: 1 - 10  
4. Uniformity of Cell Shape: 1 - 10  
5. Marginal Adhesion: 1 - 10
```

6. Single Epithelial Cell Size: 1 - 10
7. Bare Nuclei: 1 - 10
8. Bland Chromatin: 1 - 10
9. Normal Nucleoli: 1 - 10
10. Mitoses: 1 - 10
11. Class: (2 for benign, 4 for malignant)

This data can be downloaded in the form of a .csv file. As the data values are strings, first we have to convert those strings into float type and the class values into integer type. We will have to remove the ID numbers of the samples too. To read data from the .csv file, we will need to add a function to our code as follows:

```
from csv import reader

def readCsv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset
```

As our data has string format as well as some missing data values, first we need to convert

all values to the numerical datatype. We will replace all missing values with 0 at the moment. The following function will help us to do so:

```
def convert_column_to_float(dataset, column):  
    for row in dataset:  
        if row[column]=='?':  
            row[column] = 0  
        else:  
            row[column] =  
float(row[column].strip())
```

We will need one more function to split the dataset into two parts, that is, training data and testing data. We will train our model on the training data and the testing data will be used to validate the model accuracy:

```
def getTrainTestData(dataset, split):  
    training = []  
    testing = []  
    shape = np.shape(dataset)  
    trainlength = np.floor(split*shape[0])  
  
    for i in range(trainlength):  
        training.append(dataset[i])  
    for i in range(trainlength, shape[0]):  
        testing.append(dataset[i])  
  
    return training, testing
```

So, we are ready with all the helping

functions to build a decision tree and get predictions out of it. Let's start:

```
filename = 'breast_cancer_data.csv'
dataset = readCsv(filename)

# Convert attributes to numerical type
for i in range(0, len(dataset[0])):
    convert_column_to_float(dataset, i)

#Now remove index column from the dataset
dataset_new = []
for row in dataset:
    dataset_new.append([row[i] for i in
range(1,len(row))])

#Get training and testing data split
training,testing =
getTrainTestData(dataset_new, 0.7)

#We will build our tree for maximum depth of 3
and with minimum instances of 1
tree = build_tree(training,3,1)
pprint.pprint(tree)
```

Our tree, after training for the given hyperparameters, is as follows:

```
{'attribute': 1,
 'left': {'attribute': 0,
          'left': {'attribute': 7, 'left': 2.0,
                  'right': 4.0, 'value': 8.0},
          'right': {'attribute': 0, 'left':
4.0, 'right': 4.0, 'value': 10.0},
                  'value': 8.0},
          'right': {'attribute': 0, 'left': 4.0,
                  'right': 4.0, 'value': 10.0},
                  'value': 3.0}
```

Now, let's add one more function to evaluate the model performance:

```
def getAccuracy(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0
```

Let's test our model for training and testing data:

```
for row in training:
    prediction = predict(tree, row)
    pre.append(prediction)
    actual = act.append(row[-1])
acc = getAccuracy(act, pre)

print('training accuracy: ', acc)
```

The training accuracy:

```
| 'training accuracy: ', 93.25153374233128
```

Similarly, the testing accuracy:

```
| 'testing accuracy: ', 94.70672389127324
```

So, as you can see, we are getting a decent accuracy for our testing data. This can be increased by increasing the depth of our tree,

but eventually, it will take more computation time to build the tree.

Decision tree bagging

Now you know how a decision tree can be used to make predictions on numerical data. It's time to take a minute and think over it. Can we improve the accuracy we are currently getting? Or can our decision tree avoid the problem of high variance (overfitting)? The answer is surely yes. But how? Well, if you remember [Chapter 1, *Introduction to Ensemble Learning*](#), we had discussed bagging algorithms, where we trained different models on different samples of data and tried to come up with an improved version of a classifier. So, can we use that technique in here too? Surely, yes!

Decision trees have a tendency of high variance, which leads to failure in their generalization. Tree bagging is a technique that can help us solve this problem. But why? Because bagging has its unique feature of sampling; it creates different samples out of

data with replacements. This means one instance may appear in multiple samples. After creating samples from data, we will create different tree models for these samples, and at the prediction stage, we can take the average prediction of all trees.

You can understand the whole process from the following figure:

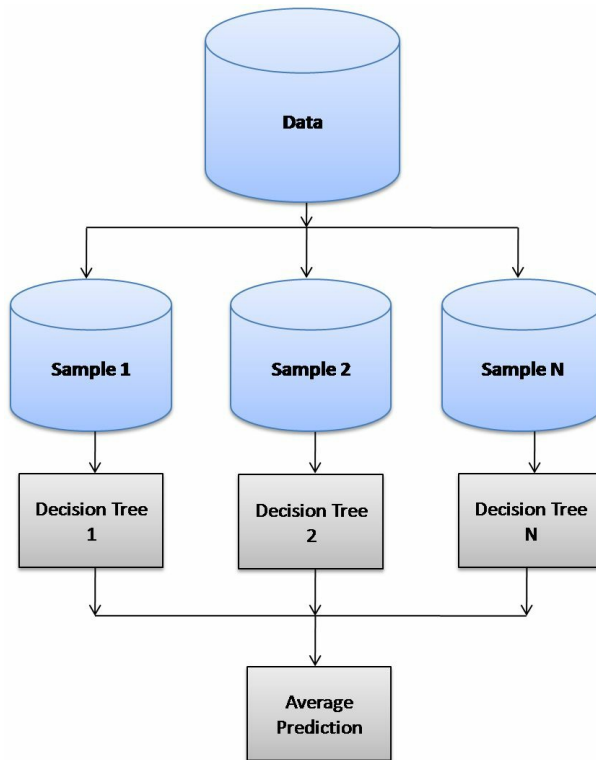


Figure 3.5: Decision tree bagging

The preceding figure shows an interpretation of decision tree bagging, where we can prepare different tree models and combine their results to get the final prediction. The sampling of this data can be understood from our previous `toy` dataset:

SN	RED	GREEN	BLUE	CLASS
1	0.95	0.30	0.63	0
2	0.83	0.40	0.61	0
3	0.75	0.25	0.59	0
4	0.63	0.19	0.39	0
5	0.65	0.45	0.45	1
6	0.53	0.30	0.19	1
7	0.32	0.50	0.35	1
8	0.77	0.55	0.41	1

--	--	--	--	--

Table 3.6

Now, if we want to create samples with replacement of instances from the preceding dataset, these samples will look something like this:

Sample-1

SN	RED	GREEN	BLUE	CLASS
1	0.95	0.30	0.63	0
2	0.83	0.40	0.61	0
8	0.77	0.55	0.41	1

Table 3.7

Sample-2

SN	RED	GREEN	BLUE	CLASS
2	0.83	0.40	0.61	0
3	0.75	0.25	0.59	0
4	0.63	0.19	0.39	0

Table 3.8

Sample-3

SN	RED	GREEN	BLUE	CLASS
2	0.83	0.40	0.61	0
5	0.65	0.45	0.45	1

6	0.53	0.30	0.19	1
---	------	------	------	---

Table 3.9

As you can see in the preceding example, each sample has overlapping instances. When we train our classification models over these samples, each tree learns a different kind of population, which automatically helps to create a more generalized model. As each classifier is sharing some common instances, their predictions are quite correlated, which helps us get improved accuracy over a single tree classifier. This is why bagging classifiers are more famous in scientific communities than single-model-based classifiers.

From bagging to random forest

As we have discussed, bagging is nothing but creating multiple samples out of our dataset and then training different decision trees on those samples. The samples are quite correlated; again, this correlation can cause high variance of the classifier. So, what we can do to make it less correlated? Well, we can choose each subsample with different features. Wait! What does that mean?

Suppose we have a dataset with 10 attributes (as we have seen in the previous example).

We can create a sample out of it with five instances, which will have only three features per sample. Let's understand it with an example. Let's add three more features to our previous pixel data. Where earlier we were having R , G , and B values of pixels, we will now add hue, saturation, and value to the

feature table:

SN	RED	GREEN	BLUE	H	S	V
1	0.95	0.30	0.63	0.12	0.78	0.56
2	0.83	0.40	0.61	0.15	0.68	0.62
3	0.75	0.25	0.59	0.26	0.65	0.60
4	0.63	0.19	0.39	0.24	0.58	0.52
5	0.65	0.45	0.45	0.38	0.45	0.50
6	0.53	0.30	0.19	0.45	0.36	0.47
7	0.32	0.50	0.35	0.52	0.38	0.42

8	0.77	0.55	0.41	0.49	0.27	0
---	------	------	------	------	------	---

Now, if we create random samples from the preceding dataset with random features, it will look something like this:

Sample-1

SN	RED	GREEN	CLASS
1	0.95	0.30	0
2	0.83	0.40	0
6	0.53	0.30	1

Sample-2

SN	H	S	CLASS
----	---	---	-------

1	0.12	0.78	0
3	0.26	0.65	0
4	0.24	0.58	0

Sample-3

SN	BLUE	S	CLASS
1	0.63	0.78	0
3	0.59	0.65	0
4	0.39	0.58	0

Each of our classifiers will learn different instances (with some overlap) with different features, which creates a less correlated but generalized predictor.

So, how do we implement bagging using decision trees? The answer is quite simple. We have to modify our code slightly and add some more helper functions to our previous decision tree algorithm's code. Let's do it.

First, we will change the function `getNode` for calculating the node value; we will add functionality to select limited features from the dataset to calculate the Gini score for node selection:

```
def getNode(dataset, features):  
    class_values = []  
    for row in dataset:  
        class_values.append(row[-1])  
    #initialize variables to store gini score,  
attribute index and split groups  
    winnerAttribute = sys.maxsize  
    attributeValue = sys.maxsize  
    gScore = sys.maxsize  
    leftGroup = None  
  
    #Run a loop to randomly select attribute  
and create a list of attributes  
    features = list()
```

```

        while len(features) < n_features:
            index = randrange(len(dataset[0])-1)

            #Do not repeat same features from the
sample
            if index not in features:
                features.append(index)
            #Run loop to access selected attributes and
their values
            for index in features:

                #for index in range(len(dataset[0])-1):
                for row in dataset:
                    groups = createSplit(index,
row[index], dataset)
                    gini = gini_index(groups,
class_values)
                    if gini < gScore:
                        winnerAttribute,
attributeValue, gScore, leftGroup = index,
row[index], gini, groups

                #Once done create a dictionary for node
                node =
{'attribute':winnerAttribute, 'value':attributeVa

        return node

```

So, you see we added a loop to randomly select features out of the dataset!

To evaluate the performance of our learned model, we will use the k-folds validation strategy. We will create different subsamples (folds) from the data, train a model for each sample, and test that model against each fold

to match the accuracy. We will use mean model error as the performance metric and classification accuracy will be used to evaluate each model. For this purpose, we will add two more helper functions to help us create validation folds and evaluate the algorithm.

The following is the code for creating folds out of dataset:

```
# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):

    #Create an empty list
    dataset_split = list()
    dataset_copy = list(dataset)

    #Define the fold size
    fold_size = int(len(dataset) / n_folds)

    #Run through the loop and create sub sets
    of the dataset
    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index =
randrange(len(dataset_copy))
        fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)

    return dataset_split
```

Now, we will write a function to evaluate the performance of the algorithm:

```
# Evaluate an algorithm using a cross
validation split
def evaluate_algorithm(dataset, algorithm,
n_folds, *args):

    #Create k-folds
    folds = cross_validation_split(dataset,
n_folds)

    #List to store the prediction scores
    scores = list()

    #Run algorithm for each fold
    for fold in folds:
        #get training set from the fold
        train_set = list(folds)

        #Remove current fold from the training
set
        train_set.remove(fold)

        #Combine other k-1 folds into one
dataset
        train_set = sum(train_set, [])

        #create empty list for test dataset
        test_set = list()

        #Create Test set from the fold
instances and remove class label
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None

        #Train model for the training set and
test it on the test set
        predicted = algorithm(train_set,
```



```

test_set, *args)

    #create a list of actual labels
    actual = [row[-1] for row in fold]

    #Calculate accuracy for the current
fold
    accuracy = accuracy_metric(actual,
predicted)
    scores.append(accuracy)
    return scores

```

This is self-explanatory; we are just creating folds, training the models, and then calculating the accuracy of the models. Another thing to consider is creating a combined model using multiple decision trees. For that, we first have to create samples out of our data. And for that, we add the following function to our code:

```

def subsample(dataset, ratio):

    #Create empty list to store the samples
    sample = list()

    #Get the sample size
    n_sample = round(len(dataset) * ratio)

    #Start creating samples out of data
    while len(sample) < n_sample:
        index = randrange(len(dataset))
        sample.append(dataset[index])
    return sample

```

Now, we are almost ready to take-off. But wait! Where is our function for creating the random forest? Don't worry, we'll be adding that in the next few lines.

Before adding the random forest function, we add a function that we will use in the random forest algorithm to make predictions out of multiple trees:

```
def bagging_predict(trees, row):  
    #Get the builded trees and make a list of  
    their predictions  
    predictions = [predict(tree, row) for tree  
in trees]  
  
    #Return the prediction with maximum  
occurance  
    return max(set(predictions),  
key=predictions.count)
```

The following is the random forest's code:

```
def random_forest(train, test, max_depth,  
min_size, sample_size, n_trees, n_features):  
    #Create a list to store decision tree  
models  
    trees = list()  
  
    #Create different dicision trees for  
different samples and features  
    for i in range(n_trees):
```

```

        #Create Sub-sample of data for the
given sample size
        sample = subsample(train, sample_size)

        #Start building tree using buildTree
function
        tree = build_tree(sample, max_depth,
min_size, n_features)

        #Append builded tree to the tree list
        trees.append(tree)

        #Create a list of predictions
        predictions = [bagging_predict(trees, row)
for row in test]
        return(predictions)

```

So, the following are key steps of the preceding code block:

1. Create a subsample of the desired size from the dataset
2. Create one sample for each tree
3. Create k-folds for each sample to evaluate the performance of the model
4. Train a tree for each sample and store it in the tree list
5. At the time of testing, feed an instance through each tree and get a prediction from each tree
6. Choose the prediction coming from the maximum number of the trees

Now, let's use the preceding code to test real-world data. Just as we used for the decision tree algorithm, we will try random forest on the same breast cancer dataset.

So, we will go ahead and start our implementation in the same manner as we have done earlier:

```
filename = 'breast_cancer_data.csv'
dataset = readCsv(filename)

# Convert attributes to numerical type
for i in range(0, len(dataset[0])):
    convert_column_to_float(dataset, i)

# Convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)

#Now remove index column from the dataset
dataset_new = []

for row in dataset:
    dataset_new.append([row[i] for i in
range(1,len(row))])
dataset = dataset_new
```

We will check the performance for 5-fold validation with three different forest sizes (5, 25, 75). In them, each tree will be grown up to a maximum depth of 5 and the sample size will be 10% of the total dataset. The number

of random features in this case will be the square root of feature length. For breast cancer data, the number of features will be three:

```
# Evaluate algorithm
n_folds = 5
max_depth = 5
min_size = 1
sample_size = 0.1
n_features = int(sqrt(len(dataset[0])-1))

# Check the performance for different number of
trees.
for n_trees in [1, 5, 10]:
    scores = evaluate_algorithm(dataset,
    random_forest, n_folds, max_depth, min_size,
    sample_size,
    n_trees, n_features)
    print('Trees: %d' % n_trees)
    print('Scores: %s' % scores)
    print('Mean Accuracy: %.3f%%' %
    (sum(scores)/float(len(scores))))
```

Execution results of the preceding codes are:

```
Trees: 5
Scores: [94.24460431654677, 93.5251798561151,
99.92805755395683, 96.40287769784173,
97.12230215827337]
Mean Accuracy: 94.245%

Trees: 25
Scores: [97.84172661870504, 97.12230215827337,
94.96402877697841, 96.40287769784173,
94.96402877697841]
```

Mean Accuracy: 96.259%

Trees: 75

Scores: [96.40287769784173, 97.84172661870504,
95.68345323741008, 98.56115107913669,
95.68345323741008]

Mean Accuracy: 96.835%

You can see the improvement in the results. When we were using a single decision tree with all of the features in one shot, our testing accuracy was hardly reaching 95%. Whereas, at the time of bagging, we started from 5 trees and got an initial accuracy of 94.24%. When we increased the number of trees to 25, the accuracy crossed the 96% mark, and when we further increased the number of trees to 75, we got almost 97% accuracy out of our model.

The preceding analysis clearly shows the advantage of bagging over a single-model-based prediction. This model is much more generalized and has a much more diverse distribution than a single model, which helps us to achieve lower variance between training and testing sets.

Summary

In this chapter, we learned how we can make decisions on real-world numerical data using decision trees. We started from a simple binary tree and converted it into a decent classifier. Finally, we used bagging for practical data with a very high prediction rate.

Now a question arises here, *can we do better than this?* And the answer is, *this is subjective!* Maybe we can improve the performance by finding more optimized parameters for our tree building process; or, there are possibilities that we can't improve the accuracy further. It all depends on the complexity of data as well as how it has been processed.

Now, there are various possibilities you can try with the random forest models. You can use them as regression models or convert them into multi-class classifiers. There are

many different ways to optimize their hyper-parameters, but greedy search algorithms are quiet popular. Many data preprocessing techniques can be used to make fruitful results out of your classification model, such as dimension reduction of data, smoothing of the dataset, and using interpolation for the missing values. I think you are ready to play with decision trees for many practical applications. Go try them, and remember, *the more you fail, the more you learn.*

Random Subspace and KNN Bagging

In [Chapter 3](#), *Random Forest*, we learned how we can use bagging for the classification of data. We saw an important aspect of bagging in there, that is, attribute selection, where we created multiple samples of data by limiting the number of features in each sample. We have also seen that we can improve the classification accuracy by doing that. This is known as **subspace bagging** or **attribute bagging**.

In this chapter, we will go into the details of this bagging type and use it with another widely used classification algorithm, **K-Nearest Neighbor (KNN)**, which is very popular in the data science world because of its simplicity and ease to implement. Earlier we have seen how to do subspace bagging; here, we will see some technical aspects of it

and learn why this strategy works.

Subspace bagging

So, I think this is enough to talk about. Now, let's jump straight to the point with the use of an example. As I have discussed in the previous chapter, subspace bagging helps create classifiers that are more generalized than classifiers trained without attribute bagging. But why? We can understand this process with a daily-life example. Suppose we have a bunch of kids from kindergarten. Let's take them to the city zoo and show them different animals, mammals, birds, reptiles, and so on, and tell them to remember the animals. After visiting the zoo, we will call every child individually and ask them to describe which animals they have seen in the zoo. Sounds easy? No, folks! It is very difficult for a child to remember all the animals of a zoo; so, we will tell them to tell us about any three animals of the zoo.

Now remember, we have asked them to

choose any three animals, not unique animals. This means two different children can tell us the same animal names. So, we can get the following answers from them:

- Some children choose elephant (because of their size), deer, and peacock
- Some are more interested in elephant, giraffe, and python
- Some are going for tiger, rabbit, and giraffe
- Some are choosing deer, elephant, and python
- And many are going for elephant, tiger, and rabbit

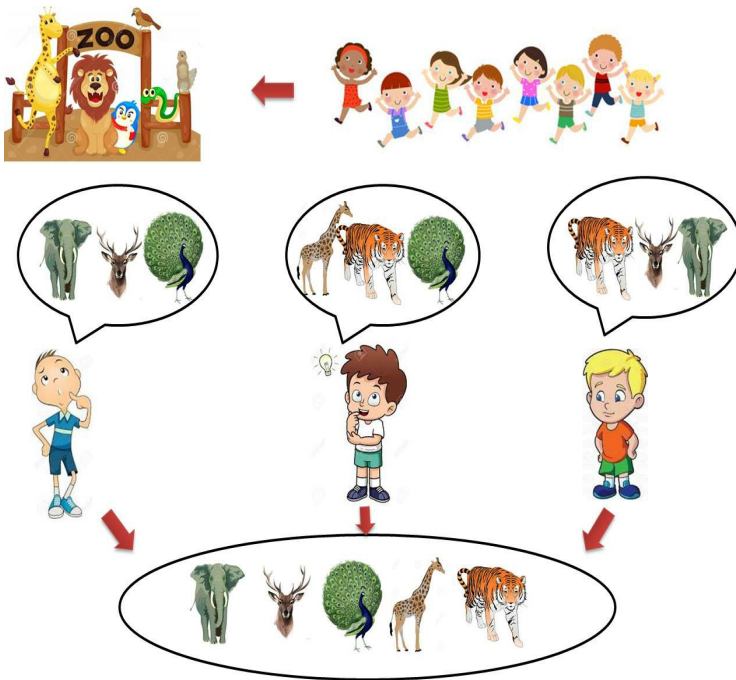


Figure 4.1: Sub-space bagging example

So, what have we got after this exercise?

- Each group of children has remembered something different from the others
- Some groups having some common animals
- If we sum up their review, we will get almost all of the animals of the zoo

So what? Well, you see, we have got our bagged classifier. Let me summarize it. As each child has the ability to learn about four to five animals only, we have asked them to tell us about any three of them, so each child who visits the zoo tells us about the animals. Some are quite easy to remember (say, elephant) but some are not that easy (some birds). What happens if we combine the results of their test?

- We will get all of the animals present in the zoo (which is almost impossible for a single child)
- As some animals are common in their answers, it tells us that all of them are talking about the same zoo
- As we have told each child to remember only three animals, there are fewer chances that they will forget those animals in the future

Can we relate this example with the classification model? Of course!
The same strategy is what we used to train

our random forest classifier and is the reason for getting an accuracy improvement over a single decision tree:

- We have covered (fit) a model by covering all of the features (which is difficult in the case of a single classifier)
- As some features will be shared by the classifier, it shows that all of the data is derived by the same population
- The problem of high variance (overfitting) is less likely in this case, because we get different perspectives (by different classifiers) of our problem

Now that we have understood the concept behind attribute bagging, let's see its impact on practical applications. We will use the helper function from [Chapter 3](#), *Random Forest*, which we have developed to build a random forest algorithm as well as a decision tree.

Case study – subspace bagging

We will use a publicly available dataset of sonar signal returns from different surfaces; the dataset has 208 observations and 60 features for classifying the instances into two groups—mine (M) and rock (R). The variables are in the range of 0 to 1.

Here are some more details about the dataset:

```
Location: https://archive.ics.uci.edu/ml/datasets/Connectionist+Bench+\(Sonar,+Mines+vs.+Rocks\);  
Data set Name: Connectionist Bench (Sonar, Mines vs. Rocks) Data Set  
Number of Instances: 208  
Attributes characteristics: Float  
Number of attributes: 60  
Number of classes: 2
```


More information about the dataset

The `sonar.mines` file contains 111 patterns obtained by bouncing sonar signals off a metal cylinder at various angles and under various conditions. The `sonar.rocks` file contains 97 patterns obtained from rocks under similar conditions. The transmitted sonar signal is a frequency-modulated chirp, rising in frequency. The dataset contains signals obtained from a variety of different aspect angles, spanning 90 degrees for a cylinder and 180 degrees for a rock. Each pattern is a set of 60 numbers in the range of 0.0 to 1.0. Each number represents the energy within a particular frequency band, integrated over a certain period of time. The integration aperture for higher frequencies occurs later in time, since these frequencies are transmitted later during

the chirp. The label associated with each record contains the letter R if the object is a rock and M if it is a mine (metal cylinder). The numbers in the labels are in increasing order of aspect angle, but they do not encode the angle directly.

We will use the same random forest algorithm as we used in [Chapter 3](#), *Random Forest* to compare the effect of subspace bagging with the normal bagging algorithm.

For this, we have to use function `getNode()` with subspace bagging (with feature selection) and without subspace bagging (with all features); we will make the changes in the following code block:

This is the version without subspace selection of function `getNode()`, which we have used earlier to build decision trees:

```
def getNode(dataset, n_features):  
    class_values = []  
    for row in dataset:  
        class_values.append(row[-1])  
  
    #initialize variables to store gini score,
```

```

attribute index and split groups
    winnerAttribute = sys.maxsize
    attributeValue = sys.maxsize
    gScore = sys.maxsize
    leftGroup = None

    for index in range(len(dataset[0])-1):
        for row in dataset:
            groups = createSplit(index,
row[index], dataset)
            gini = gini_index(groups,
class_values)
            if gini < gScore:
                winnerAttribute,
attributeValue, gScore, leftGroup = index,
row[index], gini, groups

        #Once done create a dictionary for node
        node =
{'attribute':winnerAttribute, 'value':attributeVa

    return node

#Let's start the comparison;

# Test the random forest algorithm
seed(1)

# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)

# convert string attributes to integers
for i in range(0, len(dataset[0])-1):
    str_column_to_float(dataset, i)

# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
dataset_new = []
for row in dataset:
    dataset_new.append([row[i] for i in
range(1,len(row))])
dataset = dataset_new

```

```

# evaluate algorithm
n_folds = 5
max_depth = 5
min_size = 1
sample_size = 0.1

for n_trees in [5, 25, 50]:
    scores = evaluate_algorithm(dataset,
                                random_forest, n_folds, max_depth, min_size,
                                sample_size,
                                n_trees, n_features)

    print('Trees: %d' % n_trees)
    print('Scores: %s' % scores)
    print('Mean Accuracy: %.3f%%' %
          (sum(scores)/float(len(scores))))

```

We will use 5-fold validation using decision trees with depth 5. The subsample size will be 10% of the number of instances. We will create a forest of 5, 25, and 50 trees. All helper functions will remain the same except `getNode` without subspace selection. After executing the code, we will get:

```

Trees: 5
Scores: [75.60975609756098, 51.21951219512195,
65.85365853658537, 68.29268292682927,
68.29268292682927]
Mean Accuracy: 65.854%

Trees: 25
Scores: [70.73170731707317, 65.85365853658537,
78.04878048780488, 73.17073170731707,
70.73170731707317]
Mean Accuracy: 71.707%

```

```
Trees: 50
Scores: [58.536585365853654, 65.85365853658537,
80.48780487804879, 75.60975609756098,
70.73170731707317]
Mean Accuracy: 70.244%
```

The `getNode()` function and use it in a similar manner as we used it in [Chapter 3, Random Forest](#) to create a random forest:

```
def getNode(dataset, n_features):
    class_values = []
    for row in dataset:
        class_values.append(row[-1])

    #initialize variables to store gini score,
    attribute index and split groups
    winnerAttribute = sys.maxsize
    attributeValue = sys.maxsize
    gScore = sys.maxsize
    leftGroup = None

    #Run loop to access each attribute and
    attribute values
    features = list()
    while len(features) < n_features:
        index = randrange(len(dataset[0])-1)
        if index not in features:
            features.append(index)

    for index in features:
        for row in dataset:
            groups = createSplit(index,
row[index], dataset)
            gini = gini_index(groups,
class_values)
            if gini < gScore:
                winnerAttribute,
attributeValue, gScore, leftGroup = index,
```

```

row[index], gini, groups

    #Once done create a dictionary for node
    node=
    {'attribute':winnerAttribute,'value':attributeVa

    return node

```

Now, we will use subspace bagging for the same dataset:

```

# Test the random forest algorithm
seed(1)

# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)

# convert string attributes to integers
for i in range(0, len(dataset[0])-1):
    str_column_to_float(dataset, i)

# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
dataset_new = []
for row in dataset:
    dataset_new.append([row[i] for i in
range(1,len(row))])
dataset = dataset_new

# evaluate algorithm
n_folds = 5
max_depth = 5
min_size = 1
sample_size = 0.1
n_features = 10

for n_trees in [5, 25,50]:
    scores = evaluate_algorithm(dataset,
random_forest, n_folds, max_depth, min_size,

```

```

sample_size,
n_trees, n_features)
    print('Trees: %d' % n_trees)
    print('Scores: %s' % scores)
    print('Mean Accuracy: %.3f%%' %
          (sum(scores)/float(len(scores))))

```

We will use 5-fold validation using decision trees with depth 5; the subsample size will be 10% of the number of instances. As we are using attribute bagging, we will limit the number of features to 10. These features will be selected randomly from the created sample. We will create a forest of 5, 25, and 50 trees; all helper functions will remain the same except `getNode()` without subspace selection. After executing the code, we will get:

```

Trees: 5
Scores: [65.85365853658537, 78.04878048780488,
60.97560975609756, 65.85365853658537,
68.29268292682927]
Mean Accuracy: 67.805%

Trees: 25
Scores: [70.73170731707317, 68.29268292682927,
75.60975609756098, 60.97560975609756,
75.60975609756098]
Mean Accuracy: 70.244%

Trees: 50
Scores: [75.60975609756098, 82.92682926829268,
75.60975609756098, 82.92682926829268,

```

| 68.29268292682927]
| Mean Accuracy: 77.073%

Can you see that! What an improvement we have got in the classification accuracy! Let's feed the data into a table:

SN	Trees	Mean accuracy without bagging
1	5	65.85%
2	25	71.707%
3	50	70.244%

Table 4.1: Accuracy difference – bagged subspace versus all features

The preceding table shows a clear difference in accuracy by two different methods; if we use more number of trees with subspace

bagging, we get a significant improvement in the classification accuracy.

Random subspace can be used in any bagging or boosting algorithm. In the following sections, we will see how to use it with the KNN algorithm to get a generalized predictor.

KNN classification

We have used decision trees for classification purposes earlier and have seen their practical applications, which were quite impressive. Now, it's time to move ahead and add one more simple, yet very powerful, tool to our tool set for classification of data. KNN algorithm, also known as K-Nearest Neighbors, is widely used in the machine learning community. It is a non-parametric algorithm; that means it doesn't learn any underlying distributions of the dataset. It is originally derived from the k-means clustering algorithm, which is another very popular algorithm for unsupervised classification of data. In k-means clustering, similarity is the criteria to create clusters out of data. The procedure of the k-means algorithm is quite simple:

1. Choose random values (equal to the desired number of clusters) out of our dataset; for example, x and y are two

values.

2. Find the similarity between the chosen values (x and y) and each instance of the dataset using a distance metric. You will get two distance vectors, one for x and another for y .
3. Put all the instances that have less distance from x in the x -cluster, and put others in the y -cluster.
4. Now, find the mean values of both the clusters and replace them with x and y .
5. Repeat steps 3 and 4 until the distance metric reaches a very small value.

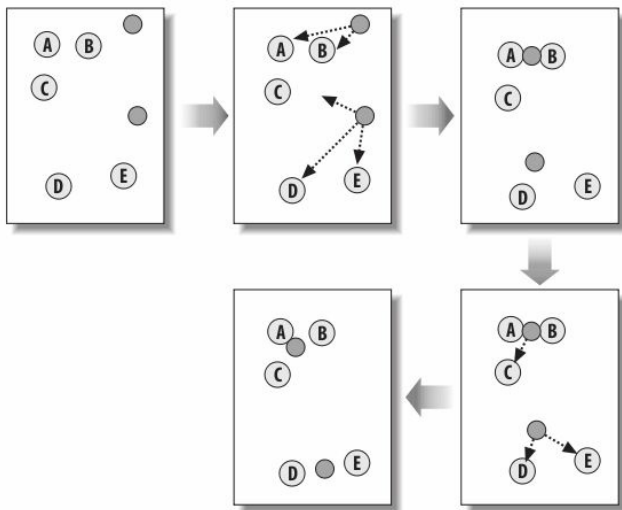


Figure 4.2 Process of K-means clustering

The preceding figure shows the process of clustering. So, how is it really extended to the KNN algorithm? To know this, first we will take a walk through the KNN algorithm. As we have seen in the first step of k-means clustering, we choose some random values to make clusters out of data using the distance metric. Here, we are going to do the same, except that those values (cluster centers) will not be chosen randomly. The following will be the procedure:

1. As we are currently working with a supervised learning algorithm, we will provide data with known labels to the algorithm
2. Try to find the similarity between the unseen instance and known data using a distance metric, and get the most similar k number of instance's label; these instances are known as **nearest neighbors**
3. After getting the nearest neighbors, we

can get the label with the highest number, which will be the winner

The following figure shows the classification using KNN algorithm:

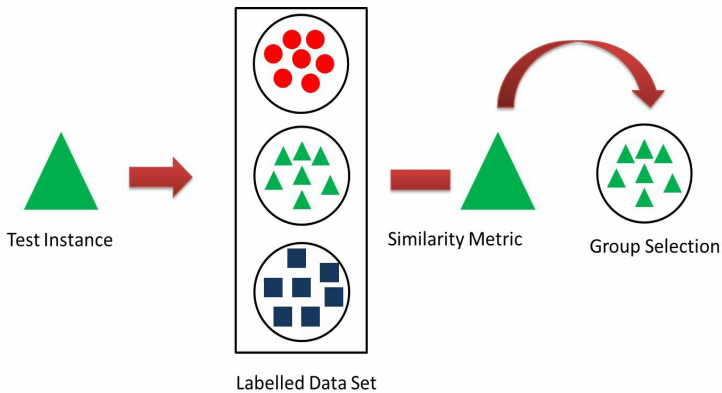


Figure 4.3: Classification using KNN algorithm.

KNN is known as a competitive learning algorithm, as it creates competition between instances to reach the final prediction. When we perform a similarity check between data instances, it causes each data instance to compete to win, or be most similar to a given unseen data instance, and contribute to a prediction.

KNN is also known as the **lazy learning** algorithm. Lazy learning refers to the fact that the algorithm does not build a model until the time a prediction is required. It is lazy because it only does work at the last second. This has the benefit of including only data that is relevant to the unseen data, called a **localized model**. A disadvantage is that it can be computationally expensive to repeat the same or similar searches over larger training datasets.

So, as we always do, we will implement this algorithm in Python using the preceding steps, and then we will use it for a practical application to build our concepts stronger. So, let's start with a simple example for a toy dataset:

SN	F1	F2	F3	F4	Class
1	5.1	3.5	1.4	0.2	1

2	4.9	3.0	1.4	0.2	1
3	4.7	3.2	1.3	0.2	1
4	4.6	3.1	1.5	0.2	1
5	5.0	3.6	1.4	0.2	1
6	7.0	3.2	4.7	1.4	2
7	6.4	6.2	4.5	1.5	2
8	6.9	3.1	4.9	1.5	2
9	5.5	2.3	4.0	1.3	2
10	6.5	2.8	4.6	1.5	2

11	6.3	3.3	6.0	2.5	3
12	5.8	2.7	5.1	1.9	3
13	7.1	3.0	5.9	2.1	3
14	6.3	2.9	5.6	1.8	3
15	6.5	3.0	5.8	2.2	3

Table 4.2: Toy dataset

The preceding dataset has been cropped from the original IRIS classification dataset; there are 150 instances available in the original dataset with four attributes and three classes. We will create a KNN classifier using the preceding dataset to classify unknown instances.

As we know, the heart of the algorithm is the

similarity metric, which is used to get nearest neighbors. There are many distance metrics available in the literature such as Euclidean distance (mean-based), city block distance (median-based), Mahalanobis distance (covariance-based), and so on. The Euclidean distance metric is the simplest and a very useful distant metric, and we will use it to find similarity between instances. The formula of Euclidean distance is:

$$d(p, q) = d(q, p) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \dots + (q_n - p_n)^2}$$
$$d(p, q) = \sqrt{\sum_{i=1}^n (q_i - p_i)^2}$$

Where p and q are input instances and p_1 and q_1 the elements of instances (in our case, feature values). To calculate the Euclidean distance between two instances, we will perform element-wise squared difference (that is, subtraction between feature values) to get a non-zero value, and then we will just sum up all of the differences and get the

square root of the summed value. This is also known as sum of squared error.

We will implement a Python function to calculate the Euclidean distance metric; the code listing will be as follows:

```
def DistanceMetric(instance1, instance2,
isClass=None):

    #If Class variable is in the instance
    if isClass:
        length = len(instance1)-1
    else:
        length = len(instance1)

    #Initialize variable to store distance
    distance = 0

    #Lets run a loop to calculate element wise
differences
    for x in range(length):

        #Euclidean distance
        distance += pow((instance1[x] -
instance2[x]), 2)

    return math.sqrt(distance)
```

The preceding function will calculate the Euclidean distance between two vectors.

Let's test this function for a simple array of three values:

```
| data1 = [1, 3, 2]
| data2 = [3, 3, 2]
| distance = DistanceMetric(data1, data2)
| print('Euclidean Distance is : %.2f'%
| (distance))
```

After execution:

```
| Euclidean Distance is: 2.00
```

Good, it's working! So, the first and important part of our code is completed. The next thing is to get neighbors using this distance, so our function `getNeighbors` goes like this:

```
| import operator
| def getNeighbors(trainingSet, testInstance, k):
|
|     #Create a list variable to store distances
|     between test and
|     #training instance.
|     distances = []
|
|     #Get distance between each instance in the
|     training set and the
|     #test instance.
|     for x in range(len(trainingSet)):
|
|         #As we will have class variable in the
|         training set
|         isClass will be true
|         dist = DistanceMetric(testInstance,
| trainingSet[x],
|                             isClass=True)
|
|         #Append the distance of each instance
```

```

to the distance list
    distances.append((trainingSet[x],
dist))

    #Sort the distances in ascending order
    distances.sort(key=operator.itemgetter(1))

    #Create a list to store the neighbors
    neighbors = []

    #Run a loop to get k neighbors from the sorted distances.
    for x in range(k):
        neighbors.append(distances[x][0])
    return neighbors

```

Let's test our `getNeighbors` function:

```

#Test neighbors
trainSet = [[2, 2, 2, 'a'], [3, 3, 3, 'a'], [4,
4, 4, 'b'], [6, 6, 6, 'b']]
testInstance = [5, 5, 5]
k = 2
neighbors = getNeighbors(trainSet,
testInstance, k)
print(neighbors)

```

After execution:

```

| [[4, 4, 4, 'b'], [6, 6, 6, 'b']]

```

Yup, it's working! Now that we have located the most similar neighbors for a test instance, the next task is to get a predicted class based on these neighbors.

We can do this by allowing each neighbor to vote for their class attribute and taking the majority vote as the prediction.

The following function does work to get the majority voted response from a number of neighbors. It assumes that the class is the last attribute for each neighbor:

```
import operator
def getPrediction(neighbors):

    #Create a dictionary variable to store
    votes from the neighbors
    #We will use class attribute as the
    dictionary keys and their
    #occurrence as key value.
    classVotes = {}

    #Go to each neighbor and take the vote for
    the class
    for x in range(len(neighbors)):

        #Get the class value of the neighbor
        response = neighbors[x][-1]

        #Create class key if its not there;
        #If class key is in the dictionary
        increase it by one.
        if response in classVotes:
            classVotes[response] += 1
        else:
            classVotes[response] = 1
    #Sort the dictionary keys on the basis of
    key values in descending order
    sortedVotes =
    sorted(classVotes.iteritems(),
```

```

key=operator.itemgetter(1),
                    reverse=True)

    #Return the key name (class) with the
    highest value
    return sortedVotes[0][0]

```

Now we have all of the functions needed to test the KNN performance on our toy dataset. So, let's start with this:

```

#Lets create our toy dataset of iris flower
classification.
dataset = [[5.1,      3.5,      1.4,      0.2,      1],
           [4.9,      3.0,      1.4,      0.2,      1],
           [4.7,      3.2,      1.3,      0.2,      1],
           [4.6,      3.1,      1.5,      0.2,      1],
           [5.0,      3.6,      1.4,      0.2,      1],
           [7.0,      3.2,      4.7,      1.4,      2],
           [6.4,      6.2,      4.5,      1.5,      2],
           [6.9,      3.1,      4.9,      1.5,      2],
           [5.5,      2.3,      4.0,      1.3,      2],
           [6.5,      2.8,      4.6,      1.5,      2],
           [6.3,      3.3,      6.0,      2.5,      3],
           [5.8,      2.7,      5.1,      1.9,      3],
           [7.1,      3.0,      5.9,      2.1,      3],
           [6.3,      2.9,      5.6,      1.8,      3],
           [6.5,      3.0,      5.8,      2.2,      3]]

#Lets put our test instance.
testInstance=[4.8,3.1,3.0,1.3,1]

#Now lets find out 3 neighbors for our test
instance using getNeighbor
k = 3
neighbors = getNeighbors(dataset, testInstance,
k)

#Print neighbors
print(neighbors)

```

```
| [[5.5, 2.3, 4.0, 1.3, 2], [4.6, 3.1, 1.5, 0.2,  
| 1], [4.9, 3.0, 1.4, 0.2, 1]]
```

If you look closer, you will notice that the class attribute of the first neighbor is 2 while the other two neighbors are of class 1; this is the place where we will require the `getPrediction` function, where we will have voting for each class:

```
| #Get the class prediction out of neighbors  
| prediction = getPrediction(neighbors)  
| #Print prediction  
| print("Predicted class for the test instance  
| is: %d"%prediction)
```

```
| Predicted class for the test instance is: 1
```

It's working perfectly!

So, as always, it's time to test our algorithm on a practical dataset. As we always try to cover a practical application from a different field, this time, we will apply our implemented algorithm to the cyberworld field. Yes we will try to apply our algorithm for a cybersecurity element—spam classification!!

KNN for spam filtering

It is impossible to tell exactly who was the first one to come upon a simple idea that if you send out an advertisement to millions of people, then at least one person will react to it no matter what the proposal is. E-mail provides the perfect way to send these millions of advertisements at no cost for the sender, and this unfortunate fact is nowadays extensively exploited by several organizations. As a result, the mailboxes of millions of people get cluttered with all this so-called unsolicited bulk e-mail, also known as **spam** or **junk mail**. Being incredibly cheap to send, spam causes a lot of trouble to the internet community. Large amounts of spam-traffic between servers cause delays in deliveries of legitimate e-mail. People with dial-up internet access have to spend

bandwidth downloading junk mail. Sorting out unwanted messages takes time and introduces a risk of deleting important mails by mistake. Finally, there is quite an amount of pornographic spam that children should not be exposed to.

Dataset

We will use a publicly available dataset for our application; this dataset is available at: <https://archive.ics.uci.edu/ml/datasets/spambase>. You can easily download it and store it in the form of a `.csv` file, as we already have functions to read a `.csv` file and load the dataset into a NumPy array (refer to [Chapter 3](#), *Random Forest*).

The following is the dataset information as found on the given web address.

Dataset information

The spam concept is diverse: advertisements for products/web sites, make money fast schemes, chain letters, pornography, and so on.

Our collection of spam e-mails came from our postmaster and individuals who had filed spam. Our collection of non-spam e-mails came from filed work and personal e-mails, and hence the word *george* and the area code 650 are indicators of non-spam. These are useful when constructing a personalized spam filter. One would either have to blind such non-spam indicators or get a very wide collection of non-spam to generate a general-purpose spam filter.

Attribute information

You can get all of the available information on the preceding link. I will be sharing only basic information about the dataset. There are 57 attributes in the dataset; each attribute signifies frequency of that word occurring in the spam or non-spam emails. For example, in a spam mail, there are certain words that have more occurrences such as money, rich, hot, nearby, business, and so on. The following is a snapshot of the attribute names:

word_freq_order:	continuous.
word_freq_mail:	continuous.
word_freq_receive:	continuous.
word_freq_will:	continuous.
word_freq_people:	continuous.
word_freq_report:	continuous.
word_freq_addresses:	continuous.
word_freq_free:	continuous.
word_freq_business:	continuous.
word_freq_email:	continuous.

On the left, attribute names are mentioned, and on the right is the type (continuous, discrete, and so on).

There are 2,906 instances in the dataset where the last instance has an incomplete set of attributes, so we will keep an eye on it when we perform our calculation.

We have written most of the functions we require in this example; if we add anything new, we will discuss it. So, let's start with loading our dataset:

```
#Read CSV file  
dataName = 'spamData.csv'  
  
#Use function load_csv from chapter 3  
dataset = load_csv(dataName)  
  
#Create an empty list to store the dataset  
dataset_new = []  
  
#We will remove incomplete instance from the dataset  
for i in range(len(dataset)-1):  
    dataset_new.append(dataset[i])  
dataset = dataset_new  
  
#Use function str_column_to_float from chapter 3 to convert string values to float  
from utilityFunctions import
```

```

str_column_to_float
for i in range(0, len(dataset[0])-1):
    str_column_to_float(dataset,
i,len(dataset))

#Split train and test dataset using function
getTrainTestData
#We will use 80% of the dataset as training set
and rest for testing
train,test = getTrainTestData(dataset,0.8)

#Create empty list to store predictions and
actual output
testPredictions=[]
testActual=[]

#We will choose 3 nearest neighbor
k = 3

#Get prediction for each test instance and
store them into the list
for i in range(0,len(test)):
    test_instance = test[i]
    neighbors = getNeighbors(train,
test_instance, k)
    pred = getPrediction(neighbors)
    testActual.append(test_instance[-1])
    testPredictions.append(pred)
    print ("Actual: %s Predicted: %s"%
(test_instance[-1],pred))

#Use accurcay_metric function to evaluate our
results
accuracy =
accuracy_metric(testActual,testPredictions)

#Print accuracy
print("Accuracy of the classification:
%0.2f"%accuracy)

```

After execution of the preceding code, we

will get:

| Accuracy of the classification: 79.21

So, we are getting almost 80% classification accuracy! Don't be sad; it's not that bad! In spam classification, we can't go with a zero tolerance policy, because if we do that, it may put many useful e-mails into spam classes, and we don't want that. But yes, we can improve it. How? Simple. If you are not getting enough accuracy with this classifier, change it! Yes, because non-parametric classifiers are pretty straightforward. They do not make any assumptions on the training data. They do not learn any distribution underlying the dataset, and that's why they are easy to understand and easy to implement. If we want to get a further enhancement in the performance of the classifier, we have to go for a complex one. SVMs perform quite a lot better with spam classification. We will cover SVMs in the later chapters; we will discuss this there.

But wait! Why can't we apply subspace

bagging to our classifier and create an ensemble of multiple KNNs? Yes, we are going to do the same for the same application with the same dataset, but now we are going to use the power of ensembles in our algorithm. And let's see whether we can improve the performance of the current KNN.

KNN bagging with random subspaces

As you know, bagging can be implemented in the following way:

1. Create samples with replacement from the training set
2. Apply random subspace (limiting the number of features to train)
3. For each sample, create one classifier
4. At the time of testing, use voting from each classifier to get a prediction

Let's start with modifying `DistanceMetric()` for a subspace bagging implementation:

```
from random import randrange
def DistanceMetricBagged(instance1,
instance2,n_features, isClass=None):

    #If Class variable is in the instance
    if isClass:
        length = len(instance1)-1
    else:
        length = len(instance1)
```

```

#Initialize variable to store distance
distance = 0
features = list()

#Select random features to apply subspace
bagging
while len(features) < n_features:
    index = randrange(len(dataset[0])-1)
    if index not in features:
        features.append(index)

#Lets run a loop to calculate element wise
differences for the selected
features only.
    for x in features:
        #Euclidean distance
        distance += pow((instance1[x] -
instance2[x]), 2)

    return math.sqrt(distance)

```

Now we have modified the `distance` function; we will make changes to `getNeighbors()` too so that we can call `DistanceMetricBagged()` from there:

```

def getNeighborsBagged(trainingSet,
testInstance, k,n_features):

    #Create a list variable to store distances
between test and training instance.
    distances = []

    #Get distance between each instance in the
training set and the test
instance.
    for x in range(len(trainingSet)):
        #As we will have class variable in the

```

```

training set isClass will
    be true
    dist =
DistanceMetricBagged(testInstance,
trainingSet[x],n_features,isClass=True)

    #Append the distance of each instance
to the distance list
    distances.append((trainingSet[x],
dist))

    #Sort the distances in ascending order
    distances.sort(key=operator.itemgetter(1))

    #Create a list to store the neighbors
    neighbors = []

    #Run a loop to get k neighbors from the
sorted distances.
    for x in range(k):
        neighbors.append(distances[x][0])
    return neighbors

```

So we are ready to use bagging; let's start with it. The following code block will first create random samples of a given size from the training set and then create a single classifier for each sample. At testing time, we will get a vote from each classifier to classify an instance:

```

import math
import operator
from Chapter_03.DecisionTree_CART_RF import
load_csv, getTrainTestData, accuracy_metric,
str_column_to_float

```

```

import numpy as np
#Read CSV file
dataName = 'spamData.csv'

#Use function load_csv from chapter 3
dataset = load_csv(dataName)

#Create an empty list to store the dataset
dataset_new = []

#We will remove incomplete instance from the
dataset
for i in range(len(dataset)-1):
    dataset_new.append(dataset[i])
dataset = dataset_new

#Use function str_column_to_float from chapter
3 to convert string values to float
for i in range(0, len(dataset[0])-1):
    str_column_to_float(dataset, i)

#Split train and test dataset using function
getTrainTestData
#We will use 80% of the dataset as training set
and rest for testing
train,test = getTrainTestData(dataset,0.8)

#Create empty list to store predictions and
actual output
testPredictions=[]
testActual=[]

#Select number of neighbors for each classifier
k = 7

#Select sample size
sample_size = 500

#Select number of random features
n_features = 20

#Calculate number of classifier on the basis of

```

```

number of samples.
n_classifier = np.uint8(len(train)/sample_size)
#Get prediction for each test instance and
store them into the list
for i in range(0,len(test)):
    predictions = []

    #Run loop for each sample
    for c1 in range(1,n_classifier):

        #Randomly shuffle training set and
create sample out of it
        np.random.shuffle(train)
        sample = [train[row] for row in
range(sample_size)]

        #Pick test instance
        test_instance = test[i]

        #Get neighbors and prediction on the
basis of neighbor
        neighbors = getNeighborsBagged(sample,
test_instance,
                                k,n_features)
        pred = getPrediction(neighbors)

        #Append prediction against each sample
with random features
        predictions.append(pred)

        #Get final prediction using majority voting
from each classifier
        fin_pred = max(set(predictions),
key=predictions.count)
        testActual.append(test_instance[-1])
        testPredictions.append(fin_pred)
        print ("Actual: %s   Predicted: %s"%
(test_instance[-1],pred))

```

After executing the preceding code, we will get:

```
Actual: 1    Predicted: 1
Actual: 1    Predicted: 1
Actual: 0    Predicted: 1
.
.
.
Actual: 1    Predicted: 0
Actual: 1    Predicted: 1
Actual: 1    Predicted: 1
Actual: 0    Predicted: 0

