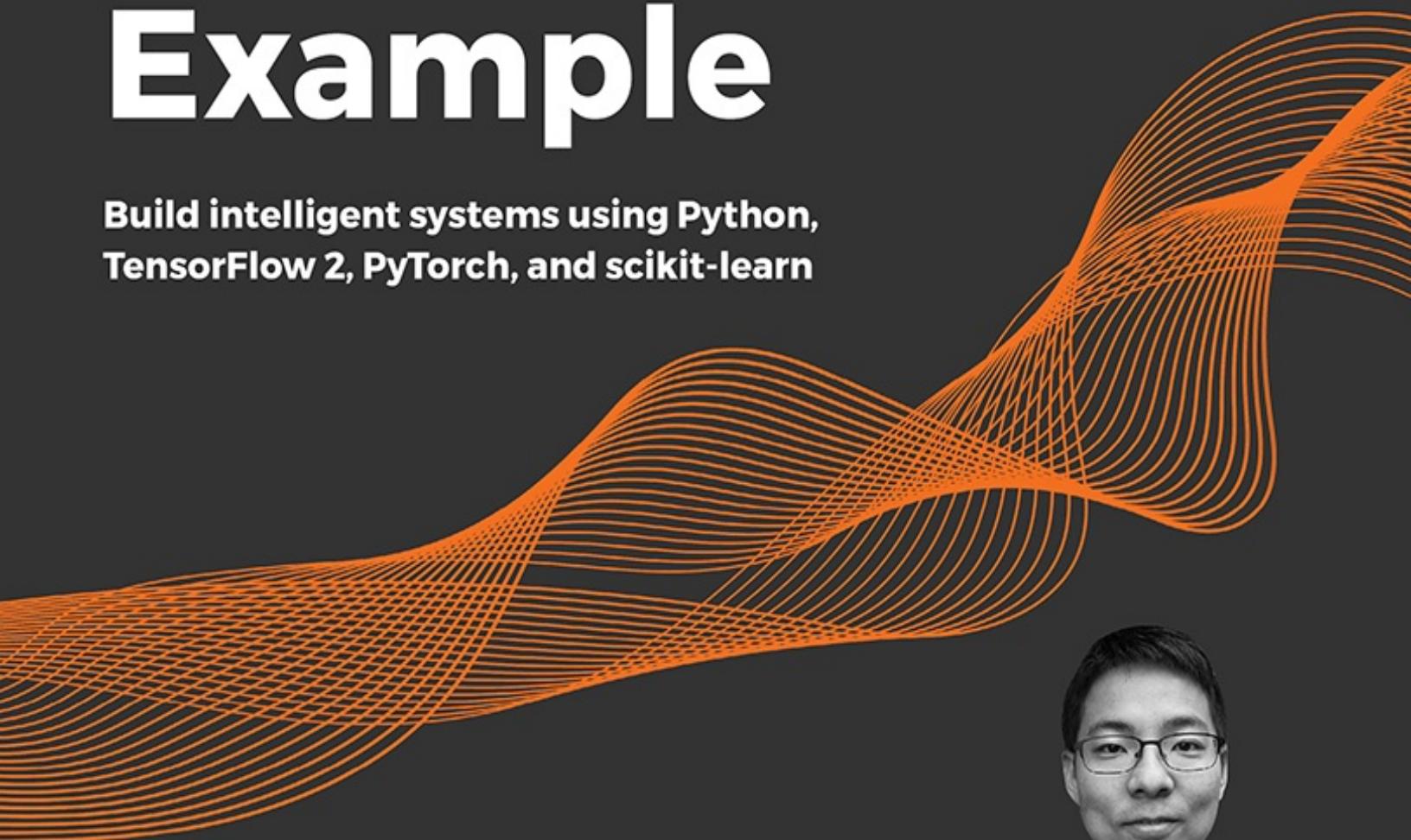


EXPERT INSIGHT

Python Machine Learning By Example

**Build intelligent systems using Python,
TensorFlow 2, PyTorch, and scikit-learn**



Third Edition



Yuxi (Hayden) Liu

Packt

Python Machine Learning By Example

Third Edition

Build intelligent systems using Python, TensorFlow
2, PyTorch, and scikit-learn

Yuxi (Hayden) Liu



BIRMINGHAM - MUMBAI

Python Machine Learning By Example

Third Edition

Copyright © 2020 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 or its dealers and distributors, will be held liable for any damages caused or alleged to have been 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.

Producer: Tushar Gupta

Acquisition Editor – Peer Reviews: Divya Mudaliar

Content Development Editor: Joanne Lovell

Technical Editor: Aditya Sawant

Project Editor: Janice Gonsalves

Copy Editor: Safis Editing

Proofreader: Safis Editing

Indexer: Tejal Daruwale Soni

Presentation Designer: Sandip Tadge

First published: May 2017

Second edition: February 2019

Third edition: October 2020

Production reference: 1281020

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-80020-971-8

www.packt.com



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

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.Packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.Packt.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.

Contributors

About the author

Yuxi (Hayden) Liu is a machine learning software engineer at Google. Previously he worked as a machine learning scientist in a variety of data-driven domains and applied his expertise in computational advertising, marketing, and cybersecurity. He is now developing and improving machine learning models and systems for ad optimization on the largest search engine in the world.

He is an education enthusiast, and the author of a series of machine learning books. His first book, the first edition of *Python Machine Learning By Example*, was ranked the #1 bestseller in Amazon back in 2017 and 2018, and was translated into many different languages. His other books include *R Deep Learning Projects*, *Hands-On Deep Learning Architectures with Python*, and *PyTorch 1.x Reinforcement Learning Cookbook*.

I would like to thank all the great people who made this book possible. Without any of you, this book would only exist in my mind. I would especially like to thank all of my editors at Packt Publishing, as well as my reviewers. Without them, this book would be harder to read and to apply to real-world problems. Last but not least, I'd like to thank all the readers for their support, which encouraged me to write the third edition of this book.

About the reviewers

Juantomás García leads and is the chief envisioning officer for Sngular's Data Science team. Since joining Sngular in 2018, Juantomás has leveraged his extensive experience to harness the potential of new technologies and implement them across the company's solutions and services.

Juantomás is a Google developer expert for cloud and machine learning, a co-author of the software book *La Pastilla Roja*, and the creator of "AbadIA", the artificial intelligence platform built to solve the popular Spanish game La Abadía del Crimen. He's an expert on free software technologies and has been a speaker at more than 200 international industry events. Among the various positions he has held during his 20-year career, he has been a data solutions manager at Open Sistemas, a chief data officer at ASPgems, and was the president of Hispanilux for seven years.

He studied IT engineering at the Universidad Politécnica de Madrid and plays an active role as a tech contributor and mentor to various academic organizations and startups. He regularly organizes Machine Learning Spain and GDG cloud Madrid meetups, is a mentor at Google Launchpad for entrepreneurs, and is also an advisor to Penn State University on its Deep Learning Hyperspectral Image Classification for EE project.

I want to thank my family for their support when I was working on revisions of this book. Thanks, Elisa, Nico, and Olivia.

Raghav Bali is a senior data scientist at one of the world's largest healthcare organizations. His work involves research and development of enterprise-level solutions based on machine learning, deep learning, and natural language processing for healthcare- and insurance-related use cases. He is also a mentor with Springboard and an active speaker at machine learning and deep learning conferences. In his previous role at Intel, he was involved in enabling proactive data-driven IT initiatives using natural language processing, deep learning, and traditional statistical methods. He has also worked in finance with American Express, working on digital engagement and customer retention solutions.

Raghav is the author of multiple well-received books on machine learning, deep learning, natural language processing, and transfer learning based on Python and R, and produced with leading publishers. His most recent books are *Hands-on Transfer Learning with Python*, *Practical Machine Learning with Python*, *Learning Social Media Analytics with R*, and *R Machine Learning by Example*.

I would like to take this opportunity to thank my wife, who has been a pillar of support. I would also like to thank my family for always supporting me in all my endeavors. Yuxi (Hayden) Liu is an excellent author, and I would like to thank and congratulate him on his new book. Last but not least, I would like to thank Divya Mudaliar, the whole Expert Network team, and Packt Publishing for the opportunity and their hard work in making this book a success.

Contents

Preface

Who this book is for

What this book covers

To get the most out of this book

Get in touch

1. Getting Started with Machine Learning and Python

An introduction to machine learning

Understanding why we need machine learning

Differentiating between machine learning and automation

Machine learning applications

Knowing the prerequisites

Getting started with three types of machine learning

A brief history of the development of machine learning algorithms

Digging into the core of machine learning

Generalizing with data

Overfitting, underfitting, and the bias-variance trade-off

Overfitting

Underfitting

The bias-variance trade-off

Avoiding overfitting with cross-validation

Avoiding overfitting with regularization

Avoiding overfitting with feature selection and dimensionality reduction

Data preprocessing and feature engineering

Preprocessing and exploration

Dealing with missing values

Label encoding

One-hot encoding

Scaling

Feature engineering

Polynomial transformation

Power transforms

Binning

Combining models

Voting and averaging

Bagging

Boosting

Stacking

Installing software and setting up

Setting up Python and environments

Installing the main Python packages

NumPy

SciPy

Pandas

Scikit-learn

TensorFlow

Introducing TensorFlow 2

Summary

Exercises

2. Building a Movie Recommendation Engine with Naïve Bayes

Getting started with classification

Binary classification

Multiclass classification

Multi-label classification

Exploring Naïve Bayes

Learning Bayes' theorem by example

The mechanics of Naïve Bayes

Implementing Naïve Bayes

Implementing Naïve Bayes from scratch

Implementing Naïve Bayes with scikit-learn

Building a movie recommender with Naïve Bayes

Evaluating classification performance

Tuning models with cross-validation

Summary

Exercise

References

3. Recognizing Faces with Support Vector Machine

Finding the separating boundary with SVM

Scenario 1 – identifying a separating hyperplane

Scenario 2 – determining the optimal hyperplane

[Scenario 3 – handling outliers](#)

[Implementing SVM](#)

[Scenario 4 – dealing with more than two classes](#)

[Scenario 5 – solving linearly non-separable problems with kernels](#)

[Choosing between linear and RBF kernels](#)

[Classifying face images with SVM](#)

[Exploring the face image dataset](#)

[Building an SVM-based image classifier](#)

[Boosting image classification performance with PCA](#)

[Fetal state classification on cardiotocography](#)

[Summary](#)

[Exercises](#)

4. [Predicting Online Ad Click-Through with Tree-Based Algorithms](#)

[A brief overview of ad click-through prediction](#)

[Getting started with two types of data – numerical and categorical](#)

[Exploring a decision tree from the root to the leaves](#)

[Constructing a decision tree](#)

[The metrics for measuring a split](#)

[Gini Impurity](#)

[Information Gain](#)

[Implementing a decision tree from scratch](#)

[Implementing a decision tree with scikit-learn](#)

[Predicting ad click-through with a decision tree](#)

[Ensembling decision trees – random forest](#)

[Ensembling decision trees – gradient boosted trees](#)

[Summary](#)

[Exercises](#)

5. [Predicting Online Ad Click-Through with Logistic Regression](#)

[Converting categorical features to numerical—one-hot encoding and ordinal encoding](#)

[Classifying data with logistic regression](#)

[Getting started with the logistic function](#)

[Jumping from the logistic function to logistic regression](#)

[Training a logistic regression model](#)

[Training a logistic regression model using gradient descent](#)

[Predicting ad click-through with logistic regression using gradient descent](#)

[Training a logistic regression model using stochastic gradient descent](#)

[Training a logistic regression model with regularization](#)

[Feature selection using L1 regularization](#)

[Training on large datasets with online learning](#)

[Handling multiclass classification](#)

[Implementing logistic regression using TensorFlow](#)

[Feature selection using random forest](#)

[Summary](#)

[Exercises](#)

6. [Scaling Up Prediction to Terabyte Click Logs](#)

[Learning the essentials of Apache Spark](#)

[Breaking down Spark](#)

[Installing Spark](#)

[Launching and deploying Spark programs](#)

[Programming in PySpark](#)

[Learning on massive click logs with Spark](#)

[Loading click logs](#)

[Splitting and caching the data](#)

[One-hot encoding categorical features](#)

[Training and testing a logistic regression model](#)

[Feature engineering on categorical variables with Spark](#)

[Hashing categorical features](#)

[Combining multiple variables – feature interaction](#)

[Summary](#)

[Exercises](#)

7. [Predicting Stock Prices with Regression Algorithms](#)

[A brief overview of the stock market and stock prices](#)

[What is regression?](#)

[Mining stock price data](#)

[Getting started with feature engineering](#)

[Acquiring data and generating features](#)

[Estimating with linear regression](#)

[How does linear regression work?](#)

[Implementing linear regression from scratch](#)

[Implementing linear regression with scikit-learn](#)
[Implementing linear regression with TensorFlow](#)
[Estimating with decision tree regression](#)
[Transitioning from classification trees to regression trees](#)
[Implementing decision tree regression](#)
[Implementing a regression forest](#)
[Estimating with support vector regression](#)
[Implementing SVR](#)
[Evaluating regression performance](#)
[Predicting stock prices with the three regression algorithms](#)
[Summary](#)
[Exercises](#)

8. [Predicting Stock Prices with Artificial Neural Networks](#)

[Demystifying neural networks](#)
[Starting with a single-layer neural network](#)
[Layers in neural networks](#)
[Activation functions](#)
[Backpropagation](#)
[Adding more layers to a neural network: DL](#)
[Building neural networks](#)
[Implementing neural networks from scratch](#)
[Implementing neural networks with scikit-learn](#)
[Implementing neural networks with TensorFlow](#)
[Picking the right activation functions](#)
[Preventing overfitting in neural networks](#)
[Dropout](#)
[Early stopping](#)
[Predicting stock prices with neural networks](#)
[Training a simple neural network](#)
[Fine-tuning the neural network](#)
[Summary](#)
[Exercise](#)

9. [Mining the 20 Newsgroups Dataset with Text Analysis Techniques](#)

[How computers understand language – NLP](#)
[What is NLP?](#)
[The history of NLP](#)
[NLP applications](#)

Touring popular NLP libraries and picking up NLP basics

Installing famous NLP libraries

Corpora

Tokenization

PoS tagging

NER

Stemming and lemmatization

Semantics and topic modeling

Getting the newsgroups data

Exploring the newsgroups data

Thinking about features for text data

Counting the occurrence of each word token

Text preprocessing

Dropping stop words

Reducing inflectional and derivational forms of words

Visualizing the newsgroups data with t-SNE

What is dimensionality reduction?

t-SNE for dimensionality reduction

Summary

Exercises

10. Discovering Underlying Topics in the Newsgroups Dataset with Clustering and Topic Modeling

Learning without guidance – unsupervised learning

Clustering newsgroups data using k-means

How does k-means clustering work?

Implementing k-means from scratch

Implementing k-means with scikit-learn

Choosing the value of k

Clustering newsgroups data using k-means

Discovering underlying topics in newsgroups

Topic modeling using NMF

Topic modeling using LDA

Summary

Exercises

11. Machine Learning Best Practices

Machine learning solution workflow

Best practices in the data preparation stage

Best practice 1 – Completely understanding the project goal

Best practice 2 – Collecting all fields that are relevant

Best practice 3 – Maintaining the consistency of field values

Best practice 4 – Dealing with missing data

Best practice 5 – Storing large-scale data

Best practices in the training sets generation stage

Best practice 6 – Identifying categorical features with numerical values

Best practice 7 – Deciding whether to encode categorical features

Best practice 8 – Deciding whether to select features, and if so, how to do so

Best practice 9 – Deciding whether to reduce dimensionality, and if so, how to do so

Best practice 10 – Deciding whether to rescale features

Best practice 11 – Performing feature engineering with domain expertise

Best practice 12 – Performing feature engineering without domain expertise

Binarization

Discretization

Interaction

Polynomial transformation

Best practice 13 – Documenting how each feature is generated

Best practice 14 – Extracting features from text data

Tf and tf-idf

Word embedding

Word embedding with pre-trained models

Best practices in the model training, evaluation, and selection stage

Best practice 15 – Choosing the right algorithm(s) to start with

Naïve Bayes

Logistic regression

SVM

Random forest (or decision tree)

Neural networks

Best practice 16 – Reducing overfitting

Best practice 17 – Diagnosing overfitting and underfitting

Best practice 18 – Modeling on large-scale datasets

Best practices in the deployment and monitoring stage

Best practice 19 – Saving, loading, and reusing models

Saving and restoring models using pickle

Saving and restoring models in TensorFlow

Best practice 20 – Monitoring model performance

Best practice 21 – Updating models regularly

Summary

Exercises

12. Categorizing Images of Clothing with Convolutional Neural Networks

Getting started with CNN building blocks

The convolutional layer

The nonlinear layer

The pooling layer

Architecting a CNN for classification

Exploring the clothing image dataset

Classifying clothing images with CNNs

Architecting the CNN model

Fitting the CNN model

Visualizing the convolutional filters

Boosting the CNN classifier with data augmentation

Horizontal flipping for data augmentation

Rotation for data augmentation

Shifting for data augmentation

Improving the clothing image classifier with data augmentation

Summary

Exercises

13. Making Predictions with Sequences Using Recurrent Neural Networks

Introducing sequential learning

Learning the RNN architecture by example

Recurrent mechanism

Many-to-one RNNs

One-to-many RNNs

Many-to-many (synced) RNNs

[Many-to-many \(unsynced\) RNNs](#)

[Training an RNN model](#)

[Overcoming long-term dependencies with Long Short-Term Memory](#)

[Analyzing movie review sentiment with RNNs](#)

[Analyzing and preprocessing the data](#)

[Building a simple LSTM network](#)

[Stacking multiple LSTM layers](#)

[Writing your own War and Peace with RNNs](#)

[Acquiring and analyzing the training data](#)

[Constructing the training set for the RNN text generator](#)

[Building an RNN text generator](#)

[Training the RNN text generator](#)

[Advancing language understanding with the Transformer model](#)

[Exploring the Transformer's architecture](#)

[Understanding self-attention](#)

[Summary](#)

[Exercises](#)

14. [Making Decisions in Complex Environments with Reinforcement Learning](#)

[Setting up the working environment](#)

[Installing PyTorch](#)

[Installing OpenAI Gym](#)

[Introducing reinforcement learning with examples](#)

[Elements of reinforcement learning](#)

[Cumulative rewards](#)

[Approaches to reinforcement learning](#)

[Solving the FrozenLake environment with dynamic programming](#)

[Simulating the FrozenLake environment](#)

[Solving FrozenLake with the value iteration algorithm](#)

[Solving FrozenLake with the policy iteration algorithm](#)

[Performing Monte Carlo learning](#)

[Simulating the Blackjack environment](#)

[Performing Monte Carlo policy evaluation](#)

[Performing on-policy Monte Carlo control](#)

[Solving the Taxi problem with the Q-learning algorithm](#)

[Simulating the Taxi environment](#)

[Developing the Q-learning algorithm](#)

[Summary](#)

[Exercises](#)

[Other Books You May Enjoy](#)

[Index](#)

Preface

Python Machine Learning By Example, Third Edition serves as a comprehensive gateway into the world of **machine learning (ML)**.

With six new chapters, covering topics such as movie recommendation engine development with Naïve Bayes, recognizing faces with support vector machines, predicting stock prices with artificial neural networks, categorizing images of clothing with convolutional neural networks, predicting with sequences using recurring neural networks, and leveraging reinforcement learning for decision making, the book has been considerably updated for the latest enterprise requirements.

At the same time, the book provides actionable insights on the key fundamentals of ML with Python programming. Hayden applies his expertise to demonstrate implementations of algorithms in Python, both from scratch and with libraries such as TensorFlow and Keras.

Each chapter walks through an industry-adopted application. With the help of realistic examples, you will gain an understanding of the mechanics of ML techniques in areas such as exploratory data analysis, feature engineering, classification, regression, clustering, and natural language processing.

By the end of this book, you will have gained a broad picture of the ML ecosystem and will be well-versed in the best practices of applying ML techniques with Python to solve problems.

Who this book is for

If you're a machine learning enthusiast, data analyst, or data engineer who's highly passionate about machine learning and you want to begin working on ML assignments, this book is for you.

Prior knowledge of Python coding is assumed and basic familiarity with statistical concepts will be beneficial, although this is not necessary.

What this book covers

Chapter 1, Getting Started with Machine Learning and Python, will kick off your Python machine learning journey. It will start with what machine learning is, why we need it, and its evolution over the last few decades. It will then discuss typical machine learning tasks and explore several essential techniques of working with data and working with models, in a practical and fun way. You will also set up the software and tools needed for examples and projects in the upcoming chapters.

Chapter 2, Building a Movie Recommendation Engine with Naïve Bayes, will focus on classification, specifically binary classification and Naïve Bayes. The goal of the chapter is to build a movie recommendation system. You will learn the fundamental concepts of classification, and about Naïve Bayes, a simple yet powerful algorithm. It will also demonstrate how to fine-tune a model, which is an important skill for every data science or machine learning practitioner to learn.

Chapter 3, Recognizing Faces with Support Vector Machine, will continue the journey of supervised learning and classification. Specifically, it will focus on multiclass classification and support vector machine classifiers. It will discuss how the support vector machine algorithm searches for a decision boundary in order to separate data from different classes. Also, you will implement the algorithm with scikit-learn, and apply it to solve various real-life problems including face recognition.

Chapter 4, Predicting Online Ad Click-Through with Tree-Based Algorithms, will introduce and explain in depth tree-based algorithms (including decision trees, random forests, and boosted trees) throughout the course of solving the advertising click-through rate problem. You will explore decision trees from the root to the leaves, and work on implementations of tree models from scratch, using scikit-learn and XGBoost. Feature importance, feature selection, and ensemble will be covered alongside.

Chapter 5, Predicting Online Ad Click-Through with Logistic Regression, will be a continuation of the ad click-through prediction project, with a focus on a very scalable classification model—logistic regression. You will explore how logistic regression works, and how to work with large datasets. The chapter will also cover categorical variable encoding, L1 and L2 regularization, feature selection, online learning, and stochastic gradient descent.

Chapter 6, Scaling Up Prediction to Terabyte Click Logs, will be about a more scalable solution to massive ad click prediction, utilizing powerful parallel computing tools including Apache Hadoop and Spark. It will cover the essential concepts of Spark such as installation, RDD, and core programming, as well its ML components. You will work with the entire ad click dataset, build classification models, and perform feature engineering and performance evaluation using Spark.

Chapter 7, Predicting Stock Prices with Regression Algorithms, will focus on several popular regression algorithms, including linear regression, regression tree and regression forest, and support vector regression. It will encourage you to utilize them to tackle a billion (or trillion) dollar problem—stock price prediction. You will practice solving regression problems using scikit-learn and TensorFlow.

Chapter 8, Predicting Stock Prices with Artificial Neural Networks, will introduce and explain in depth neural network models. It will cover the building blocks of neural networks, and important concepts such as activation functions, feedforward, and backpropagation. You will start by building the simplest neural network and go deeper by adding more layers to it. We will implement neural networks from scratch, use TensorFlow and Keras, and train a neural network to predict stock prices.

Chapter 9, Mining the 20 Newsgroups Dataset with Text Analysis Techniques, will start the second step of your learning journey—unsupervised learning. It will explore a natural language processing problem—exploring newsgroups data. You will gain hands-on experience in working with text data, especially how to convert words and phrases into machine-readable values and how to clean up words with little meaning.

You will also visualize text data using a dimension reduction technique called t-SNE.

Chapter 10, Discovering Underlying Topics in the Newsgroups Dataset with Clustering and Topic Modeling, will talk about identifying different groups of observations from data in an unsupervised manner. You will cluster the newsgroups data using the K-means algorithm, and detect topics using non-negative matrix factorization and latent Dirichlet allocation. You will be amused by how many interesting themes you are able to mine from the 20 newsgroups dataset!

Chapter 11, Machine Learning Best Practices, will aim to fully prove your learning and get you ready for real-world projects. It includes 21 best practices to follow throughout the entire machine learning workflow.

Chapter 12, Categorizing Images of Clothing with Convolutional Neural Networks, will be about using convolutional neural networks (CNNs), a very powerful modern machine learning model, to classify images of clothing. It will cover the building blocks and architecture of CNNs, and their implementation using TensorFlow and Keras. After exploring the data of clothing images, you will develop CNN models to categorize the images into ten classes, and utilize data augmentation techniques to boost the classifier.

Chapter 13, Making Predictions with Sequences using Recurrent Neural Networks, will start by defining sequential learning, and exploring how recurrent neural networks (RNNs) are well suited for it. You will learn about various types of RNNs and their common applications. You will implement RNNs with TensorFlow, and apply them to solve two interesting sequential learning problems: sentiment analysis on IMDb movie reviews and text auto-generation. Finally, as a bonus section, it will cover the Transformer as a state-of-the-art sequential learning model.

Chapter 14, Making Decisions in Complex Environments with Reinforcement Learning, will be about learning from experience, and interacting with the environment. After exploring the fundamentals of reinforcement learning, you will explore the `FrozenLake` environment with a simple dynamic programming algorithm. You will learn about Monte

Carlo learning and use it for value approximation and control. You will also develop temporal difference algorithms and use Q-learning to solve the taxi problem.

To get the most out of this book

You are expected to have a basic foundation of knowledge of Python, the basic machine learning algorithms, and some basic Python libraries, such as TensorFlow and Keras, in order to create smart cognitive actions for your projects.

Download the example code files

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

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781800209718_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example; "Then, we'll load the `en_core_web_sm` model and parse the sentence using this model."

A block of code is set as follows:

```
>>> from sklearn import datasets  
>>> iris = datasets.load_iris()  
>>> X = iris.data[:, 2:4]  
>>> y = iris.target
```

Any command-line input or output is written as follows:

```
conda install pytorch torchvision -c pytorch
```

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "A new window will pop up and ask us which collections (the **Collections** tab in the following screenshot) or corpus (the identifiers in the **Corpora** tab in the following screenshot) to download and where to keep the data."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

Getting Started with Machine Learning and Python

It is believed that in the next 30 years, **artificial intelligence** (AI) will outpace human knowledge. Regardless of whether it will lead to job losses, analytical and **machine learning** skills are becoming increasingly important. In fact, this point has been emphasized by the most influential business leaders, including the Microsoft co-founder, Bill Gates, Tesla CEO, Elon Musk, and former Google executive chairman, Eric Schmidt.

In this chapter, we will kick off our machine learning journey with the basic, yet important, concepts of machine learning. We will start with what machine learning is all about, why we need it, and its evolution over a few decades. We will then discuss typical machine learning tasks and explore several essential techniques of working with data and working with models.

At the end of the chapter, we will also set up the software for Python, the most popular language for machine learning and data science, and its libraries and tools that are required for this book.

We will go into detail on the following topics:

- The importance of machine learning
- The core of machine learning—generalizing with data
- Overfitting and underfitting
- The bias-variance trade-off
- Techniques to avoid overfitting
- Techniques for data preprocessing

- Techniques for feature engineering
- Techniques for model aggregation
- Setting up a Python environment
- Installing the main Python packages
- Introducing TensorFlow 2

An introduction to machine learning

In this first section, we will kick off our machine learning journey with a brief introduction to machine learning, why we need it, how it differs from automation, and how it improves our lives.

Machine learning is a term that was coined around 1960, consisting of two words—**machine**, which corresponds to a computer, robot, or other device, and **learning**, which refers to an activity intended to acquire or discover event patterns, which we humans are good at. Interesting examples include facial recognition, translation, responding to emails, and making data-driven business decisions. You will see many more examples throughout this book.

Understanding why we need machine learning

Why do we need machine learning and why do we want a machine to learn the same way as a human? We can look at it from three main perspectives: maintenance, risk mitigation, and advantages.

First and foremost, of course, computers and robots can work 24/7 and don't get tired, need breaks, call in sick, or go on strike. Machines cost a lot less in the long run. Also, for sophisticated problems that involve a variety of

huge datasets or complex calculations, it's much more justifiable, not to mention intelligent, to let computers do all of the work. Machines driven by algorithms that are designed by humans are able to learn latent rules and inherent patterns, enabling them to carry out tasks.

Learning machines are better suited than humans for tasks that are routine, repetitive, or tedious. Beyond that, automation by machine learning can mitigate risks caused by fatigue or inattention.

Self-driving cars, as shown in *Figure 1.1*, are a great example: a vehicle is capable of navigating by sensing its environment and making decisions without human input. Another example is the use of robotic arms in production lines, which are capable of causing a significant reduction in injuries and costs.



Figure 1.1: An example of a self-driving car

Let's assume that humans don't fatigue or we have the resources to hire enough shift workers; would machine learning still have a place? Of course, it would! There are many cases, reported and unreported, where machines

perform comparably, or even better, than domain experts. As algorithms are designed to learn from the ground truth, and the best thought-out decisions made by human experts, machines can perform just as well as experts.

In reality, even the best expert makes mistakes. Machines can minimize the chance of making wrong decisions by utilizing collective intelligence from individual experts. A major study that identified that machines are better than doctors at diagnosing certain types of cancer is a proof of this philosophy (<https://www.nature.com/articles/d41586-020-00847-2>). **AlphaGo** (<https://deepmind.com/research/case-studies/alphago-the-story-so-far>) is probably the best-known example of machines beating humans.

Also, it's much more scalable to deploy learning machines than to train individuals to become experts, from the perspective of economic and social barriers. We can distribute thousands of diagnostic devices across the globe within a week, but it's almost impossible to recruit and assign the same number of qualified doctors.

You may argue against this: what if we have sufficient resources and the capacity to hire the best domain experts and later aggregate their opinions—would machine learning still have a place? Probably not (at least right now)—learning machines might not perform better than the joint efforts of the most intelligent humans. However, individuals equipped with learning machines can outperform the best group of experts. This is actually an emerging concept called **AI-based assistance** or **AI plus human intelligence**, which advocates for combining the efforts of machines and humans. We can summarize the previous statement in the following inequality:

$$\text{human} + \text{machine learning} \rightarrow \text{most intelligent tireless human} \geq \text{machine learning} > \text{human}$$

A medical operation involving robots is one great example of human and machine learning synergy. *Figure 1.2* shows robotic arms in an operation room alongside the surgeon:



Figure 1.2: AI-assisted surgery

Differentiating between machine learning and automation

So, does machine learning simply equate to automation that involves the programming and execution of human-crafted or human-curated rule sets? A popular myth says that machine learning is the same as automation because it performs instructive and repetitive tasks and thinks no further. If the answer to that question is yes, why can't we just hire many software programmers and continue programming new rules or extending old rules?

One reason is that defining, maintaining, and updating rules becomes increasingly expensive over time. The number of possible patterns for an activity or event could be enormous and, therefore, exhausting all enumeration isn't practically feasible. It gets even more challenging when it comes to events that are dynamic, ever-changing, or evolving in real time. It's much easier and more efficient to develop learning algorithms that command computers to learn, extract patterns, and to figure things out themselves from abundant data.

The difference between machine learning and traditional programming can be seen in *Figure 1.3*:

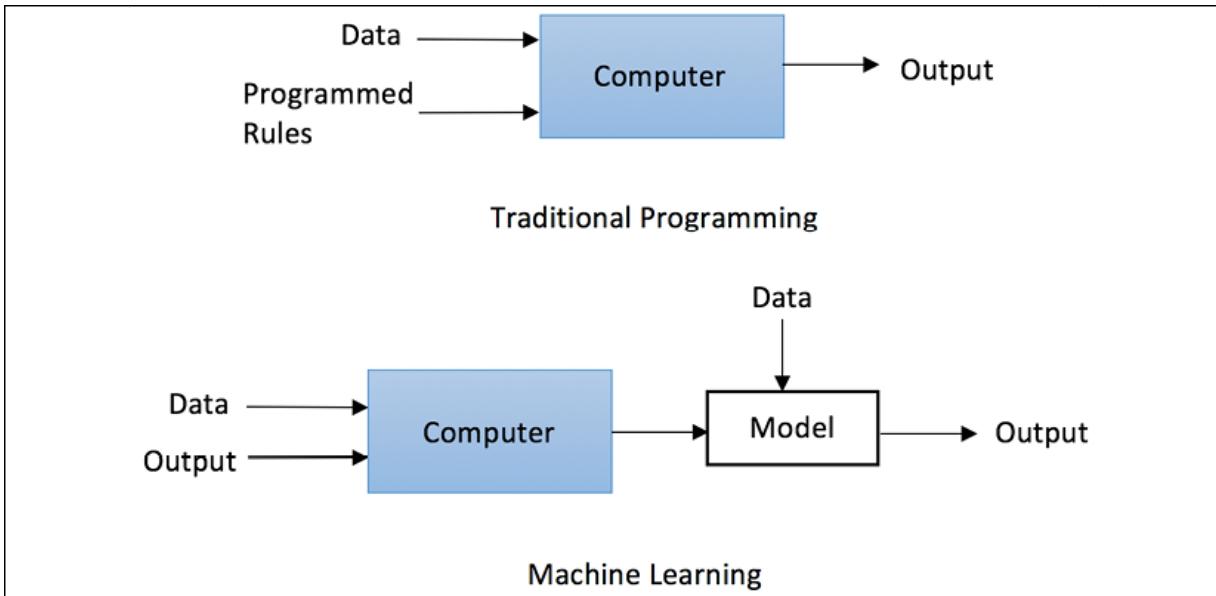


Figure 1.3: Machine learning versus traditional programming

In traditional programming, the computer follows a set of predefined rules to process the input data and produce the outcome. In machine learning, the computer tries to mimic human thinking. It interacts with the input data, expected outcome, and environment, and it derives patterns that are represented by one or more mathematical models. The models are then used to interact with future input data and to generate outcomes. Unlike in automation, the computer in a machine learning setting doesn't receive explicit and instructive coding.

The volume of data is growing exponentially. Nowadays, the floods of textual, audio, image, and video data are hard to fathom. The **Internet of Things (IoT)** is a recent development of a new kind of Internet, which interconnects everyday devices. The IoT will bring data from household appliances and autonomous cars to the fore. This trend is likely to continue and we will have more data that is generated and processed. Besides the quantity, the quality of data available has kept increasing in the past few years due to cheaper storage. This has empowered the evolution of machine learning algorithms and data-driven solutions.

Machine learning applications

Jack Ma, co-founder of the e-commerce company Alibaba, explained in a speech that IT was the focus of the past 20 years but, for the next 30 years, we will be in the age of **data technology (DT)**

(<https://www.alizila.com/jack-ma-dont-fear-smarter-computers/>). During the age of IT, companies grew larger and stronger thanks to computer software and infrastructure. Now that businesses in most industries have already gathered enormous amounts of data, it's presently the right time to exploit DT to unlock insights, derive patterns, and boost new business growth. Broadly speaking, machine learning technologies enable businesses to better understand customer behavior, engage with customers, and optimize operations management.

As for us individuals, machine learning technologies are already making our lives better every day. One application of machine learning with which we're all familiar is spam email filtering. Another is online advertising, where adverts are served automatically based on information advertisers have collected about us. Stay tuned for the next few chapters, where you will learn how to develop algorithms for solving these two problems and more.

A search engine is an application of machine learning we can't imagine living without. It involves information retrieval, which parses what we look for, queries related top records, and applies contextual ranking and personalized ranking, which sorts pages by topical relevance and user preference. E-commerce and media companies have been at the forefront of employing recommendation systems, which help customers to find products, services, and articles faster.

The application of machine learning is boundless and we just keep hearing new examples everyday: credit card fraud detection, presidential election prediction, instant speech translation, and robo advisors—you name it!