Accuracy of the classification: 84.71
```

Can you see that!!! We have got an accuracy of 85% over 79%. Do you know what a significant improvement we have done?

We have 20% of the dataset as testing set, which is equal to 582 instances. Earlier, we were getting an accuracy of 79%, which is about 466 correct predictions out of 582. Now, after using ensembles, we are getting an accuracy of almost 85%, which is about 495 correct predictions. This is an improvement of 29 more correct predictions than simple KNN.

Summary

So, you saw how bagging can improve the classification accuracy of a simple classifier. We learned about random subspaces in detail. We saw how this method can improve the results when we use ensemble methods. We also worked with the KNN algorithm and its practical applications, and improved our classification accuracy using subspace bagging with KNN for spam classification.

Now you guys have a good understanding of bagging ensembles as we have used it with three to four different practical applications with code implementation. This book will later cover another important concept of ensemble methods, known as boosting. We will start boosting from the next chapter; until then, you can try our implemented bagging algorithm for more practical datasets and improve your understanding of the data science field.

AdaBoost Classifier

Hey folks, we have seen in previous chapters how a classifier works, how to practically implement a classifier in Python, and how to use many such classifiers to get higher prediction accuracy. But did you notice one important thing there? Our classifiers were not communicating! By communication, I mean that our classifiers were working independent of each other to get the prediction. So, I am raising another question for you. What if our classifiers start sharing information together? Can it help us get more optimized results in the end? In this section of the book, we are going to look at this aspect, where we will try to combine the different classifiers with information sharing.

Boosting

Do you remember those kids from kindergarten (from [Chapter 4](#), *Random Subspace and KNN Bagging*) who helped us create a random subsample algorithm? We are going to take their help again, but this time in a different way.

Do you remember the last time we went to the city zoo with the kids and told them to remember the animals? Later we performed a memory test on them to check how many animals they learned about. Here is the brief summary:

We have a bunch of kids from kindergarten. We take them to the city zoo and show them different animals, mammals, birds, reptiles and so on. We tell them to remember the animals. After visiting the zoo, we call each child individually. We show them different pictures of animals and ask them to recognize

whether it is a picture of a mammal or a reptile. Of course, we will tell them the basic difference between mammals and reptiles.

Do you notice that we have advanced the level of complexity this time? Earlier, we just asked the names of the animals they had seen in the zoo. Now we are asking them to divide their pictures into mammals and reptiles. We know we cannot expect 100% correct classifications from the kids; we can expect an average of 50% correct classifications. What do you think? Is it sufficient or can they improve? I will tell you the answer, but first let's think about the situation:

- As we know, each child is capable of identifying at least 50% of the animals as mammals
- Can we use one child's wrong identification to improve the identification from the next child?
- Can we somehow force their minds to concentrate more on those animals that have been incorrectly identified?

Now, can you guess my solution for getting the best results in this scenario? I will suggest that you use chocolates for this. Chocolates!! Really! Repeatedly use chocolates and you will get the best identification from them. How? Let's see.

Let's pick a child and show him/her some pictures of mammals and reptiles for identification. We will show four mammals (elephant, deer, giraffe, and tiger) and four reptiles/amphibians (crocodile, frog, python, and turtle);

So we keep eight images in front of each child, and yes we keep one chocolate with each picture! Why? Here is the trick my friends: whenever a child predict an animal's category correctly, he/she can take the chocolate placed before the picture, but if he/she predicts wrong, the chocolate will stay in its place.



Figure 5.1: Animal classification

Now call the first child and ask him his predictions; suppose he picks the mammals and reptiles as follows:

Animal	Class
Elephant	Mammal
Deer	Reptile
Turtle	Reptile
Tiger	Mammal
Python	Reptile
Frog	Mammal

Giraffe	Mammal
Crocodile	Mammal

Table 5.1

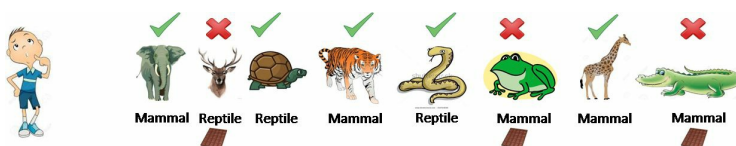


Figure 5.2: First child's prediction (weak learner)

As you can see in the previous figure, he predicted three animals wrongly so he has to leave three chocolates in front of the pictures.

It's time to call the next kid. When he reaches there, he sees chocolates first (HAHA! Of course, isn't it obvious?). So he knows that someone put these three animals in the wrong category. He puts in more effort for those three animals and puts them in the correct class, but he also makes wrong predictions

for some other animals:

Animal	Class
Elephant	Mammal
Deer	Mammal
Turtle	Reptile
Tiger	Mammal
Python	Mammal
Frog	Reptile
Giraffe	Mammal

Crocodile	Mammal
-----------	---------------

Table 5.2: Error made by first kid

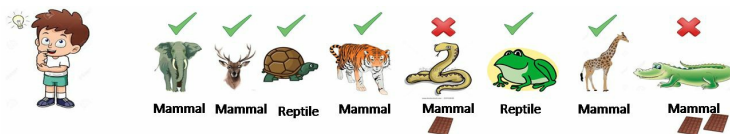


Figure 5.3: Second child's prediction (second weak learner)

He placed crocodile and python in the wrong category. This time, we want the next child to make more effort to get the crocodile into the correct category, so we will put one more chocolate in front of the crocodile picture. There is already one chocolate for the python.

Let's call the third child. He will look at the picture of the crocodile with two chocolates and the one chocolate for the python. He too will know that these two pictures were classified wrongly, so he will make the correct predictions for those two. As we

know, he will also make some wrong predictions, as shown in the following table:

Animal	Class
Elephant	Mammal
Deer	Mammal
Turtle	Mammal
Tiger	Mammal
Python	Reptile
Frog	Mammal
Giraffe	Mammal

Crocodile	Reptile

Table 5.3: Error made by the third child

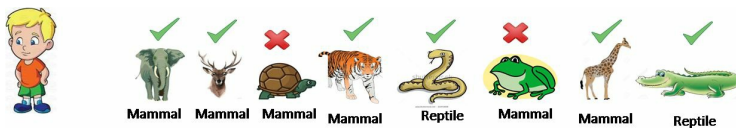


Figure 5.4: Prediction by the third child (third weak learner)

What do you think we have achieved in the whole process? Now let's just combine the predictions of all three of them and we will get all eight predictions correct! Yes, we will get this because, as you have seen in the previous analyses, each child cares about the animal that has chocolates in front of it. This forces them to make a correct prediction as we are working with a two-class problem.

What the kids have done in this process is known as **boosting**!! Yes, boosting is the

process where you put multiple weak learners in a series. One learner's output will be the input of the next learner, so the next learner is boosted by the current one's error. The current classifier will take into account the error made by the previous classifier and this error will force it to learn from the previous one's mistakes. We can get a better understanding from the following figure. This is the same figure from [Chapter 1, Introduction to Ensemble Machine Learning](#), where we discussed boosting ensemble methods:

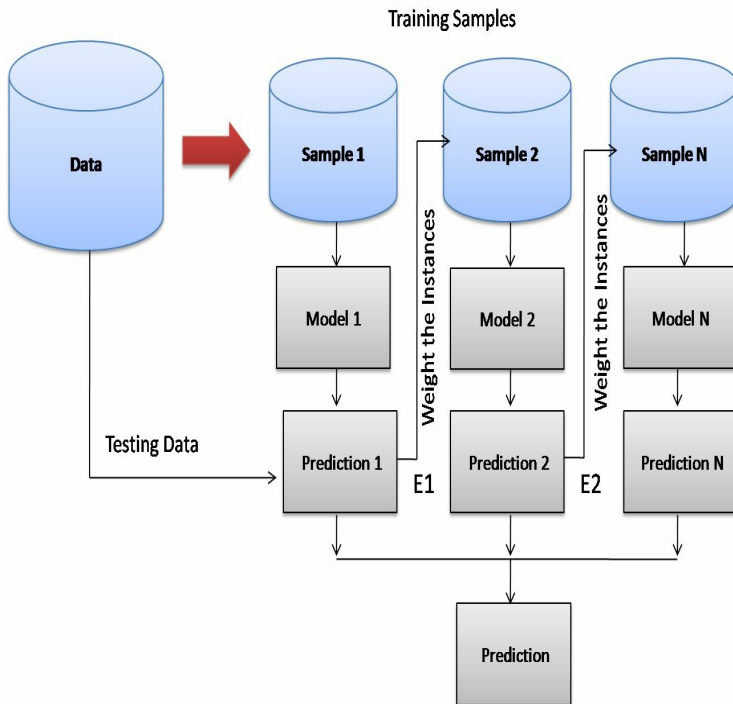


Figure 5.5: Boosting ensemble algorithm

So you can see and believe me! Our kindergarden kids have done the same thing. If you analyze the mammal versus reptile problem you will come to know that we were working with a binary class problem, where we have used multiple weak learners (our kindergarden kids) one after another. When

the classifier-1 (child -1) makes a wrong prediction, we have to leave the chocolate their this chocolate is nothing but the weights which forces next classifier to make more effort on wrong predicted instances. The process of boosting can be understood via the following example more clearly.

We will understand the whole procedure with the following problem. Suppose we have some positive and negative instances in a box, and these instances cannot be separated with a simple classifier, as shown in the following figure:

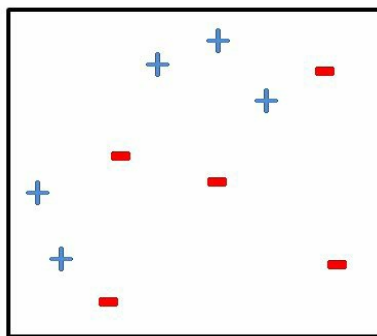


Figure 5.6: Classification of a nonlinear problem

Box 1: You can see that we have assigned equal weights to each data point and applied a decision stump to classify them as + (plus) or – (minus). The decision stump (D1) has generated a vertical line on the left to classify the data points. We see that this vertical line has incorrectly predicted three + (plus) as – (minus). In such a case, we'll assign higher weights to these three + (pluses) and apply another decision stump:

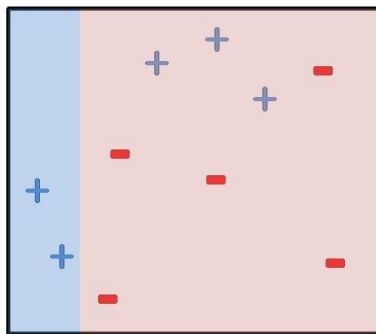


Figure 5.7: First decision boundary (first weak learner)

Box 2: Here, you can see that the size of three incorrectly predicted + (pluses) is bigger compared to rest of the data points. In this case, the second decision stump (D2) will try

to predict them correctly. Now, a vertical line (D2) at the right-hand side of this box has classified three misclassified + (pluses) correctly. But again, it has caused misclassification errors, this time with three - (minuses). So again we will assign a higher weight to the three - (minuses) and apply another decision stump:

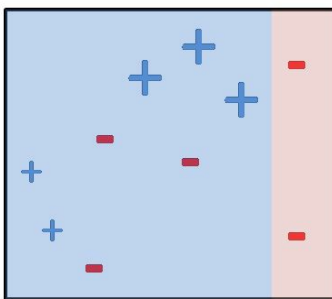


Figure 5.8: Second decision boundary (second weak learner)

Box 3: Here, the three - (minuses) are given higher weights. A decision stump (D3) is applied to predict these misclassified observations correctly. This time, a horizontal line is generated to classify + (plus) and - (minus) based on higher weights of

misclassified observation.

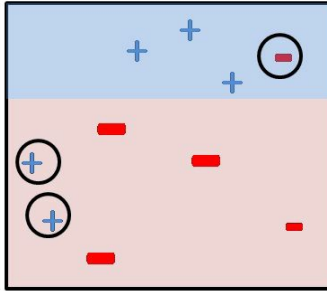


Figure 5.9: Third decision boundary (Third weak learner)

Box 4: Here, we have combined D1, D2, and D3 to form a strong prediction having a complex rule compared to an individual weak learner. You can see that this algorithm has classified these observations quite well compared to any individual weak learner.

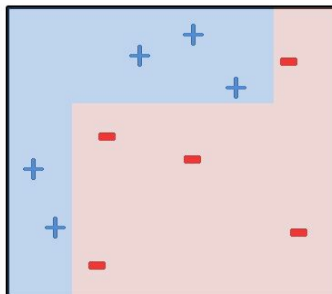


Figure 5.10: Combined decision boundary (ensemble of weak learners)

Adaptive Boosting (AdaBoost) works on a similar method as discussed before. It fits a sequence of weak learners on different pieces of weighted training data. It starts by predicting the original dataset and gives equal weights to each observation. If the prediction is incorrect using the first learner, then it gives a higher weight to that observation. Being an iterative process, it continues to add learner(s) until a limit is reached in the number of models or accuracy.

AdaBoost in a nutshell

Through the previous example, you have got a basic introduction to the AdaBoost algorithm and it's time to see its practicality. Let's get much deeper into its math and see why this algorithm works well—let's get into the algorithm itself.

AdaBoost is best used to boost the performance of decision trees on binary classification problems. AdaBoost was originally called **AdaBoost.M1** by the authors of the technique, **Freund** and **Schapire**. More recently, it may be referred to as discrete AdaBoost because it is used for classification rather than regression. AdaBoost can be used to boost the performance of any machine learning algorithm. It is best used with weak learners. These are models that achieve accuracy just

above a random chance on a classification problem.

The most suited and therefore most common algorithm used with AdaBoost is decision trees with one level. Because these trees are so short and only contain one decision for classification, they are often called decision stumps.

So, the following will be the flowchart of our algorithm development:

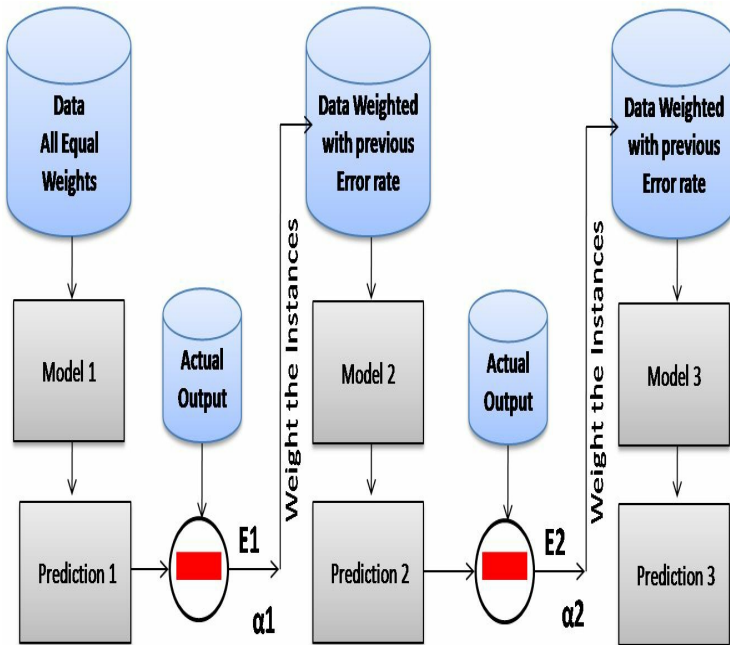


Figure 5.11: AdaBoost algorithm learning

The previous figure shows the following procedure to get to the final prediction; we will understand the procedure using our previous example of kids:

1. Create a model out of data and get the initial prediction (our little kindergarten kid)
2. Get the error between the actual and

predicted output

3. On the basis of the error, decide the rate of the change of weights (similar to deciding the number of chocolates for the wrong prediction)
4. Feed the weighted instances to the next classifier for classification (similar to showing images with chocolates to the next kid)
5. Repeat the procedure a number of times, and create an ensemble of all the classifiers

You see, it's a simple procedure to get a powerful AdaBoost classifier! So, the next question: how do we implement this algorithm in Python? We will see this in a moment, but before going for the implementation, let's see a bit of the mathematics behind the algorithm.

To learn about AdaBoost, go through a tutorial written by one of the original authors of the algorithm, Robert Schapire. The tutorial is available at <http://rob.schapire.net/papers/e>

[xplaining-adaboost.pdf](#).

Now we will break the problem into small pieces to get an in-depth understanding of the procedure; before moving ahead, let's see some mathematical terms we will see in the following explanation.

In the data science world, classification rules are known as hypotheses; so when we will say *building the hypothesis*, it is nothing but building the classification rule or something as simple as choosing a threshold to categorize the data instances. These hypotheses are denoted by the alphabet h , so if we want to write a classifier in the form of a function, we will write it as:

$$y = h(x)$$

In the previous equation, x is the input and y is the output of the classifier, so for the i^{th} instance, the output of the classifier will be in the following form:

$$y_i = h(x_i)$$

Similarly, if we are working with an ensemble of multiple classifiers, then for an instance i , the output of classifier t will be written as:

$$y_i = h_t(x_i)$$

In the case of boosting, where the output is the weighted sum of multiple classifiers, we can write it in the following form:

$$H(x) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(x) \right)$$

The final classifier consists of T weak classifiers. $h_t(x)$ is the output of weak classifier t (outputs are limited to -1 or $+1$). α_t is the weight applied to classifier t as determined by AdaBoost. So the final output is just a linear combination of all the weak classifiers, and then we make our final decision simply by looking at the sign of this sum.

The classifiers are trained one at a time. After

each classifier is trained, we update the probabilities of each of the training examples that appear in the training set for the next classifier.

The first classifier ($t = 1$) is trained with equal probability given to all training examples. After it's trained, we compute the output weight (alpha) for that classifier:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$$

The output weight, α_t , is fairly straightforward. It's based on the classifier's error rate, ϵ_t . ϵ_t is just the number of misclassifications over the training set divided by the training set size.

Here's a plot of what α_t will look like for classifiers with different error rates:

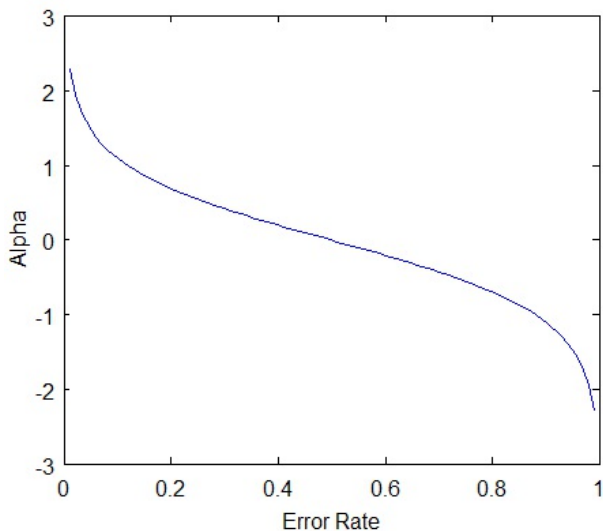


Figure 5.12: Alpha versus Error Rate

There are three bits of intuition from this graph:

1. The classifier weight grows exponentially as the error approaches zero. Better classifiers are given exponentially more weight.
2. The classifier weight is zero if the error rate is 0.5. A classifier with 50% accuracy is no better than random guessing, so we ignore it.

3. The classifier weight grows exponentially negative as the error approaches one. We give a negative weight to classifiers with worse than 50% accuracy.

Whatever that classifier says, do the opposite!

After computing the alpha for the first classifier, we update the training example weights using the following formula:

$$D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_t h_t(x_i))}{Z_t}$$

The variable D_t is a vector of weights, with one weight for each training example in the training set. i is the training example number. This equation shows you how to update the weight for the i^{th} training example.

The paper describes D_t as a distribution. This just means that each weight $D(i)$ represents the probability that the training example i will be selected as part of the training set.

To make it a distribution, all of these probabilities should add up to 1. To ensure this, we normalize the weights by dividing each of them by the sum of all the weights, Z_t . So, for example, if all of the calculated weights add up to 12.2, then we divide each of the weights by 12.2 so that they sum up to 1.0 instead.

This vector is updated for each new weak classifier that is trained. D_t refers to the weight vector used when training classifier t .

This equation needs to be evaluated for each of the training samples i (x_i, y_i). Each weight from the previous training round is going to be scaled up or down by this exponential term.

To understand how this exponential term behaves, let's first look at how **exp(x)** behaves:

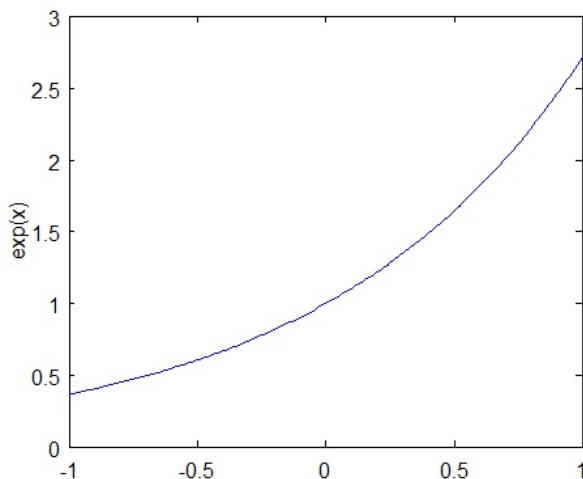


Figure 5.13: x vs exponent of x

The function **exp(x)** will return a fraction for negative values of x , and a value greater than 1 for positive values of x . So the weight for the training sample i will be either increased or decreased depending on the final sign of the term $\alpha * y * h(x)$. For binary classifiers, whose output is constrained to either -1 or $+1$, the terms y and $h(x)$ only contribute to the sign and not the magnitude.

y_i is the correct output for the training example i , and $h_t(x_i)$ is the predicted output

by classifier t on this training example. If the predicted and actual output agree, $y * h(x)$ will always be $+1$ (either $1 * 1$ or $-1 * -1$). If they disagree, $y * h(x)$ will be negative.

Ultimately, misclassifications by a classifier with a positive alpha will cause this training example to be given a larger weight, and vice versa.

Note that by including an alpha in this term, we are also incorporating the classifier's effectiveness into consideration when updating the weights. If a weak classifier misclassifies an input, we don't take that as seriously as a strong classifier's mistake.

So, after completing the preceding procedure, we can predict output of a test instance in the following manner:

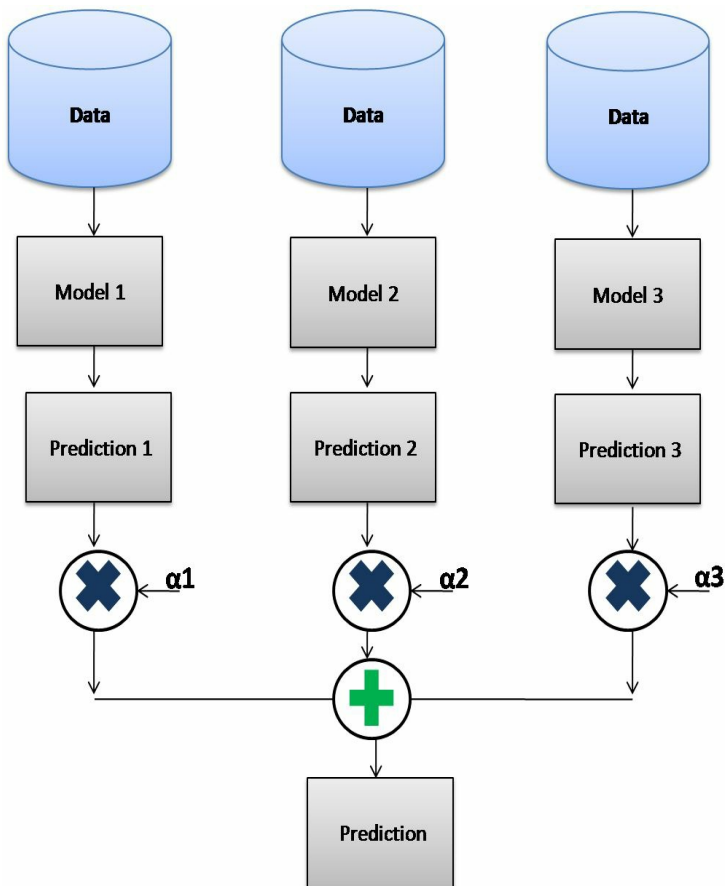


Figure 5.14: Testing the AdaBoost classifier

Alphas are working as the weights of the classifier; if the classifier has a lesser error rate, then the alpha will be more. So we will multiply the predicted output of the classifier

with the alpha value that we have calculated during the training procedure. At the end, after getting all of the weighted predictions, we will sum them up to get the final prediction.

So now, enough about the working procedure of an AdaBoost classifier! It's time for action. We will implement all the discussions in Python code to see how it will help us to create a strong classifier.

So what do we need to implement a framework of classification? The following are the ingredients:

- First, and most important, is a classifier that supports the classification of weighted instances
- A function that calculates the weighted error to calculate the rate of change of weights, in short, alphas
- The final piece of the puzzle will be the creation of an ensemble of classifiers

So let's start with it, step by step.

Weak classifier

A **weak classifier** is nothing but a threshold value that can divide the data into two classes. Experiments show that the error rate must be less than 0.5 ; that is, we should get an accuracy of the classification more than 50%. Now, why is this known as a weak classifier? The answer is simple. Because with a single threshold value, we can't get a very high accuracy. So I have a question for you: how to pick a threshold value that can assure you maximum accuracy? Did we have done something like that earlier? Yes we did! Don't you remember? Gini index! Yes, we have tested and picked a threshold value for a perfect split. And we will do the same again; we will pick a threshold and test whether it can be our weak classifier or not. Such threshold values are known as decision stumps; these are nothing but decision trees with depth 1:

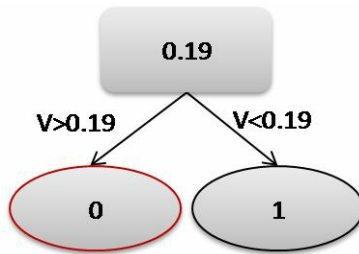


Figure 5.15: Decision stump (weak learner)

But there is a big difference in the Gini score calculation for decision stumps in the AdaBoost method; let's look at the code block of the previously used Gini score and a bit of explanation of it.

To find out the Gini score from a dataset, our input ingredient will be:

- Class proportion
- The number of instances that belong to a group

The class proportion can be calculated using the following formula:

```
| Proportion =  
| class_value_count/number_of_instances_in_the_group
```

So for the preceding formula, the total class proportion for table 1 will be 0.5, and if we consider two groups of less than 0 and greater than 0, we will get class proportion 1 for each group.

The formula for Gini score is:

```
| Gini_index = sum(proportion*(1.0-proportion))
```

As you can see, we have used class occurrence to calculate class probability or proportion; this tells us how well splits can be found in the given groups. The code implementation of the preceding formula is as follows:

```
import numpy as np
# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):

    #Initialize Gini variable
    gini = 0.0

    #Calculate proportion for each class
    for class_value in class_values:

        #Extract groups
        for group in groups:

            #Number of instance in the group
            size = len(group)
```

```

        if size == 0:
            continue

        #Initialize a list to store class index
of the instances
        r = []

        #get class of each instance in the
group
        for row in group:
            r.append(row[-1])

            #Count number of instances belongs
to current class
            class_count = r.count(class_value)

            #Calculate class proportion
            proportion =
class_count/float(size)

            #Calculate Gini index
            gini += (proportion * (1.0 -
proportion))
        return gini

```

Now, as we have a good understanding of the concept of AdaBoost classifier, we need to incorporate weights to calculate the class proportion so that the created split can reduce the error made by the previous classifier. Where do we need to make the change? As we have seen before, a calculation of proportion is required to calculate the Gini score for the given threshold value:

```
| Proportion =  
| class_value_count/number_of_instances_in_the_group
```

What we will do is replace these class counts with the sum of the weights for the given class. The expression will look like:

```
| Proportion =  
| class_weights_sum/sum_of_all_the_weights
```

We will do this to get the weighted class proportion for the each class. Definitely, we need to make many changes to the code of our Gini score calculations. As we need to include weights in the calculations, we will put the weights into the dataset itself as a last column so that the class values will be the second last column of the dataset. I will explain the process of weight merging soon. So the code for Gini calculation for AdaBoost will look like:

```
| import numpy as np;  
| def gini_index(groups, class_values):  
|     #Initialize Gini variable  
|     gini = 0.0  
  
|     #Calculate proportion for each class  
|     for class_value in class_values:  
  
|         #Extract groups
```

```

    for group in groups:
        #Number of instance in the group
        size = len(group)
        if size == 0:
            continue

        #Initialize a list to store class
index of the instances
        r = []
        cl = []

        #get class of each instance in the
group
        for row in group:
            r.append(row[-1])#Weight Append
            cl.append(row[-2])#Class Append

        r = np.array(r)
        #Extract Class indexes of the
current class value
        class_index =
np.where(cl==class_value)

        #Initialize a variable to add the
weights of current class
        w_add=0

        #Add the weights of the current
class using class indexes
        for w in class_index[0]:
            w_add+= r[w];

        #Calculate class proportion using
weights
        proportion = w_add/np.sum(r)

        #Calculate Gini index
        gini += (proportion * (1.0 -
proportion))
    return gini

```

As you can see in the code, we are extracting class values from the second last column and corresponding weight values from the last column. Now let's summarize the preceding process in the following points:

1. We will get the groups and the class values at the input.
2. We will loop for each class value and pick groups one by one.
3. When we get the first group, we will find the row indexes of the current class occurrence.
4. Using class indexes, we will extract the weights for the current class and add them.
5. To get the proportion of the class, we will divide the summed weight value by the sum of the weights of both classes.
6. We will put the proportion into the Gini index formula and calculate the Gini index.
7. Repeat the preceding procedure for the next group and add the values of all Gini scores together for the given classes.

Return the final Gini.

So once we have got the Gini score of the given threshold value, we will store it. After calculating the Gini score for all threshold values, we will choose the threshold with the lowest Gini score, as it will give us maximum split. The following will be the procedure to select a threshold value:

1. Choose an arbitrary value from the attribute
2. Use this value as a threshold and create two groups from the attribute values such that one group will have values less than the threshold and other group will have values greater than or equal to the threshold
3. Calculate the Gini index for the groups
4. Choose the value that gives the highest Gini score as the node

We will use the same `createSplit()` function that we used in [Chapter 3, Random Forest](#), which is as follows:


```
def createSplit(attribute, threshold, dataset):

    #Initialize two lists to store the sub sets
    lesser, greater = list(), list()

    #Loop through the attribute values and
    create sub set out of it
    for values in dataset:

        #Apply threshold and create two groups
        out of data set
        if values[attribute] < threshold:
            lesser.append(values)
        else:
            greater.append(values)
    return lesser, greater
```

Now it's time to decide whether the given threshold is worth becoming a decision stump or not; we will use the same `getNode()` function from [Chapter 3, Random Forest](#), but with some modifications, because we have changed the column of the class attribute from last to last but one:

```
def getNode(dataset):
    class_values = []

    #Extract unique class values present in the
    data set
    for row in dataset:
        class_values.append(row[-2]) #Class
    values are in the second last
    column
    class_values = np.unique(class_values)
```

```

    #initialize variables to store gini score,
attribute index and
    split groups
    winnerAttribute = sys.maxsize
    attributeValue = sys.maxsize
    gScore = sys.maxsize
    leftGroup = None

    #Run loop to access each attribute and
attribute values
    for index in
range(len(dataset[0])-2):#leave last two
columns
        for row in dataset:

            #Create the groups
            groups = createSplit(index,
row[index], dataset)

            #Extract gini score for the
threshold
            gini = gini_index(groups,
class_values)

            #If gini score is lower than the
previous one choose and
            return it
            if gini < gScore:
                winnerAttribute,
attributeValue, gScore, leftGroup =
                index, row[index], gini, groups

            print("winner attribute is A%d with value
%.2f gini is:
            %.2f"%
(winnerAttribute+1,attributeValue,gScore))

    #Once done create a dictionary for node
    node =

{'attribute':winnerAttribute,'value':attributeVa
    return node

```

So we have made some changes to extract class indexes. Previously we were extracting class values from the last column, but as we have appended the weight column in the end, we need to change the class index. The second change is to run a loop to extract the threshold value. This will be up to the second last column only so that we can choose only attribute values as the threshold, not the class or weight values.

Now, as we have a node with the lowest Gini score, we have to create a decision stump using this node. As we have discussed earlier, a decision stump is nothing but a decision tree with depth equal to one, so we will modify the function `buildTree()` from [Chapter 3, Random Forest](#), to get our decision stump, as follows.

We are going to use the `terminalNode()` function from [Chapter 3, Random Forest](#), without any change:

```
| def terminalNode(group):  
|     #Get class values
```

```

        outcomes = [row[-2] for row in group]

        #Choose maximum occurred class as the
child value of the Node
        return max(set(outcomes),
key=outcomes.count)

```

Following is function for creating a decision stump;

```

def decision_stump(dataset):

    #Get node value with best gini score
    node = getNode(dataset)

    #Separate out the groups from the node and
remove them
    left, right = node['groups']
    del(node['groups'])

    #Check whether there is any element in the
groups or not
    #If there is not any element put the class
value with maximum
    occurence
    if not left or not right:
        node['left'] = node['right'] =
terminalNode(left + right)
    return node

    #Put left group's maximum occur class value
in left branch
    node['left']=terminalNode(left)

    #Put right group's maximum occur class
value in right branch
    node['right'] = terminalNode(right)

    return node

```

As you can see in the preceding function, we are first getting the `node` value and then creating the `left` and `right` branches using the maximum occurring class value in the respected group. In the end, our decision stump will look something like:

```
{'attribute': 0, 'right': -1.0, 'value': 0.5,  
'left': 1.0}
```

AdaBoost in action

Now it's time to create an AdaBoost classifier to build our concepts more strongly on the algorithm. Do you remember the box problem we had seen in the second example of the AdaBoost explanation? Well, we will solve the same problem and create an AdaBoost classifier to classify the instances into respected classes.

Let's look at the problem once again.

You can see the box with some blue + and red - signs; let's assume the + sign instances are positive instances with class label $+1$ and red ones are negative instances with class label -1 :

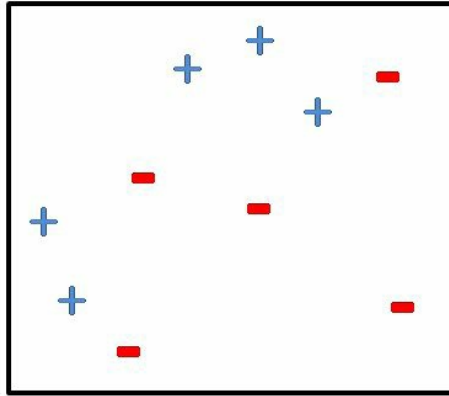


Figure 5.16: Nonlinear separation problem

First, we need to create these instances using their location coordinates; in the following figure, you can see these points. Let's create an array for the preceding coordinate values and visualize them on the graph:

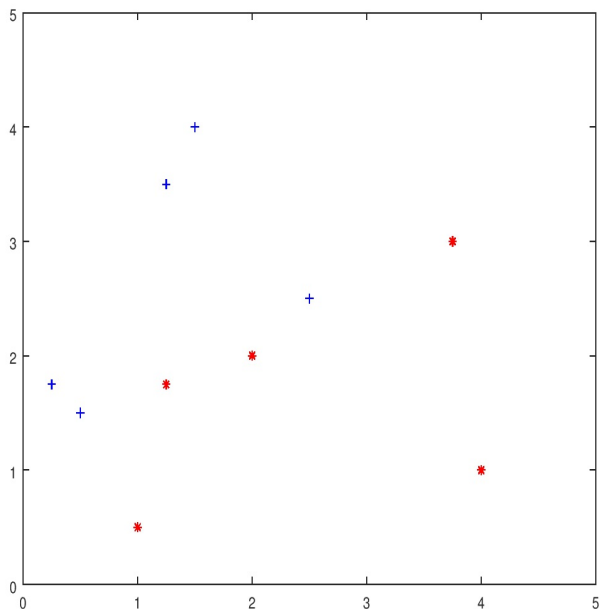


Figure 5.17: Nonlinear separation problem

Let's load the dataset in the array:

```
dataset = [[0.25000, 1.75000, 1.00000],
            [1.25000, 1.75000, -1.00000],
            [0.50000, 1.50000, 1.00000],
            [1.00000, 0.50000, -1.00000],
            [1.25000, 3.50000, 1.00000],
            [1.50000, 4.00000, 1.00000],
            [2.00000, 2.00000, -1.00000],
            [2.50000, 2.50000, 1.00000],
            [3.75000, 3.00000, -1.00000],
            [4.00000, 1.00000, -1.00000]]
```


The preceding figure is not created in Python; it's done on octave, another programming language used by the data science community. So you can see in there that points cannot be separated by using just a line of the plane. It looks like only 10 points are there, but trust me folks, it's way more difficult to classify the preceding points with 100% accuracy. For a proof of my point, I will first apply the classification and regression trees to segment the preceding dataset. Then we will go ahead in the game. We will use CART implemented in [Chapter 3](#), *Random Forest*.

Let's build the tree and see what happens:

```
tree = build_tree(dataset,5,1)#We will build
the tree for depth 5
pprint.pprint(tree)
pre = []#For storing Prediction
act = []#For storing Actual values
for row in dataset:
    prediction = predict(tree, row)
    pre.append(prediction)
    actual = act.append(row[-1])
acc = accuracy_metric(act, pre)
print('training accuracy: %.2f'%acc)
```

After execution, we will get the following

results.

Our tree will look like the following:

```
| {'attribute': 0,  
  'left': 1.0,  
  'right': {'attribute': 1,  
            'left': {'attribute': 0,  
                    'left': -1.0,  
                    'right': -1.0,  
                    'value': 4.0},  
            'right': 1.0,  
            'value': 3.0},  
  'value': 0.5}
```

The following are the predictions made by the preceding tree:

```
| Expected=1, Got=1  
  Expected=-1, Got=-1  
  Expected=1, Got=1  
  Expected=-1, Got=-1  
  Expected=1, Got=1  
  Expected=1, Got=1  
  Expected=-1, Got=-1  
  Expected=1, Got=-1  
  Expected=-1, Got=-1  
  Expected=-1, Got=-1
```

The accuracy of the decision tree classification is:

```
| training accuracy: 90.00
```

So, you can see clearly that our tree is

underfitting for this highly nonlinear classification problem with just 10 points.

We will change the algorithm and apply the AdaBoost algorithm to create some decision boundaries to classify our data into two classes.

Let's start with the implementation of the component for our AdaBoost algorithm.

We have components for building a decision stump; now we will implement functions that are part of the AdaBoost algorithm.

The first function we will need is computation of the weighted error to decide the rate of change of the weights; we will add the `getError()` function to our code block to estimate the error of the classification:

```
def getError(actual,predicted,weights):  
    #Initialize the error variable  
    error = 0  
  
    #We will store the error of each instance  
    in a vector  
    error_vec=[]
```

```

        #Run a loop to calculate error for each
instance
        for i in range(len(actual)):
            diff = predicted[i]!=actual[i]
            #Weights multiplication to the
difference of actual and
predicted values
            error+= weights[i]*(diff)

        #Append the difference to the error
vector
        error_vec.append(diff)

    return error,error_vec

```

Now we will need a function to predict the output of the decision stump we have created; for this, we will use the same function we implemented in [Chapter 3](#), *Random Forest* to make predictions using a decision tree:

```

def predict(node, row):
    #Get the node value and check whether the
attribute value is less than or
equal.
    if row[node['attribute']] <= node['value']:
        #If yes enter into left branch and
check whether it has another node or
the class value.
        #If there is no node in the branch
        return node['left']
    else:
        return node['right']

```

We have modified the preceding function by removing recursion from the code; as we are

dealing with a decision tree with a depth of only one, we need not to put any recursion in the code.

Now let's jump straight into the function definition, `AdaBoostAlgorithm()`, in which we will put all the elements together to create the classifier:

```
def AdaBoostAlgorithm(dataset, iterations):  
    #Initialize the weights of the size of data set  
    weights =  
    np.ones(len(dataset), dtype="float32")/len(dataset)  
  
    dataset = np.array(dataset)  
  
    #Add Weights column to the data set(Now last column will be the weights)  
    dataset = np.c_[dataset, weights]  
  
    #Create an empty list to store alpha values  
    alphas = []  
  
    #Create a list to add weak learners(decision stumps)  
    weaks = []  
  
    #Lets run the loop for number of iteration(number of classifiers)  
    for itr in range(iterations):  
  
        #Create decision tree from the non weighted data-set  
        ds = decision_stump(dataset)
```

```

        #Create a list to store the predictions
of the decision stump
        pred=[]

        #Create a list to store actual outputs
        actual = []

        #Let's predict output for each instance
in the data set
        for row in dataset:
            actual.append(row[-2])
            pred.append(predict(ds, row))

        #Here we will find out difference
between predicted and actual output
        error,error_vec = getError(actual,
pred,weights)

        #If error is equal to 0.5 classifier is
not able to classify the data set
        if error==0.5:
            continue
        eps = sys.float_info.epsilon

        #Let's find out the alpha with the help
of error
        alpha = 0.5 * np.log((1-
error)/(error+eps))

        print("Error: %.3f and alpha: %.3f"%
(error,alpha))

        #Create empty vector to store weight
updates
        w = np.zeros(len(weights))

        # Update the weights using alpha value
for i in range(len(error_vec)):

            #For wrong prediction increase the
weights
            if error_vec[i]!=0:
                w[i] = weights[i] *

```

```

np.exp(alpha)
        #For correct prediction decrease
the weights
        else:
            w[i] = weights[i] * np.exp(-
alpha)

        #Normalize the weights and update
previous weight vector
        weights = w / w.sum()

        #Put the updated weights into the data
set by over-writing previous
weights
        dataset[:, -1]=weights

        #Append alpha value to the list to used
at the time of testing
        alphas.append(alpha)

        #Append the weak learner to the list
        weaks.append(ds)

return weaks, alphas

```

Yes it is a very big code block, but believe me, there is nothing fancy happening there. We can summarize the whole code in the following points:

1. First initialize the instances' weights with uniform distribution
2. Now create a decision stump and feed the data instances for prediction

3. Calculate the weighted error using a weight matrix between actual and predicted
4. Using the error value, calculate the rate of change (alpha) for the weights.
5. According to the prediction error, update the weights
6. Replace the previous weights by the updated weights
7. Repeat the procedure for the number of classifiers needed

You see, it's very simple! Now it's time to apply it to our example.

Let's start with loading our data into the variable:

```
dataset = [[0.25000, 1.75000, 1.00000],  
           [1.25000, 1.75000, -1.00000],  
           [0.50000, 1.50000, 1.00000],  
           [1.00000, 0.50000, -1.00000],  
           [1.25000, 3.50000, 1.00000],  
           [1.50000, 4.00000, 1.00000],  
           [2.00000, 2.00000, -1.00000],  
           [2.50000, 2.50000, 1.00000],  
           [3.75000, 3.00000, -1.00000],  
           [4.00000, 1.00000, -1.00000]]
```


And call the `AdaBoostAlgorithm()` function; we will build 9 cascade classifiers to classify our dataset:

```
| [weaks, alphas] = AdaBoostAlgorithm(dataset, 9)
```

`weak`s and `alpha`s will be the trained weak classifiers and their `weights`; let's see what happens internally when we call the preceding function.

First we initialize `weights` for appending into dataset using:

```
| weights =  
| np.ones(len(dataset), dtype="float32")/len(dataset)
```

Our `weights` will be uniform for the first classifier; it will look like:

```
| weights[ 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  
| 0.1  0.1]
```

Append these `weights` to dataset:

```
| dataset = np.c_[dataset, weights]
```

The modified dataset will look like:

```
[[ 0.25  1.75  1.    0.1 ]
 [ 1.25  1.75 -1.    0.1 ]
 [ 0.5   1.5   1.    0.1 ]
 [ 1.    0.5  -1.    0.1 ]
 [ 1.25  3.5   1.    0.1 ]
 [ 1.5   4.    1.    0.1 ]
 [ 2.    2.   -1.    0.1 ]
 [ 2.5   2.5   1.    0.1 ]
 [ 3.75  3.   -1.    0.1 ]
 [ 4.    1.   -1.    0.1 ]]
```

Let's get our first `decision_stump` and see how it performs:

```
| ds = decision_stump(dataset)
```

This `decision_stump` will look like:

```
| {'right': -1.0, 'value': 0.5, 'left': 1.0,
|  'attribute': 0}
```

As you can see, the threshold value is 0.5 from attribute is 0 (column 1), in which the right branch has a negative class while the left has a positive class value. Let's see how it will perform in classifying our dataset:

```
| for row in dataset:
|     actual.append(row[-2])
|     pred.append(predict(ds, row))
```

The misclassifications made by our decision

stump will be:

SN	Actual	Pred
1	1	1
2	-1	-1
3	1	1
4	-1	-1
5	1	-1
6	1	-1
7	-1	-1

8	1	-1
9	-1	-1
10	-1	-1

Table 5.4 Errors made by the first weak learner

So, the preceding table shows that our weak learner is predicting three instances wrongly while seven instances are correct; it's not a bad prediction considering a single threshold value.

We have got our first decision boundary, which will look like this figure, clearly showing the three wrong predictions of positive instances:

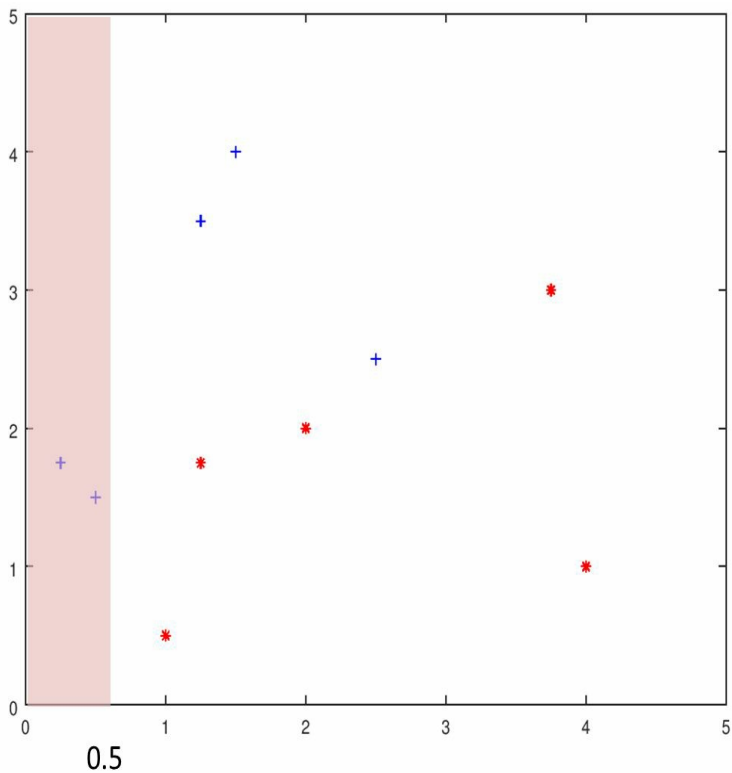


Figure 5.18: First decision boundary

So let's calculate the `error` made by the first classifier and then calculate the rate of change of weight using the `error` value:

```
| error,error_vec = getError(actual, pred,  
| weights)
```

As `weights` for each of the instances are the same this time, they have no effect on `error` calculation:

```
| Error between predicted and actual output is:  
| 0.30
```

Let's calculate `alpha`:

```
| alpha = 0.5 * np.log((1-error)/(error+eps))  
| Alpha value for changing the weights is: 0.42
```

It's time to update the previous `weights` such that a wrong prediction should get a higher weight value compared to a correct prediction, and we will do this with the following line:

```
| #For wrong prediction increase the weights  
| if error_vec[i]!=0:  
|     w[i] = weights[i] * np.exp(alpha)  
| #For correct prediction decrease the weights  
| else:  
|     w[i] = weights[i] * np.exp(-alpha)
```

Let's look at the updated weights after the preceding line after normalization:

--	--	--	--

SN	Actual	Predicted	
1	1	1	
2	-1	-1	
3	1	1	
4	-1	-1	
5	1	-1	
6	1	-1	
7	-1	-1	
8	1	-1	

9	-1	-1	
10	-1	-1	

Table 5.4 Update weights for the next iteration

It looks perfectly fine, as the incorrect predictions are getting more than double the weights compared to the correct instances. In the next step, we will append the updated weights to the dataset and again get a decision stump; let's get straight to the new stump value:

```
| {'value': 2.5, 'left': 1.0, 'attribute': 0,
|   'right': -1.0}
```

The new decision stump creates a boundary as follows:

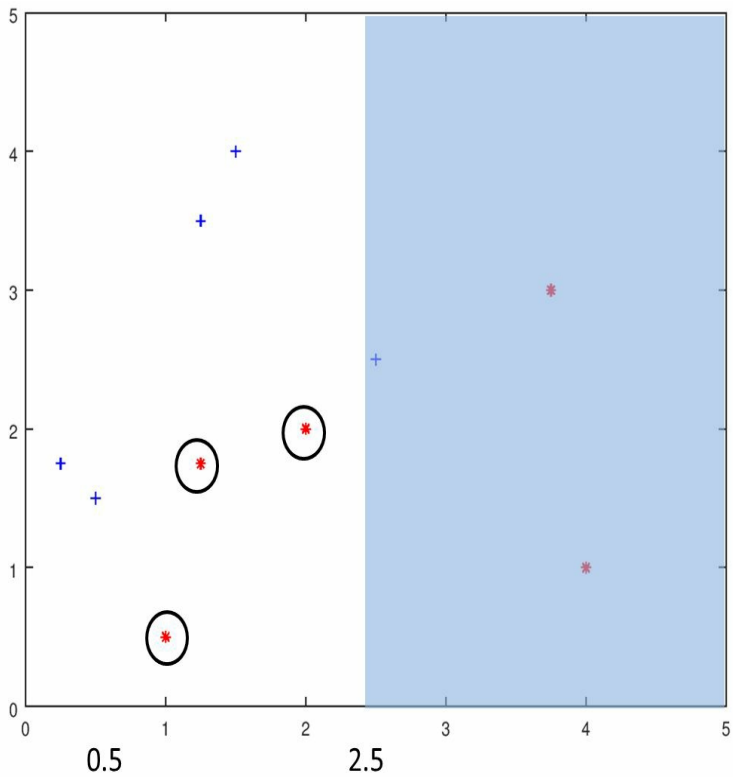


Figure 5.19: Second decision boundary

Following table shows prediction made by decision stump:

SN	Actual	Predicted

1	1	1
2	-1	1
3	1	1
4	-1	1
5	1	1
6	1	1
7	-1	1
8	1	1
9	-1	-1

10	-1	-1
----	----	----

Table 5.5 Error made by the second weak learner

The `Error` and `alpha` for the classifier are:

| **Error: 0.214** and **alpha: 0.650**

You can see that for a lower error, we are getting a high alpha value, which forces the final prediction to be biased to the classifiers with higher alpha values and lower errors. This logically makes the correct prediction at the end.

Now let's take on the updated weights:

SN	Actual	Predicted	Weights
1	1	1	0.04

2	-1	1	0.16
3	1	1	0.04
4	-1	1	0.16
5	1	1	0.10
6	1	1	0.10
7	-1	1	0.16
8	1	1	0.10
9	-1	-1	0.04
10	-1	-1	0.04

Table 5.6: Weight update for next iteration

Again the updates look fine, which indicates that we are on the right path.

the next decision stump will be:

```
{'value': 3.0, 'left': -1.0, 'attribute': 1, 'right': 1.0}
```

The prediction made by the preceding decision stump is:

SN	Actual	Predicted
1	1	-1
2	-1	-1
3	1	-1
4	-1	-1

5	1	1
6	1	1
7	-1	-1
8	1	-1
9	-1	-1
10	-1	-1

Table 5.7: Error made by the third weak learner

The Error and α value for the classifier are:

Error: 0.197 and α : 0.703

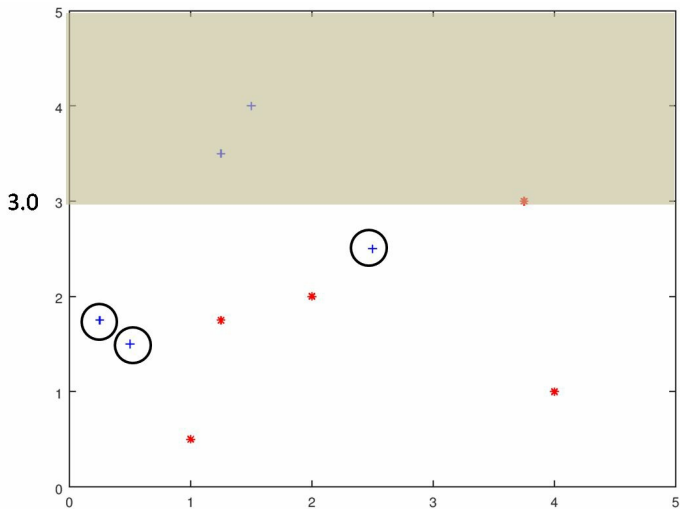


Figure 5.20: Third decision boundary

The following is the summary of all nine classifier statistics:

Classifier 0 stats:

winner attribute is 1 with value 0.50 gini is:
0.47

{'attribute': 0, 'value': 0.5, 'left': 1.0,
'right': -1.0}

Error: 0.300 and alpha: 0.424

Classifier 1 stats:

winner attribute is 1 with value 2.50 gini is:
0.38

{'attribute': 0, 'value': 2.5, 'left': 1.0,
'right': -1.0}

Error: 0.214 and alpha: 0.650

Classifier 2 stats:

winner attribute is 2 with value 3.00 gini is:
0.38
{'attribute': 1, 'value': 3.0, 'left': -1.0,
'right': 1.0}
Error: 0.197 and alpha: 0.703

Classifier 3 stats:

winner attribute is 2 with value 1.00 gini is:
0.40
{'attribute': 1, 'value': 1.0, 'left': -1.0,
'right': 1.0}
Error: 0.236 and alpha: 0.588

Classifier 4 stats:

winner attribute is 1 with value 0.50 gini is:
0.43
{'attribute': 0, 'value': 0.5, 'left': 1.0,
'right': -1.0}
Error: 0.263 and alpha: 0.516

Classifier 5 stats:

winner attribute is 2 with value 1.00 gini is:
0.46
{'attribute': 1, 'value': 1.0, 'left': -1.0,
'right': 1.0}
Error: 0.339 and alpha: 0.334

Classifier 6 stats:

winner attribute is 2 with value 3.00 gini is:
0.47
{'attribute': 1, 'value': 3.0, 'left': -1.0,
'right': 1.0}
Error: 0.331 and alpha: 0.351

Classifier 7 stats:

winner attribute is 1 with value 2.50 gini is:
0.47
{'attribute': 0, 'value': 2.5, 'left': 1.0,
'right': -1.0}
Error: 0.355 and alpha: 0.299

Classifier 8 stats:


```
winner attribute is 2 with value 2.00 gini is:  
0.42  
{'attribute': 1, 'value': 2.0, 'left': -1.0,  
'right': 1.0}  
Error: 0.125 and alpha: 0.971
```

As you can see in the preceding nodes, some nodes have repeating node values, but the error and alphas associated with them are different, which forces the classifiers towards different predictions. So don't think that those classifiers are redundant in the calculations. Let's see all the boundaries created by the classifiers:

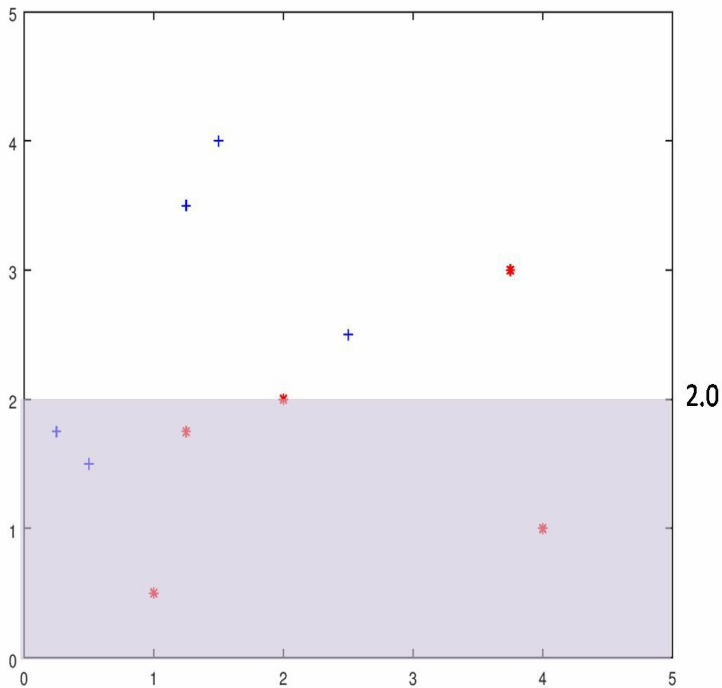


Figure 5.21: N^{th} decision boundary

All other boundaries are repeated, so there is no need to plot them here. Their predictions will be the same, but due to their different alpha value, the contributions of their predictions will vary. For example, if many of the classifiers with x as the node value are classifying a point in class 1 with a low

$\alpha=0.1$, and one classifier is classifying the same point in class -1 with $\alpha=0.5$, then the weight contribution of the classifier (with higher) will be more. This causes the decision boundary to move towards class -1 . This is because classifier with a higher alpha value shows less error—error is inversely proportional to the alpha.

This figure shows the final classification boundary achieved by the algorithm:

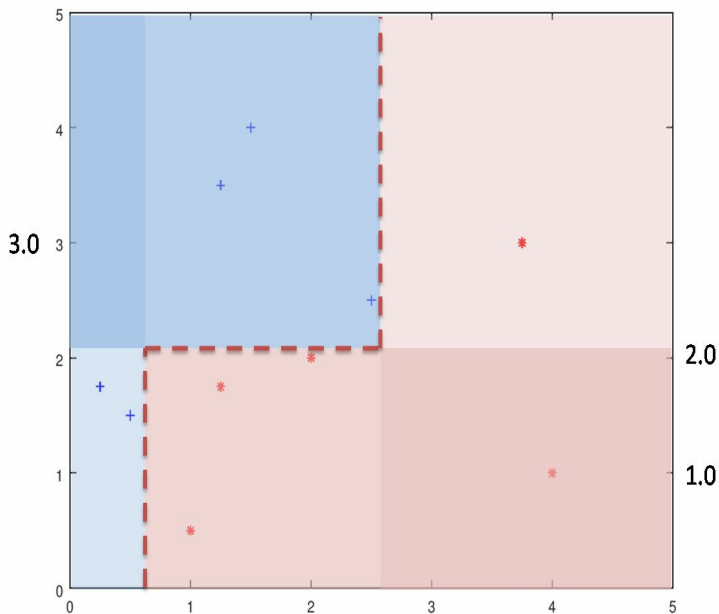


Figure 5.22: Final decision Boundary by the ensemble of weak learners

Let's put all the classifiers together and see what the final output of the algorithm is; to evaluate the final output, we will follow the steps mentioned here:

1. Pick an instance and predict its output with all the classifiers
2. Multiply the alpha values of the classifier with the predicted output
3. Sum all the values
4. Choose the sign of the value as the class label

We will implement a code block to do the preceding task; it will look like:

```
#Make empty lists to store predicted and actual
outputs
prediction=[]
actual = []

#Run a loop to extract each instance from the
data set
for row in dataset:

#Create a list to store predictions from
different classifier for the test
#instance
```

```

preds = []

#Feed the instance to different classifiers
for i in range(len(weaks)):

    #Multiply the predicted output with the
    alpha value of the classifier
    p = alphas[i]*predict(weaks[i], row)

    #Add the weighted prediction to the list
    preds.append(p)

#Sum up output of all the classifiers and take
their sign as the prediction
final = np.sign(sum(preds))

#Append the final output to the prediction list
and actual output to the actual
#list
prediction.append(final)
actual.append(row[-1])

```

Now it's time to calculate the accuracy of the ensemble classifier:

```

def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0

```

So, the output of our classifier is:

```

Expected=1, Got=1
Expected=-1, Got=-1
Expected=1, Got=1
Expected=-1, Got=-1
Expected=1, Got=1

```

```
Expected=1, Got=1  
Expected=-1, Got=-1  
Expected=1, Got=1  
Expected=-1, Got=-1  
Expected=-1, Got=-1  
  
Accuracy: 100.00
```

As you can see, our classifier has dealt with a highly nonlinear problem with the 100 percent accuracy. There are very few algorithms available that can separate the preceding dataset perfectly into two groups, and you just have learned one of them.

Application of the AdaBoost classifier in face detection

Yup, you have read it right! We are going to discuss our first computer vision domain algorithm, and trust me it is one of the most popular domains in the machine learning world. So the first question here is: what is computer vision? In its simplest form, computer vision is a set of algorithms that are applied to real-world images to extract meaningful information from them, such as object detection in images (finding faces or a football in an image, or tracking a human in a video). All these are part of the computer vision domain, where algorithms give the power of visualization to the machine; that is why this field is also known as machine vision.

So, I think this is enough information regarding the domain. Now we will discuss a bit about Images. Images are two-dimensional matrices with numbers; these numbers are known as intensity values and the coordinates of the matrix are known as pixels. So an image consists of lots of pixels, which have different intensity values and form different structures. We can visualize these through our eyes. So what's the big deal in here? Maths!!! If we have a matrix with real data values, we can apply matrix mathematics to our images, and that's what we are going to do next in the example.

Face detection using Haar cascades

Object detection using Haar feature-based cascade classifiers is an effective object detection method proposed by Paul Viola and Michael Jones in their paper *Rapid Object Detection using a Boosted Cascade of Simple Features* in 2001. It is a machine-learning-based approach where a **cascade** function is trained from a lot of positive and negative images. It is then used to detect objects in other images.

Here, we will work with face detection. Initially, the algorithm needs a lot of positive images (images of faces) and negative images (images without faces) to train the classifier. Then we need to extract features from it. Features are nothing but numerical information extracted from the images that can be used to distinguish one image from

another; for example, a histogram (distribution of intensity values) is one of the features that can be used to define several characteristics of an image even without looking at the image, such as dark or bright image, the intensity range of the image, contrast, and so on. We will use Haar features to detect faces in an image. Here is a figure showing different Haar features:

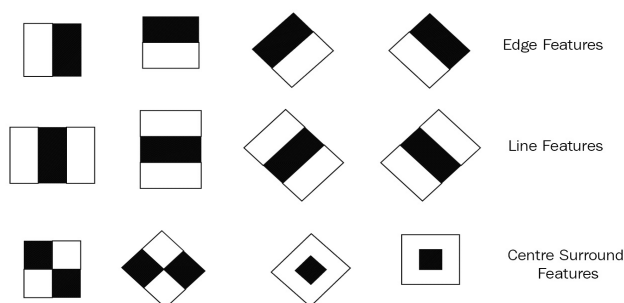


Figure 5.23: Haar features

These features are just like the convolution kernel; to know about convolution, you need to wait for the following chapters. For a basic understanding, convolutions can be described as in the following figure:

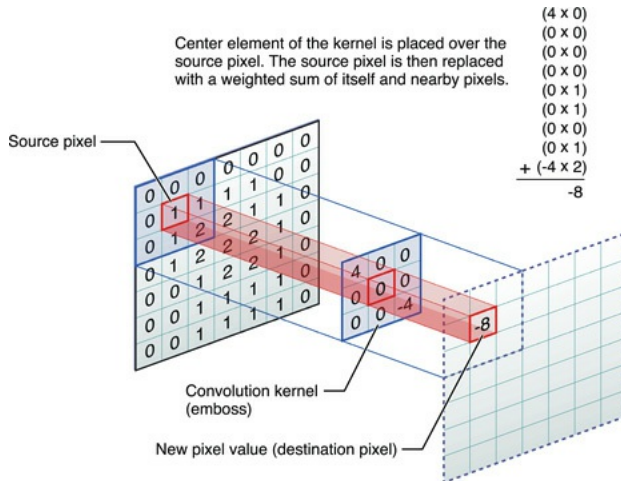


Figure 5.24: Process of convolution on image

So we can summarize convolution with these steps:

1. Pick a pixel location from the image
2. Now crop a sub-image with the selected pixel as the center from the source image with the same size as the convolution kernel
3. Calculate an element-wise product between the values of the kernel and sub-image
4. Add the result of the product
5. Put the resultant value into the new

image at the same place where you picked up the pixel location

Now we are going to do a similar kind of procedure, but with a slight difference for our images. Each feature of ours is a single value obtained by subtracting the sum of the pixels under the white rectangle from the sum of the pixels under the black rectangle.

Now, all possible sizes and locations of each kernel are used to calculate plenty of features. (Just imagine how much computation it needs. Even a 24x24 window results in over 160,000 features.) For each feature calculation, we need to find the sum of the pixels under the white and black rectangles. To solve this, we will use the concept of integral image; we will discuss this concept very briefly here, as it's not a part of our context.

Integral image

Integral images are those images in which the pixel value at any (x,y) location is the sum of the all pixel values present before the current pixel. Its use can be understood by the following example:

5	4	3	8	3
3	9	1	2	6
9	6	0	5	7
7	3	6	5	9
1	2	2	8	3

5	9	12	20	23
8	21	25	35	44
17	36	40	55	71
24	46	56	76	101
25	49	61	89	117

Figure 5.25: Image on the left and its integral image on the right

Let's see how this concept can help reduce computation time; let us assume a matrix A of size 5×5 representing an image, as shown

here:

5	4	3	8	3
3	9	1	2	6
9	6	0	5	7
7	3	6	5	9
1	2	2	8	3

Figure 5.26: Example matrix

Now, let's say we want to calculate the average intensity over the area highlighted:

5	4	3	8	3
3	9	1	2	6
9	6	0	5	7
7	3	6	5	9
1	2	2	8	3

Figure 5.27: Region for addition

Normally, you'd do the following:

$$\left| \begin{array}{l} 9 + 1 + 2 + 6 + 0 + 5 + 3 + 6 + 5 = 37 \\ 37 / 9 = 4.11 \end{array} \right.$$

This requires a total of 9 operations.

Doing the same for 100 such operations would require:

| 100 * 9 = **900 operations.**

Now, let us first make a integral image of the preceding image:

5	9	12	20	23
8	21	25	35	44
17	36	40	55	71
24	46	56	76	101
25	49	61	89	117

Figure 5.28: Integral matrix

Making this image required a total of 56 operations.

Again, focus on the highlighted portion:

5	9	12	20	23
8	21	25	35	44
17	36	40	55	71
24	46	56	76	101
25	49	61	89	117

Figure 5.29: Region for addition

To calculate the avg intensity, all you have to do is:

$$\left| \begin{array}{l} (76 - 20) - (24 - 5) = 37 \\ 37 / 9 = 4.11 \end{array} \right.$$

This required a total of 4 operations.

To do this for 100 such operations, we would require:

$$| 56 + 100 * 4 = \mathbf{456 \text{ operations}} .$$

For just a hundred operations over a 5x5 matrix, using an integral image requires about 50% less computations. Imagine the difference it makes for large images and other such operations.

Creation of an integral image changes other sum difference operations by almost $O(1)$ time complexity, thereby decreasing the number of calculations.

It simplifies the calculation of the sum of pixels—no matter how large the number of pixels—to an operation involving just four pixels. Nice, isn't it? It makes things superfast.

However, among all of these features we calculated, most of them are irrelevant. For example, consider the following image. The top row shows two good features. The first feature selected seems to focus on the property that the region of the eyes is often darker than the region of the nose and cheeks. The second feature selected relies on the property that the eyes are darker than the bridge of the nose. But the same windows applying on cheeks or any other part is irrelevant. So how do we select the best features out of 160000+ features? It is achieved by AdaBoost.

To do this, we apply each and every feature on all the training images. For each feature, it finds the best threshold that will classify the faces as positive and negative. Obviously, there will be errors or misclassifications. We select the features with the minimum error rate, which means they are the features that best classify the face and non-face images.

The process is not as simple as



this. Each image is given an equal weight in the beginning. After each classification, the weights of misclassified images are increased. Again, the same process is done. New error rates are calculated among the new weights. This process continues until the required accuracy or error rate is achieved or the required number of features is found.

The final classifier is a weighted sum of these weak classifiers. It is called weak because it alone can't classify the image, but together with others, it forms a strong classifier. The paper says that even 200 features provide detection with 95% accuracy. Their final setup had around 6,000 features. (Imagine a reduction from 160,000+ to 6000 features. That is a big gain.)

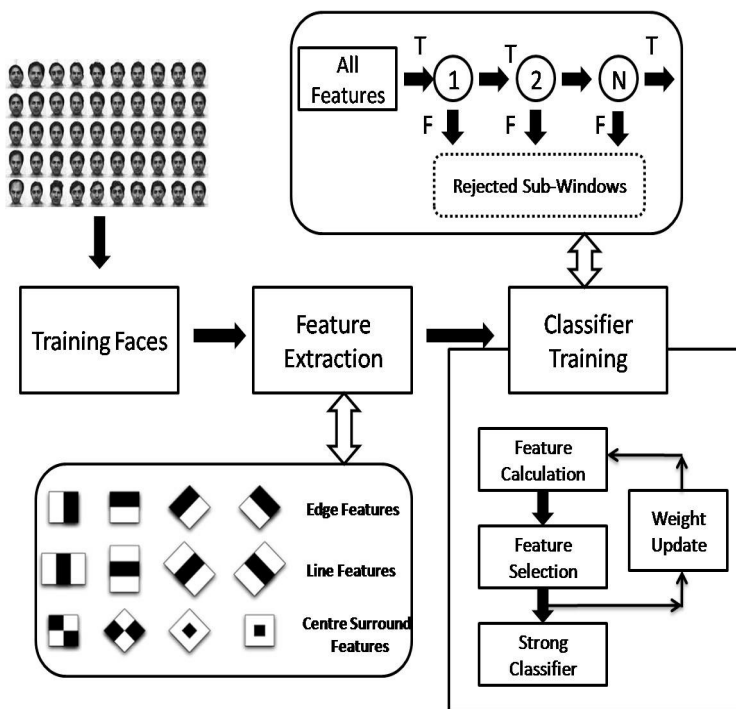


Figure 5.30: Face detection framework using the Haar cascade and AdaBoost algorithm

So now, you take an image take each 24x24 window, apply 6,000 features to it, and check if it is a face or not. Wow! Wow! Isn't this a little inefficient and time consuming? Yes, it is. The authors of the algorithm have a good solution for that.

In an image, most of the image region is non-face. So it is a better idea to have a simple method to verify that a window is not a face region. If it is not, discard it in a single shot. Don't process it again. Instead, focus on the region where there can be a face. This way, we can find more time to check a possible face region.

For this, they introduced the concept of a cascade of classifiers. Instead of applying all the 6,000 features to a window, we group the features into different stages of classifiers and apply one by one (normally first few stages will contain very few features). If a window fails in the first stage, discard it. We don't consider the remaining features in it. If it passes, apply the second stage of features and continue the process. The window that passes all stages is a face region. How cool is the plan!!!

The authors' detector had 6,000+ features with 38 stages, with 1, 10, 25, 25, and 50 features in the first five stages (two features

in the preceding image were actually obtained as the best two features from AdaBoost). According to the authors, on average, *10* features out of *6,000+* are evaluated per subwindow.

So this is a simple, intuitive explanation of how **Viola-Jones** face detection works. Read the paper for more details or check out the references in the Additional Resources section.

Implementation using OpenCV

Open Source Computer Vision (OpenCV) is a library. Open source means it's free to download and we can use it to implement various computer vision algorithms; face detection is one of them. So let's see how to do this.

Here, we will deal with detection. OpenCV contains many pre-trained classifiers for face, eyes, smiles, and so on. These XML files are stored in the `opencv/data/haarcascades/` folder. Let's create a face and eye detector with OpenCV.

First, we need to load the required XML classifiers. Then we load our input image (or video) in grayscale mode:

```
#So We will load required libraries numpy for  
matrix operations
```

```

import numpy as np

#Import OpenCV library, in python we can call
it cv2
import cv2

#OpenCV have module cascade classifier which is
based on haar cascade and #Adaboost algorithm,
so we will call direct method.

#First we will load the pre trained classifiers
for frontal face and eye #detection, which are
in the form of xml file.
face_cascade =
cv2.CascadeClassifier('haarcascade_frontalface_d
eye_cascade =
cv2.CascadeClassifier('haarcascade_eye.xml')

#Now let us load an image from the local
directory
img = cv2.imread('sachin.jpg')

#Let's convert the image from color space to
intensity value(Gray Image)
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

```

Now we find faces in the image. If they are found, it returns the positions of detected faces as `Rect(x,y,w,h)`. Once we get these locations, we can create an ROI for the face and apply eye detection to this ROI (since eyes are always on the face!!!):

```

#Here we will call the method which will find
the faces in our input image
faces = face_cascade.detectMultiScale(gray,
1.3, 5)

```

```

#Lets run a loop to create sub images of faces
from the input image using #cv2.rectangle
function
for (x,y,w,h) in faces:
    img = cv2.rectangle(img,(x,y),(x+w,y+h),
    (255,0,0),2)
    roi_gray = gray[y:y+h, x:x+w]
    roi_color = img[y:y+h, x:x+w]
    #Now let's run the classifier for eyes
detection over detected face
    #windows
    eyes =
    eye_cascade.detectMultiScale(roi_gray)
    #the following function will create the
rectangles around the eyes
    for (ex,ey,ew,eh) in eyes:
        cv2.rectangle(roi_color,(ex,ey),
        (ex+ew,ey+eh),(0,255,0),2)
#The following Lines will show the detected
face images
    cv2.imshow('img',img)
    cv2.waitKey(0)
    cv2.destroyAllWindows()

```

The result of the execution:

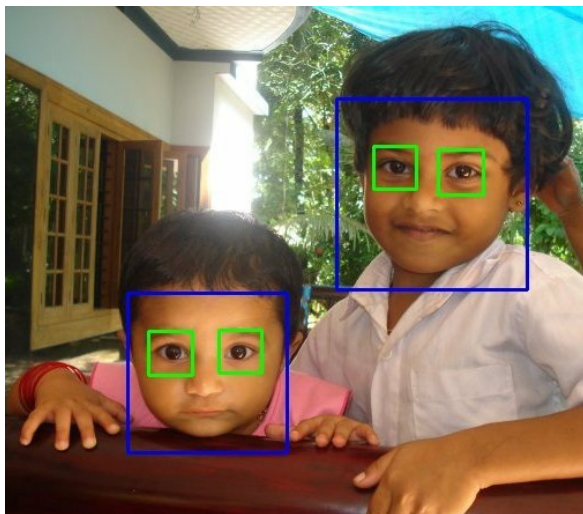


Figure 5.31: The face detection framework's result

You can get this description at the official link of the OpenCV (http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_objdetect/py_face_detection/py_face_detection.html).

Summary

It looks like we have traveled far on the journey of AdaBoost classifiers. We started with a simple example of boosting, in which our kindergarten kids helped us understand the concepts of the method. Using the same concepts, we built a working implementation of AdaBoost algorithms where our weak learner was a decision stump. Here I want to point out to you that the AdaBoost algorithm has no dependency on the choice of weak learner. It can be trained with any kind of algorithm such as support vector machines, artificial neural networks, our own random forest, and so on. However, there is only one requirement: the algorithm must support weighted inputs to calculate decision boundaries. Because of this limitation, I encourage you to use AdaBoost on the scikit-learn library. As most machine learning algorithms in scikit-learn support weighted

input instances, you can easily use those classifiers to create ensembles. Here, we saw an example implementation of AdaBoost, which is good to build the concepts behind the actual algorithm, but I strongly suggest that you go with scikit-learn's AdaBoost—a more optimized algorithm for practical applications.

So, I will say this is it for AdaBoost. We will learn a new algorithm for boosting in the following chapter. Till then, goodbye!!

Gradient Boosting Machines

In the previous chapter, we saw the power of multiple weak learners that can do magic and learn nonlinear data. We discussed boosting and saw how it can be used to solve extremely complex problems such as face detection, and we it did quite well. I just want to repeat the points we followed in the AdaBoost algorithms before moving ahead:

1. Loading data and weight each instance equally
2. Training a weak learner (we used decision stump) on the data
3. Evaluating the errors made by the classifier and giving more weight to wrongly classified instances
4. Repeating the procedure from point to for number of iterations
5. Using the error rate of the weak learner

as the weight of prediction made by the classifier when learning is complete

The preceding procedure creates multiple partitions of the dataset and creates decision boundaries, as follows:

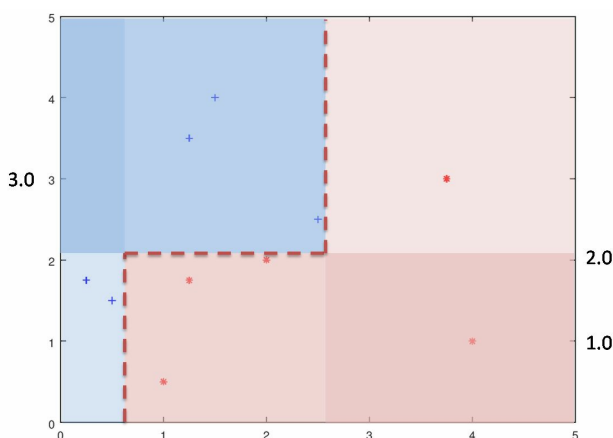


Figure 6.1: AdaBoost classifier boundaries

So, I think you have had enough of your recall session. Now it is time to ask you a question, and here it is: what will be your classifier's output for the following problem?

SN	Input	Output
----	-------	--------

1	0.00	0.00
2	1.00	0.05
3	2.00	0.10
4	3.00	0.15
63	62.00	0.10
64	63.00	0.05
65	64.00	0.00

Table 6.1: Nonlinear problem

If we will plot the preceding table, it will look as follows:

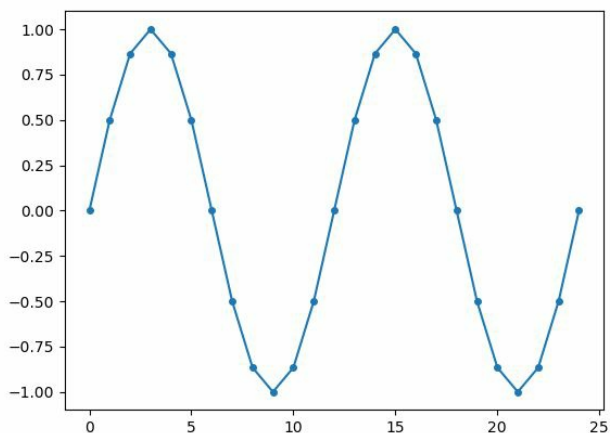


Figure 6.2: Nonlinear problem visualization

I know the first sentence that came to your mind, *what is this, man?* Well, don't panic folks! This is the problem of regression, where your input may be a continuous or categorical value, but your output is always a continuous value. As you can see in the preceding figure, for $x=1$, we get 0.05 as the output, and for $x=3$, the output is 0.15 . Do you think any classifier that we have developed can solve this problem? No! None of our classifiers can solve this

problem... oh! Wait a minute. You people are always in a hurry! I have told you this is a problem of regression. Did you hear about regression earlier? Yes, of course! We developed classification and regression trees in [Chapter 3](#), *Random Forest*.

So, we can solve the preceding problem using a regression tree (I will come back to this later) as a weak learner with AdaBoost. In the case of AdaBoost, our every classifier gives more attention to those instances that were wrongly classified by the previous classifier in the series, so we have actually changed the sample distribution by giving importance to certain instances.

Now, we will talk about a different way to get the attention of a weak learner toward the error made by the previous weak learner.

Suppose we have trained our first classifier and got the prediction as follows:

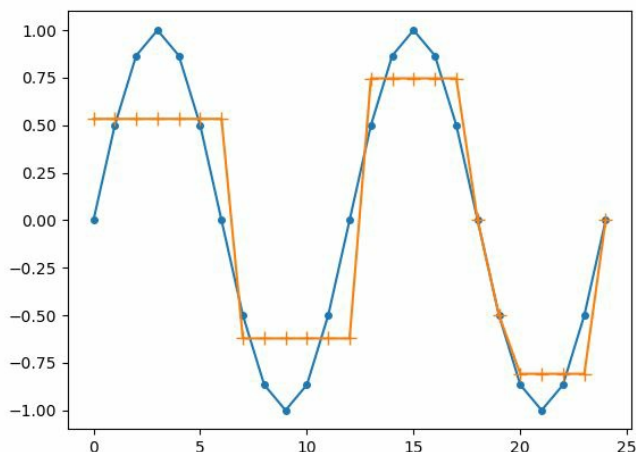


Figure 6.3: Prediction of a classifier

As you can see very clearly, our tree is underfitting for the problem as it is not able to reconstruct the output on its own. Now, we can go with the AdaBoost approach to weigh more to the miss-classified instances for the next classifier, but wait! We will do something different to this; let's look at an error made by our classifier:

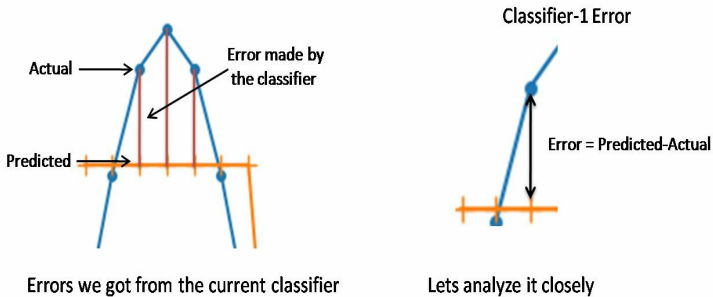


Figure 6.4: Analysis of prediction error

The preceding figure shows the amount of errors produced by our first classifier. If we write it in the form of an equation, it should look as follows:

$$\text{error} = \text{Actual} - \text{Predicted}$$

In a more mathematical way, we can write it as follows:

$$e = y - f(x)$$

Where x is the input, $f(x)$ is our classifier, model y is the actual output, and e is the error made by the classifier. Now, what if we rewrite this equation in the following form?

$$actual = prediction + error$$

Can you tell me, what happened in the preceding equation? Well, the preceding equation says that if the *error* is 0, then our actual output will be equal to the predicted output and our classifier is doing a perfect job. However, if the predicted value is more than the *actual* value, the error will be negative and we have to subtract the *error* from our *prediction*. Similarly, when the *error* is positive, we will need to add it to get the correct *prediction*.

So, what if we next add a weak learner to remember (learn) this error? Let's just add the next weak learner in the system to do this; our equation will look as follows:

$$y = f(x) + h(e)$$

Then, the error made by this classifier will be written as follows:

$$e = h(e) - f_1(x)$$

Alternatively, we can write it as follows:

$$e_n = y_{n-1} - f_n(x)$$

So, an error at n^{th} classifier will be the difference between the error learned by the previous classifier and the output of the current classifier. Similarly, the equation for the output of n^{th} classifier will be as follows:

$$y_n = f_n(x) + f_{n-1}$$

According to the preceding equation, the final prediction of the classifier will be the sum of all predictions made by the previous classifiers.

I know that now you are more confused, and you may be thinking, *Why the hell we are doing all the previous equations?* So, for your information, you have just learned the concept of **Gradient Boosting Machine (GBM)**! Yes, this is it! And believe me, there is nothing more to it. Oh! Don't you believe me? Let's break down the method itself.

Gradient Boosting Machines

In the simplest way, we can say that the gradient represents the rate of change (or the amount of change). Yes! This is the simplest definition of the gradient. It may be a positive change (positive gradient) or it may be a negative change (negative gradient). Let's understand the gradient in brief with the help of the following example:

Suppose we are traveling in a car on a mountain, I will not tell you upward or downward; at point a , we attain a height of *1000* meters and at point b , we reach *1500* meters. However, we have just covered a distance of *200* meters. Can you tell me what the rate of change (speed) of our car is and whether we are moving upward or downward? Well, the second part of the question is very funny-we all know upwards,

but how did you know? You just subtract height one by height two and the difference is positive, but the first part of the question is important, so maths gives us the following way to get rid of it:

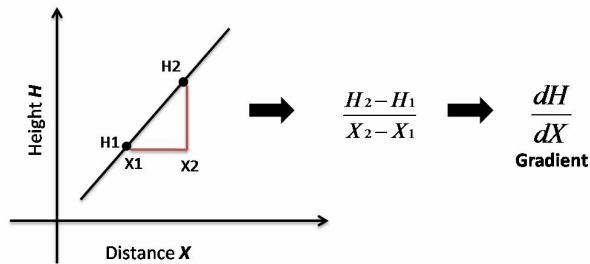


Figure 6.5: The explanation of gradient

If we put our numbers in the preceding formula, we get a difference of heights that is, 500 divided by the difference of distance 200. So, the slope will be 2.5, which is a positive gradient. In case of downhill with the same changes, the gradient will be -2.5, which is a negative gradient.

So, why are we discussing the gradient? Because gradients can also be applied on any differential function that can be used to find

the minimum value of that function by estimating the direction and the rate of change in the function with respect to any input variable. I know this was too technical. Let's understand this with an example.

We will use our weak learner's error as the **differentiable** function. Suppose at the n^{th} classifier, we get an output p_n with an error on e_n . Now, our task is to create the next weak learner ($n+1$) so that it can reduce the error generated by the current classifier. For this, we should know whether the error is positive (a positive gradient) or negative (a negative gradient). So, we will introduce our new weak learner ($n+1$) and evaluate its prediction with the previous output (prediction of n^{th}) to know whether we have reduced the error or not and accordingly, we will do it until we reach up to the minimum of the error value.

So, basically, the gradient of our error function will tell us the direction, where we can achieve the minimum value of our error

function and usually, it should be a negative value (for going downhill). This can be understood by the following figure:

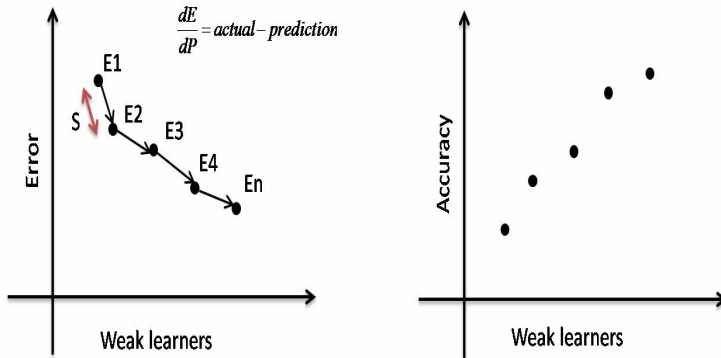


Figure 6.6: The explanation of the gradient descent algorithm

So, let's understand the preceding figure in the following steps:

- Put the first weak learner and get a prediction of it.
- Calculate the gradient of error function E by subtracting the predicted and actual output.
- Step in the direction of the greatest descent (the negative gradient) with step size S , which means:

$$f_n = f_{n-1} - S * E$$

- In the preceding step, S is the multiplier to the residual (error) between the previous output (*for the first learner, it will be the actual output*) and the predicted output. It will help us to control the speed of convergence.
- We will keep adding the weak learners and train them on the residual (multiplied by step size) of the previous prediction, to move in the direction of the gradient until we reach the minimum value (close to zero) of the error function.

The algorithm got its name GBM due to the gradient descent algorithm, which is a very popular algorithm to train neural networks and other machine learning classifiers.

There is one important point that you should remember. As we can use any differential error function to calculate the loss, the algorithm gives us the leverage of using

different kinds of error functions, such as the **sum of squared error**, **logarithmic loss**, or **entropy-based losses**.

Now, let's take a look at the complete picture of a GBM:

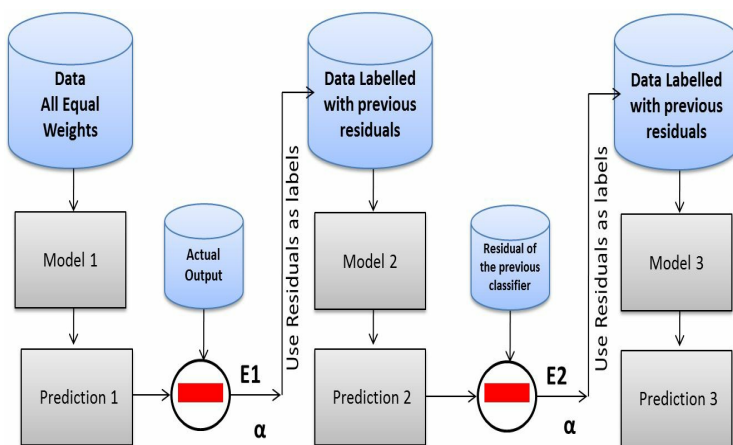


Figure 6.7: A GBM (training version)

As you can see the complete flow of training of a GBM, we can summarize the preceding process as follows:

- Train the first weak classifier with the original output labels

- Choose an error between the predicted values and the actual output as labels for the next classifier
- Use a multiplier α to control the rate of convergence
- Keep adding the weak learners until the function reaches the convergence

At the time of deployment, the following approach will be taken:

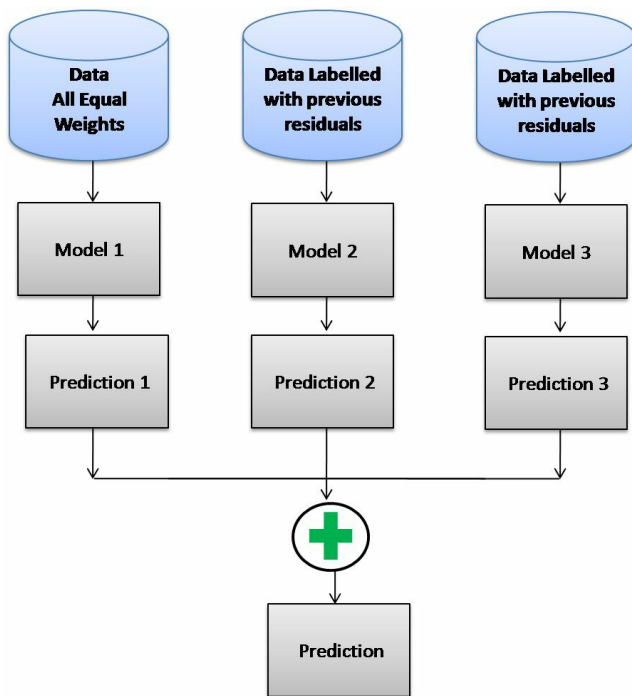


Figure 6.8: The GBM (testing version)

We have discussed enough theory here, and it's time to jump into the practical implementation part of GBM. As you have seen, we can use any kind of classifier algorithm to boost and get results out of it, but in the world of data science, people always prefer decision trees as the first choice of the weak learners because it is easy to

understand their architecture. We will also do the same as you have already learned many things about them and we know how to implement them very well. But, GBM doesn't use classification trees for making predictions, which makes our task a bit difficult as we have not seen any kind of regression tree yet, so before jumping to the implementation of the GBM algorithm, we will first develop our very own regression tree.

<h1>Regression trees

We have already discussed the CART algorithm in [Chapter 3](#), *Random Forest*, where we developed decision trees for the random forest as well as for standalone tree-based classification. Now, we will see how to implement a regression tree and how we can use them to learn almost any kind of linear or nonlinear function.

What is the difference?

The basic and the most important difference between a classification and a regression tree is their output value. While for classification trees, the output values are always discrete or categorical in nature, the output of the regression trees is always a continuous value. One of the most important differences between them is an evaluation of the splits. While we used Gini index and Shannon entropy to evaluate splits in case of classification trees, regression trees use some loss functions to do this; the most popular loss is the sum of squares. There are many more differences between the two; we will see them during our progress. The following is the figure that describes the use cases of different tree algorithms:

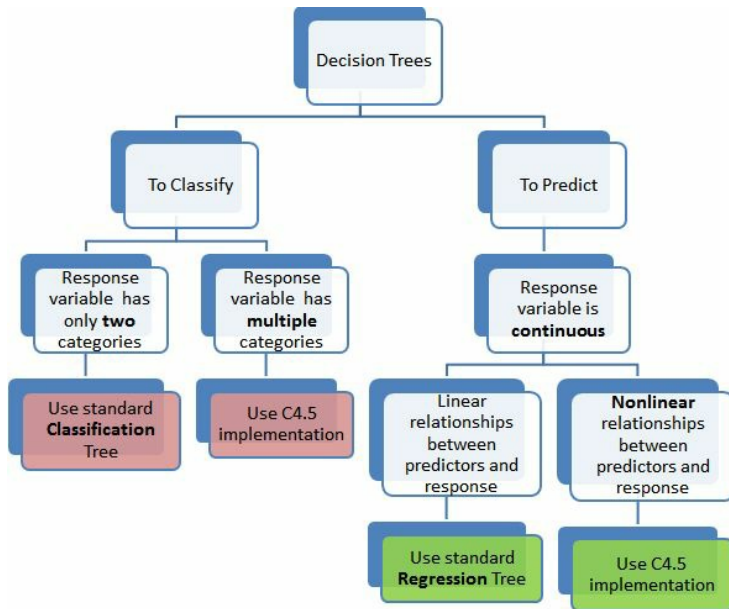


Figure 6.9: Decision tree algorithms

One of the decision tree algorithm C4.5 is just the customization of the original CART algorithm that we have seen earlier (in random forest); the basic difference in any kind of tree-based algorithms is based on how they choose the node values for different branches.

There is one important aspect of regression trees; we can use a regression tree as a

classifier or a regression predict, or both; the same case is not applicable for the classification tree.

As we know, the tree structure of a classification problem looks as follows:

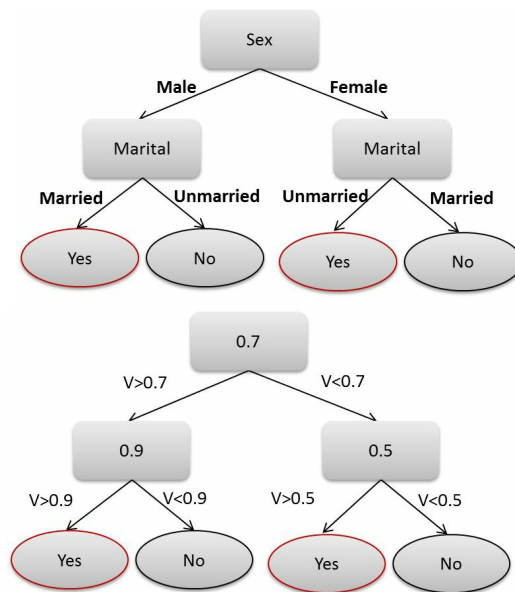


Figure 6.10: Decision tree on categorical and numerical data

Node values may be a categorical or continuous number; similarly, branches may have categorical or continuous values, but the

leaf node will always have discrete values.

Regression tree will look as follows:

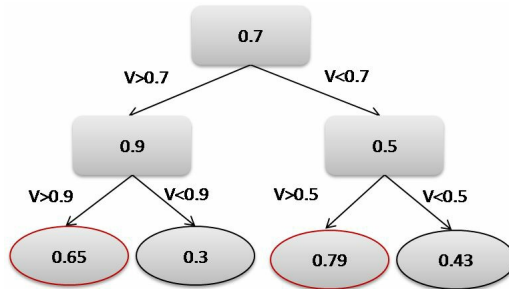


Figure 6.11: An example of a regression tree

We can use the preceding tree as a classification by adding a transfer function at the end:

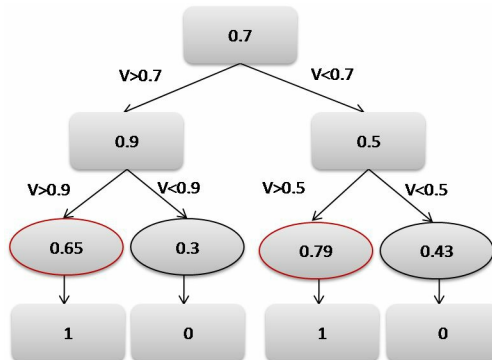


Figure 6.12: An example of a regression tree as a classifier

Here, we have just round the fractional values, so the values more than **0.5** go to class **1** and less than or equal to **0.5** will goes to class **0**. Other options to convert continuous data into some meaningful class is to convert the values into probabilities (softmax) or apply a sigmoid function, which can convert the values in the range of **0** to **1**. We will look at these kinds of transfer functions in the future chapters.

Now, the question is how do we build a regression tree? To answer this question, let's look at the fundamental elements required to build the tree:

- We should have some metric on the basis of which we can decide what should be the value of our node (including the root node)
- We must know where to split a tree branch
- We must know some stopping criteria for our tree growing process or it may grow for an infinite time

So, what we have done for these in the past is as follows:

- We have used the Gini index as the decision metric for choosing the node value
- We have split the branches with the lowest Gini index node value
- We have used various stopping criteria, such as depth of the tree and length of the splits to stop the tree growth

Guess what; we will do the same for the regression tree implementation, too! Yes, except one change that is the metric for choosing the node value. As earlier, we have used the Gini index for the task; now, we will use the sum of squared error for that, except that one change to our algorithm will remain almost the same as the classification tree algorithm.

The algorithm will consist of the following steps:

1. Create split
2. Node selection
3. Tree building

So, what are we waiting for? Let's start understanding each section.

Create split

The create split procedure will be the same as for the classification tree, which we have already seen in [Chapter 3, Random Forest](#). It will have two steps:

1. Choose an arbitrary value from the attribute
2. Use this value as a threshold, and create two groups from the attribute values such that one group will have values less than the threshold and the other group will have values greater than or equal to the threshold

The code for this will be similar as earlier we have seen in *Decision tree bagging* in [Chapter 3, Random Forest](#):

```
#Create splits to test for node values
def createSplit(attribute,threshold,dataset):

    #Initialize two lists to store the sub sets
    lesser, greater = list(),list()
```

```
#Loop through the attribute values and
create sub set out of it
for values in dataset:

    #Apply threshold
    if values[attribute]<=threshold:
        lesser.append(values)
    else:
        greater.append(values)
return lesser,greater
```

Once we have split the data into two groups, it's time to check whether the `attribute` value we have chosen creates a perfect split or not.

Node selection

To test the splits, we will use the sum of squared error between the mean of the output values and the output value of each instance. But why do we need to find the error between the mean and the output values? This is because this error can tell us how uniform the group's target values are; this is similar to finding out the variance of the output values, so if the variance is low, it means that the values are very similar to each other and if the variance is high, the values are distinct. Let's understand this in the following example.

Suppose we have created two groups with a threshold value; the first group looks as follows:

A1	A2	Out

0.30	0.50	0.20
0.35	0.45	0.23
0.42	0.53	0.25

Table 6.2: Continuous output variable with similar range of values

Additionally, the second group looks as follows:

A1	A2	Out
0.55	0.67	0.60
0.57	0.63	0.65
0.51	0.68	0.67

Table 6.3: Continuous output variable with similar range of values

Our loss function is as follows:

$$E = \sum_{i=0}^M (y_i - \bar{y})$$

Where E is an error, y_i is the prediction of the i^{th} instance and \bar{y} is the mean of all the y (target) in the group. So, on applying the preceding loss function to our groups, we will get the error for group one as follows:

$$Error = (0.20 - 0.22) * 2 + (0.23 - 0.22) * 2 + (0.25 - 0.22) * 2$$

The mean value is 0.22 for group one and the error is 0.0013 , which is very low; similarly, for group two, it is 0.0026 with the mean value of 0.64 . For getting the total error for the split, we will add them. This will result in 0.0039 , which shows the minimum variance in the groups, and this suggests that the current value is a good split point. Let's see a case where the split has different values:

Group one is as follows:

A1	A2	Out
0.30	0.50	0.20
0.35	0.65	0.43
0.20	0.53	0.35

Table 6.4: Continuous output variable with variable range of values

Group two is as follows:

A1	A2	Out
0.55	0.17	0.25
0.57	0.63	0.65

0.51	0.49	0.42
------	------	------

Table 6.5: Continuous output variable with variable range of values

In the preceding case, the error for group one is 0.027 and for group two is 0.080; the combined error for the split is 0.107, which is quite higher in comparison to 0.0039. So, the preceding exercise shows us that this metric can help us find out the best split in the dataset.

Let's put it into our code with the function name `SquaredError()`:

```
def SquaredError(groups):
    #Initialize the variable for SSE
    sse = 0.0

    #Iterate for both the groups
    for group in groups:
        size = len(group)

        #If length is 0 continue for the next
        group
        if size == 0:
            continue

    #Take all the class values into a list
```

```

        class_values = [row[-1] for row in
group]

        #Calculate SSE for the group
        sse += np.sum((class_values-
np.mean(class_values))**2)
    return sse

```

Now, we have the metric to evaluate the splits; it's time to add a function similar to `getNode()` from [Chapter 3, Random Forest](#)), which returns us the `node` value with the best split; the function goes as follows:

```

#Function to get new node
def getNode(dataset):

    #initialize variables to store error score,
attribute index and split
    groups
    winnerAttribute = sys.maxsize
    attributeValue = sys.maxsize
    errorScore = sys.maxsize
    leftGroup = None

    #Run loop to access each attribute and
attribute values
    for index in range(len(dataset[0])-1):
        for row in dataset:

            #Get split for the attribute value
            groups = createSplit(index,
row[index], dataset)

            #Calculate SSE for the group
            sse = SquaredError(groups)

            #If SSE is less than previous

```

```

attribute's SSE return attribute value
    as Node
    if sse < errorScore:
        winnerAttribute,
attributeValue, errorScore, leftGroup = index,
row[index], sse, groups

    #Once done create a dictionary for node
    node =

    {'attribute':winnerAttribute, 'value':attributeVa

    return node

```

The preceding function works exactly the same as our `getNode` function previously did, except for one change. There we called the Gini index for split evaluation; here, we replace that function with `SquaredError`.

So, we are good to go now for adding the nodes to our regression tree, but before that, we need to make one more change, that is, modifying the function `terminalNode()`. This function is responsible for the values of the leaf node in the tree. In the previous case, we set the class that occurs maximum in the group as the leaf value of the branch, but now, as we don't have the discrete classes, we will replace the maximum occurrence with

the mean value of the group's target:

```
def terminalNodeReg(group):  
    #Get all the target labels into the List  
    class_values = [row[-1] for row in group]  
  
    #Return the Mean value of the list  
    return np.mean(class_values)
```

So, now we have all of the ingredients to build our first regression tree; as always, we will understand the tree with the help of an example.

Build tree

We will use a function fitting example to understand the concept of the regression tree, and we will also see how a regression tree easily fits on a highly nonlinear dataset.

Let's choose the function of sine wave to fit with our tree as it covers our two important criteria of creating a regression tree that is, continuous values for the given time axes and it has a nonlinear function fitting problem, too. Let's start by creating a sine wave for 25 sample points:

```
#Create a Sine wave for demonstration of non-  
linearity  
#Set the number of samples  
N = 25  
  
#Create time values  
ix = np.arange(N)  
  
#Create the sine wave using the formula  
sin(2*pi*f)  
signal = np.sin(2*np.pi*ix/float(N/2))  
  
#Combine both time and amplitude  
dataset = range(0,N)
```

```
| dataset = np.c_[ix, signal]
```

Let's plot the signal using matplotlib and see how it looks:

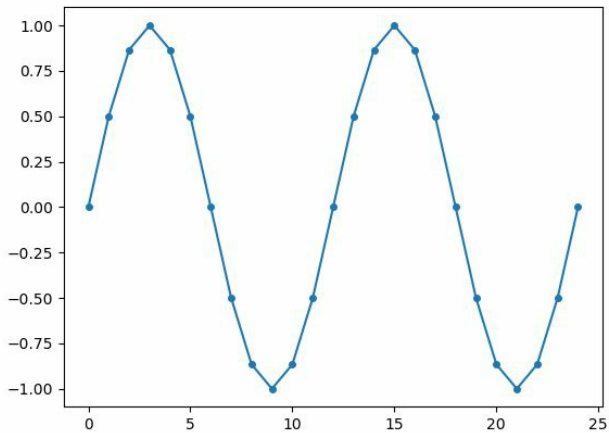


Figure 6.13: The sinusoidal function

So, you can see our sine wave with 25 time samples and the amplitude range is from -1 to 1; let's start solving the problem by adding a function for building the tree:

```
| # Build a decision tree
| def build_tree(train, max_depth, min_size):
|     #Add the root node to the tree
```



```

    root = getNode(train)

    #Start building the from the root's
    branches tree
    buildTreeReg(root, max_depth, min_size, 1)
    return root

```

The preceding function will add a `root` node to the tree and then, it will start building the tree through the branches of the root. Let's see our root node:

```

{'attribute': 0, 'value': 18.0}

    SSE for the attribute 1.00's value 0.00 is
    12.000
    SSE for the attribute 1.00's value 1.00 is
    11.864
    SSE for the attribute 1.00's value 2.00 is
    11.293
    .
    .
    .
    SSE for the attribute 1.00's value 17.00 is
    9.236
    SSE for the attribute 1.00's value 18.00 is
    8.946
    SSE for the attribute 1.00's value 19.00 is
    9.388
    SSE for the attribute 1.00's value 19.00 is
    9.388
    SSE for the attribute 1.00's value 20.00 is
    10.334
    SSE for the attribute 1.00's value 21.00 is
    11.293
    SSE for the attribute 1.00's value 22.00 is
    11.864
    SSE for the attribute 1.00's value 23.00 is
    12.000

```

SSE for the attribute 1.00's value 24.00 is
12.000

As we have only one attribute, we got our first root at time sample 18.0 with the error of 8.946, which is the lowest amongst all others. So, here we will have two groups—one group will have values less than 18 and the other will have values greater than 18.0. The next process involves a similar approach. We will recursively call the function for adding the nodes under branches created by the root node. For doing this, we will add the `buildTreeReg()` function to our code:

```
# Create child splits for a node or make
terminal
def buildTreeReg(node, max_depth, min_size,
depth):

    #Lets get groups information first.
    left, right = node['groups']
    del(node['groups'])

    # check if there are any element in the
    left and right group
    if not left or not right:

        #If there is no element in the groups
        call terminal Node
        combined = left+right
        node['left'] =
terminalNodeReg(combined)
        node['right']=
```

```

terminalNodeReg(combined)
    return

    # check if we have reached to maximum depth
    if depth >= max_depth:
        node['left']=terminalNodeReg(left)
        node['right'] = terminalNodeReg(right)
        return

    # if all okay let's start building tree for
left side nodes
    # if minimum instances are done by the node
stop further build
    if len(left) <= min_size:
        node['left'] = terminalNodeReg(left)

    else:

        #Create new node under left side of the
tree
        node['left'] = getNode(left)

        #append node under the tree and
increase depth by one.
        buildTreeReg(node['left'], max_depth,
min_size, depth+1)
                                                    #recursion
will take place in here

    # Similar procedure for the right side
nodes
    if len(right) <= min_size:
        node['right'] = terminalNodeReg(right)

    else:
        node['right'] = getNode(right)
        buildTreeReg(node['right'], max_depth,
min_size, depth+1)

```

The preceding function is self-explanatory;
let's see the summary of the preceding

function:

- It will take the previous node as the input
- Extract the groups created with the attribute value
- If there is no group under the node value, terminate the tree by calling the terminal node
- If the tree reaches the maximum depth, terminate the tree. If not, go to the next step
- Check the minimum number of instances in the group; if less than `min_size`, terminate the tree, if not, go to the node in the branch
- Recursively call the preceding steps until we reach one of the thresholds from `max_depth` OR `min_size`

So, after doing the preceding process, our tree looks as follows:

```
#Maximum Depth
max_depth = 3

#Minimum number of instances to process in the
```

```

group
min_size = 1

#Start building the tree
tree_rg = build_tree(dataset,max_depth,
min_size)

#Print the Tree
pprint.pprint(tree_rg)

{'attribute': 0,
 'left': {'attribute': 0,
          'left': {'attribute': 0,
                   'left': 0.53315011536698242,
                   'right':
-0.62200846792814624,
                   'value': 6.0},
          'right': {'attribute': 0,
                   'left':
0.74641016151377548,
                   'right':
3.6739403974420594e-16,
                   'value': 17.0},
          'value': 12.0},
 'right': {'attribute': 0,
          'left': {'attribute': 0,
                   'left':
-0.49999999999999917,
                   'right':
-0.80801270189221985,
                   'value': 19.0},
          'right': -4.8985871965894128e-16,
          'value': 23.0},
 'value': 18.0}

```

Let's see how this tree fits our input function for depth three:

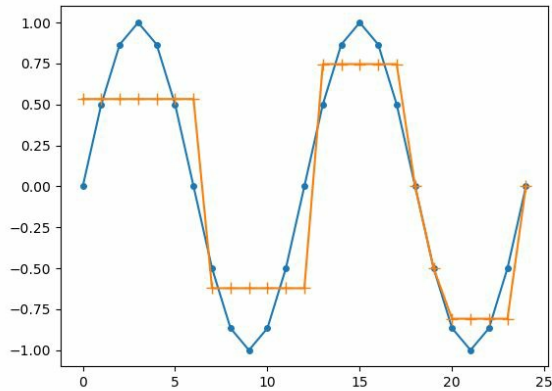


Figure 6.14: Under fitting by a shallow decision tree

As you can see for the depth three, our tree is not fitting the function well. There are errors for almost each instance in the dataset. We will increase the depth of our tree until we get the minimum error between our input and prediction. After putting depth of nine, we got a perfect fit for our function which looks as follows:

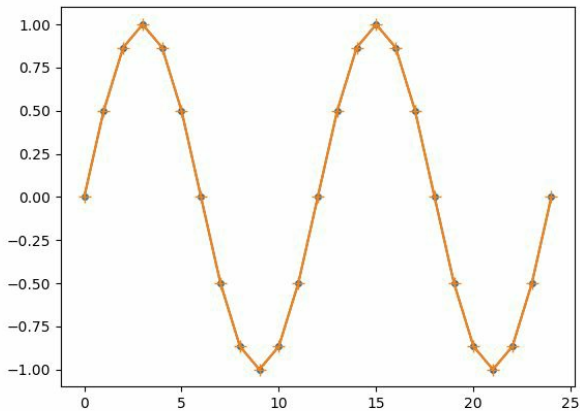


Figure 6.15: Perfect fitting by decision tree

So, you can see how we got the perfect fit at depth nine previously. We can't print the tree for depth nine as it will take a lot of space, which is not a good idea.

Regression tree as a classifier

We have seen how to develop a regression tree from scratch; now, it's time to use it for a practical dataset. We will use our regression tree for the classification of the data. We will apply the regression tree for the data of a breast cancer study. We have used a similar dataset in [Chapter 3](#), *Random Forest*, where we applied a random forest algorithm to predict the output.

We will use the data of a breast cancer study which is available online at the location: <https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Original%29>. Here is the summary of the dataset:

```
Data set Name: Breast Cancer Wisconsin  
(Original) Data Set  
Number of Instances: 699  
Attributes characteristics: Integer  
Number of attributes: 10
```


| Number of classes: 2

Attribute information:

1. Sample code number: id number
2. Clump Thickness: 1 - 10
3. Uniformity of Cell Size: 1 - 10
4. Uniformity of Cell Shape: 1 - 10
5. Marginal Adhesion: 1 - 10
6. Single Epithelial Cell Size: 1 - 10
7. Bare Nuclei: 1 - 10
8. Bland Chromatin: 1 - 10
9. Normal Nucleoli: 1 - 10
10. Mitoses: 1 - 10
11. Class: (2 for benign, 4 for malignant)

We will use the following steps to build our trees:

1. Load the CSV file into the program
2. Convert string values into numerical data
3. Create training and testing set to evaluate the model
4. Build the tree on train data and test it on the training set

We will use helping functions from [Chapter 3, Random Forest](#) for the preceding steps. I will provide the full code listing at the end of

the chapter. So, let's start the implementation:

```
filename = 'breast_cancer_data.csv'
dataset = readCsv(filename)

# Convert attributes to numerical type
for i in range(0, len(dataset[0])):
    convert_column_to_float(dataset, i)

# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)

#Now remove index column from the data set
dataset_new = []
for row in dataset:
    dataset_new.append([row[i] for i in
range(1,len(row))])

#Get training and testing data split
training,testing =
getTrainTestData(dataset_new, 0.8)

#We will going to build our tree for maximum
depth of 11 and with minimum instances for a
node of 5
tree = build_tree(training,11,5)

#Evaluation of the function on training data
pre = []
act = []

for row in training:
    prediction = np.round(rg.predict(tree,
row))
    pre.append(prediction)
    actual = act.append(row[-1])
acc = accuracy_metric(act, pre)
print("Training Accuracy of Model is:
%.2f"%acc)

#Evaluation of the function on testing data
```

```

pre = []
act = []

for row in testing:
    prediction = np.round(rg.predict(tree,
row))
    pre.append(prediction)
    actual = act.append(row[-1])
    acc = accuracy_metric(act, pre)

print("Testing Accuracy of Model is: %.2f"%acc)

```

Let's see what we have got after execution:

```

Training Accuracy of Model is: 98.39
Testing Accuracy of Model is: 97.14

```

Do you know what accuracy we were getting in the CART as a classification mode in [Chapter 3, Random Forest](#)? Well, for your reference, here are the numbers for depth of 11 and minimum size 5:

```

Training Accuracy of Model is: 95.91
Testing Accuracy of Model is: 95.28

```

So, you can see that we have improved our training and testing accuracies with a good margin. I want to present you a case of high variance (overfitting) of our regression tree on the same data set. Let's put the same depth of 11, but the minimum size for a node is 1.

Can you guess the output?

```
| Training Accuracy of Model is: 100.00  
| Testing Accuracy of Model is: 96.43
```

Can you see what we have got? We have got a training accuracy of 100, while the testing accuracy of the model is 96.43, which clearly shows the overfitting of our model, and it is a big drawback of the regression tree. To overcome this overfitting, we can use ensembles of the tree in the random forest algorithm. The discussion of random forest here is out of context, so I will encourage you guys to use the regression tree's random forest for the same data set to see what kind of accuracy we can get.

GBM implementation

We have implemented a regression tree successfully, we have tested it on a real dataset, and got a very good accuracy. Now, this is the time to build a boosting ensemble of the regression tree to turn it into a very powerful nonlinear classifier. As we already know, the basic ingredient of GBM is a classifier that can predict continuous values at the output and also have a differential error function so that we can apply gradient decent on it. Regression trees are the perfect match for the task, so let's try to implement a GBM with the use of our regression trees.

Algorithm

As we have already seen the theoretical procedure of the GBM, we can summarize the algorithm in the following points:

- Build a decision tree on the initial dataset and find the error between the actual output and the prediction
- Use this error (or residual in literature) as the new output values for the dataset's instances
- Build a new tree on the dataset with residual as the label of the instances and train the tree to reconstruct the error created by the previous tree
- Add trees to the process until we minimize the error between the previous output and the current prediction up to the desired level

I think that the procedure is quite clear. To implement the algorithm in our code, we will

require some more helper functions to our regression tree code.

So, first we will add the function to calculate error values between the prediction and the actual output; let's do it by adding a function `getResidual()`:

```
def getResidual(actual,pred):  
    #Create an empty list to store individual  
error of the instances  
    residual = []  
  
    # Run a loop to get difference between  
output and prediction of each instance  
    for i in range(len(actual)):  
  
        #Get the difference and add the  
difference to the list of residuals  
        diff = (actual[i]-pred[i])  
        residual.append(diff)  
  
    #Calculate the Sum of squared error between  
output and prediction  
    mse = np.sum(np.array(residual)**2)  
    return residual,mse
```

The preceding function will create a list of differences between each instance as well as the calculation of the total error made by the tree. We will use this error as the new label for the instances of the dataset.

Now, we are ready for the gradient boosting algorithm; let's add function `GradientBoost()`. The following code block will do the process we discussed previously:

```
def
GradientBoost(dataset,depth,mincount,iterations)

    dataset = np.array(dataset)

    #Create a list to add weak
learners(decision stumps)
    weaks = []

    #Lets run the loop for number of
iteration(number of classifiers)
    for itr in range(iterations):

        #Create decision tree from the data-set
        ds = build_tree(dataset,depth,mincount)

        #Create a list to store the predictions
of the decision stump
        pred=[]

        #Create a list to store actual outputs
        actual = []

        #Let's predict output for each instance
in the data set
        for row in dataset:
            actual.append(row[-1])
            pred.append(predict(ds, row))
        #Here we will find out difference
between predicted and actual output
        residuals,error = getResidual(actual,
pred)

        #Print the error status
```



```

        print("\nClassifier %i error is %.5f"%
(itr,error))

        #Check for the convergence
        if error<=0.00001:
            break

        #Replace the previous labels with the
current differences(Residuals)
        dataset[:, -1] = residuals

        #Append the weak learner to the list
        weaks.append(ds)

    return weaks

```

As you can see, the preceding code is pretty simple to understand. The input of the algorithm will be `dataset`, the maximum `depth` of each tree, the minimum size of a group for a node, and the maximum number of the trees to be added to our GBM model. As this is a sample implementation, we keep complexity away from it. If you look closely, we have followed exactly the same procedure to build our GBM as we discussed earlier.

So, first let's define our problem statement and then, we will see how our GBM performs on the nonlinear dataset.

We will create a sine wave with the 256 samples and try to fit our GBM on this:

```
#Create a Sine wave for demonstration of non-
linearity
#Set the number of samples
N = 256

#Create time value
ix = np.arange(N)

#Create the sine wave using the formula
sin(2*pi*f)
signal = np.sin(2*np.pi*ix/float(N/2))

#Combine both time and amplitude
dataset = range(0,N)
dataset = np.c_[ix,signal]
dataset_ = dataset.copy()
```

We will create our GBM with a different number of trees to evaluate the performance of the ensemble. For simplicity, we will choose a tree with the depth one, that is, decision stump. Let's see how our algorithm works:

```
error = []
max_number_of_trees=50
interval = 10

for ntree in
range(1,max_number_of_trees,interval):
    weaks = rg.GradientBoost(dataset,1,1,ntree)
    preiction=[]
    actual = []
```

```

        #Run a loop to extract each instance from
the data set
        for row in dataset_:

            #Create a list to store predictions
from different classifier for the
test instance
            preds = []

            #Feed the instance to different
classifiers
            for i in range(len(weak)):
                #Multiply the predicted output with
the alpha value of the classifier
                p = rg.predict(weak[i], row)
                #Add the weighted prediction to the
list
                preds.append(p)

            #Sum up output of all the classifiers
and take their sign as the prediction
            final = (sum(preds))

            #Append the final output to the
prediction list and actual output to the actual
list
            prediction.append(final)
            actual.append(row[-1])

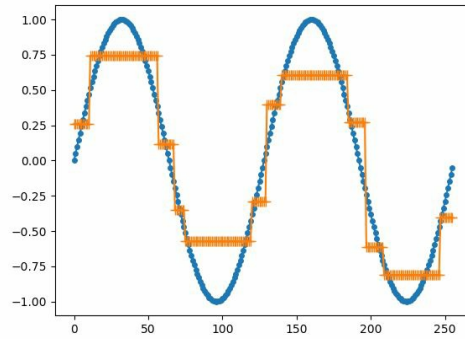
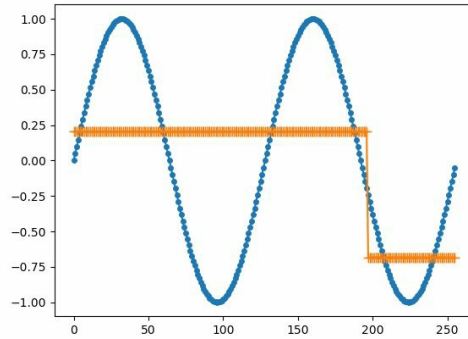
            #Append the error of the current
configuration
            _,mse = rg.getResidual(actual, prediction)
            error.append(mse)

#Lets Plot the error in each configuration
plt.figure()
plt.plot(range(1,max_number_of_trees,interval),e
plt.show()

```

Let's see the execution summary of the

preceding code:



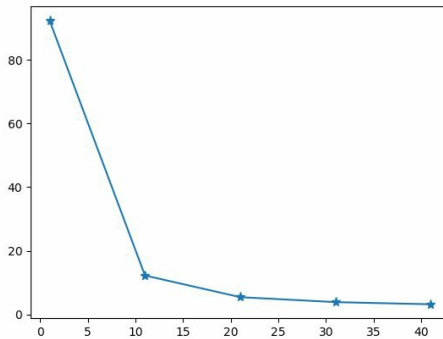
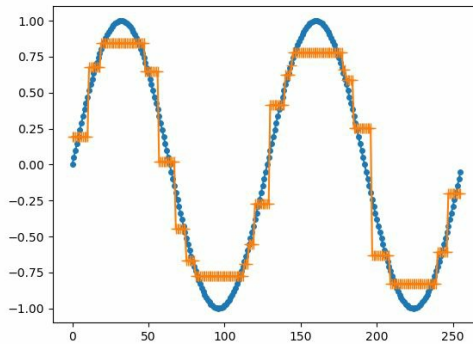


Figure 6.16: From left to right a,b,c,d—showing performance of GBM for different number of trees

As you can see in figure a, our prediction is underfitting the input function by a huge margin. When we add 10 more decision stumps (in figure b), our system starts to

reconstruct errors created by the previous classifiers and reducing it, figure *b* shows the output after appending 20 decision stumps.

Figure *c* shows a recreation of the function after adding 30 decision stumps, and you can see improvement in the reconstruction of a signal from our system. This reconstruction is slow because we used a very shallow decision tree of depth one only, but still, you can see the reduction in the error from figure *d*. If we keep adding the trees into the system, our accuracy will increase accordingly. Now, if we increase the depth to five, we will get the following output by adding just 26 trees in the ensemble:

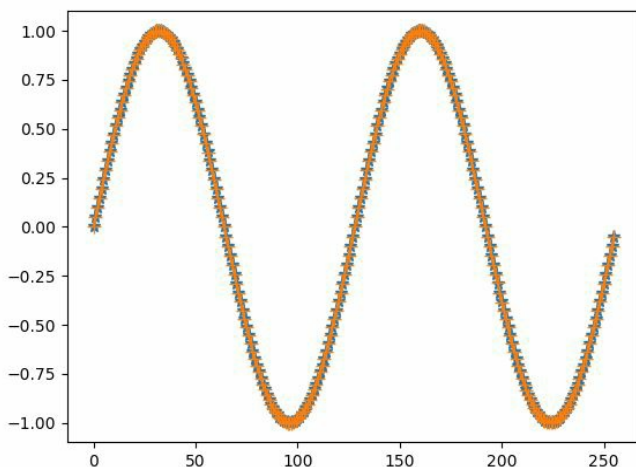


Figure 6.17: Perfect fit by GBM

You can see that we have a perfect fit in very less iteration with the trees of more depth, so here, we have the following parameters to control the output behavior of the GBM in our implemented code.

Improvements to basic gradient boosting

Gradient boosting is a greedy algorithm and can overfit a training dataset quickly.

It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting.

In this section, we will look at four enhancements to basic gradient boosting:

- Tree constraints
- Shrinkage
- Random sampling
- Penalized learning

Tree constraints

It is important that the weak learners have skill but remain weak. There are a number of ways in which the trees can be constrained. A good general heuristic is that the more constrained tree creation is, the more trees you will need in the model, and the reverse, where the fewer constrained individual trees, the fewer trees that will be required.

The following are some constraints that can be imposed on the construction of decision trees:

- **Number of trees:** Generally, adding more trees to the model can be very slow to overfit. The advice is to keep adding trees until no further improvement is observed.
- **Tree depth:** Deeper trees are more complex trees, and shorter trees are preferred. Generally, better results are

seen with 4-8 levels.

- **Number of nodes or number of leaves:** Like depth, this can constrain the size of the tree, but is not constrained to a symmetrical structure if other constraints are used.
- **Number of observations per split:** This imposes a minimum constraint on the amount of training data at a training node before a split can be considered.
- **Minimum improvement to loss:** This is a constraint on the improvement of any split added to a tree.

Weighted updates

The predictions of each tree are added together sequentially. The contribution of each tree to this sum can be weighted to slow down the learning by the algorithm. This weighting is called a shrinkage or a learning rate.

The effect is that learning is slowed down, in turn requiring more trees to be added to the model, in turn taking longer to train, providing a configuration trade-off between the number of trees and the learning rate.

Stochastic gradient boosting

A big insight into bagging ensembles and the random forest was allowing trees to be greedily created from the subsamples of the training dataset. This same benefit can be used to reduce the correlation between the trees in the sequence in the gradient boosting models. This variation of boosting is called stochastic gradient boosting. A few variants of stochastic boosting that can be used are as follows:

- Subsample rows before creating each tree
- Subsample columns before creating each tree
- Subsample columns before considering each split

Generally, aggressive subsampling, such as

selecting only 50% of the data has shown to be beneficial.

Penalized gradient boosting

Additional constraints can be imposed on the parameterized trees in addition to their structure. Classical decision trees, such as CART, are not used as weak learners, instead, a modified form called a regression tree is used, which has numeric values in the leaf nodes (also called terminal nodes). The values in the leaves of the trees can be called weights in some literature.

As such, the leaf weight values of the trees can be regularized using popular regularization functions, such as:

- L1 regularization of weights
- L2 regularization of weights

Summary

So, this was the basis of GBMs. We started from the basics concepts of gradient boosting. We have seen the theoretical explanation of the technique, which shows great promises to improve the accuracy of an ensemble system of weak learners. We also saw what regression is and how the regression tree works. Then, we implemented working regression tree by ourselves and used it to fit a sinusoidal function, which is nonlinear in nature. After this, we have seen how to use a regression tree as a classifier by just rounding the predicted value; by the way, this is not a good way to do it, but we have achieved a significant improvement in the output accuracy in comparison with the classification tree. Then, we implemented the theoretical concept of the GBM into the practical code and saw how it can reduce the error of prediction by increasing the number

of trees in the ensemble.

So, this is not the end of gradient boosting. We have just implemented a basic version of the GBM algorithm to understand the concepts behind it. In the next chapter, we will see a more robust, scalable, and widely used version of the gradient boosted machine —XGBoost. We will see how to use XGBoost and how to tune its parameter to get the maximum prediction accuracy out of it. See you in the next chapter; till then keep implementing the code discussed in this chapter. In the following section, I have shared the full code listing of regression trees and GBM.

XGBoost – eXtreme Gradient Boosting

In the previous chapter, we learned how a gradient boosting machine can help us learn complex functions. We implemented a **Gradient Boosting Machine (GBM)** using regression trees as weak learners. In this chapter, we will talk about the XGBoost library, which is based on the same GBM principle we implemented earlier. It is a third-party library that has been written in Python and R. Both are very famous programming languages used to implement data analytics solutions. So first, we are going to understand why we should move towards a third-party library when we can implement algorithm concepts on our own. The answer is quite simple—performance optimization!!

There is no doubt that we can implement any algorithm concept in any programming

language, but to implement an algorithm in a very efficient way so that we can train a system in very less time, as well as deploy these systems in real time is very difficult. This requires an in-depth knowledge of various data structures and strong hands-on programming. We can't spend out time learning all kinds of programming concepts before we implement a solution. For this purpose, we use third-party libraries, which are written by developers who have excellent programming skills. A similar kind of library is XGBoost, which is very fast to train and very efficient at using available resources so that we can get real-time performance from the implemented framework.

XGBoost – supervised learning

eXtreme Gradient Boosting (XGBoost). It is an implementation of GBM created by Tianqi Chen, and now it has contributions from many developers. It belongs to a broader collection of tools under the umbrella of the **Distributed Machine Learning Community (DMLC)**, who are also the creators of the popular `mxnet` deep learning library.

XGBoost is a software library that you can download and install on your machine; then you can access it from a variety of interfaces. Specifically, XGBoost supports the following main interfaces:

- **Command-Line Interface (CLI)**
- C++ (the language in which the library is written)

- The Python interface as well as a model in scikit-learn
- The R interface as well as a model in the `caret` package
- Julia
- Java and JVM languages such as Scala and platforms such as Hadoop

XGBoost is used for supervised learning problems, where we use the training data (with multiple features) x_i to predict a target variable y_i . Before we dive into trees, let's start by reviewing the basic elements in supervised learning.

Models and parameters

The model in supervised learning usually refers to the mathematical structure for making the prediction y_i given x_i . For example, a common model is a linear model, where the prediction is given by:

$$\bar{y} = \sum_j^M W * X$$

It's a linear combination of weighted input features. The prediction value can have different interpretations depending on the task, that is, regression or classification. For example, it can be logistically transformed to get the probability of a positive class in logistic regression, and it can also be used as a ranking score when we want to rank our outputs.

The parameters are the undetermined part that we need to learn from the data. In linear regression problems, the parameters are the coefficients W . Usually we will use W to denote the parameters (there are many parameters in a model; our definition here is sloppy).

Objective function – training loss + regularization

Based on different understandings of y_i , we can have different problems such as regression, classification, ordering, and so on. We need to find a way to find the best parameters given the training data. In order to do so, we need to define a so-called **objective function** to measure the performance of the model given a certain set of parameters.

A very important fact about objective functions is that they must always contain two parts, training loss and regularization:

$$Obj(\theta) = L(W) + \Omega(\theta)$$

L is the training loss function and Ω is the regularization term. The training loss

measures how predictive our model is on training data. For example, a commonly used training loss is mean squared error:

$$L(W) = \sum_i (y_i - \bar{y}_i)^2$$

Another commonly used loss function is logistic loss for logistic regression:

$$L(W) = \sum_i [y_i \ln(1 + e^{-\bar{y}_i}) + (1 - y_i) \ln(1 + e^{\bar{y}_i})]$$

The **regularization term** is what people usually forget to add. It controls the complexity of the model, which helps us to avoid overfitting. This sounds a bit abstract, so let's consider the following problem in the following picture. You are asked to fit a visual step function given the input data points in the top-left corner of the image. Which solution among the three do you think is the best fit?

The correct answer is marked in red. Please consider whether this seems a reasonable fit

visually. The general principle is that we want both a simple and predictive model. The trade-off between the two is also referred to as bias-variance trade-off in machine learning:

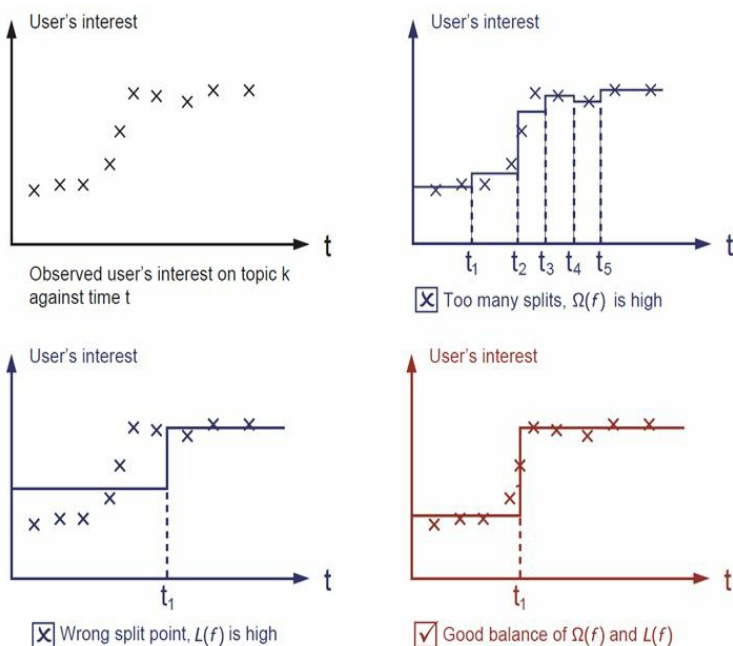


Figure 7.1: Optimal decision boundaries

Why introduce the general principle?

The elements introduced so far have been the basic elements of supervised learning, and they are naturally the building blocks of machine learning toolkits. For example, you should be able to describe the differences and commonalities between boosted trees and random forests. Understanding the process in a formalized way also helps us to understand the objective that we are learning and the reason behind the heuristics such as pruning and smoothing.

So we want to design a system that can incorporate the aforementioned concepts to generate predictions without any bias-variance trade-off; gradient boosting is the technique that can incorporate all of the aforementioned changes. If you remember, we discussed these kinds of changes at the

end of the previous chapter, where I suggested some variations to create an efficient GBM.

XGBoost is a well-written library that allows the user to control these parameters; let's see how we can use these parameter variations. In the next section, we will talk about the features offered by the XGBoost library.

XGBoost features

The library is laser focused on computational speed and model performance; as such, there are few frills. Nevertheless, it does offer a number of advanced features.

Model features

The implementation of the model supports the features of the scikit-learn and R implementations, with new additions such as regularization. Three main forms of gradient boosting are supported:

- **Gradient boosting:** This algorithm is also called GBM and includes the learning rate. We have seen a very detailed description of the GBM in the previous chapter.
- **Stochastic gradient boosting:** A big insight into bagging ensembles and random forests was allowing trees to be greedily created from subsamples of the training dataset. This same benefit can be used to reduce the correlation between trees in a sequence in gradient boosting models. This variation of boosting is called stochastic gradient boosting. A few variants of stochastic

boosting that can be used are:

- Subsample rows before creating each tree
- Subsample columns before creating each tree
- Subsample columns before considering each split

Generally, aggressive subsampling such as selecting only 50% of the data has shown to be beneficial. Stochastic gradient boosting reduces high variance trade-off chances and it allows using parallel architecture implementations, which helps in computations on distributed computing systems.

- **Regularized gradient**

boosting: Classical decision trees such as CART are not used as weak learners; instead, a modified form called a regression tree is used. It has numeric values in the leaf nodes (also called

terminal nodes). The values in the leaves of the trees can be called weights in some literature.

As such, the leaf weight values of the trees can be regularized using popular regularization functions such as:

- L1 regularization of weights
- L2 regularization of weights

System features

The library provides a system for use in a range of computing environments, not least the following:

- **Parallelization** of tree construction using all of your CPU cores during training. This makes training of the trees very fast; it is essential in case of larger datasets, where we have millions of instances to train.
- **Distributed computing** for training very large models using a cluster of machines. This utilizes the power of multiple workers; we can use many CPUs in working together on the large datasets.
- **Out-of-Core computing** for very large datasets that don't fit into memory. This will be required when the size of our dataset is in GB; in such cases, lower end systems can't load the whole dataset

into the system RAM at once.

- **Cache optimization** of data structures and algorithms to make best use of hardware. This allows maximum utilization of the hardware, which results in highest efficiency in real-time-based applications.

Algorithm features

The implementation of the algorithm was engineered for efficiency of compute time and memory resources. One design goal was to make the best use of available resources to train the model. Some key algorithm implementation features include:

- **Sparse-aware** implementation with automatic handling of missing data values. It is very helpful functionality in real-world datasets, where you will find missing values of attributes very often; if you discard the whole instance because of the missing value, it affects the size of the dataset. XGBoost's algorithm uses interpolation to predict the missing value.
- **Block structure** to support parallelization of tree construction.
- **Continued training** so that you can further boost an already fitted model on

new data. This is very useful functionality in the case of online training systems where you don't have a fixed-length dataset; in such cases, you can use a previously trained model to append new trees to the system for the new instances.

- **Tree pruning** a GBM would stop splitting a node when it encounters a negative loss in the split. Thus it is more of a greedy algorithm. XGBoost on the other hand makes splits up to the `max_depth` specified and then starts pruning the tree backwards to remove splits beyond which there is no positive gain. Another advantage is that sometimes a split of negative loss, say -2 , may be followed by a split of positive loss, say $+10$. GBM will stop as soon as it encounters -2 . But XGBoost will go deeper; it will see a combined effect of $+8$ of the split and keep both.
- **Built-in cross-validation** XGBoost allows the user to run a cross-validation in each iteration of the boosting process

and thus it is easy to get the exact optimum number of boosting iterations in a single run. This is unlike GBM, where we have to run a grid search and only limited values can be tested.

So you can see the benefits of a third-party library you can use them in a generalized way.

Why use XGBoost?

The two reasons to use XGBoost are:

- Execution speed
- Model performance

XGBoost execution speed

Generally, XGBoost is fast; it is really fast compared to other implementations of gradient boosting.

Szilard Pafka, a chief data scientist, performed some objective benchmarks, comparing the performance of XGBoost with other implementations of gradient boosting and bagged decision trees. He wrote his results in May 2015 in a blog post titled *Benchmarking Random Forest Implementations*.

He also provides all of the code on GitHub (<https://github.com/szilard/benchm-ml>) and a more extensive report of results with hard numbers:

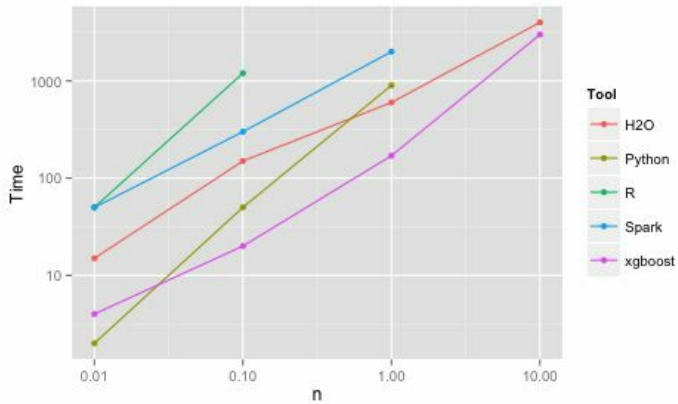


Figure 7.2: Speed comparison of XGBoost

His results showed that XGBoost was almost always faster than the other benchmarked implementations from R, Python, Spark, and H2O.

Model performance

XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems. The evidence is that it is the go-to algorithm for competition winners on the **Kaggle** competitive data science platform.

Again, we have already talked about various parameters that improve its performance over other ensemble methods such as tree bagging.

How to install

Installing XGBoost on a Windows system is a bit painful method, while on Linux-based machines, it is quite easy. By the way, I will share an easy solution to install it on a Windows machine later. You can find the detailed descriptions of installation on various platforms from the official page of XGBoost (<http://xgboost.readthedocs.io/en/latest/build.html#building-on-windows>). We will discuss the installation on two major platforms, that is, Windows and Linux.

To install XGBoost on any system, there are two steps to be taken:

1. First, build the shared library from the C++ code (`libxgboost.so` for Linux/OS X and `libxgboost.dll` for Windows versions). Note the exception: for an R package installation, please directly refer to the R package section.

2. Then install the language packages (for example, Python package).

Building the shared library

Our goal is to build the shared library:

- On Linux/OS X, the target library
is `libxgboost.so`
- On Windows, the target library
is `libxgboost.dll`

The minimal building requirement is a recent C++ compiler supporting C++ 11 (g++-4.8 or higher).

We can edit `make/config.mk` to change the compile options, and then build by `make`. If everything goes well, we can go to the specific language installation section.

Building on Ubuntu/Debian

On Ubuntu, one builds `xgboost` by typing the following:

```
| git clone --recursive  
| https://github.com/dmlc/xgboost  
| cd xgboost; make -j4
```

Building on Windows

You need to first clone the `xgboost` repo with recursive option and clone the submodules. If you are using GitHub tools, you can open the git-shell, and type the following command. We recommend using <https://git-for-windows.github.io/> because it brings a standard bash shell. This will highly ease the installation process:

```
| git submodule init  
| git submodule update
```

XGBoost supports both builds: by *MSVC* or *MinGW*. Here is how you can build the XGBoost library using MinGW.

After installing <https://git-for-windows.github.io/>, you should have a shortcut Git Bash. All of the following steps are in Git Bash.

In MinGW, the `make` command comes with the name `mingw32-make`. You can add the following line into the `.bashrc` file:

```
| alias make='mingw32-make'
```

To build with MinGW:

```
| cp make/mingw64.mk config.mk; make -j4
```

To build with Visual Studio 2013, use `cmake`. Make sure you have a recent version of `cmake` added to your path, and then from the `xgboost` directory, do this:

```
| mkdir build  
| cd build  
| cmake .. -G"Visual Studio 12 2013 Win64"
```

This specifies an out-of-source build using the MSVC 12 64-bit generator. Open the `.sln` file in the `build` directory and build with Visual Studio. To use the Python module, you can copy `libxgboost.dll` into `Python-package\xgboost`.

A trick for easy installation on a Windows machine

Now this is the interesting part; to build the library, you will need Microsoft Visual Studio. You can download a trial version from the official website of the product; I have already put a ZIP file that contains the important files needed to build the shared library. The dropbox location of the ZIP is <https://www.dropbox.com/s/fsa4xtbom3bg09b/xgboost-0.40.zip?dl=0>. The following are the steps:

1. Download the .zip file
2. Unzip the file to access the directories using any extractor tool such as WinRAR.
3. Locate the folder named `windows`
4. Inside the folder, you will find the Microsoft Visual Studio solution file

`xgboost.sln`

5. Open the solution file in Visual Studio and build it in release mode for the desired system architecture (32-bit or 64-bit)
6. Once the library successfully builds, go to the folder named `wrapper` in the `xgboost` folder
7. Now open a command window here and type:

```
| Python setup.py install
```

This will install the library in your Python site-packages folder and now you are ready to use the library in any Python IDE.

XGBoost in action

So after building successfully, it's time to test the library in Python. We will use an old dataset for predictions of diabetes; the dataset is available on the UCI machine learning repository. Its name is Pima Indians onset of diabetes dataset. Here is the direct link: <https://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data>.

Dataset information

Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females, at least 21 years age, and of Pima Indian heritage. ADAP is an adaptive learning routine that generates and executes digital analogs of perceptron-like devices. It is a unique algorithm; see the paper for details.

Attribute information

Here is the attribute information of the instances:

- Number of times pregnant
- Plasma glucose concentration a two hours in an oral glucose tolerance test
- Diastolic blood pressure (mm Hg)
- Triceps skin fold thickness (mm)
- Two hour serum insulin (μ U/ml)
- Body mass index (weight in kg/(height in m)²)
- Diabetes pedigree function
- Age (years)
- Class variable (zero or one)

Let's start coding; we have all the required files with us and now we are ready to implement the code for the problem. Let's do it!