In the 1983 *War Games* movie, a computer made life-and-death decisions that could have resulted in World War III. As far as we know, technology wasn't able to pull off such feats at the time. However, in 1997, the Deep Blue supercomputer did manage to beat a world chess champion ([https://en.wikipedia.org/wiki/Deep_Blue_\(chess_computer\)](https://en.wikipedia.org/wiki/Deep_Blue_(chess_computer))

[puter](#)). In 2005, a Stanford self-driving car drove by itself for more than 130 miles in a desert

([https://en.wikipedia.org/wiki/DARPA_Grand_Challenge_\(2005\)](https://en.wikipedia.org/wiki/DARPA_Grand_Challenge_(2005))). In 2007, the car of another team drove through regular urban traffic for more than 60 miles

([https://en.wikipedia.org/wiki/DARPA_Grand_Challenge_\(2007\)](https://en.wikipedia.org/wiki/DARPA_Grand_Challenge_(2007))). In 2011, the Watson computer won a quiz against human opponents

([https://en.wikipedia.org/wiki/Watson_\(computer\)](https://en.wikipedia.org/wiki/Watson_(computer))). As mentioned earlier, the AlphaGo program beat one of the best Go players in the world in 2016. If we assume that computer hardware is the limiting factor, then we can try to extrapolate into the future. A famous American inventor and futurist Ray Kurzweil did just that and, according to him, we can expect human-level intelligence around 2029. What's next?

Can't wait to launch your own machine learning journey? Let's start with the prerequisites, and the basic types of machine learning.

Knowing the prerequisites

Machine learning mimicking human intelligence is a subfield of AI—a field of computer science concerned with creating systems. Software engineering is another field in computer science. Generally, we can label Python programming as a type of software engineering. Machine learning is also closely related to linear algebra, probability theory, statistics, and mathematical optimization. We usually build machine learning models based on statistics, probability theory, and linear algebra, and then optimize the models using mathematical optimization.

The majority of you reading this book should have a good, or at least sufficient, command of Python programming. Those who aren't feeling confident about mathematical knowledge might be wondering how much time should be spent learning or brushing up on the aforementioned subjects. Don't panic: we will get machine learning to work for us without going into any mathematical details in this book. It just requires some basic

101 knowledge of probability theory and linear algebra, which helps us to understand the mechanics of machine learning techniques and algorithms. And it gets easier as we will be building models both from scratch and with popular packages in Python, a language we like and are familiar with.

For those who want to learn or brush up on probability theory and linear algebra, feel free to search for basic probability theory and basic linear algebra. There are a lot of resources available online, for example, https://people.ucsc.edu/~abrsrn/intro_prob_1.pdf regarding probability 101, and <http://www.maths.gla.ac.uk/~ajb/dvi-ps/2w-notes.pdf> regarding basic linear algebra.

Those who want to study machine learning systematically can enroll in computer science, **AI**, and, more recently, data science master's programs. There are also various data science boot camps. However, the selection for boot camps is usually stricter as they're more job-oriented and the program duration is often short, ranging from four to 10 weeks. Another option is the free **Massive Open Online Courses (MOOCs)**, Andrew Ng's popular course on machine learning. Last but not least, industry blogs and websites are great resources for us to keep up with the latest developments.

Machine learning isn't only a skill but also a bit of sport. We can compete in several machine learning competitions, such as Kaggle (www.kaggle.com)—sometimes for decent cash prizes, sometimes for joy, and most of the time to play to our strengths. However, to win these competitions, we may need to utilize certain techniques, which are only useful in the context of competitions and not in the context of trying to solve a business problem. That's right, the **no free lunch theorem** (https://en.wikipedia.org/wiki/No_free_lunch_theorem) applies here.

Next, we'll take a look at the three types of machine learning.

Getting started with three types of machine learning

A machine learning system is fed with input data—this can be numerical, textual, visual, or audiovisual. The system usually has an output—this can be a floating-point number, for instance, the acceleration of a self-driving car, or it can be an integer representing a category (also called a **class**), for example, a cat or tiger from image recognition.

The main task of machine learning is to explore and construct algorithms that can learn from historical data and make predictions on new input data. For a data-driven solution, we need to define (or have it defined by an algorithm) an evaluation function called **loss** or **cost function**, which measures how well the models are learning. In this setup, we create an optimization problem with the goal of learning in the most efficient and effective way.

Depending on the nature of the learning data, machine learning tasks can be broadly classified into the following three categories:

- **Unsupervised learning:** When the learning data only contains indicative signals without any description attached, it's up to us to find the structure of the data underneath, to discover hidden information, or to determine how to describe the data. This kind of learning data is called **unlabeled** data. Unsupervised learning can be used to detect anomalies, such as fraud or defective equipment, or to group customers with similar online behaviors for a marketing campaign. Data visualization that makes data more digestible, and dimensionality reduction that distills relevant information from noisy data, are also in the family of unsupervised learning.
- **Supervised learning:** When learning data comes with a description, targets, or desired output besides indicative signals, the learning goal is to find a general rule that maps input to output. This kind of learning data is called **labeled** data. The learned rule is then used to label new data with unknown output. The labels are usually provided by event-

logging systems or evaluated by human experts. Besides, if it's feasible, they may also be produced by human raters, through crowd sourcing, for instance. Supervised learning is commonly used in daily applications, such as face and speech recognition, products or movie recommendations, sales forecasting, and spam email detection.

- **Reinforcement learning:** Learning data provides feedback so that the system adapts to dynamic conditions in order to achieve a certain goal in the end. The system evaluates its performance based on the feedback responses and reacts accordingly. The best-known instances include robotics for industrial automation, self-driving cars, and the chess master, AlphaGo. The key difference between reinforcement learning and supervised learning is the interaction with the environment.

The following diagram depicts types of machine learning tasks:

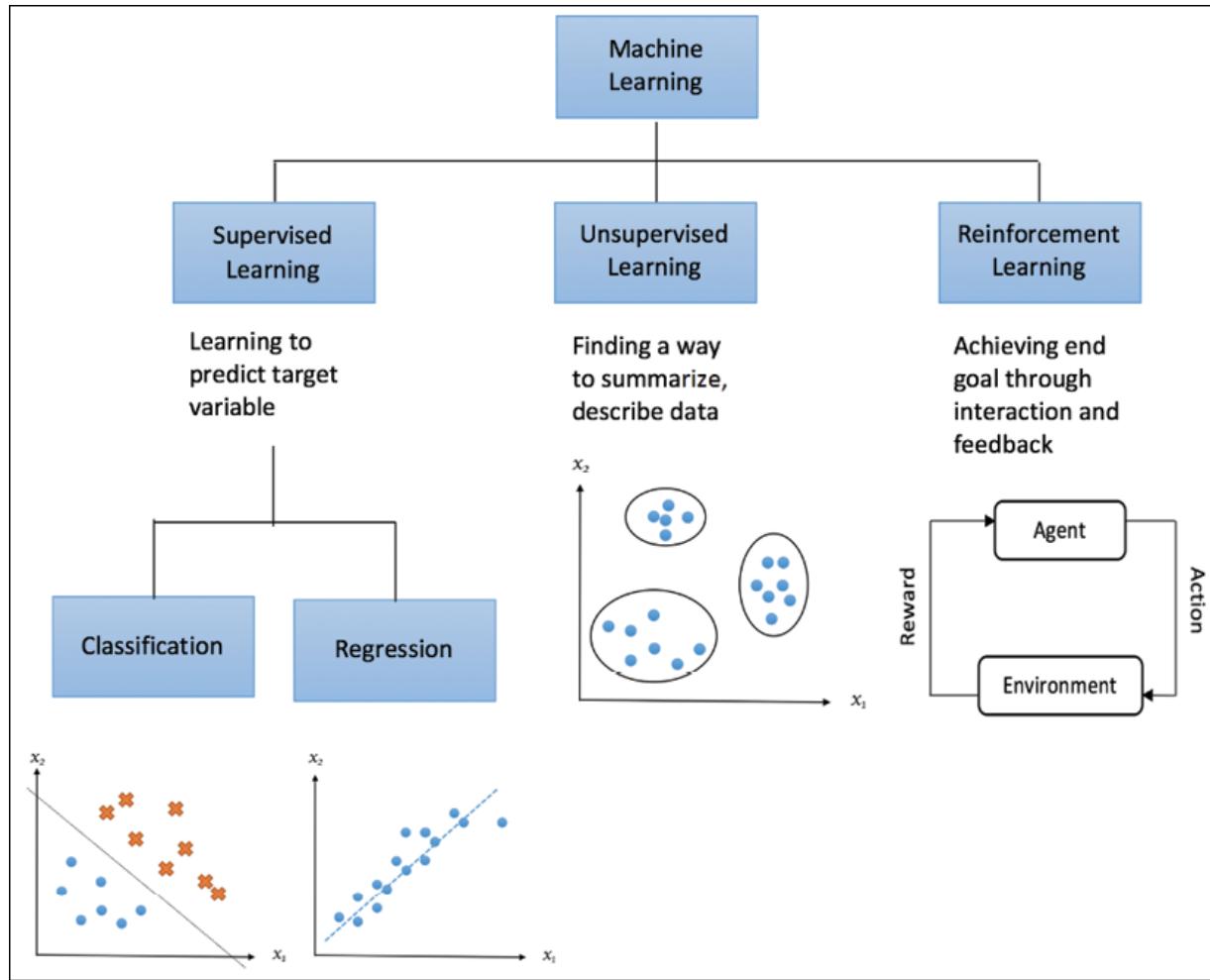


Figure 1.4: Types of machine learning tasks

As shown in the diagram, we can further subdivide supervised learning into regression and classification. **Regression** trains on and predicts continuous-valued responses, for example, predicting house prices, while **classification** attempts to find the appropriate class label, such as analyzing a positive/negative sentiment and predicting a loan default.

If not all learning samples are labeled, but some are, we will have **semi-supervised learning**. This makes use of unlabeled data (typically a large amount) for training, besides a small amount of labeled data. Semi-supervised learning is applied in cases where it is expensive to acquire a fully labeled dataset and more practical to label a small subset. For example, it often requires skilled experts to label hyperspectral remote sensing images, while acquiring unlabeled data is relatively easy.

Feeling a little bit confused by the abstract concepts? Don't worry. We will encounter many concrete examples of these types of machine learning tasks later in this book. For example, in *Chapter 2, Building a Movie Recommendation Engine with Naïve Bayes*, we will dive into supervised learning classification and its popular algorithms and applications. Similarly, in *Chapter 7, Predicting Stock Prices with Regression*, we will explore supervised learning regression. We will focus on unsupervised techniques and algorithms in *Chapter 9, Mining the 20 Newsgroups Dataset with Text Analysis Techniques*. Last but not least, the third machine learning task, reinforcement learning, will be covered in *Chapter 14, Making Decisions in Complex Environments with Reinforcement Learning*.

Besides categorizing machine learning based on the learning task, we can categorize it in a chronological way.

A brief history of the development of machine learning algorithms

In fact, we have a whole zoo of machine learning algorithms that have experienced varying popularity over time. We can roughly categorize them into four main approaches: logic-based learning, statistical learning, artificial neural networks, and genetic algorithms.

The **logic-based** systems were the first to be dominant. They used basic rules specified by human experts and, with these rules, systems tried to reason using formal logic, background knowledge, and hypotheses.

Statistical learning theory attempts to find a function to formalize the relationships between variables. In the mid-1980s, **artificial neural networks (ANNs)** came to the fore, only to then be pushed aside by statistical learning systems in the 1990s. ANNs imitate animal brains and consist of interconnected neurons that are also an imitation of biological neurons. They try to model complex relationships between input and output values and to capture patterns in data. **Genetic algorithms (GA)** were popular in the 1990s. They mimic the biological process of evolution and try to find the optimal solutions using methods such as mutation and crossover.

We are currently seeing a revolution in **deep learning**, which we might consider a rebranding of neural networks. The term deep learning was coined around 2006 and refers to deep neural networks with many layers. The breakthrough in deep learning was the result of the integration and utilization of **Graphical Processing Units (GPUs)**, which massively speed up computation.

GPUs were originally developed to render video games and are very good in parallel matrix and vector algebra. It's believed that deep learning resembles the way humans learn. Therefore, it may be able to deliver on the promise of sentient machines. Of course, in this book, we will dig deep into deep learning in *Chapter 12, Categorizing Images of Clothing with Convolutional Neural Networks*, and *Chapter 13, Making Predictions with Sequences Using Recurrent Neural Networks*, after touching on it in *Chapter 8, Predicting Stock Prices with Artificial Neural Networks*.

Some of us may have heard of **Moore's law**—an empirical observation claiming that computer hardware improves exponentially with time. The law was first formulated by Gordon Moore, the co-founder of Intel, in 1965. According to the law, the number of transistors on a chip should double every two years. In the following diagram, you can see that the law holds up nicely (the size of the bubbles corresponds to the average transistor count in GPUs):

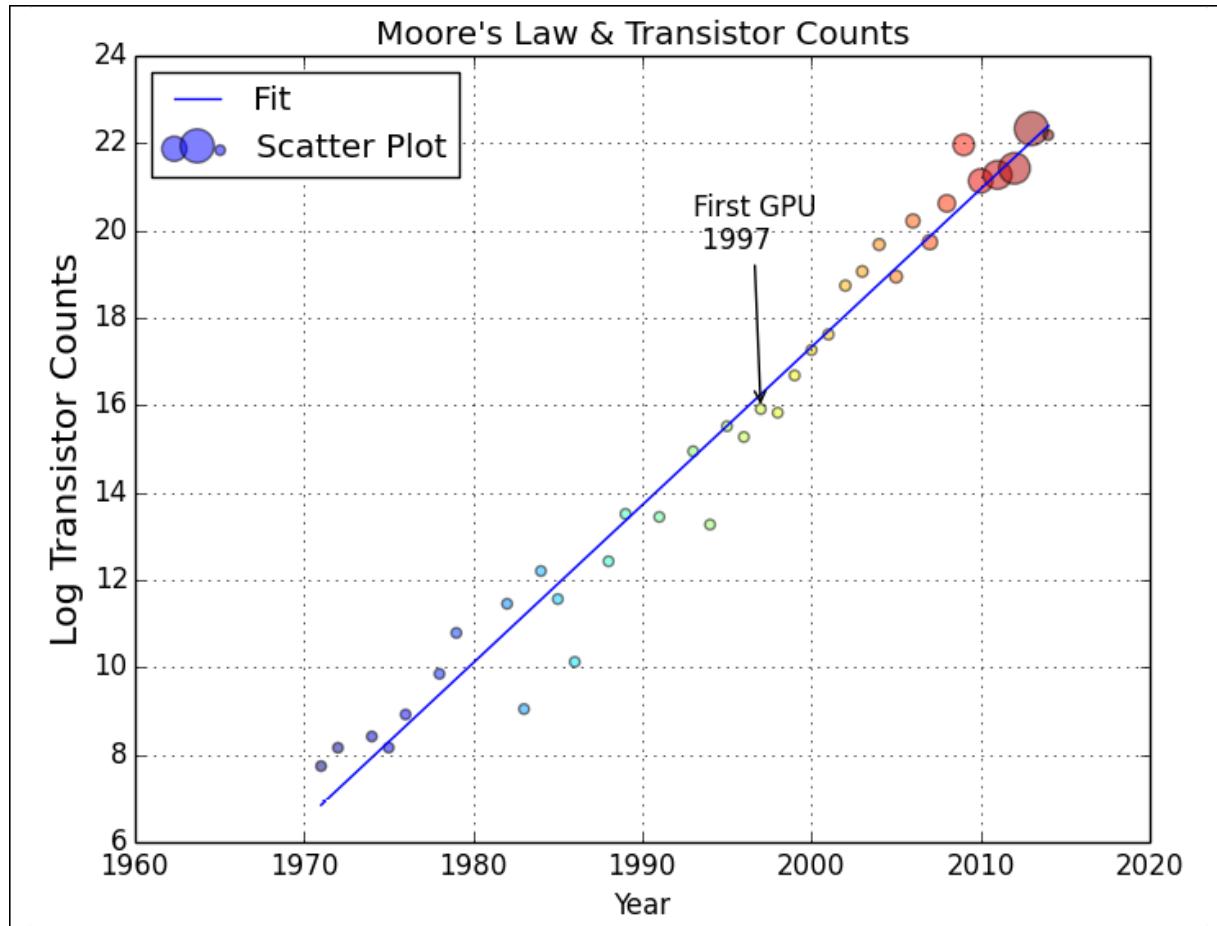


Figure 1.5: Transistor counts over the past decades

The consensus seems to be that Moore's law should continue to be valid for a couple of decades. This gives some credibility to Ray Kurzweil's predictions of achieving true machine intelligence by 2029.

Digging into the core of machine learning

After discussing the categorization of machine learning algorithms, we are going to dig into the core of machine learning—generalizing with data, and different levels of generalization, as well as the approaches to attain the right level of generalization.

Generalizing with data

The good thing about data is that there's a lot of it in the world. The bad thing is that it's hard to process this data. The challenge stems from the diversity and noisiness of the data. We humans usually process data coming into our ears and eyes. These inputs are transformed into electrical or chemical signals. On a very basic level, computers and robots also work with electrical signals. These electrical signals are then translated into ones and zeros. However, we program in Python in this book and, on that level, normally we represent the data either as numbers, images, or texts. Actually, images and text aren't very convenient, so we need to transform images and text into numerical values.

Especially in the context of supervised learning, we have a scenario similar to studying for an exam. We have a set of practice questions and the actual exams. We should be able to answer exam questions without knowing the answers to them. This is called **generalization**—we learn something from our practice questions and, hopefully, are able to apply the knowledge to other similar questions. In machine learning, these practice questions are called **training sets** or **training samples**. This is where the machine learning models derive patterns from. And the actual exams are **testing sets** or **testing samples**. They are where the models eventually apply to. And learning effectiveness is measured by the compatibility of the learning models and the testing. Sometimes, between practice questions and actual exams, we have mock exams to assess how well we will do in actual exams and to aid revision. These mock exams are known as **validation sets** or **validation samples** in machine learning. They help us to verify how well the models will perform in a simulated setting, and then we fine-tune the models accordingly in order to achieve greater hits.

An old-fashioned programmer would talk to a business analyst or other expert, and then implement a tax rule that adds a certain value multiplied by another corresponding value, for instance. In a machine learning setting, we can give the computer a bunch of input and output examples; or, if we want to be more ambitious, we can feed the program the actual tax texts. We can let the machine consume the data and figure out the tax rule, just as an autonomous car doesn't need a lot of explicit human input.

In physics, we have almost the same situation. We want to know how the universe works and formulate laws in a mathematical language. Since we don't know the actual function, all we can do is measure the error produced and try to minimize it. In supervised learning tasks, we compare our results against the expected values. In unsupervised learning, we measure our success with related metrics. For instance, we want clusters of data to be well defined; the metrics could be how similar the data points within one cluster are, and how different the data points from two clusters are. In reinforcement learning, a program evaluates its moves, for example, using a predefined function in a chess game.

Aside from correct generalization with data, there can be two levels of generalization, overfitting and underfitting, which we will explore in the next section.

Overfitting, underfitting, and the bias-variance trade-off

Let's take a look at both levels in detail and also explore the bias-variance trade-off.

Overfitting

Reaching the right fit model is the goal of a machine learning task. What if the model overfits? **Overfitting** means a model fits the existing observations **too well** but fails to predict future new observations. Let's look at the following analogy.

If we go through many practice questions for an exam, we may start to find ways to answer questions that have nothing to do with the subject material. For instance, given only five practice questions, we might find that if there are two occurrences of *potatoes*, one of *tomato*, and three of *banana* in a question, the answer is always *A*, and if there is one occurrence of *potato*, three of *tomato*, and two of *banana* in a question, the answer is always *B*. We could then conclude that this is always true and apply such a theory

later on, even though the subject or answer may not be relevant to potatoes, tomatoes, or bananas. Or, even worse, we might memorize the answers to each question verbatim. We would then score highly on the practice questions, leading us to hope that the questions in the actual exams would be the same as the practice questions. However, in reality, we would score very low on the exam questions as it's rare that the exact same questions occur in exams.

The phenomenon of memorization can cause overfitting. This can occur when we're over extracting too much information from the training sets and making our model just work well with them, which is called **low bias** in machine learning. In case you need a quick recap of bias, here it is: **bias** is the difference between the average prediction and the true value. It is computed as follows:

$$Bias[\hat{y}] = E[\hat{y} - y]$$

Here, \hat{y} is the prediction. At the same time, however, overfitting won't help us to generalize to new data and derive true patterns from it. The model, as a result, will perform poorly on datasets that weren't seen before. We call this situation **high variance** in machine learning. Again, a quick recap of variance: *variance* measures the spread of the prediction, which is the variability of the prediction. It can be calculated as follows:

$$Variance = E[\hat{y}^2] - E[\hat{y}]^2$$

The following example demonstrates what a typical instance of overfitting looks like, where the regression curve tries to flawlessly accommodate all observed samples:

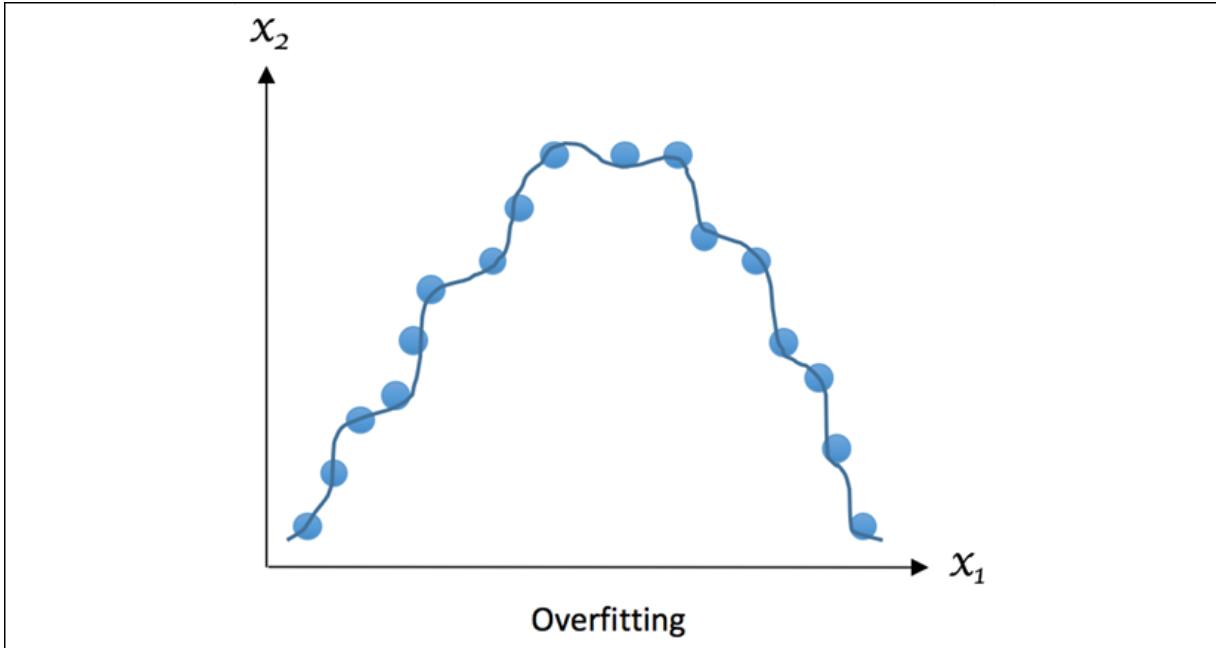


Figure 1.6: Example of overfitting

Overfitting occurs when we try to describe the learning rules based on too many parameters relative to the small number of observations, instead of the underlying relationship, such as the preceding example of potato and tomato, where we deduced three parameters from only five learning samples. Overfitting also takes place when we make the model excessively complex so that it fits every training sample, such as memorizing the answers for all questions, as mentioned previously.

Underfitting

The opposite scenario is **underfitting**. When a model is underfit, it doesn't perform well on the training sets and won't do so on the testing sets, which means it fails to capture the underlying trend of the data. Underfitting may occur if we aren't using enough data to train the model, just like we will fail the exam if we don't review enough material; this may also happen if we're trying to fit a wrong model to the data, just like we will score low in any exercises or exams if we take the wrong approach and learn it the wrong way. We call any of these situations a high **bias** in machine learning; although its variance is low as the performance in training and test sets is pretty consistent, in a bad way.

The following example shows what a typical underfitting looks like, where the regression curve doesn't fit the data well enough or capture enough of the underlying pattern of the data:

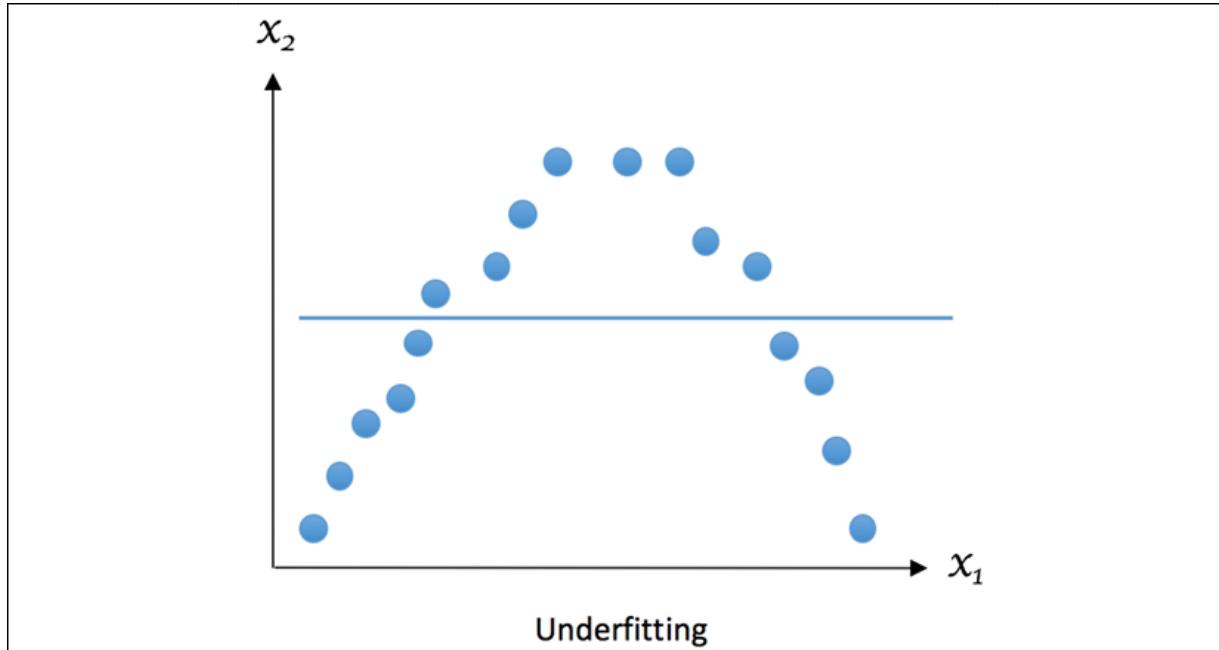


Figure 1.7: Example of underfitting

Now, let's look at what a well-fitting example should look like:

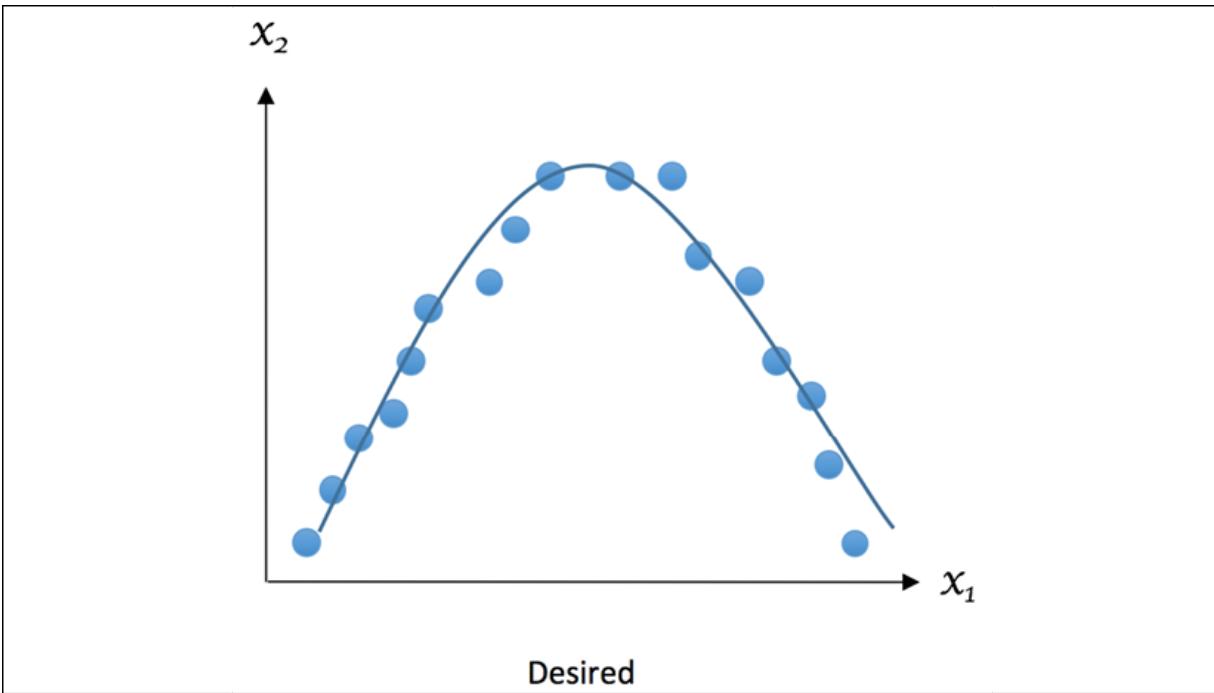


Figure 1.8: Example of desired fitting

The bias-variance trade-off

Obviously, we want to avoid both overfitting and underfitting. Recall that **bias** is the error stemming from incorrect assumptions in the learning algorithm; high bias results in underfitting. **Variance** measures how sensitive the model prediction is to variations in the datasets. Hence, we need to avoid cases where either bias or variance is getting high. So, does it mean we should always make both bias and variance as low as possible? The answer is yes, if we can. But, in practice, there is an explicit trade-off between them, where decreasing one increases the other. This is the so-called **bias-variance trade-off**. Sounds abstract? Let's take a look at the next example.

Let's say we're asked to build a model to predict the probability of a candidate being the next president in America based on phone poll data. The poll is conducted using zip codes. We randomly choose samples from one zip code and we estimate there's a 61% chance the candidate will win. However, it turns out he loses the election. Where did our model go wrong? The first thing we think of is the small size of samples from only one zip

code. It's a source of high bias also, because people in a geographic area tend to share similar demographics, although it results in a low variance of estimates. So, can we fix it simply by using samples from a large number of zip codes? Yes, but don't get happy so early. This might cause an increased variance of estimates at the same time. We need to find the optimal sample size—the best number of zip codes to achieve the lowest overall bias and variance.

Minimizing the total error of a model requires a careful balancing of bias and variance. Given a set of training samples, x_1, x_2, \dots, x_n , and their targets, y_1, y_2, \dots, y_n , we want to find a regression function $\hat{y}(x)$ that estimates the true relation $y(x)$ as correctly as possible. We measure the error of estimation, how good (or bad) the regression model is, in **mean squared error (MSE)**:

$$MSE = E[(y(x) - \hat{y}(x))^2]$$

The E denotes the expectation. This error can be decomposed into bias and variance components following the analytical derivation, as shown in the following formula (although it requires a bit of basic probability theory to understand):

$$\begin{aligned} MSE &= E[(y - \hat{y})^2] \\ &= E[(y - E[\hat{y}] + E[\hat{y}] - \hat{y})^2] \\ &= E[(y - E[\hat{y}])^2] + E[(E[\hat{y}] - \hat{y})^2] + E[2(y - E[\hat{y}])(E[\hat{y}] - \hat{y})] \\ &= E[(y - E[\hat{y}])^2] + E[(E[\hat{y}] - \hat{y})^2] + 2(y - E[\hat{y}])(E[\hat{y}] - E[\hat{y}]) \\ &= (E[\hat{y} - y])^2 + E[\hat{y}^2] - E[\hat{y}]^2 \\ &= Bias[\hat{y}]^2 + Variance[\hat{y}] \end{aligned}$$

The *Bias* term measures the error of estimations and the *Variance* term describes how much the estimation, \hat{y} , moves around its mean, $E[\hat{y}]$. The more complex the learning model $\hat{y}(x)$ is, and the larger the size of the

training samples, the lower the bias will become. However, this will also create more shift to the model in order to better fit the increased data points. As a result, the variance will be lifted.

We usually employ the cross-validation technique as well as regularization and feature reduction to find the optimal model balancing bias and variance and to diminish overfitting. We will talk about these next.



You may ask why we only want to deal with overfitting: how about underfitting? This is because underfitting can be easily recognized: it occurs as long as the model doesn't work well on a training set. And we need to find a better model or tweak some parameters to better fit the data, which is a must under all circumstances. On the other hand, overfitting is hard to spot. Oftentimes, when we achieve a model that performs well on a training set, we are overly happy and think it ready for production right away. This can be very dangerous. We should instead take extra steps to ensure that the great performance isn't due to overfitting and the great performance applies to data excluding the training data.

Avoiding overfitting with cross-validation

As a gentle reminder, you will see cross-validation in action multiple times later in this book. So don't panic if you ever find this section difficult to understand as you will become an expert of it very soon.

Recall that between practice questions and actual exams, there are mock exams where we can assess how well we will perform in actual exams and use that information to conduct necessary revision. In machine learning, the validation procedure helps to evaluate how the models will generalize to independent or unseen datasets in a simulated setting. In a conventional validation setting, the original data is partitioned into three subsets, usually 60% for the training set, 20% for the validation set, and the rest (20%) for the testing set. This setting suffices if we have enough training samples after partitioning and we only need a rough estimate of simulated performance. Otherwise, cross-validation is preferable.

In one round of cross-validation, the original data is divided into two subsets, for **training** and **testing** (or **validation**), respectively. The testing performance is recorded. Similarly, multiple rounds of cross-validation are performed under different partitions. Testing results from all rounds are finally averaged to generate a more reliable estimate of model prediction performance. Cross-validation helps to reduce variability and, therefore, limit overfitting.



When the training size is very large, it's often sufficient to split it into training, validation, and testing (three subsets) and conduct a performance check on the latter two. Cross-validation is less preferable in this case since it's computationally costly to train a model for each single round. But if you can afford it, there's no reason not to use cross-validation. When the size isn't so large, cross-validation is definitely a good choice.

There are mainly two cross-validation schemes in use: exhaustive and non-exhaustive. In the **exhaustive scheme**, we leave out a fixed number of observations in each round as testing (or validation) samples and use the remaining observations as training samples. This process is repeated until all possible different subsets of samples are used for testing once. For instance, we can apply **Leave-One-Out-Cross-Validation (LOOCV)**, which lets each sample be in the testing set once. For a dataset of the size n , LOOCV requires n rounds of cross-validation. This can be slow when n gets large. This following diagram presents the workflow of LOOCV:

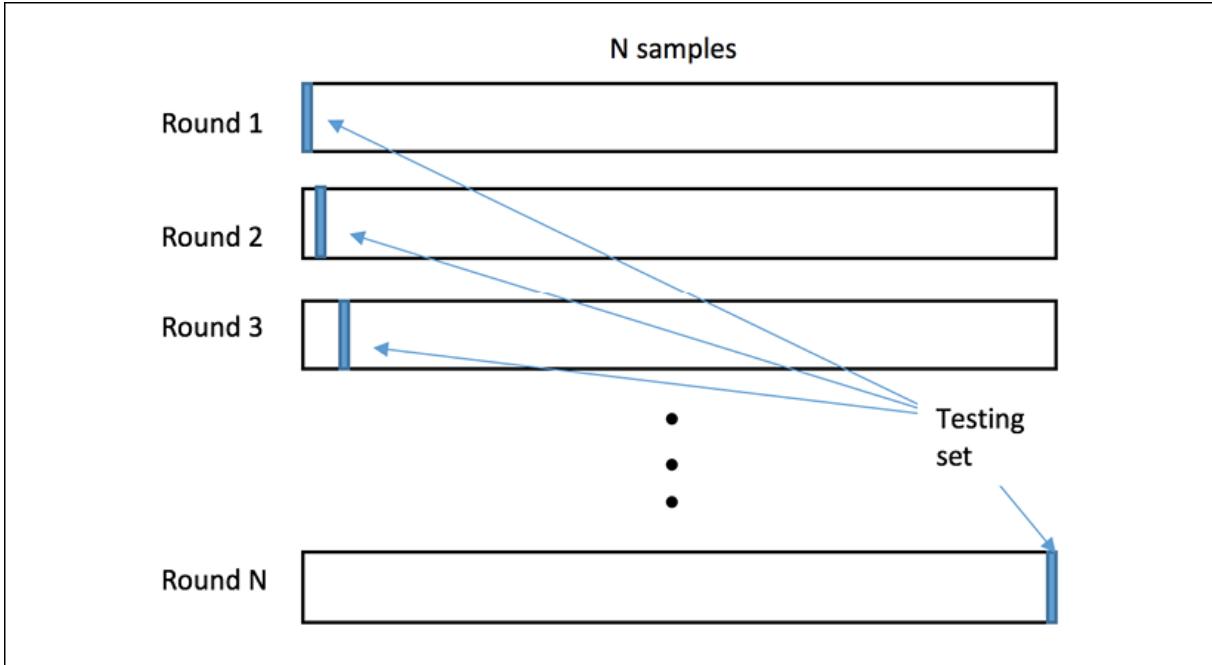


Figure 1.9: Workflow of leave-one-out-cross-validation

A **non-exhaustive scheme**, on the other hand, as the name implies, doesn't try out all possible partitions. The most widely used type of this scheme is **k-fold cross-validation**. We first randomly split the original data into **k equal-sized** folds. In each trial, one of these folds becomes the testing set, and the rest of the data becomes the training set.