```

# First XGBoost model for Pima Indians dataset
#Load the required libraries
#Numpy for reading the csv file

from numpy import loadtxt

#Import XGBoost classifier
from xgboost import XGBClassifier

#We will use sklearn to divide our data set
into training and test set
from sklearn.model_selection import
train_test_split

#We will use sklearn's accuracy metric to
evaluate the performance of the trained model
from sklearn.metrics import accuracy_score

#Let's load the dataset into the numpy array
dataset = loadtxt('pima-indians-diabetes.csv',
delimiter=",")

```

We must separate the columns (attributes or features) of the dataset into input patterns (x) and output patterns (y). We can do this easily by specifying the column indices in the NumPy array format:

```

#split data into X (input variables)and
y(output variable/Class)
X = dataset[:,0:8]
Y = dataset[:,8]

```

Finally, we must split the x and y data into a training and a test dataset. The training set will be used to prepare the XGBoost model

and the test set will be used to make new predictions, from which we can evaluate the performance of the model.

For this, we will use the `train_test_split()` function from the scikit-learn library. We also specify `seed` for the random number generator so that we always get the same split of data each time this example is executed:

```
#Create training and test set with 33% data in  
test set and 66% for the training of the model  
seed = 7  
test_size = 0.33  
X_train, X_test, y_train, y_test =  
train_test_split(X, Y, test_size=test_size,  
random_state=seed)
```

XGBoost provides a wrapper class to allow models to be treated like classifiers or regressors in the scikit-learn framework. This means we can use the full scikit-learn library with XGBoost models.

The XGBoost model for classification is called `XGBClassifier`. We can create and fit it to our training dataset. Models are fit using the scikit-learn API and the `model.fit()` function.

The parameters for training the model can be passed to the model in the constructor. Here, we use the sensible defaults:

```
#Train our first model on created training set
model = XGBClassifier()
model.fit(X_train, y_train)
```

To see the default parameter, we can print the model's default parameters with a simple

`print(model):`

```
XGBClassifier(base_score=0.5,
               colsample_bytree=1, gamma=0, learning_rate=0.1,
               max_delta_step=0, max_depth=3,
               min_child_weight=1, n_estimators=100,
               nthread=-1, objective='binary:logistic',
               seed=0, silent=True,
               subsample=1)
```

We can make predictions using the fit model on the test dataset. To make predictions, we use the scikit-learn function `model.predict()`. By default, the predictions made by XGBoost are probabilities. Because this is a binary classification problem, each prediction is the probability of the input pattern belonging to the first class. We can easily convert them to binary class values by rounding them to 0 or

1:

```
#Lets see the prediction from the trained model  
y_pred = model.predict(X_test)  
  
#Create a list of predictions for evaluation purpose  
predictions = [round(value) for value in  
y_pred]
```

Now that we have used the fit model to make predictions on new data, we can evaluate the performance of the predictions by comparing them to the expected values. To do this, we will use the built-in `accuracy_score()` function in `scikit-learn`:

```
#Evaluate predictions using accuracy metric  
accuracy = accuracy_score(y_test, predictions)  
  
#Print the accuracy  
print("Accuracy of the trained model is:  
%.2f%%" % (accuracy * 100.0))
```

After the execution of the preceding script, we will get:

```
| Accuracy of the trained model is: 76.38%
```

XGBoost parameters

It is necessary to understand the parameters of an algorithm to use it efficiently; we cannot rely on the default parameters for training the model as parameter tuning is a very subjective area. Some parameters can work best for a particular dataset while the same parameters may not give the same performance on other datasets. In this section, we will first understand the parameters and then look at how to tune them.

The overall parameters have been divided into three categories by XGBoost's authors:

- **General parameters:** These guide the overall functioning
- **Booster parameters:** These guide the individual booster (tree/regression) at each step
- **Learning task parameters:** These guide the optimization performed

General parameters

These define the overall functionality of XGBoost:

- `booster [default=gbtree]`: Select the type of model to run at each iteration. It has two options:
 - `gbtree`: The tree-based models
 - `gblinear`: The linear models
- `silent [default=0]`:
 - Silent mode is activated and set to one; that is, no running messages will be printed
 - It's generally good to keep this at zero as the messages might help in understanding the model
- `nthread [default to maximum number of threads available if not set]`:
 - This is used for parallel processing and the number of cores in the system should be entered

- If you wish to run on all cores, no value should be entered and the algorithm will detect automatically

There are two more parameters that are set automatically by XGBoost, so you need not worry about them. Let's move on to booster parameters.

Booster parameters

Though there are two types of boosters, I'll consider only the tree booster here because it always outperforms the linear booster and thus the latter is rarely used:

- `eta [default=0.3]:`
 - Analogous to learning rate in GBM
 - Makes the model more robust by shrinking the weights on each step
 - Typical final values to be used are *0.01-0.2*
- `min_child_weight [default=1]:`
 - This defines the minimum sum of weights of all observations required in a child.
 - This is similar to `min_child_leaf` in GBM but not exactly. It refers to the minimum sum of weights of observations while GBM has the minimum number of observations.
 - It is used to control overfitting.

Higher values prevent the model from learning relations that might be highly specific to the particular sample selected for a tree.

- Values that are too high can lead to under-fitting hence, it should be tuned using CV(Cross validation).
- `max_depth [default=6]`:
 - The maximum depth of a tree, it's the same as GBM
 - Used to control overfitting as a higher depth will allow the model to learn relations very specific to a particular sample
 - Should be tuned using CV
 - Typical values are 3-10
- `max_leaf_nodes`:
 - The maximum number of terminal nodes or leaves in a tree.
 - This can be defined in place of `max_depth`. Since binary trees are created, a depth of n would produce a maximum of 2^n leaves.
 - If this is defined, GBM will ignore `max_depth`.

- `gamma [default=0]:`
 - A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split.
 - It makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.
- `max_delta_step [default=0]:`
 - Is the maximum delta step, we allow each tree's weight estimation to be. If the value is set to zero, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative.
 - Usually this parameter is not needed, but it might help in logistic regression when a class is extremely imbalanced.
 - It is generally not used but you can explore further if you wish.

- `subsample [default=1]:`
 - The same as the subsample of GBM. This denotes the fraction of observations to be randomly sampled for each tree.
 - Lower values make the algorithm more conservative and prevent overfitting, but values that are too small might lead to underfitting.
 - Typical values are *0.5-1*.
- `colsample_bytree [default=1]:`
 - Similar to `max_features` in GBM. This denotes the fraction of columns to be randomly sampled for each tree.
 - Typical values are *0.5-1*.
- `colsample_bylevel [default=1]:`
 - Denotes the subsample ratio of columns for each split in each level.
 - I don't use this often, nor should you, because the `subsample` and `colsample_bytree` will do the job for you. But you can explore further if you feel the need.
- `lambda [default=1]:`
 - The L2 regularization term on

weights (analogous to Ridge regression).

- This used to handle the regularization part of XGBoost. Though many data scientists don't use it often, it should be explored to reduce overfitting.
- `alpha [default=0]:`
 - The L1 regularization term on weights (analogous to **Lasso regression**)
 - Can be used in case of very high dimensionality so that the algorithm runs faster when implemented
- `scale_pos_weight [default=1]:`
 - A value greater than zero should be used in case of high class imbalance as it helps in faster convergence

Learning task parameters

These parameters are used to define the optimization objective, the metric to be calculated at each step:

1. `objective [default=reg:linear]`: This defines the loss function to be minimized. The most commonly used values are:
 - `binary:logistic`: Logistic regression for binary classification; this returns the predicted probability (not class)
 - `multi:softmax`: Multiclass classification using the softmax objective; this returns the predicted class (not probabilities)
 - You also need to set an additional `num_class` (number of classes) parameter defining the number of unique classes

- `multi:softprob`: The same as softmax but returns the predicted probability of each data point belonging to each class

2. `eval_metric` [default according to objective]:

- The metric to be used for validation data
- The default values are `rmse` for regression and `error` for classification
- Typical values are:
 - `rmse`: Root Mean Square Error
 - `mae`: Mean Absolute Error
 - `logloss`: Negative log-likelihood
 - `error`: Binary classification error rate (0.5 threshold)
 - `merror`: Multiclass classification error rate
 - `mlogloss`: Multiclass logloss
 - `auc`: Area under the curve

3. `seed [default=0]:`

- The random number seed
- Can be used to generate reproducible results and also for parameter tuning

If you've been using scikit-learn until now, these parameter names might not look familiar. A good news is that the `xgboost` module in Python has an sklearn wrapper called `XGBClassifier`. It uses sklearn-style naming conventions. The parameters names that will change are:

- `eta` -> `learning_rate`
- `lambda` -> `reg_lambda`
- `alpha` -> `reg_alpha`

You must be wondering that we have defined everything except something similar to the `n_estimators` parameter in GBM. Well, this exists as a parameter in `XGBClassifier`.

However, it has to be passed as `num_boosting_rounds` while calling the `fit` function in the standard XGBoost

implementation.

Parameter tuning – number and size of decision trees

As we have seen, there are various parameters available for tuning in XGBoost, but the most important and most useful parameters are the number and size of the decision trees. In this part, we will discuss these in detail with the help of a Kaggle competition problem.

Problem description – Otto dataset

In this discussion, we will use the <https://www.kaggle.com/c/otto-group-product-classification-challenge> dataset.

This dataset is available for free from Kaggle (you will need to sign up with Kaggle to be able to download this dataset). You can download the training dataset, `train.csv.zip`, from <https://www.kaggle.com/c/otto-group-product-classification-challenge/data> and place the unzipped `train.csv` file in your working directory.

This dataset describes 93 obfuscated details of more than 61,000 products grouped into 10 product categories (for example, fashion, electronics, and so on). The input attributes

are counts of different events of some kind.

The goal is to make predictions for new products as an array of probabilities for each of the 10 categories, and models are evaluated using multiclass logarithmic loss (also called cross-entropy).

This competition was completed in May 2015, and this dataset is a good challenge for XGBoost because of the non-trivial number of examples, the difficulty of the problem, and the fact that little data preparation is required (other than encoding the string class variables as integers).

Tune the number of decision trees in XGBoost

Most implementations of gradient boosting are configured by default with a relatively small number of trees, such as hundreds or thousands. The general reason is that on most problems, adding more trees beyond a limit does not improve the performance of the model.

The reason is in the way the boosted tree model is constructed—sequentially—where each new tree attempts to model and correct the errors made by the sequence of the previous trees. Quickly, the model reaches a point of diminishing returns.

We can demonstrate this point of diminishing returns easily on the `otto` dataset. The number

of trees (or rounds) in an XGBoost model is specified to the `XGBClassifier` OR `XGBRegressor` class in the `n_estimators` argument. The default in the XGBoost library is 100.

Using scikit-learn, we can perform a grid search of the `n_estimators` model parameter, evaluating a series of values from 50 to 350 with a step size of 50 (50, 150, 200, 250, 300, 350):

```
# grid search
model = XGBClassifier()
n_estimators = [50, 100, 150, 200]
max_depth = [2, 4, 6, 8]
print(max_depth)
param_grid = dict(max_depth=max_depth,
n_estimators=n_estimators)
kfold = StratifiedKFold(n_splits=10,
shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid,
scoring="neg_log_loss", n_jobs=-1, cv=kfold,
verbose=1)
grid_result = grid_search.fit(X,
label_encoded_y)
```

We can perform this grid search on the `otto` dataset using 10-fold cross validation, requiring 60 models to be trained (6 *configurations* * 10 *folds*).

The full code listing is provided here for completeness:

```
# XGBoost on Otto dataset, Tune n_estimators

# These imports are common for all
# configurations we will discuss
from pandas import read_csv
from xgboost import XGBClassifier
from sklearn.model_selection import
GridSearchCV
from sklearn.model_selection import
StratifiedKFold
from sklearn.preprocessing import LabelEncoder
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot

# load data
data = read_csv('train.csv')
dataset = data.values

# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]

# encode string class values as integers
label_encoded_y =
LabelEncoder().fit_transform(y)

# grid search
model = XGBClassifier()
n_estimators = range(50, 400, 50)
param_grid = dict(n_estimators=n_estimators)
kfold = StratifiedKFold(n_splits=10,
shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid,
scoring="neg_log_loss", n_jobs=-1, cv=kfold)
grid_result = grid_search.fit(X,
label_encoded_y)
```

```

# summarize results
print("Best: %f using %s" %
      (grid_result.best_score_,
       grid_result.best_params_))
means =
grid_result.cv_results_['mean_test_score']
stds =
grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds,
                              params):
    print("%f (%f) with: %r" % (mean, stdev,
                                param))

# plot the results
pyplot.errorbar(n_estimators, means, yerr=stds)
pyplot.title("XGBoost n_estimators vs Log
Loss")
pyplot.xlabel('n_estimators')
pyplot.ylabel('Log Loss')
pyplot.savefig('n_estimators.png')

```

Running this example prints the following results:

```

Best: -0.001152 using {'n_estimators': 250}
-0.010970 (0.001083) with: {'n_estimators': 50}
-0.001239 (0.001730) with: {'n_estimators':
100}
-0.001163 (0.001715) with: {'n_estimators':
150}
-0.001153 (0.001702) with: {'n_estimators':
200}
-0.001152 (0.001702) with: {'n_estimators':
250}
-0.001152 (0.001704) with: {'n_estimators':
300}
-0.001153 (0.001706) with: {'n_estimators':
350}

```

We can see that the cross-validation log loss scores are negative. This is because the scikit-learn cross-validation framework inverted them. The reason? Internally, the framework requires that all metrics that are being optimized are to be maximized, whereas log loss is a minimization metric. It can easily be maximized by inverting the scores.

The best number of trees was `n_estimators=250`, resulting in a log loss of `0.001152`, but that's really not a significant difference from `n_estimators=200`. In fact, there is not a large relative difference in the number of trees between `100` and `350` if we plot the results.

Here is a line graph showing the relationship between the number of trees and mean (inverted) logarithmic loss, with the standard deviation shown as error bars:

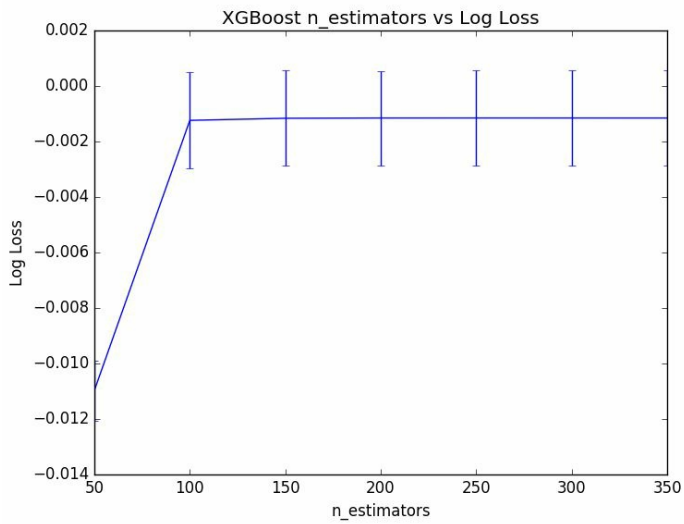


Figure 7.3: number of tree versus loss

Tuning the size of decision trees in XGBoost

In gradient boosting, we can control the size of decision trees, also called the number of layers or depth.

Shallow trees are expected to have poor performance because they capture few details of the problem and are generally referred to as weak learners. Deeper trees generally capture too many details of the problem and overfit the training dataset, limiting the ability to make good predictions on new data.

Generally, boosting algorithms are configured with weak learners. Decision trees with few layers, sometimes as simple as just a root node, are also called decision stumps rather than a decision tree.

The maximum depth can be specified in the `XGBClassifier` and `XGBRegressor` wrapper classes for XGBoost in the `max_depth` parameter. This parameter takes an integer value and defaults to a value of 3.

We can tune this hyperparameter of XGBoost using the grid search infrastructure in scikit-learn on the `otto` dataset. Now we will evaluate the odd values for `max_depth` between one and nine (1, 3, 5, 7, 9).

Each of the five configurations is evaluated using 10-fold cross validation, resulting in 50 models being constructed. The full code listing is provided here for completeness:

```
# XGBoost on Otto dataset, Tune max_depth
# load data
data = read_csv('train.csv')
dataset = data.values

# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]

# encode string class values as integers
label_encoded_y =
LabelEncoder().fit_transform(y)

# grid search
```

```

model = XGBClassifier()
max_depth = range(1, 11, 2)
print(max_depth)
param_grid = dict(max_depth=max_depth)
kfold = StratifiedKFold(n_splits=10,
shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid,
scoring="neg_log_loss", n_jobs=-1, cv=kfold,
verbose=1)
grid_result = grid_search.fit(X,
label_encoded_y)

# summarize results
print("Best: %f using %s" %
(grid_result.best_score_,
grid_result.best_params_))
means =
grid_result.cv_results_['mean_test_score']
stds =
grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds,
params):
    print("%f (%f) with: %r" % (mean, stdev,
param))

# plot
pyplot.errorbar(max_depth, means, yerr=stds)
pyplot.title("XGBoost max_depth vs Log Loss")
pyplot.xlabel('max_depth')
pyplot.ylabel('Log Loss')
pyplot.savefig('max_depth.png')

```

Running this example prints the log loss for each `max_depth`. The optimal configuration was `max_depth=5`, resulting in a log loss of 0.001236:

```
| Best: -0.001236 using {'max_depth': 5}
```

```
-0.026235 (0.000898) with: {'max_depth': 1}  
-0.001239 (0.001730) with: {'max_depth': 3}  
-0.001236 (0.001701) with: {'max_depth': 5}  
-0.001237 (0.001701) with: {'max_depth': 7}  
-0.001237 (0.001701) with: {'max_depth': 9}
```

Reviewing the plot of log loss scores, we can see a marked jump from `max_depth=1` to `max_depth=3` and then a pretty even performance for the rest the values of `max_depth`. Although the best score was observed for `max_depth=5`, it is interesting to note that there was practically little difference between using `max_depth=3` and `max_depth=7`.

This suggests a point of diminishing returns in `max_depth` on a problem that you can tease out using grid search. A graph of `max_depth` values is plotted against (inverted) logarithmic loss here:

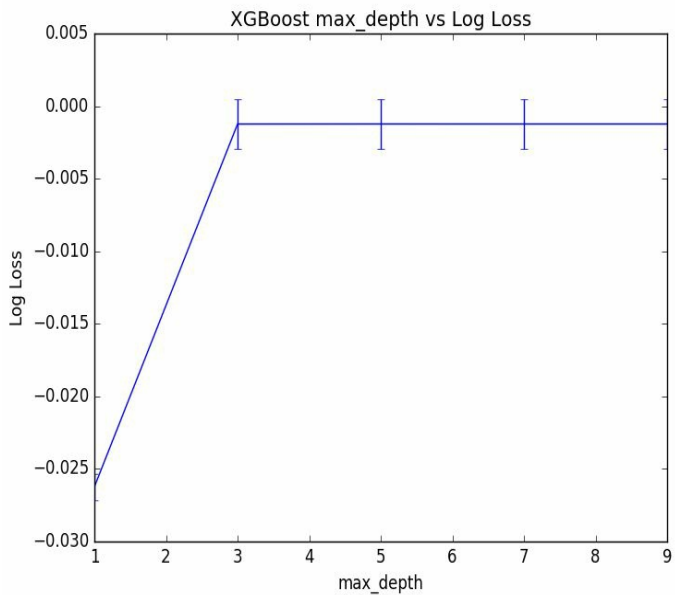


Figure 7.4: Depth of tree versus loss

Tuning the number of trees and max depth in XGBoost

There is a relationship between the number of trees in the model and the depth of each tree. We would expect that deeper trees result in fewer trees being required in the model, and vice versa, where simpler trees (such as decision stumps) require many more trees to achieve similar results.

We can investigate this relationship by evaluating a grid of `n_estimators` and `max_depth` configuration values. To prevent the evaluation from taking too long, we will limit the total number of configuration values evaluated. Parameters are chosen to tease out the relationship rather than optimize the model.

We will create a grid of four different `n_estimators` values (50, 100, 150, 200) and four different `max_depth` values (2, 4, 6, 8); each combination will be evaluated using 10-fold cross validation. A total of $4*4*10$ or 160 models will be trained and evaluated.

The full code listing is here:

```
# XGBoost on Otto dataset, Tune n_estimators
and max_depth
# load data
data = read_csv('train.csv')
dataset = data.values

# split data into X and y
X = dataset[:,0:94]
y = dataset[:,94]

# encode string class values as integers
label_encoded_y =
LabelEncoder().fit_transform(y)

# grid search
model = XGBClassifier()
n_estimators = [50, 100, 150, 200]
max_depth = [2, 4, 6, 8]
print(max_depth)
param_grid = dict(max_depth=max_depth,
n_estimators=n_estimators)
kfold = StratifiedKFold(n_splits=10,
shuffle=True, random_state=7)
grid_search = GridSearchCV(model, param_grid,
scoring="neg_log_loss", n_jobs=-1, cv=kfold,
verbose=1)
grid_result = grid_search.fit(X,
label_encoded_y)
```

```

# summarize results
print("Best: %f using %s" %
      (grid_result.best_score_,
       grid_result.best_params_))
means =
grid_result.cv_results_['mean_test_score']
stds =
grid_result.cv_results_['std_test_score']
params = grid_result.cv_results_['params']
for mean, stdev, param in zip(means, stds,
                             params):
    print("%f (%f) with: %r" % (mean, stdev,
                                param))

# plot results
scores =
numpy.array(means).reshape(len(max_depth),
                           len(n_estimators))
for i, value in enumerate(max_depth):
    pyplot.plot(n_estimators, scores[i],
                label='depth: ' + str(value))
pyplot.legend()
pyplot.xlabel('n_estimators')
pyplot.ylabel('Log Loss')
pyplot.savefig('n_estimators_vs_max_depth.png')

```

Running the code produces a listing of the log loss for each parameter pair:

```

Best: -0.001141 using {'n_estimators': 200,
                      'max_depth': 4}
-0.012127 (0.001130) with: {'n_estimators': 50,
                          'max_depth': 2}
-0.001351 (0.001825) with: {'n_estimators':
100, 'max_depth': 2}
-0.001278 (0.001812) with: {'n_estimators':
150, 'max_depth': 2}
-0.001266 (0.001796) with: {'n_estimators':
200, 'max_depth': 2}

```

```

-0.010545 (0.001083) with: {'n_estimators': 50,
'max_depth': 4}
-0.001226 (0.001721) with: {'n_estimators':
100, 'max_depth': 4}
-0.001150 (0.001704) with: {'n_estimators':
150, 'max_depth': 4}
-0.001141 (0.001693) with: {'n_estimators':
200, 'max_depth': 4}
-0.010341 (0.001059) with: {'n_estimators': 50,
'max_depth': 6}
-0.001237 (0.001701) with: {'n_estimators':
100, 'max_depth': 6}
-0.001163 (0.001688) with: {'n_estimators':
150, 'max_depth': 6}
-0.001154 (0.001679) with: {'n_estimators':
200, 'max_depth': 6}
-0.010342 (0.001059) with: {'n_estimators': 50,
'max_depth': 8}
-0.001237 (0.001701) with: {'n_estimators':
100, 'max_depth': 8}
-0.001161 (0.001688) with: {'n_estimators':
150, 'max_depth': 8}
-0.001153 (0.001679) with: {'n_estimators':
200, 'max_depth': 8}

```

We can see that the best result was achieved with `n_estimators=200` and `max_depth=4`, similar to the best values found from the previous two rounds of standalone parameter tuning (`n_estimators=250`, `max_depth=5`).

We can plot the relationship between each series of `max_depth` values for a given `n_estimators`:

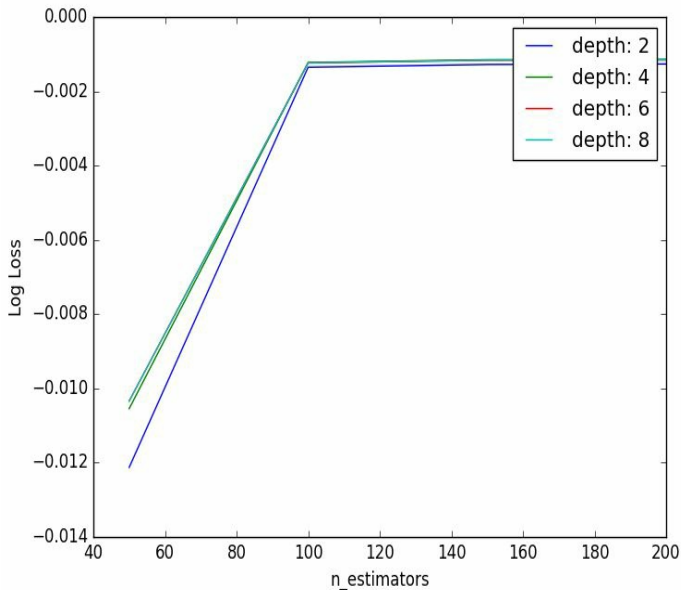


Figure 7.5: The relationship between max depth and number of trees

The lines overlap, making it hard to see the relationship, but generally we can see the interaction we expect. Fewer boosted trees are required with increased tree depth.

Further, we would expect the increased complexity provided by deeper individual trees to result in greater overfitting of the

training data. This would be exacerbated by having more trees, in turn resulting in a lower cross-validation score. We don't see this here as our trees are not that deep, nor do we have too many. Exploring this expectation is left as an exercise you can do yourself.

Summary

We started with a general description of XGBoost, then we saw the advantages of third-party libraries. We discussed various parameters, which we can tune to obtain good prediction accuracy from the classifier in quite some detail. We also worked with two practical applications. The first one was a gentle introduction to using a library with default parameters, and in the second application, we saw how to tune two important hyper parameters (number of trees and depth of trees). This can help us apply the algorithm on the complex datasets.

Overall, we have learned almost everything needed to start working with XGBoost. Again, practice makes a man perfect and it is the only key to success. I encourage you to use this library for your projects and try to learn more using the available web resources. It will help you to understand the limitations

of the algorithm in practical scenarios.

Stacked Generalization

We have seen the basic introduction to the process of stacking in [Chapter 1, *Introduction to Ensemble Learning*](#); stacking is an ensemble of more than one classifiers. So what? We have already seen bagging and boosting techniques in which we have already used many classifiers together, so what is the difference in stacking and the other two?

Well, there is a huge difference between them; in stacking, we combine multiple classifiers based on different algorithms, while in bagging and boosting, we combine the same kind of classifiers. For example, in bagging, we have combined many decision trees together to get a prediction, while in boosting, we have combined trees in cascading through an error sharing mechanism; however, in stacking, we usually

combine multiple classifiers from completely different mechanisms such as a combination of perceptron and logistic regression; what! What are perceptron and logistic regression? We will discuss them later in the chapter; let's once again see the basic framework of stacking, or in other words, stacked generalization.

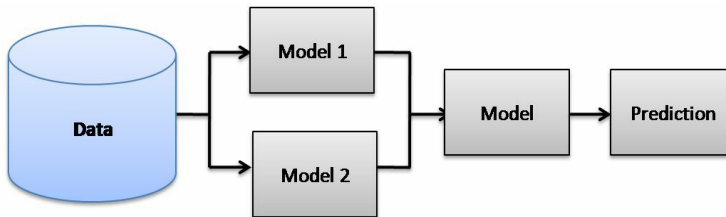


Figure 8.1: Stacking

Well, as we have seen in the previous chapters, a single decision tree in boosting or in bagging can help us only to make partial predictions to reduce bias error from our model; we need to increase the number of classifiers in our ensemble framework. In the same way, for complex datasets, a single solution might not give us higher prediction rate; for these situations, we need to combine

multiple different kinds of classifiers, where the output of one classifier can be the input of another. An interesting part of the stacking process is that the new model is trained to combine the prediction from previously trained models on the same dataset.

These are the topics that we will be learning in detail in this chapter :

- Stacked generalization
- Submodel training
- Stacked generalization implementation
- Practical application

Stacked generalization

Stacked generalization or stacking is an ensemble algorithm, where a new model is trained to combine the predictions from two or more models already trained on your dataset.

The predictions from the existing models or submodels are combined using a new model, and as such, stacking is often referred to as blending as the predictions from the submodels are blended together.

It is typical to use a simple linear method to combine the predictions for submodels, such as simple averaging or voting to a weighted sum using linear regression or logistic regression.

Models that have their predictions combined

must have skill on the problem, but do not need to be the best possible models. This means that you do not need to tune the submodels intently as long as the model shows some advantage over a baseline prediction.

It is important that submodels produce different predictions, the so-called **uncorrelated predictions**. Stacking works best when the predictions that are combined are all skillful, but skillful in different ways. This may be achieved using algorithms that use very different internal representations (trees compared to instances) and/or models trained on different representations or projections of the training data:

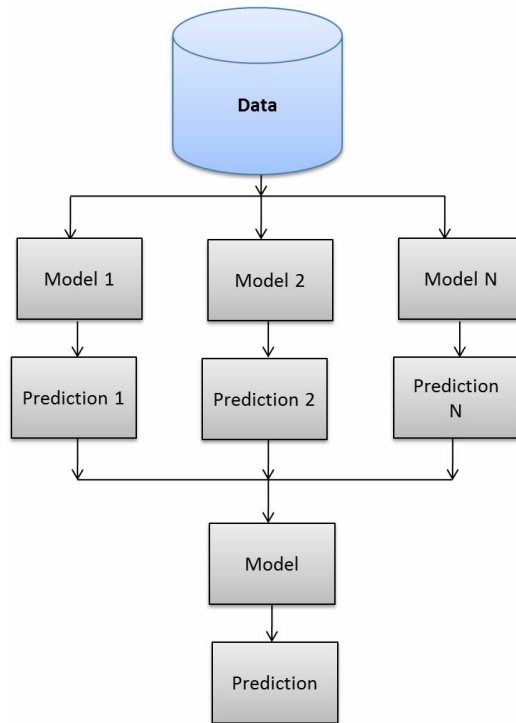


Figure 8.2: The stack generalization framework

The preceding figure shows a typical stacking framework, where we do not create different samples out of training data to train classifiers as we did in case of bagging, or we do not share error information between the classifiers as we did in boosting; instead, we train each classifier with the entire training

data that is, each classifier is independent of the other, which allows us to use classifiers with different hypothesis as well as algorithms. For example, we can use a linear regression classifier and random forest for training and then, we can combine their prediction using a **Support Vector Machine (SVM)**.

I think I have talked enough, and you guys have got a general understanding about the process, so this is the time to implement the procedure by ourselves and see what we can achieve at the end of the entire process.

As we have discussed, stacked generalization involves the following two parts:

- Training of multiple submodels
- Training a final model to combine the results of the submodels

We will implement both preceding steps, where we will train two submodels based on k nearest neighbor classifier and perceptron

respectively; then, we will combine their results using logistic regression. Sound interesting? So, let's start the ride.

Submodel training

Let's start with the submodel training; we will use two submodels, which will be trained separately and independently. Our submodels will be as follows:

- KNN classifier
- Perceptron

KNN classification

We will use the same KNN classifier that we have implemented in [Chapter 4, Random Subspace and KNN Bagging](#). It is a non-parametric algorithm, that means, it doesn't learn any underlying distributions of the dataset. It is originally derived from the k-means clustering algorithm, which is another very popular algorithm for unsupervised classification of data. In k means clustering, the similarity is the criteria to create clusters out of the data.

The following is the procedure of the KNN algorithm:

- As it is a supervised learning algorithm, we will provide data with the known labels to the algorithm
- Try to find similarity between unseen instance and known data using a distance metric and get most similar k

number of instance's label; these instances are known as **nearest neighbors**

- After getting nearest neighbors, we can get the label with the highest number, which will be the winner

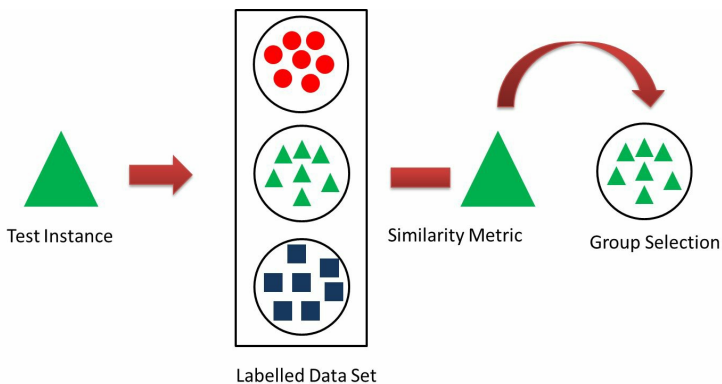


Figure 8.3: The KNN algorithm

The preceding figure shows the summary of the algorithm; we will use the same code as we have used previously. The code has three definitions, as follows:

- Distance calculation (Euclidean)
- Estimate the neighbors

- Make predictions using voting

Distance calculation (Euclidean)

The following is the code for distance calculation between two vectors:

```
def DistanceMetric(instance1, instance2,
isClass=None):
    #If Class variable is in the instance
    if isClass:
        length = len(instance1)-1
    else:
        length = len(instance1)

    #Initialize variable to store distance
    distance = 0

    #Lets run a loop to calculate element wise
differences
    for x in range(length):

        #Euclidean distance
        distance += pow((instance1[x] -
instance2[x]), 2)
    return math.sqrt(distance)
```

Estimating the neighbors

The following is the method for calculating neighbors on the basis of distance metric:

```
import operator
def getNeighbors(trainingSet, testInstance, k):

    #Create a list variable to store distances
    between test and
    #training instance.
    distances = []

    #Get distance between each instance in the
    training set and the
    #test instance.
    for x in range(len(trainingSet)):
        #As we will going to have class
        variable in the training set isClass will be
        true

    dist=DistanceMetric(testInstance,trainingSet[x],
        #Append the distance of each instance
        to the distance list
        distances.append((trainingSet[x],
        dist))

    #Sort the distances in ascending order
    distances.sort(key=operator.itemgetter(1))

    #Create a list to store the neighbors
    neighbors = []
```

```
        #Run a loop to get k neighbors from the  
sorted distances.  
        for x in range(k):  
            neighbors.append(distances[x][0])  
        return neighbors
```


Making predictions using voting

The following function will be responsible for getting predictions from `neighbors` using voting; it will select the label with the maximum `neighbors` as the prediction of the classifier:

```
import operator
def getPrediction(neighbors):

    #Create a dictionary variable to store votes from the neighbors
    #We will use class attribute as the dictionary keys and their occurrence as key value.
    classVotes = {}

    #Go to each neighbor and take the vote for the class
    for x in range(len(neighbors)):

        #Get the class value of the neighbor
        response = neighbors[x][-1]

        #Create class key if its not there;

        #If class key is in the dictionary increase it by one.
        if response in classVotes:
```

```

        classVotes[response] += 1
    else:
        classVotes[response] = 1

    #Sort the dictionary keys on the basis of
key values in descending order
    sortedVotes =
sorted(classVotes.iteritems(),
key=operator.itemgetter(1),
reverse=True)

    #Return the key name (class) with the
highest value
    return sortedVotes[0][0]

```

So, we have created our KNN classifier and it's time to create the next sub classifier which will be a perceptron; this will be a new kind of classifier that you will learn; let's see what perceptron is and how it works.

Perceptron

Perceptron is the building block of the **Artificial Neural Network (ANN)**; what is a neural network? To get started, I'll explain a type of artificial neuron called a **perceptron**. Perceptrons were developed in the 1950s and 1960s by a scientist, Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. Let's get into this discussion.

So, the first question that comes to the mind is, what is perceptron? The perceptron is a mathematical model of a biological neuron. While in actual neurons, the dendrite receives electrical signals from the axons of other neurons, in the perceptron, these electrical signals are represented as numerical values. At the synapses, between the dendrite and axons, electrical signals are modulated in various amounts. This is also modeled in the perceptron by multiplying each input value

by a value called the weight. An actual neuron fires an output signal, only when the total strength of the input signals exceeds a certain threshold. We model this phenomenon in a perceptron by calculating the weighted sum of the inputs to represent the total strength of the input signals and applying a step function on the sum to determine its output. As in biological neural networks, this output is fed to other perceptrons.

Now, how does perceptron work? The perceptron receives the input data multiplied by random weights and adds a bias value; put in **Activation function** to get a result. If the result value is wrong, it uses backpropagation and gradient descent to go back and tweak the weights to get a correct result.

The following figure will give you a basic understanding of the process:

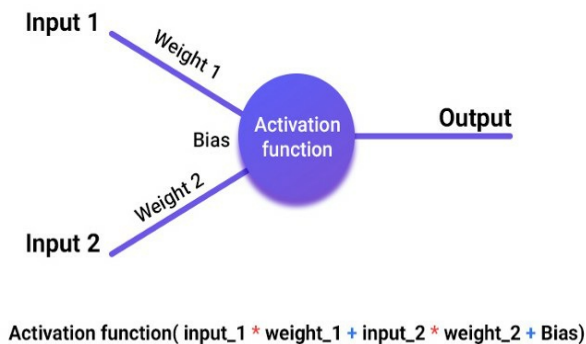


Figure 8.4: Basic perceptron

Let's elaborate the concept with the help of the following example:

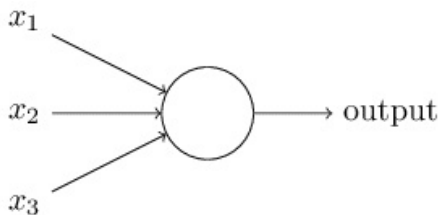


Figure 8.4a: Perceptron in practical scenario

The preceding figures show that the perceptron has three inputs, x_1 , x_2 , and x_3 . In general, it could have more or fewer inputs.

Rosenblatt proposed a simple rule to compute the output. He introduced weights, w_1, w_2, \dots , real numbers expressing the importance of the respective inputs to the output. The neuron's output, zero or one, is determined by whether the weighted sum, $\sum_j w_j x_j$, is less than or greater than some threshold value. Just like the weights, a threshold is a real number, which is a parameter of the neuron.

Let's put it in precise algebraic terms:

$$output = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq threshold \\ 1 & \text{if } \sum_j w_j x_j > threshold \end{cases}$$

That's all there is to how a perceptron works!

That's the basic mathematical model. A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence. Let me give an example. It's not a very realistic example, but it's easy to understand, and we'll soon get to more realistic examples. Suppose there is a soccer

tournament near your city and you want to watch the match this weekend. You love to watch live soccer and want to plan it with your friends; your decision to go for the match most likely depends on the following factors:

1. Is the weather good?
2. Does your group want to go with you?
3. Is the place of the tournament easy to access?

We can represent these three factors by corresponding binary variables x_1 , x_2 , and x_3 . For instance, we'd have $x_1=1$ if the weather is good and $x_1=0$ if the weather is bad. Similarly, $x_2=1$ if your friends want to go with you and $x_2=0$ if not, and similarly again for x_3 .

Now, suppose you are a hardcore fan of soccer and want to go, whether your friends want to go with you or not, so you will be less concerned about the second factor.

However, if the weather is bad, there are fewer chances of the match taking place and it is an important factor then. You can use perceptrons to model this kind of decision-making. Now, as you know which factors are more important, you can give more weight to them as the weather is a key factor. Suppose we assign it a weight of six ($w_1=6$), the larger value of w_1 indicates that the weather matters a lot to you, much more than whether your friends join you or the accessibility of the venue. Finally, suppose you choose a threshold of five for the perceptron. With these choices, the perceptron implements the desired decision-making model outputting one whenever the weather is good and zero whenever the weather is bad. It makes no difference to the output whether your friends want to go or whether the venue is nearby.

The following figure will give you a better understanding of decision-making process as we have seen that all our inputs are in binary (zero and one), and we have decided the

weights $W_1=6$, $W_2=3$ and $W_3 = 1$. Now, we can prepare a result table for all of the possible combinations (as shown in the figure) in which the column will represent the preceding three factors. If we multiply the weights to the factors and add them together, we will get the sum of the product value in the last column. Now, you can see that when the weather is good, you will surely go to watch the match as we have decided five as the threshold value:

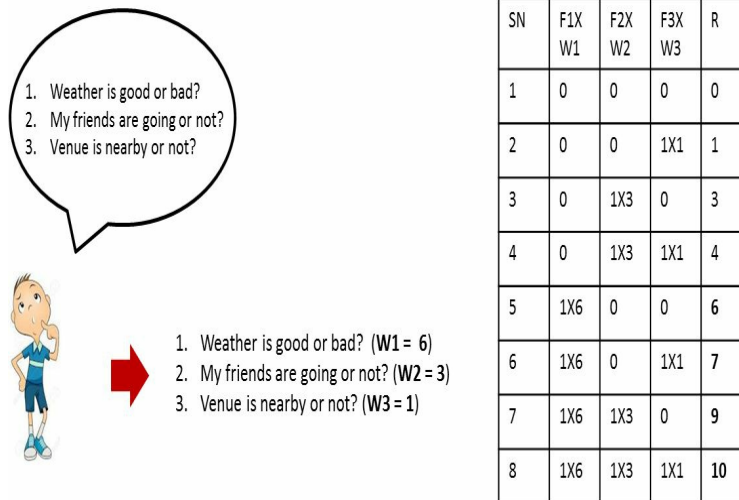


Figure 8.5: The decision-making process

By varying the weights and threshold, we can get different models of decision-making. For example, suppose we chose a threshold of three, then the perceptron would decide that you should go to the match whenever the weather was good or when both the match was near you and your friends were willing to join you. In other words, it'd be a different model of decision-making. Dropping the threshold means you're more willing to go to the match.

Obviously, the perceptron isn't a complete model of human decision-making! But what the example illustrates is how a perceptron can weigh different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions:

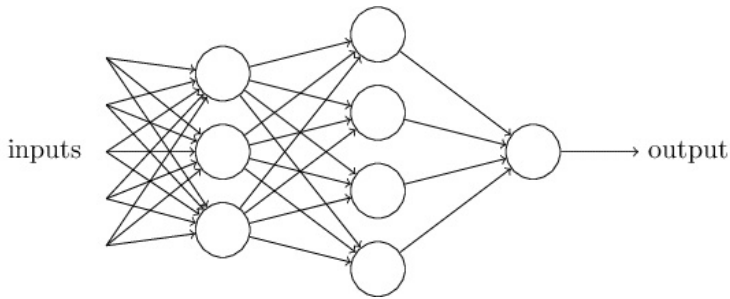


Figure 8.6: ANN using multilayer perceptrons

In this network, the first column of perceptrons—we'll call it the first layer of perceptrons—makes three very simple decisions, by weighing the input evidence. What about the perceptrons in the second layer? Each of those perceptrons makes a decision by weighing up the results from the first layer of decision-making. In this way, a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer, and even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision-making.

Incidentally, when I defined perceptrons, I said that a perceptron has a single output only. In the preceding network, the perceptrons look like they have multiple outputs. In fact, they're still single output. The multiple output arrows are merely a useful way of indicating that the output from a perceptron is being used as the input to several other perceptrons. It's less unwieldy than drawing a single output line, which then splits.

Let's simplify the way we describe perceptrons. We will write $\sum_j w_j x_j$ as a dot product, $w \cdot x \equiv \sum_j w_j x_j$, where w and x are vectors whose components are weights and inputs, respectively. The second change is to move the threshold to the other side of inequality and to replace it by what's known as the **perceptron's bias**, $b \equiv -\text{threshold}$. Using the bias instead of the threshold, the perceptron rule can be rewritten, as follows:

$$output = \begin{cases} 0 & \text{if } w \cdot x + b \leq threshold \\ 1 & \text{if } w \cdot x + b > threshold \end{cases}$$

You can think of the bias as a measure of how easy it is to get the perceptron to *output*, a *1*, or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to fire. For a perceptron with a really big bias, it's extremely easy for the perceptron to *output* a *1*. However, if the bias is negative, then it's difficult for the perceptron to *output* a *1*. Obviously, introducing the bias is only a small change in how we describe perceptrons, but we'll see later that it leads to further notational simplifications. Because of this, in the remainder of the book, we won't use the threshold; we'll always use the bias.

As you can see from the result table, perceptron works similar to the logic gates. This means as we can get a universal gate, we can simulate any kind of logic (for example, OR, AND, and so on) from that gate; for example, from NAND gate, which is known

as a universal gate, we can create any logic using a NAND gate. Similarly, we can use perceptrons as the universal decision maker. Wait! It means that a perceptron is a NAND gate, so what is the use of it as we already have the NAND gates from decades ago! However, the situation is better than this view suggests. It turns out that we can devise learning algorithms that can automatically tune the weights and biases of a network of artificial neurons. This tuning happens in response to external stimuli without the direct intervention of a programmer. These learning algorithms enable us to use artificial neurons in a way that is radically different to conventional logic gates. Instead of explicitly laying out a circuit of NAND and other gates, our neural networks can simply learn to solve problems, sometimes problems where it would be extremely difficult to directly design a conventional circuit.

Training the perceptron

Sounds quite interesting, and believe me, this is interesting. If you remember, we have discussed the concept of training a machine learning system in a gradient boosting machine, where you learned that training a system is nothing but minimization of the objective function, which is always the error of the system itself. So, how does it imply in the current case? Well, during the discussion of GBM, we have used the gradient-based approach for adding more trees to the system; here, we will use gradient descent to update the weights of the system.

Let's see some basics of the gradient descent before implementing it.

Gradient descent

As we have seen earlier in GBM, our objective function to minimize is an error between predicted and actual output. Let's rewrite the equation for this:

$$E = \sum_{i=0}^M (y_i - \bar{y})^2$$

equation (1)

Our final goal always is to reduce this error by improving our predictions; for perceptrons, error is simply a function of two variables that is weights and bias, so we can rewrite the equation, as follows:

$$C(W, b) = \sum_i (y_i - \bar{y})^2$$

equation (2)

From now on, we will call the preceding

objective function as the cost function, which is the combination of weights and bias. This cost function can tell us how accurate our prediction is; if the cost function is high, it shows the poor reconstruction of data, and lower cost function signifies higher prediction accuracy. In *equation (2)*, the predicted output is:

$$\bar{y} = \sum_j^M W * X + b$$

equation (3)

As you can see in the preceding equation, if we make changes in W and b , the change will directly reflect on predictions. So, we can modify the preceding equation as follows:

$$\Delta \bar{y} = \sum_j^M \Delta W * X + \Delta b$$

equation (4)

This is a very important concept toward optimization. As the preceding equation

shows the direct impact of modifying W and b , it suggests us that if we can find the optimal values of W and b , we can actually minimize the cost function.

So, the simplest way to find out the optimal value of W and b is random guessing. We can choose a scale of real numbers and try each value pair to find out the best pair of W and b . Don't you think it is a good idea? Certainly, there are many doubts about this concept of random guessing itself and the very first thing that comes to the mind is, where to start the scale? Suppose you say from -1 to 1 , then what if the best pair values are beyond the range? Suppose we have our best pair inside the range, now how you will decide whether the pair you have got is the best pair? See, random guessing does not help you much as it does not guarantee you an optimal solution. So, what to do? Well, to know this, let's see an example if it can help us to find out the solution to our problem.

Suppose you have planned a trip to a hill

station and you have reached it on time, but in the night, you have missed the last bus from the hill station to your town. Now, you want to get down from the hill; what will you do? The best possible way is to start moving in the direction of descent of the hill; if you continuously move toward the descent, you will surely reach the bottom of the hill:

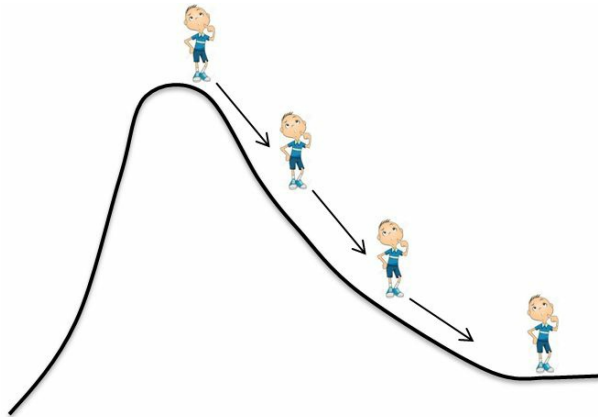


Figure 8.7: The gradient descent intuition

We can apply the same thing to our mathematical system, too! How? Let's understand this.

As you already know, we have a cost

function $C(\mathbf{W}, \mathbf{b})$ with two variables \mathbf{W} and \mathbf{b} ; our task is to find the pair of value for \mathbf{W} and \mathbf{b} , which can minimize this cost function. Now, if we assume the cost function as a three-dimensional surface, it will look similar to the following figure:

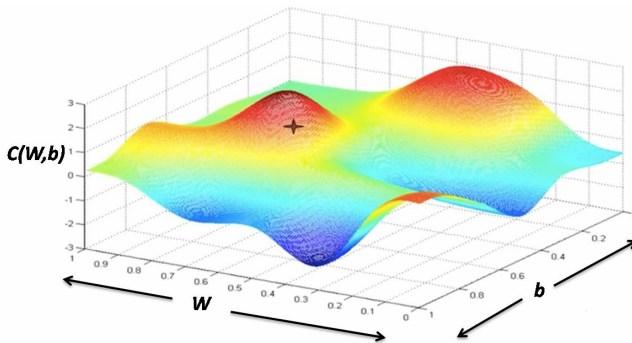


Figure 8.8: The surface plot of cost function

Now, suppose we started with some random values of \mathbf{W} and \mathbf{b} , which is shown in the figure as a star; we will choose the next value such that our cost follows the slope (gradient) toward the descent. If we continue toward the direction of the gradient, we will reach the minimum surface. The following figure summarizes the process:

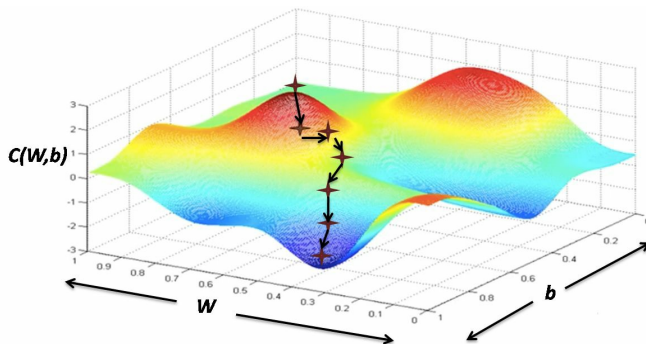


Figure 8.9: The cost function optimization using gradient descent

Now, the question is how to apply a gradient descent to minimize the cost function? Let's see a small derivation for this; I will try to avoid the mathematics here to make it simple.

Let's break down the problem; as we know whenever we will change the **(W,b)** pair, it will reflect the change in the cost function as well. So, when we start descending a hill, we will make small changes in W and b that will change the cost function as follows:

$$\Delta C \approx \frac{\partial C}{\partial W} \Delta W + \frac{\partial C}{\partial b} \Delta b$$

equation (5)

Here, ΔW and Δb are small amount changes made in W and b ; to see their effect, we will take the partial derivative of cost function C with respect to W and b . We will find a way of choosing ΔW and Δb , so we can make ΔC negative so that we can go downhill. To figure out how to make such changes, let's consider a Δv vector of changes in variable W and b ; further, we can write $\Delta v = (\Delta W, \Delta b)^T$, where T is the transpose operation for converting rows into columns. Similarly, we can also write the gradient of C as a vector of the partial derivative, as follows:

$$\nabla C \equiv \left(\frac{\partial C}{\partial W} + \frac{\partial C}{\partial b} \right)^T$$

equation (6)

Where ∇C is gradient vector, with these definitions, *expression (5)* can be rewritten as follows:

$$\Delta \approx \nabla C \cdot \Delta v$$

equation (7)

This equation helps to explain why ∇C is called the gradient vector: ∇C relates changes in v to changes in C , just as we'd expect something called a gradient to do. However, what's really exciting about the equation is that it lets us see how to choose Δv so as to make ∇C negative. In particular, suppose we choose the following equation:

$$\Delta v = -\eta \cdot \nabla C$$

equation (8)

Here, η is a small, positive parameter (also known as **learning rate**), then, the *equation (7)* tells us that $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$ because of $\|\nabla C\|^2 \geq 0$; this guarantees that $\Delta C \leq 0$ that is, C will always decrease, never increase. If we change v according to the *equation (8)*, this is exactly the property we wanted! So, we will use the *equation (8)* to compute a value for Δv ; then, move the value of v by that amount:

$$v \rightarrow v' = v - \eta \cdot \nabla C$$

equation (9)

Then, we'll use this update rule again to make another move. If we keep doing this over and over, we'll keep decreasing C until—we hope—we reach a global minimum.

To make gradient descent work correctly, we need to choose the learning rate η to be small enough that *equation (7)* is a good approximation. If we don't, we might end up with $\Delta C > 0$, which obviously would not be good! At the same time, we don't want η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly. In practical implementations, η is often varied so that *equation (7)* remains a good approximation, but the algorithm isn't too slow. We'll see later how this works.

You can think of this update rule as *defining* the gradient descent algorithm. It gives us a way of repeatedly changing the position v , in order to find a minimum of the

function C . The rule doesn't always work - several things can go wrong and prevent gradient descent from finding the global minimum of C . But, in practice, gradient descent often works extremely well, and in neural networks, we'll find that it's a powerful way of minimizing the cost function and helping the net learn.

How can we apply gradient descent to learn in a perceptron? The idea is to use gradient descent to find the weights w_k and biases b_l that minimize the cost in *equation (2)*. To see how this works, let's restate the gradient descent update rule, with the weights and biases replacing the variable v . In other words, our position now has components w_k and b_l , and the gradient vector ∇C has corresponding components $\partial C / \partial w_k$ and $\partial C / \partial b_l$. Let's write it for our variable update:

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

By repeatedly applying this update rule, we can *roll down the hill* and hopefully, find a minimum of the cost function. In other words, this is a rule that can be used to learn in a perceptron.

Stochastic gradient descent

An idea called **stochastic gradient descent** can be used to speed up learning. The idea is to estimate the gradient ∇C by computing ∇C_x for a small sample of randomly chosen training inputs. By averaging over this small sample, it turns out that we can quickly get a good estimate of the true gradient ∇C , and this helps speed up gradient descent and thus learning.

To make these ideas more precise, stochastic gradient descent works by randomly picking out a small number m of randomly chosen training inputs. We'll label these random training inputs X_1, X_2, \dots, X_m and refer to them as a **mini-batch**. Provided the sample size m is large enough, we expect that the average value of ∇C_{X_j} will be roughly equal

to the average over all ∇C_x that is, as follows:

$$\frac{\sum_{j=1}^m \nabla C_{x_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C$$

Here, the second sum is over the entire set of training data; swapping sides, we get:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^m \nabla C_{x_j}$$

Confirming this, we can estimate the overall gradient by computing gradients just for the randomly chosen mini-batch.

To connect this explicitly to learning in neural networks, suppose w_k and b_l denote the weights and biases in our neural network. Then, stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs and training with those:

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial b_l}$$

Here, the sum is over all the training examples x_j in the current mini-batch; then, we pick out another randomly chosen mini-batch and train with those and so on, until we've exhausted the training inputs, which is said to complete an epoch of training. At this point, we start over with a new training epoch.

Implementation of perceptron

Now, as we have enough mathematical description of perceptron algorithm and its training, we are ready to implement it in a function. In this section, we will implement our first perceptron algorithm along with its training, and it is worth noting that this is our first step to make an ANN, so let's not waste our time and let's start the implementation.

As we have discussed quite a lot, a perceptron is nothing but a linear combination of weights and input instances with some bias value, and it can be written as follows:

$$activation = sum(weight_i * x_i) + bias$$

Here, x_i is the i^{th} variable value of the input instance and $weight_i$ is the weight value

corresponding to the variable. We will add a *bias* value to *sum* of the product of all the variables and weights in the instance and we will call it activation of perceptron. The activation is then transformed into an output value or *prediction* using a transfer function, such as the step transfer function:

$$\textit{prediction} = 1.0 \text{ if } \textit{activation} \geq 0.0 \\ \text{else } 0.0$$

In this way, the perceptron is a classification algorithm for problems with two classes (zero and one), where a linear equation (like or hyperplane) can be used to separate the two classes.

It is closely related to linear regression and logistic regression that make predictions in a similar way (for example, a weighted sum of inputs).

The weights of the perceptron algorithm must be estimated from your training data using stochastic gradient descent.

For the perceptron algorithm, we will update the weights in each iteration with the help of the following equation:

$$w = w + \text{learning_rate} * (\text{expected} - \text{predicted}) * x$$

Here, w is the weight being optimized, learning_rate is a learning rate that you must configure (for example, 0.01), $(\text{expected} - \text{predicted})$ is the prediction error for the model on the training data attributed to the weight, and x is the input value for which we want to update the weights.

So, here, $(\text{expected} - \text{predicted})$ is the gradient that will give us the direction to which we should move, and learning_rate will help us to decide how fast we want to move toward the descent.

I want to discuss the behavior of learning with you guys, as it is a very crucial parameter that decides the speed of

convergence. Anybody can be misled by this and want to put a higher learning rate, so one can find the optimization of cost function faster. However, this is not the actual case; let's understand it with the following figure:

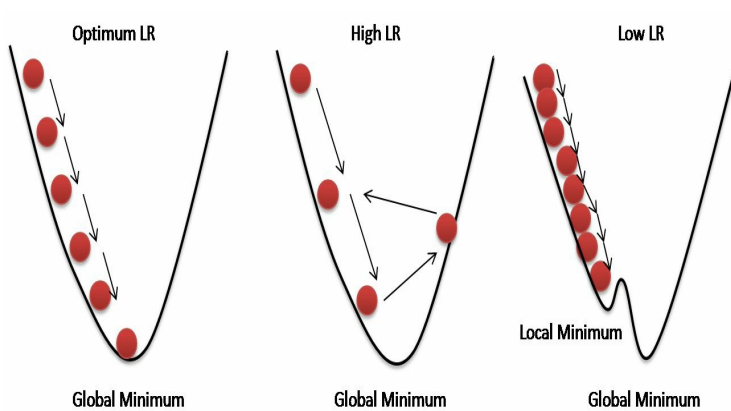


Figure 8.10: Variation of learning rate

The preceding figure shows three different cases of learning rates:

- When we choose an optimum learning rate, our function starts gradually decreasing and reaches the global minimum
- In case of a high learning rate, our

function takes very long steps toward the descent and misses the global minimum because of the long steps and just starts wandering here and there

- When we input a learning rate too slow, the function starts descending very slowly and may be trapped in a local minimum because it thinks that this is the best minimum value of the cost function for the given weight and bias pairs

So, we should choose our function's learning rate very carefully, so we can avoid the preceding two conditions.

Now, we will add a predict method to our perceptron algorithm that will be responsible for calculating the first `activation` and then, return a `1` or `0` based on the activation's value:

```
# Make a prediction with weights
def predict(row, weights):
    #Row is the input instance

    #We will consider first weight as the bias
    for simplified the calculations
        activation = weights[0]
```