We repeat this process k times, with each fold being the designated testing set once. Finally, we average the k sets of test results for the purpose of evaluation. Common values for k are 3, 5, and 10. The following table illustrates the setup for five-fold:

Round	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5
1	Testing	Training	Training	Training	Training
2	Training	Testing	Training	Training	Training
3	Training	Training	Testing	Training	Training

4	Training	Training	Training	Testing	Training
5	Training	Training	Training	Training	Testing

Table 1.1: Setup for 5-fold cross-validation

K-fold cross-validation often has a lower variance compared to LOOCV, since we're using a chunk of samples instead of a single one for validation.

We can also randomly split the data into training and testing sets numerous times. This is formally called the **holdout** method. The problem with this algorithm is that some samples may never end up in the testing set, while some may be selected multiple times in the testing set.

Last but not the least, **nested cross-validation** is a combination of cross-validations. It consists of the following two phases:

- **Inner cross-validation:** This phase is conducted to find the best fit and can be implemented as a k -fold cross-validation
- **Outer cross-validation:** This phase is used for performance evaluation and statistical analysis

We will apply cross-validation very intensively throughout this entire book. Before that, let's look at cross-validation with an analogy next, which will help us to better understand it.

A data scientist plans to take his car to work and his goal is to arrive before 9 a.m. every day. He needs to decide the departure time and the route to take. He tries out different combinations of these two parameters on certain Mondays, Tuesdays, and Wednesdays and records the arrival time for each trial. He then figures out the best schedule and applies it every day. However, it doesn't work quite as well as expected.

It turns out the scheduling **model** is overfit with data points gathered in the first three days and may not work well on Thursdays and Fridays. A better solution would be to test the best combination of parameters derived from Mondays to Wednesdays on Thursdays and Fridays and similarly repeat this

process based on different sets of learning days and testing days of the week. This analogized cross-validation ensures that the selected schedule works for the whole week.

In summary, cross-validation derives a more accurate assessment of model performance by combining measures of prediction performance on different subsets of data. This technique not only reduces variance and avoids overfitting, but also gives an insight into how the model will generally perform in practice.

Avoiding overfitting with regularization

Another way of preventing overfitting is **regularization**. Recall that the unnecessary complexity of the model is a source of overfitting.

Regularization adds extra parameters to the error function we're trying to minimize, in order to penalize complex models.

According to the principle of Occam's razor, simpler methods are to be favored. William Occam was a monk and philosopher who, around the year 1320, came up with the idea that the simplest hypothesis that fits data should be preferred. One justification is that we can invent fewer simple models than complex models. For instance, intuitively, we know that there are more high-polynomial models than linear ones. The reason is that a line ($y = ax + b$) is governed by only two parameters—the intercept, b , and slope, a . The possible coefficients for a line span two-dimensional space. A quadratic polynomial adds an extra coefficient for the quadratic term, and we can span a three-dimensional space with the coefficients. Therefore, it is much easier to find a model that perfectly captures all training data points with a **high-order polynomial function**, as its search space is much larger than that of a linear function. However, these easily obtained models generalize worse than linear models, which are more prone to overfitting. And, of course, simpler models require less computation time. The following diagram displays how we try to fit a linear function and a high order polynomial function, respectively, to the data:

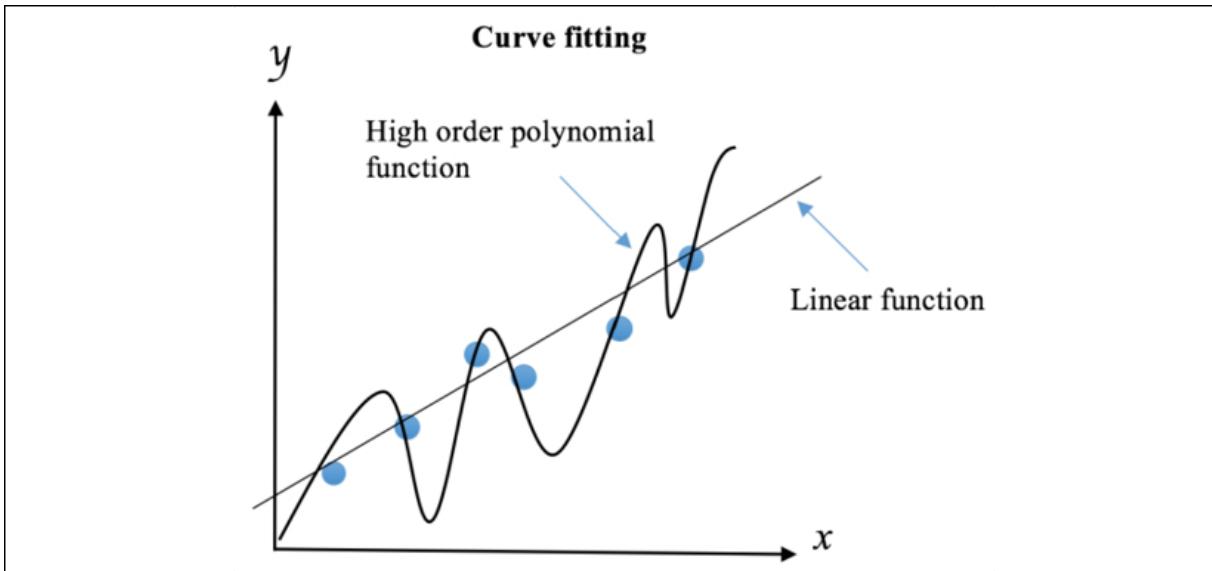


Figure 1.10: Fitting data with a linear function and a polynomial function

The linear model is preferable as it may generalize better to more data points drawn from the underlying distribution. We can use regularization to reduce the influence of the high orders of polynomial by imposing penalties on them. This will discourage complexity, even though a less accurate and less strict rule is learned from the training data.

We will employ regularization quite often starting from *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*. For now, let's look at an analogy that can help you better understand regularization.

A data scientist wants to equip his robotic guard dog with the ability to identify strangers and his friends. He feeds it with the following learning samples:

Male	Young	Tall	With glasses	In grey	Friend
Female	Middle	Average	Without glasses	In black	Stranger
Male	Young	Short	With glasses	In white	Friend
Male	Senior	Short	Without glasses	In black	Stranger

Female	Young	Average	With glasses	In white	Friend
Male	Young	Short	Without glasses	In red	Friend

Table 1.2: Training samples for the robotic guard dog

The robot may quickly learn the following rules:

- Any middle-aged female of average height without glasses and dressed in black is a stranger
- Any senior short male without glasses and dressed in black is a stranger
- Anyone else is his friend

Although these perfectly fit the training data, they seem too complicated and unlikely to generalize well to new visitors. In contrast, the data scientist limits the learning aspects. A loose rule that can work well for hundreds of other visitors could be as follows: anyone without glasses dressed in black is a stranger.

Besides penalizing complexity, we can also stop a training procedure early as a form of regularization. If we limit the time a model spends learning or we set some internal stopping criteria, it's more likely to produce a simpler model. The model complexity will be controlled in this way and, hence, overfitting becomes less probable. This approach is called **early stopping** in machine learning.

Last but not least, it's worth noting that regularization should be kept at a moderate level or, to be more precise, fine-tuned to an optimal level. Too small a regularization doesn't make any impact; too large a regularization will result in underfitting, as it moves the model away from the ground truth. We will explore how to achieve optimal regularization in *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*, *Chapter 7, Stock Prices Prediction with Regression Algorithms*, and *Chapter 8, Predicting Stock Prices with Artificial Neural Networks*.

Avoiding overfitting with feature selection and dimensionality reduction

We typically represent data as a grid of numbers (a **matrix**). Each column represents a variable, which we call a **feature** in machine learning. In supervised learning, one of the variables is actually not a feature, but the label that we're trying to predict. And in supervised learning, each row is an example that we can use for training or testing.

The number of features corresponds to the dimensionality of the data. Our machine learning approach depends on the number of dimensions versus the number of examples. For instance, text and image data are very high dimensional, while stock market data has relatively fewer dimensions.

Fitting high-dimensional data is computationally expensive and is prone to overfitting due to the high complexity. Higher dimensions are also impossible to visualize, and therefore we can't use simple diagnostic methods.

Not all of the features are useful and they may only add randomness to our results. It's therefore often important to do good feature selection. **Feature selection** is the process of picking a subset of significant features for use in better model construction. In practice, not every feature in a dataset carries information useful for discriminating samples; some features are either redundant or irrelevant, and hence can be discarded with little loss.

In principle, feature selection boils down to multiple binary decisions about whether to include a feature. For n features, we get 2^n feature sets, which can be a very large number for a large number of features. For example, for 10 features, we have 1,024 possible feature sets (for instance, if we're deciding what clothes to wear, the features can be temperature, rain, the weather forecast, and where we're going). Basically, we have two options: we either start with all of the features and remove features iteratively, or we start with a minimum set of features and add features iteratively. We then take the best feature sets for each iteration and compare them. At a certain point, brute-force evaluation becomes infeasible. Hence, more advanced feature selection algorithms were invented to distill the most useful

features/signals. We will discuss in detail how to perform feature selection in *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*.

Another common approach of reducing dimensionality is to transform high-dimensional data into lower-dimensional space. This is known as **dimensionality reduction** or **feature projection**. We will get into this in detail in *Chapter 9, Mining the 20 Newsgroups Dataset with Text Analysis Techniques*, *Chapter 10, Discovering Underlying Topics in the Newsgroups Dataset with Clustering and Topic Modeling*, and *Chapter 11, Machine Learning Best Practices*.

In this section, we've talked about how the goal of machine learning is to find the optimal generalization to the data, and how to avoid ill-generalization. In the next two sections, we will explore tricks to get closer to the goal throughout individual phases of machine learning, including data preprocessing and feature engineering in the next section, and modeling in the section after that.

Data preprocessing and feature engineering

Data mining, a buzzword in the 1990s, is the predecessor of data science (the science of data). One of the methodologies popular in the data mining community is called the **Cross-Industry Standard Process for Data Mining (CRISP-DM)**

(https://en.wikipedia.org/wiki/Cross-industry_standard_process_for_data_mining). CRISP-DM was created in 1996, and machine learning basically inherits its phases and general framework.

CRISP-DM consists of the following phases, which aren't mutually exclusive and can occur in parallel:

- **Business understanding:** This phase is often taken care of by specialized domain experts. Usually, we have a businessperson formulate a business problem, such as selling more units of a certain product.
- **Data understanding:** This is also a phase that may require input from domain experts; however, often a technical specialist needs to get involved more than in the business understanding phase. The domain expert may be proficient with spreadsheet programs but have trouble with complicated data. In this machine learning book, it's usually termed the **exploration phase**.
- **Data preparation:** This is also a phase where a domain expert with only Microsoft Excel knowledge may not be able to help you. This is the phase where we create our training and test datasets. In this book, it's usually termed the **preprocessing phase**.
- **Modeling:** This is the phase most people associate with machine learning. In this phase, we formulate a model and fit our data.
- **Evaluation:** In this phase, we evaluate how well the model fits the data to check whether we were able to solve our business problem.
- **Deployment:** This phase usually involves setting up the system in a production environment (it's considered good practice to have a separate production system). Typically, this is done by a specialized team.

We will cover the preprocessing phase first in this section.

Preprocessing and exploration

When we learn, we require high-quality learning material. We can't learn from gibberish, so we automatically ignore anything that doesn't make sense. A machine learning system isn't able to recognize gibberish, so we need to help it by cleaning the input data. It's often claimed that cleaning the data forms a large part of machine learning. Sometimes, cleaning is already done for us, but you shouldn't count on it.

To decide how to clean the data, we need to be familiar with the data. There are some projects that try to automatically explore the data and do something intelligent, such as produce a report. For now, unfortunately, we don't have a solid solution in general, so you need to do some work.

We can do two things, which aren't mutually exclusive: first, scan the data and second, visualize the data. This also depends on the type of data we're dealing with—whether we have a grid of numbers, images, audio, text, or something else.

In the end, a grid of numbers is the most convenient form, and we will always work toward having numerical features. Let's pretend that we have a table of numbers in the rest of this section.

We want to know whether features have missing values, how the values are distributed, and what type of features we have. Values can approximately follow a normal distribution, a binomial distribution, a Poisson distribution, or another distribution altogether. Features can be binary: either yes or no, positive or negative, and so on. They can also be categorical: pertaining to a category, for instance, continents (Africa, Asia, Europe, South America, North America, and so on). Categorical variables can also be ordered, for instance, high, medium, and low. Features can also be quantitative, for example, the temperature in degrees or the price in dollars. Now, let me get into how we can cope with each of these situations.

Dealing with missing values

Quite often we miss values for certain features. This could happen for various reasons. It can be inconvenient, expensive, or even impossible to always have a value. Maybe we weren't able to measure a certain quantity in the past because we didn't have the right equipment or just didn't know that the feature was relevant. However, we're stuck with missing values from the past.

Sometimes, it's easy to figure out that we're missing values and we can discover this just by scanning the data or counting the number of values we

have for a feature and comparing this figure with the number of values we expect based on the number of rows. Certain systems encode missing values with, for example, values such as 999,999 or -1. This makes sense if the valid values are much smaller than 999,999. If you're lucky, you'll have information about the features provided by whoever created the data in the form of a data dictionary or metadata.

Once we know that we're missing values, the question arises of how to deal with them. The simplest answer is to just ignore them. However, some algorithms can't deal with missing values, and the program will just refuse to continue. In other circumstances, ignoring missing values will lead to inaccurate results. The second solution is to substitute missing values with a fixed value—this is called **imputing**. We can impute the arithmetic **mean**, **median**, or **mode** of the valid values of a certain feature. Ideally, we will have some prior knowledge of a variable that is somewhat reliable. For instance, we may know the seasonal averages of temperature for a certain location and be able to impute guesses for missing temperature values given a date. We will talk about dealing with missing data in detail in *Chapter 11, Machine Learning Best Practices*. Similarly, techniques in the following sections will be discussed and employed in later chapters, in case you feel lost.

Label encoding

Humans are able to deal with various types of values. Machine learning algorithms (with some exceptions) require numerical values. If we offer a string such as `Ivan`, unless we're using specialized software, the program won't know what to do. In this example, we're dealing with a categorical feature—names, probably. We can consider each unique value to be a label. (In this particular example, we also need to decide what to do with the case—is `Ivan` the same as `ivan`?). We can then replace each label with an integer—**label encoding**.

The following example shows how label encoding works:

Label	Encoded Label
-------	---------------

Africa	1
Asia	2
Europe	3
South America	4
North America	5
Other	6

Table 1.3: Example of label encoding

This approach can be problematic in some cases, because the learner may conclude that there is an order (unless it is expected, for example, $bad=0$, $ok=1$, $good=2$, $excellent=3$). In the preceding mapping table, `Asia` and `North America` in the preceding case differ by `4` after encoding, which is a bit counter-intuitive as it's hard to quantify them. One-hot encoding in the next section takes an alternative approach.

One-hot encoding

The **one-of-K**, or **one-hot encoding**, scheme uses dummy variables to encode categorical features. Originally, it was applied to digital circuits. The dummy variables have binary values such as bits, so they take the values zero or one (equivalent to true or false). For instance, if we want to encode continents, we will have dummy variables, such as `is_asia`, which will be true if the continent is `Asia` and false otherwise. In general, we need as many dummy variables as there are unique labels minus one. We can determine one of the labels automatically from the dummy variables, because the dummy variables are exclusive.

If the dummy variables all have a false value, then the correct label is the label for which we don't have a dummy variable. The following table illustrates the encoding for continents:

Label	Is_africa	Is_asia	Is_europe	Is_sam	Is_nam
Africa	1	0	0	0	0
Asia	0	1	0	0	0
Europe	0	0	1	0	0
South America	0	0	0	1	0
North America	0	0	0	0	1
Other	0	0	0	0	0

Table 1.4: Example of one-hot encoding

The encoding produces a matrix (grid of numbers) with lots of zeros (false values) and occasional ones (true values). This type of matrix is called a **sparse matrix**. The sparse matrix representation is handled well by the `scipy` package and shouldn't be an issue. We will discuss the `scipy` package later in this chapter.

Scaling

Values of different features can differ by orders of magnitude. Sometimes, this may mean that the larger values dominate the smaller values. This depends on the algorithm we're using. For certain algorithms to work properly, we're required to scale the data.

There are the following several common strategies that we can apply:

- Standardization removes the mean of a feature and divides by the standard deviation. If the feature values are normally distributed, we will get a **Gaussian**, which is centered around zero with a variance of one.
- If the feature values aren't normally distributed, we can remove the median and divide by the interquartile range.
The **interquartile range** is the range between the first and third quartile (or 25th and 75th percentile).
- Scaling features to a range is a common choice of range between zero and one.

We will use this method in many projects throughout the book.

An advanced version of data preprocessing is usually called feature engineering. We will cover that next.

Feature engineering

Feature engineering is the process of creating or improving features. It is more of a dark art than a science. Features are often created based on common sense, domain knowledge, or prior experience. There are certain common techniques for feature creation; however, there is no guarantee that creating new features will improve your results. We are sometimes able to use the clusters found by unsupervised learning as extra features. **Deep neural networks** are often able to derive features **automatically**.

We will briefly look at several techniques such as polynomial features, power transformations, and binning.

Polynomial transformation

If we have two features, a and b , we can suspect that there is a polynomial relationship, such as $a^2 + ab + b^2$. We can consider each term in the sum to

be a feature—in the previous example, we have three features, which are a , b , and $a^2 + ab + b^2$. The product ab in the middle is called an **interaction**. An interaction doesn't have to be a product—although this is the most common choice—it can also be a sum, a difference, or a ratio. If we're using a ratio to avoid dividing by zero, we should add a small constant to the divisor and dividend.

The number of features and the order of the polynomial for a polynomial relation aren't limited. However, if we follow Occam's razor, we should avoid higher-order polynomials and interactions of many features. In practice, complex polynomial relations tend to be more difficult to compute and tend to overfit, but if you really need better results, they may be worth considering. We will see polynomial transformation in action in the *Best practice 12 – performing feature engineering without domain expertise* section in *Chapter 11, Machine Learning Best Practices*.

Power transforms

Power transforms are functions that we can use to transform numerical features in order to conform better to a normal distribution. A very common transformation for values that vary by orders of magnitude is to take the **logarithm**.

Taking the logarithm of a zero value and negative values isn't defined, so we may need to add a constant to all of the values of the related feature before taking the logarithm. We can also take the square root for positive values, square the values, or compute any other power we like.

Another useful power transform is the **Box-Cox transformation**, named after its creators, two statisticians called George Box and Sir David Roxbee Cox. The Box-Cox transformation attempts to find the best power needed to transform the original data into data that's closer to the normal distribution. In case you are interested, the transform is defined as follows:

$$y_i^{(\lambda)} = \begin{cases} \frac{y_i^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(y_i) & \text{if } \lambda = 0, \end{cases}$$

Binning

Sometimes, it's useful to separate feature values into several bins. For example, we may only be interested in whether it rained on a particular day. Given the precipitation values, we can binarize the values, so that we get a true value if the precipitation value isn't zero, and a false value otherwise. We can also use statistics to divide values into high, low, and medium bins. In marketing, we often care more about the age group, such as 18 to 24, than a specific age, such as 23.

The binning process inevitably leads to loss of information. However, depending on your goals, this may not be an issue, and actually reduces the chance of overfitting. Certainly, there will be improvements in speed and reduction of memory or storage requirements and redundancy.

Any real-world machine learning system should have two modules: a data preprocessing module, which we just covered in this section, and a modeling module, which will be covered next.

Combining models

A model takes in data (usually preprocessed) and produces predictive results. What if we employ multiple models; will we make better decisions by combining predictions from individual models? We will talk about this in this section.

Let's start with an analogy. In high school, we sit together with other students and learn together, but we aren't supposed to work together during the exam. The reason is, of course, that teachers want to know what we've

learned, and if we just copy exam answers from friends, we may not have learned anything. Later in life, we discover that teamwork is important. For example, this book is the product of a whole team, or possibly a group of teams.

Clearly, a team can produce better results than a single person. However, this goes against Occam's razor, since a single person can come up with simpler theories compared to what a team will produce. In machine learning, we nevertheless prefer to have our models cooperate with the following schemes:

- Voting and averaging
- Bagging
- Boosting
- Stacking

Let's get into each of them now.

Voting and averaging

This is probably the most understandable type of model aggregation. It just means the final output will be the **majority** or **average** of prediction output values from multiple models. It is also possible to assign different weights to individual models in the ensemble, for example, some models that are more reliable might be given two votes.

Nonetheless, combining the results of models that are highly correlated to each other doesn't guarantee a spectacular improvement. It is better to somehow diversify the models by using different features or different algorithms. If you find two models are strongly correlated, you may, for example, decide to remove one of them from the ensemble and increase proportionally the weight of the other model.

Bagging

Bootstrap aggregating, or **bagging**, is an algorithm introduced by Leo Breiman, a distinguished statistician at the University of California, Berkeley, in 1994, which applies **bootstrapping** to machine learning problems. Bootstrapping is a statistical procedure that creates multiple datasets from the existing one by sampling data with replacement. Bootstrapping can be used to measure the properties of a model, such as bias and variance.

In general, a bagging algorithm follows these steps:

1. We generate new training sets from input training data by sampling with replacement
2. For each generated training set, we fit a new model
3. We combine the results of the models by averaging or majority voting

The following diagram illustrates the steps for bagging, using classification as an example (the circles and crosses represent samples from two classes):

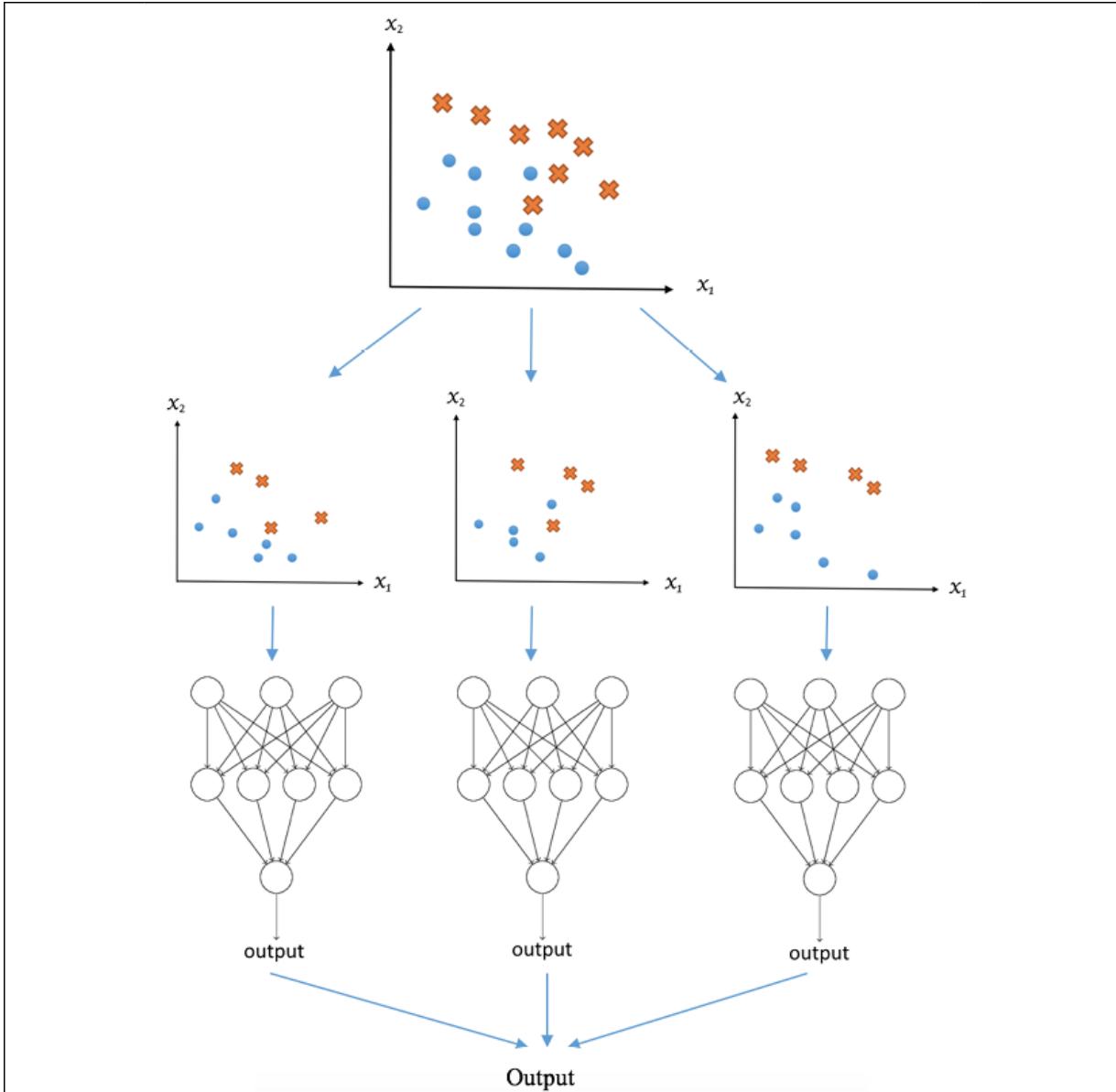


Figure 1.11: Workflow of bagging for classification

As you can imagine, bagging can reduce the chance of overfitting.

We will study bagging in depth in *Chapter 4, Predicting Online Ad Click-Through with Tree-Based Algorithms*.

Boosting

In the context of supervised learning, we define **weak learners** as learners who are just a little better than a baseline, such as randomly assigning classes or average values. Much like ants, weak learners are weak individually, but together they have the power to do amazing things.

It makes sense to take into account the strength of each individual learner using weights. This general idea is called **boosting**. In boosting, all models are trained in sequence, instead of in parallel as in bagging. Each model is trained on the same dataset, but each data sample is under a different weight factoring in the previous model's success. The weights are reassigned after a model is trained, which will be used for the next training round. In general, weights for mispredicted samples are increased to stress their prediction difficulty.

The following diagram illustrates the steps for boosting, again using classification as an example (the circles and crosses represent samples from two classes, and the size of a circle or cross indicates the weight assigned to it):

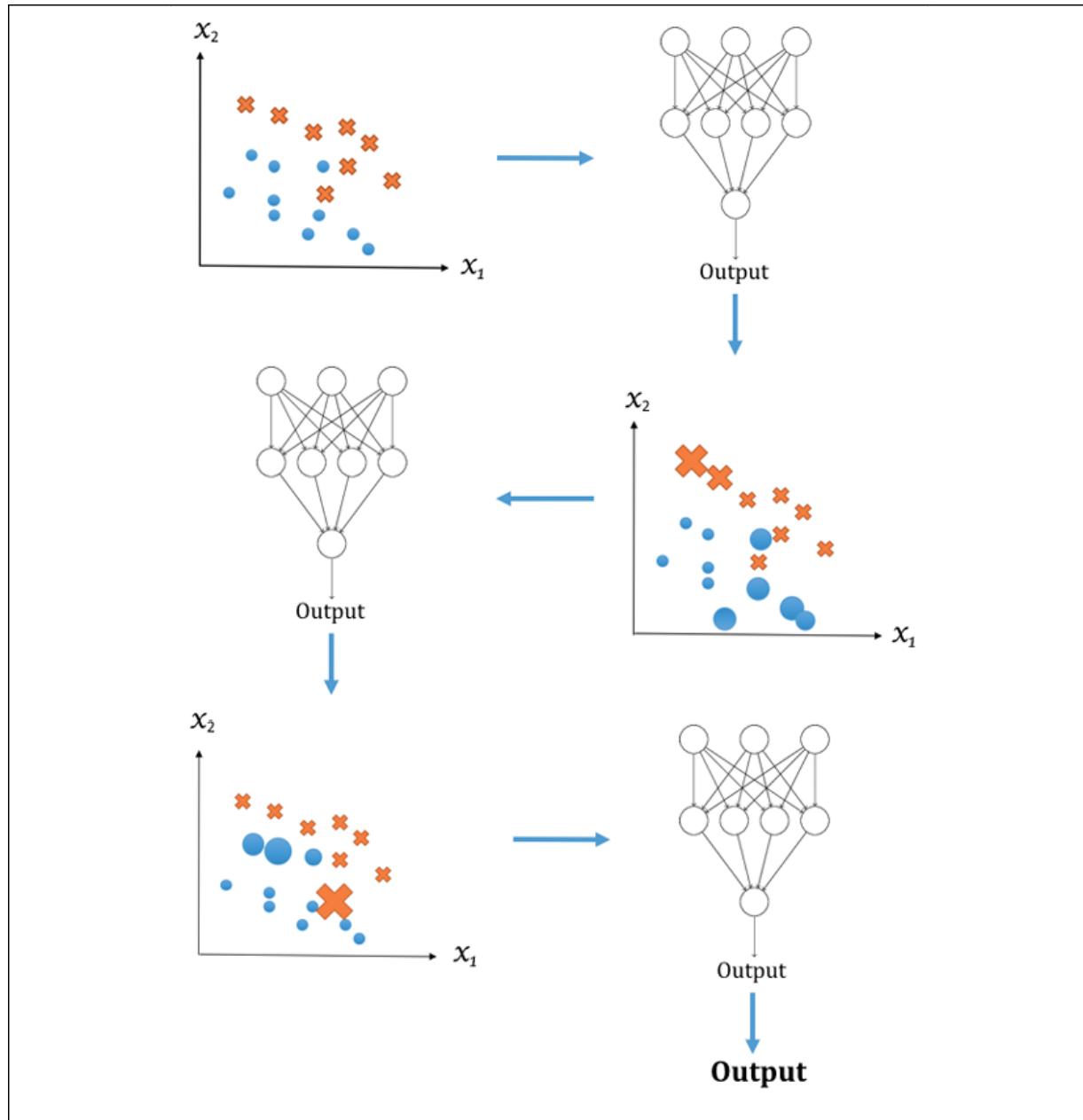


Figure 1.12: Workflow of boosting for classification

There are many boosting algorithms; boosting algorithms differ mostly in their weighting scheme. If you've studied for an exam, you may have applied a similar technique by identifying the type of practice questions you had trouble with and focusing on the hard problems.

Face detection in images is based on a specialized framework that also uses boosting. Detecting faces in images or videos is supervised learning. We

give the learner examples of regions containing faces. There's an imbalance, since we usually have far more regions (about 10,000 times more) that don't have faces.

A cascade of classifiers progressively filters out negative image areas stage by stage. In each progressive stage, the classifiers use progressively more features on fewer image windows. The idea is to spend the most time on image patches that contain faces. In this context, boosting is used to select features and combine results.

Stacking

Stacking takes the output values of machine learning models and then uses them as input values for another algorithm. You can, of course, feed the output of the higher-level algorithm to another predictor. It's possible to use any arbitrary topology but, for practical reasons, you should try a simple setup first as also dictated by Occam's razor.

A fun fact is that stacking is commonly used in the winning models in the Kaggle competition. For instance, the first place for the Otto Group Product Classification challenge (www.kaggle.com/c/otto-group-product-classification-challenge) went to a stacking model composed of more than 30 different models.

So far, we have covered the tricks required to more easily reach the right generalization for a machine learning model throughout the data preprocessing and modeling phase. I know you can't wait to start working on a machine learning project. Let's get ready by setting up the working environment.

Installing software and setting up

As the book title says, Python is the language we will use to implement all machine learning algorithms and techniques throughout the entire book. We will also exploit many popular Python packages and tools such as NumPy, SciPy, TensorFlow, and scikit-learn. By the end of this kick-off chapter, make sure you set up the tools and working environment properly, even if you are already an expert in Python or might be familiar with some of those tools.

Setting up Python and environments

We will be using Python 3 in this book. As you may know, Python 2 will no longer be supported after 2020, so starting with or switching to Python 3 is strongly recommended. Trust me, the transition is pretty smooth. But if you're stuck with Python 2, you still should be able to modify the codes to work for you. The Anaconda Python 3 distribution is one of the best options for data science and machine learning practitioners.

Anaconda is a free Python distribution for data analysis and scientific computing. It has its own package manager, `conda`. The distribution (<https://docs.anaconda.com/anaconda/packages/pkg-docs/>, depending on your OS, or version 3.7, 3.6, or 2.7) includes more than 600 Python packages (as of 2020), which makes it very convenient. For casual users, the **Miniconda** (<https://conda.io/miniconda.html>) distribution may be the better choice. Miniconda contains the `conda` package manager and Python. Obviously, Miniconda takes much less disk space than Anaconda.

The procedures to install Anaconda and Miniconda are similar. You can follow the instructions from <https://docs.conda.io/projects/conda/en/latest/user-guide/install/>. First, you have to download the appropriate installer for your OS and Python version, as follows:

Regular installation

Follow the instructions for your operating system:

- [Windows](#).
- [macOS](#).
- [Linux](#).

Figure 1.13: Installation entry based on your OS

Follow the steps listed in your OS. You can choose between a GUI and a CLI. I personally find the latter easier.

I was able to use the Python 3 installer, although the Python version in my system was 2.7 at the time I installed it. This is possible since Anaconda comes with its own Python. On my machine, the `Anaconda` installer created an `anaconda` directory in my home directory and required about 900 MB. Similarly, the `Miniconda` installer installs a `miniconda` directory in your home directory.

Feel free to play around with it after you set it up. One way to verify that you have set up Anaconda properly is by entering the following command line in your terminal on Linux/Mac or Command Prompt on Windows (from now on, we will just mention terminal):

```
python
```

The preceding command line will display your Python running environment, as shown in the following screenshot:

```
Python 3.7.2 (default, Dec 29 2018, 00:00:04)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda custom (64-bit) on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

Figure 1.14: Screenshot after running "python" in the terminal

If this isn't what you're seeing, please check the system path or the path Python is running from.

At the end of this section, I want to emphasize the reasons why Python is the most popular language for machine learning and data science. First of all, Python is famous for its high readability and simplicity, which makes it easy to build machine learning models. We spend less time in worrying about getting the right syntax and compilation and, as a result, have more time to find the right machine learning solution. Second, we have an extensive selection of Python libraries and frameworks for machine learning:

Data analysis	NumPy, SciPy, pandas
Data visualization	Matplotlib, Seaborn
Modeling	scikit-learn, TensorFlow, Keras

Table 1.5: Popular Python libraries for machine learning

The next step involves setting up some of these packages that we will use throughout this book.

Installing the main Python packages

For most projects in this book, we will be using NumPy (<http://www.numpy.org/>), scikit-learn (<http://scikit-learn.org/stable/>), and TensorFlow (<https://www.tensorflow.org/>). In the sections that follow, we will cover the installation of several Python packages that we will be mainly using in this book.

NumPy

NumPy is the fundamental package for machine learning with Python. It offers powerful tools including the following:

- The N -dimensional array `ndarray` class and several subclasses representing matrices and arrays
- Various sophisticated array functions
- Useful linear algebra capabilities

Installation instructions for NumPy can be found at <http://docs.scipy.org/doc/numpy/user/install.html>. Alternatively, an easier method involves installing it with `pip` in the command line as follows:

```
pip install numpy
```

To install `conda` for Anaconda users, run the following command line:

```
conda install numpy
```

A quick way to verify your installation is to import it into the shell as follows:

```
>>> import numpy
```

It has installed correctly if no error message is visible.

SciPy

In machine learning, we mainly use NumPy arrays to store data vectors or matrices composed of feature vectors. SciPy (<https://www.scipy.org/scipylib/index.html>) uses NumPy arrays and offers a variety of scientific and mathematical functions. Installing `SciPy` in the terminal is similar, again as follows:

```
pip install scipy
```

Pandas

We also use the `pandas` library (<https://pandas.pydata.org/>) for data wrangling later in this book. The best way to get `pandas` is via `pip` or `conda`:

```
conda install pandas
```

Scikit-learn

The `scikit-learn` library is a Python machine learning package optimized for performance as a lot of the code runs almost as fast as equivalent C code. The same statement is true for NumPy and SciPy. Scikit-learn requires both NumPy and SciPy to be installed. As the installation guide in <http://scikit-learn.org/stable/install.html> states, the easiest way to install scikit-learn is to use `pip` or `conda` as follows:

```
pip install -U scikit-learn
```

TensorFlow

TensorFlow is a Python-friendly open source library invented by the Google Brain team for high-performance numerical computation. It makes machine learning faster and deep learning easier with the Python-based convenient frontend API and high-performance C++-based backend execution. Plus, it allows easy deployment of computation across CPUs and GPUs, which empowers expensive and large-scale machine learning. In this book, we will focus on CPU as our computation platform. Hence, according to <https://www.tensorflow.org/install/>, installing TensorFlow 2 is done via the following command line:

```
pip install tensorflow
```

There are many other packages we will be using intensively, for example, **Matplotlib** for plotting and visualization, **Seaborn** for visualization, **NLTK** for natural language processing, **PySpark** for large-scale machine learning, and **PyTorch** for reinforcement learning. We will provide installation details for any package when we first encounter it in this book.

Introducing TensorFlow 2

TensorFlow provides us with an end-to-end scalable platform for implementing and deploying machine learning algorithms. TensorFlow 2 was largely redesigned from its first mature version 1.0 and was released at the end of 2019.

TensorFlow has been widely known for its deep learning modules. However, its most powerful point is **computation graphs**, which algorithms are built on. Basically, a computation graph is used to convey relationships between the input and the output via tensors. For instance, if we want to evaluate a linear relationship, $y = 3 * a + 2 * b$, we can represent it in the following computation graph:

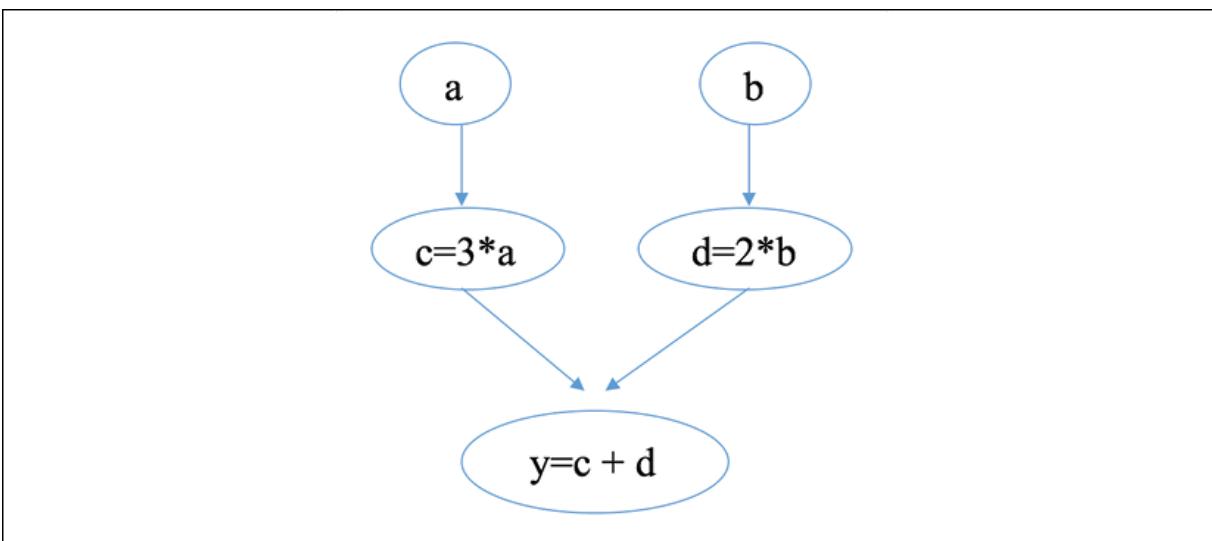


Figure 1.15: Computation graph for a $y = 3 * a + 2 * b$ machine

Here, `a` and `b` are the input tensors, `c` and `d` are the intermediate tensors, and `y` is the output.

You can think of a computation graph as a network of nodes connected by edges. Each node is a tensor and each edge is an operation or function that takes its input node and returns a value to its output node. To train a machine learning model, TensorFlow builds the computation graph and computes the **gradients** accordingly (gradients are vectors providing the steepest direction where an optimal solution is reached). In the upcoming chapters, you will see some examples of training machine learning models using `TensorFlow`.

At the end, we highly recommend you go through <https://www.tensorflow.org/guide/data> if you are interested in exploring more about TensorFlow and computation graphs.

Summary

We just finished our first mile on the Python and machine learning journey! Throughout this chapter, we became familiar with the basics of machine learning. We started with what machine learning is all about, the importance of machine learning (DT era) and its brief history, and looked at recent developments as well. We also learned typical machine learning tasks and explored several essential techniques of working with data and working with models. Now that we're equipped with basic machine learning knowledge and we've set up the software and tools, let's get ready for the real-world machine learning examples ahead.

In the next chapter, we will be building a movie recommendation engine as our first machine learning project!

Exercises

1. Can you tell the difference between machine learning and traditional programming (rule-based automation)?
2. What's overfitting and how do we avoid it?
3. Name two feature engineering approaches.
4. Name two ways to combine multiple models.
5. Install Matplotlib (<https://matplotlib.org/>) if this is of interest to you. We will use it for data visualization throughout the book.

2

Building a Movie Recommendation Engine with Naïve Bayes

As promised, in this chapter, we will kick off our supervised learning journey with machine learning classification, and specifically, binary classification. The goal of the chapter is to build a movie recommendation system. It is a good starting point to learn classification from a real-life example—movie streaming service providers are already doing this, and we can do the same. You will learn the fundamental concepts of classification, including what it does and its various types and applications, with a focus on solving a binary classification problem using a simple, yet powerful, algorithm, Naïve Bayes. Finally, the chapter will demonstrate how to fine-tune a model, which is an important skill that every data science or machine learning practitioner should learn.

We will go into detail on the following topics:

- What is machine learning classification?
- Types of classification
- Applications of text classification
- The Naïve Bayes classifier
- The mechanics of Naïve Bayes
- Naïve Bayes implementations
- Building a movie recommender with Naïve Bayes
- Classification performance evaluation

- Cross-validation
- Tuning a classification model

Getting started with classification

Movie recommendation can be framed as a machine learning classification problem. If it is predicted that you like a movie, for example, then it will be on your recommended list, otherwise, it won't. Let's get started by learning the important concepts of machine learning classification.

Classification is one of the main instances of supervised learning. Given a training set of data containing observations and their associated categorical outputs, the goal of classification is to learn a general rule that correctly maps the **observations** (also called **features** or **predictive variables**) to the target **categories** (also called **labels** or **classes**). Putting it another way, a trained classification model will be generated after the model learns from the features and targets of training samples, as shown in the first half of *Figure 2.1*. When new or unseen data comes in, the trained model will be able to determine their desired class memberships. Class information will be predicted based on the known input features using the trained classification model, as displayed in the second half of *Figure 2.1*:

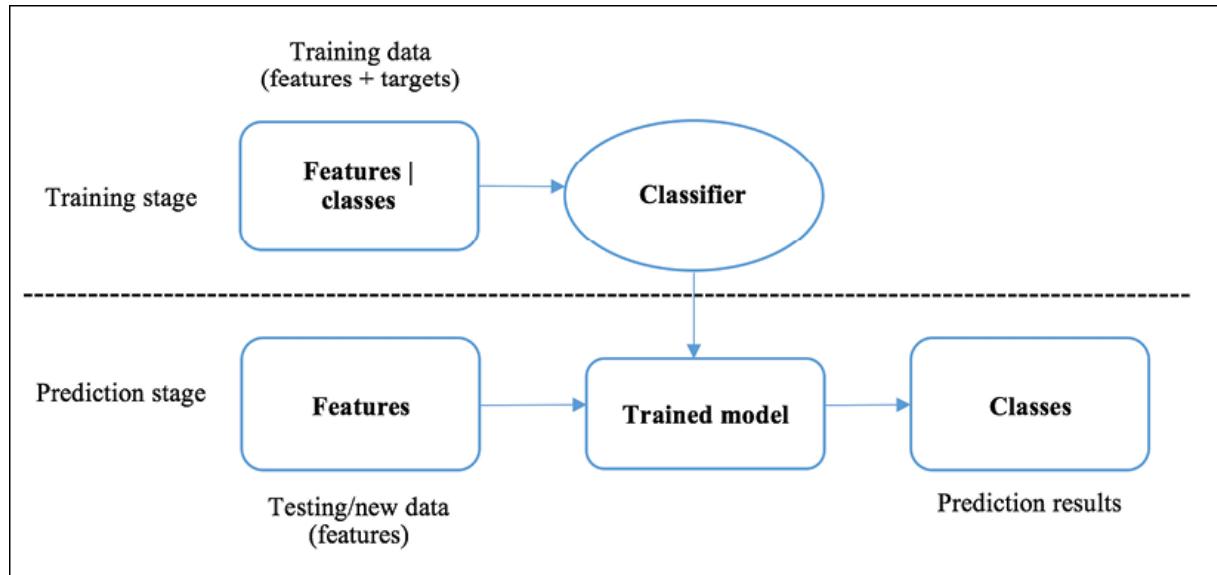


Figure 2.1: The training and prediction stages in classification

In general, there are three types of classification based on the possibility of class output—**binary**, **multiclass**, and **multi-label classification**. We will cover them one by one in the next section.

Binary classification

This classifies observations into one of two possible classes. The example of spam email filtering we encounter every day is a typical use case of binary classification, which identifies email messages (input observations) as spam or not spam (output classes). Customer churn prediction is another frequently mentioned example, where a prediction system takes in customer segment data and activity data from CRM systems and identifies which customers are likely to churn.

Another application in the marketing and advertising industry is click-through prediction for online ads—that is, whether or not an ad will be clicked, given users' cookie information and browsing history. Last but not least, binary classification has also been employed in biomedical science, for example, in early cancer diagnosis, classifying patients into high or low risk groups based on MRI images.

As demonstrated in *Figure 2.2*, binary classification tries to find a way to separate data from two classes (denoted by dots and crosses):

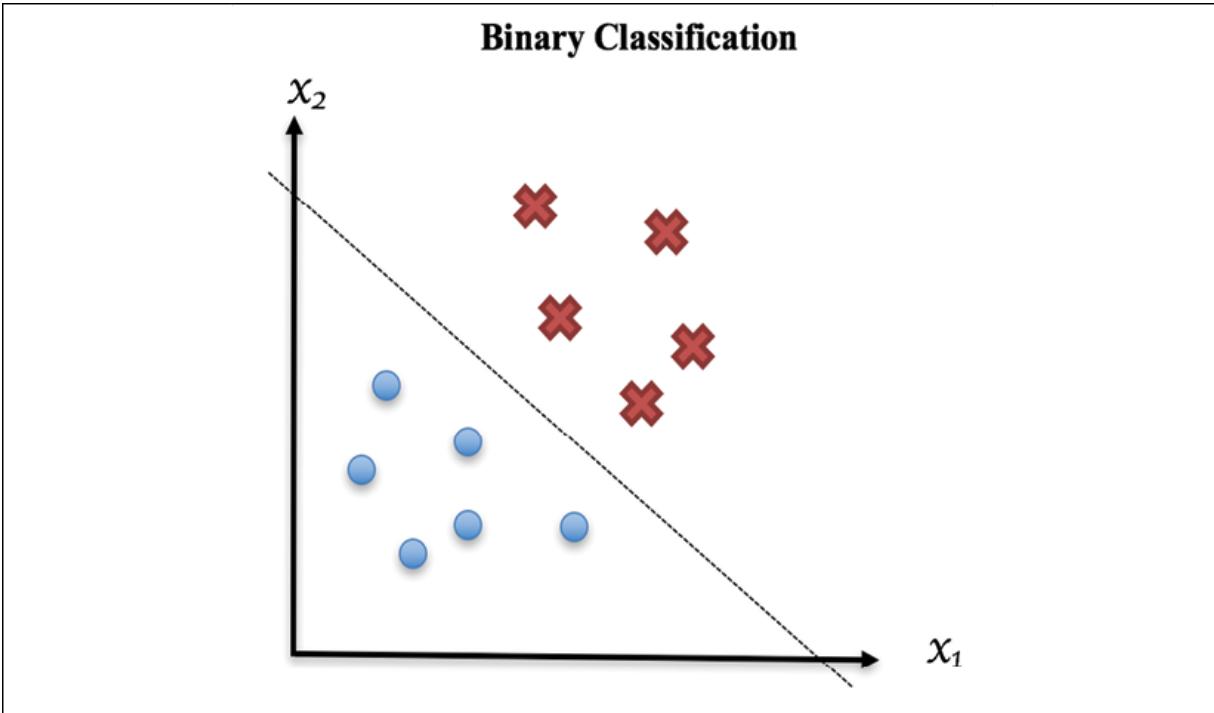


Figure 2.2: Binary classification example

Don't forget that predicting whether a person likes a movie is also a binary classification problem.

Multiclass classification

This type of classification is also referred to as **multinomial classification**. It allows more than two possible classes, as opposed to only two in binary cases. Handwritten digit recognition is a common instance of classification and has a long history of research and development since the early 1900s. A classification system, for example, can learn to read and understand handwritten ZIP codes (digits from 0 to 9 in most countries) by which envelopes are automatically sorted.

Handwritten digit recognition has become a "*Hello, World!*" in the journey of studying machine learning, and the scanned document dataset

constructed from the National Institute of Standards and Technology, called **MNIST (Modified National Institute of Standards and Technology)**, is a benchmark dataset frequently used to test and evaluate multiclass classification models. *Figure 2.3* shows four samples taken from the MNIST dataset:

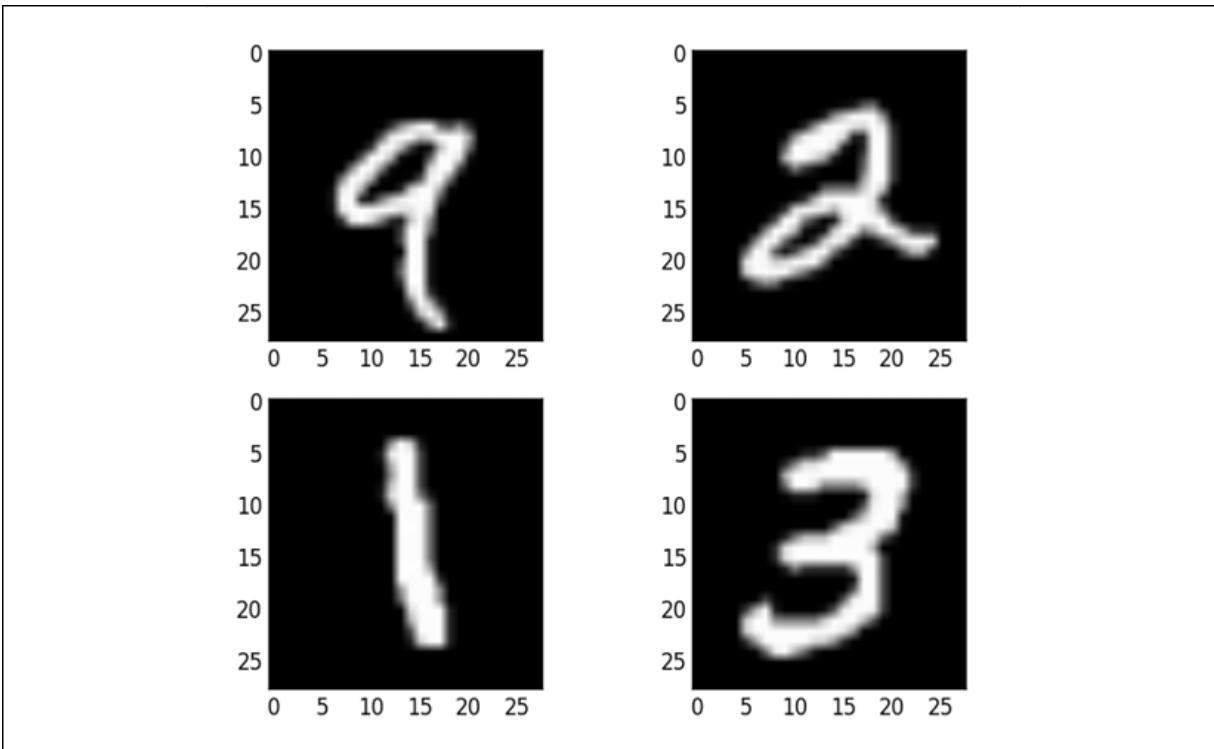


Figure 2.3: Samples from the MNIST dataset

In *Figure 2.4*, the multiclass classification model tries to find segregation boundaries to separate data from the following three different classes (denoted by dots, crosses, and triangles):

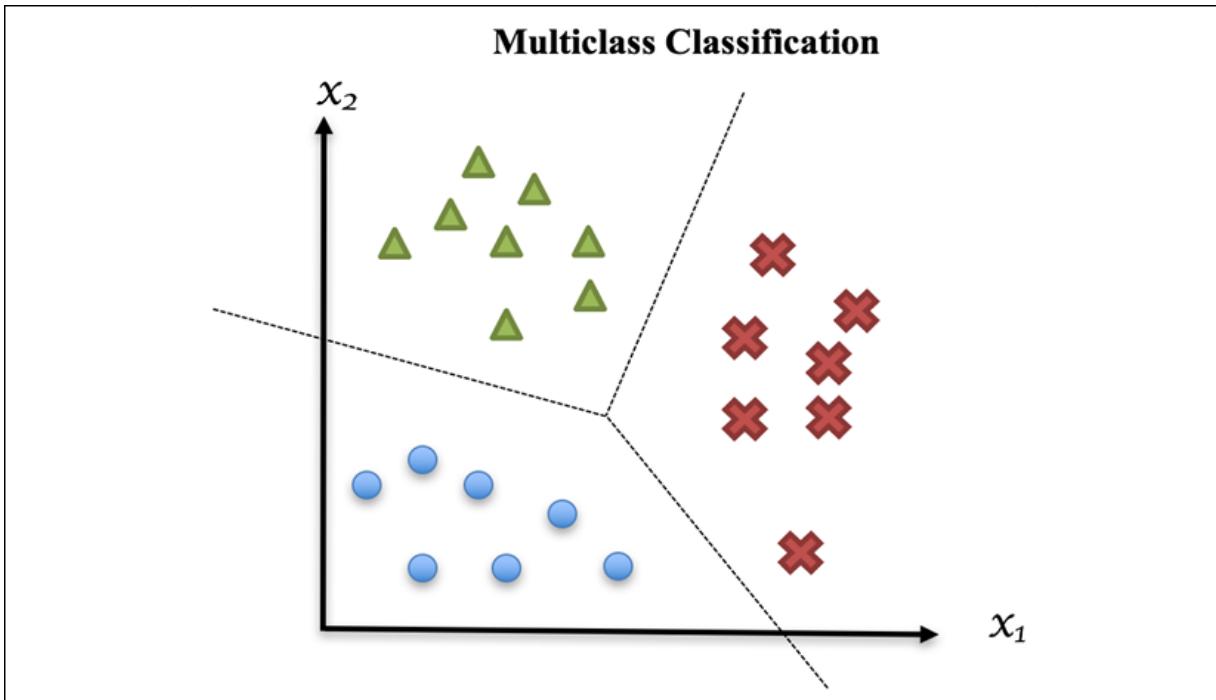


Figure 2.4: Multiclass classification example

Multi-label classification

In the first two types of classification, target classes are mutually exclusive and a sample is assigned *one, and only one*, label. It is the opposite in multi-label classification. Increasing research attention has been drawn to multi-label classification by the nature of the omnipresence of categories in modern applications. For example, a picture that captures a sea and a sunset can simultaneously belong to both conceptual scenes, whereas it can only be an image of either a cat or dog in a binary case, or one type of fruit among oranges, apples, and bananas in a multiclass case. Similarly, adventure films are often combined with other genres, such as fantasy, science fiction, horror, and drama.

Another typical application is protein function classification, as a protein may have more than one function—storage, antibody, support, transport, and so on.

A typical approach to solving an n -label classification problem is to transform it into a set of n binary classification problems, where each binary classification problem is handled by an individual binary classifier.

Refer to *Figure 2.5* to see the restructuring of a multi-label classification problem into a multiple binary classification problem:

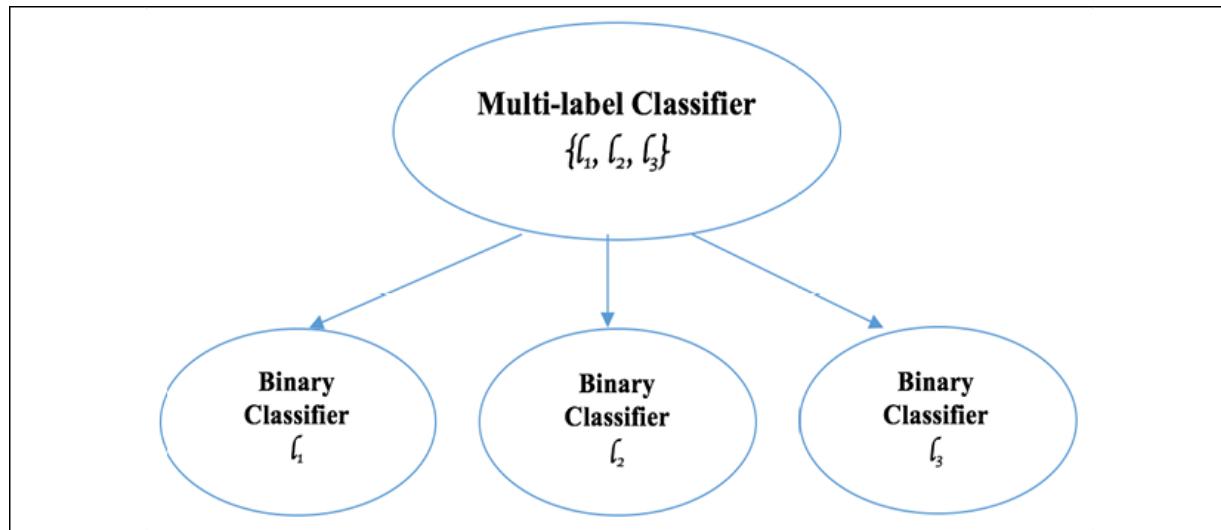


Figure 2.5: Transforming three-label classification into three independent binary classifications

To solve these problems, researchers have developed many powerful classification algorithms, among which Naïve Bayes, **support vector machine (SVM)**, decision tree, and logistic regression are often used. In the following sections, we will cover the mechanics of Naïve Bayes and its in-depth implementation, along with other important concepts, including classifier tuning and classification performance evaluation. Stay tuned for upcoming chapters that cover the other classification algorithms.

Exploring Naïve Bayes

The **Naïve Bayes** classifier belongs to the family of probabilistic classifiers. It computes the probabilities of each predictive **feature** (also referred to as an **attribute** or **signal**) of the data belonging to each class in order to make a prediction of probability distribution over all classes. Of course,

from the resulting probability distribution, we can conclude the most likely class that the data sample is associated with. What Naïve Bayes does specifically, as its name indicates, is as follows:

- **Bayes:** As in, it maps the probability of observed input features given a possible class to the probability of the class given observed pieces of evidence based on Bayes' theorem.
- **Naïve:** As in, it simplifies probability computation by assuming that predictive features are mutually independent.

I will explain Bayes' theorem with examples in the next section.

Learning Bayes' theorem by example

It is important to understand Bayes' theorem before diving into the classifier. Let A and B denote two events. Events could be that *it will rain tomorrow*; *two kings are drawn from a deck of cards*, or *a person has cancer*. In Bayes' theorem, $P(A | B)$ is the probability that A occurs given that B is true. It can be computed as follows:

$$P(A | B) = \frac{P(B | A) P(A)}{P(B)}$$

Here, $P(B | A)$ is the probability of observing B given that A occurs, while $P(A)$ and $P(B)$ are the probability that A and B occur, respectively. Is that too abstract? Let's consider the following concrete examples:

- Example 1: Given two coins, one is unfair, with 90% of flips getting a head and 10% getting a tail, while the other one is fair. Randomly pick one coin and flip it. What is the probability that this coin is the unfair one, if we get a head?

We can solve this by first denoting U for the event of picking the unfair coin, F for the fair coin, and H for the event of getting a head. So, the probability that the unfair coin has been picked when we get a head, $P(U | H)$, can be calculated with the following:

$$P(U | H) = \frac{P(H | U) P(U)}{P(H)}$$

As we know, $P(H | U)$ is 90%. $P(U)$ is 0.5 because we randomly pick a coin out of two. However, deriving the probability of getting a head, $P(H)$, is not that straightforward, as two events can lead to the following, where U is when the unfair coin is picked, and F is when the fair coin is picked:

$$P(H) = P(H | U)P(U) + P(H | F)P(F)$$

Now, $P(U | H)$ becomes the following:

$$P(U | H) = \frac{P(H | U) P(U)}{P(H)} = \frac{P(H | U) P(U)}{P(H | U)P(U) + P(H | F)P(F)} = \frac{0.9 * 0.5}{0.9 * 0.5 + 0.5 * 0.5} = 0.64$$

- Example 2: Suppose a physician reported the following cancer screening test scenario among 10,000 people:

	Cancer	No Cancer	Total
Test Positive	80	900	980
Test Negative	20	9000	9020
Total	100	9900	10000

Table 2.1: Example of a cancer screening result

This indicates that 80 out of 100 cancer patients are correctly diagnosed, while the other 20 are not; cancer is falsely detected in 900 out of 9,900 healthy people.

If the result of this screening test on a person is positive, what is the probability that they actually have cancer? Let's assign the event of having cancer and positive testing results as C and Pos , respectively. So we have

$P(Pos | C) = 80/100 = 0.8$, $P(C) = 100/1000 = 0.1$, and $P(Pos) = 980/1000 = 0.98$.

We can apply Bayes' theorem to calculate $P(C | Pos)$:

$$P(C | Pos) = \frac{P(Pos | C) P(C)}{P(Pos)} = \frac{0.8 * 0.1}{0.98} = 8.16\%$$

Given a positive screening result, the chance that the subject has cancer is 8.16%, which is significantly higher than the one under general assumption (100/10000=1%) without the subject undergoing the screening.

- Example 3: Three machines A , B , and C in a factory account for 35%, 20%, and 45% of bulb production. The fraction of defective bulbs produced by each machine is 1.5%, 1%, and 2%, respectively. A bulb produced by this factory was identified as defective, which is denoted as event D . What are the probabilities that this bulb was manufactured by machine A , B , and C , respectively?

Again, we can simply follow Bayes' theorem:

$$P(A|D) = \frac{P(D|A)P(A)}{P(D)} = \frac{P(D|A)P(A)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)}$$

$$= \frac{0.015 * 0.35}{0.015 * 0.35 + 0.01 * 0.2 + 0.02 * 0.45} = 0.323$$

$$P(B|D) = \frac{P(D|B)P(B)}{P(D)} = \frac{P(D|B)P(B)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)}$$

$$= \frac{0.01 * 0.2}{0.015 * 0.35 + 0.01 * 0.2 + 0.02 * 0.45} = 0.123$$

$$P(C|D) = \frac{P(D|C)P(C)}{P(D)} = \frac{P(D|C)P(C)}{P(D|A)P(A) + P(D|B)P(B) + P(D|C)P(C)}$$

$$= \frac{0.02 * 0.45}{0.015 * 0.35 + 0.01 * 0.2 + 0.02 * 0.45} = 0.554$$

Also, either way, we do not even need to calculate $P(D)$ since we know that the following is the case:

$$P(A|D):P(B|D):P(C|D) = P(D|A)P(A):P(D|B)P(B):P(D|C)P(C) = 21:8:36$$

We also know the following concept:

$$P(A|D) + P(B|D) + P(C|D) = 1$$

So, we have the following formula:

$$P(A|D) = \frac{21}{21+8+36} = 0.323$$

$$P(B|D) = \frac{8}{21+8+36} = 0.133$$

Now that you understand Bayes' theorem as the backbone of Naïve Bayes, we can easily move forward with the classifier itself.

The mechanics of Naïve Bayes

Let's start by discussing the magic behind the algorithm—how Naïve Bayes works. Given a data sample, x , with n features, x_1, x_2, \dots, x_n (x represents a feature vector and $x = (x_1, x_2, \dots, x_n)$), the goal of Naïve Bayes is to determine the probabilities that this sample belongs to each of K possible classes y_1, y_2, \dots, y_K , which is $P(y_K | x)$ or $P(x_1, x_2, \dots, x_n)$, where $k = 1, 2, \dots, K$.

This looks no different from what we have just dealt with: x or x_1, x_2, \dots, x_n . This is a joint event where a sample that has observed feature values x_1, x_2, \dots, x_n . y_K is the event that the sample belongs to class k . We can apply Bayes' theorem right away:

$$P(y_k | x) = \frac{P(x | y_k) P(y_k)}{P(x)}$$

Let's look at each component in detail:

- $P(y_K)$ portrays how classes are distributed, with no further knowledge of observation features. Thus, it is also called **prior** in Bayesian probability terminology. Prior can be either predetermined (usually in a uniform manner where each class has an equal chance of occurrence) or learned from a set of training samples.
- $P(y_K | x)$, in contrast to prior $P(y_K)$, is the **posterior**, with extra knowledge of observation.
- $P(x | y_K)$, or $P(x_1, x_2, \dots, x_n | y_K)$, is the joint distribution of n features, given that the sample belongs to class y_K . This is how likely the features with such values co-occur. This is named **likelihood** in Bayesian terminology. Obviously, the likelihood will be difficult to compute as the number of features increases. In Naïve Bayes, this is solved thanks to the feature independence assumption. The joint conditional distribution of n features can be expressed as the joint product of individual feature conditional distributions:

$$P(x | y_k) = P(x_1 | y_k) * P(x_2 | y_k) * \dots * P(x_n | y_k)$$

Each conditional distribution can be efficiently learned from a set of training samples.

- $P(x)$, also called **evidence**, solely depends on the overall distribution of features, which is not specific to certain classes and is therefore a normalization constant. As a result, posterior is proportional to prior and likelihood:

$$P(y_k | x) \propto P(x | y_k)P(y_k) = P(x_1 | y_k) * P(x_2 | y_k) * \dots * P(x_n | y_k)$$

Figure 2.6 summarizes how a Naïve Bayes classification model is trained and applied to new data:

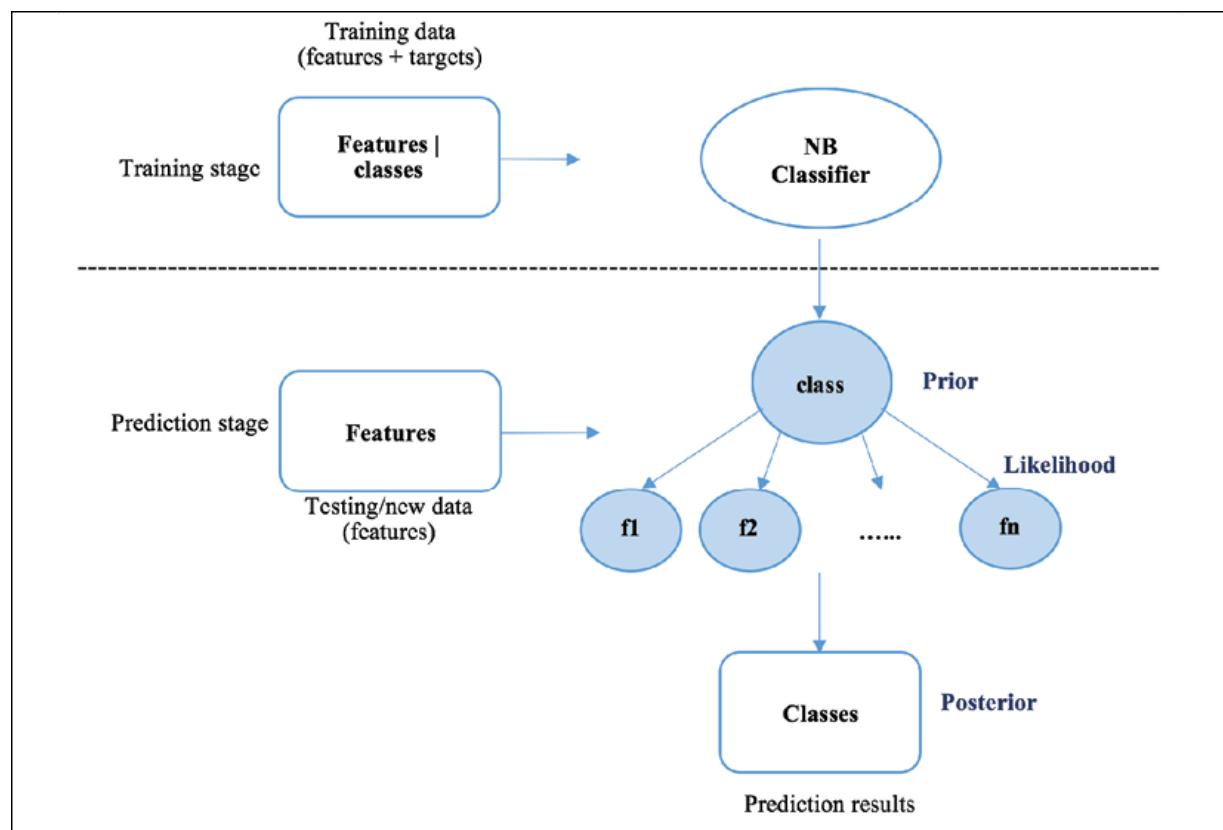


Figure 2.6: Training and prediction stages in Naïve Bayes classification

Let's see a Naïve Bayes classifier in action through a simplified example of movie recommendation before we jump to the implementations of Naïve

Bayes. Given four (pseudo) users, whether they like each of three movies, m_1, m_2, m_3 (indicated as 1 or 0), and whether they like a target movie (denoted as event Y) or not (denoted as event N), as shown in the following table, we are asked to predict how likely it is that another user will like that movie:

	ID	m1	m2	m3	Whether the user likes the target movie
Training data	1	0	1	1	Y
	2	0	0	1	N
	3	0	0	0	Y
	4	1	1	0	Y
Testing case	5	1	1	0	?

Table 2.2: Toy data example for a movie recommendation

Whether users like three movies, m_1, m_2, m_3 , are features (signals) that we can utilize to predict the target class. The training data we have are the four samples with both ratings and target information.

Now, let's first compute the prior, $P(Y)$ and $P(N)$. From the training set, we can easily get the following:

$$P(Y) = 3/4$$

$$P(N) = 1/4$$

Alternatively, we can also impose an assumption of a uniform prior that $P(Y) = 50\%$, for example.

For simplicity, we will denote the event that a user likes three movies or not as f_1, f_2, f_3 , respectively. To calculate posterior $P(Y|x)$, where $x = (1, 1, 0)$, the first step is to compute the likelihoods, $P(f_1 = 1|Y)$, $P(f_2 = 1|Y)$, and $P(f_3 = 0|Y)$, and similarly, $P(f_1 = 1|N)$, $P(f_2 = 1|N)$, and $P(f_3 = 0|N)$ based on the training set. However, you may notice that since $f_1 = 1$ was not seen in the N class, we will get $P(f_1 = 1|N) = 0$. Consequently, we will have $P(N|x) \propto P(f_1 = 1|N) * P(f_2 = 1|N) = 0$, which means we will recklessly predict class = Y by any means.

To eliminate the zero-multiplication factor, the unknown likelihood, we usually assign an initial value of 1 to each feature, that is, we start counting each possible value of a feature from one. This technique is also known as **Laplace smoothing**. With this amendment, we now have the following:

$$P(f_1 = 1|N) = \frac{0 + 1}{1 + 2} = \frac{1}{3}$$

$$P(f_1 = 1|Y) = \frac{1 + 1}{3 + 2} = \frac{2}{5}$$

Here, given class N , $0 + 1$ means there are zero likes of m_1 plus +1 smoothing; $1 + 2$ means there is one data point (ID = 2) plus two (two possible values) + 1 smoothings. Given class Y , $1 + 1$ means there is one like of m_1 (ID = 4) plus +1 smoothing; $3 + 2$ means there are three data points (ID = 1, 3, 4) plus two (two possible values) + 1 smoothings.

Similarly, we can compute the following:

$$P(f_2 = 1|N) = \frac{0 + 1}{1 + 2} = \frac{1}{3}$$

$$P(f_2 = 1|Y) = \frac{2 + 1}{3 + 2} = \frac{3}{5}$$

$$P(f_3 = 0|N) = \frac{0 + 1}{1 + 2} = \frac{1}{3}$$

$$P(f_3 = 0 \mid Y) = \frac{2 + 1}{3 + 2} = \frac{3}{5}$$

Now, we can compute the ratio between two posteriors as follows:

$$\frac{P(N \mid x)}{P(Y \mid x)} \propto \frac{P(N) * P(f_1 = 1 \mid N) * P(f_2 = 1 \mid N) * P(f_3 = 0 \mid N)}{P(Y) * P(f_1 = 1 \mid Y) * P(f_2 = 1 \mid Y) * P(f_3 = 0 \mid Y)} = \frac{125}{1458}$$

Also, remember this:

$$P(N \mid x) + P(Y \mid x) = 1$$

So, finally, we have the following:

$$P(Y \mid x) = 92.1\%$$

There is a 92.1% chance that the new user will like the target movie.

I hope that you now have a solid understanding of Naïve Bayes after going through the theory and a toy example. Let's get ready for its implementation in the next section.

Implementing Naïve Bayes

After calculating by hand the movie preference prediction example, as promised, we are going to code Naïve Bayes from scratch. After that, we will implement it using the `scikit-learn` package.

Implementing Naïve Bayes from scratch

Before we develop the model, let's define the toy dataset we just worked with:

```

>>> import numpy as np
>>> X_train = np.array([
...     [0, 1, 1],
...     [0, 0, 1],
...     [0, 0, 0],
...     [1, 1, 0]])
>>> Y_train = ['Y', 'N', 'Y', 'Y']
>>> X_test = np.array([[1, 1, 0]])

```

For the model, starting with the prior, we first group the data by label and record their indices by classes:

```

>>> def get_label_indices(labels):
...     """
...     Group samples based on their labels and return indices
...     @param labels: list of labels
...     @return: dict, {class1: [indices], class2: [indices]}
...     """
...     from collections import defaultdict
...     label_indices = defaultdict(list)
...     for index, label in enumerate(labels):
...         label_indices[label].append(index)
...     return label_indices

```

Take a look at what we get:

```

>>> label_indices = get_label_indices(Y_train)
>>> print('label_indices:\n', label_indices)
label_indices
defaultdict(<class 'list'>, {'Y': [0, 2, 3], 'N': [1]})

```

With `label_indices`, we calculate the prior:

```

>>> def get_prior(label_indices):
...     """
...     Compute prior based on training samples
...     @param label_indices: grouped sample indices by class
...     @return: dictionary, with class label as key, corresponding
...             prior as the value
...     """
...     prior = {label: len(indices) for label, indices in
...              label_indices.items()}

```

```
...             label_indices.items())
...         total_count = sum(prior.values())
...         for label in prior:
...             prior[label] /= total_count
...     return prior
```

Take a look at the computed prior:

```
>>> prior = get_prior(label_indices)
>>> print('Prior:', prior)
Prior: {'Y': 0.75, 'N': 0.25}
```

With `prior` calculated, we continue with `likelihood`, which is the conditional probability, $P(\text{feature}|\text{class})$:

```
>>> def get_likelihood(features, label_indices, smoothing=0):
...     """
...     Compute likelihood based on training samples
...     @param features: matrix of features
...     @param label_indices: grouped sample indices by class
...     @param smoothing: integer, additive smoothing parameter
...     @return: dictionary, with class as key, corresponding
...             conditional probability  $P(\text{feature}|\text{class})$  vector
...             as value
...
...     """
...     likelihood = {}
...     for label, indices in label_indices.items():
...         likelihood[label] = features[indices, :].sum(axis=0)
...                         + smoothing
...         total_count = len(indices)
...         likelihood[label] = likelihood[label] /
...                            (total_count + 2 * smoothing)
...
...     return likelihood
```

We set the `smoothing` value to 1 here, which can also be 0 for no smoothing, or any other positive value, as long as a higher classification performance is achieved:

```

>>> smoothing = 1
>>> likelihood = get_likelihood(X_train, label_indices, smoothir
>>> print('Likelihood:\n', likelihood)
Likelihood:
{'Y': array([0.4, 0.6, 0.4]), 'N': array([0.33333333, 0.33333333])

```

If you ever find any of this confusing, feel free to check *Figure 2.7* to refresh your memory:

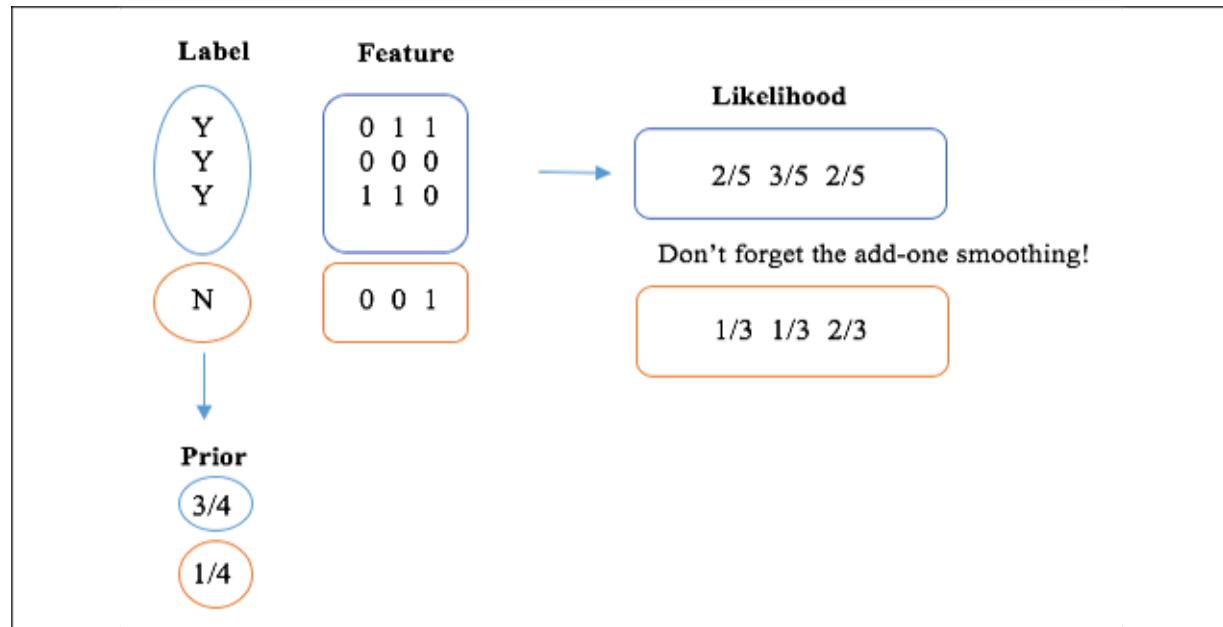


Figure 2.7: A simple example of computing prior and likelihood

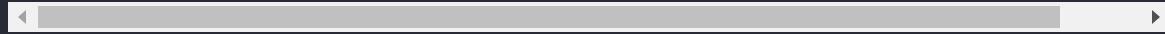
With prior and likelihood ready, we can now compute the posterior for the testing/new samples:

```

>>> def get_posterior(X, prior, likelihood):
...     """
...     Compute posterior of testing samples, based on prior and
...     likelihood
...     @param X: testing samples
...     @param prior: dictionary, with class label as key,
...                 corresponding prior as the value
...     @param likelihood: dictionary, with class label as key,
...                       corresponding conditional probability
...                       vector as value
...
...     Output: dictionary, with class label as key, corresponding
...             posterior probability vector as value

```

```
...     @return: dictionary, with class label as key, correspond-
...             ing posterior as value
...
...
...     posteriors = []
...     for x in X:
...         # posterior is proportional to prior * likelihood
...         posterior = prior.copy()
...         for label, likelihood_label in likelihood.items():
...             for index, bool_value in enumerate(x):
...                 posterior[label] *= likelihood_label[index]
...                 bool_value else (1 - likelihood_label[index])
...         # normalize so that all sums up to 1
...         sum_posterior = sum(posterior.values())
...         for label in posterior:
...             if posterior[label] == float('inf'):
...                 posterior[label] = 1.0
...             else:
...                 posterior[label] /= sum_posterior
...         posteriors.append(posterior.copy())
...
...     return posteriors
```



Now, let's predict the class of our one sample test set using this prediction function:

```
>>> posterior = get_posterior(X_test, prior, likelihood)
>>> print('Posterior:\n', posterior)
Posterior:
[{'Y': 0.9210360075805433, 'N': 0.07896399241945673}]
```

This is exactly what we got previously. We have successfully developed Naïve Bayes from scratch and we can now move on to the implementation using `scikit-learn`.

Implementing Naïve Bayes with scikit-learn

Coding from scratch and implementing your own solutions is the best way to learn about machine learning models. Of course, you can take a shortcut by directly using the `BernoulliNB` module (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html)

[learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html](https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html)) from the scikit-learn API:

```
>>> from sklearn.naive_bayes import BernoulliNB
```

Let's initialize a model with a smoothing factor (specified as `alpha` in `scikit-learn`) of `1.0`, and `prior` learned from the training set (specified as `fit_prior=True` in `scikit-learn`):

```
>>> clf = BernoulliNB(alpha=1.0, fit_prior=True)
```

To train the Naïve Bayes classifier with the `fit` method, we use the following line of code:

```
>>> clf.fit(X_train, Y_train)
```

And to obtain the predicted probability results with the `predict_proba` method, we use the following lines of code:

```
>>> pred_prob = clf.predict_proba(X_test)
>>> print('[scikit-learn] Predicted probabilities:\n', pred_prob)
[scikit-learn] Predicted probabilities:
[[0.07896399 0.92103601]]
```

Finally, we do the following to directly acquire the predicted class with the `predict` method (0.5 is the default threshold, and if the predicted probability of class `Y` is greater than 0.5, class `Y` is assigned; otherwise, `N` is used):

```
>>> pred = clf.predict(X_test)
>>> print('[scikit-learn] Prediction:', pred)
[scikit-learn] Prediction: ['Y']
```

The prediction results using scikit-learn are consistent with what we got using our own solution. Now that we've implemented the algorithm both from scratch and using `scikit-learn`, why don't we use it to solve the movie recommendation problem?

Building a movie recommender with Naïve Bayes

After the toy example, it is now time to build a movie recommender (or, more specifically, movie preference classifier) using a real dataset. We herein use a movie rating dataset

(<https://grouplens.org/datasets/movielens/>). The movie rating data was collected by the GroupLens Research group from the MovieLens website (<http://movielens.org>).

For demonstration purposes, we will use the small dataset, ml-latest-small (downloaded from the following link:

<http://files.grouplens.org/datasets/movielens/ml-latest-small.zip> of ml-latest-small.zip (size: 1 MB)) as an example. It has around 100,000 ratings, ranging from 1 to 5, given by 6,040 users on 3,706 movies (last updated September 2018).

Unzip the `ml-1m.zip` file and you will see the following four files:

- `movies.dat` : It contains the movie information in the format of MovieID::Title::Genres.
- `ratings.dat` : It contains user movie ratings in the format of UserID::MovieID::Rating::Timestamp. We will only be using data from this file in this chapter.
- `users.dat` : It contains user information in the format of UserID::Gender::Age::Occupation::Zip-code.
- `README`

Let's attempt to determine whether a user likes a particular movie based on how users rate other movies (again, ratings are from 1 to 5).

First, we import all the necessary modules and variables:

```
>>> import numpy as np
>>> from collections import defaultdict
>>> data_path = 'ml-1m/ratings.dat'
>>> n_users = 6040
>>> n_movies = 3706
```

We then develop the following function to load the rating data from `ratings.dat`:

```
>>> def load_rating_data(data_path, n_users, n_movies):
...     """
...     Load rating data from file and also return the number of
...     ratings for each movie and movie_id index mapping
...     @param data_path: path of the rating data file
...     @param n_users: number of users
...     @param n_movies: number of movies that have ratings
...     @return: rating data in the numpy array of [user, movie]
...             movie_n_rating, {movie_id: number of ratings};
...             movie_id_mapping, {movie_id: column index in
...                                 rating data}
...     """
...     data = np.zeros([n_users, n_movies], dtype=np.float32)
...     movie_id_mapping = {}
...     movie_n_rating = defaultdict(int)
...     with open(data_path, 'r') as file:
...         for line in file.readlines()[1:]:
...             user_id, movie_id, rating, _ = line.split(":")
...             user_id = int(user_id) - 1
...             if movie_id not in movie_id_mapping:
...                 movie_id_mapping[movie_id] =
...                     len(movie_id_mapping)
...             rating = int(rating)
...             data[user_id, movie_id_mapping[movie_id]] = rating
...             if rating > 0:
...                 movie_n_rating[movie_id] += 1
...     return data, movie_n_rating, movie_id_mapping
```

And then we load the data using this function:

```
>>> data, movie_n_rating, movie_id_mapping =  
...     load_rating_data(data_path, n_users, n_movies)
```

It is always recommended to analyze the data distribution. We do the following:

```
>>> def display_distribution(data):  
...     values, counts = np.unique(data, return_counts=True)  
...     for value, count in zip(values, counts):  
...         print(f'Number of rating {int(value)}: {count}')  
>>> display_distribution(data)  
Number of rating 0: 21384032  
Number of rating 1: 56174  
Number of rating 2: 107557  
Number of rating 3: 261197  
Number of rating 4: 348971  
Number of rating 5: 226309
```

As you can see, most ratings are unknown; for the known ones, 35% are of rating 4, followed by 26% of rating 3, and 23% of rating 5, and then 11% and 6% of ratings 2 and 1, respectively.

Since most ratings are unknown, we take the movie with the most known ratings as our target movie:

```
>>> movie_id_most, n_rating_most = sorted(movie_n_rating.items())  
...     key=lambda d: d[1], reverse=True)[0]  
>>> print(f'Movie ID {movie_id_most} has {n_rating_most} ratings')  
Movie ID 2858 has 3428 ratings.
```

The movie with ID 2858 is the target movie, and ratings of the rest of the movies are signals. We construct the dataset accordingly:

```
>>> X_raw = np.delete(data, movie_id_mapping[movie_id_most],  
...                     axis=1)  
>>> Y_raw = data[:, movie_id_mapping[movie_id_most]]
```

We discard samples without a rating in movie ID 2858:

```
>>> X = X_raw[Y_raw > 0]
>>> Y = Y_raw[Y_raw > 0]
>>> print('Shape of X:', X.shape)
Shape of X: (3428, 3705)
>>> print('Shape of Y:', Y.shape)
Shape of Y: (3428,)
```

Again, we take a look at the distribution of the target movie ratings:

```
>>> display_distribution(Y)
Number of rating 1: 83
Number of rating 2: 134
Number of rating 3: 358
Number of rating 4: 890
Number of rating 5: 1963
```

We can consider movies with ratings greater than 3 as being liked (being recommended):

```
>>> recommend = 3
>>> Y[Y <= recommend] = 0
>>> Y[Y > recommend] = 1
>>> n_pos = (Y == 1).sum()
>>> n_neg = (Y == 0).sum()
>>> print(f'{n_pos} positive samples and {n_neg} negative
...           samples.')
2853 positive samples and 575 negative samples.
```

As a rule of thumb in solving classification problems, we need to always analyze the label distribution and see how balanced (or imbalanced) the dataset is.

Next, to comprehensively evaluate our classifier's performance, we can randomly split the dataset into two sets, the training and testing sets, which simulate learning data and prediction data, respectively. Generally, the proportion of the original dataset to include in the testing split can be 20%,

25%, 33.3%, or 40%. We use the `train_test_split` function from `scikit-learn` to do the random splitting and to preserve the percentage of samples for each class:

```
>>> from sklearn.model_selection import train_test_split  
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,  
...           test_size=0.2, random_state=42)
```



It is a good practice to assign a fixed `random_state` (for example, 42) during experiments and exploration in order to guarantee that the same training and testing sets are generated every time the program runs. This allows us to make sure that the classifier functions and performs well on a fixed dataset before we incorporate randomness and proceed further.

We check the training and testing sizes as follows:

```
>>> print(len(Y_train), len(Y_test))  
2742 686
```

Another good thing about the `train_test_split` function is that the resulting training and testing sets will have the same class ratio.

Next, we train a Naïve Bayes model on the training set. You may notice that the values of the input features are from 0 to 5, as opposed to 0 or 1 in our toy example. Hence, we use the `MultinomialNB` module (https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html) from scikit-learn instead of the `BernoulliNB` module, as `MultinomialNB` can work with integer features. We import the module, initialize a model with a smoothing factor of 1.0 and `prior` learned from the training set, and train this model against the training set as follows:

```
>>> from sklearn.naive_bayes import MultinomialNB  
>>> clf = MultinomialNB(alpha=1.0, fit_prior=True)  
>>> clf.fit(X_train, Y_train)
```

Then, we use the trained model to make predictions on the testing set. We get the predicted probabilities as follows:

```
>>> prediction_prob = clf.predict_proba(X_test)
>>> print(prediction_prob[0:10])
[[7.50487439e-23 1.00000000e+00]
 [1.01806208e-01 8.98193792e-01]
 [3.57740570e-10 1.00000000e+00]
 [1.00000000e+00 2.94095407e-16]
 [1.00000000e+00 2.49760836e-25]
 [7.62630220e-01 2.37369780e-01]
 [3.47479627e-05 9.99965252e-01]
 [2.66075292e-11 1.00000000e+00]
 [5.88493563e-10 9.99999999e-01]
 [9.71326867e-09 9.99999990e-01]]
```

We get the predicted class as follows:

```
>>> prediction = clf.predict(X_test)
>>> print(prediction[:10])
[1. 1. 1. 0. 0. 0. 1. 1. 1. 1.]
```

Finally, we evaluate the model's performance with classification accuracy, which is the proportion of correct predictions:

```
>>> accuracy = clf.score(X_test, Y_test)
>>> print(f'The accuracy is: {accuracy*100:.1f}%')
The accuracy is: 71.6%
```

The classification accuracy is around 72%, which means that the Naïve Bayes classifier we just developed correctly recommends movies to around 72% of users. This is not bad, given that we extracted user-movie relationships only from the movie rating data where most ratings are unknown. Ideally, we could also utilize movie genre information from the file `movies.dat`, and user demographics (gender, age, occupation, and zip code) information from the file `users.dat`. Obviously, movies in similar genres tend to attract similar users, and users of similar demographics likely have similar movie preferences.

So far, we have covered in depth the first machine learning classifier and evaluated its performance by prediction accuracy. Are there any other classification metrics? Let's see in the next section.

Evaluating classification performance

Beyond accuracy, there are several metrics we can use to gain more insight and to avoid class imbalance effects. These are as follows:

- Confusion matrix
- Precision
- Recall
- F1 score
- Area under the curve

A **confusion matrix** summarizes testing instances by their predicted values and true values, presented as a contingency table:

		Predicted	
		Negative	Positive
Actual	Negative	TN	FP
	Positive	FN	TP

TN = True Negative
FP = False Positive
FN = False Negative
TP = True Positive

Table 2.3: Contingency table for a confusion matrix

To illustrate this, we can compute the confusion matrix of our Naïve Bayes classifier. We use the `confusion_matrix` function from `scikit-learn` to compute it, but it is very easy to code it ourselves:

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(Y_test, prediction, labels=[0, 1]))
[[ 60  47]
 [148 431]]
```

As you can see from the resulting confusion matrix, there are 47 false positive cases (where the model misinterprets a dislike as a like for a movie), and 148 false negative cases (where it fails to detect a like for a movie). Hence, classification accuracy is just the proportion of all true cases:

$$\frac{TN + TP}{TN + TP + FP + FN} = \frac{60 + 431}{60 + 431 + 47 + 148} = 71.6\%$$

Precision measures the fraction of positive calls that are correct, which is $\frac{TP}{TP + FP}$, and $\frac{431}{431 + 47} = 0.90$ in our case.

Recall, on the other hand, measures the fraction of true positives that are correctly identified, which is $\frac{TP}{TP + FN}$ and $\frac{431}{431 + 148} = 0.74$ in our case. Recall is also called the **true positive rate**.

The **f1 score** comprehensively includes both the precision and the recall, and equates to their **harmonic mean**: $f_1 = 2 * \frac{precision * recall}{precision + recall}$. We tend to value the f1 score above precision or recall alone.

Let's compute these three measurements using corresponding functions from `scikit-learn`, as follows:

```
>>> from sklearn.metrics import precision_score, recall_score, f1_score
>>> precision_score(Y_test, prediction, pos_label=1)
0.9016736401673641
>>> recall_score(Y_test, prediction, pos_label=1)
0.7443868739205527
>>> f1_score(Y_test, prediction, pos_label=1)
0.815515610217597
```

On the other hand, the negative (dislike) class can also be viewed as positive, depending on the context. For example, assign the `0` class as `pos_label` and we have the following:

```
>>> f1_score(Y_test, prediction, pos_label=0)
0.38095238095238093
```

To obtain the precision, recall, and f1 score for each class, instead of exhausting all class labels in the three function calls as shown earlier, a quicker way is to call the `classification_report` function:

```
>>> from sklearn.metrics import classification_report
>>> report = classification_report(Y_test, prediction)
>>> print(report)
           precision    recall  f1-score   support
          0.0       0.29      0.56      0.38      107
          1.0       0.90      0.74      0.82      579
    micro avg       0.72      0.72      0.72      686
  macro avg       0.60      0.65      0.60      686
weighted avg       0.81      0.72      0.75      686
```

Here, `weighted avg` is the weighted average according to the proportions of the class.

The classification report provides a comprehensive view of how the classifier performs on each class. It is, as a result, useful in imbalanced classification, where we can easily obtain a high accuracy by simply classifying every sample as the dominant class, while the precision, recall, and f1 score measurements for the minority class, however, will be significantly low.

Precision, recall, and the f1 score are also applicable to **multiclass** classification, where we can simply treat a class we are interested in as a positive case, and any other classes as negative cases.

During the process of tweaking a binary classifier (that is, trying out different combinations of hyperparameters, for example, the smoothing factor in our Naïve Bayes classifier), it would be perfect if there was a set

of parameters in which the highest averaged and class individual f1 scores are achieved at the same time. It is, however, usually not the case.

Sometimes, a model has a higher average f1 score than another model, but a significantly low f1 score for a particular class; sometimes, two models have the same average f1 scores, but one has a higher f1 score for one class and a lower score for another class. In situations such as these, how can we judge which model works better? The **area under the curve (AUC)** of the **receiver operating characteristic (ROC)** is a consolidated measurement frequently used in binary classification.

The ROC curve is a plot of the true positive rate versus the false positive rate at various probability thresholds, ranging from 0 to 1. For a testing sample, if the probability of a positive class is greater than the threshold, then a positive class is assigned; otherwise, we use a negative class. To recap, the true positive rate is equivalent to recall, and the false positive rate is the fraction of negatives that are incorrectly identified as positive. Let's code and exhibit the ROC curve (under thresholds of 0.0, 0.1, 0.2, ..., 1.0) of our model:

```
>>> pos_prob = prediction_prob[:, 1]
>>> thresholds = np.arange(0.0, 1.1, 0.05)
>>> true_pos, false_pos = [0]*len(thresholds), [0]*len(thresholds)
>>> for pred, y in zip(pos_prob, Y_test):
...     for i, threshold in enumerate(thresholds):
...         if pred >= threshold:
...             # if truth and prediction are both 1
...             if y == 1:
...                 true_pos[i] += 1
...             # if truth is 0 while prediction is 1
...             else:
...                 false_pos[i] += 1
...     else:
...         break
```

Then, let's calculate the true and false positive rates for all threshold settings (remember, there are 516.0 positive testing samples and 1191 negative ones):

```
>>> n_pos_test = (Y_test == 1).sum()
>>> n_neg_test = (Y_test == 0).sum()
>>> true_pos_rate = [tp / n_pos_test for tp in true_pos]
>>> false_pos_rate = [fp / n_neg_test for fp in false_pos]
```

Now, we can plot the ROC curve with `Matplotlib`:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> lw = 2
>>> plt.plot(false_pos_rate, true_pos_rate,
...           color='darkorange', lw=lw)
>>> plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')
>>> plt.xlim([0.0, 1.0])
>>> plt.ylim([0.0, 1.05])
>>> plt.xlabel('False Positive Rate')
>>> plt.ylabel('True Positive Rate')
>>> plt.title('Receiver Operating Characteristic')
>>> plt.legend(loc="lower right")
>>> plt.show()
```

Refer to *Figure 2.8* for the resulting ROC curve:

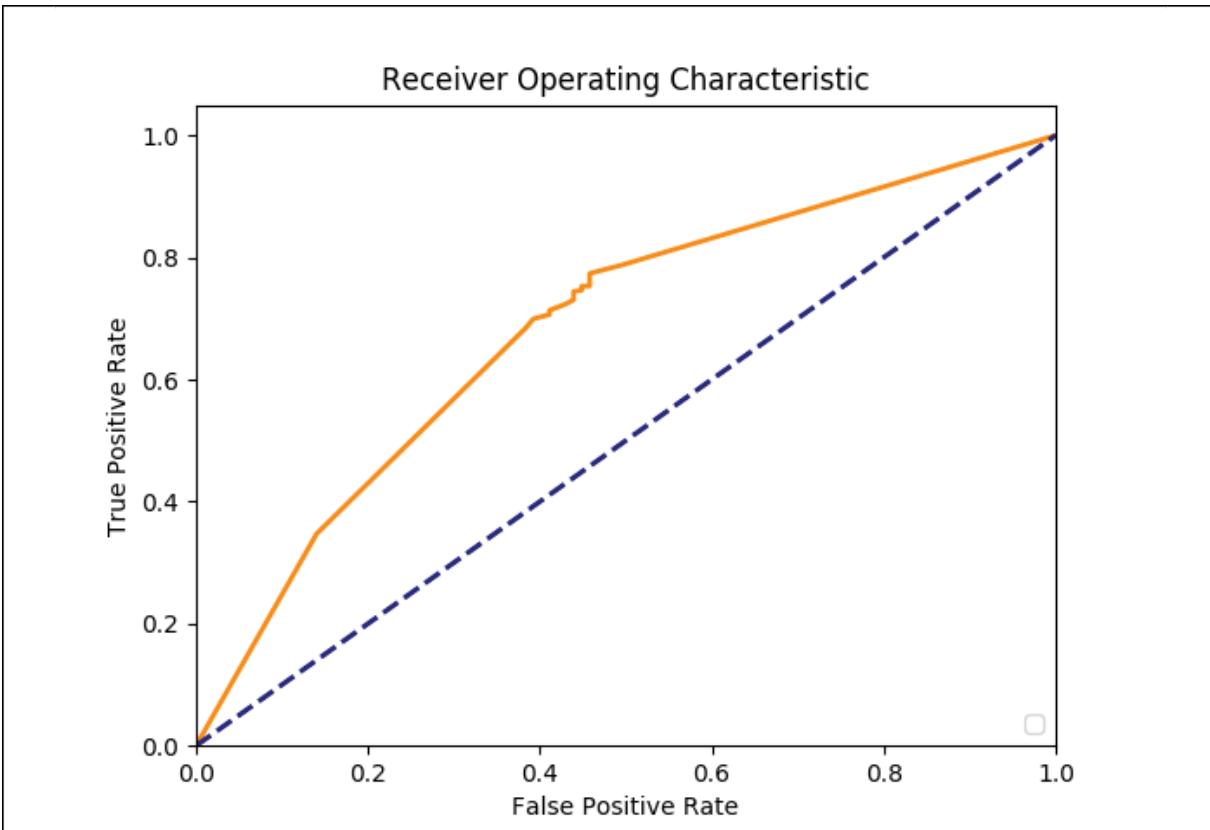


Figure 2.8: ROC curve

In the graph, the dashed line is the baseline representing random guessing, where the true positive rate increases linearly with the false positive rate; its AUC is 0.5. The solid line is the ROC plot of our model, and its AUC is somewhat less than 1. In a perfect case, the true positive samples have a probability of 1, so that the ROC starts at the point with 100% true positive and 0% false positive. The AUC of such a perfect curve is 1. To compute the exact AUC of our model, we can resort to the `roc_auc_score` function of `scikit-learn`:

```
>>> from sklearn.metrics import roc_auc_score
>>> roc_auc_score(Y_test, pos_prob)
0.6857375752586637
```



What AUC value leads to the conclusion that a classifier is good?
Unfortunately, there is no such "magic" number. We use the following rule of thumb as general guidelines: classification models achieving an AUC of



0.7 to 0.8 are considered acceptable, 0.8 to 0.9 are great, and anything above 0.9 are superb. Again, in our case, we are only using the very sparse movie rating data. Hence, an AUC of 0.69 is actually acceptable.

You have learned several classification metrics, and we will explore how to measure them properly and how to fine-tune our models in the next section.

Tuning models with cross-validation

We can simply avoid adopting the classification results from one fixed testing set, which we did in experiments previously. Instead, we usually apply the **k-fold cross-validation** technique to assess how a model will generally perform in practice.

In the k-fold cross-validation setting, the original data is first randomly divided into k equal-sized subsets, in which class proportion is often preserved. Each of these k subsets is then successively retained as the testing set for evaluating the model. During each trial, the rest of the $k - 1$ subsets (excluding the one-fold holdout) form the training set for driving the model. Finally, the average performance across all k trials is calculated to generate an overall result:

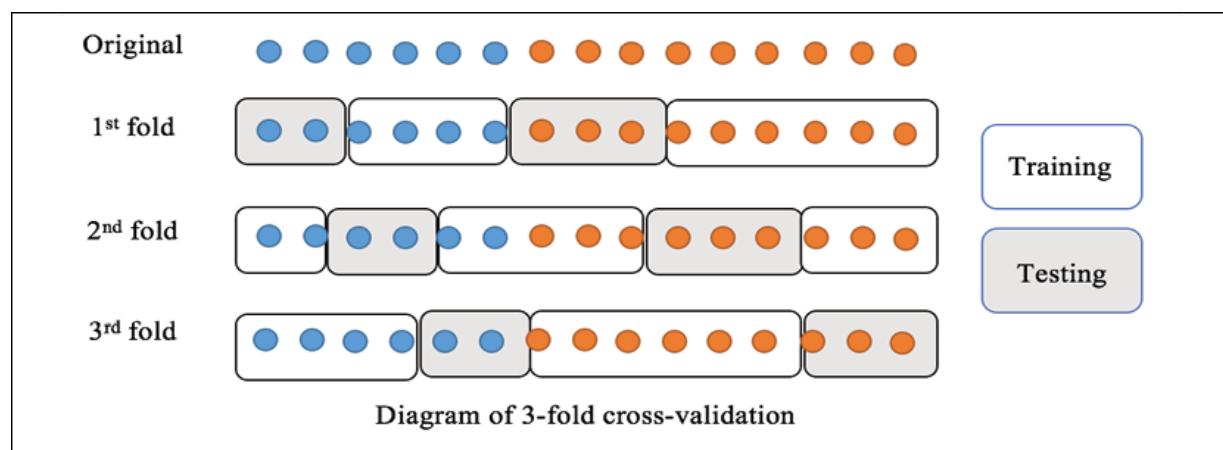


Figure 2.9: Diagram of 3-fold cross-validation

Statistically, the averaged performance of k-fold cross-validation is a better estimate of how a model performs in general. Given different sets of parameters pertaining to a machine learning model and/or data preprocessing algorithms, or even two or more different models, the goal of model tuning and/or model selection is to pick a set of parameters of a classifier so that the best averaged performance is achieved. With these concepts in mind, we can now start to tweak our Naïve Bayes classifier, incorporating cross-validation and the AUC of ROC measurements.



In k-fold cross-validation, k is usually set at 3, 5, or 10. If the training size is small, a large k (5 or 10) is recommended to ensure sufficient training samples in each fold. If the training size is large, a small value (such as 3 or 4) works fine since a higher k will lead to an even higher computational cost of training on a large dataset.

We will use the `split()` method from the `StratifiedKFold` class of `scikit-learn` to divide the data into chunks with preserved class distribution:

```
>>> from sklearn.model_selection import StratifiedKFold  
>>> k = 5  
>>> k_fold = StratifiedKFold(n_splits=k, random_state=42)
```

After initializing a 5-fold generator, we choose to explore the following values for the following parameters:

- `alpha` : This represents the smoothing factor, the initial value for each feature.
- `fit_prior` : This represents whether to use prior tailored to the training data.

We start with the following options:

```
>>> smoothing_factor_option = [1, 2, 3, 4, 5, 6]  
>>> fit_prior_option = [True, False]
```

```
>>> auc_record = {}
```

Then, for each fold generated by the `split()` method of the `k_fold` object, we repeat the process of classifier initialization, training, and prediction with one of the aforementioned combinations of parameters, and record the resulting AUCs:

```
>>> for train_indices, test_indices in k_fold.split(X, Y):
...     X_train, X_test = X[train_indices], X[test_indices]
...     Y_train, Y_test = Y[train_indices], Y[test_indices]
...     for alpha in smoothing_factor_option:
...         if alpha not in auc_record:
...             auc_record[alpha] = {}
...         for fit_prior in fit_prior_option:
...             clf = MultinomialNB(alpha=alpha,
...                                 fit_prior=fit_prior)
...             clf.fit(X_train, Y_train)
...             prediction_prob = clf.predict_proba(X_test)
...             pos_prob = prediction_prob[:, 1]
...             auc = roc_auc_score(Y_test, pos_prob)
...             auc_record[alpha][fit_prior] = auc +
...                 auc_record[alpha].get(fit_prior, 0.0)
```

Finally, we present the results, as follows:

```
>>> for smoothing, smoothing_record in auc_record.items():
...     for fit_prior, auc in smoothing_record.items():
...         print(f'{smoothing} {fit_prior} {auc/k:.5f}')
smoothing  fit prior  auc
1          True    0.65647
1          False   0.65708
2          True    0.65795
2          False   0.65823
3          True    0.65740
3          False   0.65801
4          True    0.65808
4          False   0.65795
5          True    0.65814
5          False   0.65694
6          True    0.65663
6          False   0.65719
```

The `(2, False)` set enables the best averaged AUC, at `0.65823`.

Finally, we retrain the model with the best set of hyperparameters `(2, False)` and compute the AUC:

```
>>> clf = MultinomialNB(alpha=2.0, fit_prior=False)
>>> clf.fit(X_train, Y_train)
>>> pos_prob = clf.predict_proba(X_test)[:, 1]
>>> print('AUC with the best model:', roc_auc_score(Y_test,
...         pos_prob))
AUC with the best model: 0.6862056720417091
```

An AUC of `0.686` is achieved with the fine-tuned model. In general, tweaking model hyperparameters using cross-validation is one of the most effective ways to boost learning performance and reduce overfitting.

Summary

In this chapter, you learned the fundamental and important concepts of machine learning classification, including types of classification, classification performance evaluation, cross-validation, and model tuning. You also learned about the simple, yet powerful, classifier Naïve Bayes. We went in depth through the mechanics and implementations of Naïve Bayes with a couple of examples, the most important one being the movie recommendation project.

Binary classification was the main talking point of this chapter, and multiclass classification will be the subject of the next chapter. Specifically, we will talk about SVMs for image classification.

Exercise

1. As mentioned earlier, we extracted user-movie relationships only from the movie rating data where most ratings are unknown. Can you also utilize data from the files `movies.dat` and `users.dat`?
2. Practice makes perfect—another great project to deepen your understanding could be heart disease classification. The dataset can be downloaded directly at <https://www.kaggle.com/ronitf/heart-disease-uci>, or from the original page at <https://archive.ics.uci.edu/ml/datasets/Heart+Disease>.
3. Don't forget to fine-tune the model you obtained from Exercise 2 using the techniques you learned in this chapter. What is the best AUC it achieves?

References

To acknowledge the use of the MovieLens dataset in this chapter, I would like to cite the following paper:

F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: *History and Context*. **ACM Transactions on Interactive Intelligent Systems (TiiS)** 5, 4, Article 19 (December 2015), 19 pages.
DOI=<http://dx.doi.org/10.1145/2827872>

Recognizing Faces with Support Vector Machine

In the previous chapter, we built a movie recommendation system with Naïve Bayes. This chapter continues our journey of supervised learning and classification. Specifically, we will be focusing on multiclass classification and **support vector machine (SVM)** classifiers. SVM is one of the most popular algorithms when it comes to high-dimensional spaces. The goal of the algorithm is to find a decision boundary in order to separate data from different classes. We will be discussing in detail how that works. Also, we will be implementing the algorithm with scikit-learn, and applying it to solve various real-life problems, including our main project of face recognition, along with fetal state categorization in cardiotocography and breast cancer prediction. A dimensionality reduction technique called **principal component analysis**, which boosts the performance of the image classifier, will also be covered in the chapter.

This chapter explores the following topics:

- The mechanics of SVM explained through different scenarios
- The implementations of SVM with scikit-learn
- Multiclass classification strategies
- SVM with kernel methods
- How to choose between linear and Gaussian kernels
- Face recognition with SVM
- Principal component analysis
- Tuning with grid search and cross-validation

- Fetal state categorization using SVM with a non-linear kernel

Finding the separating boundary with SVM

Now that you have been introduced to a powerful yet simple classifier, Naïve Bayes, we will continue with another great classifier, SVM, which is effective in cases with high-dimensional spaces or where the number of dimensions is greater than the number of samples.

In machine learning classification, SVM finds an optimal hyperplane that best segregates observations from different classes. A **hyperplane** is a plane of $n - 1$ dimensions that separates the n -dimensional feature space of the observations into two spaces. For example, the hyperplane in a two-dimensional feature space is a line, and in a three-dimensional feature space the hyperplane is a surface. The optimal hyperplane is picked so that the distance from its nearest points in each space to itself is maximized. And these nearest points are the so-called **support vectors**. The following toy example demonstrates what support vectors and a separating hyperplane (along with the distance margin, which I will explain later) look like in a binary classification case:

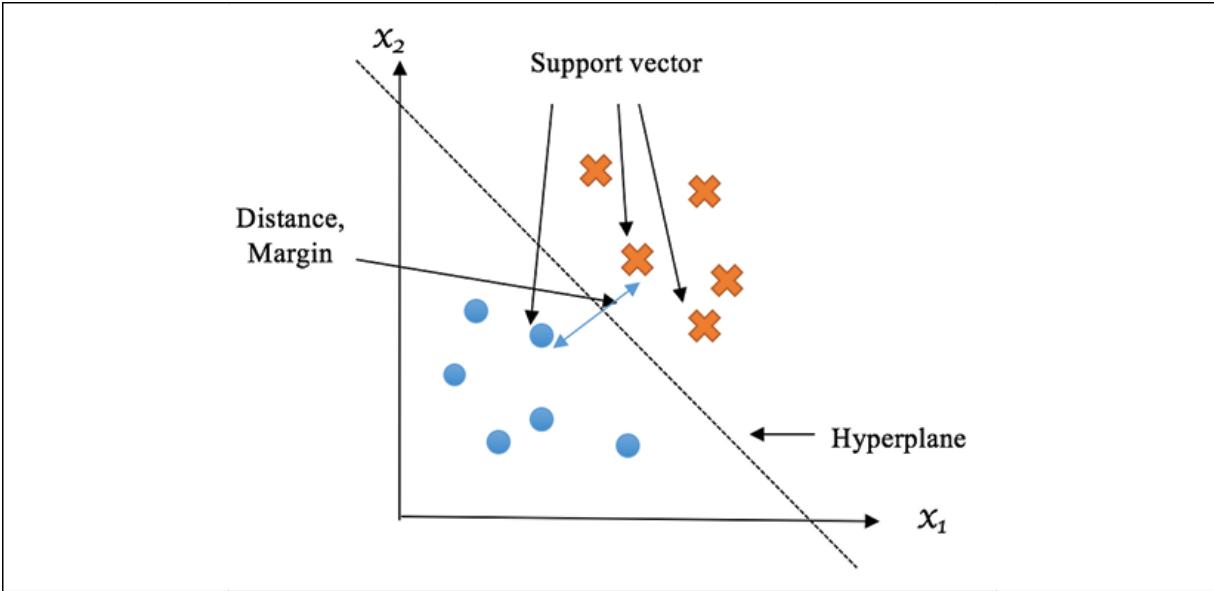


Figure 3.1: Example of support vectors and a hyperplane in binary classification

The ultimate goal of SVM is to find an optimal hyperplane, but the burning question is "how can we find this optimal hyperplane?" You will get the answer as we explore the following scenarios. It's not as hard as you may think. The first thing we will look at is how to find a hyperplane.

Scenario 1 – identifying a separating hyperplane

First, you need to understand what qualifies as a separating hyperplane. In the following example, hyperplane C is the only correct one, as it successfully segregates observations by their labels, while hyperplanes A and B fail:

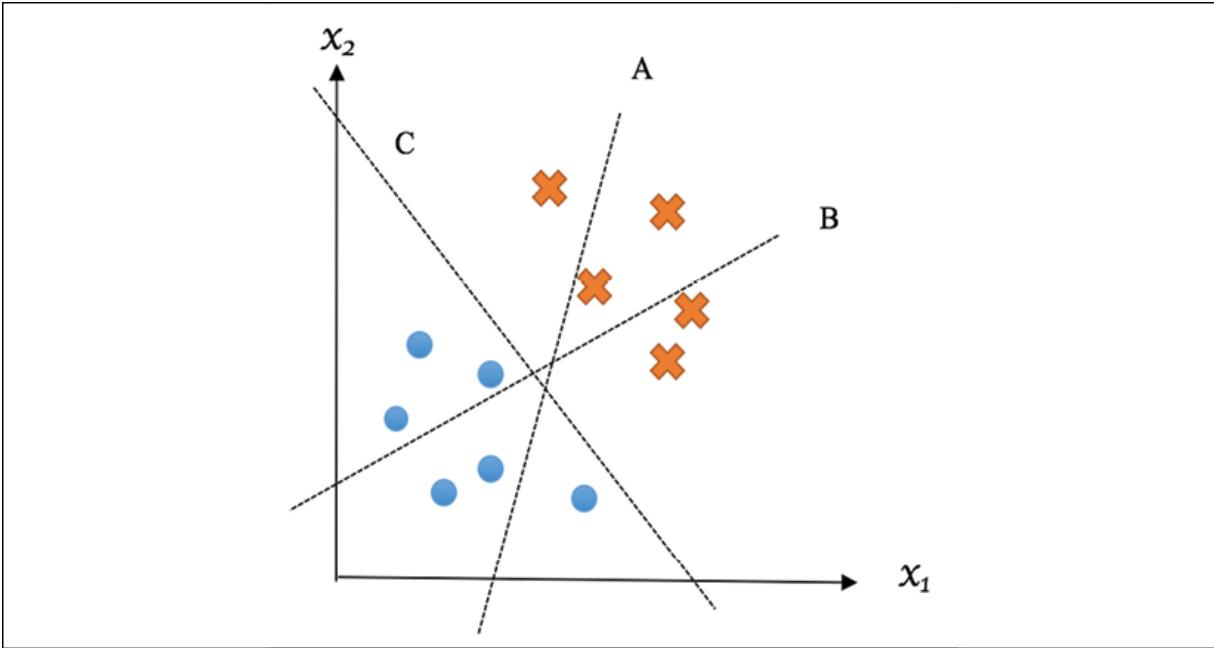


Figure 3.2: Example of qualified and unqualified hyperplanes

This is an easy observation. Let's express a separating hyperplane in a formal or mathematical way next.

In a two-dimensional space, a line can be defined by a slope vector w (represented as a two-dimensional vector), and an intercept b . Similarly, in a space of n dimensions, a hyperplane can be defined by an n -dimensional vector w , and an intercept b . Any data point x on the hyperplane satisfies $wx + b = 0$. A hyperplane is a separating hyperplane if the following conditions are satisfied:

- For any data point x from one class, it satisfies $wx + b > 0$
- For any data point x from another class, it satisfies $wx + b < 0$

However, there can be countless possible solutions for w and b . You can move or rotate hyperplane C to certain extents and it will still remain a separating hyperplane. Next, you will learn how to identify the best hyperplane among various possible separating hyperplanes.

Scenario 2 – determining the optimal hyperplane

Look at the following example: hyperplane C is the preferred one as it enables the maximum sum of the distance between the nearest data point on the positive side and itself and the distance between the nearest data point on the negative side and itself:

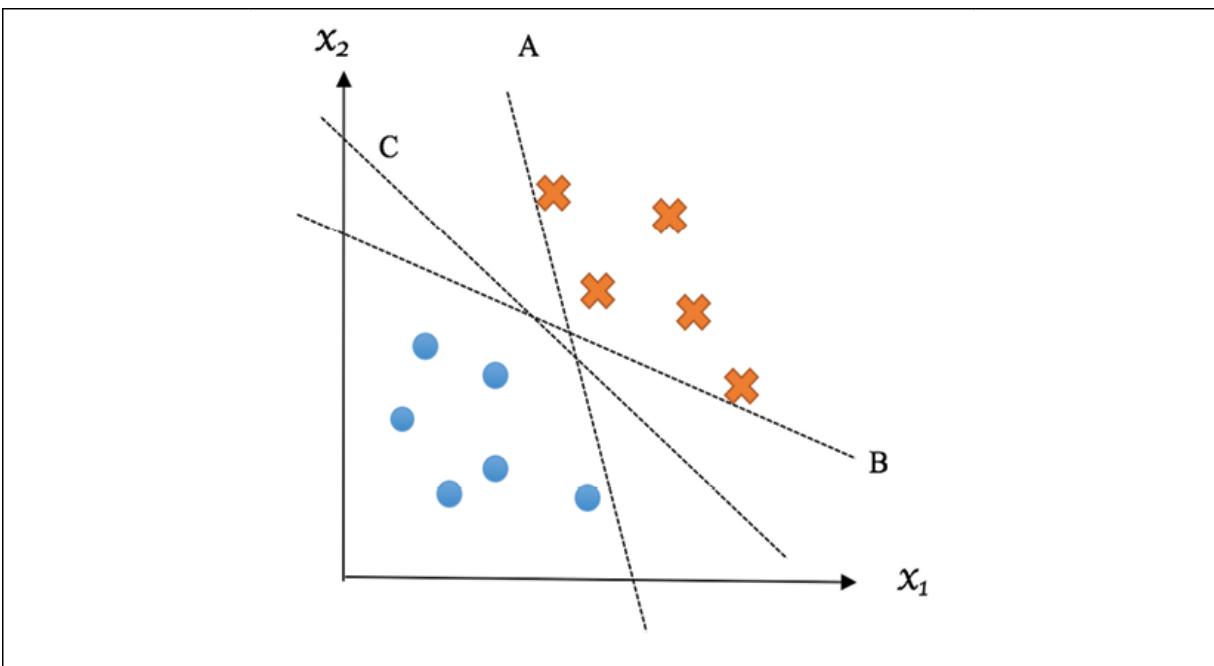


Figure 3.3: An example of optimal and suboptimal hyperplanes

The nearest point(s) on the positive side can constitute a hyperplane parallel to the decision hyperplane, which we call a **positive hyperplane**; on the other hand, the nearest point(s) on the negative side can constitute the **negative hyperplane**. The perpendicular distance between the positive and negative hyperplanes is called the **margin**, the value of which equates to the sum of the two aforementioned distances. A **decision** hyperplane is deemed **optimal** if the margin is maximized.

The optimal (also called **maximum-margin**) hyperplane and the distance margins for a trained SVM model are illustrated in the following diagram.

Again, samples on the margin (two from one class, and one from another class, as shown) are the so-called **support vectors**:

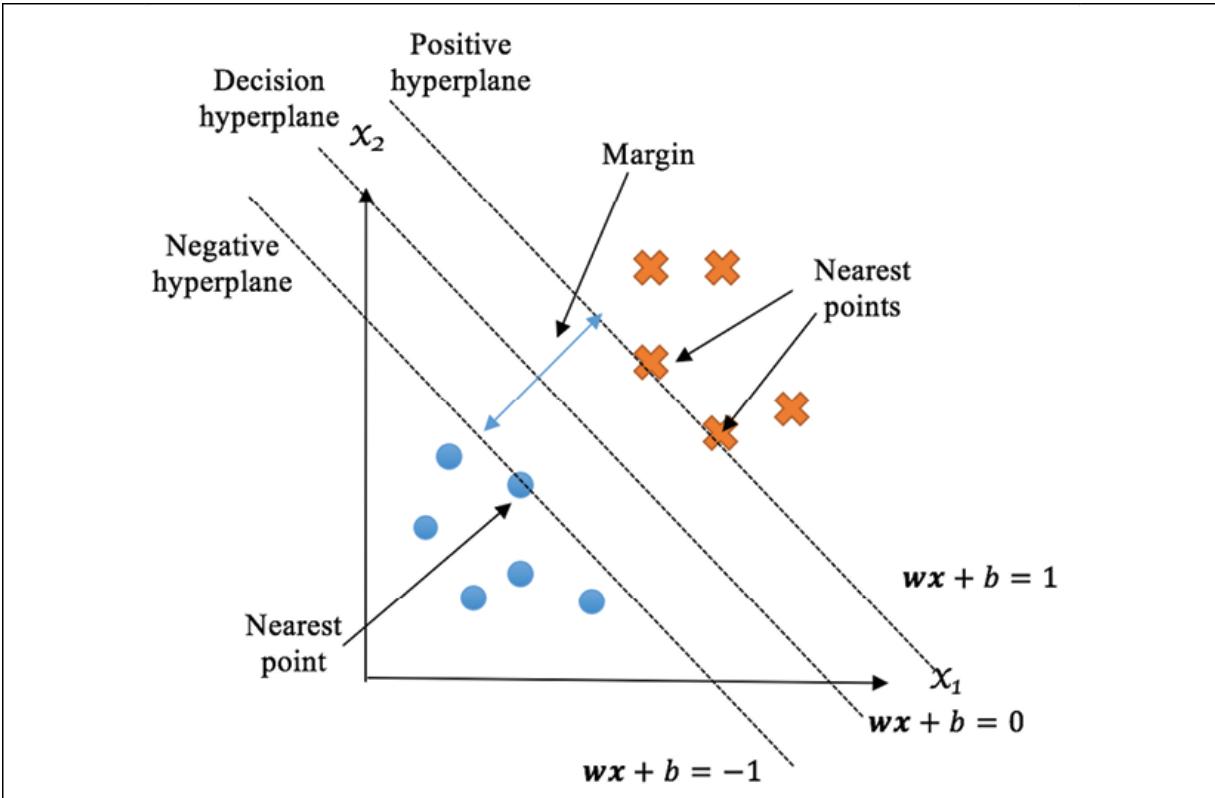


Figure 3.4: An example of an optimal hyperplane and distance margins

We can interpret it in a mathematical way by first describing the positive and negative hyperplanes as follows:

$$w\mathbf{x}^{(p)} + b = 1$$

$$w\mathbf{x}^{(n)} + b = -1$$

Here, $\mathbf{x}^{(p)}$ is a data point on the positive hyperplane, and $\mathbf{x}^{(n)}$ is a data point on the negative hyperplane.

The distance between a point $\mathbf{x}^{(p)}$ and the decision hyperplane can be calculated as follows:

$$\frac{wx^{(p)} + b}{||w||} = \frac{1}{||w||}$$

Similarly, the distance between a point $x^{(n)}$ and the decision hyperplane is as follows:

$$\frac{wx^{(n)} + b}{||w||} = \frac{1}{||w||}$$

$\frac{2}{||w||}$

So, the margin becomes $\frac{2}{||w||}$. As a result, we need to minimize $||w||$ in order to maximize the margin. Importantly, to comply with the fact that the support vectors on the positive and negative hyperplanes are the nearest data points to the decision hyperplane, we add a condition that no data point falls between the positive and negative hyperplanes:

$$wx^{(i)} + b \geq 1, \text{ if } y^{(i)} = 1$$

$$wx^{(i)} + b \leq -1, \text{ if } y^{(i)} = -1$$

Here, $(x^{(i)}, y^{(i)})$ is an observation. This can be combined further into the following:

$$y^{(i)}(wx^{(i)} + b) \geq 1$$

To summarize, w and b , which determine the SVM decision hyperplane, are trained and solved by the following optimization problem:

- Minimizing $||w||$
- Subject to $y^{(i)}(wx^{(i)} + b) \geq 1$, for a training set of $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$, ..., $(x^{(i)}, y^{(i)})$, ..., $(x^{(m)}, y^{(m)})$

To solve this optimization problem, we need to resort to quadratic programming techniques, which are beyond the scope of our learning journey. Therefore, we will not cover the computation methods in detail and instead will implement the classifier using

the `svc` and `LinearSVC` modules from scikit-learn, which are realized respectively based on `libsvm` (<https://www.csie.ntu.edu.tw/~cjlin/libsvm/>) and `liblinear` (<https://www.csie.ntu.edu.tw/~cjlin/liblinear/>), two popular open-source SVM machine learning libraries. But it is always valuable to understand the concepts of computing SVM.



Shai Shalev-Shwartz et al. "Pegasos: Primal estimated sub-gradient solver for SVM" (Mathematical Programming, March 2011, volume 127, issue 1, pp. 3-30), and Cho-Jui Hsieh et al. "A dual coordinate descent method for large-scale linear SVM" (Proceedings of the 25th international conference on machine learning, pp 408-415) are great learning materials. They cover two modern approaches, sub-gradient descent and coordinate descent.

The learned model parameters w and b are then used to classify a new sample x' based on the following conditions:

$$y' = \begin{cases} 1, & \text{if } \mathbf{w}\mathbf{x}' + b > 0 \\ -1, & \text{if } \mathbf{w}\mathbf{x}' + b < 0 \end{cases}$$

Moreover, $||\mathbf{w}\mathbf{x}' + b||$ can be portrayed as the distance from the data point x' to the decision hyperplane, and also interpreted as the confidence of prediction: the higher the value, the further away the data point is from the decision boundary, hence the higher prediction certainty.

Although you might be eager to implement the SVM algorithm, let's take a step back and look at a common scenario where data points are not linearly separable in a strict way. Try to find a separating hyperplane in the following example:

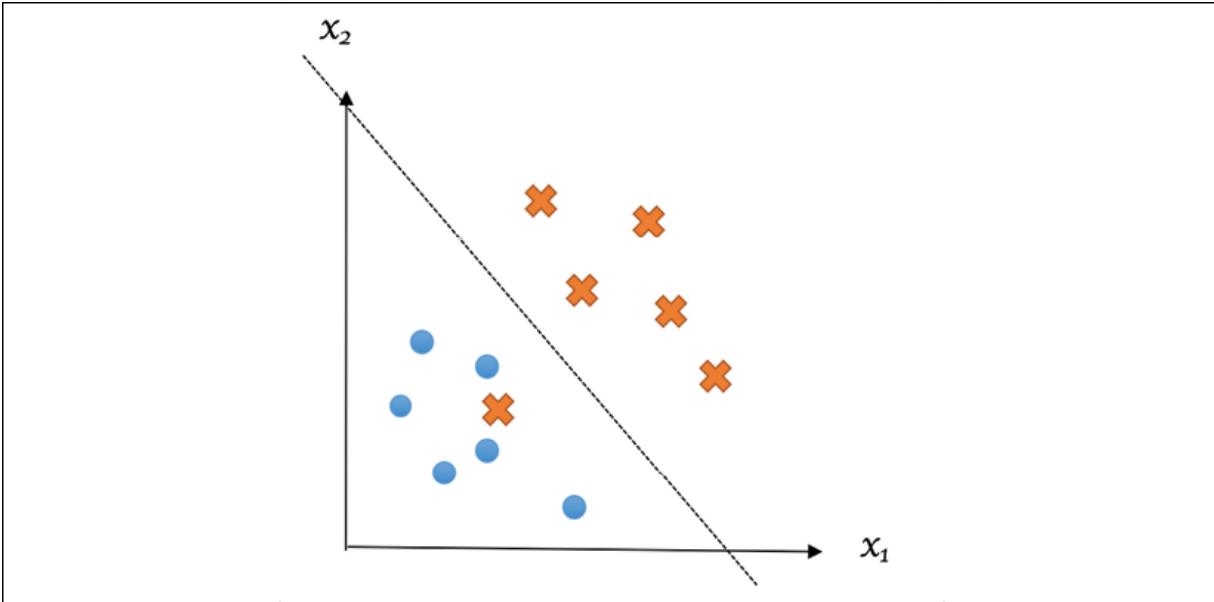


Figure 3.5: An example of data points that are not strictly linearly separable

Scenario 3 – handling outliers

How can we deal with cases where it is impossible to strictly linearly segregate a set of observations containing outliers? We can actually allow the misclassification of such outliers and try to minimize the error introduced. The misclassification error $\zeta^{(i)}$ (also called **hinge loss**) for a sample $x^{(i)}$ can be expressed as follows:

$$\zeta^{(i)} = \begin{cases} 1 - y^{(i)}(\mathbf{w}x^{(i)} + b), & \text{if misclassified} \\ 0 & , \text{otherwise} \end{cases}$$

Together with the ultimate term $\|\mathbf{w}\|$ that we want to reduce, the final objective value we want to minimize becomes the following:

$$\|\mathbf{w}\| + C \frac{\sum_{i=1}^m \zeta^{(i)}}{m}$$

As regards a training set of m samples $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}) \dots (x^{(i)}, y^{(i)}) \dots, (x^{(m)}, y^{(m)})$, where the hyperparameter C controls the trade-off between the two terms:

- If a large value of C is chosen, the penalty for misclassification becomes relatively high. This means the rule of thumb of data segregation becomes stricter and the model might be prone to overfitting, since few mistakes are allowed during training. An SVM model with a large C has a low bias, but it might suffer from high variance.
- Conversely, if the value of C is sufficiently small, the influence of misclassification becomes fairly low. This model allows more misclassified data points than a model with a large C . Thus, data separation becomes less strict. Such a model has low variance, but it might be compromised by high bias.

A comparison between a large and small C is shown in the following diagram:

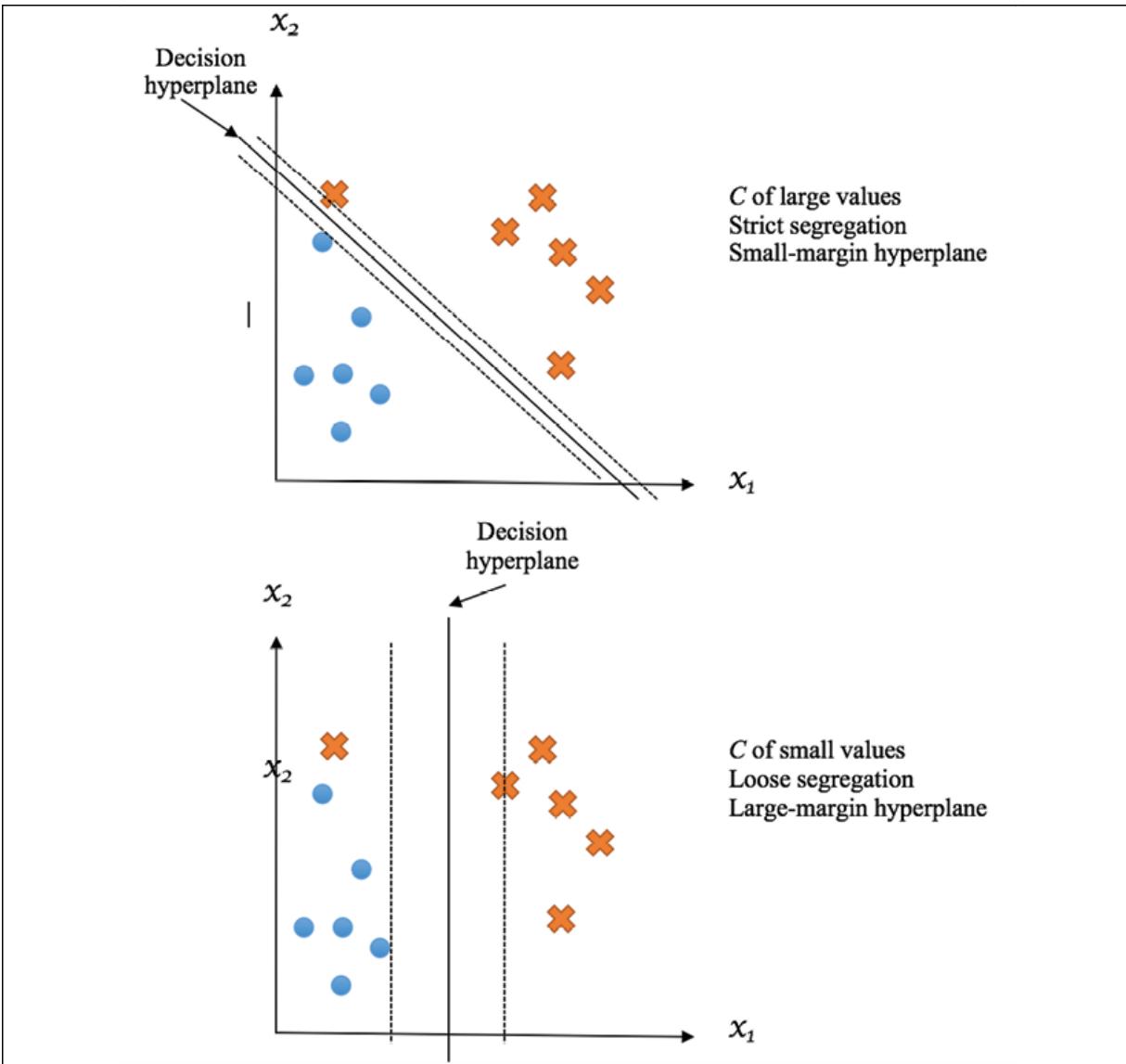


Figure 3.6: How the value of C affects the strictness of segregation and the margin

The parameter C determines the balance between bias and variance. It can be fine-tuned with cross-validation, which we will practice shortly.

Implementing SVM

We have largely covered the fundamentals of the SVM classifier. Now, let's apply it right away to an easy binary classification dataset. We will use the classic breast cancer Wisconsin dataset (https://scikit-learn.org/stable/datasets/toy_dataset.html)

[learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_breast_cancer.html)) from scikit-learn.

Let's take a look at the following steps:

1. We first load the dataset and do some basic analysis, as follows:

```
>>> from sklearn.datasets import load_breast_cancer
>>> cancer_data = load_breast_cancer()
>>> X = cancer_data.data
>>> Y = cancer_data.target
>>> print('Input data size :', X.shape)
Input data size : (569, 30)
>>> print('Output data size :', Y.shape)
Output data size : (569,)
>>> print('Label names:', cancer_data.target_names)
Label names: ['malignant' 'benign']
>>> n_pos = (Y == 1).sum()
>>> n_neg = (Y == 0).sum()
>>> print(f'{n_pos} positive samples and {n_neg} negative s
357 positive samples and 212 negative samples.
```

As you can see, the dataset has 569 samples with 30 features; its label is binary, and 63% of samples are positive (benign). Again, always check whether classes are imbalanced before trying to solve any classification problem. In this case, they are relatively balanced.

2. Next, we split the data into training and testing sets:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=42)
```

For reproducibility, don't forget to specify a random seed.

3. We can now apply the SVM classifier to the data. We first initialize an `svc` model with the `kernel` parameter set to `linear` (I will explain what kernel means in the next section) and the penalty hyperparameter `c` set to the default value, `1.0`:

```
>>> from sklearn.svm import SVC
>>> clf = SVC(kernel='linear', C=1.0, random_state=42)
```

4. We then fit our model on the training set as follows:

```
>>> clf.fit(X_train, Y_train)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3,
    gamma='auto_deprecated', kernel='linear', max_iter=-1,
    probability=False, random_state=42, shrinking=True,
    tol=0.001, verbose=False)
```

5. And we predict on the testing set with the trained model and obtain the prediction accuracy directly:

```
>>> accuracy = clf.score(X_test, Y_test)
>>> print(f'The accuracy is: {accuracy*100:.1f}%')
The accuracy is: 95.8%
```

Our first SVM model works just great, achieving an accuracy of 95.8%. How about dealing with more than two topics? How does SVM handle multiclass classification?

Scenario 4 – dealing with more than two classes

SVM and many other classifiers can be applied to cases with more than two classes. There are two typical approaches we can take, **one-vs-rest** (also called **one-vs-all**) and **one-vs-one**.

In the one-vs-rest setting, for a K -class problem, we construct K different binary SVM classifiers. For the k^{th} classifier, it treats the k^{th} class as the positive case and the remaining $K-1$ classes as the negative case as a whole; the hyperplane denoted as (w_k, b_k) is trained to separate these two cases. To predict the class of a new sample, x' , it compares the resulting predictions $w_k x' + b_k$ from K individual classifiers from 1 to k . As we discussed in the previous section, the larger value of $w_k x' + b_k$ means higher confidence that x' belongs to the positive case. Therefore, it assigns x' to the class i where $w_i x' + b_i$ has the largest value among all prediction results:

$$y' = \operatorname{argmax}_i (w_i x' + b_i)$$

The following diagram shows how the one-vs-rest strategy works in a three-class case:

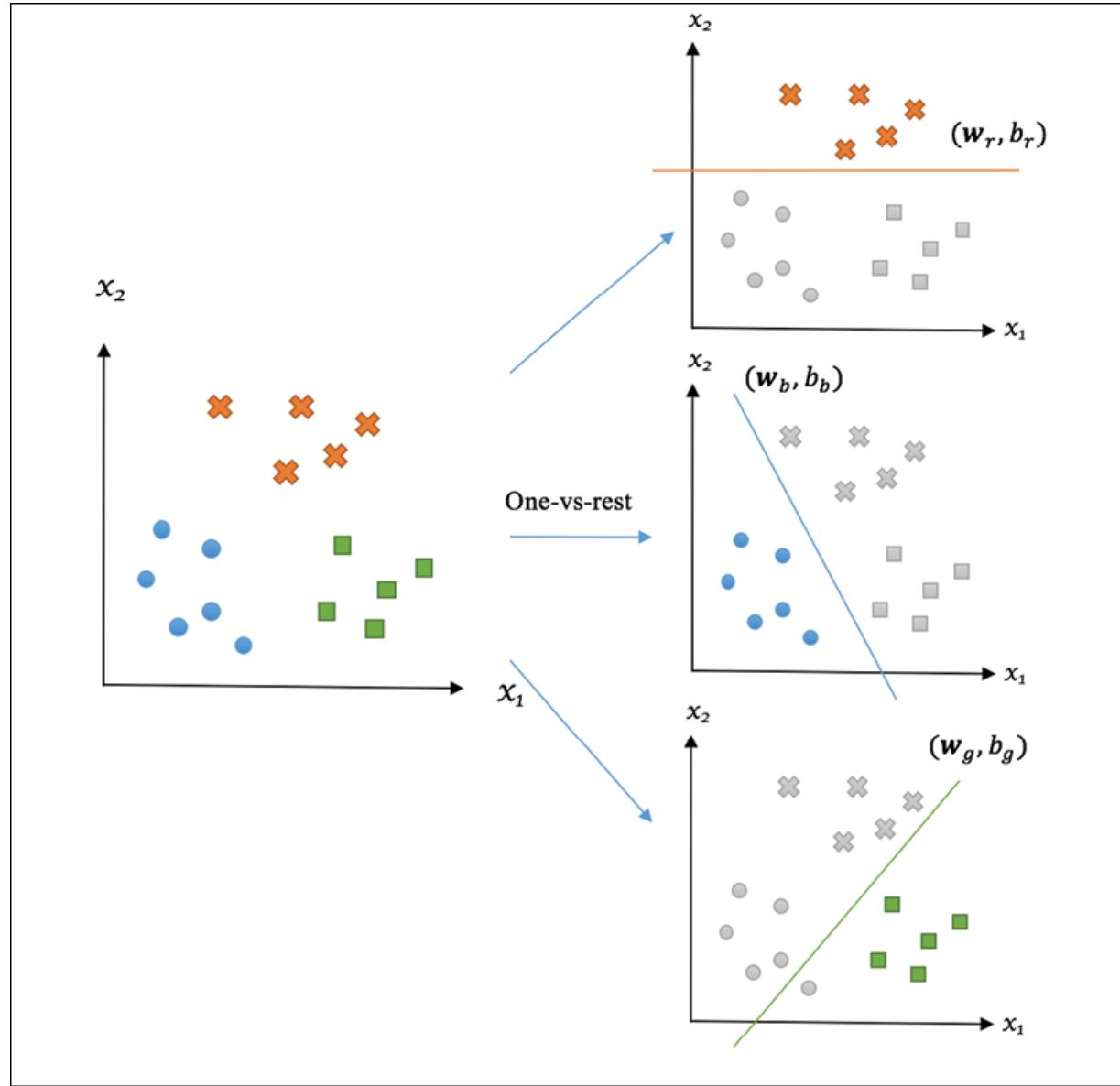


Figure 3.7: An example of three-class classification using the one-vs-rest strategy

For instance, if we have the following (r , b , and g denote the red, blue, and green classes, respectively):

$$w_r x' + b_r = 0.78$$

$$w_b x' + b_b = 0.35$$

$$w_g x' + b_g = -0.64$$

we can say x' belongs to the red class since $0.78 > 0.35 > -0.64$. If we have the following:

$$w_r x' + b_r = -0.78$$

$$w_b x' + b_b = -0.35$$

$$w_g x' + b_g = -0.64$$

then we can determine that x' belongs to the blue class regardless of the sign since $-0.35 > -0.64 > -0.78$.

In the one-vs-one strategy, we conduct a pairwise comparison by building a set of SVM classifiers that can distinguish data points from each pair of classes. This will result in $\frac{K(K - 1)}{2}$ different classifiers.

For a classifier associated with classes i and j , the hyperplane denoted as (w_{ij}, b_{ij}) is trained only on the basis of observations from i (can be viewed as a positive case) and j (can be viewed as a negative case); it then assigns the class, either i or j , to a new sample, x' , based on the sign of $w_{ij}x' + b_{ij}$. Finally, the class with the highest number of assignments is considered the predicting result of x' . The winner is the class that gets the most votes.