```

        #Now run a loop to multiply each attribute
value of the instance with the
weight
        #And add the result to the activation of
previous attribute
        for i in range(len(row)-1):
            activation += weights[i + 1] * row[i]

        #Here we will return 1 if activation is a
non negative value and zero in
other case
    return 1.0 if activation >= 0.0 else 0.0

```

Let's test the preceding function on a toy dataset:

```

# test predictions
dataset = [[2.7810836, 2.550537003, 0],
           [1.465489372, 2.362125076, 0],
           [3.396561688, 4.400293529, 0],
           [1.38807019, 1.850220317, 0],
           [3.06407232, 3.005305973, 0],
           [7.627531214, 2.759262235, 1],
           [5.332441248, 2.088626775, 1],
           [6.922596716, 1.77106367, 1],
           [8.675418651, -0.242068655, 1],
           [7.673756466, 3.508563011, 1]]

weights = [-0.1, 0.20653640140000007,
           -0.23418117710000003]

for row in dataset:
    prediction = predict(row, weights)
    print("Expected=%d, Predicted=%d" % (row[-1],
prediction))

```

There are two inputs values (X_1 and X_2) and

three weight values (*bias*, *w1*, and *w2*). The activation equation we have modeled for this problem is as follows:

$$\text{activation} = (w1 * X1) + (w2 * X2) + \text{bias}$$

Alternatively, with the specific weight values we chose by hand are as follows:

$$\text{activation} = (0.206 * X1) + (-0.234 * X2) + -0.1$$

Running this function, we get predictions that match the expected output (*y*) values:

```
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
```

Now, we are ready to implement the stochastic gradient descent to optimize our weight values. We can estimate the weight

values for our training data using the stochastic gradient descent.

Stochastic gradient descent requires two parameters:

- **Learning rate:** Used to limit the amount each weight is corrected each time it is updated
- **Epochs:** The number of times to run through the training data while updating the weight

These, along with the training data will be the arguments to the function.

There are three loops we need to perform in the function:

1. Loop over each epoch
2. Loop over each row in the training data for an epoch
3. Loop over each weight and update it for a row in an epoch

As you can see, we update each weight for

each row in the training data, each epoch.

Weights are updated based on the error the model made. The error is calculated as the difference between the expected output value and the prediction made with the candidate weights.

There is one weight for each input attribute and these are updated in a consistent way, for example:

$$w(t+1) = w(t) + \text{learning_rate} * (\text{expected}(t) - \text{predicted}(t)) * x(t)$$

The *bias* is updated in a similar way, except without an input as it is not associated with a specific input value:

$$\text{bias}(t+1) = \text{bias}(t) + \text{learning_rate} * (\text{expected}(t) - \text{predicted}(t))$$

Now, we can put all of this together. The following is a function named `train_weights()` that calculates the

weight values for a training dataset using stochastic gradient descent:

```
# Estimate Perceptron weights using stochastic
gradient descent
def train_weights(train, l_rate, n_epoch):

    #Lets initialize the weights by 0
    weights = [0.0 for i in
range(len(train[0]))]

    #We will update the weights for given
number of epoch
    for epoch in range(n_epoch):

        #Extract each row from the training set
        for row in train:

            #Predict the value for the instance
            prediction = predict(row, weights)

            #Calculate the difference(gradient)
between actual and predicted
            value
            error = row[-1] - prediction

            #Update the bias value using given
learning rate and error
            weights[0] = weights[0] + l_rate *
error

            #Update the weights for each
attribute using learning rate
            for i in range(len(row)-1):
                weights[i + 1] = weights[i + 1]
+ l_rate * error * row[i]

        #Return the updated weights and biases
        return weights
```

Let's try to update the weight:

```
# Calculate weights
dataset = [[2.7810836, 2.550537003, 0],
           [1.465489372, 2.362125076, 0],
           [3.396561688, 4.400293529, 0],
           [1.38807019, 1.850220317, 0],
           [3.06407232, 3.005305973, 0],
           [7.627531214, 2.759262235, 1],
           [5.332441248, 2.088626775, 1],
           [6.922596716, 1.77106367, 1],
           [8.675418651, -0.242068655, 1],
           [7.673756466, 3.508563011, 1]]

l_rate = 0.1
n_epoch = 5
weights = train_weights(dataset, l_rate,
                        n_epoch)
print(weights)
```

We will use a learning rate of 0.1 and train the model for only 5 epochs or 5 exposures of `weights` to the entire training dataset.

Running the example prints a message of each `epoch` with the sum squared error for that `epoch` and the final set of weights:

```
epoch=0, lrate=0.100, error=2.000
epoch=1, lrate=0.100, error=1.000
epoch=2, lrate=0.100, error=0.000
epoch=3, lrate=0.100, error=0.000
epoch=4, lrate=0.100, error=0.000
[-0.1, 0.20653640140000007,
 -0.23418117710000003]
```


You can see how the problem is learned very quickly by the algorithm.

So, we have the KNN algorithm in our hand and now we have the perceptron too; it's time to build the algorithm for the last block of our stacked generalization process logistic regression.

Logistic regression

Logistic regression is a classification algorithm. It is used to predict a binary outcome (1/0, yes/no, or true/false) given a set of independent variables. To represent binary/categorical outcome, we use dummy variables. You can also think of logistic regression as a special case of linear regression when the outcome variable is categorical, where we use a log of odds as a dependent variable. In simple words, it predicts the probability of occurrence of an event by fitting the data to a **logit** function.

Logistic regression is a part of a larger class of algorithms known as **Generalized Linear Model (GLM)**. In 1972, Nelder and Wedderburn proposed this model with an effort to provide a means of using linear regression to the problems that were not directly suited for application of linear regression. In fact, they proposed a class of

different models (linear regression, ANOVA, Poisson regression, and so on) that included logistic regression as a special case.

The logistic function

The logistic function, also called the **sigmoid** function, was developed by statisticians to describe properties of population growth in ecology, rising quickly and maxing out at the carrying capacity of the environment. It's an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits:

$$1 / (1 + e^{-value})$$

Here, e is the base of the natural logarithms (Euler's number or the `exp()` function in your spreadsheet) and the value is the actual numerical value that you want to transform. The following is a plot of numbers between -5 and 5 transformed into a range of 0 and 1 using the logistic function:

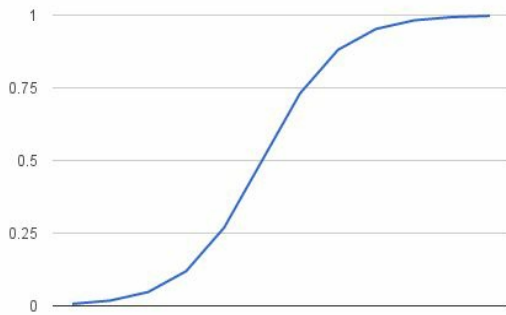


Figure 8.11: The logistic or sigmoid function

The preceding figure shows the form of a logistic function.

Representation of logistic regression

Logistic regression uses an equation as a representation, very much like the linear regression.

Input values (x) are combined linearly using weights or coefficient values (referred to as the Greek capital letter, beta) to predict an output value (y). A key difference from the linear regression is that the output value being modeled is a binary value (0 or 1), rather than a numeric value.

The following is an example of a logistic regression equation:

$$y = \frac{e^{b_0 + b_1 x}}{1 + e^{b_0 + b_1 x}}$$

Here, y is the predicted output, b_0 is the bias

or intercept term, and b_1 is the coefficient for the single input value (x). Each column in your input data has an associated b coefficient (a constant real value) that must be learned from your training data.

The actual representation of the model that you would store in memory or in a file is the coefficient in the equation (the beta value or b).

Modeling probability using logistic regression

Logistic regression models the probability of the default class (for example, the first class).

For example, if we are modeling people's *sex* as *male* or *female* from their *height*, then the first class could be *male* and the logistic regression model could be written as the probability of *male* given a person's *height*, or more formally:

$$P(\textit{sex} = \textit{male} | \textit{height})$$

Written another way, we are modeling the probability that an input (x) belongs to the default class ($y=1$), we can write this formally as:

$$P(x) = P(y = 1|x)$$

Are we predicting probabilities? I thought logistic regression was a classification algorithm?

Note that the probability prediction must be transformed into a binary value (0 or 1) in order to actually make a probability prediction.

Logistic regression is a linear method but the predictions are transformed using the logistic function. The impact of this is that we can no longer understand the predictions as a linear combination of inputs as we can with linear regression, for example, continuing on from the previous example, the model can be stated as:

$$P(x) = \frac{e^{b_0 + b_1 x}}{1 + e^{b_0 + b_1 x}}$$

I don't want to dive into the math too much, but we can turn around the preceding equation as follows (remember, we can

remove e from one side by adding a natural logarithm (\ln) to the other):

$$\ln \left(\frac{P(x)}{1 - P(x)} \right) = b_0 + b_1 x$$

This is useful because we can see that the calculation of the output on the right is linear again (just like linear regression), and the input on the left is a log of the probability of the default class.

This ratio on the left is called the odds of the default class (it's historical that we use *odds*, for example, *odds* are used in horse racing rather than probabilities). Odds are calculated as a ratio of the probability of the event divided by the probability of not the event, for example, $0.8/(1-0.8)$, which has the *odds* of 4. So, we could instead write the following equation:

$$\ln(odds) = b_0 + b_1 x$$

Because the *odds* are log transformed, we call this left-hand side the log-odds or the probit (

<https://en.wikipedia.org/wiki/Probit>). It is possible to use other types of functions for the transform (which is out of scope, but as such it is common to refer to the transform that relates the linear regression equation to the probabilities as the link function, for example, the **probit link** function.

We can move the exponent back to the right and write it as follows:

$$odds = e^{(b_0 + b_1 x)}$$

All of this helps us understand that indeed the model is still a linear combination of the inputs, but that this linear combination relates to the log-odds of the default class.

Learning the model

The coefficients (beta values b) of the logistic regression algorithm must be estimated from your training data. This is done using the maximum-likelihood estimation.

The maximum-likelihood estimation is a common learning algorithm used by a variety of machine learning algorithms, although it does make assumptions about the distribution of your data (more on this when we talk about preparing your data).

The best coefficients would result in a model that would predict a value very close to 1 (for example, *male*) for the default class and a value very close to 0 (for example, *female*) for the other class. The intuition of maximum-likelihood for logistic regression is that a search procedure seeks values for the coefficients (beta values) that minimize the error in the probabilities predicted by the

model to those in the data (for example, the probability of 1 if the data is the primary class).

We are not going to go into the math of maximum likelihood. It is enough to say that a minimization algorithm is used to optimize the best values for the coefficients for your training data. This is often implemented in practice using the efficient numerical optimization algorithm (like the **Quasi-Newton** method).

We will implement it by ourselves from scratch using a much simpler gradient descent algorithm.

Prediction using logistic regression

Making predictions with a logistic regression model is as simple as plugging in numbers into the logistic regression equation and calculating a result. Let's make this concrete with a specific example.

Let's say we have a model that can predict whether a person is male or female based on their height (completely fictitious). Given a height of 150 cm, is the person *male* or *female*?

You have learned that the coefficients of $b_0 = -100$ and $b_1 = 0.6$. Using the preceding equation, we can calculate the probability of male given a height of 150 cm or, more formally, $P(\text{male}|\text{height}=150)$:

$$y = \exp(-100 +$$

$$\frac{0.6*150}{1 + \exp(-100 + 0.6*X)}$$

$$y = 0.0000453978687$$

In practice, we can use the probabilities directly. Because this is a classification and we want a crisp answer, we can snap the probabilities to a binary class value, for example:

- 0 if $p(\text{male}) < 0.5$
- 1 if $p(\text{male}) \geq 0.5$

Now that we know how to make predictions using logistic regression, let's look at how we can prepare our data to get the most from the technique.

Implementation of algorithm

Like the perceptron algorithm, logistic regression uses a set of weights, called coefficients, as the representation of the model, and like the perceptron algorithm, the coefficients are learned by iteratively making predictions on the training data and updating them.

The following are the helper functions for implementing the logistic regression algorithm.

The `logistic_regression_model()` function is used to train the coefficients on the training dataset and `logistic_regression_predict()` is used to make a prediction for a row of data:

```
# Make a prediction with coefficients
def logistic_regression_predict(model, row):

    #First weight of the model will be bias
    similar as Perceptron function
```



```

    yhat = model[0]

    #We will run a loop to multiply each
    attribute value with the corresponding
    weights
    #This is similar to activation calculation
    in perceptron algorithm
    for i in range(len(row)-1):
        yhat += model[i + 1] * row[i]

    #Here we will apply logistic function on
    the linear combination of weights
    and attributes
    #This is the place where linear and
    logistic regression differs
    return 1.0 / (1.0 + exp(-yhat))

```

As you can see, the preceding function is very similar to perceptron's predict function, except here, we are adding nonlinearity to the activation using the exponential.

Now, you will learn weights for the preceding function using the stochastic gradient descent algorithm as we have done in the perceptron algorithm:

```

def logistic_regression_model(train,
    l_rate=0.01, n_epoch=5000):

    #Initialize the weights with the zero
    values
    coef = [0.0 for i in range(len(train[0]))]

    #Repeat the procedure for given number of
    epochs

```

```

    for epoch in range(n_epoch):

        #Get prediction for each row and update
weights based on error value
        for row in train:

            #Predict y for the given x
            yhat =
logistic_regression_predict(coef, row)

            #Get the error value
            (gradient/slope/change)
            error = row[-1] - yhat

            #Apply gradient descent here to
update the weights and biases
            #Update Bias first
            coef[0] = coef[0] + l_rate * error
* yhat * (1.0 - yhat)

            #Now update the Weights
            for i in range(len(row)-1):
                coef[i + 1] = coef[i + 1] +
l_rate * error * yhat * (1.0 - yhat)
                                * row[i]

            #Return the trained weights and biases
            return coef

```

Now, we are ready to implement our very first stacked generalization algorithm.

Stacked generalization implementation

For a machine learning algorithm, learning how to combine predictions is much the same as learning from a training dataset.

A new training dataset can be constructed from the predictions of the submodels, as follows:

- Each row represents one row in a training dataset
- The first column contains predictions for each row in the training dataset made by the first submodel, such as KNN
- The second column contains predictions for each row in the training dataset made by the second submodel, such as the perceptron algorithm
- The third column contains the expected output value for the row in the training

dataset

The following is a contrived example of what a constructed stacking dataset may look like:

kNN,	Per,	Y
0,	0	0
1,	0	1
0,	1	0
1,	1	1
0,	1	0

A machine learning algorithm, such as logistic regression can then be trained on this new dataset. In essence, this new meta-algorithm learns how to best combine the prediction from multiple submodels.

The following is a function named `to_stacked_row()` that implements this procedure for creating new rows for this stacked dataset.

The function takes a list of models as input; these are used to make the predictions. The function also takes a list of functions as input, one function used to make a prediction for each model. Finally, a single row from the

training dataset is included.

A new row is constructed one column at a time. Predictions are calculated using each model and the row of training data. The expected output value from the training dataset row is then added as the last column to the row:

```
# Make predictions with sub-models and
construct a new stacked row
def to_stacked_row(models, predict_list, row):

    #Let's Create an empty list to store
    predictions from sub models
    stacked_row = list()

    #Run a loop to fetch stored models in the
    List
    for i in range(len(models)):

        #Start prediction for each row by each
        model
        prediction = predict_list[i](models[i],
        row)

        #Store the prediction in the list
        stacked_row.append(prediction)

    #Append class values to the new row
    stacked_row.append(row[-1])

    #Extend the old row aby adding stacked row
    return row[0:len(row)-1]
```

On some predictive modeling problems, it is

possible to get an even larger boost by training the aggregated model on both the training row and the predictions made by submodels.

This improvement gives the aggregator model both the context of all of the data in the training row to help determine how and when to best combine the predictions of the submodels.

We can update our `to_stacked_row()` function to include this by aggregating the training row (minus the final column) and the stacked row as created previously.

The following is an updated version of the `to_stacked_row()` function that implements this improvement:

```
# Make predictions with sub-models and
construct a new stacked row
def to_stacked_row(models, predict_list, row):

    #Let's Create an empty list to store
    predictions from sub models
    stacked_row = list()

    #Run a loop to fetch stored models in the
```

```

List
    for i in range(len(models)):

        #Start prediction for each row by each
model
        prediction = predict_list[i](models[i],
row)

        #Store the prediction in the list
        stacked_row.append(prediction)

        #Append class values to the new row
        stacked_row.append(row[-1])

        #Extend the old row aby adding stacked row
return row[0:len(row)-1] + stacked_row

```

It is a good idea to try both approaches to your problem to see which works best.

Now that we have all of the pieces for stacked generalization, we can apply it to a real-world problem.

Practical application

– Sonar dataset (Mine and Rock prediction)

We will use a publicly available dataset of sonar signal returns from different surfaces; the dataset has 208 observations and 60 features to classify the instances into two groups mine (M) and rock (R). The variables are in the range of 0 to 1.

Here is the detail of the dataset:

```
Location: https://archive.ics.uci.edu/ml/dataset/s/Connectionist+Bench+\(Sonar,+Mines+vs.+Rocks\);  
Data set Name: Connectionist Bench (Sonar, Mines vs. Rocks) Data Set  
Number of Instances: 208  
Attributes characteristics: Float  
Number of attributes: 60  
Number of classes: 2
```


More information about the dataset

The file `sonar.mines` contains 111 patterns obtained by bouncing sonar signals off a metal cylinder at various angles and under various conditions. The file `sonar.rocks` contains 97 patterns obtained from rocks under similar conditions. The transmitted sonar signal is a frequency-modulated chirp, rising in frequency. The dataset contains signals obtained from a variety of different aspect angles, spanning 90 degrees for the cylinder and 180 degrees for the rock. Each pattern is a set of 60 numbers in the range 0.0 to 1.0. Each number represents the energy within a particular frequency band, integrated over a certain period of time. The integration aperture for higher frequencies occurs later in time since these frequencies are transmitted later during the chirp. The label associated

with each record contains the letter *R* if the object is a rock and *M* if it is a mine (metal cylinder). The numbers in the labels are in the increasing order of the aspect angle but they do not encode the angle directly.

The example assumes that a CSV copy of the dataset is in the current working directory with the filename `sonar.all-data.csv`.

The dataset is first loaded, the string values converted to numeric, and the output column is converted from strings to the integer values of 0 to 1. This is achieved with helper functions `load_csv()`, `str_column_to_float()` and `str_load`, and the dataset is prepared.

We will use the *k*-fold cross-validation to estimate the performance of the learned model on the unseen data. This means that we will construct and evaluate *k* models and estimate the performance as the mean model error. Classification accuracy will be used to evaluate the model. These behaviors are provided in

the `cross_validation_split()`, `accuracy_metric()`, and `evaluate_algorithm()` helper functions. We have used these helper functions multiple times, so I don't think that it is required to discuss this functions working here; you can get the full code at the end of the chapter.

We will use the KNN, perceptron, and logistic regression algorithms implemented previously. We will also use our technique for creating the new stacked dataset defined in the previous step.

A new function name `stacking()` is developed. This function does four things:

1. It first trains a list of models (KNN and perceptron)
2. It then uses the models to make predictions and create a new stacked dataset
3. It then trains an aggregator model (logistic regression) on the stacked dataset
4. It then uses the submodels and the

aggregator model to make predictions on the test dataset

```
# Stacked Generalization Algorithm
def stacking(train, test):

    #Let's define the sub model first
    model_list = [knn_model, perceptron_model]

    #We will create a prediction list to create
    new row
    predict_list = [knn_predict,
perceptron_predict]

    #Create an empty list to store the trained
    models
    models = list()

    #Lets train each sub model individually on
    the dataset
    for i in range(len(model_list)):
        model = model_list[i](train)
        models.append(model)

    #Create a new stacked data set from
    prediction of sub models
    stacked_dataset = list()
    for row in train:

        #Get new row
        stacked_row = to_stacked_row(models,
predict_list, row)

        #Append it to new dataset
        stacked_dataset.append(stacked_row)

    #We will train our final classifier on the
    stacked dataset
    stacked_model =
    logistic_regression_model(stacked_dataset)
```

```

    #lets create a list of prediction of the
    stacked output
    predictions = list()

    #Here we will combine all the classifier
    together to make stack of
    classifiers
    for row in test:

        #Get new row from prediction of sub
        models
        stacked_row = to_stacked_row(models,
        predict_list, row)

        #Append new row to the new dataset
        stacked_dataset.append(stacked_row)

        #Classify the new row using final
        classifier
        prediction =
        logistic_regression_predict(stacked_model,
        stacked_row)

        #As final classifier gives a continuous
        value round it to nearest integer
        prediction = round(prediction)

        #Append the prediction to the final
        list of predictions
        predictions.append(prediction)
    return predictions

```

Now, it is time to put it all together and see the results:

```

# Test stacking on the sonar dataset
seed(1)
# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)

```

```
# convert string attributes to integers
for i in range(len(dataset[0])-1):
    str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
n_folds = 5
scores = evaluate_algorithm(dataset, stacking,
n_folds)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' %
(sum(scores)/float(len(scores))))
```

A k value of 5 was used for cross-validation, giving each fold $208/3 = 69.3$ or just under 70 records to be evaluated upon each iteration.

I evaluated each submodel separately on the training dataset and achieved 74.879% for KNN and 72.464% for the perceptron algorithm; both algorithms are better than the baseline accuracy of 53%.

Training logistic regression on the stacked dataset without the training rows for context achieves an accuracy of 75.845%. Including the training rows in the stacked dataset gives a further bump to 76.329%.

Running the example prints the scores and mean of the scores for the final configuration:

Scores: [65.85365853658537, 85.36585365853658,
75.60975609756098, 82.92682926829268,
75.60975609756098]
Mean Accuracy: 77.073%

Summary

So, we have successfully implemented our stacked generalization problem. We started with a simple introduction to the stacking process. As this chapter was towards the introduction, we covered linear classifiers in the combination; we created a stack of three classifiers. Two of them were submodels and the last one was an aggregator that was responsible for combining the results of the previous two to generate the final prediction. During the process, you learned two very useful classification models—perceptron and logistic regression. As the perceptron is the basic building block of an ANN, it is a widely used concept in the current trends of machine learning algorithms. We saw how gradient descent algorithm can help us train a single perceptron for a prediction purpose. Later in the chapter, we saw the core concepts of logistic regression, which is a regression

algorithm that can be used for classification purposes, too. It is an extension of the linear regression algorithm by inducing nonlinearity using the logistic function. We trained our first logistic regression model using the same gradient descent as it is quite easy to understand and implement; finally, we created a stack and applied it to a practical dataset successfully.

Now, what next? I encourage you to keep practicing the concepts you learned in this chapter, as stacking is the most used ensemble algorithm. It allows you to handle almost every complexity in datasets. I encourage you to explore more algorithms to optimize cost functions such as the least of squares and the Quasi-Newton methods, and always keep an eye on the available online resources regarding the topic.

Stacked Generalization – Part 2

In the previous chapter, we saw how to do stacking of multiple classifiers using a combination of linear (perceptron and logistic regression) and nonlinear (KNN) algorithms. In this chapter, we will implement the stacking of nonlinear classifiers as well as learn how to improve a classifier's accuracy by selecting features that are more significant in the particular dataset.

We will start with feature selection first. I want to clarify to you about this topic; it is directly not related to model stacking, but we will build a solution with the use of a classifier to select the features, and then we will classify the modified dataset using another classifier. Although the process looks

like model stacking, it is not a conventional model stacking process. However, we can reduce the complexity of our data as well as classifier by this procedure.

So, let's start with the feature selection process first; then, we will proceed to the stacking of nonlinear algorithms.

Feature selection

Feature selection is also known as **attribute selection** in the machine learning community. As it is quite clear from its name itself, it is the process of selecting attributes from a dataset that are more significant in the decision-making process. In other words, we train our classifier only on those features that reduce the correlation between the features so that we can avoid redundancy in our dataset.

The following figure shows a generalized framework for any machine learning system, where you can see that feature selection appears just before classifier training:

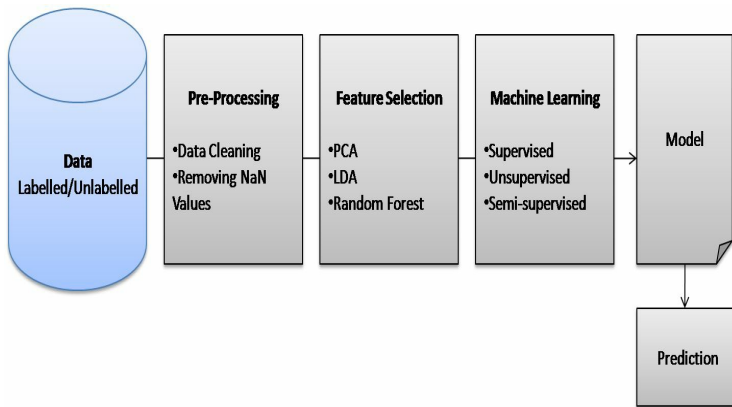


Figure 9.1: A general machine learning framework

Let's understand the feature selection process. Suppose you want to classify animals, for example, based on a plethora of relevant collected data. You quickly realize that all sorts of potential data attributes, or features, are relatively unhelpful for classification. For example, given that most living creatures have precisely one heart, this particular feature would not be beneficial from a learning perspective. On the other hand, an attribute denoting whether a given animal is hooved or not would likely be a powerful predictor.

Further, using all of these irrelevant attributes mixed with powerful predictors may actually have a negative effect on the resulting model. And we need not say anything about the increased training times that may come along with the inclusion of useless attributes, or the overfitting that may occur on the training data.

In real-world studies, a dataset always contains noise in it, and this noise can affect the performance of the classifier. Because noise is a random variable, we can't model it and that is why we have to somehow remove it from the process. So, how do we find such variables that are redundant in the process? Well, we will discuss that in some time. First, let's see why we need to do feature selection.

Why feature selection?

Feature selection is the process of narrowing down a subset of features, or attributes, to be used in the predictive modeling process. Feature selection is useful on a variety of fronts. It is the best weapon against the curse of dimensionality. It can reduce overall training times, and it is a powerful defense against overfitting, increasing model generalization. Mainly, there are the following reasons to go for the feature selection process:

- Simplification of models to make them easier to interpret by researchers/users
- Shorter training times
- To avoid the curse of dimensionality
- Enhanced generalization by reducing overfitting or high variance

The central premise when using a feature selection technique is that the data contains many features that are either redundant or irrelevant, and thus it can be removed without incurring much loss of information.

Redundant and irrelevant features are two distinct notions, since one relevant feature may be redundant in the presence of another relevant feature with which it is strongly correlated.



Feature selection techniques should be distinguished from feature extraction.

Feature extraction creates new features from functions of the original features, whereas feature selection returns a subset of features. Feature selection techniques are often used in domains where there are many features and comparatively few samples (or data points). Archetypal cases for the application of feature selection include analyses of written texts and DNA microarray data, where there are many thousands of features and a few tens

to hundreds of samples.

Let's discuss the reasons behind feature selection in a bit more detail.

Simplification of models

It's quite intuitive—as feature selection reduces the size of dataset by selecting the significant attributes only, it directly affects the model complexity. Let's understand this by an example.

I will use the iris classification dataset available with the `sklearn` library. I will also use `sklearn` decision tree algorithm to demonstrate the impact of feature reduction on tree architecture.

Dataset information

This is perhaps the best known database to be found in pattern recognition literature.

Fisher's paper is a classic in this field and is referenced frequently to this day. The dataset contains three classes of 50 instances each, where each class refers to a type of iris plant. One class is linearly separable from the other two; the latter are not linearly separable from each other.

Predicted attribute

Class of an iris plant

This is an exceedingly simple domain. This data differs from the data presented in Fisher's article. The 35th sample should be 4.9, 3.1, 1.5, 0.2, Iris Setosa, where the error is in the fourth feature. The 38th sample is 4.9, 3.6, 1.4, 0.1, Iris Setosa, where the errors are in the second and third features.

Attribute information

Following are the attributes in the dataset:

- Sepal length in cm
- Sepal width in cm
- Petal length in cm
- Petal width in cm
- **Class:** *Iris Setosa*, *Iris Versicolour*, and *Iris Verginica*

We will use the `pydotplus` library to create a graph out of the trained tree; so if you want to run the code on your system, you need to download the library from Python packages using `pip`.

First, we will create a tree using all of the features in the dataset. The code goes like this:

```
#Import Sklearn Datasets of IRIS flower classification  
import sklearn.datasets as datasets  
  
#Import Pandas library to create data frame from the data  
import pandas as pd  
  
#Load the dataset  
iris=datasets.load_iris()  
  
#Extract data part from the dataset  
data = iris.data  
  
#Select dimension of data  
data = data[:,0:4]  
  
#Load dataset into the data frame  
df=pd.DataFrame(data)  
  
#Extract target variable from the dataset  
y=iris.target  
  
#Import decision tree classifier from sklearn  
from sklearn.tree import DecisionTreeClassifier  
  
#We will create a tree with maximum depth of 5, other parameters will be default  
dtree=DecisionTreeClassifier(max_depth=5)  
  
#Train the classifier  
dtree.fit(df,y)  
  
#Import graphviz from sklearn to create the graph out of tree  
from sklearn.tree import export_graphviz  
  
#We will use StringIO to create graph with all characters  
from sklearn.externals.six import StringIO  
dot_data = StringIO()  
  
#Import pydotplus to create tree as a graph and
```

```
store it on the disk
import pydotplus

#Create Graph out of tree and store it on the
disk
export_graphviz(dtree, out_file=dot_data,
                filled=True, rounded=True,
                special_characters=True)
graph =
pydotplus.graph_from_dot_data(dot_data.getvalue(
graph.write_png("graph_feat_4.png")
```

After execution, it should generate a graph for the build tree, which will be stored as a PNG image in your local directory. It should look like this:



Figure 9.2: Tree with four features

The `value` row in each node tells us how many of the observations that were sorted into that node fall under each of our three categories. We can see that our feature X_2 , which is the petal length, was able to completely distinguish one species of flower (*Iris setosa*) from the rest.

Now we will select the last two columns of the feature by changing the line like this:

```
| #Select dimension of data  
| data = data[:,2:4]  
| Now graph will be look like;
```

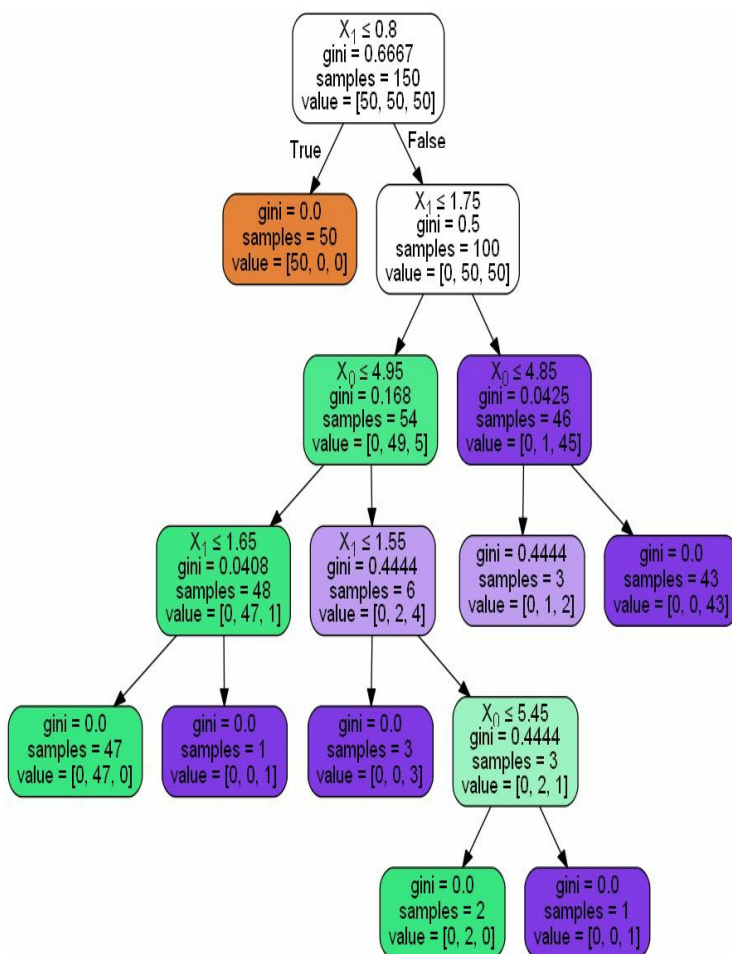


Figure 9.3: Tree with selected two features

Now, if you will look closely, at depth three, the number of children in the second graph is

less than in the first graph. This is because there was no split found in the case of less features. One important point should be noted here—this simplification does not happen due to reduction in feature length; it happens because we have selected the correct features that contribute more to the final decision than others.

Shorter training time

We have already trained decision trees in previous chapters, where we saw that the number of features is inversely proportional to the training time. If we take the example of decision trees, we know that to find the best split, we must go through each attribute value and test it for the candidate of the node. So, if the number of attributes is less, the time to fit a tree will also be less, as there are fewer candidates available for node selection.

We will see the timing impact using the `otto` dataset, which we have used earlier in the XGBoost library application. The dataset contains 95 attributes and more than 61,000 instances; we will use 50000 instances to build our model. We will fit a random forest classifier. First we will build the model for all 95 attributes. Then we will train the model with 40 features and see what kind of output we get at the end:

```
#Import the supporting libraries  
#Import pandas to load the dataset from csv  
file  
from pandas import read_csv  
  
#Import numpy for array based operations and  
calculations  
import numpy as np  
  
#Import Random Forest classifier class from  
sklearn  
from sklearn.ensemble import  
RandomForestClassifier  
  
#Load dataset as pandas data frame  
data = read_csv('train.csv')  
  
#Extract attribute names from the data frame  
feat = data.keys()  
feat_labels = feat.get_values()  
  
#Extract data values from the data frame  
dataset = data.values  
  
#Shuffle the dataset  
np.random.shuffle(dataset)  
  
#We will select 50000 instances to train the  
classifier  
inst = 50000  
  
#Extract 50000 instances from the dataset  
dataset = dataset[0:inst,:]  
  
#Split data into input and output variable with  
selected features  
Xtrain = dataset[:,1:40]  
ytrain = dataset[:,94]  
  
# Create a random forest classifier with the  
following Parameters  
trees      = 250  
max_feat   = 7
```

```

max_depth = 30
min_sample = 2

clf =
RandomForestClassifier(n_estimators=trees,

max_features=max_feat,

max_depth=max_depth,
min_samples_split=
min_sample,
random_state=0,
n_jobs=-1)

# Train the classifier and calculate the
training time
import time
start = time.time()
clf.fit(Xtrain, ytrain)
end = time.time()

print("Execution time for building the Tree is:
%f"%(float(end)-float(start)))

```

This is the time to execute the code; for all of the features, we are getting the total training time as:

```

Execution time for building the Tree is:
16.887000

```

And when we change the number of the attribute to 40, we get the following timings:

```

Execution time for building the Tree is:
13.759000

```

So, as you can see, we are getting a difference of approximately three seconds in the training of the same classifier but with a modified dataset.

To avoid the curse of dimensionality

Suppose, you drop a coin on a 100-meter line; how do you find it? Simple, just walk along the line and search. But what if it's a *100x100* sq. m. field? It's already tough trying to search a (roughly equivalent) football ground for a single coin. Furthermore, what if it's a *100x100x100* cu. m. space?! Your football ground now has a 30-storeyed height. Good luck finding a coin there! That, in essence is the **curse of dimensionality**.

The curse of dimensionality, a term initially introduced by Richard Bellman (an American applied mathematician), is a phenomena that arises when applying machine learning algorithms to highly dimensional data.

What do we mean by dimensions of data?
Well, dimensions are nothing but the number

of attributes in a dataset. In the simplest form, we can say that if we have many features in our dataset (with redundancy), then it is a bit difficult to get an optimum solution from the classifier algorithm because the number of features increases the complexity of the model (as we have seen earlier):

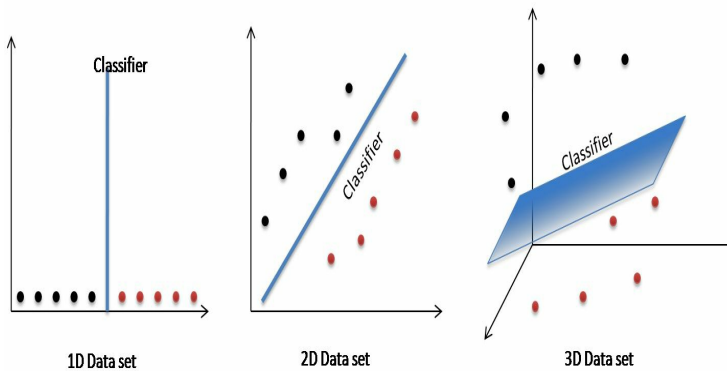


Figure 9.4: Data in different dimensions

The preceding figure shows how the complexity of a classifier depends on the dimensions of the dataset. When there is a one-dimensional dataset, a simple threshold can be used as a classifier to segment the instances into different groups. If we increase

the dimensions to two, the separation between the classes needs to be classified with the decision line. For a three-dimensional dataset classifier, we will need to fit a decision plan to separate the instances in different classes. Thus, for an n -dimensional dataset, our classifier's complexity will also be of the n^{th} order.

So when do the dimensions of data become difficult to handle? Well, many ML methods use distance measures.

Most segmentation and clustering methods rely on computing distances between observations. The well-known k-means segmentation assigns points to the nearest center. DBSCAN and hierarchical clustering also required distance metrics. Distribution and density-based outlier detection algorithms also make use of distance relative to other distances to mark outliers.

Supervised classification solutions such as KNN's method also use distances between

observations to assign a class to an unknown observation. The **Support Vector Machine (SVM)** method involves transforming observations around select kernels based on the distance between the observation and the kernel.

Let's see how distance plays havoc in higher dimensions.

Many algorithms measure distance between two data points to define some sort of nearness (DBSCAN, kernels, and KNN) in reference to some predefined distance threshold. In two dimensions, we can imagine that two points are near if one falls within a certain radius of another. Consider the left-hand-side image in the following figure. What share of uniformly spaced points within the black square fall inside the red circle? That is about:

$$\frac{\pi r^2}{(2r^2)} = \frac{\pi}{4} = 78.5\%$$

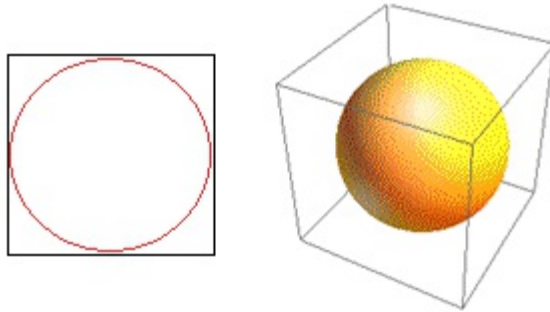


Figure 9.5: Data in different dimensions

So if you fit the biggest circle possible inside a square, you cover 78.5% of the square. Yet, the biggest sphere possible inside a cube covers only:

$$\frac{\frac{4}{3}\pi r^3}{(2r)^3} = \frac{\pi}{6} = 52\%$$

of the cube's volume! This volume reduces exponentially to 0.24% for just 10 dimensions! What this essentially means is that in a high-dimensional world, every single data point is at the corners and nothing really is in the center of the volume. In other words, the center volume reduces to nothing because there is (almost) no center! This has huge

consequences on distance-based clustering algorithms. All distances start looking the same and any distance more or less than any other is more of a random fluctuation in data rather than a measure of dissimilarity!

Apart from distances and volumes, the number of dimensions creates other practical problems too. Solution runtime and system memory requirements often escalate nonlinearly with an increase in the number of dimensions. Due to exponential increases in feasible solutions, many optimization methods cannot reach the global optima and have to make do with a local optima. Further, instead of a closed-form solution, the optimization must use search-based algorithms such as gradient descent, genetic algorithm, and simulated annealing. More dimensions introduce the possibility of correlation, and parameter estimation can become difficult in regression approaches.

So, the bottom line is that selecting useful features can reduce the number of variables

in the dataset, which eventually reduces the complexity (dimensions) of the dataset. Also, it becomes quite easy to fit a model on the modified dataset.

Enhanced generalization by reducing overfitting

Keeping irrelevant attributes in your dataset can result in overfitting. Decision tree algorithms such as C4.5 seek to make optimal splits in attribute values. Those attributes that are more correlated with the prediction are split on first. Deeper in the tree, less relevant and irrelevant attributes are used to make prediction decisions that may only be beneficial by chance in the `training` dataset. This overfitting of the training data can negatively affect the modeling power of the method and cripple the predictive accuracy.

We will see the code example for this point in the next section, where we will use feature selection by the random forest algorithm.

Feature selection for machine learning

In this section, we will see different methods to select features from the dataset; we will discuss the following types of feature selection algorithms and their implementation in Python using the Scikit-learn (`sklearn`) library:

- Univariate selection
- **Recursive Feature Elimination (RFE)**
- **Principle Component Analysis (PCA)**
- Choosing important features (feature importance)

We will discuss the first three algorithms and their implementation in short. We will discuss the *Choosing important features (feature importance)* part in more detail because it is a widely used technique in the data science community.

Univariate selection

Statistical tests can be used to select those features that have the strongest relationships with the output variable.

The scikit-learn library provides the `SelectKBest` class, which can be used with a suite of different statistical tests to select a specific number of features.

The following example uses the chi squared (χ^2) statistical test for non-negative features to select four of the best features from the Pima Indians onset of diabetes dataset:

```
#Feature Extraction with Univariate Statistical  
#Tests (Chi-squared for classification)  
#Import the required packages  
#Import pandas to read csv  
import pandas  
  
#Import numpy for array related operations  
import numpy  
  
#Import sklearn's feature selection algorithm
```

```
from sklearn.feature_selection import
SelectKBest

#Import chi2 for performing chi square test
from sklearn.feature_selection import chi2

#URL for loading the dataset
url = "https://archive.ics.uci.edu/ml/machine-
learning-databases/pima-indians
diabetes/pima-indians-diabetes.data"

#Define the attribute names
names = ['preg', 'plas', 'pres', 'skin',
'test', 'mass', 'pedi', 'age', 'class']

#Create pandas data frame by loading the data
from URL
dataframe = pandas.read_csv(url, names=names)

#Create array from data values
array = dataframe.values

#Split the data into input and target
X = array[:,0:8]
Y = array[:,8]

#We will select the features using chi square
test = SelectKBest(score_func=chi2, k=4)

#Fit the function for ranking the features by
score
fit = test.fit(X, Y)

#Summarize scores
numpy.set_printoptions(precision=3)
print(fit.scores_)

#Apply the transformation on to dataset
features = fit.transform(X)

#Summarize selected features
print(features[0:5,:])
```

You can see the scores for each attribute and the four attributes chosen (those with the highest scores): plas, test, mass, and age.

Scores for each feature:

[111.52	1411.887	17.605	53.108
2175.565	127.669	5.393	181.304]

Selected features:

[148.	0.	33.6	50.]
[85.	0.	26.6	31.]
[183.	0.	23.3	32.]
[89.	94.	28.1	21.]
[137.	168.	43.1	33.]]

Recursive Feature Elimination

RFE works by recursively removing attributes and building a model on attributes that remain.

It uses model accuracy to identify which attributes (and combinations of attributes) contribute the most to predicting the target attribute.

You can learn more about the RFE class in the [scikit-learn documentation](#).

The following example uses RFE with the logistic regression algorithm to select the top three features. The choice of algorithm does not matter too much as long as it is skillful and consistent:

```
| #Import the required packages  
| #Import pandas to read csv
```

```
import pandas

#Import numpy for array related operations
import numpy

#Import sklearn's feature selection algorithm
from sklearn.feature_selection import RFE

#Import LogisticRegression for performing chi
square test
from sklearn.linear_model import
LogisticRegression

#URL for loading the dataset
url = "https://archive.ics.uci.edu/ml/machine-
learning-databases/pima-indians-diabetes/pima-
indians-diabetes.data"

#Define the attribute names
names = ['preg', 'plas', 'pres', 'skin',
'test', 'mass', 'pedi', 'age', 'class']

#Create pandas data frame by loading the data
from URL
dataframe = pandas.read_csv(url, names=names)

#Create array from data values
array = dataframe.values

#Split the data into input and target
X = array[:,0:8]
Y = array[:,8]

#Feature extraction
model = LogisticRegression()
rfe = RFE(model, 3)
fit = rfe.fit(X, Y)

print("Num Features: %d"% fit.n_features_)
print("Selected Features: %s"% fit.support_)
print("Feature Ranking: %s"% fit.ranking_)
```

After execution, we will get:

```
| Num Features: 3  
| Selected Features: [ True False False False  
| False  True  True False]  
| Feature Ranking: [1 2 3 5 6 1 1 4]
```

You can see that RFE chose the top three features as `preg`, `mass`, and `pedi`.

These are marked `True` in the `support_` array and marked with a choice 1 in the `ranking_` array.

Principle Component Analysis

PCA uses linear algebra to transform the dataset into a compressed form.

Generally, it is considered a data reduction technique. A property of PCA is that you can choose the number of dimensions or principal components in the transformed result.

In the following example, we use PCA and select three principal components:

```
#Import the required packages
#Import pandas to read csv
import pandas

#Import numpy for array related operations
import numpy

#Import sklearn's PCA algorithm
from sklearn.decomposition import PCA

#URL for loading the dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/pima-indians-diabetes/pima-indians-diabetes.data"
```

```

#Define the attribute names
names = ['preg', 'plas', 'pres', 'skin',
'test', 'mass', 'pedi', 'age', 'class']

#Create pandas data frame by loading the data from URL
dataframe = pandas.read_csv(url, names=names)

#Create array from data values
array = dataframe.values

#Split the data into input and target
X = array[:,0:8]
Y = array[:,8]

#Feature extraction
pca = PCA(n_components=3)
fit = pca.fit(X)

#Summarize components
print("Explained Variance: %s") %
fit.explained_variance_ratio_
print(fit.components_)

```

You can see that the transformed dataset (three principal components) bears little resemblance to the source data:

```

Explained Variance: [ 0.88854663  0.06159078
0.02579012]
[[ -2.02176587e-03   9.78115765e-02
 1.60930503e-02   6.07566861e-02
 9.93110844e-01   1.40108085e-02   5.37167919e-
04  -3.56474430e-03]
 [ -2.26488861e-02  -9.72210040e-01
-1.41909330e-01   5.78614699e-02
 9.46266913e-02  -4.69729766e-02  -8.16804621e-
04  -1.40168181e-01
 [ -2.24649003e-02   1.43428710e-01

```



```
| -9.22467192e-01  -3.07013055e-01  
| 2.09773019e-02  -1.32444542e-01  -6.39983017e-  
04  -1.25454310e-01]]
```

Choosing important features (feature importance)

Feature importance is the technique used to select features using a trained supervised classifier. When we train a classifier such as a decision tree, we evaluate each attribute to create splits; we can use this measure as a feature selector. Let's understand it in detail.

Random forests are among the most popular machine learning methods thanks to their relatively good accuracy, robustness, and ease of use. They also provide two straightforward methods for feature selection — **mean decrease impurity** and **mean decrease accuracy**.

A random forest consists of a number of decision trees. Every node in a decision tree

is a condition on a single feature, designed to split the dataset into two so that similar response values end up in the same set. The measure based on which the (locally) optimal condition is chosen is known as **impurity**. For classification, it is typically either the Gini impurity or information gain/entropy, and for regression trees, it is the variance. Thus when training a tree, it can be computed by how much each feature decreases the weighted impurity in a tree. For a forest, the impurity decrease from each feature can be averaged and the features are ranked according to this measure.

Let's see how to do feature selection using a random forest classifier and evaluate the accuracy of the classifier before and after feature selection.

We will use the `otto` dataset, which we have used earlier in [Chapter 7](#), *XGBoost – eXtreme Gradient Boosting*, where we discussed the XGBoost library.

This dataset is available for free from kaggle (you will need to sign up to kaggle to be able to download this dataset). You can download training dataset, `train.csv.zip`, from the <https://www.kaggle.com/c/otto-group-product-classification-challenge/data> and place the unzipped `train.csv` file in your working directory.

This dataset describes 93 obfuscated details of more than 61,000 products grouped into 10 product categories (for example, fashion, electronics, and so on). Input attributes are the counts of different events of some kind.

The goal is to make predictions for new products as an array of probabilities for each of the 10 categories, and models are evaluated using multiclass logarithmic loss (also called cross entropy).

We will start with importing all of the libraries:

```
#Import the supporting libraries
#Import pandas to load the dataset from csv
file
```

```

from pandas import read_csv

#Import numpy for array based operations and calculations
import numpy as np

#Import Random Forest classifier class from sklearn
from sklearn.ensemble import
RandomForestClassifier

#Import feature selector class select model of sklearn
from sklearn.feature_selection import
SelectFromModel

np.random.seed(1)

```

Let's define a method to split our dataset into training and testing data; we will train our dataset on the training part and the testing part will be used for evaluation of the trained model:

```

#Function to create Train and Test set from the original dataset
def getTrainTestData(dataset,split):
    np.random.seed(0)
    training = []
    testing = []

    np.random.shuffle(dataset)
    shape = np.shape(dataset)
    trainlength =
np.uint16(np.floor(split*shape[0]))

    for i in range(trainlength):
        training.append(dataset[i])

```

```

    for i in range(trainlength, shape[0]):
        testing.append(dataset[i])
        training = np.array(training)
        testing = np.array(testing)
    return training, testing

```

We also need to add a function to evaluate the accuracy of the model; it will take the predicted and actual output as input to calculate the percentage accuracy:

```

#Function to evaluate model performance
def getAccuracy(pre, ytest):
    count = 0
    for i in range(len(ytest)):
        if ytest[i]==pre[i]:
            count+=1
    acc = float(count)/len(ytest)
    return acc

```

This is the time to load the dataset. We will load the `train.csv` file; this file contains more than *61,000* training instances. We will use *50000* instances for our example, in which we will use *35,000* instances to train the classifier and *15,000* instances to test the performance of the classifier:

```

#Load dataset as pandas data frame
data = read_csv('train.csv')

#Extract attribute names from the data frame

```

```

feat = data.keys()
feat_labels = feat.get_values()

#Extract data values from the data frame
dataset = data.values

#Shuffle the dataset
np.random.shuffle(dataset)

#We will select 50000 instances to train the classifier
inst = 50000

#Extract 50000 instances from the dataset
dataset = dataset[0:inst,:]

#Create Training and Testing data for performance evaluation
train,test = getTrainTestData(dataset, 0.7)

#Split data into input and output variable with selected features
Xtrain = train[:,0:94]
ytrain = train[:,94]
shape = np.shape(Xtrain)
print("Shape of the dataset ",shape)

#Print the size of Data in MBs
print("Size of Data set before feature
selection: %.2f MB"%(Xtrain.nbytes/1e6))

```

Let's take note of the data size here; as our dataset contains about 35000 training instances with 94 attributes; the size of our dataset is quite large. Let's see:

```

Shape of the dataset (35000, 94)
Size of Data set before feature selection:
26.32 MB

```

As you can see, we are having 35000 rows and 94 columns in our dataset, which is more than 26 MB data.

In the next code block, we will configure our random forest classifier; we will use 250 trees with a maximum depth of 30 and the number of random features will be 7. Other hyperparameters will be the default of sklearn:

```
#Lets select the test data for model evaluation
purpose
Xtest = test[:,0:94]
ytest = test[:,94]

#Create a random forest classifier with the
following Parameters
trees      = 250
max_feat   = 7
max_depth  = 30
min_sample = 2
clf =
RandomForestClassifier(n_estimators=trees,

max_features=max_feat,

max_depth=max_depth,

min_sample,                                min_samples_split=

                                         random_state=0,
                                         n_jobs=-1)

#Train the classifier and calculate the
training time
import time
start = time.time()
```



```
clf.fit(Xtrain, ytrain)
end = time.time()

#Lets Note down the model training time
print("Execution time for building the Tree is:
%f"%(float(end)-float(start)))
pre = clf.predict(Xtest)
```

Let's see how much time is required to train the model on the training dataset:

```
Execution time for building the Tree is:
2.913641

#Evaluate the model performance for the test
data
acc = getAccuracy(pre, ytest)
print("Accuracy of model before feature
selection is %.2f"%(100*acc))
```

The accuracy of our model is:

```
Accuracy of model before feature selection is
98.82
```

As you can see, we are getting very good accuracy as we are classifying almost 99% of the test data into the correct categories. This means we are classifying about 14,823 instances out of 15,000 in correct classes.

So, now my question is: should we go for further improvement? Well, why not? We

should definitely go for more improvements if we can; here, we will use feature importance to select features. As you know, in the tree building process, we use impurity measurement for node selection. The attribute value that has the lowest impurity is chosen as the node in the tree. We can use similar criteria for feature selection. We can give more importance to features that have less impurity, and this can be done using the `feature_importances_` function of the `sklearn` library. Let's find out the importance of each feature:

```
#Once we have trained the model we will rank
all the features
for feature in zip(feats_labels,
clf.feature_importances_):
    print(feature)

('id', 0.33346650420175183)
('feat_1', 0.0036186958628801214)
('feat_2', 0.0037243050888530957)
('feat_3', 0.011579217472062748)
('feat_4', 0.010297382675187445)
('feat_5', 0.0010359139416194116)
('feat_6', 0.00038171336038056165)
('feat_7', 0.0024867672489765021)
('feat_8', 0.0096689721610546085)
('feat_9', 0.007906150362995093)
('feat_10', 0.0022342480802130366)
```

As you can see here, each feature has a different importance based on its contribution to the final prediction.

We will use these importance scores to rank our features; in the following part, we will select those features that have feature importance more than 0.01 for model training:

```
#Select features which have higher contribution  
in the final prediction  
sfm = SelectFromModel(clf, threshold=0.01)  
sfm.fit(Xtrain,ytrain)
```

Here, we will transform the input dataset according to the selected feature attributes. In the next code block, we will transform the dataset. Then, we will check the size and shape of the new dataset:

```
#Transform input dataset  
Xtrain_1 = sfm.transform(Xtrain)  
Xtest_1 = sfm.transform(Xtest)  
  
#Let's see the size and shape of new dataset  
print("Size of Data set before feature  
selection: %.2f MB"%(Xtrain_1.nbytes/1e6))  
shape = np.shape(Xtrain_1)  
print("Shape of the dataset ",shape)  
  
Size of Data set before feature selection: 5.60  
MB
```

| Shape of the dataset (35000, 20)

Do you see the shape of the dataset? We are left with only 20 features after the feature selection process, which reduces the size of the database from 26 MB to 5.60 MB. That's about 80% reduction from the original dataset.

In the next code block, we will train a new random forest classifier with the same hyperparameters as earlier and test it on the testing dataset. Let's see what accuracy we get after modifying the training set:

```
#Model training time
start = time.time()
clf.fit(Xtrain_1, ytrain)
end = time.time()
print("Execution time for building the Tree is:
%f"%(float(end)-float(start)))

#Let's evaluate the model on test data
pre = clf.predict(Xtest_1)
count = 0
acc2 = getAccuracy(pre, ytest)

print("Accuracy after feature selection %.2f"%
(100*acc2))

Execution time for building the Tree is:
1.711518
Accuracy after feature selection 99.97
```

Can you see that!! We have got 99.97 percent accuracy with the modified dataset, which means we are classifying 14,996 instances in correct classes, while previously we were classifying only 14,823 instances correctly.

This is a huge improvement we have got with the feature selection process; we can summarize all the results in the following table:

Evaluation criteria	Before feature selection	After feature selection
Number of features	94	20
Size of dataset	26.32 MB	5.60 MB
Training		

time	2.91 seconds	1.71 seconds
Accuracy	98.82 percent	99.97 percent

Table 9.1: A comparison of classifier performance on feature selection versus raw data

The preceding table shows the practical advantages of feature selection. You can see that we have reduced the number of features significantly, which reduces the model complexity and dimensions of the dataset. We are getting less training time after the reduction in dimensions, and at the end, we have overcome the overfitting issue, getting higher accuracy than before.

So, I think we have talked a lot about feature selection, and now it's time to come back to stacking. We are going to create a stack of nonlinear algorithms this time.

We will use an SVM as one of the classifiers in the ensemble stack; So let's start by

understanding SVMs.

Understanding the SVM

SVM are perhaps one of the most popular and talked about machine learning algorithms. An SVM is a supervised machine learning algorithm that can be used for both classification and regression tasks.

However, it is mostly used in classification problems. They were extremely popular around the time they were developed in the 1990s and continue to be the go-to method for a high-performing algorithm with little tuning.



*SVMs are also known as **maximal margin classifier**, **soft margin classifier**, **linear SVM**, and **kernel-based SVM**.*

All of these are different ways to use the classifiers; we will cover a basic

understanding of these flavors.

In this algorithm, we plot each data item as a point in n -dimensional space (where n is number of features we have), with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding a hyperplane that differentiates the two classes very well; the following figure shows the working of an SVM:

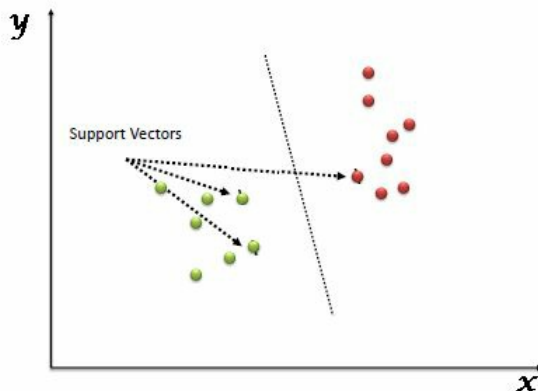


Figure 9.6: Support Vector Machine

Support Vectors are simply the coordinates of individual observation. The SVM is a frontier that best segregates the two classes

(hyperplane/line).

Now, the question is, how does it actually work? So, in the following section, we will start working on the development and implementation of the SVM.

How does SVM work?

Support vectors are simply the coordinates of individual observation. Let's understand this with the help of an example.

We have a population composed of 50% males and 50% females. Using a sample of this population, we want to create a set of rules that will guide us in the gender class for the rest of the population. Using this algorithm, we intend to build a robot that can identify whether a person is a male or a female. This is a sample problem of classification analysis. Using some set of rules, we will try to classify the population into two possible segments. For simplicity, let's assume that the two differentiating factors identified are the height of the individual and hair length. The following is a scatter plot of the sample:

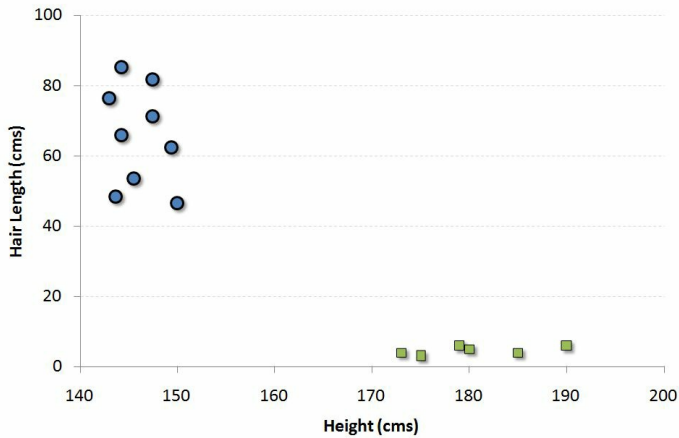


Figure 9.7: Data representation of males and females

The blue circles in the plot represent females and green squares represent males. A few expected insights from the graph are that:

- Males in our population have a higher average height
- Females in our population have longer scalp hairs

If we were to see an individual with height 180 cm and hair length 4 cm, our best guess would be to classify this individual as a male. This is how we do a classification analysis.

Now, as I have mentioned earlier, SVMs are the coordinates of individual observations; for instance, $(45, 150)$ is a support vector that corresponds to a female. SVM is a frontier that best segregates the males from the females. In this case, the two classes are well separated from each other; hence, it is easier to find an SVM.

Now the question is: how to find the frontiers? For the current example, the following figure shows three possible frontiers:

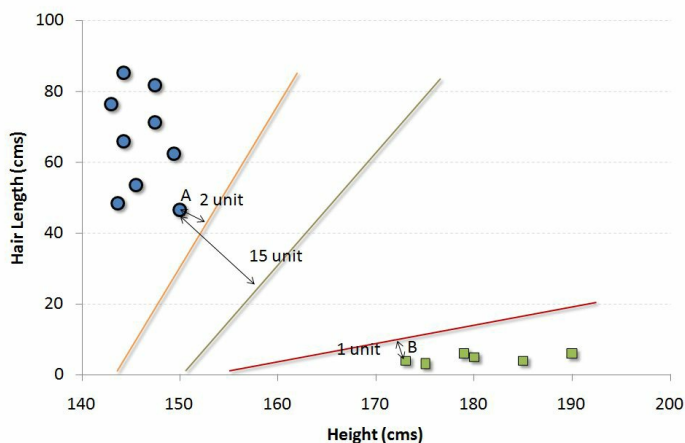


Figure 9.8: Different frontiers to separate data

So, what do you think? How do we decide which is the best frontier for this particular problem statement?

The easiest way to interpret the objective function in an SVM is to find the minimum distance of the frontier from the closest support vector (this can belong to any class). For instance, the orange frontier is closest to the blue circles and the closest blue circle is **2 units** away from the frontier. Once we have these distances for all frontiers, we simply choose the frontier with the maximum distance (from the closest support vector). Out of the three frontiers shown, we see that the black frontier is farthest from the nearest support vector (that is, **15 units**).

These frontiers are known as **hyperplanes**. What is a hyperplane?

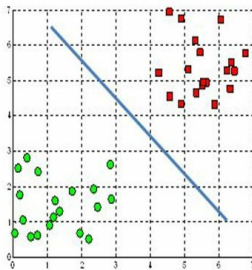
Hyperplane – separation between the data points

Geometry tells us that a hyperplane is a subspace of one dimension less than its ambient space. For instance, a hyperplane of an n -dimensional space is a flat subset with dimension $n-1$. By its nature, it separates the space into two half spaces. For machine learning tasks, we can reimagine hyperplanes as follows:

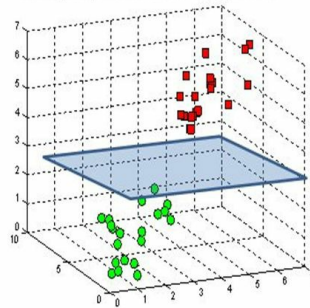
- A linear decision surface that splits a space into two parts
- A binary classifier

The following figure shows hyperplanes:

A hyperplane in \mathbb{R}^2 is a line



A hyperplane in \mathbb{R}^3 is a plane



A hyperplane in \mathbb{R}^n is an $n-1$ dimensional subspace

Figure 9.9: Hyperplane in linear separable data

This hyperplane will work very well for linear classification problems of N classes with M features; we can learn a mapping that is a linear combination. (such as $y = mx + b$) or even a multidimensional hyperplane ($y = x + z + b + q$). No matter how many dimensions/features a set of classes has, we can represent the mapping using a linear function.

But what if we get a nonlinear classification problem where the data is not linearly separable, such as a quadratic mapping?

Luckily for us, SVMs can efficiently perform a nonlinear classification using what is called the kernel trick. We will not talk about the kernel trick right now.

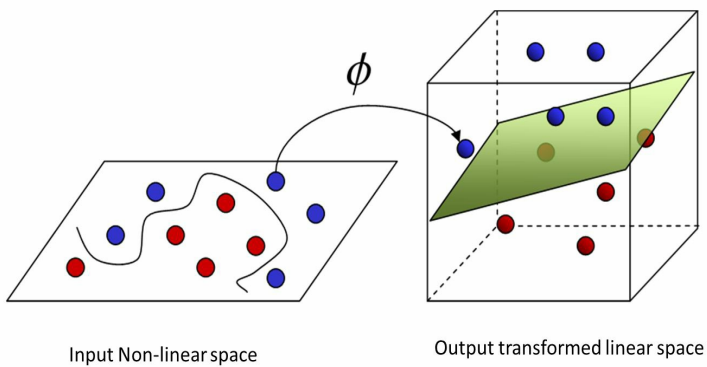


Figure 9.10: Hyperplane in nonlinear data

Implementation of an SVM

Let's start with an implementation of an SVM; we will formulate the concepts behind the SVM during the development process so that it will give you a clear understanding of the implementation.

Suppose we have some points on a graph; each point belongs to either a positive class or a negative class:

SN	X	Y	Class
1	-2	4	-1
2	4	1	-1
3	1	6	1

4	2	4	1
5	6	2	1

Table 9.2: Our toy dataset

Let's plot these points on a graph using Python:

```
#Import numpy to help us perform math
operations
import numpy as np

#Import matplotlib to plot our data and model
visually
from matplotlib import pyplot as plt

#Input data - Of the form [X value, Y value,
Bias term]
X = np.array([
    [-2,4, -1],
    [4,1, -1],
    [1, 6, -1],
    [2, 4, -1],
    [6, 2, -1],
])
```

If you guys look closely, you'll see that we have added a bias value column to the original dataset; although it is not necessary

to add the bias to the dataset, it is a good practice to do so:

```
#Associated output labels - First 2 examples  
are labelled '-1' and last 3 are labelled '+1'  
y = np.array([-1,-1,1,1,1])  
  
#lets plot these examples on a 2D graph!  
#for each example  
for d, sample in enumerate(X):  
  
# Plot the negative samples (the first 2)  
    if d < 2:  
        plt.scatter(sample[0], sample[1],  
s=120, marker='_', linewidths=2)  
  
        # Plot the positive samples (the last 3)  
    else:  
        plt.scatter(sample[0], sample[1],  
s=120, marker='+', linewidths=2)
```

After execution of the preceding code, we will get:

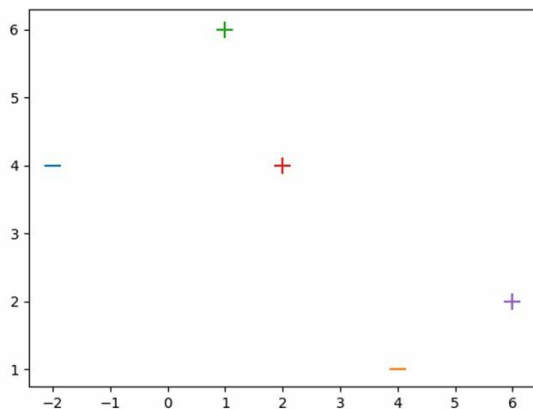


Figure 9.11 Positive and negative class

So you see, we have a linearly separable dataset. Let's put a line to separate these two classes by adding the following line to the code:

```
#Print a possible hyperplane, that is separating  
the two classes. #we'll two points and draw the  
line between them (naive guess)  
plt.plot([-2, 6], [6, 0.5])  
plt.show()
```

This will add a hyperplane to the preceding figure:

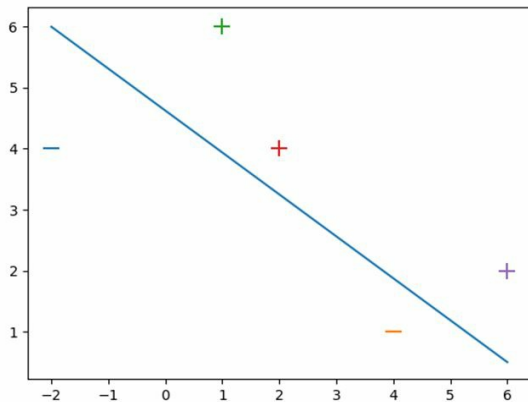


Figure 9.12: Hyperplane for separating two classes

Although we have created this line ourselves,

this is what we actually want from the algorithm at the end; now, we will see how to predict such a hyperplane by our algorithm. But before doing that, it is necessary to understand the underlying concepts of SVM.

What do you think? How can we get such a hyperplane? One way to do this is to select coordinates randomly to plot a line and then check the distance of the line with all of the points. If the distance is not optimal, we drop the coordinate and select a new pair again randomly until we get the best separation between the points and the line. This procedure of drawing a line and checking the separation is known as **Random Sample Consensus (RANSAC)**, but is a very time-consuming procedure and you never know when you will get the convergence. Do you know anything else?

Well, you know we can formulate this problem; we want to create a line between two classes. This line should classify input points into one of the classes by checking on

which side of the line the input point is lying.

Objective function

Let's consider our support vector as W , for example:

$$\bar{y} = \sum_j^M W * X + b \quad (1)$$

Where X is the input vector, y is the predicted output, and b is the bias. As we are working with the binary class problem, we want our output to be either $+1$ or -1 . We write it as:

$$\sum W * X + b = \pm 1 \quad (2)$$

The preceding equation tells us two things. To check the correctness (or loss) from our function, we have to choose a loss function so that it can indicate loss in a binary form; that is, if we are getting the correct output, we

should get $+1$ at the output, and if we are incorrect, we must get -1 . To make such a function, we need to modify the preceding function so that it can produce a binary output.

We can do this by adding a special kind of loss function to our classifier; this function is known as hinge loss. This loss can be written as:

$$c(x, y, f(x)) = (1 - y * f(x))_+ \quad (3)$$

Where c is the loss function, x is the input vector, y is the output vector (actual prediction), $f(x)$ is the predictor function ($W * X + b$), and $+$ denotes that this function will always have a positive value at the output. The preceding function has a special characteristic; if y and $f(x)$ are the same, then our loss will always be 0 . If y and $f(x)$ are different, then we will end up with a value less than 1 :

$$c(x, y, f(x)) = \begin{cases} 0 & \text{if } y * f(x) \geq 1 \\ 1 - y * f(x) & \text{else} \end{cases}$$

(4)

Now we can treat this problem as an optimization problem as we want to reduce the loss between the predicted and actual output of the classifier; so we can define our objective function for optimization as:

$$\sum_{i=1}^n (1 - y_i \langle x_i, w \rangle)_+$$

(5)

Doesn't this function look similar to the cost function we have used earlier in the book? If it is a similar kind of function, then what is the change that makes SVM a different classifier? Well, there is a big change in the formula of the objective function itself, and that is the regularization variable. This is the heart of the SVM algorithm; the regularizer balances between margin maximization and loss. We want to find the decision surface

that is maximally far away from any data points. We will rewrite our objective function with the regularizer term as:

$$\min_w \lambda \|w\|^2 + \sum_{i=1}^n (1 - y_i \langle x_i, w \rangle)_+ \quad (6)$$

As you can see, our objective in an SVM consists of two terms. The first term is a regularizer, the heart of the SVM; the second term is the loss.

If you have noticed, we have not included the bias term in our objective function. The reason? If you remember, we have already included the bias in our `toy` dataset. We will use a hardcoded bias to avoid complexity in our model. So, when we need to choose the number of weights, we will count the bias column as a feature and assign the weight for that.

Now, it's time for model learning and we will use the same algorithm for optimization of

our objective function as we have used to train GBM and perceptron. Yes, you guessed the name: gradient descent algorithm.

Function optimization

As we have discussed gradient descent a lot in the previous chapter, we will not emphasize much on it, but we will see how to change weights using it; as this time, the objective function is different, we have to derive the formula for the weight update rule.

Before going ahead, here is a small recap of the gradient descent algorithm. Gradient descent is the most popular optimization algorithm in the machine learning world; it works on the principle of motion. We assume our objective function as convex in nature and try to travel into the valley by updating the function parameter so that it follows the gradient of the function, which eventually helps us find the minimum of our objective function.

As we have discussed, we have to derive the weight update rule for our objective function. As you can see, the function consists of two terms; the first is the regularizer and the second is the loss itself. We will derive them separately using the sum rule in differentiation.

First, we will find the partial derivative of the regularizer term with respect to w , and then we will find the partial derivative of the loss function with respect to w , which will give us the following formulas:

$$\begin{aligned}\frac{\delta}{\delta w_k} \lambda \|w\|^2 &= 2\lambda w_k \\ \frac{\delta}{\delta w_k} (1 - y_i \langle x_i, w \rangle)_+ &= \begin{cases} 0 & \text{if } y_i \langle x_i, w \rangle \geq 1 \\ -y_i x_{ik}, & \text{else} \end{cases} \end{aligned} \quad (7)$$

As you can see, the derivatives are quite straight and simple. The preceding terms indicate that if we have a misclassified sample, we update the weight vector w using the gradients of both terms. Otherwise, if it is

classified correctly, we just update w by the gradient of the regularizer.

And as we know from the equation (4), the condition for misclassification is:

$$y_i \langle x_i, w \rangle < 1 \quad (8)$$

So we update the rule for our weights (w) by gradient descent whenever we misclassify an input; the weight will update as:

$$w = w + \eta(y_i x_i - 2\lambda w) \quad (9)$$

As you can see in equation (9), we will change the weights according to the gradients we have got in equation (7). Here we also have included the learning rate η , which helps us to control the rate of convergence. We can see that the regularization parameter λ will help us control the rate of change in the weights. We can conclude the following points from the preceding equation:

- If we make the learning rate too high, the algorithm might overshoot the optimal point
- If we make the learning rate too low, it could take too long to converge or never converge

We have discussed both of the points in detail in [Chapter 8](#), *Stacked Generalization*, so you can take the reference from there.

Now, we will talk a bit about the regularizer; the regularizer controls the trade-off between achieving a low training error and a low testing error, that is, the ability to generalize your classifier to unseen data. As a regularizing parameter, we will choose a value equal to $1/epochs$, so this parameter will decrease as the number of epochs increases:

- If we make the regularizer too high, we can face the problem of overfitting of the classifier (large testing error)
- If we make the regularizer too low, we

can face underfitting of the classifier
(large training error)

Now, for a correctly classified instance, we will not change the weights in the following manner:

$$w = w + \eta(-2\lambda w)$$

(10)

In this equation, as you can see, we have not included input and output values for the instance.

So, now we are all ready to implement the preceding build concepts in code form; let's see whether we can create a hyperplane for our toy example:

```
#lets perform stochastic gradient descent to
learn the separating hyper plane between both
classes
def svm_learning(X, Y):

    #Initialize our SVMs weight vector with
    zeros (3 values including bias as
    feature)
    w = np.zeros(len(X[0]))

    #The learning rate
```

```

eta = 0.9

#how many iterations to train for
epochs = 1000

#store miss-classifications so we can plot
how they change over time
errors = []

#training part, gradient descent part
for epoch in range(1, epochs):

    #Initialize the error variable
    error = 0
    for i, x in enumerate(X):

        #Check for miss-classification
        (Equation no. 8)
        if (Y[i]*np.dot(X[i], w)) < 1:

            #Update the weights for miss
            classified input
            #Here we are using lambda =
            1/epochs (Equation no. 9)
            w = w + eta * ( (X[i] * Y[i]) +
            (-2 * (1/epoch)* w) )
            error = 1
        else:

            #correct classification, update
            our weights
            #Equation (10)
            w = w + eta * (-2 * (1/epoch)*
            w)

        errors.append(error)

    #lets plot the rate of classification
    errors during training for our SVM
    plt.plot(errors, '|')
    plt.ylim(0.5,1.5)
    plt.axes().set_yticklabels([])
    plt.xlabel('Epoch')
    plt.ylabel('Misclassified')

```

```
plt.show()

#Return the updated weights
return w
```

The preceding function will train the SVM using stochastic gradient descent algorithm; we have just implemented equation (8), and equation (10) to perform the optimization. This figure shows how the error reduces during each epoch:

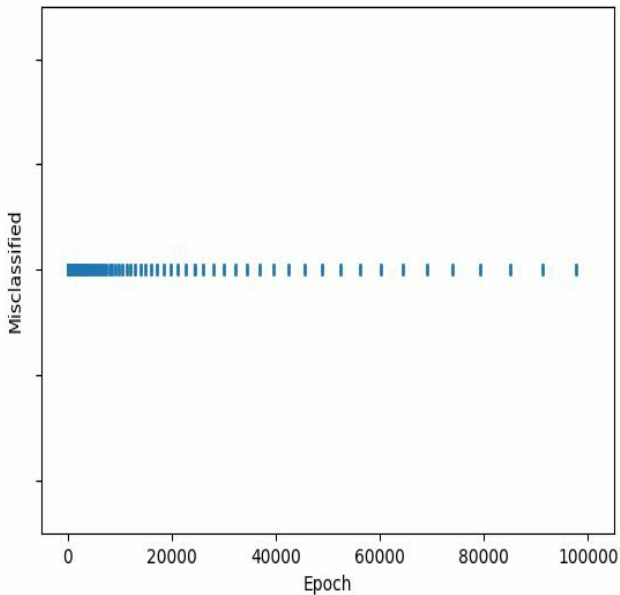


Figure 9.13: Reduction in error in each epoch

Now, let's call this function for our toy example:

```
#Here we will train our classifier
w = svm_learning(X,y)

#Lets plot the points again
for d, sample in enumerate(X):

    # Plot the negative samples
    if d < 2:
        plt.scatter(sample[0], sample[1],
s=120, marker='_', linewidths=2)

    # Plot the positive samples
    else:
        plt.scatter(sample[0], sample[1],
s=120, marker='+', linewidths=2)
```

We will add two test points to our graph and see whether our learned boundary is able to separate them into the correct class or not:

```
#Let's Add our test samples
plt.scatter(2,2, s=120, marker='_',
linewidths=2, color='yellow')
plt.scatter(4,3, s=120, marker='+',
linewidths=2, color='blue')

#Here we will print the hyperplane calculated
by svm_train()
x2=[w[0],w[1],-w[1],w[0]]
x3=[w[0],w[1],w[1],-w[0]]

#Following lines will create a plot with our
```

hyperplane and data

```
x2x3 = np.array([x2,x3])  
X,Y,U,V = zip(*x2x3)  
ax = plt.gca()  
ax.quiver(X,Y,U,V, scale=1, color='blue')  
plt.show()
```

The following graphs show the decreasing error rate and the hyperplane we get after training our classifier:

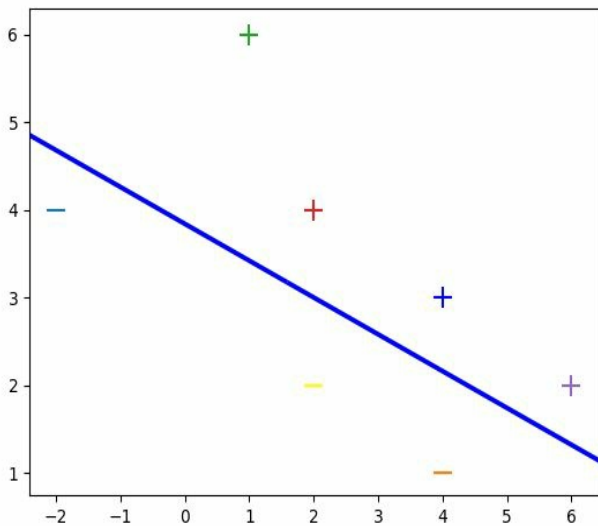


Figure 9.14: Optimized hyperplane showing accurate classification of test points

Wow! Can you see that! We have successfully implemented the SVM algorithm. As you can see, we are able to classify the new input points quite accurately from the preceding algorithm.

Handling a nonlinear dataset

Suppose you need to train a classifier for the data shown in the following figure. What will be your approach to classify the data into two classes?

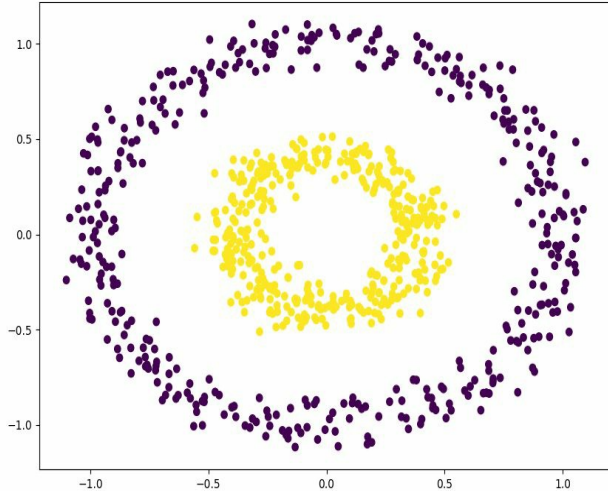


Figure 9.15: Nonlinear dataset

As you can see in the preceding figure, we can't find a decision boundary using a straight line or even a plane because this data is a linearly nonseparable dataset. So, what do we do with this kind of problem?

Well, mathematics has solutions for almost every kind of problem, so for this too. What we need to do is change our perception toward the data. I know you have not got anything I said! Let's understand the preceding problem with an example. Suppose you have some ping pong balls in a bucket. The balls are of two colors: yellow and purple. Someone asks you to find the number of yellow balls in the bucket. Don't think that it is an easy problem, as there may be some balls you cannot see. So, what will you do to find the number? Well, the answer is not that difficult. Just empty the bucket on the surface and you will not have the problem of invisible balls anymore. And now you can count them! Trust me, this is the only

solution that gives you very fast results.

So what is the connection between this bucket and the nonlinear data problem? To solve both the problems, you need to change the space. In case of balls, to see the hidden balls, we empty the bucket on the surface (space) where we can see all of the balls at once. Similarly, we need to change the subspace of our data to see the chances of separation using a linear classifier.

This change in the subspace is known as the **transformation** of data, and the function that helps us to do such a transformation is known as the **kernel**. So, what do you think? Can this strategy of changing spaces and visualizing our data in a new space help us? Let's see.

We will apply a polynomial function to convert our 2D subspace into a 3D subspace; let's write a code and see whether it can help us.

First, we will generate the data for our example:

```
#We will use sklearn's make_circle to create the data  
from sklearn.datasets import make_circles  
  
#Numpy will help us for array related operations  
import numpy as np  
  
#We will use pylab for visualization of plots  
import pylab as pl  
  
#Generate the dataset using make_circle function  
X1, Y1 = make_circles(n_samples=800,  
noise=0.07, factor=0.4)  
  
#Let's Plot the Point and see  
print ("...Showing dataset in new window...")  
pl.figure(figsize=(10, 8))  
pl.subplot(111)  
pl.scatter(X1[:, 0], X1[:, 1], marker='o',  
c=Y1)  
pl.show()
```

The preceding function will generate the nonlinear data with which we have started the problem. Now, we will write a polynomial kernel that will take the x and y coordinates and will transform the points in a different space:

```
#Kernel to convert sub space of data  
def fn_kernel(x1, x2):
```

```

    # Implements a kernel  $\phi(x_1, y_1) = [x_1, y_1, x_1^2 + y_1^2]$ 
    return np.array([x1, x2, x1**2.0 + x2**2.0])

```

If you look carefully, you will understand how the kernel is converting the points from 2D space to 3D space; let's execute it using the following code block:

```

#Create a list to store transformed points
transformed = []

#Transform each point to the new sub space
for points in X:
    transformed.append(fn_kernel(points[0], points[1]))
transformed = np.array(transformed)

#We will 3D plots to visualize data in higher dimension
from mpl_toolkits.mplot3d import Axes3D

#Import matplotlib to plot the data
import matplotlib.pyplot as plt

#Let's plot the original data first
fig = plt.figure(figsize=(20,8))
ax = fig.add_subplot(121)
ax.scatter(X[:, 0], X[:, 1], marker='o', c=Y)
ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_title("Data in 2D (Non-separable)")

#Here we will plot the transformed data
ax = fig.add_subplot(122, projection='3d')
ax.scatter(transformed[:, 0], transformed[:, 1], transformed[:, 2], marker='o', c=Y)

```

```

ax.set_xlabel('X Label')
ax.set_ylabel('Y Label')
ax.set_zlabel('Z Label')
ax.set_title("Data in 3D (separable)")

#Finally show all the plots
plt.show()

```

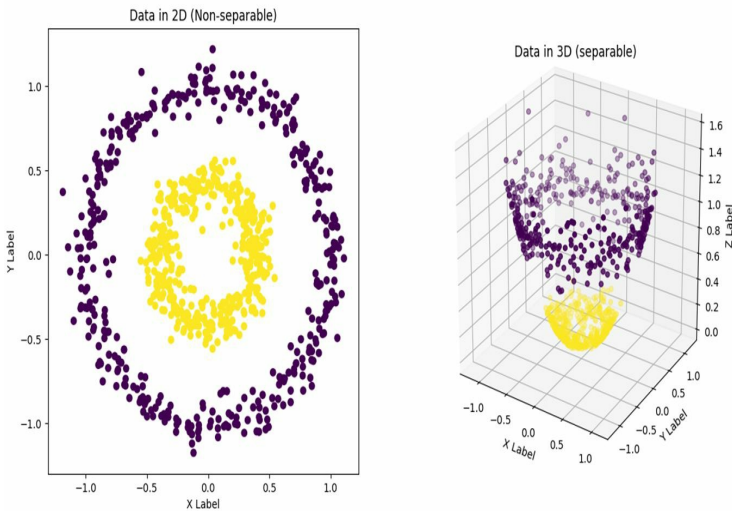


Figure 9.16: Kernel trick for nonlinear dataset

Can you see that? Our transformed dataset can be separated by a hyperplane. The kernel trick is generally used to tackle non-separable datasets. There are many kinds of kernels available according to different requirements, such as the radial basis kernel (Gaussian kernel), polynomial kernel, sigmoid kernel,

and so on.

Let's see how to use a kernel in the SVM:

```
#Import our SVM classifier from sklearn
from sklearn.svm import SVC
#Let's merge input and output variable to
create train and test data
dataset = np.c_[X,Y]

#We will use our train and test split function
train,test = getTrainTestData(dataset, 0.7)

#Extract training input and output
x_train = train[:,0:2]
y_train = train[:,2]

#Extract testing input and output
x_test = test[:,0:2]
y_test = test[:,2]
```

We will train our classifier with a linear kernel first; it will show you how the kernel choice is very critical for the classifier's performance. We will see the prediction accuracy of our classifier on the testing dataset:

```
#First we will train our classifier with linear
kernel
clf = SVC(kernel='linear')
clf.fit(x_train,y_train)

#Predict the output on test set
pred = clf.predict(x_test)
```

```
| acc = getAccuracy(pred, y_test)
| print("Accuracy of the classifier with linear
| kernel is %.2f"%(100*acc))

| Accuracy of the classifier with linear kernel
| is 45.83
```

As you can see, this is what we were expecting from our classifier. As our data is not linearly separable, linear classification will not work for us; we are getting an accuracy as bad as a random classification. Now we will train our new classifier with the **Radial Basis Function (RBF)**:

```
| #Now we will train our classifier with RBF
| kernel
| clf = SVC(kernel='rbf',C=3.0)
| clf.fit(x_train,y_train)

| #Predict the output on test set
| pred = clf.predict(x_test)
| acc = getAccuracy(pred, y_test)
| print("Accuracy of the classifier with rbf
| kernel is %.2f"%(100*acc))

| Accuracy of the classifier with rbf kernel is
| 100.00
```

Can you see the change of choosing a kernel? We are getting correct classifications for all of our test instances; this is why RBF kernels are the best choice for nonlinear data.

So, I think we have discussed enough on SVMs, and we are ready to use them in our stacked generalization. Remember, we will always use an SVM with some kernel. Now, which kernel can perform best depends on the dataset. To find out the optimal hyperparameters for the classifier, `sklearn` uses a class for grid search; this class tests the classifier performance for a set of different parameter values. So, it is a good practice to apply grid search and cross validation to decide on a classifier.

Stacking of nonlinear algorithms

So, it's time to return to the current topic—stacked generalization. We have seen stacked generalization in the previous chapter and implemented it successfully. This time, we will go for a more advanced version of stacked generalization; we will stack all of the classifier variations we have worked on in this book, such as decision trees, random forest, AdaBoost classifier, SVMs, KNN, and logistic regression. We will combine their result by voting.

So, what is the difference from the previous approach of stacking? Well, this time, we will use bagging to stack the classifier; this is the procedure we will follow:

1. Create a stack of multiple classifiers
2. According to number of classifiers,

create a sample out of the data by replacement

3. Now, train each classifier with a different sample
4. At the end of the process, take the vote of each classifier to classify the data

The following figure will clarify the concept of our process:

>

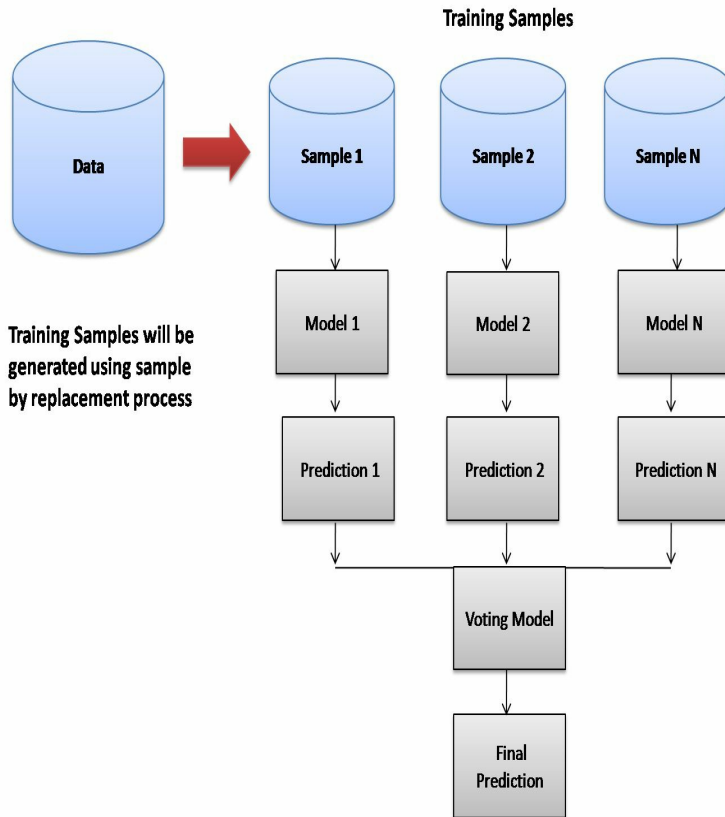


Figure 9.17: Bagged stack of classifiers

Why do we do bagging for a stacking process? As we have already discussed, bagging is very effective at reducing correlations between classifiers, as all of the classifiers receive different samples from the

same data population. If the correlation between the classifiers is less, the problem of high variance and high bias can also be countered by the process.

We have already used bagging with decision trees successfully, so why are we combining different models? As we have seen in the previous chapter, each classifier has its own weaknesses and strengths; stacking can overcome the weakness of a classifier by using the strengths of others.

As we have seen data of linearly separable and non-separable classes, stacking can also counter the problem of choosing the right classifier, because a single classifier may not be that powerful to fit complex datasets.

So, I think we have talked enough about the process and it's time for action. We are going to implement the most powerful classifier for a well-known practical issue: Spam classification. But wait! We have already implemented the KNN algorithm for this

problem, so why are we again going to target the same problem? You are right, but do you remember that the classification accuracy we were getting was about *80%*? Do you think it is an acceptable performance in the real world? Well, the performance of a classifier depends on various factors and it is quite subjective to the dataset, too. As a spam classification dataset is very complex, it is difficult to get high accuracy on this data; so, we will use the power of unity for this.

Spam classification with stacking

Let's start with the process. We will use a publicly available dataset for our application; this dataset is available at: <https://archive.ics.uci.edu/ml/datasets/spambase>. You can easily download it and store it in the form of a `.csv` file, as we already have functions to read a `.csv` file and load the dataset into a numpy array (refer to [Chapter 3, Random Forest](#)).

The following is the dataset information as found on the given web address.

Dataset information

The spam concept is diverse: advertisements for products/websites, *make money fast* schemes, chain letters, pornography, and so on.

Our collection of spam e-mails came from our postmaster and individuals who had filed spam. Our collection of non-spam e-mails came from filed work and personal e-mails; hence, the word *george* and the area code 650 are indicators of non-spam. These are useful when constructing a personalized spam filter. One would either have to blind such non-spam indicators or get a very wide collection of non-spam to generate a general-purpose spam filter.

Attribute information

You can get all available information on the preceding link. I will be covering only a basic understanding of the dataset. There are 57 attributes in the dataset; each attribute signifies the frequency of that word in the spam or non-spam emails. For example, in a spam mail, there are certain words that have more occurrences, such as money, rich, hot, nearby, business, and so on. The following is a snapshot of the attribute names:

word_freq_order:	continuous.
word_freq_mail:	continuous.
word_freq_receive:	continuous.
word_freq_will:	continuous.
word_freq_people:	continuous.
word_freq_report:	continuous.
word_freq_addresses:	continuous.
word_freq_free:	continuous.
word_freq_business:	continuous.
word_freq_email:	continuous.

On the left, there are attribute names, and on

the right is the type (continuous, discrete, and so on).

There are 2,906 instances in the dataset where the last instance has an incomplete set of attributes, so we will keep an eye on it when we perform our calculation.

For this purpose, we will need the following dependencies:

- A function to read .csv files
- A function to convert string attribute values to numerical form
- We will need a function to create samples of data
- A function to stack the prediction for the voting
- Finally, a function to evaluate the performance of the system

So we will use the same function to read .csv files as we have used in [Chapter 3, Random Forest](#). The code block is as follows:

```
| #Function to read csv file
```



```
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset
```

Now, we will add the function to convert string values present in the data to the numerical values; we have to do it first for input attributes and then for the output class variable. This function will also replace missing values (if any) in the data with 0:

```
#Function to convert string attribute values to
float
def str_column_to_float(dataset, column):
    for row in dataset:
        if row[column]=='?':
            row[column] = 0
        else:
            row[column] =
float(row[column].strip())
```

We will create samples of data using the `cross_validation` function(which we have created in [Chapter 3, Random Forest](#)) to evaluate random forest algorithm. This function will create samples from the input data by randomly selecting row indices:

```

#Create cross validation sets
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)

    for i in range(n_folds):
        fold = list()

        while len(fold) < fold_size:
            index =
randrange(len(dataset_copy))

        fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)

    return dataset_split

```

Now, we will add a function to stack the predictions from different classifiers:

```

def stacking(dataset, models):
    stackedData = []
    for model in models:
        pred = model.predict(dataset)
        stackedData.append(pred)
    return np.transpose(stackedData)

```

We will add the `get_accuracy` function to evaluate the model performance:

```

#Function to evaluate model performance
def getAccuracy(pre, ytest):
    count = 0
    for i in range(len(ytest)):
        if ytest[i]==pre[i]:
            count+=1

```

```
    acc = float(count)/len(ytest)
    return acc
```

We will add a helper function to create input and output variables in separate arrays, because all classifier classes in the `sklearn` library need training data in (input, output) form. The function shown here will do the same work:

```
def getXY(dataset):
    dataset = np.array(dataset)
    shape = np.shape(dataset)
    X = dataset[:,0:shape[1]-1]
    Y = dataset[:,shape[1]-1]
    return X,Y
```

We create one small function to train the classifier; so we train multiple classifiers in a for loop:

```
def stack_fit(model,x,y):
    return model.fit(x,y)
```

Now, we are ready to go; let's start with the main body of the function:

```
#Import numpy for array based operations
import numpy as np

#Import Support vector machine
from sklearn.svm.classes import SVC
```

```

#Decision Tree Classifier
from sklearn.tree import DecisionTreeClassifier

#KNN
from sklearn.neighbors import
KNeighborsClassifier

#Logistic Regression
from sklearn.linear_model import
LogisticRegression

#Random Forest Classifier
from sklearn.ensemble import
RandomForestClassifier

#Ada-boost Classifier
from sklearn.ensemble import AdaBoostClassifier

#Set Random seed
np.random.seed(1)

#Specify the file name
dataName = 'spamData.csv'

#Use function load_csv
dataset = load_csv(dataName)

#Create an empty list to store the dataset
dataset_new = []

#We will remove incomplete instance from the dataset
for i in range(len(dataset)-1):
    dataset_new.append(dataset[i])
dataset = dataset_new

#Use function str_column_to_float from chapter 3 to convert string values to float
for i in range(0, len(dataset[0])-1):
    str_column_to_float(dataset, i)

```

```

#Convert class variable to the numerical value
str_column_to_int(dataset, len(dataset[0])-1)

#Shuffle the dataset
np.random.shuffle(dataset)

#Create the sample out of datasets
splits = cross_validation_split(dataset,7)

#Load all the classifiers
clf1 = AdaBoostClassifier()
clf2 = DecisionTreeClassifier()
clf3 = KNeighborsClassifier(n_neighbors=1)
clf4 = RandomForestClassifier()
clf5 = LogisticRegression()
clf6 = SVC(kernel='rbf')

#Stack all the classifier
models = [clf1,clf2,clf3,clf4,clf5,clf6]

#Initialize the variable for trained classifier
trained =[]

#Train the model and add to the stack
for i in range(len(models)):
    model = models[i]
    x,y = getXY(splits[i])
    trained.append(stack_fit(model, x, y))

#Create test data from left split
xtest,ytest = getXY(splits[6])

#Generate the stacked predictions
stackedData = stacking(xtest, trained)

#Here we will calculate individual accuracies of models
for i in range(np.shape(stackedData)[1]):
    acc = getAccuracy(stackedData[:,i], ytest)
    print("Accuracy of model %i is %.2f"%(i,
(100*acc)))

```

Let's see the individual accuracy of each model:

```
Accuracy of model 0 is 93.73 # AdaBoost classifier  
Accuracy of model 1 is 88.19 # Decision Tree classifier  
Accuracy of model 2 is 73.73 # KNN classifier  
Accuracy of model 3 is 91.33 # Random Forest Classifier  
Accuracy of model 4 is 93.49 # Logistic Regression  
Accuracy of model 5 is 79.04 # Support Vector Machine
```

As you can see, all of our six classifiers perform differently on the same population of the data. SVM and KNN classifiers are underperforming compared to other classifiers. Well, this was expected as the data is very complex and we already know the previous performance of the KNN classifier. So, this is not a surprise, but notice that the AdaBoost classifier and logistic regression are outperforming the random forest and decision tree classifier. This is quite surprising because logistic regression is way simpler than random forest. Now, it's time to take the votes from each classifier and conclude at the final level. What do you

think? Can we outperform the performance of logistic regression (93.49%)?

```
#Take the vote of each classifier and create  
final prediction  
predLr =  
[np.bincount(np.array(pred, dtype="int64")).argmax  
 for pred in stackedData]  
  
#Evaluate the stacked model performance  
accLr = getAccuracy(ytest, predLr)  
print("\nAccuracy of stacking is %.2f"%  
(100*accLr))
```

After execution, we get:

```
| Accuracy of stacking is 94.94
```

WOW! We have got an accuracy of almost 95%; this is quite a good performance as compared to 80% of the previous KNN. And this is with all of the default parameters. If we tune the parameters using grid search and evaluate with cross validation, there are chances of more improvement in the final accuracy.

I have run some heuristics and got that if we remove the SVM from the classifier stack, we will get the following stats:

```
| Accuracy of model 0 is 94.83  
| Accuracy of model 1 is 90.29  
| Accuracy of model 2 is 70.66  
| Accuracy of model 3 is 94.63  
| Accuracy of model 4 is 94.01  
| Accuracy of stacking is 96.28
```

Can you believe this!! We have got an accuracy of more than 96% with all of the default parameters. Remember, I am taking random samples, so there are chances that you may get a different performance. But believe me, it will be more than what we have got previously.

How to choose classifiers?

Now the question is: how to choose a classifier to include in the stack? The answer is very difficult; it is similar to selecting hyperparameters for the classifier. The choice of classifiers for stacking is quite subjective and it is also an area of research. For the time being, you can run some heuristics tests and decide which classifiers are giving you less correlation in their prediction results.

However, you can work on the following points:

- Use feature importance as a selection criteria. It can give an overview of information overlap between the classifiers. We have discussed feature importance earlier in the chapter.
- Check the correlation between the

predictions of the classifiers.

- You can use the grid search method to choose the best combination of classifiers.
- Cross validation is always a good choice for choosing the best classifier combination.

Summary

We started with feature selection methods, where we discussed the importance of feature selection and also its benefits by implementing various feature selection algorithms using the `sklearn` library. We saw how feature selection reduces the curse of dimensionality and improves the performance of the classifier by reducing the variance and bias trade-off.

We discussed SVMs in quite a bit of detail by implementing one from scratch. We also saw how it optimizes its loss function using the gradient descent algorithm, and used the kernel trick to address the problem of non-separable datasets.

In the end, we came back to stacking, this time with a bang. We used six classifiers to create a stack of classifiers and used the bagging strategy to predict the output of the

classifier, not to mention that we got over 16% improvement in the classification of the spam dataset than with KNN classification. Finally, we talked a little bit about the strategy for selecting the classifiers for the stacking.

So, I think this is enough talk about stacking classifiers. There are innumerable ways to improve the classification accuracy of ensemble systems. As we cannot discuss all of them at once, I encourage you all to visit <https://www.kaggle.com/arthurtok/introduction-to-ensembling-stacking-in-python> to get more information regarding the stacking of classifiers. At the end, keep practicing because practice makes a better classification module!

Modern Day Machine Learning

We have just finished a discussion on traditional and conventional ensemble methods used in machine learning and data analytics. We have seen parametric (logistic regression and perceptron) and nonparametric algorithms (decision trees, KNN, and AdaBoost) for predictive analysis tasks. Nowadays, there are quite a plethora of data analytics field as we are overfilled with digital data. As this field is growing too fast, data complexity is also increasing proportionally. The algorithms we have learned so far may not give the perfect solution for very complex datasets, such as image recognition tasks or tasks related to word embedding. These two terms are quite related to natural language processing.

In short, **Natural Language Processing**

(NLP) is mainly about solutions in speech recognition, image classification, document classification, or a very new but dominating field of generative models. We can deal with most of the domains with our conventional machine learning, too, but there is always a compromise with the accuracy as it is very difficult to find the correct set of distinctive features to represent the dataset. If we cannot find the correct set of features for prediction tasks, we will get irrelevant predictions from any state-of-the-art algorithm.

Let's understand this with an example.

Suppose we want to create an ML solution for classification of images of cars from one million other images of *1,000* other objects such as buses, animals, houses, and so on. As we know, a traditional machine learning pipeline consists of the following key points:

- Data interpretation or exploratory data analysis (the behavior of a dataset, linear or nonlinear)
- Feature extraction or feature engineering

(to represent a dataset in a concise, meaningful way)

- Feature selection (to reduce redundant or irrelevant features)
- Algorithm selection (the one best suited for the dataset)
- Deployment of the solution

So again, we will end up with a solution like this:

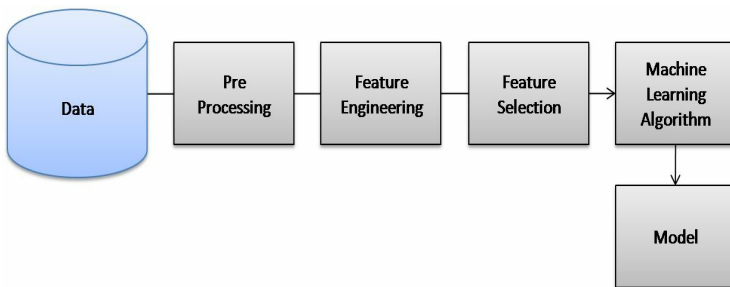


Figure 10.1: Machine learning pipeline

Let's try to fit the preceding example into this pipeline. We will assume that our data is cleaned so that we can simplify the concept.

Our task will start with feature engineering. Here, feature engineering is related to

representing images of cars in a meaningful way such that we can differentiate the images and other objects with these features. One can think of raw pixel intensities for the task, but if we keep in mind that there may be a huge variation in color, lighting conditions, and car structures, it will be quite difficult to distinguish cars from other objects. We have dealt with feature engineering in the **face detection task earlier**. For such tasks, feature engineering is a useful thing. Where we apply different kinds of algorithms to quantify colors of images, we can apply various frequency or shape-based filters to find out high-frequency components (such as edges and corners) of the object or low-frequency components such as uniformity of intensity levels using weighted averages, and so on. Please keep in mind that we want to extract the features that can differentiate a car from 1,000 other objects.

What do you think? What are the chances that we've got very relative features to classify our object of interest? Suppose we want extract

frequency-based features. What should be the range of frequencies we need to extract for our images? And there are many more variables in the images such as another object overlapping with the object of interest. In such cases, we can go for wrong detection and classification.

So, is there no way to find out the correct features for our kind of problem? There is. In fact, we have used these types of features earlier in the face detection task:

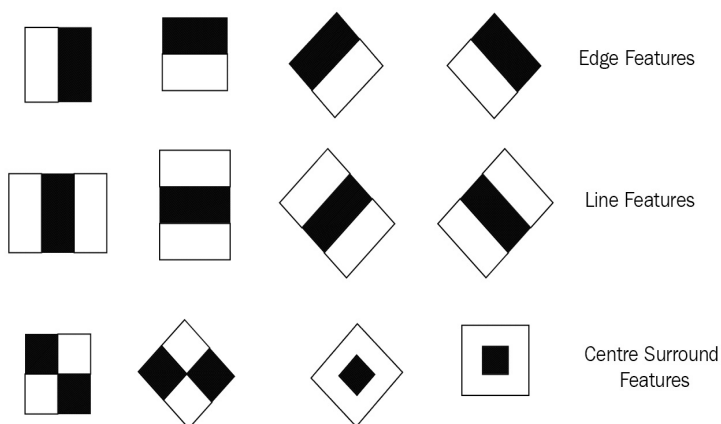


Figure 10.2: Haar cascade features used in object detection

However, these features have their own limitations; for example, these features are not rotation or scale invariant. If we want to detect an image that consists of various sizes of faces, we have to perform the operation for multiple scales (different sizes of images). If the faces are not straight, there are less chances of detection. Last but not least, as we have a huge dataset but a limited set of classification parameters, there are chances of high bias in the classification. This is because it's very difficult to construct a huge classifier for such an application.

So, is there any solution for this kind of problem? Yes, there is. It is modern day machine learning algorithms: **Artificial Neural Networks (ANNs)**, or more precisely, **Convolutional Neural Networks (CNNs)**. Or the modern day ML legend—deep learning! But how can CNNs help us to solve these kinds of problems? Well, the answer is pretty simple. They learn the features by themselves. What? By themselves! Yes, you read it correctly. CNNs

are a special kind of machine learning algorithms that learn the underlying properties of data by themselves. More precisely, these algorithms learn weight parameters as the features of datasets.

CNN are a special kind of ANN themselves. They have many similarities as well as differences, but the origin of both of them is the same. The perceptron. Yes, the same perceptron we saw earlier.

If you forgot, the following is a figure showing a perceptron:

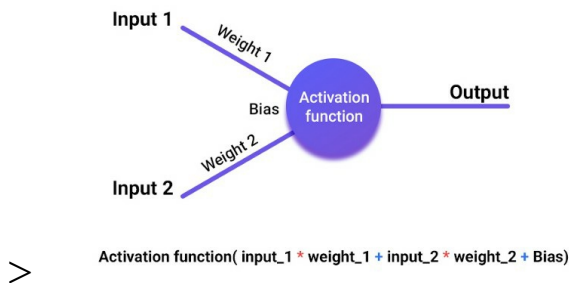


Figure 10.3: Perceptron architecture

ANNs are also known as multilayer

perceptron models; they consist of a lot more than one neuron. We will start our discussion with a description of ANNs and move on to CNNs. So let's not waste time and start the discussion.

Artificial Neural Networks (feed-forward)

We have got an informal introduction to ANNs as multilayer perceptron models. What does this actually mean? We developed a perceptron algorithm in [Chapter 8](#), *Stacked Generalization*. According to its definition, the perceptron is a mathematical model of a biological neuron. In actual neurons, the dendrite receives electrical signals from the axons of other neurons, but in a perceptron, these electrical signals are represented as numerical values. At the synapses between the dendrite and axons, electrical signals are modulated in various amounts. This is also modeled in the perceptron by multiplying each input value by a value called weight. An actual neuron fires an output signal only

when the total strength of the input signals exceeds a certain threshold. We model this phenomenon in a perceptron by calculating the weighted sum of the inputs to represent the total strength of the input signals and then applying a step function to the sum to determine its output. As in biological neural networks, this output is fed to other perceptrons.

When we combine many perceptrons in a structured way, as shown in the next figure, the structure is known as ANN. From now on, we will call them **neurons**, but there is a basic difference between neurons and perceptrons. Perceptrons always output binary values because of their step function behavior, while neurons are a special kind of perceptron with a transfer function, such as sigmoid. So, they can generate continuous values at the output.

If you observe the figure, you can see that there are total *14* perceptrons in the model: *6* in the first level and *4*, *3*, and *1* in the

following levels. These levels are known as different layers of the network.

The leftmost layer of the network is known as **input layer**, and the rightmost layer is known as **output layer**. All other layers between these two are known as **hidden layers**. There is no technical explanation on why to call them hidden layers. They are called so because the user does not interact in any kind of data-related operation with them:

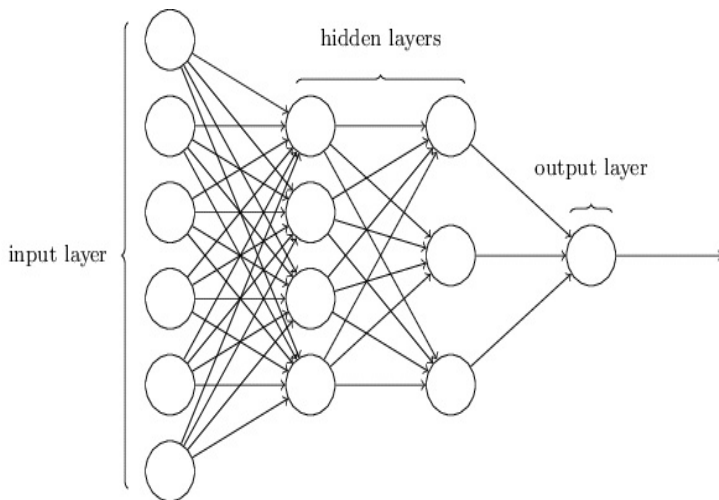


Figure 10.4: Artificial Neural Network (feed-forward network)

The connections you can see between one layer of neurons to the next layer are weighted connections; that means when one neuron from the first layer passes any value to the next neuron, it will be multiplied with the connection's weight (W) and it will pass it to the next layer. The multiplication of input values and weight values is straightforward:

$$y = W * X + b$$

Here, W is the weight of the connection and b is the bias, while X and y are the input and output values, respectively.

So, when a neuron from the hidden layer receives any input from the previous layer (currently the input layer), it is the sum of all weighted connections from the previous layer to the hidden layer.

The following figure shows this operation:

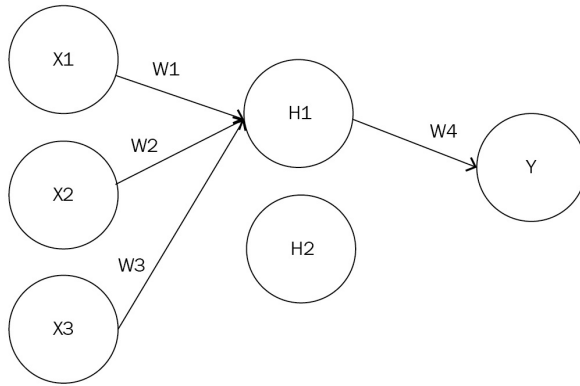


Figure 10.5: How a neural network works

The preceding figure shows the calculation of input received at H_1 . If we expand it, it is like:

$$H_1 = X_1 * W_1 + X_2 * W_2 + X_3 * W_3$$

We can summarize this as:

$$H_{jl} = \sum X_{il-1} * W_{kl}$$

Where i and j are i^{th} and j^{th} neurons of layers $l-1$ and l , respectively. k is the k^{th} weight between layer $l-1$ and layer l . So, by the preceding equation, we can estimate the

input of any neuron in a hidden layer.

Now, let us talk about the output of neuron H_1 . As simple as that, the output of a neuron should be the sum of the products, as we have seen in a previous paragraph. However, there is one problem in this. If we choose the sum of products as the output of any neuron in the network, it can be a very large number, or it may be a negative number. So, to keep the number in a defined range, we will choose an activation function to transform the input number into a defined range. There are many types of activation functions available, such as sigmoid (logistic), tanh, softmax, and so on. There should be an important property of activation. It must be a differentiable function. Why? Because gradient-based methods are involved during the weight learning process, such as gradient descent, so we need to calculate the derivative of the activation function.

After applying the *activation* to the neuron, the output will change as follows:

$$H_{jl} = \text{activation} \left(\sum X_{il-1} * W_{kl} \right)$$

Sigmoid is very a popular choice of *activation* function. It is applied as follows:

$$H_{jl} = \frac{1}{1 - e^{-\sum X_{il-1} * W_{kl}}}$$



Now, this output will pass through the weighted connection of the current layer and the next layer and will be summed with the output of other neurons. And the process will continue until the output layer.

As you have seen in this process, we have just moved forward from the input to the output layer. There was no feedback or recursion applied. Such neural networks are known as **feed-forward neural networks** and the procedure of moving forward is known as forward propagation.

How does ANN work?

I will discuss this with an example of a very popular machine problem: handwriting recognition. This problem consists of images of handwritten digits. They are written by more than *50,000* people; so, there is huge variability in the structure (shape and alignment) of each digit as each person has his/her own style of writing, which makes it a very complex visual recognition problem. To recognize individual digits, we will use a three-layer neural network. The input layer of the network contains neurons encoding the values of the input pixels. As discussed in the next section, our training data for the network will consist of many 28×28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons. For simplicity, I've omitted most of the *784* input

neurons in the diagram.

The input pixels are grayscale, with a value of 0.0 representing white, 1.0 representing black, and in-between values representing gradually darkening shades of gray:

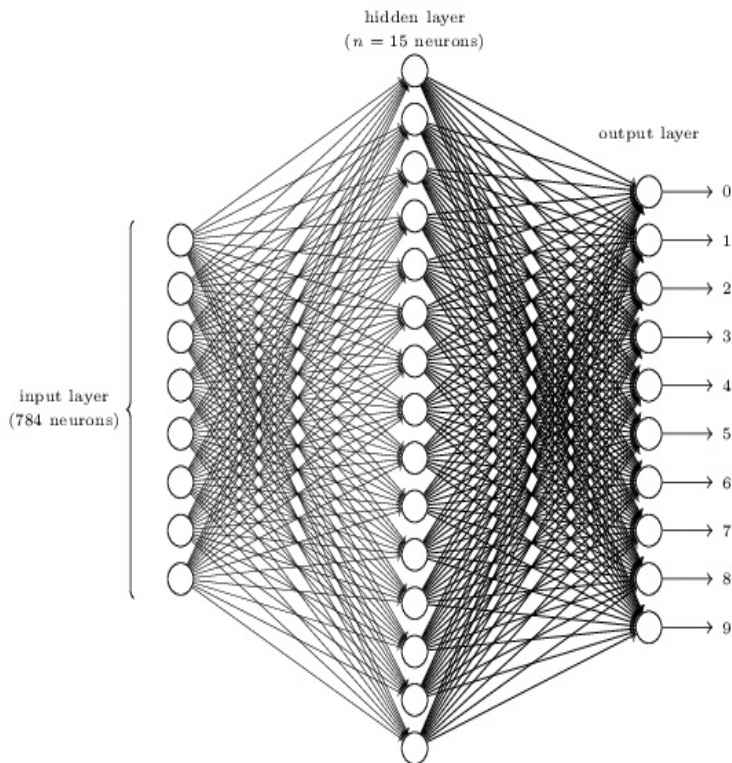


Figure 10.6: ANN for a digit recognition problem

The second layer of the network is a hidden layer. We denote the number of neurons in this hidden layer by n , and we'll experiment with different values for n . The example shown illustrates a small hidden layer containing just $n=15$ neurons.

The output layer of the network contains 10 neurons. If the first neuron fires, that is, has an output ≈ 1 , then that will indicate that the network thinks the digit is a 0. If the second neuron fires, then that will indicate that the network thinks the digit is a 1, and so on. A little more precisely, we number the output neurons from 0 through 9 and figure out which neuron has the highest activation value. If that neuron is, say, neuron number 6, then our network will guess that the input digit was a 6, and so on for the other output neurons.

Our input images look something like this:

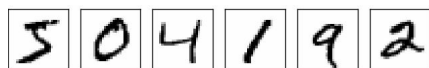


Figure 10.7: Digits' images in the MNIST dataset

So, for instance, we'd like our program to recognize that the first digit in the preceding image:



You might wonder why we use 10 output neurons. After all, the goal of the network is to tell us which digit (0, 1, 2,...,9) corresponds to the input image. A seemingly natural way of doing that is to use just four output neurons, treating each neuron as taking on a binary value depending on whether the neuron's output is closer to zero or to one. Four neurons are enough to encode the answer, since $2^4=16$ is more than the ten possible values for the input digit. Why should our network use ten neurons instead? Isn't that inefficient?

To understand why we do this, it helps to think about what the neural network is doing from the first principles. Consider the case

where we use *1010* output neurons. Let's concentrate on the first output neuron; the one that's trying to decide whether or not the digit is a *0*. It does this by weighing up evidence from the hidden layer of neurons. What are those hidden neurons doing? Well, just suppose for the sake of argument that the first neuron in the hidden layer detects whether or not an image like the following is present:

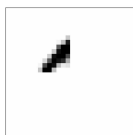


Figure 10.8: Part of a digit image

It can do this by heavily weighting input pixels that overlap with the image, and only lightly weighting the other inputs.

In a similar way, let's suppose for the sake of argument that the second, third, and fourth neurons in the hidden layer detect whether or not the following images are present:

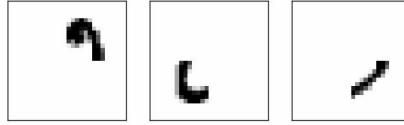


Figure 10.9: Different parts of a digit image

As you may have guessed, these four images together make up the **0**:

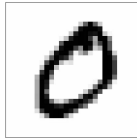


Figure 10.10A: Complete digit image by combining different parts

So, if all four of these hidden neurons are firing, then we can conclude that the digit is a **0**. Of course, that's not the only sort of evidence we can use to conclude that the image was a **0**; we can legitimately get a **0** in many other ways (say, through translations of the preceding images or slight distortions). But it seems safe to say that at least in this case, we'd conclude that the input was a **0**.

Supposing that the neural network functions

in this way, we can give a plausible explanation for why it's better to have ten outputs from the network rather than four. If we had four outputs, then the first output neuron would be trying to decide what the most significant bit of the digit was, and there's no easy way to relate that most significant bit to simple shapes like those shown in the preceding images. It's hard to imagine that there's any good historical reason the component shapes of the digit will be closely related to (say) the most significant bit in the output.

Now, with all that said, this is all just a heuristic. Nothing says that the three-layer neural network has to operate in the way I described, with the hidden neurons detecting simple component shapes. Maybe, a clever learning algorithm will find some assignment of weights that lets us use only four output neurons. But as a heuristic, the way of thinking I've described works pretty well and can save you a lot of time in designing good neural network architectures.

Now, we have discussed quite a lot about the architecture and working of ANN, but we have not discussed how to train the network. Well, in the next section, we will look for that.

Training of ANNs

The training of ANNs is a tricky task. We will use the same gradient descent algorithm we had used during training of perceptron, logistic regression, and SVM, but it will be very much similar to the perceptron model.

Let's first summarize the process of the gradient descent algorithm. For a detailed explanation, please refer to [Chapter 8, Stacked Generalization](#). Here, we will do a quick review of the algorithm.

Gradient descent is a gradient-based optimization algorithm. It helps us move our function's parameters towards the direction of gradients, which eventually helps us to obtain the minimum value of our function for selected parameter values. We will start with defining our cost function. As we have already discussed, a cost function is a function of our network variables. We want

to calculate the optimum value for the variables so that we can minimize the value of the cost function. Cost functions are also known as objective functions. Here, we will use sum of squares as a cost function to optimize the network parameters. It is a quadratic function.

$$C(W, b) = \sum_i (y_i - \bar{y})^2$$

The preceding equations shows the cost function, which we want to minimize. We are simply calculating the sum of squares of error between the actual and predicted output.

Where the predicted output is:

$$\bar{y} = \sum_j^M W * X + b$$

The preceding function will lead us to an assumption. If we make slight changes to the values of W and b , we can make changes to cost C . So, our task will be to find out the

best pair of W and b that will lead us to a negative value of C .

The partial derivative of C with respect to W and b will help us to estimate the direction of the gradient, which will eventually help us find out the best values of W and b that allow us to get the minimum value of the function.

When we make a derivation for the preceding function step by step, we will end up with the weight and bias changing rules. Let's see that:

$$w_k \rightarrow w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \rightarrow b'_l = b_l - \eta \frac{\partial C}{\partial b_l} \quad (\text{A})$$

By repeatedly applying this update rule, we can *roll down the hill* and hopefully find a minimum of the cost function.

For **Stochastic Gradient Descent (SGD)**:

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial w_k}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial b_l} \quad (\text{B})$$

Where m is the size of the mini batch on which we want to update the weights and biases.

I have just written down the most important equations for limiting our scope to practical implementation. I will strongly advise you to revisit our gradient descent discussion from [Chapter 8, *Stacked Generalization*](#). Only then will you be able to understand the whole process.

So, gradient descent will help us to update the weights in the network, the same as we have done in the case of perceptron. Here, I want to ask you a question. We have an ANN with three layers: input, hidden, and output. When we calculate the error (or cost) between the

predicted output and actual output, how will we use this error to learn the weights of the hidden layer?

Learning by backpropagation

At the end of the previous paragraph, I left you with a question on how a hidden layer of the network will know about the error, because only the output layer's neuron will know about the difference between actual output and predicted output! So, as you can see, the hidden layer is completely unaware of what is happening at the last layer. What to do in that case? Well, there is a solution. We can pass the output error to the hidden layer through the same connections that were used during the forward pass. This procedure is known as backpropagation.

So, during the training of the network, we will calculate the error at the output layer, and then we will pass the same error to the hidden layer using the same connections and weights. Thus, we can use gradient descent

for optimization of the layer parameters (W, b) . How does it work? We have to look into that.

The backpropagation algorithm was originally introduced in the 1970s, but its importance wasn't fully appreciated until a discussion of learning representations by back-propagating errors (<http://www.nature.com/nature/journal/v323/n6088/pdf/323533a0.pdf>) presented by *David Rumelhart*, *Geoffrey Hinton*, and *Ronald Williams*. That paper describes several neural networks where backpropagation works far faster than earlier approaches to learning, making it possible to use neural nets to solve problems that had previously been unsolvable. Today, the backpropagation algorithm is the workhorse of learning in neural networks.

As the name suggests, we will start moving backwards from the output layer to the input layer. So, let's start with it. As you know, we want to optimize the cost function, which is our ultimate objective. This optimization is

directly related to changing variables (W , b) in the network. So, our total concentration will always be on changing variable values and the amount of change will depend on the error produced at the output layer. We will start from there onwards.

We will denote the error by δ . The error for a neuron j in any layer l will be:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \quad (1)$$

Here, a is the activation applied to the actual output z of neuron j . σ' is the derivative of our sigmoid function applied to input z . C is our quadratic cost function. We can write the preceding equation in a simplified form as:

$$\delta^L = (a^L - y) \odot \sigma'(z^L) \quad (1.1)$$

The preceding equation is quite intuitive as we know the cost function is nothing but the difference between our final layer activation a (that is, the predicted output) and actual

output y .

Now, we will pass this error to the next layer (backwards) through the connected weights, as follows:

$$\delta^l = ((W^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \quad (2)$$

Where $(W^{l+1})^T$ is the transpose of the weight matrix W^{l+1} for the $(l+1)^{th}$ layer. This equation appears complicated, but each element has a nice interpretation. Suppose we know the error δ^{l+1} at the $(l+1)^{th}$ layer. When we apply the transpose weight matrix, $(W^{l+1})^T$, we can think intuitively of this as moving the error *backward* through the network, giving us some sort of measure of the error at the output of the l^{th} layer. We then take the Hadamard product $\odot \sigma'(z^l)$. This moves the error backward through the activation function in layer l , giving us the error δ^l in the weighted input to layer l .

So, if you pay attention to equations (1.1) and (2), we can calculate the error for any layer

from output to input layer, and it will help us to optimize our cost function.

Now, we have the error in our hands. This is the time to calculate the change required to make in the weight value and bias value so that we can move our variables in the direction of gradients.

First, we will talk about the change required to make in the weight value.

For that, we want to calculate the value of W' , which can be written in equation form as:

$$W' = \frac{\partial C}{\partial W_{jk}^l} \quad (3)$$

We can calculate this value by:

$$\frac{\partial C}{\partial W_{jk}^l} = a_k^{l-1} \delta_j^L \quad (4)$$

Here, you need to understand that W_{jk} is the connection between weight j of layer l and

weight k of layer $(l+1)$. So, we can get the change in the weight using the activation of the previous layer $(l-1)$, which is the input to the current layer l and error at the current layer. This can be calculated from equation (2). We can generalize the preceding equation so that we can calculate the change in weight for any layer:

$$\frac{\partial C}{\partial W} = a_{in} \delta_{out} \quad (5)$$

Similarly, we can get the change in bias. As bias is just an offset value, we can treat the error at neuron j as the required change:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (6)$$

In a more generalized form, we can write it as:

$$\frac{\partial C}{\partial b} = \delta \quad (7)$$

As you can see, we can calculate all of them using NumPy in Python, but it is not in our current scope to implement the backpropagation algorithm in Python for training the neural network.

Now we can use expression (A) of gradient descent when we work with batch gradient descent, where we will update the weights of the network after calculating the gradients for all the examples in the dataset.

Or we can use expression (B), where we can divide our dataset into many small batches and then train the network in the batch gradient style; but now we will update the weights using gradients of individual batch samples. These batches have randomly shuffled data samples. This procedure is known as mini-batch gradient descent, or by the well-known term, SGD. In expression (B), m is the size of mini-batches; we will talk about its effect on the network training when we really train a network.

So, we have seen the derivation of backpropagation in this section. I have tried to avoid math as much as I can; if you want more information about the backpropagation concepts, you can refer to: <http://neuralnetworksanddeeplearning.com/chap2.html>. This is a very nice article that will give you an in-depth understanding of the equations we have used in this section.

Now, we've learned all the required concepts of the process of training a neural network. The rest of the concepts we will try to complete during our practical implementation of the problem in Python using Keras and TensorFlow libraries.

ANN implementation using Keras and TensorFlow

In this section, we will implement the solution for the handwriting recognition problem. We introduced the problem in previous sections. Now, we will see how we can create, train, and test a neural network using two third-party libraries: Keras and TensorFlow.

Before moving towards the solution of the problem, we should discuss both of the libraries first. We will just cover an introduction to them.

TensorFlow for machine learning

TensorFlow is a machine learning library developed by Google. It is a free open source library, designed mainly for building ANN for practical applications. It is library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them.

The flexible architecture allows you to deploy computations to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API. TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google's machine intelligence research organization for the purpose of conducting machine learning and deep neural networks

research, but the system is generally enough to be applicable in a wide variety of other domains as well.

You can download and install the package using `pip` in Python via the command line, as follows:

```
| C:\>Python -m pip install Tensorflow
```

TensorFlow contains many Python classes to construct, train, validate, and deploy a neural network with or without GPU support.

Keras for machine learning

Keras is also an open source free library. It is a high-level neural networks API written in Python and capable of running on top of TensorFlow. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

Actually, Keras is not a machine learning library; it is more of a wrapper that can simplify code writing for machine learning solutions. When we build and train a neural network in Keras, it calls TensorFlow at the backend and uses TensorFlow's classes to perform the required computation tasks.

These are the guiding principles of Keras:

- **User friendliness:** Keras is an API

designed for human beings, not machines. It puts user experience in the front and center. Keras follows best practices for reducing cognitive load; it offers consistent and simple APIs, minimizes the number of user actions required for common use cases, and provides clear and actionable feedback upon user error.

- **ModularityL:** A model is understood as a sequence or a graph of standalone, fully configurable modules that can be plugged together with as few restrictions as possible. In particular, neural layers, cost functions, optimizers, initialization schemes, activation functions, and regularization schemes are all standalone modules that you can combine to create new models.
- **Easy extensibility:** New modules are simple to add (as new classes and functions) and existing modules provide ample examples to be able to easily create new modules allows for total expressiveness, making Keras suitable

for advanced research.

- **Work with Python:** There are no separate model configuration files in a declarative format. Models are described in Python code, which is compact, is easier to debug, and allows ease of extensibility.

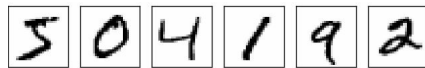
Digit classification using Keras and TensorFlow

As we have discussed before, we will have a dataset with images of handwritten digits. Here is an introduction to the dataset.

The MNIST database contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset. There have been a number of scientific papers on attempts to achieve the lowest error rate. One paper, by using a hierarchical system of CNNs, manages to get an error rate on the MNIST database of 0.23 percent. The original creators of the database keep a list of some of

the methods tested on it. In their original paper, they used a support vector machine to get an error rate of *0.8* percent.

Images in the dataset look like this:



So let's not waste our time and start implementing our very first neural network in Python.

Let's start the code by importing the supporting projects.

```
# Imports for array-handling and plotting
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
```

Keras already has the MNIST dataset as a sample dataset, so we can import it as it is. Generally, it downloads the data over the internet and stores it into the database. So, if your system does not have the dataset, Internet will be required to download it:


```
# Keras imports for the dataset and building  
our neural network  
from keras.datasets import mnist
```

Now, we will import the `Sequential` and `load_model` classes from the `keras.model` class. We are working with sequential networks as all layers will be in forward sequence only. We are not using any split in the layers. The `Sequential` class will create a sequential model by combining the layers sequentially. The `load_model` class will help us to load the trained model for testing and evaluation purposes:

```
#Import Sequential and Load model for creating  
and loading model  
from keras.models import Sequential, load_model
```

In the next line, we will call three types of layers from the `keras` library. Dense layer means a fully connected layer; that is, each neuron of current layer will have a connection to the each neuron of the previous as well as next layer.

The dropout layer is for reducing overfitting in our model. It randomly selects some neurons and does not use them for training

for that iteration. So there are less chances that two different neurons of the same layer learn the same features from the input. By doing this, it prevents redundancy and correlation between neurons in the network, which eventually helps prevent overfitting in the network.

The activation layer applies the activation function to the output of the neuron. We will use **rectified linear units (ReLU)** and the `softmax` function as the activation layer. We will discuss their operation when we use them in network creation:

```
#We will use Dense, Drop out and Activation layers
from keras.layers.core import Dense, Dropout, Activation
from keras.utils import np_utils
```

So we will start with loading our dataset by `mnist.load`. It will give us training and testing input and output instances.

Then, we will visualize some instances so that we know what kind of data we are

dealing with. We will use matplotlib to plot them.

As the images have gray values, we can easily plot a histogram of the images, which can give us the pixel intensity distribution:

```
#Let's Start by loading our dataset  
(X_train, y_train), (X_test, y_test) =  
mnist.load_data()  
#Plot the digits to verify  
plt.figure()  
for i in range(9):  
    plt.subplot(3,3,i+1)  
    plt.tight_layout()  
    plt.imshow(X_train[i], cmap='gray',  
interpolation='none')  
    plt.title("Digit: {}".format(y_train[i]))  
    plt.xticks([])  
    plt.yticks([])  
plt.show()
```

When we execute our code for the preceding code block, we will get the output as:

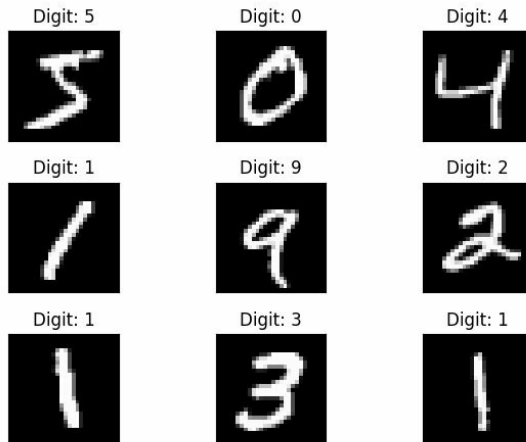


Figure 10.10: MNIST digit images sample

```
#Lets analyze histogram of the image
plt.figure()
plt.subplot(2,1,1)
plt.imshow(X_train[0], cmap='gray',
interpolation='none')
plt.title("Digit: {}".format(y_train[0]))
plt.xticks([])
plt.yticks([])
plt.subplot(2,1,2)
plt.hist(X_train[0].reshape(784))
plt.title("Pixel Value Distribution")
plt.show()
```

The histogram of an image will look like this:

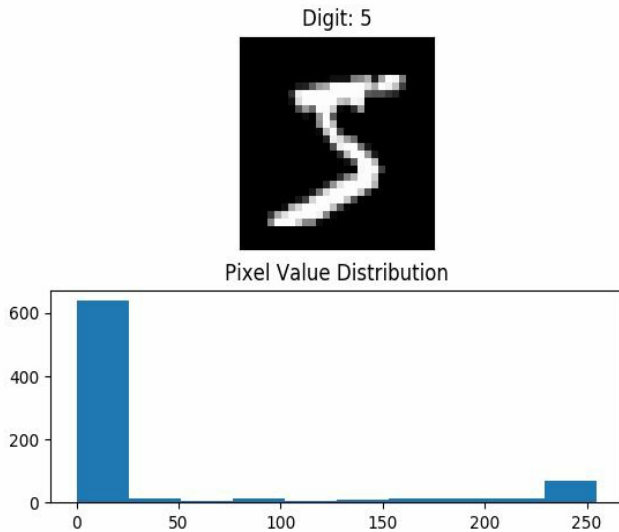


Figure 10.11: Pixel intensity distribution of an image

```
# Print the shape before we reshape and
normalize
print("X_train shape", X_train.shape)
print("y_train shape", y_train.shape)
print("X_test shape", X_test.shape)
print("y_test shape", y_test.shape)
```

Currently, this is shape of the dataset we have:

```
X_train shape (60000, 28, 28)
y_train shape (60000,)
X_test shape (10000, 28, 28)
y_test shape (10000,)
```

As we are working with 2D images, we cannot train them as with our neural network. For training 2D images, there are different types of neural networks available; we will discuss those in the future.

To remove this data compatibility issue, we will reshape the input images into 1D vectors of 784 values (as images have size 28X28). We have 60000 such images in training data and 10000 in testing:

```
# As we have data in image form convert it to  
row vectors  
X_train = X_train.reshape(60000, 784)  
X_test = X_test.reshape(10000, 784)  
X_train = X_train.astype('float32')  
X_test = X_test.astype('float32')
```

Normalize the input data into the range of 0 to 1 so that it leads to a faster convergence of the network. The purpose of normalizing data is to transform our dataset into a bounded range; it also involves relativity between the pixel values. There are various kinds of normalizing techniques available such as mean normalization, min-max normalization, and so on:

```
# Normalizing the data to between 0 and 1 to
help with the training
X_train /= 255
X_test /= 255

# Print the final input shape ready for
training
print("Train matrix shape", X_train.shape)
print("Test matrix shape", X_test.shape)
```

Let's print the shape of the data:

```
Train matrix shape (60000, 784)
Test matrix shape (10000, 784)
```

Now, our training set contains output variables as discrete class values; say, for an image of number eight, the output class value is eight. But our output neurons will be able to give an output only in the range of zero to one. So, we need to convert discrete output values to categorical values so that eight can be represented as a vector of zero and one with the length equal to the number of classes. For example, for the number eight, the output class vector should be:

```
8 = [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0]

# One-hot encoding using keras' numpy-related
utilities
n_classes = 10
```

```
| print("Shape before one-hot encoding: ",  
| y_train.shape)  
| Y_train = np_utils.to_categorical(y_train,  
| n_classes)  
| Y_test = np_utils.to_categorical(y_test,  
| n_classes)  
| print("Shape after one-hot encoding: ",  
| Y_train.shape)
```

After one-hot encoding of our output, the variable's shape will be modified as:

```
| Shape before one-hot encoding: (60000,)
| Shape after one-hot encoding: (60000, 10)
```

So, you can see that now we have an output variable of 10 dimensions instead of 1.

Now, we are ready to define our network parameters and layer architecture. We will start creating our network by creating a `Sequential` class object, `model`. We can add different layers to this `model` as we have done in the following code block.

We will create a network of an input layer, two hidden layers, and one output layer. As the input layer is always our data layer, it doesn't have any learning parameters. For hidden layers, we will use 512 neurons in

each. At the end, for a 10-dimensional output, we will use 10 neurons in the final layer:

```
# Here, we will create model of our ANN
# Create a linear stack of layers with the
sequential model
model = Sequential()

#Input Layer with 512 Weights
model.add(Dense(512, input_shape=(784,)))

#We will use relu as Activation
model.add(Activation('relu'))

#Put Drop out to prevent over-fitting
model.add(Dropout(0.2))

#Add Hidden layer with 512 neurons with relu
activation
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.2))

#This is our Output layer with 10 neurons
model.add(Dense(10))model.add(Activation('softma
```

After defining the preceding structure, our neural network will look something like this:

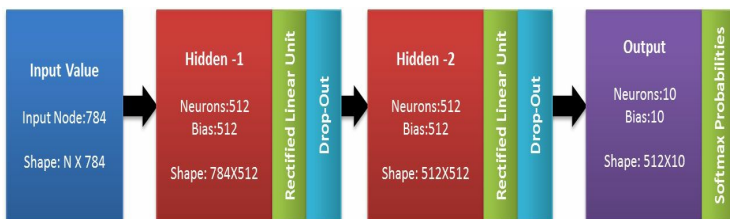


Figure 10.12: Feed-forward neural network for our problem's solution

The `shape` field in each layer shows the shape of the data matrix in that layer, and it is quite intuitive. As we first get the multiplication of input with length of 784 values to 512 neurons, the data shape at **Hidden-1** will be **784 X 512**. It will be calculated similarly for the other two layers.

We have used two different kinds of activation functions here. The first one is ReLU and the second one is softmax probabilities.

We will give some time to discuss these two. ReLU prevent the output of the neuron from becoming negative. The expression for `relu` function is:

$$f(x) = \max(0, x) \quad (8)$$

So if any neuron produces an output less than 0, it converts it to 0. We can write it in conditional form as:

$$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases} \quad (9)$$

You just need to know that ReLU is a slightly better activation function than sigmoid. If we plot a sigmoid function, it will look like:

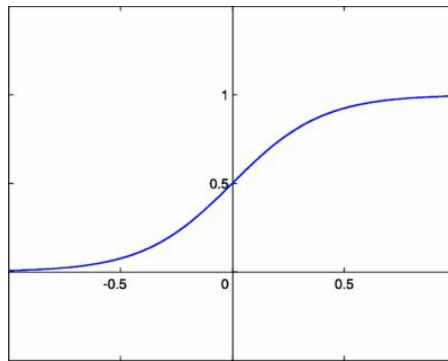


Figure 10.13: Sigmoid neuron

If you look closer, the sigmoid function starts getting saturated before reaching its minimum (**0**) or maximum (**1**) values. So at the time of gradient calculation, values in the saturated region result in a very small gradient. That causes a very small change in the weight values, which is not sufficient to optimize the cost function. Now, as we go

more backward during the backpropagation, that small change becomes smaller and almost reaches zero. This problem is known as the problem of **vanishing gradients**. So, in practical cases, we avoid sigmoid activation when our network has many stacked layers.

Whereas if we see the expression of ReLU activation, it is more like a straight line:

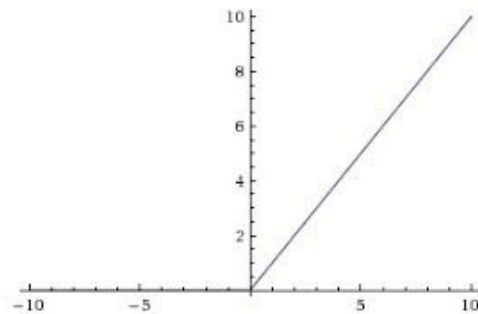


Figure 10.14: Rectification linear unit

So, the gradient of the preceding function will always be a non-zero value unless the output itself is a zero value. Thus, it prevents the problem of vanishing gradients.

We have discussed the significance of the

dropout layer earlier and I don't think that it is further required. We are using 20% neuron dropout during the training time. We will not use the dropout layer during the testing time.

Now, we are all set to train our very first ANN, but before starting training, we have to define the values of the network hyperparameters.

We will use SGD using adaptive momentum. There are many algorithms to optimize the performance of the SGD algorithm. You just need to know that adaptive momentum is a better choice than simple gradient descent because it modifies the learning rate using previous errors created by the network. So, there are less chances of getting trapped at the local minima or missing the global minima conditions. We are using SGD with ADAM, using its default parameters.

Here, we use `batch_size` of 128 samples. That means we will update the weights after calculating the error on these 128 samples. It is

a sufficient batch size for our total data population.

We are going to train our network for 20 epochs for the time being. Here, one epoch means one complete training cycle of all mini-batches.

Now, let's start training our network:

```
#Here we will be compiling the sequential model  
model.compile(loss='categorical_crossentropy',  
metrics=['accuracy'], optimizer='adam')  
  
# Start training the model and saving metrics  
in history  
history = model.fit(X_train, Y_train,  
                    batch_size=128, epochs=20,  
                    verbose=2,  
                    validation_data=(X_test, Y_test))
```

We will save our trained model on disk so that we can use it for further fine-tuning whenever required. We will store the model in the HDF5 file format:

```
# Saving the model on disk  
path2save =  
'E:/PyDevWorkspaceTest/Ensembles/Chapter_10/kera  
model.save(path2save)  
print('Saved trained model at %s ' % path2save)  
  
# Plotting the metrics
```

```

fig = plt.figure()
plt.subplot(2,1,1)
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='lower
right')

plt.subplot(2,1,2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper
right')
plt.tight_layout()
plt.show()

```

Let's analyze the loss with each iteration during the training of our neural network; we will also plot the accuracies for validation and test set. You should always monitor validation and training loss as it can help you know whether your model is underfitting or overfitting:

```

| Test Loss 0.0824991761778
| Test Accuracy 0.9813

```

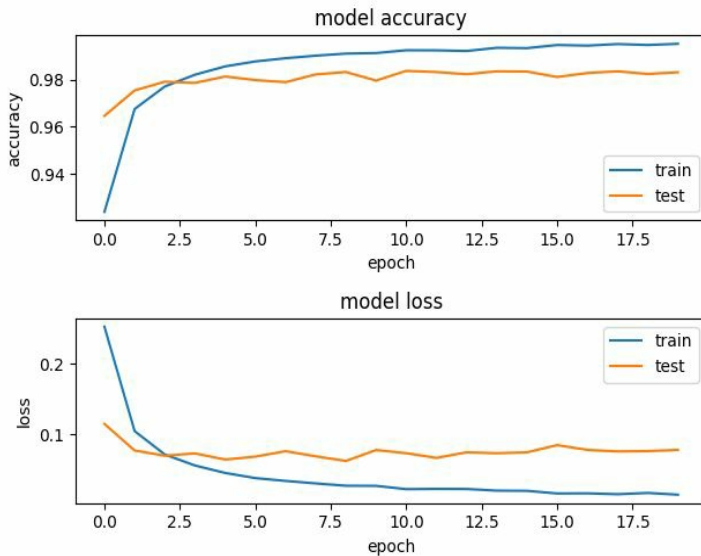


Figure 10.15: Performance of our model during training phase

As you can see, we are getting almost similar performance for our training and validation sets in terms of loss and accuracy. You can see how accuracy is increasing as the number of epochs increases. This shows that our network is learning.

Now, we have trained and stored our model. It's time to reload it and test it with the 10000 test instances:


```

#Let's load the model for testing data
path2save =
'D:/PyDevWorkspace/EnsembleMachineLearning/Chapt
mnist_model = load_model(path2save)

#We will use Evaluate function
loss_and_metrics = mnist_model.evaluate(X_test,
Y_test, verbose=2)
print("Test Loss", loss_and_metrics[0])
print("Test Accuracy", loss_and_metrics[1])

#Load the model and create predictions on the test set
mnist_model = load_model(path2save)
predicted_classes =
mnist_model.predict_classes(X_test)

#See which we predicted correctly and which not
correct_indices = np.nonzero(predicted_classes
== y_test)[0]
incorrect_indices =
np.nonzero(predicted_classes != y_test)[0]
print(len(correct_indices), " classified
correctly")
print(len(incorrect_indices), " classified
incorrectly")

```

So, here is the performance of our model on the test set:

```

9813  classified correctly
187  classified incorrectly

```

As you can see, we have misclassified 187 instances out of 10000, which I think is a very good accuracy on such a complex dataset. In the next code block, we will analyze such

cases where we detect false labels:

```
#Adapt figure size to accomodate 18 subplots
plt.rcParams['figure.figsize'] = (7,14)
plt.figure()

# plot 9 correct predictions
for i, correct in
enumerate(correct_indices[:9]):
    plt.subplot(6,3,i+1)
    plt.imshow(X_test[correct].reshape(28,28),
cmap='gray', interpolation='none')
    plt.title(
        "Predicted: {}, Truth:
{}".format(predicted_classes[correct],
y_test[correct]))
    plt.xticks([])
    plt.yticks([])
# plot 9 incorrect predictions
for i, incorrect in
enumerate(incorrect_indices[:9]):
    plt.subplot(6,3,i+10)

plt.imshow(X_test[incorrect].reshape(28,28),
cmap='gray',
interpolation='none')
    plt.title(
        "Predicted {}, Truth:
{}".format(predicted_classes[incorrect],
y_test[incorrect]))
    plt.xticks([])
    plt.yticks([])
plt.show()
```

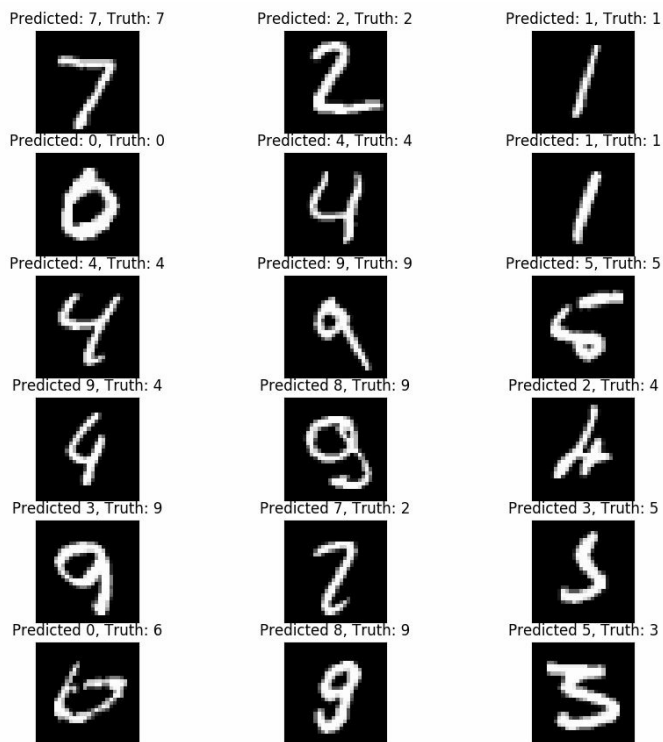


Figure 10.16: Classification of digit data using a trained model

If you look closely, our network is failing on such cases that are very difficult to identify by a human, too. So, we can say that we are getting quite a good accuracy from a very simple model.

In this section, we discussed the basic concepts of a feed-forward neural network. Now, we will extend this network for classifying images without reshaping them in 1D vectors, and we will also discuss how these networks do feature engineering by themselves.

Deep learning

While our neural network gives impressive performance, that performance is somewhat mysterious. The weights and biases in the network were discovered automatically, and that means we don't immediately have an explanation of how the network does what it does. Can we find some way to understand the principles by which our network is classifying handwritten digits? And given such principles, can we do better?

To put these questions more starkly, suppose that a few decades hence, neural networks lead to Artificial Intelligence (AI). Will we understand how such intelligent networks work? Perhaps the networks will be opaque to us, with weights and biases we don't understand because they've been learned automatically. In the early days of AI research, people hoped that the effort to build an AI would also help us understand the

principles behind intelligence and, maybe, the functioning of the human brain. But perhaps the outcome will be that we end up understanding neither the brain nor how AI works!

Suppose we want to detect faces in an image. How do we approach the problem? We can attack this problem the same way as we attacked the handwriting recognition one—using pixels in an image as input to a neural network, with the output from the network as a single neuron indicating either *Yes, it's a face* or *No, it's not a face*.

Let's suppose we do this, but we're not using a learning algorithm. Instead, we're going to try to design a network by hand, choosing appropriate weights and biases. How might we go about it? Forgetting neural networks entirely for the moment, a heuristic we could use is to decompose the problem into sub-problems: Does the image have an eye in the top left? Does it have an eye in the top right? Does it have a nose in the middle? Does it

have a mouth in the bottom middle? Is there hair on top? And so on.

If the answers to several of these questions are *yes* or even *probably yes*, then we'd conclude that the image is likely to be a face. Conversely, if the answers to most of the questions are *no*, then the image probably isn't a face.

Here's a possible architecture, with rectangles denoting the sub-networks. Note that this isn't intended as a realistic approach to solving a face-detection problem. Rather, it's done to help us build an intuition of how networks work. Here's the architecture:

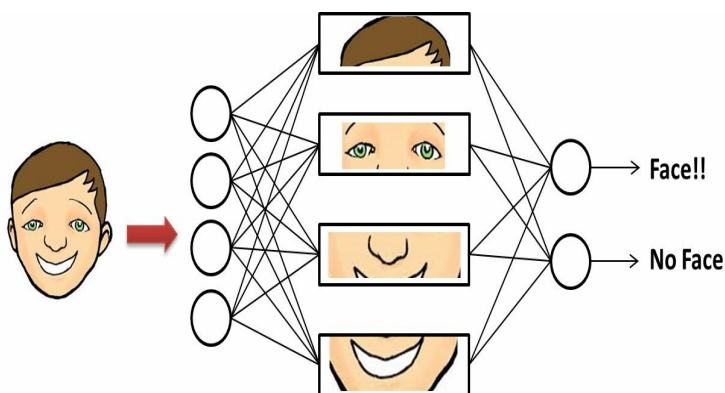


Figure 10.17: Deep learning intuition

It's also plausible that the sub-networks can be decomposed. Suppose we're considering the question, *Is there an eye in the top left?* This can be decomposed into questions such as:

- *Is there an eyebrow?*
- *Are there eyelashes?*
- *Is there an iris?*
- And so on

Of course, these questions should really include positional information as well—*Is the eyebrow in the top left and above the iris?* that kind of thing— but let's keep it simple.

These questions too can be broken down further and further through multiple layers. Ultimately, we'll be working with sub-networks that answer questions so simple that they can easily be answered at the level of single pixels. Those questions might, for example, be about the presence or absence of

very simple shapes at particular points in the image. Such questions can be answered by single neurons connected to the raw pixels in the image.

The end result is a network that breaks down a very complicated question—*Does this image show a face or not?*—into very simple questions answerable at the level of single pixels. It does this through a series of many layers, with early layers answering very simple and specific questions about the input image, and later layers building up a hierarchy of ever more complex and abstract concepts. Networks with this kind of many-layer structure—of two or more hidden layers—are called **deep neural networks**.

Convolutional Neural Networks

We have worked with fully connected neural networks (each neuron has a connection with each neuron of the next and previous layer). In this section, we will discuss a different kind of neural network that is pretty useful in image classification as well as the segmentation of pixels.

As we have seen earlier, we have to first reshape our image into a 1D vector and then feed it into our neural network. Due to this, we lose the spatial relationship between pixels. Let's understand the process of reshaping we have used earlier with the same face detection problem we discussed in previous section.

When we convert a face image into a 1D vector, this is what happens inside the

function. We pick up each row of the image matrix and append it in cascade form, as shown here:

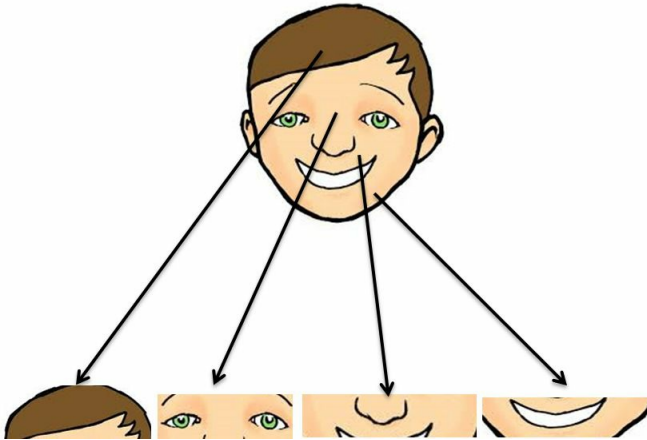


Figure 10.18: 1D data representation of an image

As you can see, in the preceding figure, there is no positional relationship left between the different rows of a face image. Due to this, our final classifier may be less sensitive towards positional changes. This may become a problem during a face recognition task where small structural details are required to differentiate between two different faces. What if we can train our images with spatial

context? This is where the concept of CNNs comes into the picture.

These networks use a special architecture that is particularly well adapted to classify images. Using this architecture makes convolutional networks fast to train. This, in turn, helps us train deep, many-layer networks, which are very good at classifying images.

Today, deep convolutional networks or some close variants are used in most neural networks for image recognition.

CNNs use three basic ideas: local receptive fields, shared weights, and pooling. Let's look at each of these ideas in turn.

Local receptive fields

In the fully connected layers shown earlier, the inputs were depicted as a vertical line of neurons. In a convolutional net, it'll instead help to think of the inputs as a 18×18 square of neurons, whose values correspond to the 18×18 pixel intensities we're using as inputs:

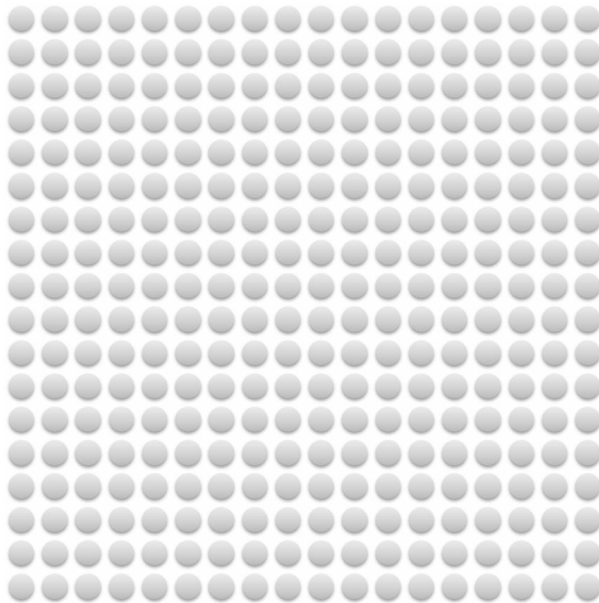


Figure 10.19: Representation of a digital image

As usual, we'll connect the input pixels to a layer of hidden neurons, but we won't connect every input pixel to every hidden neuron. Instead, we only make connections in small, localized regions of the input image.

To be more precise, each neuron in the first hidden layer will be connected to a small region of the input neurons, for example, a 5×5 region corresponding to 25 input pixels. So, for a particular hidden neuron, we might have connections that look like this:

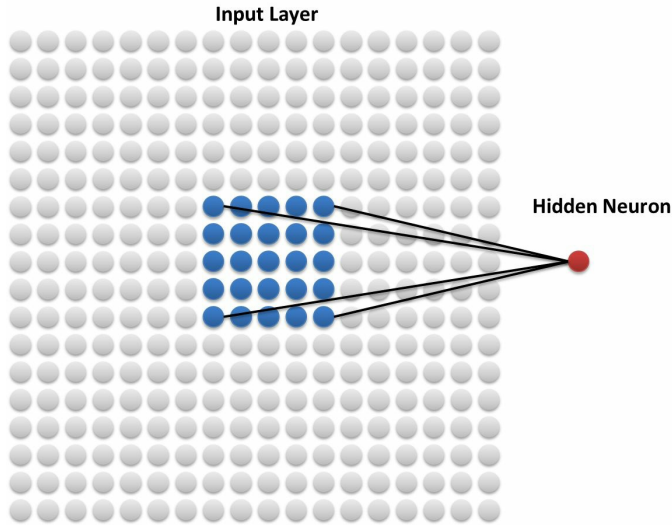


Figure 10.20: Local receptive field

Such a region in the input image is called the **local receptive field** for the hidden neuron. It's a little window on the input pixels. Each connection learns a weight, and the hidden neuron learns an overall bias as well. You can think of that hidden neuron as learning to analyze its particular local receptive field.

We then slide the local receptive field across the entire input image. For each local

receptive field, there is a different hidden neuron in the first hidden layer. To illustrate this concretely, let's start with a local receptive field in the top-left corner:

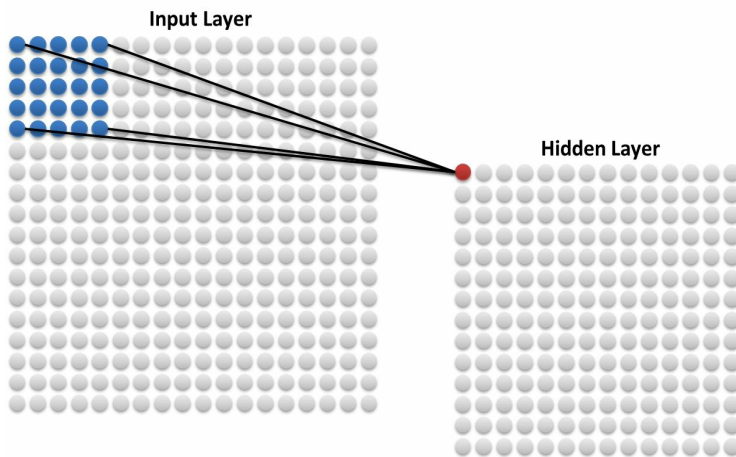


Figure 10.21: Calculation for first hidden neuron

Then we slide the local receptive field over by one pixel to the right (that is, by one neuron), to connect to a second hidden neuron:

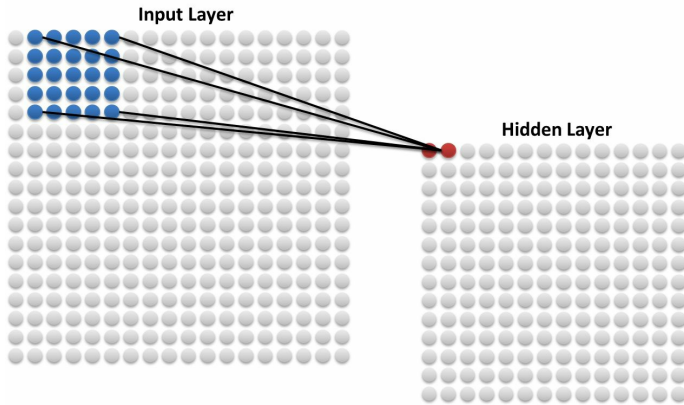


Figure 10.22: Calculation for first hidden neuron

We will continue this process of building up the first hidden layer. Note that if we have an 18×18 input image and 5×5 local receptive fields, then there will be 14×14 neurons in the hidden layer. This is because we can only move the local receptive field 13 neurons across (or 13 neurons down), before colliding with the right-hand side (or bottom) of the input image.

I've shown the local receptive field being moved by one pixel at a time. In fact, sometimes, a different stride length is used. For instance, we might move the local

receptive field two pixels to the right (or down), in which case we'd say a stride length of two is used.

Shared weights and biases

I've said that each hidden neuron has a bias and 5×5 weights connected to its local receptive field. What I did not yet mention is that we're going to use the same weights and bias for each of the 14×14 hidden neurons. In other words, for the $(j, k)^{th}$ hidden neuron, the output is:

$$\sigma \left(b + \sum_{l=0}^4 \sum_{m=0}^4 w_{l,m} a_{j+l, k+m} \right) \quad (10)$$

Here, σ is the neural activation function, perhaps the sigmoid function we used in earlier chapters. b is the shared value for the bias. $w_{l,m}$ is a 5×5 array of shared weights. And, finally, we use a_x, y to denote the input activation at position x, y .

This means that all the neurons in the first hidden layer detect exactly the same feature, just at different locations in the input image. To see why this makes sense, suppose the weights and biases are such that the hidden neuron can pick out, say, a vertical edge in a particular local receptive field. This ability is also likely to be useful at other places in the image, and so it is useful to apply the same feature detector everywhere in the image. To put it in slightly more abstract terms, convolutional networks are well adapted to the translation invariance of images: move a picture of a cat (say) a little sideways and it's still a picture of a cat.

For this reason, we sometimes call the map from the input layer to the hidden layer a **feature map**. We call the weights defining the feature map the shared weights, and we call the bias defining the feature map in this way the shared bias. Shared weights and bias are often said to define a kernel or filter.

The network structure I've described so far

can detect just a single kind of localized feature. To do image recognition, we'll need more than one feature map. So a complete convolutional layer consists of several different feature maps:

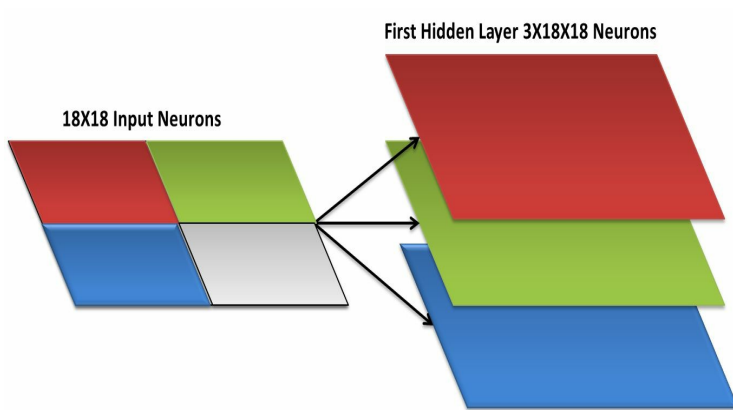


Figure 10.23: Feature map representation

In the example shown, there are three feature maps. Each feature map is defined by a set of 5×5 shared weights and a single shared bias. The result is that the network can detect three different kinds of features, with each feature being detectable across the entire image.

I've shown just three feature maps to keep the preceding diagram simple. However, in practice, convolutional networks may use more (and perhaps many more) feature maps.

One of the early convolutional networks, LeNet-5, used 6 feature maps, each associated to a 5×5 local receptive field, to recognize MNIST digits. So the example illustrated before is actually pretty close to LeNet-5.

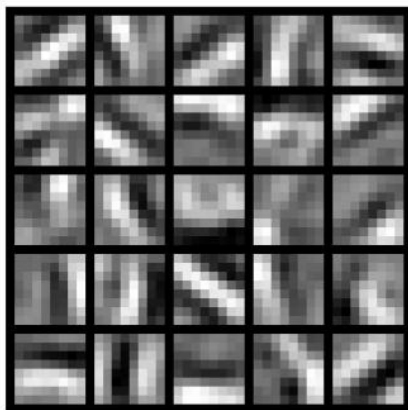


Figure 10.24: Features (Shared Weights) extracted from an inner layer of hidden convolution layers

The preceding image shows what a learned feature matrix looks like. Whiter blocks mean a smaller (typically, more negative) weight, so the feature map responds less to the corresponding input pixels. Darker blocks mean a larger weight, so the feature map responds more to the corresponding input pixels. Very roughly speaking, these images show the type of features the convolutional layer responds to.

Incidentally, the name convolutional comes from the fact that the operation in equation (10) is sometimes known as convolution. A little more precisely, people sometimes write that equation as $a_1 = \sigma(b + w * a_0)$, where a_1 denotes the set of output activations from one feature map, a_0 is the set of input activations, and $*$ is called a **convolution operation**. We're not going to make any deep use of the mathematics of convolutions, so you don't need to worry too much about this connection. But it's at least worth knowing where the name comes from.

Pooling layers

In addition to the convolutional layers just described, CNNs also contain pooling layers. Pooling layers are usually used immediately after convolutional layers. What they do is simplify the information in the output from the convolutional layer.

In detail, a pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map. For instance, each unit in the pooling layer may summarize a region of (say) 2×2 neurons in the previous layer. As a concrete example, one common procedure for pooling is known as **max-pooling**. In max-pooling, a pooling unit simply outputs the maximum activation in the 2×2 input region, as illustrated in the following diagram:

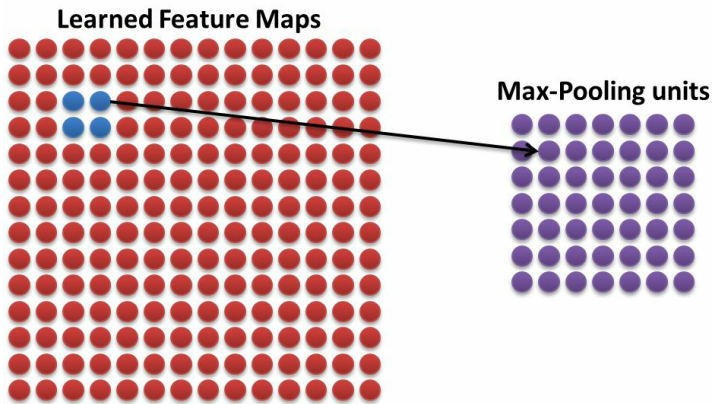


Figure 10.25: Pooling layer

Note that since we have 14×14 neurons of output from the convolutional layer, after pooling, we have 7×7 neurons.

As mentioned before, the convolutional layer usually involves more than a single feature map. We apply max-pooling to each feature map separately. So, if there were three feature maps, the combined convolutional and max-pooling layers would look like:

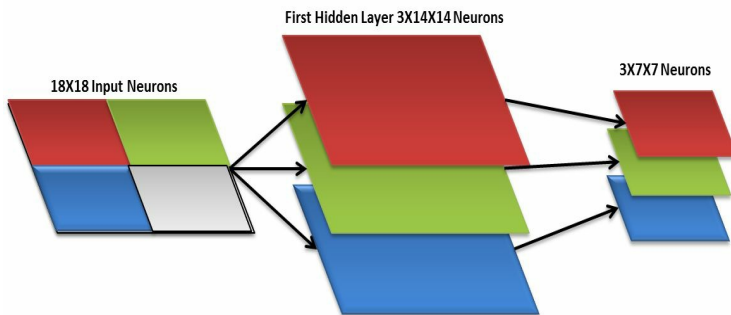


Figure 10.26: Max pooling on convolutional output

We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information. The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features. A big benefit is that there are many fewer-pooled features, and so this helps reduce the number of parameters needed in later layers.

Max-pooling isn't the only technique used for pooling. Another common approach is known as **L2 pooling**. Here, instead of taking the maximum activation of a 2×2 region of

neurons, we take the square root of the sum of squares of activations in the 2×2 region. While the details are different, the intuition is similar to max-pooling. L2 pooling is a way of condensing information from the convolutional layer. In practice, both techniques have been widely used. Sometimes, people use other types of pooling operations. If you're really trying to optimize performance, you may use validation data to compare several different approaches to pooling and choose the approach that works best. However, we're not going to worry about that kind of detailed optimization.

Combining all the layers

If we consider our older problem of MNIST digits classification, we can create a convolutional neural network by putting layers as described in the previous paragraph; we just need to add an output layer with 10 neurons (as we are expecting 10 classes at the output).

The network begins with 28×28 input neurons, which are used to encode the pixel intensities for the MNIST image. This is then followed by a convolutional layer using a 5×5 local receptive field and three feature maps.

The result is a layer of $3 \times 24 \times 24$ hidden feature neurons. The next step is a max-pooling layer applied to 2×2 regions across each of the three feature maps.

The result is a layer of $3 \times 12 \times 12$ hidden feature neurons:

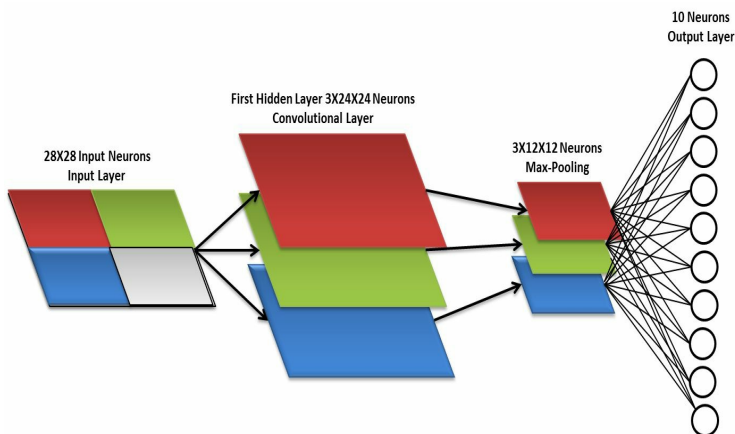


Figure 10.27: Putting it all together

The final layer of connections in the network is a fully connected layer. That is, this layer connects every neuron from the max-pooled layer to every one of the 1,010 output neurons. This fully connected architecture is the same as what we used in earlier chapters. Note, however, that in this diagram, I've used a single arrow for simplicity instead of showing all the connections. Of course, you can easily imagine the connections.

Implementation of CNN in Python

We have seen how we can design a CNN model. As we have discussed earlier, CNN architecture contains more than one set of convolutional and max-pooling layers along with activation functions.

Here, we will deal with the same MNIST digit classification problem to understand the working of a CNN. This problem is known as the *Hello World!* program of the deep learning domain. This model is adapted from Yann Lecunn's LeNet model, so we will also name our CNN architecture as LeNet.

We will create a convolutional architecture with two sets of convolutional-relu-max pooling and one dense layer for classification of extracted feature maps.

Let's create a definition for constructing our CNN:

```
#Here is our Network definition
def LeNet(width, height, depth, classes,
weightsPath=None):

    #Initialize model
    model = Sequential()

    #First set of Convolution ==>
Activation(ReLu) ==> pooling(Max Pooling)

    model.add(Convolution2D(20,5,5,border_mode='the
    same',input_shape
    (depth,height,width)))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=
    (2,2),strides=(2,2)))

    #Second set of Convolution ==>
Activation(ReLu) ==> pooling(Max Pooling)

    model.add(Convolution2D(50,5,5,border_mode="the
    same"))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=
    (2,2),strides=(2,2)))

    #Fully connected Layer FC ==> ReLu
    model.add(Flatten())
    model.add(Dense(500))
    model.add(Activation("relu"))
    model.add(Dense(classes))
    model.add(Activation("softmax"))

    # If a pre-trained model is supplied
if weightsPath is not None:
        model.load_weights(weightsPath)

    #return the constructed model
```

```
| return model
```

The architecture of the preceding model looks something like:

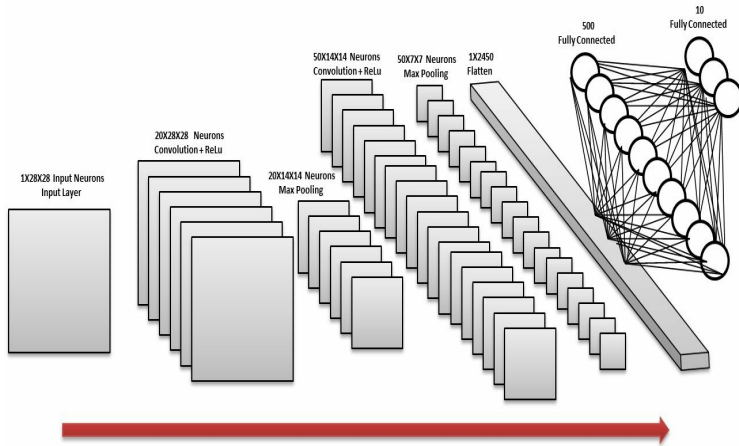


Figure 10.28: Architecture of a multilayered CNN

Now we will start building the module by importing all the packages. This code is exactly similar to the code we have used to create a simple neural network. The entire code is self-explanatory, so I don't think there is any need for further explanation:

```
# Imports for array-handling and plotting
from keras import callbacks
from keras.datasets import mnist
```



```

from keras.models import load_model
from keras.utils import np_utils
from Codes.Networks.LeNet import LeNet
import matplotlib.pyplot as plt
import numpy as np
from keras.utils.visualize_util import plot

# Keras imports for the data set and building
our neural network
#Let's Start by loading our data set
(X_train, y_train), (X_test, y_test) =
mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], 1,
28, 28).astype('float32')
X_test = X_test.reshape(X_test.shape[0], 1, 28,
28).astype('float32')

# Print the shape before we reshape and
normalize
print("X_train shape", X_train.shape)
print("y_train shape", y_train.shape)
print("X_test shape", X_test.shape)
print("y_test shape", y_test.shape)

X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# Normalizing the data to between 0 and 1 to
help with the training
X_train /= 255
X_test /= 255

# Print the final input shape ready for
training
print("Train matrix shape", X_train.shape)
print("Test matrix shape", X_test.shape)

#One-hot encoding using keras' numpy-related
utilities
n_classes = 10

# print("Shape before one-hot encoding: ",
y_train.shape)

```

```

Y_train = np_utils.to_categorical(y_train,
n_classes)
Y_test = np_utils.to_categorical(y_test,
n_classes)

print("Shape after one-hot encoding: ",
Y_train.shape)

```

As we know, here our data shape will not be a 1D vector:

```

X_train shape (60000, 1, 28, 28)
y_train shape (60000,)
X_test shape (10000, 1, 28, 28)
y_test shape (10000,)
Train matrix shape (60000, 1, 28, 28)
Test matrix shape (10000, 1, 28, 28)
Shape after one-hot encoding: (60000, 10)

```

```

#Import necessary packages for building our CNN
from keras.models import Sequential

#We will Need convolutional layer for feature
maps and max pooling layers
from keras.layers.convolutional import
Convolution2D,MaxPooling2D

#Flatten layer will convert 2D image into 1D
array for last layer computations
from keras.layers.core import
Activation,Flatten,Dense

#Here is our Network definition
def LeNet(width, height, depth, classes,
weightsPath=None):

    #Initialize model

```

```

    model = Sequential()

    #First set of Convolution ==>
Activation(ReLu) ==> pooling(Max Pooling)

    model.add(Convolution2D(20,5,5,border_mode='the
    same',input_shape
    (depth,height,width)))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=
    (2,2),strides=(2,2)))

    #Second set of Convolution ==>
Activation(ReLu) ==> pooling(Max Pooling)

    model.add(Convolution2D(50,5,5,border_mode="the
    same"))
        model.add(Activation("relu"))
        model.add(MaxPooling2D(pool_size=
    (2,2),strides=(2,2)))

    #Fully connected Layer FC ==> ReLu
    model.add(Flatten())
    model.add(Dense(500))
    model.add(Activation("relu"))
    model.add(Dense(classes))
    model.add(Activation("softmax"))

    # If a pre-trained model is supplied
if weightsPath is not None:
        model.load_weights(weightsPath)

    #return the constructed model
    return model

#Define Path to model Storage
path2save =
'E:/PyDevWorkspaceTest/Ensembles/Chapter_10/keras

#Call our model structure
model = LeNet(28, 28, 1, 10)
plot(model,show_shapes=True,show_layer_names=True,
Ensembles/Chapter_10/keras_mnist_lenet.pdf')

```

#We will only store the best model with highest validation accuracy

```
modelCheck =  
callbacks.ModelCheckpoint(path2save,  
monitor='val_acc', verbose=0,  
save_best_only=True, save_weights_only=False,  
mode='auto')
```

#Optimizer will be adaptive momentum with categorical loss

```
model.compile(optimizer="Adam", loss =  
"categorical_crossentropy", metrics=  
["accuracy"])
```

Start training the model and saving metrics in history

```
history = model.fit(X_train, Y_train,  
batch_size=128, epochs=20,  
verbose=2,  
validation_data=(X_test, Y_test),  
callbacks= [modelCheck])
```

Saving the model on disk

```
model.save(path2save)  
print('Saved trained model at %s ' % path2save)
```

Plotting the metrics

```
fig = plt.figure()  
plt.subplot(2,1,1)  
plt.plot(history.history['acc'])  
plt.plot(history.history['val_acc'])  
plt.title('model accuracy')  
plt.ylabel('accuracy')  
plt.xlabel('epoch')  
plt.legend(['train', 'test'], loc='lower  
right')  
plt.subplot(2,1,2)  
plt.plot(history.history['loss'])  
plt.plot(history.history['val_loss'])  
plt.title('model loss')  
plt.ylabel('loss')  
plt.xlabel('epoch')
```

```
plt.legend(['train', 'test'], loc='upper
right')
plt.tight_layout()
plt.show()
```

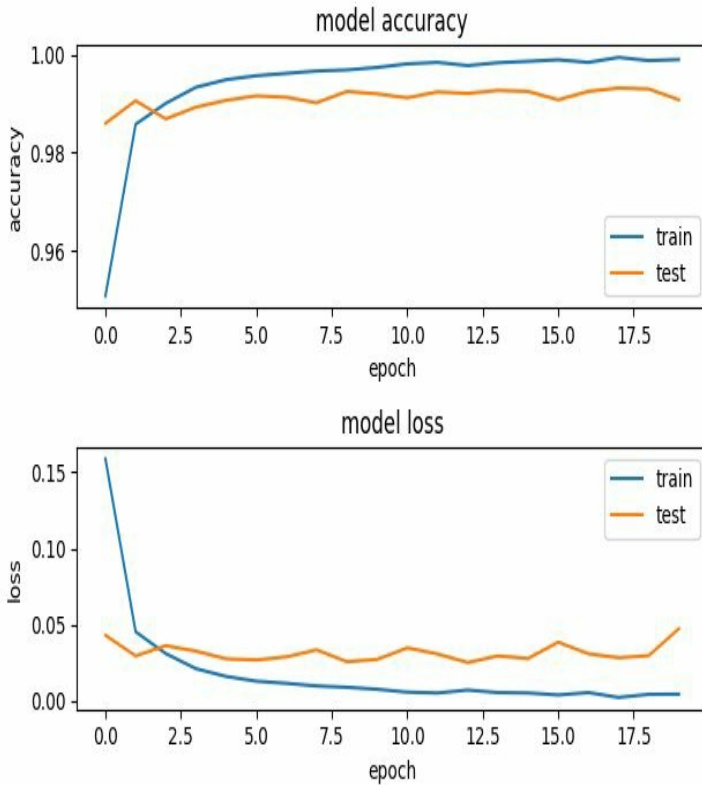


Figure 10.29: CNN model performance during the training phase

```
#Let's load the model for testing data
```

```
path2save =  
'E:/PyDevWorkspaceTest/Ensembles/Chapter_10/kera  
mnist_model = load_model(path2save)
```

```
#We will use Evaluate function  
loss_and_metrics = mnist_model.evaluate(X_test,  
Y_test, verbose=2)  
print("Test Loss", loss_and_metrics[0])  
print("Test Accuracy", loss_and_metrics[1])
```

```
Test Loss 0.0315236214503  
Test Accuracy 0.9935
```

```
#Load the model and create predictions on the  
test set  
mnist_model = load_model(path2save)  
predicted_classes =  
mnist_model.predict_classes(X_test)
```

```
#See which we predicted correctly and which not  
correct_indices = np.nonzero(predicted_classes  
== y_test)[0]  
incorrect_indices =  
np.nonzero(predicted_classes != y_test)[0]  
  
print(len(correct_indices), " classified  
correctly")  
print(len(incorrect_indices), " classified  
incorrectly")
```

```
9935 classified correctly  
65 classified incorrectly
```

```
#Adapt figure size to accomodate 18 subplots  
plt.rcParams['figure.figsize'] = (7,14)  
plt.figure()  
# plot 9 correct predictions  
for i, correct in  
enumerate(correct_indices[:9]):  
    plt.subplot(6,3,i+1)  
    plt.imshow(X_test[correct].reshape(28,28),  
cmap='gray', interpolation='none')  
    plt.title(
```

```

        "Predicted: {}, Truth:
{}".format(predicted_classes[correct],
y_test[correct]))
        plt.xticks([])
        plt.yticks([])