The following diagram shows how the one-vs-one strategy works in a three-class case:

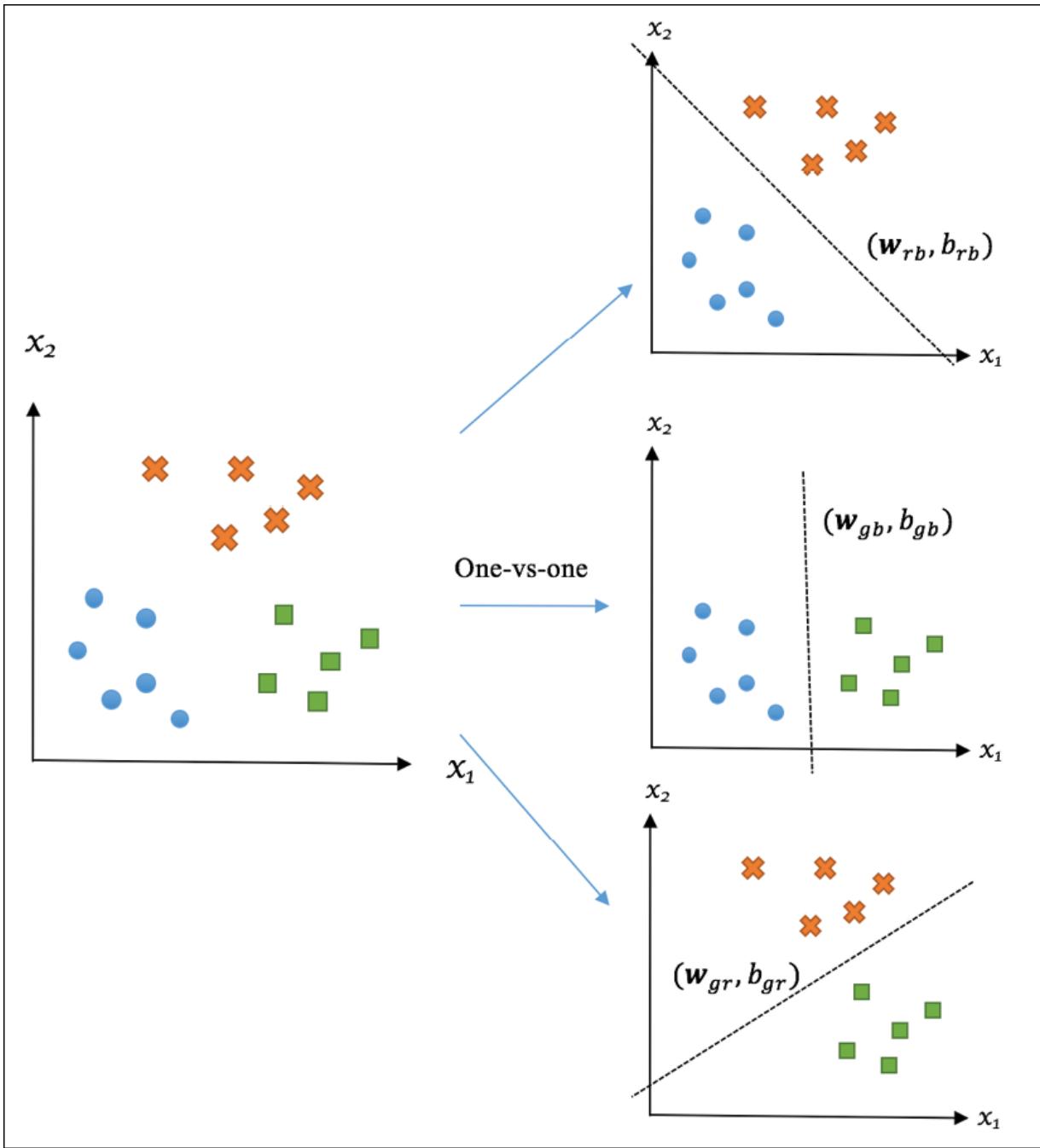


Figure 3.8: An example of three-class classification using the one-vs-one strategy

In general, an SVM classifier with a one-vs-rest setting and a classifier with a one-vs-one setting perform comparably in terms of accuracy. The choice between these two strategies is largely computational.

$$\frac{K(K - 1)}{2}$$

Although one-vs-one requires more classifiers, $\frac{K(K - 1)}{2}$, than one-vs-rest (K), each pairwise classifier only needs to learn on a small subset of data, as opposed to the entire set in the one-vs-rest setting. As a result, training an SVM model in the one-vs-one setting is generally more memory efficient and less computationally expensive, and hence is preferable for practical use, as argued in Chih-Wei Hsu and Chih-Jen Lin's *A comparison of methods for multiclass support vector machines* (IEEE Transactions on Neural Networks, March 2002, Volume 13, pp. 415-425).

In scikit-learn, classifiers handle multiclass cases internally, and we do not need to explicitly write any additional code to enable this. You can see how simple it is in the wine classification example (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_wine.html#sklearn.datasets.load_wine) with three classes, as follows:

1. We first load the dataset and do some basic analysis, as follows:

```
>>> from sklearn.datasets import load_wine
>>> wine_data = load_wine()
>>> X = wine_data.data
>>> Y = wine_data.target
>>> print('Input data size :', X.shape)
Input data size : (178, 13)
>>> print('Output data size :', Y.shape)
Output data size : (178,)
>>> print('Label names:', wine_data.target_names)
Label names: ['class_0' 'class_1' 'class_2']
>>> n_class0 = (Y == 0).sum()
>>> n_class1 = (Y == 1).sum()
>>> n_class2 = (Y == 2).sum()
>>> print(f'{n_class0} class0 samples,\n{n_class1} class1 s
59 class0 samples,
71 class1 samples,
48 class2 samples.
```

As you can see, the dataset has 178 samples with 13 features; its label has three possible values taking up 33%, 40%, and 27%, respectively.

2. Next, we split the data into training and testing sets:

```
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, `
```

3. We can now apply the SVM classifier to the data. We first initialize an `svc` model and fit it against the training set:

```
>>> clf = SVC(kernel='linear', C=1.0, random_state=42)
>>> clf.fit(X_train, Y_train)
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
     decision_function_shape='ovr', degree=3,
     gamma='auto_deprecated', kernel='linear', max_iter=-1,
     probability=False, random_state=42, shrinking=True,
     tol=0.001, verbose=False)
```

In an `svc` model, multiclass support is implicitly handled according to the one-vs-one scheme.

4. Next, we predict on the testing set with the trained model and obtain the prediction accuracy directly:

```
>>> accuracy = clf.score(X_test, Y_test)
>>> print(f'The accuracy is: {accuracy*100:.1f}%')
The accuracy is: 97.8%
```

Our SVM model also works well in the multiclass case, achieving an accuracy of 97.8%.

5. We also check how it performs for individual classes:

```
>>> from sklearn.metrics import classification_report
>>> pred = clf.predict(X_test)
>>> print(classification_report(Y_test, pred))
      precision    recall  f1-score   support
          0       1.00     1.00     1.00      15
          1       1.00     0.94     0.97      18
          2       0.92     1.00     0.96      12
    micro avg       0.98     0.98     0.98      45
  macro avg       0.97     0.98     0.98      45
weighted avg       0.98     0.98     0.98      45
```

It looks excellent! Is the example too easy? Maybe. What do we do in tricky cases? Of course, we could tweak the values of the `kernel` and `c` hyperparameters. As discussed, the factor `c` controls the strictness of separation, and it can be tuned to achieve the best trade-off between bias

and variance. How about the kernel? What does it mean and what are the alternatives to a `linear` kernel?

In the next section, we will answer those two questions we just raised. You will see how the kernel trick makes SVM so powerful.

Scenario 5 – solving linearly non-separable problems with kernels

The hyperplanes we have found so far are linear, for instance, a line in a two-dimensional feature space, or a surface in a three-dimensional one. However, in the following example, we are not able to find a linear hyperplane that can separate two classes:

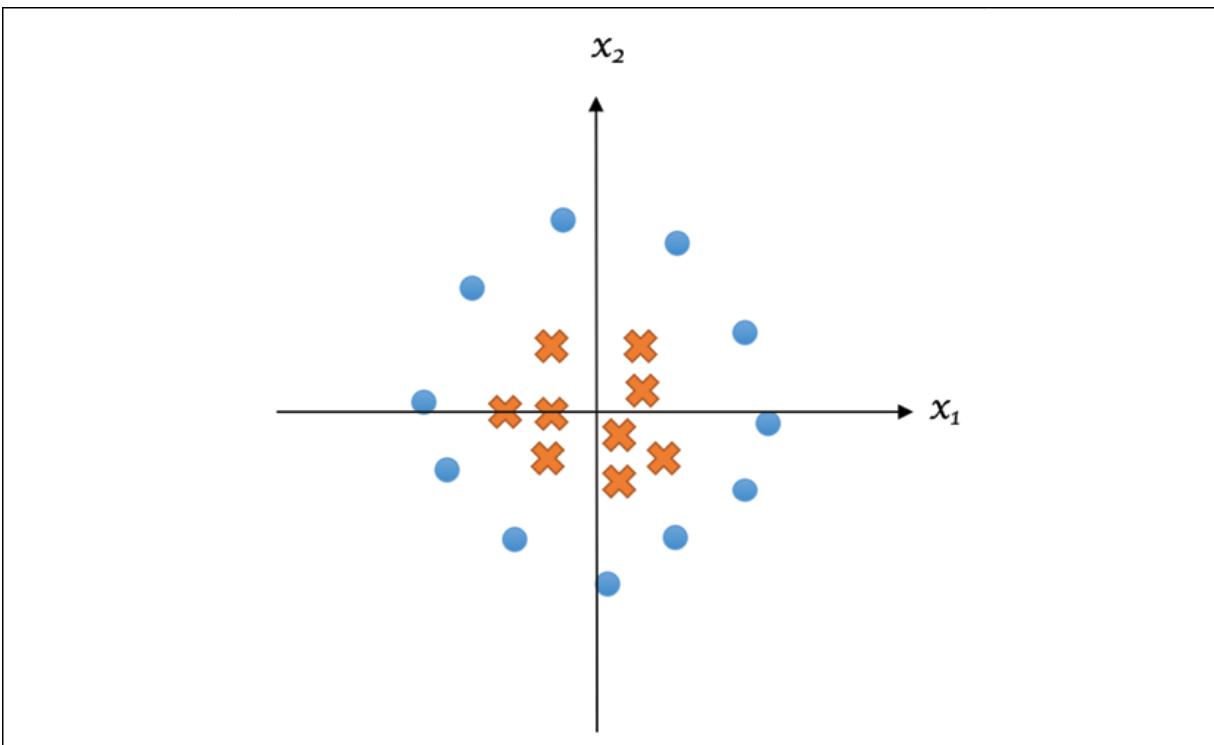


Figure 3.9: The linearly non-separable case

Intuitively, we observe that data points from one class are closer to the origin than those from another class. The distance to the origin provides distinguishable information. So we add a new feature, $z = (x_1^2 + x_2^2)^2$,

and transform the original two-dimensional space into a three-dimensional one. In the new space, as displayed in the following diagram, we can find a surface hyperplane separating the data, or a line in the two-dimension view. With the additional feature, the dataset becomes linearly separable in the higher dimensional space, (x_1, x_2, z) :

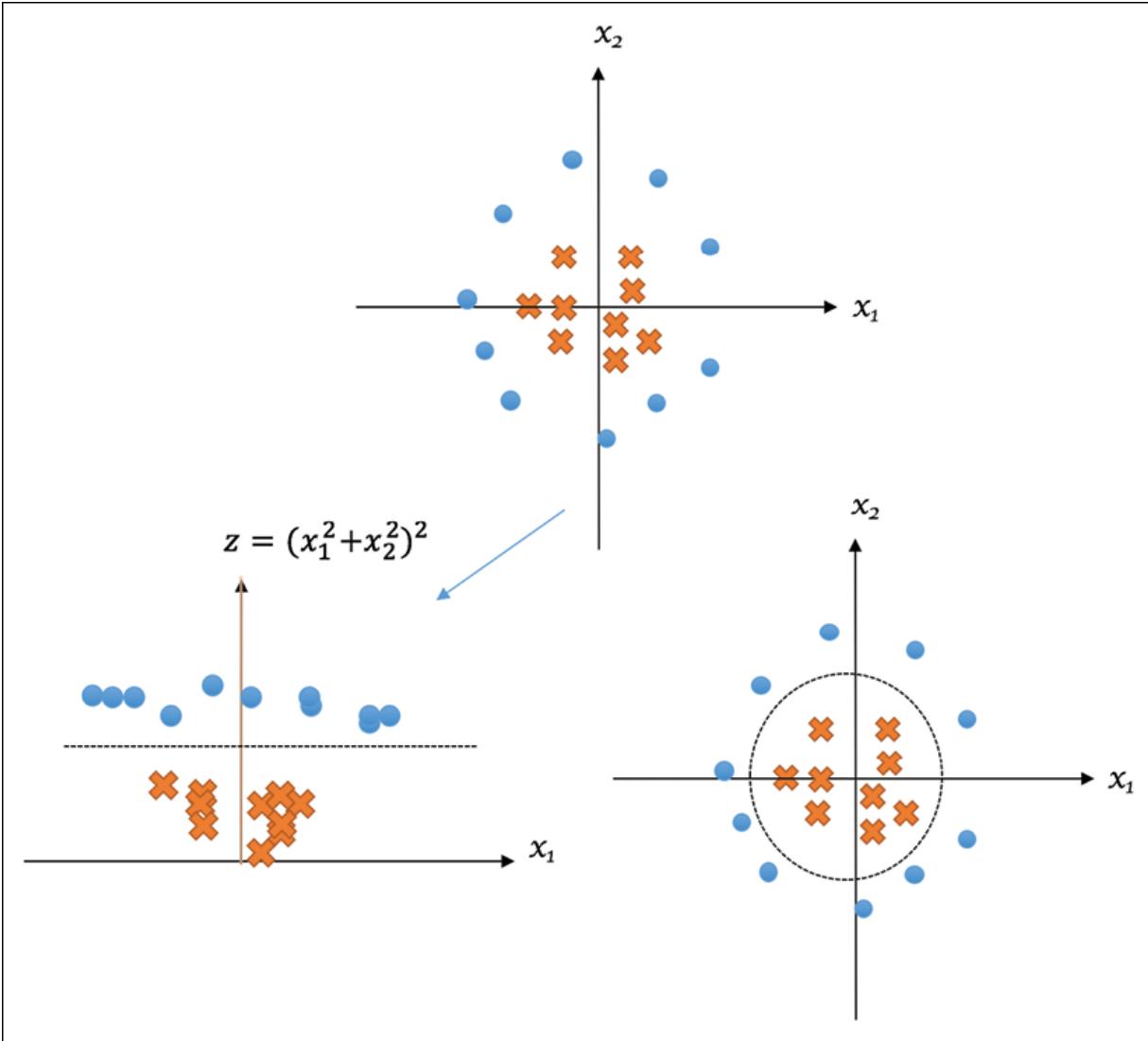


Figure 3.10: Making a non-separable case separable

Based upon similar logic, **SVMs with kernels** were invented to solve non-linear classification problems by converting the original feature space $x^{(i)}$, to a higher dimensional feature space with a transformation function, Φ , such that the transformed dataset $\Phi(x^{(i)})$ is linearly separable.

A linear hyperplane (w_Φ , b_Φ) is then learned using observations $(\Phi(x^{(i)}), y^{(i)})$. For an unknown sample x' , it is first transformed into $\Phi(x')$; the predicted class is determined by $w_\Phi x' + b_\Phi$.

An SVM with kernels enables non-linear separation, but it does not explicitly map each original data point to the high-dimensional space and then perform expensive computation in the new space. Instead, it approaches this in a tricky way.

During the course of solving the SVM quadratic optimization problems, feature vectors $x^{(1)}, x^{(2)}, \dots, x^{(m)}$ are involved only in the form of a pairwise dot product $x^{(i)} \cdot x^{(j)}$, although we will not expand this mathematically in this book. With kernels, the new feature vectors are $\Phi(x^{(1)}), \Phi(x^{(2)}), \dots, \Phi(x^{(m)})$ and their pairwise dot products can be expressed as $\Phi(x^{(i)}) \cdot \Phi(x^{(j)})$. It would be computationally efficient to first implicitly conduct a pairwise operation on two low-dimensional vectors and later map the result to the high-dimensional space. In fact, a function K that satisfies this does exist:

$$K(x^{(i)}, x^{(j)}) = \Phi(x^{(i)}) \cdot \Phi(x^{(j)})$$

The function K is the so-called **kernel function**. With this trick, the transformation Φ becomes implicit, and the non-linear decision boundary can be efficiently learned by simply replacing the term $\Phi(x^{(i)}) \cdot \Phi(x^{(j)})$ with $K(x^{(i)}, x^{(j)})$.

The most popular kernel function is probably the **radial basis function (RBF)** kernel (also called the **Gaussian** kernel), which is defined as follows:

$$K(x^{(i)}, x^{(j)}) = \exp\left(-\frac{\|x^{(i)} - x^{(j)}\|}{2\sigma^2}\right) = \exp(-\gamma \|x^{(i)} - x^{(j)}\|^2)$$

Here, $\gamma = \frac{1}{2\sigma^2}$. In the Gaussian function, the standard deviation σ controls the amount of variation or dispersion allowed: the higher the σ (or the lower the γ), the larger the width of the bell, and the wider the range is that

data points are allowed to spread out over. Therefore, γ as the **kernel coefficient** determines how strictly or generally the kernel function fits the observations. A large γ implies a small variance allowed and a relatively exact fit on the training samples, which might lead to overfitting. On the other hand, a small γ implies a high variance allowed and a loose fit on the training samples, which might cause underfitting.

To illustrate this trade-off, let's apply the RBF kernel with different values to a toy dataset:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> X = np.c_[
    ...     (.3, -.8),
    ...     (-1.5, -1),
    ...     (-1.3, -.8),
    ...     (-1.1, -1.3),
    ...     (-1.2, -.3),
    ...     (-1.3, -.5),
    ...     (-.6, 1.1),
    ...     (-1.4, 2.2),
    ...     (1, 1),
    ...     # positive class
    ...     (1.3, .8),
    ...     (1.2, .5),
    ...     (.2, -2),
    ...     (.5, -2.4),
    ...     (.2, -2.3),
    ...     (0, -2.7),
    ...     (1.3, 2.1)].T
>>> Y = [-1] * 8 + [1] * 8
```

Eight data points are from one class, and eight are from another. We take three values, `1`, `2`, and `4`, for kernel coefficient options as an example:

```
>>> gamma_option = [1, 2, 4]
```

Under each kernel coefficient, we fit an individual SVM classifier and visualize the trained decision boundary:

```
>>> import matplotlib.pyplot as plt
>>> gamma_option = [1, 2, 4]
>>> for i, gamma in enumerate(gamma_option, 1):
...     svm = SVC(kernel='rbf', gamma=gamma)
...     svm.fit(X, Y)
...     plt.scatter(X[:, 0], X[:, 1], c=['b']*8+['r']*8,
...                 zorder=10, cmap=plt.cm.Paired)
...     plt.axis('tight')
...     XX, YY = np.mgrid[-3:3:200j, -3:3:200j]
...     Z = svm.decision_function(np.c_[XX.ravel(), YY.ravel()])
...     Z = Z.reshape(XX.shape)
...     plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
...     plt.contour(XX, YY, Z, colors=['k', 'k', 'k'],
...                 linestyles=['--', '--', '--'], levels=[-.5, 0, .5])
...     plt.title('gamma = %d' % gamma)
...     plt.show()
```

Refer to the following screenshot for the end results:

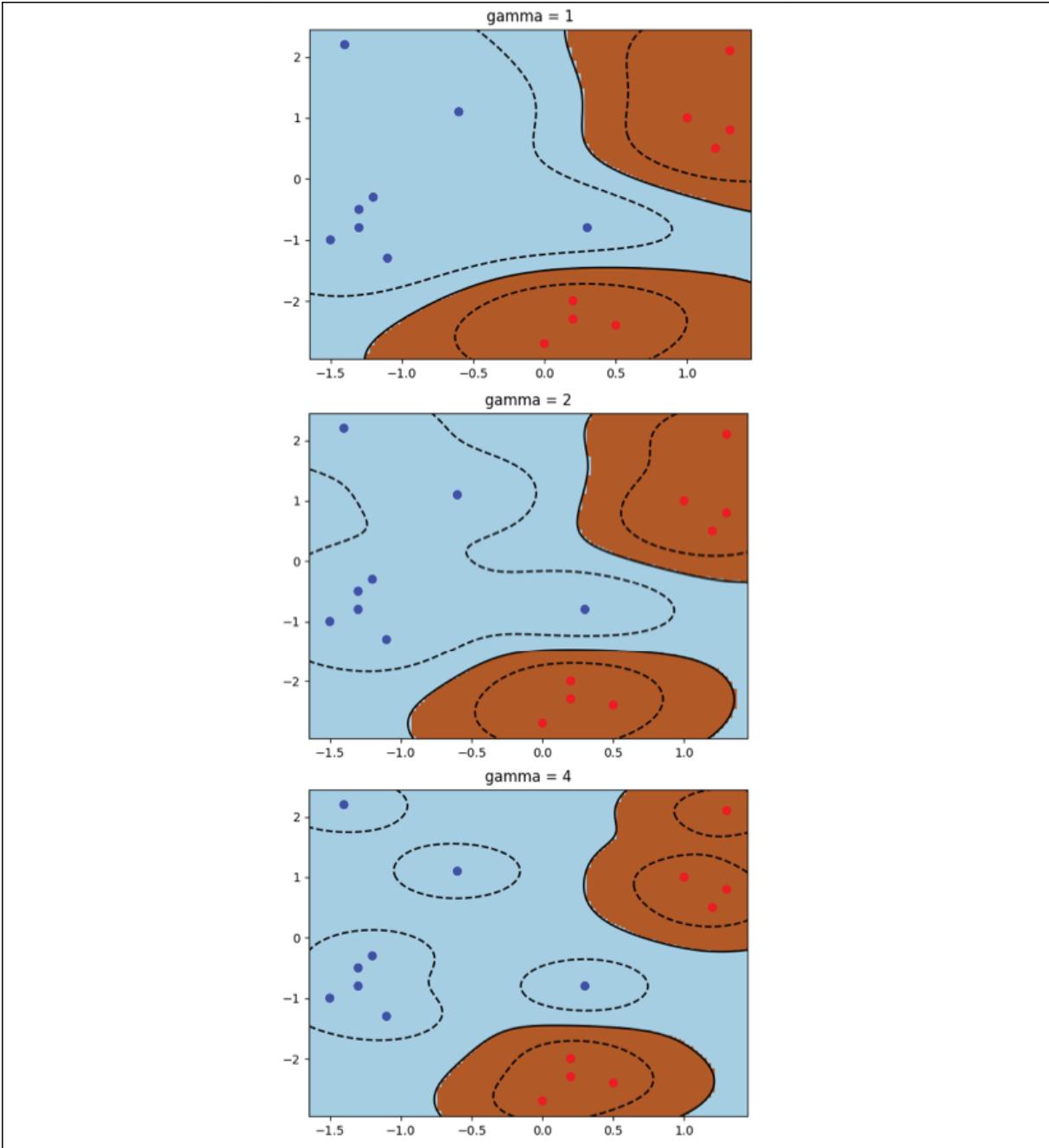


Figure 3.11: The SVM classification decision boundary under different values of γ

We can observe that a larger γ results in narrow regions, which means a stricter fit on the dataset; a smaller γ results in broad regions, which means a loose fit on the dataset. Of course, γ can be fine-tuned through cross-validation to obtain the best performance.

Some other common kernel functions include the **polynomial** kernel

$$K(x^{(i)}, x^{(j)}) = (x^{(i)} \cdot x^{(j)} + \gamma)^d$$

and the **sigmoid** kernel:

$$K(x^{(i)}, x^{(j)}) = \tanh(x^{(i)} \cdot x^{(j)} + \gamma)$$

In the absence of prior knowledge of the distribution, the RBF kernel is usually preferable in practical usage, as there is an additional parameter to tweak in the polynomial kernel (polynomial degree d) and the empirical sigmoid kernel can perform approximately on a par with the RBF, but only under certain parameters. Hence, we come to a debate between the linear (also considered no kernel) and the RBF kernel given a dataset.

Choosing between linear and RBF kernels

Of course, linear separability is the rule of thumb when choosing the right kernel to start with. However, most of the time this is very difficult to identify, unless you have sufficient prior knowledge of the dataset, or its features are of low dimensions (1 to 3).



Some general prior knowledge that is commonly known includes that text data is often linearly separable, while data generated from the XOR function (https://en.wikipedia.org/wiki/XOR_gate) is not.

Now, let's look at the following three scenarios where the linear kernel is favored over RBF.

Scenario 1: Both the number of features and the number of instances are large (more than 10⁴ or 10⁵). Since the dimension of the feature space is high enough, additional features as a result of RBF transformation will not provide a performance improvement, but this will increase the computational expense. Some examples from the UCI machine learning repository are of this type:

- *URL Reputation Dataset*:
<https://archive.ics.uci.edu/ml/datasets/URL+Reputation> (number of instances: 2,396,130; number of features: 3,231,961). This is designed for malicious URL detection based on their lexical and host information.
- *YouTube Multiview Video Games Dataset*:
<https://archive.ics.uci.edu/ml/datasets/YouTube+Multiview+Video+Games+Dataset> (number of instances: 120,000; number of features: 1,000,000). This is designed for topic classification.

Scenario 2: The number of features is noticeably large compared to the number of training samples. Apart from the reasons stated in *scenario 1*, the RBF kernel is significantly more prone to overfitting. Such a scenario occurs, for example, in the following examples:

- *Dorothea Dataset*:
<https://archive.ics.uci.edu/ml/datasets/Dorothea> (number of instances: 1,950; number of features: 100,000). This is designed for drug discovery that classifies chemical compounds as active or inactive according to their structural molecular features.
- *Arcene Dataset*: <https://archive.ics.uci.edu/ml/datasets/Arcene> (number of instances: 900; number of features: 10,000). This represents a mass spectrometry dataset for cancer detection.

Scenario 3: The number of instances is significantly large compared to the number of features. For a dataset of low dimension, the RBF kernel will, in general, boost the performance by mapping it to a higher-dimensional space. However, due to the training complexity, it usually becomes inefficient on a training set with more than 106 or 107 samples. Example datasets include the following:

- *Heterogeneity Activity Recognition Dataset*:
<https://archive.ics.uci.edu/ml/datasets/Heterogeneity+Activity+Recognition> (number of instances:

43,930,257; number of features: 16). This is designed for human activity recognition.

- *HIGGS Dataset:*

<https://archive.ics.uci.edu/ml/datasets/HIGGS>

(number of instances: 11,000,000; number of features: 28). This is designed to distinguish between a signal process producing Higgs bosons or a background process.

Aside from these three scenarios, RBF is ordinarily the first choice.

The rules for choosing between linear and RBF kernels can be summarized as follows:

Scenario	Linear	RBF
Prior knowledge	If linearly separable	If nonlinearly separable
Visualizable data of 1 to 3 dimension(s)	If linearly separable	If nonlinearly separable
Both the number of features and number of instances are large.	First choice	
Features >> Instances	First choice	
Instances >> Features	First choice	
Others		First choice

Table 3.1: Rules for choosing between linear and RBF kernels

Once again, **first choice** means we can **begin with** this option; it does not mean that this is the only option moving forward.

Next, let's take a look at classifying face images.

Classifying face images with SVM

Finally, it is time to build an SVM-based face image classifier using everything you just learned. We will do so in parts, exploring the image dataset.

Exploring the face image dataset

We will use the **Labeled Faces in the Wild (LFW)** people dataset (https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_lfw_people.html) from scikit-learn. It consists of more than 13,000 curated face images of more than 5,000 famous people. Each class has various numbers of image samples.

First, we load the face image data as follows:

```
>>> from sklearn.datasets import fetch_lfw_people
Downloading LFW metadata:c https://ndownloader.figshare.com/file
Downloading LFW metadata: https://ndownloader.figshare.com/files
Downloading LFW metadata: https://ndownloader.figshare.com/files
Downloading LFW data (~200MB): https://ndownloader.figshare.com/
>>> face_data = fetch_lfw_people(min_faces_per_person=80)
```

We only load classes with at least 80 samples so that we will have enough training data. Note that if you run into the problem of `ImportError`: The **Python Imaging Library (PIL)** is required to load data from jpeg files, please install the package `pillow` as follows in the terminal:

```
pip install pillow
```

Next, we take a look at the data we loaded:

```
>>> X = face_data.data
>>> Y = face_data.target
>>> print('Input data size :', X.shape)
Input data size : (1140, 2914)
>>> print('Output data size :', Y.shape)
Output data size : (1140,)
>>> print('Label names:', face_data.target_names)
Label names: ['Colin Powell' 'Donald Rumsfeld' 'George W Bush' ']
```

This five-class dataset contains 1,140 samples and a sample is of 2,914 dimensions. As a good practice, we analyze the label distribution as follows:

```
>>> for i in range(5):
...     print(f'Class {i} has {(Y == i).sum()} samples.')
Class 0 has 236 samples.
Class 1 has 121 samples.
Class 2 has 530 samples.
Class 3 has 109 samples.
Class 4 has 144 samples.
```

The dataset is rather imbalanced. Let's keep this in mind when we build the model.

Now let's plot a few face images:

```
>>> import matplotlib.pyplot as plt
>>>
>>> fig, ax = plt.subplots(3, 4)
>>> for i, axi in enumerate(ax.flat):
...     axi.imshow(face_data.images[i], cmap='bone')
...     axi.set(xticks=[], yticks=[],
...             xlabel=face_data.target_names[face_data.target[i]])
...
>>> plt.show()
```



You will see the following 12 images with their labels:

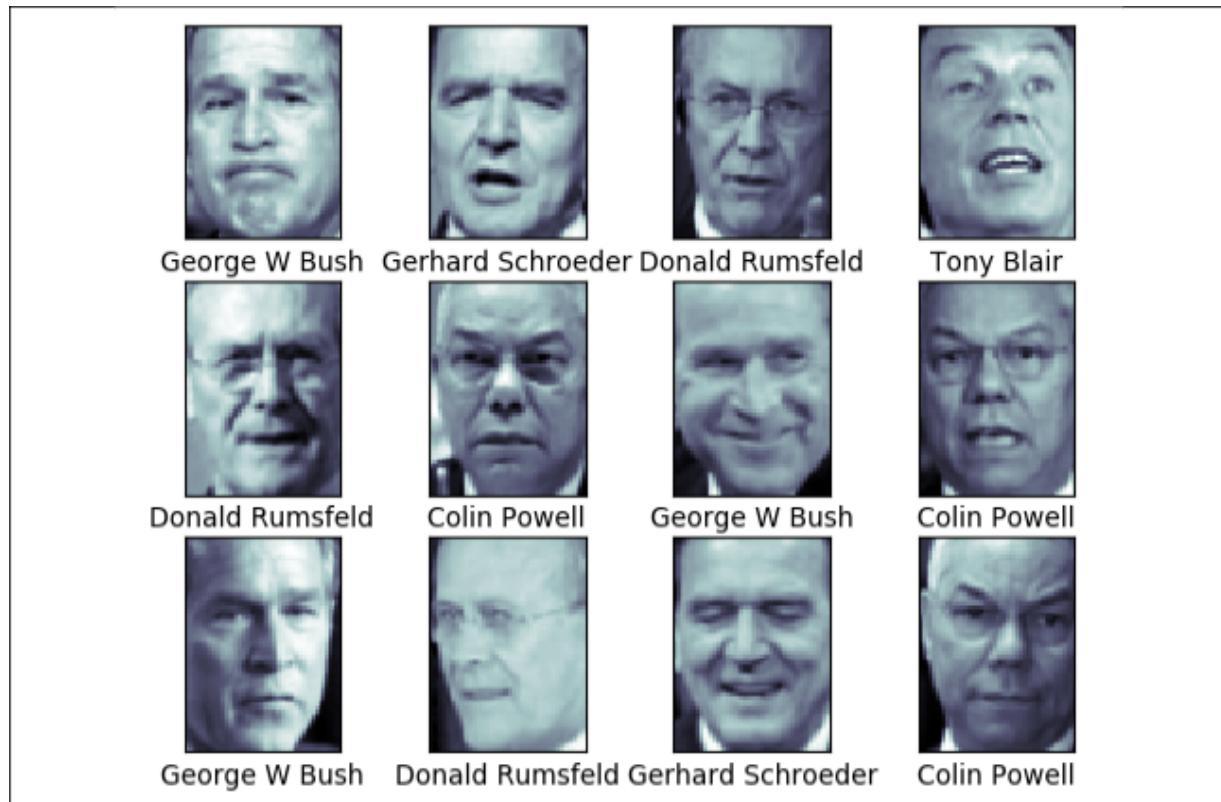


Figure 3.12: Samples from the LFW people dataset

Now that we have covered exploratory data analysis, we will move on to the model development phase in the next section.

Building an SVM-based image classifier

First, we split the data into the training and testing set:

```
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y, r
```

In this project, the number of dimensions is greater than the number of samples. This is a classification case that SVM is effective at solving. In our

solution, we will tune the hyperparameters, including the penalty `C`, the kernel (linear or RBF), and γ (for the RBF kernel) through cross-validation.

We then initialize a common SVM model:

```
>>> clf = SVC(class_weight='balanced', random_state=42)
```

The dataset is imbalanced, so we set `class_weight='balanced'` to emphasize the underrepresented classes.

The way we have conducted cross-validation so far is to explicitly split data into folds and repetitively write a `for` loop to consecutively examine each hyperparameter. To make this less redundant, we'll introduce a more elegant approach utilizing the `GridSearchCV` module from scikit-learn.

`GridSearchCV` handles the entire process implicitly, including data splitting, fold generation, cross training and validation, and finally, an exhaustive search over the best set of parameters. What is left for us is just to specify the hyperparameter(s) to tune and the values to explore for each individual hyperparameter:

```
>>> parameters = {'C': [0.1, 1, 10],  
...                 'gamma': [1e-07, 1e-08, 1e-06],  
...                 'kernel' : ['rbf', 'linear'] }  
>>> from sklearn.model_selection import GridSearchCV  
>>> grid_search = GridSearchCV(clf, parameters, n_jobs=-1, cv=5)
```

The `GridSearchCV` model we just initialized will conduct five-fold cross-validation (`cv=5`) and will run in parallel on all available cores (`n_jobs=-1`). We then perform hyperparameter tuning by simply applying the `fit` method:

```
>>> grid_search.fit(X_train, Y_train)
```

We obtain the optimal set of hyperparameters using the following code:

```
>>> print('The best model:\n', grid_search.best_params_)
The best model:
{'C': 10, 'gamma': 1e-07, 'kernel': 'rbf'}
```

And we obtain the best five-fold averaged performance under the optimal set of parameters by using the following code:

```
>>> print('The best averaged performance:', grid_search.best_scc)
The best averaged performance: 0.8526315789473684
```

We then retrieve the SVM model with the optimal set of hyperparameters and apply it to the testing set:

```
>>> clf_best = grid_search.best_estimator_
>>> pred = clf_best.predict(X_test)
```

We then calculate the accuracy and classification report:

```
>>> print(f'The accuracy is: {clf_best.score(X_test,
...             Y_test)*100:.1f}%')
The accuracy is: 87.7%
>>> print(classification_report(Y_test, pred,
...             target_names=face_data.target_names))
           precision    recall  f1-score   support
    Colin Powell      0.89      0.88      0.88       64
  Donald Rumsfeld      0.84      0.81      0.83       32
  George W Bush      0.88      0.93      0.90      127
 Gerhard Schroeder      0.84      0.72      0.78       29
    Tony Blair      0.91      0.88      0.89       33
         micro avg      0.88      0.88      0.88      285
         macro avg      0.87      0.84      0.86      285
     weighted avg      0.88      0.88      0.88      285
```

It should be noted that we tune the model based on the original training set, which is divided into folds for cross training and validation internally, and that we apply the optimal model to the original testing set. We examine the classification performance in this manner in order to measure how well generalized the model is in order to make correct predictions on a

completely new dataset. An accuracy of 87.7% is achieved with the best SVM model.

There is another SVM classifier, `LinearSVC` (<https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html>), from scikit-learn. How is it different from `svc`? `LinearSVC` is similar to `svc` with linear kernels, but it is implemented based on the `liblinear` library, which is better optimized than `libsvm` with the linear kernel, and its penalty function is more flexible.

In general, training with the `LinearSVC` model is faster than `svc`. This is because the `liblinear` library with high scalability is designed for large datasets, while the `libsvm` library with more than quadratic computation complexity is not able to scale well with more than 10^5 training instances. But again, the `LinearSVC` model is limited to only linear kernels.

Boosting image classification performance with PCA

We can also improve the image classifier by compressing the input features with **principal component analysis (PCA)** (https://en.wikipedia.org/wiki/Principal_component_analysis). It reduces the dimension of the original feature space and preserves the most important internal relationships among features. In simple terms, PCA projects the original data into a smaller space with the most important directions (coordinates). We hope that in cases where we have more features than training samples, considering fewer features as a result of dimensionality reduction using PCA can prevent overfitting.

We will implement PCA with the `PCA` module (<https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html>) from scikit-learn. We will first apply PCA to reduce the dimensionality and train the classifier on the resulting data. In machine learning, we usually concatenate multiple consecutive steps and treat them

as one "model." We call this process **pipelining**. We utilize the `pipeline` API (<https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html>) from scikit-learn to facilitate this.

Now let's initialize a PCA model, an SVC model, and a model pipelining these two:

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=100, whiten=True, random_state=42)
>>> svc = SVC(class_weight='balanced', kernel='rbf',
...             random_state=42)
>>> from sklearn.pipeline import Pipeline
>>> model = Pipeline([('pca', pca),
...                     ('svc', svc)])
```

The PCA component projects the original data into a 100-dimension space, followed by the SVC classifier with the RBF kernel. We then grid search for the best model from a few options:

```
>>> parameters_pipeline = {'svc__C': [1, 3, 10],
...                         'svc__gamma': [0.001, 0.005]}
>>> grid_search = GridSearchCV(model, parameters_pipeline)
>>> grid_search.fit(X_train, Y_train)
```

Finally, we print out the best set of hyperparameters and the classification performance with the best model:

```
>>> print('The best model:\n', grid_search.best_params_)
The best model:
{'svc__C': 3, 'svc__gamma': 0.005}
>>> print('The best averaged performance:', grid_search.best_score_
The best averaged performance: 0.8467836257309942
>>> model_best = grid_search.best_estimator_
>>> print(f'The accuracy is: {model_best.score(X_test, Y_test)*100:.2f} %')
The accuracy is: 92.3%
>>> pred = model_best.predict(X_test)
>>> print(classification_report(Y_test, pred, target_names=face_
...                               precision    recall   f1-score   support
...                               Colin Powell    0.97     0.95     0.96      64
...                               Ronald Reagan    0.99     0.91     0.99      36
```

Donald Rumsfeld	0.93	0.84	0.89	32
George W Bush	0.92	0.98	0.95	127
Gerhard Schroeder	0.88	0.79	0.84	29
Tony Blair	0.88	0.85	0.86	33
micro avg	0.92	0.92	0.92	285
macro avg	0.92	0.88	0.90	285
weighted avg	0.92	0.92	0.92	285

The model composed of a PCA and an SVM classifier achieves an accuracy of 92.3%. PCA boosts the performance of the SVM-based image classifier. You can read more about PCA at <https://www.kaggle.com/nirajvermafcb/principal-component-analysis-explained> if you are interested.