# plot 9 incorrect predictions
for i, incorrect in
enumerate(incorrect_indices[:9]):
    plt.subplot(6,3,i+10)

plt.imshow(X_test[incorrect].reshape(28,28),
cmap='gray',
interpolation='none')
    plt.title(
        "Predicted {}, Truth:
{}".format(predicted_classes[incorrect],
y_test[incorrect]))
        plt.xticks([])
        plt.yticks([])
plt.show()

```

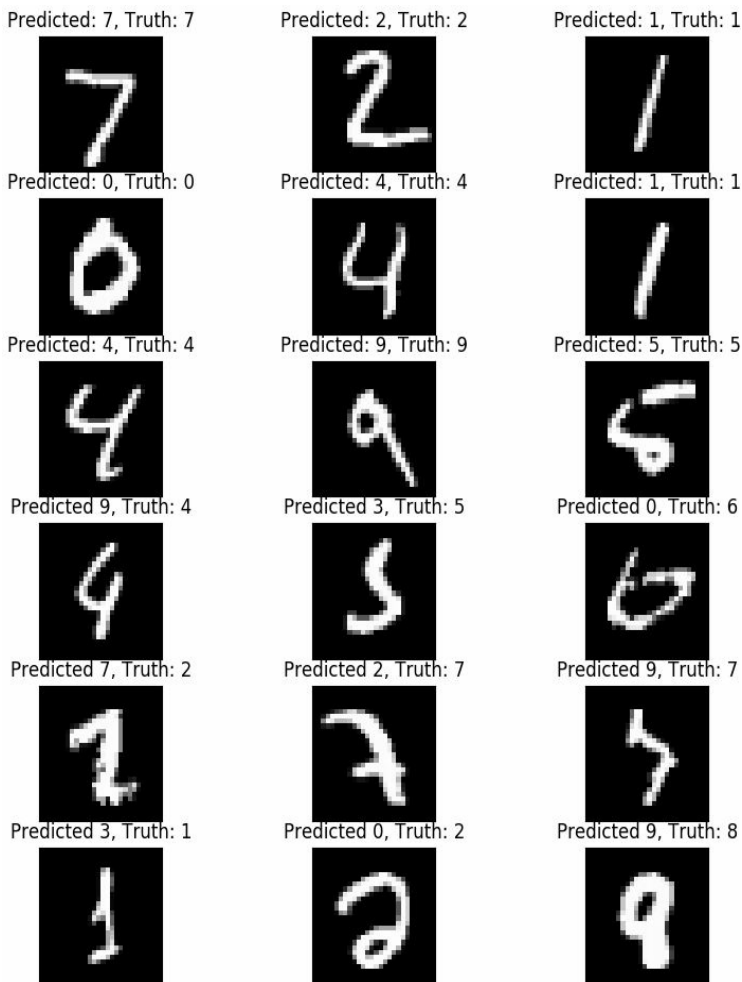


Figure 10.30: CNN model output on test data

If you notice, our classifier fails in very close cases. There are chances that a human can

fail in those cases, too. If we summarize the comparative performance for a normal NN and a CNN, we can write down the following for our example:

Parameters	Neural Nets	Conv Nets
Input data shape	1D vector	2D vector
Test loss	0.082	0.031
Test accuracy	98.13	99.35%
Correct instances	9813	9935

Table 10.1: Comparison of two networks

The preceding performance summary shows the clear victory of CNNs over normal feed-forward networks. We can see here that the correct instances for CNN are more in number than a for normal network.

Recurrent Neural Networks

I think every one of you has a smartphone, and you may be using many text-message-based services to text your loved ones.

Whenever you type a word, you get a suggestion for the next word, and at least 75% of the time the prediction is correct.

How can it be done? How does your smartphone know what the next word should be? Well, there is a predictive algorithm behind that and its work is to predict the next word using the previous word as input so that a meaningful sentence can be created. The technology is known as word embedding. We will try to implement such a kind of word prediction module with the use of **Recurrent Neural Networks (RNNs)**.

RNNs are designed to use sequential information. In a traditional neural network,

we assume that all inputs (and outputs) are independent of each other. But for many tasks, this is not true. If you want to predict the next word in a sentence, you better know which words came before it. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being dependent on the previous computations.

Another way to think about RNNs is that they have a **memory** that captures information about what has been calculated so far. In theory, RNNs can make use of information in arbitrarily long sequences, but in practice, they are limited to looking back only a few steps. Here is what a typical RNN looks like:

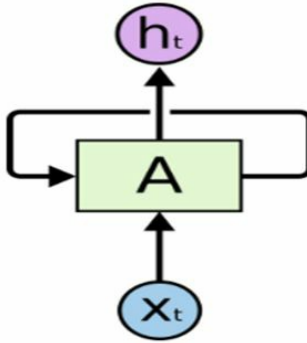


Figure 10.31: Recurrent Neural Network

In the preceding diagram, x_t is an input word or a character. h_t is the predicted output for that word and **A** is the neural units responsible for this prediction. Here you can see a loop at work, it emerges from **A** and the feedback goes to **A**. We will talk about this in a few moments. I have told earlier that a RNN is used to utilize sequential information. If we elaborate the neural units in the preceding figure, they will look something like:

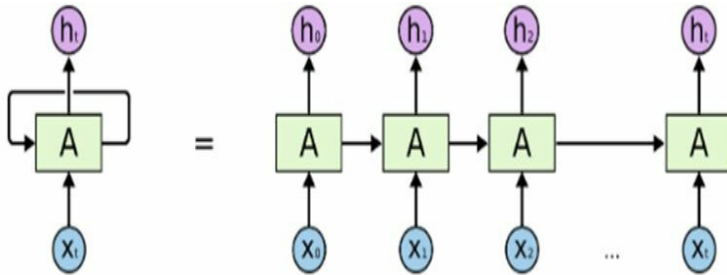


Figure 10.32: Unrolled Recurrent neural network

As you can see, each neural unit takes an input \mathbf{X} and gives an output \mathbf{h} , while also passing the information to the next unit. This is the important part of the RNN; here each unit shares its information with other neurons, which helps to use sequential information such as word or character prediction tasks.

Let's understand the structure of RNN and its working.

How RNN works (unrolling RNN)

RNN are a type of neural network where outputs from previous time steps are taken as inputs for the current time step.

We can demonstrate this with a picture.

Here, we can see that the network takes the output of the network from the previous time step as input and uses the internal state from the previous time step as a starting point for the current time step:

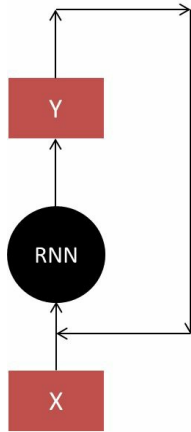


Figure 10.33: RNN unit

RNNs are fit for making predictions over many time steps. We can simplify the model by unfolding or unrolling the RNN graph over the input sequence.

Unrolling the forward pass

Consider the case where we have multiple time steps of input ($X(t), X(t+1), \dots$), multiple time steps of internal state ($u(t), u(t+1), \dots$), and multiple time steps of output ($y(t), y(t+1), \dots$).

We can unfold the preceding network schematic into a graph without any cycles.

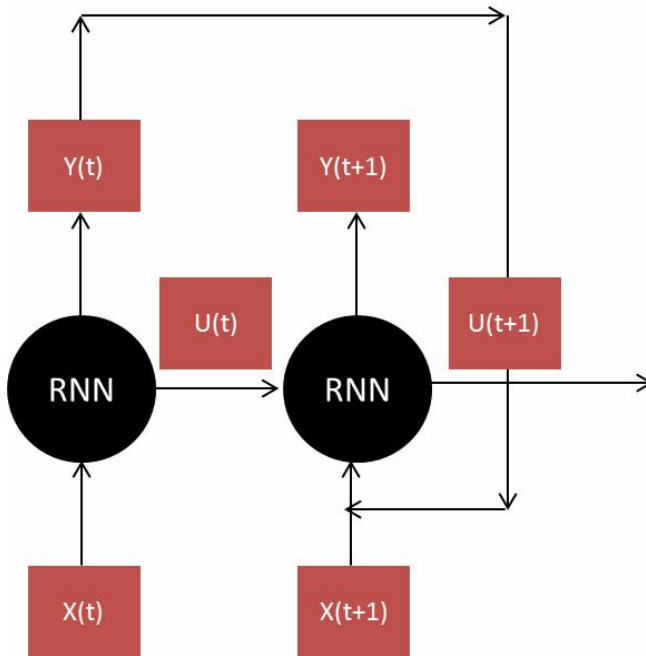


Figure 10.34: Forward pass in RNN

We can see that the cycle is removed and that the output ($\mathbf{Y(t)}$) and internal state ($\mathbf{U(t)}$) from the previous time step are passed on to the network as inputs for processing the next time step.

The key in this conceptualization is that the network (**RNN**) does not change between the unfolded time steps. Specifically, the same

weights are used for each time step and it is only the outputs and the internal states that differ.

In this way, it is as though the whole network (topology and weights) is copied for each time step in the input sequence.

Further, each copy of the network may be thought of as an additional layer of the same feed-forward neural network.

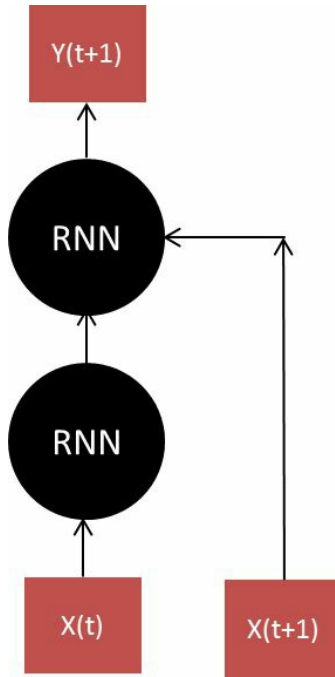


Figure 10.35: Multiple RNN units in cascade (deep RNN)

This is a useful conceptual tool and visualization to help in understanding what is going on in the network during the forward pass. Also, it may or may not be the way the network is implemented by the deep learning library.

Unrolling the backward pass

The idea of network unfolding plays a bigger part in the way RNNs are implemented for backward pass.

Importantly, the backpropagation of an error for a given time step depends on the activation of the network at the prior time step. In this way, backward pass requires the conceptualization of unfolding the network.

The error is propagated back to the first input time step of the sequence so that the error gradient can be calculated and the weights of the network can be updated.

*Like standard backpropagation,
[backpropagation through time]
consists of a repeated
application of the chain rule.*



The subtlety is that, for recurrent networks, the loss function depends on the activation of the hidden layer, not only through its influence on the output layer but also through its influence on the hidden layer at the next time step.

Unfolding the recurrent network graph also introduces additional concerns. Each time step requires a new copy of the network, which in turn takes up memory, especially for larger networks with thousands or millions of weights, the memory requirements of training large recurrent networks can quickly balloon as the number of time steps climbs to the hundreds.

It is required to unroll the RNNs by the length of the input sequence. By unrolling an RNN N times, every activation of the neurons inside the network is



*replicated N times, which consumes a huge amount of memory, especially when the sequence is very long. This hinders a small footprint implementation of online learning or adaptation. Also, this full unrolling makes parallel training with multiple sequences inefficient on shared memory models such as **graphics processing units (GPUs)**.*

Backpropagation Through Time

Backpropagation Through Time (BPTT), is the training algorithm used to update weights in RNNs such as LSTMs.

To effectively frame sequence prediction problems for RNNs, you must have a strong conceptual understanding of what BPTT is doing and how configurable variations such as Truncated BPTT will affect skill, stability, and speed when training your network.

Backpropagation training algorithm

Backpropagation refers to two things:

- The mathematical method used to calculate derivatives and an application of the derivative chain rule
- The training algorithm for updating network weights to minimize error

It is this latter understanding of backpropagation that we are using here.

The goal of the backpropagation training algorithm is to modify the weights of a neural network in order to minimize the error of the network outputs compared to some expected output in response to the corresponding inputs.

It is a supervised learning algorithm that

allows the network to be corrected with regard to the specific errors made.

The general algorithm is as follows:

1. Present a training input pattern and propagate it through the network to get an output.
2. Compare the predicted outputs to the expected outputs and calculate the error.
3. Calculate the derivatives of the error with respect to the network weights.
4. Adjust the weights to minimize the error.
5. Repeat.

The backpropagation training algorithm is suitable for training feed-forward neural networks on fixed-sized input-output pairs, but what about sequence data that may be temporally ordered?

Backpropagation Through Time

BPTT, is the application of the backpropagation training algorithm to a recurrent neural network applied to sequence data, such as a time series.

A RNN is shown with one input each time step and predicts one output.

Conceptually, BPTT works by unrolling all input time steps. Each time step has one input time step, one copy of the network, and one output. Errors are then calculated and accumulated for each time step. The network is rolled back up and the weights are updated.

Spatially, each time step of the unrolled RNN may be seen as an additional layer, given the order dependence of the problem, and the internal state from the previous time

step is taken as an input on the subsequent time step.

We can summarize the algorithm as follows:

1. Present a sequence of time steps of input and output pairs to the network.
2. Unroll the network. Then calculate and accumulate errors across each time step.
3. Roll up the network and update the weights.
4. Repeat.

BPTT can be computationally expensive as the number of time steps increases.

If the input sequences are comprised of thousands of time steps, then this will be the number of derivatives required for a single update weight update. This can cause the weights to vanish or explode (go to zero or overflow) which can make slow learning and model skill may be noisy.

Long Short-Term Memory networks

Long Short Term Memory (LSTM) networks—usually just called LSTMs—are a special kind of RNNs that are capable of learning long-term dependencies. They were introduced by Hochreiter and Schmidhuber (1997). They work tremendously well on a large variety of problems and are now widely used.

LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

All RNNs are in the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single

tanh layer:

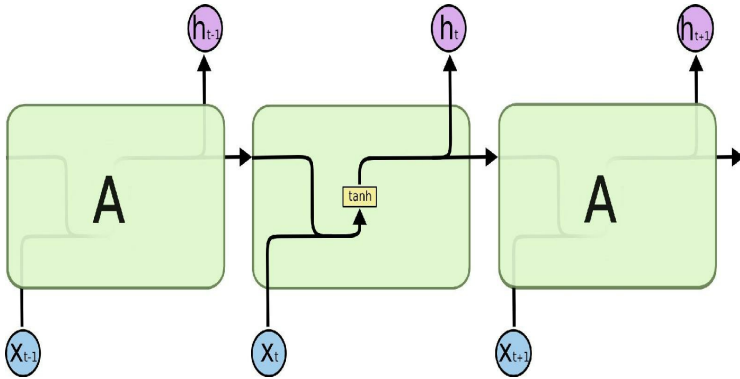


Figure 10.36: The repeating module in a standard RNN contains a single layer

LSTMs also have this chain-like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are four, interacting in a very special way:

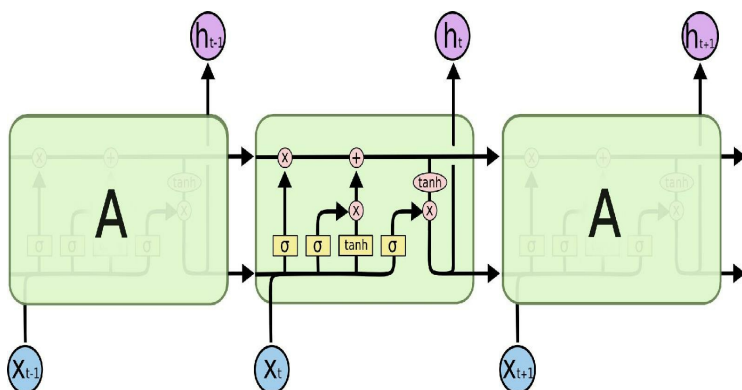


Figure 10.37: The repeating module in an LSTM contains four interacting layers

Don't worry about the details of what's going on. We'll walk through the LSTM diagram step by step later. For now, let's just try to get comfortable with the notation we'll be using:

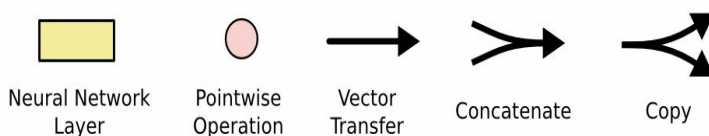


Figure 10.38: Notations

In the preceding diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles

represent point-wise operations, such as vector addition, while the yellow boxes are the learned neural network layers. Lines merging denote concatenation, while a line forking denotes that its content is being copied and the copies are going to different locations.

The idea behind LSTMs

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.

The cell state is, kind of, like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged:

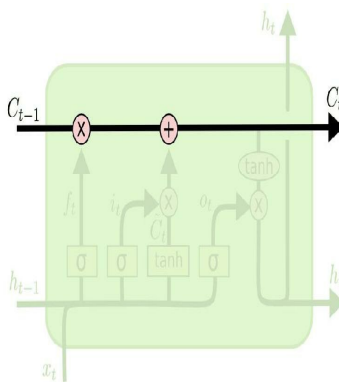


Figure 10.39: Forward pass

The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.

Gates are a way to optionally let information through. They are composed of a sigmoid neural net layer and a point-wise multiplication operation:

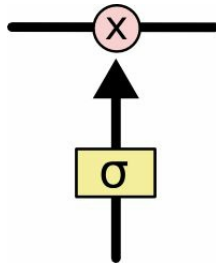


Figure 10.40: Sigmoid neuron activation

The sigmoid layer outputs numbers between zero and one, describing how much of each component should be let through. A value of 0 means *let nothing through* while a value of 1 means *let everything through*!

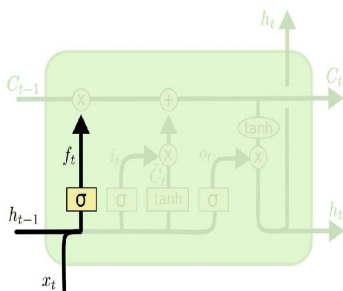
An LSTM has three of these gates to protect

and control the cell state.

Step-by-step LSTM walkthrough

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the **forget gate layer**. It looks at h_{t-1} and x_t and outputs a number between zero and one for each number in the cell state C_{t-1} . A one represents completely keep this while a zero represents completely get rid of this.

Let's go back to our example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject:

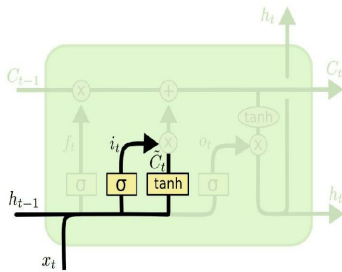


$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Figure 10.41: Calculation of a neuron's output

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the **input gate layer** decides which values we'll update. Next, a *tanh* layer creates a vector of new candidate values, $C_{\sim t}$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

In the example of our language model, we'd want to add the gender of the new subject to the cell state to replace the old one we're forgetting:



$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Figure 10.42: Decision on which value to update

It's now time to update the old cell state, C_{t-1} , into the new cell state C_t . The previous steps already decided what to do; we just need to actually do it.

We multiply the old state by f_t , forgetting the things we decided to forget earlier. Then, we add it $\ast \tilde{C}_t$. This represents the new candidate values, scaled by how much we decided to update each state value.

In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information as we decided in the previous steps.

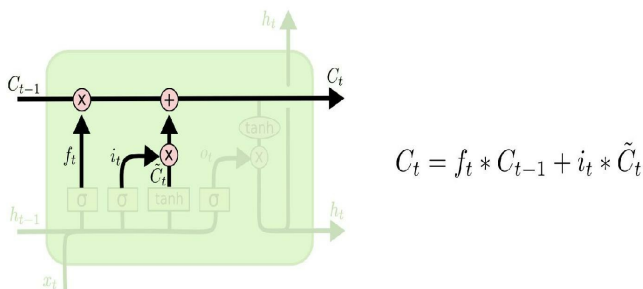


Figure 10.43: Update older values using new information

Finally, we need to decide what we're going to output. This output will be based on our cell state but will be a filtered version. First, we run a sigmoid layer that decides what parts of the cell state we're going to output. Then, we put the cell state through \tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate so that we only output the parts we decided to.

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, just in case that's what is coming next. For example, it might output whether the subject is singular

or plural so that we know what form a verb should be conjugated into if that's what follows next:

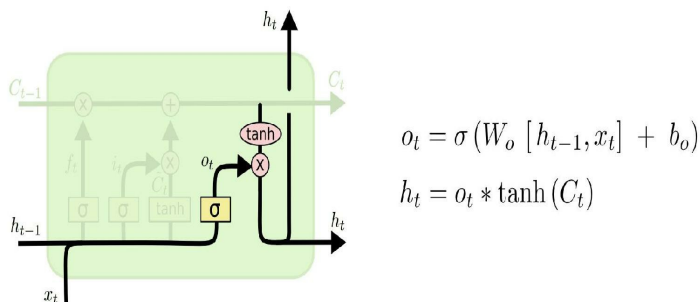


Figure 10.44: Decision of the output value

So this is how an LSTM network works. In the next section, we will implement an LSTM network using Keras and TensorFlow in Python.

Text generation using LSTM

RNNs can also be used as generative models. This means that in addition to being used for predictive models (making predictions), they can learn the sequences of a problem and then generate entirely new plausible sequences for the problem domain.

Generative models like this are useful not only to study how well a model has learned a problem, but also to learn more about the problem domain itself.

In this section, we will discover how to create a generative model for text, character by character and using LSTM RNNs in Python with Keras.

Problem description – project Gutenberg

Many classical texts are no longer protected under copyright.

This means that you can download all of the text for these books for free and use them in experiments, such as creating generative models. Perhaps the best place to get access to free books that are no longer protected by copyright is <https://www.gutenberg.org/>.

In this tutorial, we are going to use everyone's favorite book from childhood as the dataset *Alice's Adventures in Wonderland* by Lewis Carroll.

We are going to learn the dependencies between characters and the conditional

probabilities of the characters in sequences so that we can, in turn, generate wholly new and original sequences of characters.

This is a lot of fun and I recommend repeating these experiments with other books from project Gutenberg; https://www.gutenberg.org/ebooks/search/%3Fsort_order%3Ddownloads.

These experiments are not limited to text; you can also experiment with other ASCII data, such as computer source code, marked up documents in LaTeX, HTML or Markdown, and more.

You can <http://www.gutenberg.org/cache/epub/11/pg11.txt> (Plaintext UTF-8) for this book for free and place it in your working directory with the filename `wonderland.txt`.

Now we need to prepare the dataset for modeling.

Project Gutenberg adds a standard header and footer to each book and this is not part of the

original text. Open the file in a text editor and delete the header and footer.

The header is obvious and ends with the text:

```
| *** START OF THIS PROJECT GUTENBERG EBOOK  
| ALICE'S ADVENTURES IN WONDERLAND ***
```

The footer is all of the text after the line of text that says:

```
| THE END
```

You should be left with a text file that has about 3,330 lines of text.

LSTM model

In this section, we will develop a simple LSTM network to learn sequences of characters from Alice in Wonderland. In the next section, we will use this model to generate new sequences of characters:

```
#Let's start off by importing the classes and
functions we intend to use to train our model.
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils
```

Next, we need to load the ASCII text for the book into memory and convert all the characters to lowercase to reduce the vocabulary that the network must learn:

```
# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()
```

Now that the book is loaded, we must prepare the data for modeling by the neural network. We cannot model the characters directly; instead we must convert the characters into integers.

We can do this easily by first creating a set of all the distinct characters in the book and then creating a map of each character to a unique integer:

```
# create mapping of unique chars to integers
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in
    enumerate(chars))
```

For example, the list of unique sorted lowercase characters in the book is as follows:

```
[ '\n', '\r', ' ', '!', '"', "'", '(', ')', '*',
  ',', '-', '.', ':', ';', '?', '[', ']', '_',
  'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
  'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',
  's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '\xbb',
  '\xbf', '\xef']
```

You can see that there may be some characters that we could remove to further clean up the dataset and which will reduce the

vocabulary, possibly improving the modeling process.

Now that the book has been loaded and the mapping prepared, we can summarize the dataset:

```
n_chars = len(raw_text)
n_vocab = len(chars)

print ("Total Characters: ", n_chars)
print ("Total Vocab: ", n_vocab)
```

Running the code at this point produces the following output:

```
Total Characters: 147674
Total Vocab: 47
```

We can see that the book has just under *150,000* characters, and when converted to lowercase, there are only *47* distinct characters in the vocabulary for the network to learn. Much more than the 26 in the alphabet!

We now need to define the training data for the network. There is a lot of flexibility in how you choose to break up the text and

expose it to the network during training.

In this tutorial, we will split the book text into subsequences with a fixed length of 100 characters, an arbitrary length. We can just as easily split the data by sentences, pad the shorter sequences, and truncate the longer ones.

Each training pattern of the network is comprised of 100 time steps of one character input (X) followed by one character output (y). When creating these sequences, we slide this window along the whole book one character at a time, allowing each character a chance to be learned from the 100 characters that preceded it (except the first 100 characters, of course).

For example, if the sequence length is 5 (for simplicity), then the first two training patterns would be as follows:

	CHAPT	->	E
	HAPTE	->	R

As we split the book into these sequences, we convert the characters to integers using our lookup table prepared earlier:

```
# prepare the dataset of input to output pairs
encoded as integers
seq_length = 100
dataX = []
dataY = []

for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char
in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print ("Total Patterns: ", n_patterns)
```

Running the code at this point shows us that when we split up the dataset into training data for the network to learn, we have just under *150,000* training patterns. This makes sense as, excluding the first 100 characters, we have one training pattern to predict each of the remaining characters:

```
| Total Patterns: 147574
```

Now that we have prepared our training data, we need to transform it so that it is suitable for use with Keras.

First, we must transform the list of input sequences into the form [*samples, time steps, features*] expected by an LSTM network.

Next, we need to rescale the integers to the range of zero to one to make the patterns easier to learn by the LSTM network; it uses the sigmoid activation function by default.

Finally, we need to convert the output patterns (single characters converted to integers) into a one-hot encoding. This is done so that we can configure the network to predict the probability of each of the 47 different characters in the vocabulary (an easier representation) rather than trying to force it to predict precisely the next character. Each y value is converted into a sparse vector with a length of 47, full of zeros except with a one in the column for the letter (integer) that the pattern represents.

For example, when n (integer value 31) is one-hot encoded, it looks as follows:

```
[0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
 0.  0.  0.  0.  0.  0.0  0.  0.  0.  0.  0.
 0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.
 0.  0.  0.  0.  0.  0.  0.  0.  0.]
```

We can implement these steps as follows:

```
# reshape X to be [samples, time steps,
features]
X = numpy.reshape(dataX, (n_patterns,
seq_length, 1))

# normalize
X = X / float(n_vocab)

# one hot encode the output variable
y = np_utils.to_categorical(dataY)
```

We can now define our `LSTM` model. Here, we define two hidden `LSTM` layers with 256 memory units. The network uses `Dropout` with a probability of 20 with both the layers. The output layer is a `Dense` layer using the `softmax` activation function to output a probability prediction for each of the 47 characters between zero and one.

The problem is really a single-character classification problem with 47 classes, and as such, it is defined as optimizing the `loss` `log` (cross entropy), here using the `adam`

optimization algorithm for speed:

```
# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1],
X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1],
activation='softmax'))
model.compile(loss='categorical_crossentropy',
optimizer='adam')
```

There is no test dataset. We are modeling the entire training dataset to learn the probability of each character in a sequence.

We are not interested in the most accurate (classification accuracy) model of the training dataset. This would be a model that predicts each character in the training dataset perfectly. Instead, we are interested in a generalization of the dataset that minimizes the chosen loss function. We are seeking a balance between generalization and overfitting but short of memorization:

```
# define the checkpoint
filepath="weights-improvement-{epoch:02d}-
{loss:.4f}.hdf5"
checkpoint = ModelCheckpoint(filepath,
```

```
| monitor='loss', verbose=1,  
| save_best_only=True, mode='min')  
| callbacks_list = [checkpoint]
```

We can now fit our model to the data. Here, we use a modest number of 50 epochs and a large batch size of 64 patterns:

```
| model.fit(X, y, epochs=50, batch_size=6,  
| callbacks=callbacks_list)
```

You will see different results because of the stochastic nature of the model, and because it is hard to fix the random seed for LSTM models to get 100% reproducible results. This is not a concern for this generative model.

After running the example, you should have a number of weight checkpoint files in the local directory.

Generating text with an LSTM Network

Generating text using the trained LSTM network is relatively straightforward.

Firstly, we load the data and define the network in exactly the same way, except the network weights are loaded from a checkpoint file and the network does not need to be trained:

```
# load the network weights
filename = "weights-improvement-19-1.9435.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy',
optimizer='adam')
```

Also, when preparing the mapping of unique characters to integers, we must create a reverse mapping that we can use to convert the integers back to characters so that we can understand the predictions:

```
| int_to_char = dict((i, c) for i, c in
```

```
| enumerate(chars))
```

Finally, we need to actually make predictions.

The simplest way to use the Keras LSTM model to make predictions is to first start off with a seed sequence as input, generate the next character, then update the seed sequence to add the generated character on the end, and trim off the first character. This process is repeated as long as we want to predict new characters (for example, a sequence of *1,000* characters in length).

We can pick a random input pattern as our seed sequence and then print generated characters as we generate them:

```
# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]
print( "Seed:")
print ("\"", ''.join([int_to_char[value] for
value in pattern]), "\"")

# generate characters
for i in range(1000):
    x = numpy.reshape(pattern, (1,
len(pattern), 1))
    x = x / float(n_vocab)
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
```

```

        result = int_to_char[index]
        seq_in = [int_to_char[value] for value in
pattern]
        sys.stdout.write(result)
        pattern.append(index)
        pattern = pattern[1:len(pattern)]
    print ("\nDone.")

```

Running this example first outputs the selected random seed and then each character as it is generated.

For example, these are the results from one run of this text generator. The random seed was:

```

d herself lying on the bank, with her
head in the lap of her sister, who was gently
brushing away s

```

The generated text with the random seed (cleaned up for presentation) was:

```

herself lying on the bank, with her
head in the lap of her sister, who was gently
brushing away
so siee, and she sabbit said to herself and the
sabbit said to herself and the sood
way of the was a little that she was a little
lad good to the garden,
and the sood of the mock turtle said to
herself, 'it was a little that
the mock turtle said to see it said to sea it
said to sea it say it
the marge hard sat hn a little that she was so

```


sereated to herself, and
she sabbit said to herself, 'it was a little
little shated of the sooe
of the coomouse it was a little lad good to the
little gooder head. and
said to herself, 'it was a little little shated
of the mouse of the
good of the courte, and it was a little little
shated in a little that
the was a little little shated of the thmee
said to see it was a little
book of the was a little that she was so
sereated to hare a little the
began sitee of the was of the was a little that
she was so seally and
the sabbit was a little lad good to the little
gooder head of the gadseared to see it was a
little lad good to the little good

We can see that generally there are fewer spelling mistakes and the text looks more realistic, but it is still quite nonsensical.

For example, the same phrases get repeated again and again like *said to herself* and *little*. Quotes are opened but not closed.

These are better results, but there is still a lot of room for improvement.

Summary

So, we have learned a lot of modern day machine learning stuff in this chapter. We started with a simple definition of ANNs. We saw how we can adapt the single-perceptron-based model for creating a feed-forward neural network, the theory behind weight updates through backpropagation, and SGD algorithm. Then, we implanted a network for digit recognition from the MNIST dataset.

After successful implementation of ANN, we stepped towards a more powerful form of neural networks, which show very impressive performance, mainly on visual recognition tasks. We saw how CNNs work and why they have an edge over normal NNs. Afterwards, we implemented a CNN for the same digit recognition problem we had attempted with ANN. We noticed a tremendous performance improvement in the case of CNNs.

Then, we turned our learning train towards networks used in the natural language processing domain. We covered the motivation behind RNN and how forward and backward pass works in RNN. Then, we saw a very popular and successful type of RNN: LSTM networks. These networks are widely used in text generation tasks. We saw a detailed description of how these networks work. After completing the theoretical part, we implemented a small RNN for a text generation task, and we learned how LSTMs can be used for text generation. Although the performance of our network was not up to the mark, if we can go for more deeper LSTMs, maybe we can get a better performance.

In the end, as always, keep trying to scroll through online resources regarding the technology. Try to implement your algorithms for practical datasets that are easily available on the Internet, because practice is always the key to success.

All the best!!

Troubleshooting

Ensembling is a technique of combining two or more similar or dissimilar machine learning algorithms to create a model that delivers superior prediction power. This book will show you how you can use many weak algorithms to make a strong, predictive model. It contains Python code for different machine learning algorithms so that you can easily understand and implement it in your own systems.

In the appendix section, the author prefers to mention detailed code of a few chapters so that you don't struggle while implementing the code.

The following is the code for [Chapter 2](#), *Decision Trees*.

Full code of the implemented algorithm ID3

```
import numpy as np
import pandas as pd

#Function to get Information Gain of the
attribute using class entropy
def getInformationGain(subtable,classEntropy):

    #Initialize a variable for storing
probability of Classes
    fraction = 0
    #Calculate total number of instances
    denom = np.sum(np.sum(subtable))

    #Initialize variable for storing total
entropies of attribute values
    EntropyAtt = 0

    #Now we will run a loop to access each
attribute and its information gain
    for key in subtable.keys():

        #Extract Attribute
        attribute = subtable[key]
        entropy = 0 #Initialize variable for
entropy calculation
        coeff = 0 #Initialize variable to
store coefficient
```

```

        #Find out sum of class attributes(in
our case Yes and No)
        denom2 = np.sum(attribute)

        #Run a loop to get entropy of distinct
values of attribute
        for value in attribute:

            #Calculate coeff
            coeff+= float(value)/denom

            #Calculate probability of the
attribute value
            fraction = float(value)/denom2

            #Calculate Entropy
            eps = np.finfo(float).eps
            entropy+= -
fraction*np.log2(fraction+eps)
            EntropyAtt+= coeff*entropy

        #Calculate Information Gain using class
entropy
        InfGain = classEntropy - EntropyAtt
        return InfGain,EntropyAtt

```

```

#Function to get class entropy
def getClassEntropy(classAttributes):

    #Get distinct classes and how many time
they occur
    _,counts =
np.unique(classAttributes,return_counts=True)
    denom = len(classAttributes)
    entropy = 0 #Initialize entropy variable

    #Run a loop to calculate entropy of dataset
    for count in counts:
        fraction = float(count)/denom
        entropy+= -fraction*np.log2(fraction)
    return entropy

```

```

#Function to get class occurence table
def getHistTable(df,attribute):
    #This function create a subtable for the
given attribute
    #Get values for the attribute
    value = df[attribute]

    #Extract class
    classes = df['Class']

    #Get distinct classes
    classunique = df['Class'].unique()

    #Get distinct values from attribute for
example, Low, High and Med for Salary
    valunique = df[attribute].unique()

    #Create an empty table to store attribute
value and their respective class      occurance
    temp =
    np.zeros((len(classunique),len(valunique)),dtype

    subtable =
    pd.DataFrame(temp,index=classunique,columns=valu

    #Calculate class occurance for each value
for Med salary how many time class
attribute is Yes
    for i in range(len(classes)):
        subtable[value[i]][classes[i]]+= 1

    return subtable

```

```

#Function to get new node
def getNode(df):
    #This function is written for getting
winner attribute to assign node

    #Get Classes
    classAttributes = df['Class']

    #Create empty list to store Information
gain for respected attributes

```

```

    InformationGain = []
    AttributeName = []

    #Extract each attribute
    for attribute in df.keys():
        if attribute is not 'Class':
            #Get class occurance for each
attribute value
            subtable =
getHistTable(df,attribute)

            #Get class entropy of the data
            Ec =
getClassEntropy(classAttributes)

            #Calculate Information Gain for
each attribute
            InfoGain,EntropyAtt =
getInformationGain(subtable, Ec)

            #Append the value into the list
            InformationGain.append(InfoGain)
            AttributeName.append(attribute)
            print("Information Gain for %s:
%.2f and Entropy: %.2f"%
                (attribute,InfoGain,EntropyAtt))

    #Find out attribute with maximum
information gain
    indx = np.argmax(InformationGain)
    winnerNode = AttributeName[indx]
    print("\nWinner attribute is: %s"%
(winnerNode))

    return winnerNode

#Function to get sub table for the attribute
def getSubtable(df,node,atValues):
    #This function is written to get subtable
for given attribute values(such as table
for those persons whose salary is Medium)
    subtable = []
    #run a loop through the dataset and create

```



```

subtable
    for i in range(len(df[node])):
        if df[node][i]==atValues:
            row = df.loc[i,df.keys()]
            subtable.append(row)

    for c in range(len(df.keys())):
        if df.keys()[c]==node:
            break

    #Create a new dataframe
    subtable =
pd.DataFrame(subtable,index=range(len(subtable))
    print(subtable)
    return subtable

```

```

#Function to build the tree by adding new nodes
def buildTree(df,tree=None):
    #Here we build our decision tree
    #Get attribute with maximum information
gain
    node = getNode(df)

    #Get distinct value of that attribute e.g
Salary is node and Low,Med and
    High are values
    attValue = np.unique(df[node])

    #Create an empty dictionary to create tree
    if tree is None:
        tree={}
        tree[node] = {}

    #Loop below is written for building tree
using recursion of the function,
    #We will create subtable of each attribute
value and try to find whether it
    have a pure subset or not,
    #if it is a pure subset we will stop tree
growing for that node. if it is
    not a pure set then we will..
    #again call the same function.
    for value in attValue:

```

```

        print("Value: %s"%value)
        subtable = getSubtable(df,node,value)
        clValue,counts =
np.unique(subtable['Class'],return_counts=True)

        if len(counts)==1:#Checking purity of
subset
            print("Class: %s\n"%clValue)
            tree[node][value] = clValue
        else:
            tree[node][value] =
buildTree(subtable)#Recursion of the function
        return tree

```

```

#Function to get prediction out of input tree
def predict(inst,tree):
    #This function will predict an input
instance's class using given tree

```

```

    #We will use recursion to traverse through
the tree same as we have done in      case of
tree building
    for nodes in tree.keys():

        value = inst[nodes]
        for val in value:
            tree = tree[nodes][val]
            prediction = 0

            if type(tree) is dict:
                prediction = predict(inst,
tree)#Recursion
            else:
                prediction = tree
                break;
    return prediction

```

```

#Function to pre-process the dataset to get
data frame and set indexes
def preProcess(dataset):
    #Create a dataframe out of our dataset with
attribute names
    df = pd.DataFrame(dataset,columns=

```

```
['Name', 'Salary', 'Sex', 'Marital', 'Class'])

    #Remove name attribute as it is not
    required for the calculations
    df.pop('Name')

    #Make sure last attribute of our dataset
    must be Class attribute
    cols = list(df)
    cols.insert(len(cols),
    cols.pop(cols.index('Class')))
    df = df.ix[:,cols]
    print(df)

    return df
```

Code of the CART algorithm

```
import pprint
import sys
from csv import reader
import numpy as np

#Function to read csv file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

#Function to create Train and Test set from the original dataset
def getTrainTestData(dataset, split):

    training = []
    testing = []

    shape = np.shape(dataset)
    trainlength =
np.uint16(np.floor(split*shape[0]))

    for i in range(trainlength):
        training.append(dataset[i])

    for i in range(trainlength, shape[0]):
        testing.append(dataset[i])
```

```

        return training, testing

#Create splits to validate gini score
def createSplit(attribute, threshold, dataset):

    #Initialize two lists to store the sub
sets
    lesser, greater = list(), list()

    #Loop through the attribute values and
create sub set out of it
    for values in dataset:
        #Apply threshold
        if values[attribute] <= threshold:
            lesser.append(values)
        else:
            greater.append(values)
    return lesser, greater

# Calculate the Gini index for a split dataset
def gini_index(groups, class_values):
    gini = 0.0
    for class_value in class_values:
        for group in groups:
            size = len(group)
            if size == 0:
                continue
            proportion = [row[-1] for row in
group].count(class_value) /
float(size)
            gini += (proportion * (1.0 -
proportion))
    return gini

#Function to get new node
def getNode(dataset):

    class_values = []
    for row in dataset:
        class_values.append(row[-1])

    #Extract unique class values present in

```

```

the dataset
    class_values =
np.unique(np.array(class_values))

    #initialize variables to store gini score,
attribute index and split groups
    winnerAttribute = sys.maxsize
    attributeValue = sys.maxsize
    gScore = sys.maxsize
    leftGroup = None

    #Run loop to access each attribute and
attribute values
    for index in range(len(dataset[0])-1):
        for row in dataset:
            groups = createSplit(index,
row[index], dataset)
            gini = gini_index(groups,
class_values)
            if gini < gScore:
                winnerAttribute,
attributeValue, gScore, leftGroup = index,
row[index], gini, groups
            #Once done create a dictionary for node
            node =

{'attribute':winnerAttribute,'value':attributeVa

    return node

# Create a terminal node value
def terminalNode(group):
    outcomes = [row[-1] for row in group]
    return max(set(outcomes),
key=outcomes.count)

# Create child splits for a node or make
terminal
def buildTree(node, max_depth, min_size,
depth):

    #Let's get groups information first.

```

```

    left, right = node['groups']
    del(node['groups'])

    # check if there are any element in the
left and right group
    if not left or not right:

        #If there is no element in the groups
call terminal Node
        combined = left+right
        node['left'] = terminalNode(combined)
        node['right']= terminalNode(combined)
        return

    # check if we have reached to maximum
depth
    if depth >= max_depth:
        node['left']=terminalNode(left)
        node['right'] = terminalNode(right)
        return

    # if all okey lest start building tree for
left side nodes
    # if minimum instances are done by the
node stop further build
    if len(left) <= min_size:
        node['left'] = terminalNode(left)
    else:
        #Create new node under left side of
the tree
        node['left'] = getNode(left)

        #append node under the tree and
increase depth by one.
        buildTree(node['left'], max_depth,
min_size, depth+1) #recursion will
take place in here

    # Similar procedure for the right side
nodes
    if len(right) <= min_size:
        node['right'] = terminalNode(right)

```

```

        else:
            node['right'] = getNode(right)
            buildTree(node['right'], max_depth,
min_size, depth+1)

# Build a decision tree
def build_tree(train, max_depth, min_size):
    root = getNode(train)
    buildTree(root, max_depth, min_size, 1)
    return root

# Print a decision tree
def print_tree(node, depth=0):
    if isinstance(node, dict):
        print('%s[X%d < %.2f]' % ((depth*' ',
(node['attribute']+1),
            node['value'])))
        print_tree(node['left'], depth+1)
        print_tree(node['right'], depth+1)
    else:
        print('%s[%s]' % ((depth*' ', node)))

#Function to get prediction from input tree
def predict(node, row):
    #Get the node value and check whether the
    attribute value is less than or
    equal.
    if row[node['attribute']] <=
node['value']:
        #If yes enter into left branch and
        check whether it has another node or
        the class value.
        if isinstance(node['left'], dict):
            return predict(node['left'],
row)#Recursion
        else:
            #If there is no node in the branch
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

```



```

#Function to check accuracy of the dataset
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) *
100.0

#Function to convert string attribute values to
float
def str_column_to_float(dataset, column):
    for row in dataset:
        if row[column]=='?':
            row[column] = 0
        else:
            row[column] =
float(row[column].strip())

```

The following is the code for [Chapter 3](#),
Random Forest.

Code for random forest

This code includes only those functions that are different from CART:

```
# Build a decision tree
def build_tree_RF(train, max_depth,
min_size,nfeatures):
    root = getNodeRF(train,nfeatures)
    buildTreeRF(root, max_depth, min_size,
1,nfeatures)
    return root

# Create child splits for a node or make
terminal
def buildTreeRF(node, max_depth, min_size,
depth,nfeatures):

    #Let's get groups information first.
    left, right = node['groups']
    del(node['groups'])

    # check if there are any element in the
left and right group
    if not left or not right:

        #If there is no element in the groups
call terminal Node
        combined = left+right
        node['left'] = terminalNode(combined)
        node['right']= terminalNode(combined)
        return
```

```

    # check if we have reached to maximum
depth
    if depth >= max_depth:
        node['left']=terminalNode(left)
        node['right'] = terminalNode(right)
        return

    # if all okey lest start building tree for
left side nodes
    # if minimum instances are done by the
node stop further build
    if len(left) <= min_size:
        node['left'] = terminalNode(left)
    else:
        #Create new node under left side of
the tree
        node['left'] =
getNodeRF(left,nfeatures)

        #append node under the tree and
increase depth by one.
        buildTree(node['left'], max_depth,
min_size, depth+1) #recursion will
take place in here

    # Similar procedure for the right side
nodes
    if len(right) <= min_size:
        node['right'] = terminalNode(right)
    else:
        node['right'] =
getNodeRF(right,nfeatures)
        buildTree(node['right'], max_depth,
min_size, depth+1)

#Function to get new node
def getNodeRF(dataset):

    class_values = []
    for row in dataset:
        class_values.append(row[-1])

```

```

        #Extract unique class values present in
the dataset
        class_values =
np.unique(np.array(class_values))

        #initialize variables to store gini score,
attribute index and split groups
        winnerAttribute = sys.maxsize
        attributeValue = sys.maxsize
        gScore = sys.maxsize
        leftGroup = None

        #Select Random features
        features = list()
        while len(features) < n_features:
            index = randrange(len(dataset[0])-1)
            if index not in features:
                features.append(index)

        #Run loop to access each attribute and
attribute values
        for index in index:
            for row in dataset:
                groups = createSplit(index,
row[index], dataset)
                gini = gini_index(groups,
class_values)
                if gini < gScore:
                    winnerAttribute,
attributeValue, gScore, leftGroup = index,
row[index], gini, groups
            #Once done create a dictionary for node
            node =

{'attribute':winnerAttribute,'value':attributeVa

        return node

# Create a random subsample from the dataset
with replacement
def subsample(dataset, ratio):

```

```

sample = list()
n_sample = round(len(dataset) * ratio)
while len(sample) < n_sample:
    index = randrange(len(dataset))
    sample.append(dataset[index])
return sample

```

```

# Make a prediction with a list of bagged trees
def bagging_predict(trees, row):
    predictions = [predict(tree, row) for tree
in trees]
    return max(set(predictions),
key=predictions.count)

```

```

# Random Forest Algorithm
def random_forest(train, test, max_depth,
min_size, sample_size, n_trees,
n_features):
    trees = list()
    for i in range(n_trees):
        sample = subsample(train, sample_size)
        tree = build_tree_RF(sample,
max_depth, min_size, n_features)
        trees.append(tree)
    predictions = [bagging_predict(trees, row)
for row in test]
    return(predictions)

```

```

#Create cross validation sets
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index =
randrange(len(dataset_copy))
        fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split

```

```

# Evaluate an algorithm using a cross
validation split
def evaluate_algorithm(dataset, algorithm,
n_folds, *args):
    folds = cross_validation_split(dataset,
n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set,
test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual,
predicted)
        scores.append(accuracy)
    return scores

```

Up next is the code for [Chapter 4, Random Subspace and KNN Bagging](#).

Code for KNN and subspace bagging

```
#Import math for calculations of square roots
import math
import operator
from random import randrange

#Function to get distance between test instance
and training set
def DistanceMetric(instance1, instance2,
isClass=None):

    #If Class variable is in the instance
    if isClass:
        length = len(instance1)-1
    else:
        length = len(instance1)

    #Initialize variable to store distance
    distance = 0

    #Lets run a loop to calculate element wise
differences
    for x in range(length):

        #Euclidean distance
        distance += pow((instance1[x] -
instance2[x]), 2)

    return math.sqrt(distance)

#Function to get nearest neighbors
def getNeighbors(trainingSet, testInstance, k):
```

```

        #Create a list variable to store distances
        between test and training
        instance.
        distances = []

        #Get distance between each instance in the
        training set and the test
        instance.
        for x in range(len(trainingSet)):

            #As we will have class variable in the
            training set isClass
            will be true
            dist = DistanceMetric(testInstance,
            trainingSet[x], isClass=True)

            #Append the distance of each instance
            to the distance list
            distances.append((trainingSet[x],
            dist))

            #Sort the distances in ascending order
            distances.sort(key=operator.itemgetter(1))

            #Create a list to store the neighbors
            neighbors = []

            #Run a loop to get k neighbors from the
            sorted distances.
            for x in range(k):
                neighbors.append(distances[x][0])
            return neighbors

#Function to get prediction
def getPrediction(neighbors):

    #Create a dictionary variable to store
    votes from the neighbors
    #We will use class attribute as the
    dictionary keys and their occurrence as
    key value.
    classVotes = {}

```



```

    #Go to each neighbor and take the vote for
the class
    for x in range(len(neighbors)):

        #Get the class value of the neighbor
        response = neighbors[x][-1]

        #Create class key if its not there;
        #If class key is in the dictionary
increase it by one.
        if response in classVotes:
            classVotes[response] += 1
        else:
            classVotes[response] = 1
    #Sort the dictionary keys on the basis of
key values in descending order
    sortedVotes =
sorted(classVotes.iteritems(),
key=operator.itemgetter(1),
            reverse=True)

    #Return the key name (class) with the
highest value
    return sortedVotes[0][0]

```

KNN subspace bagging code

```
def DistanceMetricBagged(instance1,
instance2,n_features):

    #Initialize variable to store distance
    distance = 0
    features = list()

    #Select random features to apply subspace
    bagging
    while len(features) < n_features:
        index = randrange(len(instance1)-1)
        if index not in features:
            features.append(index)

    #Lets run a loop to calculate element wise
    differences for the selected
    features only.
    for x in features:
        #Euclidean distance
        distance += pow((instance1[x] -
instance2[x]), 2)

    return math.sqrt(distance)

def getNeighborsBagged(trainingSet,
testInstance, k,n_features):

    #Create a list variable to store distances
    between test and training
    instance.
    distances = []
```

```

        #Get distance between each instance in the
training set and the test
instance.
        for x in range(len(trainingSet)):
            #As we will have class variable in the
training set isClass
            will be true
            dist =
DistanceMetricBagged(testInstance,
trainingSet[x],n_features)

            #Append the distance of each instance
to the distance list
            distances.append((trainingSet[x],
dist))

        #Sort the distances in ascending order
distances.sort(key=operator.itemgetter(1))

        #Create a list to store the neighbors
neighbors = []

        #Run a loop to get k neighbors from the
sorted distances.
        for x in range(k):
            neighbors.append(distances[x][0])
        return neighbors

```

The following is the code for [Chapter 5](#),
AdaBoost Classifier.

Code of the AdaBoost classifier

```
import sys
import numpy as np;

from matplotlib import pyplot as plt

#Get Gini Index
def gini_index(groups, class_values):

    #Initialize Gini variable
    gini = 0.0

    #Calculate proportion for each class
    for class_value in class_values:

        #Extract groups
        for group in groups:
            #Number of instance in the group
            size = len(group)
            if size == 0:
                continue

            #Initialize a list to store class
            index of the instances
            r = []
            cl = []

            #get class of each instance in the
            group
            for row in group:
                r.append(row[-1])#Weight
                Append
                cl.append(row[-2])#Class
```

Append

```
        r = np.array(r)
        #Extract Class indexes of the
current class value
        class_index =
np.where(cl==class_value)

        #Initialize a variable to add the
weights of current class
        w_add=0

        #Add the weights of the current
class using class indexes
        for w in class_index[0]:
            w_add+= r[w];

        #Calculate class proportion using
weights
        proportion = w_add/np.sum(r)

        #Calculate Gini index
        gini += (proportion * (1.0 -
proportion))
    return gini
```

```
#function to create split for getting node
value
def createSplit(attribute,threshold,dataset):

    #Initialize two lists to store the sub sets
    lesser, greater = list(),list()

    #Loop through the attribute values and
create sub set out of it
    for values in dataset:
        #Apply threshold
        if values[attribute]<=threshold:
            lesser.append(values)
        else:
            greater.append(values)
    return lesser,greater
```

```

#Function to Node for decision stump
def getNode(dataset):
    class_values = []
    #Extract unique class values present in
the data set
    for row in dataset:
        class_values.append(row[-2])#Class
values are in the second last column
    class_values = np.unique(class_values)

    #initialize variables to store gini score,
attribute index and split groups
    winnerAttribute = sys.maxsize
    attributeValue = sys.maxsize
    gScore = sys.maxsize
    leftGroup = None

    #Run loop to access each attribute and
attribute values
    for index in
range(len(dataset[0])-2):#leave last two
columns
        for row in dataset:
            #Create the groups
            groups = createSplit(index,
row[index], dataset)
            #Extract gini score for the
threshold
            gini = gini_index(groups,
class_values)

            #If gini score is lower than the
previous one choose return it
            if gini < gScore:
                winnerAttribute,
attributeValue, gScore, leftGroup = index,
row[index], gini, groups
            #Once done create a dictionary for node
            node =
{'attribute':winnerAttribute, 'value':attributeVa
return node

```

```

def terminalNode(group):
    outcomes = [row[-2] for row in group]
    return max(set(outcomes),
key=outcomes.count)

```

#Function to create decision stump

```

def decision_stump(dataset):

    #Get node value with best gini score
    node = getNode(dataset)

    #Separate out the groups from the node and
remove them
    left, right = node['groups']
    del(node['groups'])

    #Check whether there is any element in the
groups or not
    #If there is not any element put the class
value with maximum occurrence
    if not left or not right:
        node['left'] = node['right'] =
terminalNode(left + right)
    return node

    #Put left group's maximum occur class
value in left branch
    node['left']=terminalNode(left)

    #Put right group's maximum occur class
value in right branch
    node['right'] = terminalNode(right)
    return node

```

#Function for get predict

```

def predict(node, row):
    #Get the node value and check whether the
attribute value is less than or
equal.
    if row[node['attribute']] <=
node['value']:
        #If yes enter into left branch and

```

```

    check whether it has another node or
    the class value.
    return node['left']
else:
    return node['right']

```

```

#Function to calculate error
def getError(actual,predicted,weights):
    #Initialize the error variable
    error = 0

    #We will store the error of each instance
in a vector
    error_vec=[]

    #Run a loop to calculate error for each
instance
    for i in range(len(actual)):
        diff = predicted[i]!=actual[i]
        #Weights multiplication to the
difference of actual and predicted values
        error+= weights[i]*(diff)

        #Append the difference to the error
vector
        error_vec.append(diff)

    return error,error_vec

```

```

#Function of adaboost algorithm for updating
weights
def AdaBoostAlgorithm(dataset,iterations):

    #Initialize the weights of the size of
data set
    weights =
    np.ones(len(dataset),dtype="float32")/len(dataset)

    dataset = np.array(dataset)

    #Add Weights column to the data set(Now
last column will be the weights)
    dataset = np.c_[dataset,weights]

```



```

        #Create an empty list to store alpha
values
        alphas = []

        #Create a list to add weak
learners(decision stumps)
        weaks = []

        er = sys.maxsize
        #Lets run the loop for number of
iteration(number of classifiers)
        for itr in range(iterations):

            #Create decision tree from the non
weighted data-set
            ds = decision_stump(dataset)

            #Create a list to store the
predictions of the decision stump
            pred=[]

            #Create a list to store actual outputs
            actual = []

            #Let's predict output for each
instance in the data set
            for row in dataset:
                actual.append(row[-2])
                pred.append(predict(ds, row))

            #Here we will find out difference
between predicted and actual output
            error,error_vec = getError(actual,
pred,weights)

            #If error is greater than 0.5
classifier is not able to classify the
dataset
            if error>=0.0:
                break
            eps = sys.float_info.epsilon

```

```

        #Let's find out the alpha with the
help of error
        alpha = (0.5 * np.log((1-
error)/(error+eps)))

        #Create empty vector to store weight
updates
        w = np.zeros(len(weights))

        # Update the weights using alpha value
        for i in range(len(error_vec)):

            #For wrong prediction increase the
weights
            if error_vec[i]!=0:
                w[i] = weights[i] *
np.exp(alpha)

            #For correct prediction decrease
the weights
            else:
                w[i] = weights[i] * np.exp(-
alpha)

        #Normalize the weights and update
previous weight vector
        weights = w / w.sum()

        #Put the updated weights into the data
set by over-writing previous
        weights
        dataset[:, -1]=weights

        print("\nClassifier %i stats:"%itr)
        print(ds)
        print("Error: %.3f and alpha: %.3f"%
(error,alpha))
        er = error

        #Append alpha value to the list to
used at the time of testing
        alphas.append(alpha)

```

```

        #Append the weak learner to the list
        weaks.append(ds)

    return weaks, alphas

#Function to evaluate accuracy
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) *
100.0

```

The following is the code for [Chapter 6](#), *Gradient Boosting Machines*.

Code of GBMs

```
import numpy as np
import sys
def terminalNodeReg(group):

    #Get all the target labels into the List
    class_values = [row[-1] for row in group]

    #Return the Mean value of the list
    return np.mean(class_values)

# Calculate the SSE index for a split dataset
def SquaredError(groups):

    #Initialize the variable for SSE
    sse = 0.0

    #Iterate for both the groups
    for group in groups:
        size = len(group)

        #If length is 0 continue for the next
        group
        if size == 0:
            continue

        #Take all the class values into a list
        class_values = [row[-1] for row in
        group]

        #Calculate SSE for the group
        sse += np.sum((class_values-
        np.mean(class_values))**2)
        return sse

#Function to get new node
def getNode(dataset):
```

```

        #initialize variables to store error score,
attribute index and split groups
        winnerAttribute = sys.maxsize
        attributeValue = sys.maxsize
        errorScore = sys.maxsize
        leftGroup = None

        #Run loop to access each attribute and
attribute values
        for index in range(len(dataset[0])-1):
            for row in dataset:

                #Get split for the attribute value
                groups = createSplit(index,
row[index], dataset)

                #Calculate SSE for the group
                sse = SquaredError(groups)

                #print("SSE for the attribute
%.2f's value %.2f is %.3f"%
(index+1,row[index],sse))

                #If SSE is less than previous
attribute's SSE return attribute value as Node
                if sse < errorScore:
                    winnerAttribute,
attributeValue, errorScore, leftGroup = index,
row[index], sse, groups

                #Once done create a dictionary for node
                node =
{'attribute':winnerAttribute,'value':attributeVa

        return node

#Create splits to test for node values
def createSplit(attribute,threshold,dataset):

    #Initialize two lists to store the sub sets
    lesser, greater = list(),list()

```

```

    #Loop through the attribute values and
    create sub set out of it
    for values in dataset:
        #Apply threshold
        if values[attribute]<=threshold:
            lesser.append(values)
        else:
            greater.append(values)
    return lesser,greater

```

```

# Create child splits for a node or make
terminal
def buildTreeReg(node, max_depth, min_size,
depth):

    #Lets get groups information first.
    left, right = node['groups']
    del(node['groups'])

    # check if there are any element in the
    left and right group
    if not left or not right:
        #If there is no element in the groups
        call terminal Node
        combined = left+right
        node['left'] =
terminalNodeReg(combined)
        node['right']=
terminalNodeReg(combined)
        return

    # check if we have reached to maximum depth
    if depth >= max_depth:
        node['left']=terminalNodeReg(left)
        node['right'] = terminalNodeReg(right)
        return

    # if all okay let's start building tree for
    left side nodes
    # if minimum instances are done by the node
    stop further build
    if len(left) <= min_size:
        node['left'] = terminalNodeReg(left)

```

```

        else:
            #Create new node under left side of the
tree
            node['left'] = getNode(left)
            #append node under the tree and
increase depth by one.
            buildTreeReg(node['left'], max_depth,
min_size, depth+1) #recursion will
take place in here

        # Similar procedure for the right side
nodes
        if len(right) <= min_size:
            node['right'] = terminalNodeReg(right)

        else:
            node['right'] = getNode(right)
            buildTreeReg(node['right'], max_depth,
min_size, depth+1)

```

```

# Build a decision tree
def build_tree(train, max_depth, min_size):

    #Add the root node to the tree
    root = getNode(train)

    #Start building the from the root's
branches tree
    buildTreeReg(root, max_depth, min_size, 1)
    return root

```

```

#Function to get prediction from input tree
def predict(node, row):

    #Get the node value and check whether the
attribute value is less than or equal.
    if row[node['attribute']] <= node['value']:

        #If yes enter into left branch and
check whether it has another node or the class
value.

```

```

        if isinstance(node['left'], dict):
            return predict(node['left'],
row)#Recursion
        else:
            #If there is no node in the branch
            return node['left']
    else:
        if isinstance(node['right'], dict):
            return predict(node['right'], row)
        else:
            return node['right']

def getResidual(actual,pred):

    #Create an empty list to store individual
error of the instances
    residual = []

    # Run a loop to get difference between
output and prediction of each instance
    for i in range(len(actual)):

        #Get the difference and add the
difference to the list of residuals
        diff = (actual[i]-pred[i])
        residual.append(diff)

    #Calculate the Sum of squared error between
output and prediction
    mse = np.sum(np.array(residual)**2)
    return residual,mse

def
GradientBoost(dataset,depth,mincount,iterations)

    dataset = np.array(dataset)

    #Create a list to add weak
learners(decision stumps)
    weaks = []

    #Lets run the loop for number of
iteration(number of classifiers)

```



```

    for itr in range(iterations):

        #Create decision tree from the data-set
        ds = build_tree(dataset,depth,mincount)

        #Create a list to store the predictions
of the decision stump
        pred=[]

        #Create a list to store actual outputs
        actual = []

        #Let's predict output for each instance
in the data set
        for row in dataset:
            actual.append(row[-1])
            pred.append(predict(ds, row))

        #Here we will find out difference
between predicted and actual output
        residuals,error = getResidual(actual,
pred)

        #Print the error status
        print("\nClassifier %i error is %.5f"%
(itr,error))

        #Check for the convergence
        if error<=0.00001:
            break

        #Replace the previous labels with the
current differences(Residuals)
        dataset[:, -1] = residuals

        #Append the weak learner to the list
        weaks.append(ds)

    return weaks

def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):

```

```
        if actual[i] == predicted[i]:  
            correct += 1  
    return correct / float(len(actual)) * 100.0
```

Up next is the code for [Chapter 8](#), *Stacked Generalization*.

Full code of implementation

```
# Test stacking on the sonar dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt
from math import exp

# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
    class_values = [row[column] for row in dataset]
    unique = set(class_values)
    lookup = dict()
    for i, value in enumerate(unique):
        lookup[value] = i
```

```
    for row in dataset:
        row[column] = lookup[row[column]]
    return lookup
```

Split a dataset into k folds

```
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)

    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index =
randrange(len(dataset_copy))

        fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split
```

Calculate accuracy percentage

```
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0
```

Evaluate an algorithm using a cross validation split

```
def evaluate_algorithm(dataset, algorithm,
n_folds, *args):
    folds = cross_validation_split(dataset,
n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
```

```

        row_copy[-1] = None
        predicted = algorithm(train_set,
test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual,
predicted)
        scores.append(accuracy)
    return scores

```

Calculate the Euclidean distance between two vectors

```

def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2
    return sqrt(distance)

```

Locate neighbors for a new row

```

def get_neighbors(train, test_row,
num_neighbors):
    distances = list()
    for train_row in train:
        dist = euclidean_distance(test_row,
train_row)
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(distances[i][0])
    return neighbors

```

Make a prediction with kNN

```

def knn_predict(model, test_row,
num_neighbors=2):
    neighbors = get_neighbors(model, test_row,
num_neighbors)
    output_values = [row[-1] for row in
neighbors]
    prediction = max(set(output_values),
key=output_values.count)
    return prediction

```

Prepare the kNN model

```

def knn_model(train):
    return train

# Make a prediction with perceptron
def perceptron_predict(weights,row):
    #Row is the input instance

    #We will consider first weight as the bias for
    simplyfied the calculations
    activation = weights[0]

    #Now run a loop to multiply each attribute
    value of the instance with the weight
    #And add the result to the activation of
    previous attribute
    for i in range(len(row)-1):
        activation += weights[i + 1] * row[i]

    #Here we will return 1 if activation is a non
    negative value and zero in other case
    return 1.0 if activation >= 0.0 else 0.0

# Estimate Perceptron weights using stochastic
gradient descent
def perceptron_model(train, l_rate=0.01,
n_epoch=5000):

    #Lets initialize the weights by 0
    weights = [0.0 for i in
range(len(train[0]))]

    #We will update the weights for given
    number of epoch
    for epoch in range(n_epoch):

        #Extract each row from the training set
        for row in train:

            #Predict the value for the instance
            prediction =
perceptron_predict(weights,row)

```

```

        #Calculate the difference(gradient)
between actual and predicted
        value
        error = row[-1] - prediction

        #Update the bias value using given
learning rate and error
        weights[0] = weights[0] + l_rate *
error

        #Update the weights for each
attribute using learning rate
        for i in range(len(row)-1):
            weights[i + 1] = weights[i + 1]
+ l_rate * error * row[i]

        #Return the updated weights and biases
        return weights

```

```

# Make a prediction with coefficients
def logistic_regression_predict(model, row):

    #First weight of the model will be bias
similar as Perceptron function
    yhat = model[0]

    #We will run a loop to multiply each
attribute value with the corresponding
weights
    #This is similar to activation calculation
in perceptron algorithm
    for i in range(len(row)-1):
        yhat += model[i + 1] * row[i]

    #Here we will apply logistic function on
the linear combination of weights
and attributes
    #This is the place where linear and
logistic regression differs
    return 1.0 / (1.0 + exp(-yhat))

# Estimate logistic regression coefficients
using stochastic gradient descent

```

```

def logistic_regression_model(train,
l_rate=0.01, n_epoch=5000):

    #Initialize the weights with the zero
values
    coef = [0.0 for i in range(len(train[0]))]

    #Repeat the procedure for given number of
epochs
    for epoch in range(n_epoch):

        #Get prediction for each row and update
weights based on error value
        for row in train:

            #Predict y for the given x
            yhat =
logistic_regression_predict(coef, row)

            #Get the error value
            (gradient/slope/change)
            error = row[-1] - yhat

            #Apply gradient descent here to
update the weights and biases
            #Update Bias first
            coef[0] = coef[0] + l_rate * error
            * yhat * (1.0 - yhat)

            #Now update the Weights

            for i in range(len(row)-1):
                coef[i + 1] = coef[i + 1] +
l_rate * error * yhat * (1.0 - yhat)
                * row[i]

            #Return the trained weights and biases
            return coef

# Make predictions with sub-models and
construct a new stacked row
def to_stacked_row(models, predict_list, row):

    #Let's Create an empty list to store

```



```

predictions from sub models
    stacked_row = list()

    #Run a loop to fetch stored models in the
List
    for i in range(len(models)):

        #Start prediction for each row by each
model
        prediction = predict_list[i](models[i],
row)

        #Store the prediction in the list
        stacked_row.append(prediction)

        #Append class values to the new row
        stacked_row.append(row[-1])

    #Extend the old row aby adding stacked row
    return row[0:len(row)-1] + stacked_row

# Stacked Generalization Algorithm
def stacking(train, test):

    #Let's define the sub model first
    model_list = [knn_model, perceptron_model]

    #We will create a prediction list to create
new row
    predict_list = [knn_predict,
perceptron_predict]

    #Create an empty list to store the trained
models
    models = list()

    #Lets train each sub model individually on
the dataset
    for i in range(len(model_list)):
        model = model_list[i](train)
        models.append(model)

    #Create a new stacked data set from

```

```

prediction of sub models
    stacked_dataset = list()
    for row in train:

        #Get new row
        stacked_row = to_stacked_row(models,
predict_list, row)

        #Append it to new dataset
        stacked_dataset.append(stacked_row)

    #We will train our final classifier on the stacked dataset
    stacked_model =
logistic_regression_model(stacked_dataset)

    #lets create a list of prediction of the stacked output
    predictions = list()

    #Here we will combine all the classifier together to make stack of classifiers
    for row in test:

        #Get new row from prediction of sub models
        stacked_row = to_stacked_row(models,
predict_list, row)

        #Append new row to the new dataset
        stacked_dataset.append(stacked_row)

        #Classify the new row using final classifier
        prediction =
logistic_regression_predict(stacked_model,
stacked_row)

        #As final classifier gives a continuous value round it to nearest integer
        prediction = round(prediction)

```

```

        #Append the prediction to the final
list of predictions
        predictions.append(prediction)
    return predictions

# Test stacking on the sonar dataset
seed(1)
# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)
# convert string attributes to integers
for i in range(len(dataset[0])-1):
    str_column_to_float(dataset, i)
# convert class column to integers
str_column_to_int(dataset, len(dataset[0])-1)
n_folds = 5
scores = evaluate_algorithm(dataset, stacking,
n_folds)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' %
(sum(scores)/float(len(scores))))

```

The following is the code for [Chapter 10](#),
Modern Day Machine Learning.

Full code of LSTM implementation

```
# Load Larger LSTM network and generate text
import sys
import numpy
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.callbacks import ModelCheckpoint
from keras.utils import np_utils

# load ascii text and covert to lowercase
filename = "wonderland.txt"
raw_text = open(filename).read()
raw_text = raw_text.lower()

# create mapping of unique chars to integers,
and a reverse mapping
chars = sorted(list(set(raw_text)))
char_to_int = dict((c, i) for i, c in
    enumerate(chars))
int_to_char = dict((i, c) for i, c in
    enumerate(chars))

# summarize the loaded data
n_chars = len(raw_text)
n_vocab = len(chars)
print ("Total Characters: ", n_chars)
print ("Total Vocab: ", n_vocab)

# prepare the dataset of input to output pairs
encoded as integers
seq_length = 100
```

```

dataX = []
dataY = []

for i in range(0, n_chars - seq_length, 1):
    seq_in = raw_text[i:i + seq_length]
    seq_out = raw_text[i + seq_length]
    dataX.append([char_to_int[char] for char
in seq_in])
    dataY.append(char_to_int[seq_out])
n_patterns = len(dataX)
print ("Total Patterns: ", n_patterns)

# reshape X to be [samples, time steps,
features]
X = numpy.reshape(dataX, (n_patterns,
seq_length, 1))

# normalize
X = X / float(n_vocab)

# one hot encode the output variable
y = np_utils.to_categorical(dataY)

# define the LSTM model
model = Sequential()
model.add(LSTM(256, input_shape=(X.shape[1],
X.shape[2]), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(y.shape[1],
activation='softmax'))

# load the network weights
filename = "weights-improvement-47-1.2219-
bigger.hdf5"
model.load_weights(filename)
model.compile(loss='categorical_crossentropy',
optimizer='adam')

# pick a random seed
start = numpy.random.randint(0, len(dataX)-1)
pattern = dataX[start]

```

```

print ("Seed:")
print ("\"", ''.join([int_to_char[value] for
value in pattern]), "\"")

# generate characters
for i in range(1000):
    x = numpy.reshape(pattern, (1,
len(pattern), 1))
    x = x / float(n_vocab)
    prediction = model.predict(x, verbose=0)
    index = numpy.argmax(prediction)
    result = int_to_char[index]
    seq_in = [int_to_char[value] for value in
pattern]
    sys.stdout.write(result)
    pattern.append(index)
    pattern = pattern[1:len(pattern)]
print ("\nDone.")

```