Following the successful application of SVM in image classification, we will look at one more example in the next section.

Fetal state classification on cardiotocography

We are going to build a classifier that helps obstetricians categorize **cardiotocograms (CTGs)** into one of the three fetal states (normal, suspect, and pathologic). The cardiotocography dataset we will use is from <https://archive.ics.uci.edu/ml/datasets/Cardiotocography> in the UCI Machine Learning Repository, and it can be directly downloaded from <https://archive.ics.uci.edu/ml/machine-learning-databases/00193/CTG.xls> as an .xls Excel file. The dataset consists of measurements of fetal heart rate and uterine contraction as features, and the fetal state class code (1=normal, 2=suspect, 3=pathologic) as a label. There are in total 2,126 samples with 23 features. Based on the numbers of instances and features (2,126 is not significantly larger than 23), the RBF kernel is the first choice.

We will work with the Excel file using pandas, which is suitable for table data. It might request an additional installation of the `xlrd` package when you run the following lines of codes, since its Excel module is built based on `xlrd`. If so, just run `pip install xlrd` in the terminal to install `xlrd`.

We first read the data located in the sheet named `Raw Data`:

```
>>> import pandas as pd  
>>> df = pd.read_excel('CTG.xls', "Raw Data")
```

Then, we take these 2,126 data samples and assign the feature set (from columns `D` to `AL` in the spreadsheet) and label set (column `AN`):

```
>>> X = df.iloc[1:2126, 3:-2].values  
>>> Y = df.iloc[1:2126, -1].values
```

Don't forget to check the class proportions:

```
>>> print(Counter(Y))  
Counter({1.0: 1655, 2.0: 295, 3.0: 176})
```

We set aside 20% of the original data for final testing:

```
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,  
... test_size=0.2, random_state=
```

Now, we tune the RBF-based SVM model in terms of the penalty `c`, and the `kernel` coefficient γ :

```
>>> svc = SVC(kernel='rbf')  
>>> parameters = {'C': (100, 1e3, 1e4, 1e5),  
...                 'gamma': (1e-08, 1e-07, 1e-06, 1e-05)}  
>>> grid_search = GridSearchCV(svc, parameters, n_jobs=-1, cv=5)  
>>> grid_search.fit(X_train, Y_train)  
>>> print(grid_search.best_params_)  
{'C': 100000.0, 'gamma': 1e-07}
```

```
>>> print(grid_search.best_score_)
0.9547058823529412
```

Finally, we apply the optimal model to the testing set:

```
>>> svc_best = grid_search.best_estimator_
>>> accuracy = svc_best.score(X_test, Y_test)
>>> print(f'The accuracy is: {accuracy*100:.1f}%')
The accuracy is: 96.5%
```

Also, we have to check the performance for individual classes since the data is quite imbalanced:

```
>>> prediction = svc_best.predict(X_test)
>>> report = classification_report(Y_test, prediction)
>>> print(report)
           precision    recall  f1-score   support
          1.0      0.98      0.98      0.98      333
          2.0      0.89      0.91      0.90       64
          3.0      0.96      0.93      0.95       29
    micro avg      0.96      0.96      0.96      426
  macro avg      0.95      0.94      0.94      426
weighted avg      0.96      0.96      0.96      426
```

We have now successfully built another SVM-based classifier to solve a real-world problem: fetal state classification in cardiotocography.

Summary

In this chapter, we continued our journey of supervised learning with SVM. You learned about the mechanics of an SVM, kernel techniques and implementations of SVM, and other important concepts of machine learning classification, including multiclass classification strategies and grid search, as well as useful tips for using an SVM (for example, choosing between kernels and tuning parameters). Then, we finally put into practice

what you learned in the form of real-world use cases, including face recognition and fetal state classification.

You have learned and adopted two classification algorithms so far, Naïve Bayes and SVM. Naïve Bayes is a simple algorithm (as its name implies). For a dataset with independent, or close to independent, features, Naïve Bayes will usually perform well. SVM is versatile and adaptive to the linear separability of data. In general, high accuracy can be achieved by SVM with the right kernel and parameters. However, this might be at the expense of intense computation and high memory consumption. In practice, we can simply try both and select the better one with optimal parameters.

In the next chapter, we will look at online advertising and predict whether a user will click on an ad. This will be accomplished by means of tree-based algorithms, including **decision tree** and **random forest**.

Exercises

1. Can you implement SVM using the `Linearsvc` module? What are the hyperparameters that you need to tweak, and what is the best performance of face recognition you are able to achieve?
2. Can you classify more classes in the image recognition project? As an example, you can set `min_faces_per_person=50`. What is the best performance you can achieve using grid search and cross-validation?

Predicting Online Ad Click-Through with Tree-Based Algorithms

We built a face image classifier in the previous chapter. In this chapter and the next, we will be solving one of the most data-driven problems in digital advertising: ad click-through prediction—given a user and the page they are visiting, this predicts how likely it is that they will click on a given ad. We will focus on learning tree-based algorithms (including decision tree, random forest, and boosted trees) and utilize them to tackle this billion-dollar problem. We will be exploring decision trees from the root to the leaves, as well as the aggregated version, a forest of trees. This won't be a theory-only chapter, as there are a lot of hand calculations and implementations of tree models from scratch included. We will be using scikit-learn and XGBoost, a popular Python package for tree-based algorithms.

We will cover the following topics in this chapter:

- Two types of features: numerical and categorical
- The mechanics of a decision tree classifier
- The implementation of decision tree
- Ad click-through predictions
- The ensemble method and bagging technique
- The mechanics of random forest
- Click-through predictions with random forest
- The **gradient boosted trees (GBT)** model

- The implementation of GBT using XGBoost

A brief overview of ad click-through prediction

Online display advertising is a multibillion-dollar industry. It comes in different formats, including banner ads composed of text, images, and flash, and rich media such as audio and video. Advertisers, or their agencies, place ads on a variety of websites, and even mobile apps, across the Internet in order to reach potential customers and deliver an advertising message.

Online display advertising has served as one of the greatest examples of machine learning utilization. Obviously, advertisers and consumers are keenly interested in well-targeted ads. In the last 20 years, the industry has relied heavily on the ability of machine learning models to predict the effectiveness of ad targeting: how likely it is that an audience of a certain age group will be interested in this product, that customers with a certain household income will purchase this product after seeing the ad, that frequent sports site visitors will spend more time reading this ad, and so on. The most common measurement of effectiveness is the **click-through rate (CTR)**, which is the ratio of clicks on a specific ad to its total number of views. The higher the CTR in general, the better targeted an ad is, and the more successful an online advertising campaign is.

Click-through prediction entails both the promises and challenges of machine learning. It mainly involves the binary classification of whether a given ad on a given page (or app) will be clicked on by a given user, with predictive features from the following three aspects:

- Ad content and information (category, position, text, format, and so on)
- Page content and publisher information (category, context, domain, and so on)

- User information (age, gender, location, income, interests, search history, browsing history, device, and so on)

Suppose we, as an agency, are operating ads on behalf of several advertisers, and our job is to place the right ads for the right audience. Let's say that we have an existing dataset in hand (the following small chunk is an example; the number of predictive features can easily go into the thousands in reality) taken from millions of records of campaigns run a month ago, and we need to develop a classification model to learn and predict future ad placement outcomes:

Ad category	Site category	Site domain	User age	User gender	User occupation	Interested in sports	Interested in tech	Click
Auto	News	cnn.com	25-34	M	Professional	True	True	1
Fashion	News	bbc.com	35-54	F	Professional	False	False	0
Auto	Edu	onlinestudy.com	17-24	F	Student	True	True	0
Food	Entertainment	movie.com	25-34	M	Clerk	True	False	1
Fashion	Sports	football.com	55+	M	Retired	True	False	0
...
...

Food	News	abc.com	17-24	M	Student	True	True	?
Auto	Entertainment	movie.com	35-54	F	Professional	True	False	?

Figure 4.1: Ad samples for training and prediction

As you can see in *Figure 4.1*, the features are mostly categorical. In fact, data can be either numerical or categorical. Let's explore this in more detail in the next section.

Getting started with two types of data – numerical and categorical

At first glance, the features in the preceding dataset are **categorical**, for example, male or female, one of four age groups, one of the predefined site categories, or whether the user is interested in sports. Such data is different from the **numerical** feature data we have worked with until now.

Categorical (also called **qualitative**) features represent characteristics, distinct groups, and a countable number of options. Categorical features may or may not have a logical order. For example, household income from low to medium to high is an **ordinal** feature, while the category of an ad is not ordinal.

Numerical (also called **quantitative**) features, on the other hand, have mathematical meaning as a measurement and, of course, are ordered. For instance, term frequency and the tf-idf variant are discrete and continuous numerical features, respectively; the cardiotocography dataset (<https://archive.ics.uci.edu/ml/datasets/Cardiotocography>) from the last chapter contains both discrete (such as the number of accelerations per second or the number of fetal movements per second) and continuous (such as the mean value of long-term variability) numerical features.

Categorical features can also take on numerical values. For example, 1 to 12 can represent months of the year, and 1 and 0 can indicate male and female. Still, these values do not have mathematical implications.

Of the two classification algorithms that you learned previously, Naïve Bayes and SVM, the Naïve Bayes classifier works for both numerical and categorical features as the likelihoods, $P(x | y)$ or $P(\text{feature} | \text{class})$, are calculated in the same way, while SVM requires features to be numerical in order to compute and maximize the distance margins.

Now, we are thinking of predicting click-through using Naïve Bayes and trying to explain the model to our advertising clients. However, our clients may find it difficult to understand the prior and the likelihood of individual attributes and their multiplication. Is there a classifier that is easy to interpret and explain to clients, and that is able to directly handle categorical data? Decision trees are the answer!

Exploring a decision tree from the root to the leaves

A decision tree is a tree-like graph, that is, a sequential diagram illustrating all of the possible decision alternatives and their corresponding outcomes. Starting from the **root** of a tree, every internal **node** represents the basis on which a decision is made. Each branch of a node represents how a choice may lead to the next nodes. And, finally, each **terminal node**, the **leaf**, represents the outcome produced.

For example, we have just made a couple of decisions that brought us to the point of using a decision tree to solve our advertising problem:

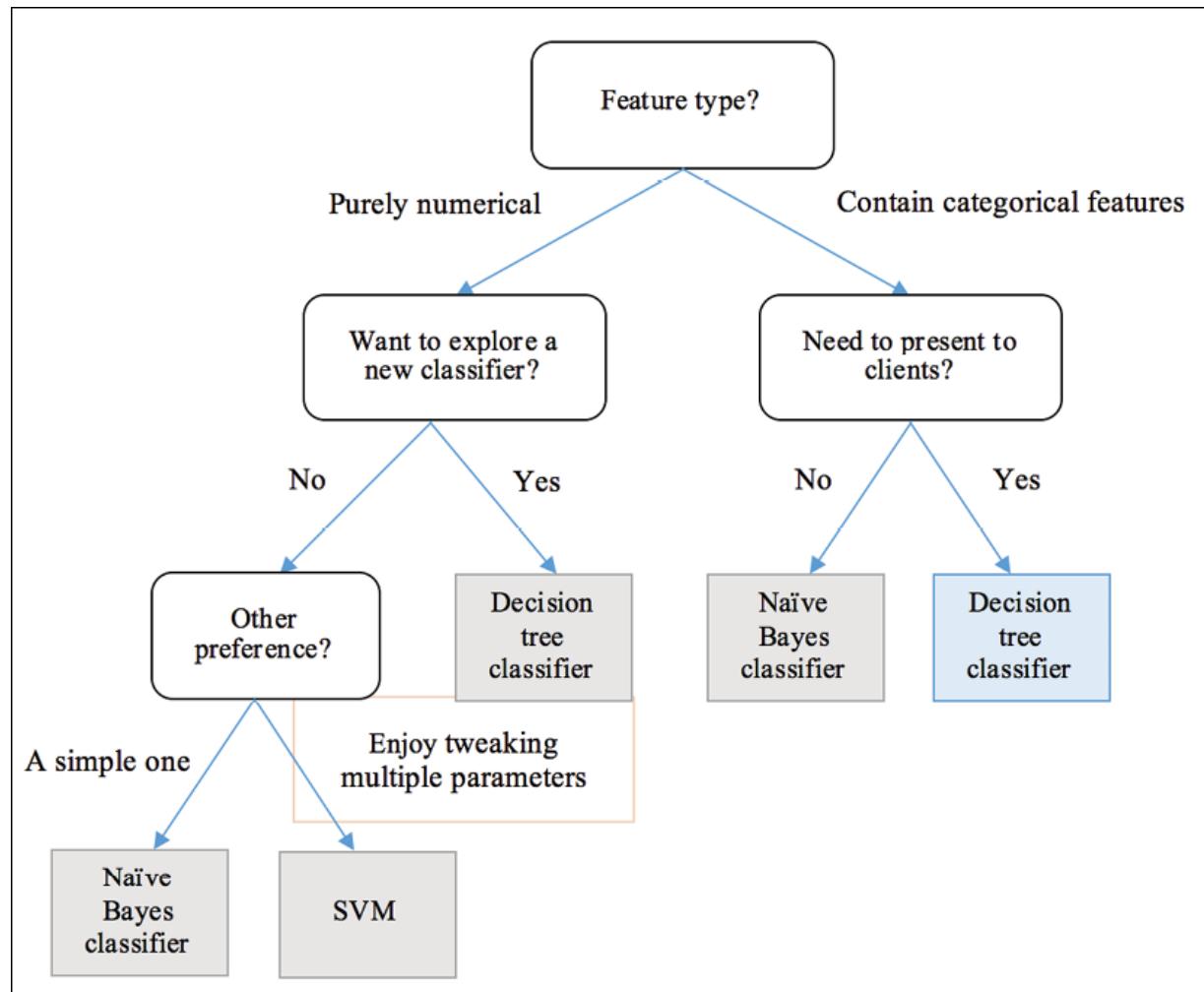


Figure 4.2: Using a decision tree to find the right algorithm

The first condition, or the root, is whether the feature type is numerical or categorical. Ad clickstream data contains mostly categorical features, so it goes to the right branch. In the next node, our work needs to be interpretable by non-technical clients. So, it goes to the right branch and reaches the leaf for choosing the decision tree classifier.

You can also look at paths and see what kinds of problems they can fit in. A decision tree classifier operates in the form of a decision tree. It maps observations to class assignments (symbolized as leaf nodes) through a series of tests (represented as internal nodes) based on feature values and corresponding conditions (represented as branches). In each node, a question regarding the values and characteristics of a feature is asked; depending on the answer to the question, the observations are split into subsets. Sequential tests are conducted until a conclusion about the observations' target label is reached. The paths from the root to the end leaves represent the decision-making process and the classification rules.

In a more simplified scenario, as shown in *Figure 4.3*, where we want to predict **Click** or **No click** on a self-driven car ad, we can manually construct a decision tree classifier that works for an available dataset. For example, if a user is interested in technology and has a car, they will tend to click on the ad; a person outside of this subset, for instance, a high-income woman, is unlikely to click on the ad. We then use the trained tree to predict two new inputs, whose results are **Click** and **No click**, respectively:

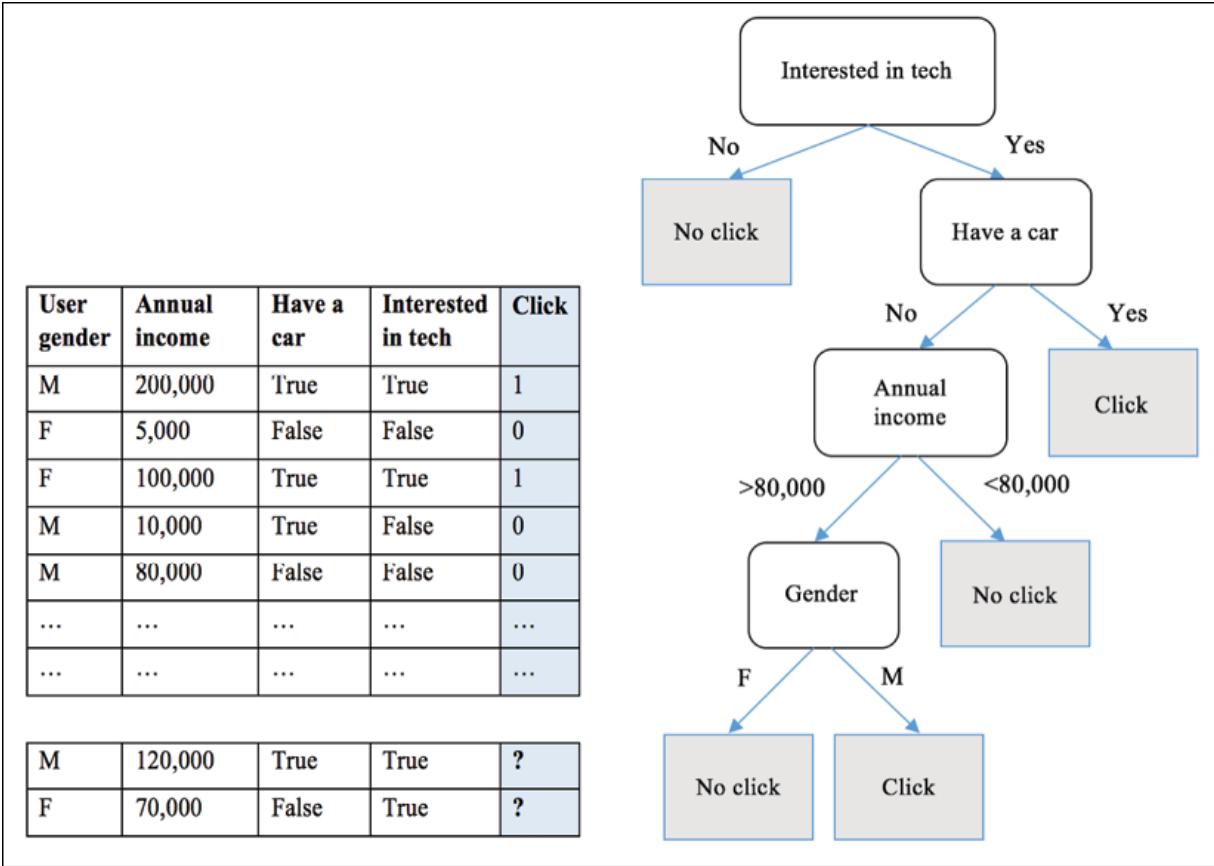


Figure 4.3: Predicting Click/No Click with a trained decision tree

After a decision tree has been constructed, classifying a new sample is straightforward, as you just saw: starting from the root, apply the test condition and follow the branch accordingly until a leaf node is reached, and the class label associated will be assigned to the new sample.

So, how can we build an appropriate decision tree?

Constructing a decision tree

A decision tree is constructed by partitioning the training samples into successive subsets. The partitioning process is repeated in a recursive fashion on each subset. For each partitioning at a node, a condition test is conducted based on the value of a feature of the subset. When the subset shares the same class label, or when no further splitting can improve the class purity of this subset, recursive partitioning on this node is finished.

Theoretically, to partition a feature (numerical or categorical) with n different values, there are n different methods of binary splitting (**Yes** or **No** to the condition test, as illustrated in *Figure 4.4*), not to mention other ways of splitting (for example, three- and four-way splitting in *Figure 4.4*):

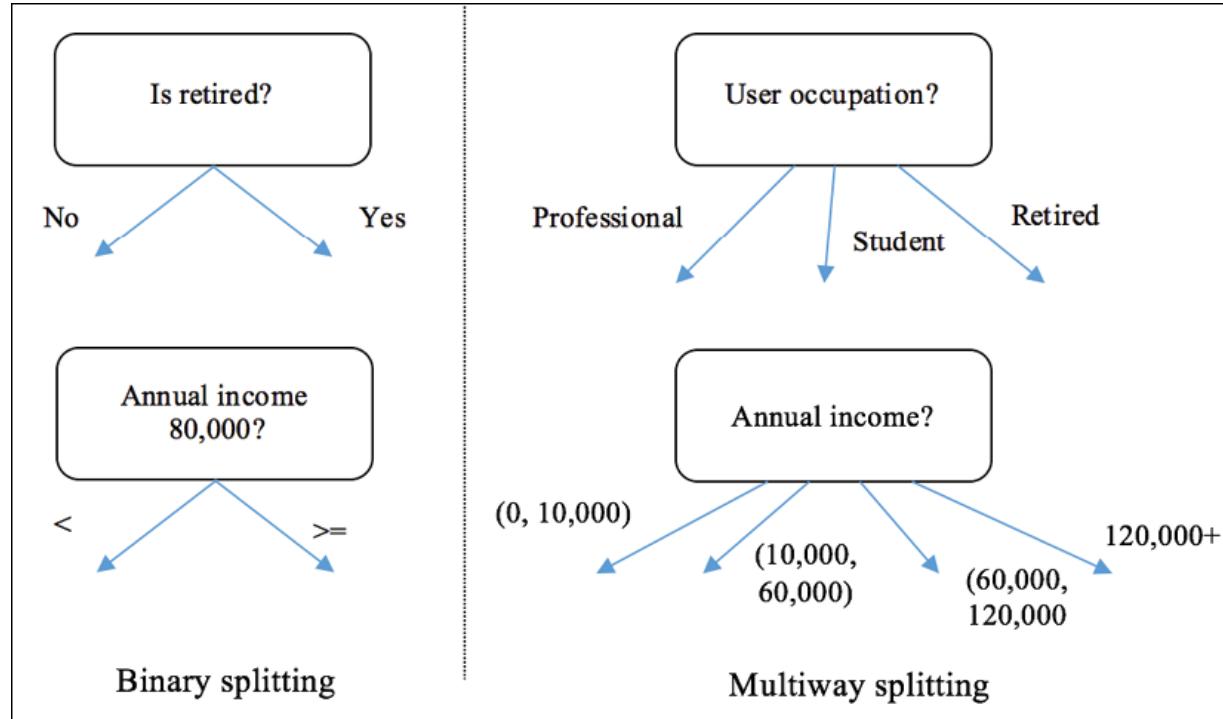


Figure 4.4: Examples of binary splitting and multiway splitting

Without considering the order of features that partitioning is taking place on, there are already n^m possible trees for an m -dimensional dataset.

Many algorithms have been developed to efficiently construct an accurate decision tree. Popular ones include the following:

- **Iterative Dichotomiser 3 (ID3):** This algorithm uses a greedy search in a top-down manner by selecting the best attribute to split the dataset on with each iteration without backtracking.
- **C4.5:** This is an improved version of ID3 that introduces backtracking. It traverses the constructed tree and replaces the branches with leaf nodes if purity is improved this way.

- **Classification and Regression Tree (CART):** This constructs the tree using binary splitting, which we will discuss in more detail shortly.
- **Chi-squared Automatic Interaction Detector (CHAID):** This algorithm is often used in direct marketing. It involves complicated statistical concepts, but basically, it determines the optimal way of merging predictive variables in order to best explain the outcome.

The basic idea of these algorithms is to grow the tree greedily by making a series of local optimizations when choosing the most significant feature to use to partition the data. The dataset is then split based on the optimal value of that feature. We will discuss the measurement of a significant feature and the optimal splitting value of a feature in the next section.

First, we will study the CART algorithm in more detail, and we will implement it as the most notable decision tree algorithm after that. It constructs the tree using binary splitting and grows each node into left and right children. In each partition, it greedily searches for the most significant combination of a feature and its value; all different possible combinations are tried and tested using a measurement function. With the selected feature and value as a splitting point, the algorithm then divides the dataset as follows:

- Samples with the feature of this value (for a categorical feature) or a greater value (for a numerical feature) become the right child
- The remaining samples become the left child

This partitioning process repeats and recursively divides up the input samples into two subgroups. When the dataset becomes unmixed, the splitting process stops at a subgroup where either of the following two criteria is met:

- **The minimum number of samples for a new node:** When the number of samples is not greater than the minimum number of samples required for a further split, the partitioning stops in order to prevent the tree from excessively tailoring to the training set and, as a result, overfitting.

- **The maximum depth of the tree:** A node stops growing when its depth, which is defined as the number of partitions taking place from the top down, starting from the root node and ending in a terminal node, is not less than the maximum tree depth. Deeper trees are more specific to the training set and can lead to overfitting.

A node with no branches becomes a leaf, and the dominant class of samples at this node is the prediction. Once all splitting processes finish, the tree is constructed and is portrayed with the assigned labels at the terminal nodes and the splitting points (feature + value) at all the internal nodes above.

We will implement the CART decision tree algorithm from scratch after studying the metrics of selecting the optimal splitting feature and value, as promised.

The metrics for measuring a split

When selecting the best combination of a feature and a value as the splitting point, two criteria, such as **Gini Impurity** and **Information Gain**, can be used to measure the quality of separation.

Gini Impurity

Gini Impurity, as its name implies, measures the impurity rate of the class distribution of data points, or the class mixture rate. For a dataset with K classes, suppose that data from class k ($1 \leq k \leq K$) takes up a fraction f_k ($0 \leq f_k \leq 1$) of the entire dataset; then the *Gini Impurity* of this dataset is written as follows:

$$\text{Gini Impurity} = 1 - \sum_{k=1}^K f_k^2$$

A lower Gini Impurity indicates a purer dataset. For example, when the dataset contains only one class, say, the fraction of this class is 1 and that of the others is 0, its Gini Impurity becomes $1 - (1^2 + 0^2) = 0$. In another example, a dataset records a large number of coin flips, and heads and tails

each take up half of the samples. The Gini Impurity is $1 - (0.5^2 + 0.5^2) = 0.5$.

In binary cases, Gini Impurity, under different values of the positive class fraction, can be visualized by the following code blocks:

```
>>> import matplotlib.pyplot as plt  
>>> import numpy as np
```

The fraction of the positive class varies from 0 to 1 :

```
>>> pos_fraction = np.linspace(0.00, 1.00, 1000)
```

The Gini Impurity is calculated accordingly, followed by the plot of **Gini Impurity** versus **Positive fraction**:

```
>>> gini = 1 - pos_fraction**2 - (1-pos_fraction)**2
```

Here, 1-pos_fraction is the negative fraction:

```
>>> plt.plot(pos_fraction, gini)  
>>> plt.ylim(0, 1)  
>>> plt.xlabel('Positive fraction')  
>>> plt.ylabel('Gini Impurity')  
>>> plt.show()
```

Refer to *Figure 4.5* for the end result:

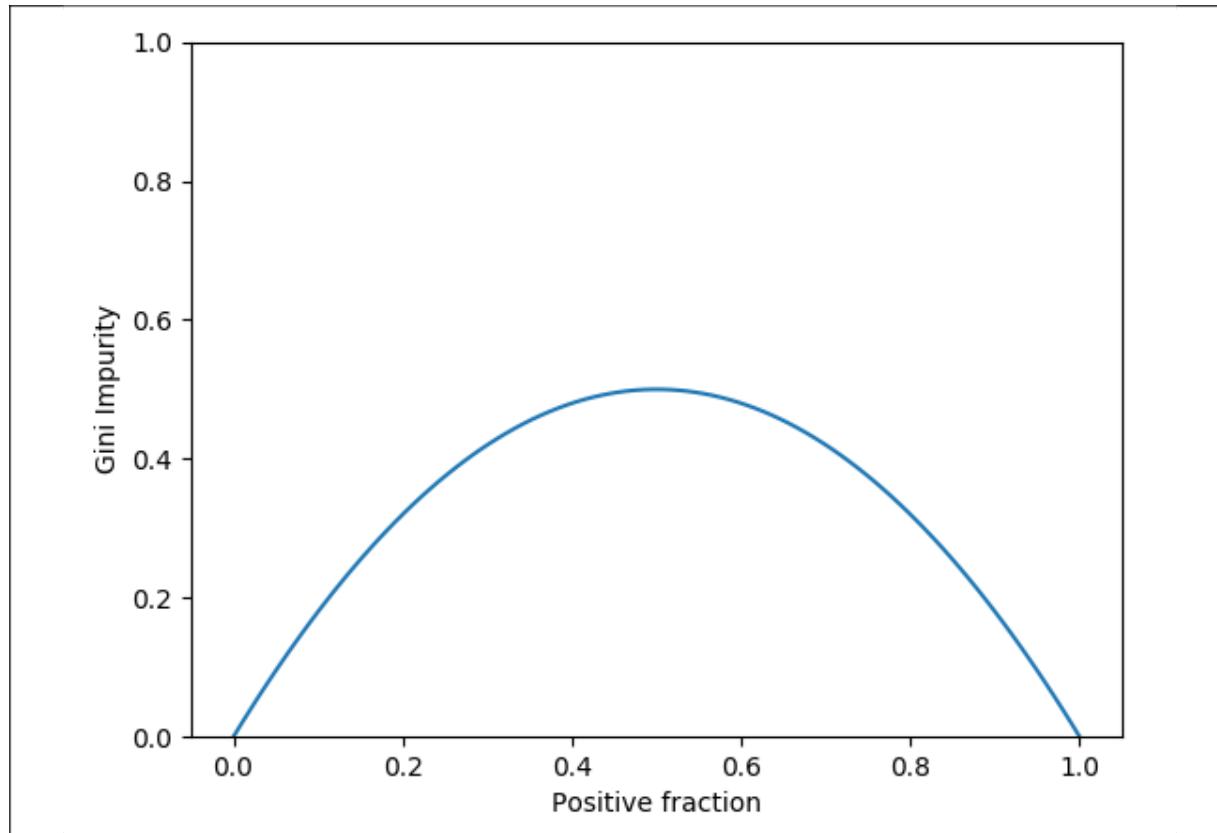


Figure 4.5: Gini Impurity versus positive fraction

As you can see, in binary cases, if the positive fraction is 50%, the impurity will be the highest at 0.5; if the positive fraction is 100% or 0%, it will reach 0 impurity.

Given the labels of a dataset, we can implement the Gini Impurity calculation function as follows:

```
>>> def gini_impurity(labels):
...     # When the set is empty, it is also pure
...     if not labels:
...         return 0
...     # Count the occurrences of each label
...     counts = np.unique(labels, return_counts=True)[1]
...     fractions = counts / float(len(labels))
...     return 1 - np.sum(fractions ** 2)
```

Test it out with some examples:

```

>>> print(f'{gini_impurity([1, 1, 0, 1, 0]):.4f}')
0.4800
>>> print(f'{gini_impurity([1, 1, 0, 1, 0]):.4f} ')
0.5000
>>> print(f'{gini_impurity([1, 1, 1, 1]):.4f}')
0.0000

```

In order to evaluate the quality of a split, we simply add up the Gini Impurity of all resulting subgroups, combining the proportions of each subgroup as corresponding weight factors. And again, the smaller the weighted sum of the Gini Impurity, the better the split.

Take a look at the following self-driving car ad example. Here, we split the data based on a user's gender and interest in technology, respectively:

User gender	Interested in tech	Click	Group by gender	User gender	Interested in tech	Click	Group by interest
M	True	1	Group 1	M	True	1	Group 1
F	False	0	Group 2	F	False	0	Group 2
F	True	1	Group 2	F	True	1	Group 1
M	False	0	Group 1	M	False	0	Group 2
M	False	1	Group 1	M	False	1	Group 2

#1 split based on gender #2 split based on interest in tech

Figure 4.6: Splitting the data based on gender or interest in tech

The weighted Gini Impurity of the first split can be calculated as follows:

$$\#1 \text{ Gini Impurity} = \frac{3}{5} [1 - (\frac{2^2}{3} + \frac{1^2}{3})] + \frac{2}{5} [1 - (\frac{1^2}{2} + \frac{1^2}{2})] = 0.467$$

The second split is as follows:

$$\#2 \text{ Gini Impurity} = \frac{2}{5} [1 - (1^2 + 0^2)] + \frac{3}{5} [1 - (\frac{1^2}{3} + \frac{2^2}{3})] = 0.267$$

Therefore, splitting data based on the user's interest in technology is a better strategy than gender.

Information Gain

Another metric, **Information Gain**, measures the improvement of purity after splitting or, in other words, the reduction of uncertainty due to a split. Higher Information Gain implies better splitting. We obtain the Information Gain of a split by comparing the **entropy** before and after the split.

Entropy is a probabilistic measure of uncertainty. Given a K -class dataset, and f_k ($0 \leq f_k \leq 1$) denoted as the fraction of data from class k ($1 \leq k \leq K$), the *entropy* of the dataset is defined as follows:

$$\text{Entropy} = -\sum_{k=1}^K f_k * \log_2 f_k$$

Lower entropy implies a purer dataset with less ambiguity. In a perfect case, where the dataset contains only one class, the entropy is $-(1 * \log_2 1 + 0) = 0$. In the coin flip example, the entropy becomes $-(0.5 * \log_2 0.5 + 0.5 * \log_2 0.5) = 1$.

Similarly, we can visualize how entropy changes with different values of the positive class' fraction in binary cases using the following lines of code:

```
>>> pos_fraction = np.linspace(0.00, 1.00, 1000)
>>> ent = - (pos_fraction * np.log2(pos_fraction)) +
...           (1 - pos_fraction) * np.log2(1 - pos_fraction))
>>> plt.plot(pos_fraction, ent)
>>> plt.xlabel('Positive fraction')
>>> plt.ylabel('Entropy')
>>> plt.ylim(0, 1)
>>> plt.show()
```

This will give us the following output:

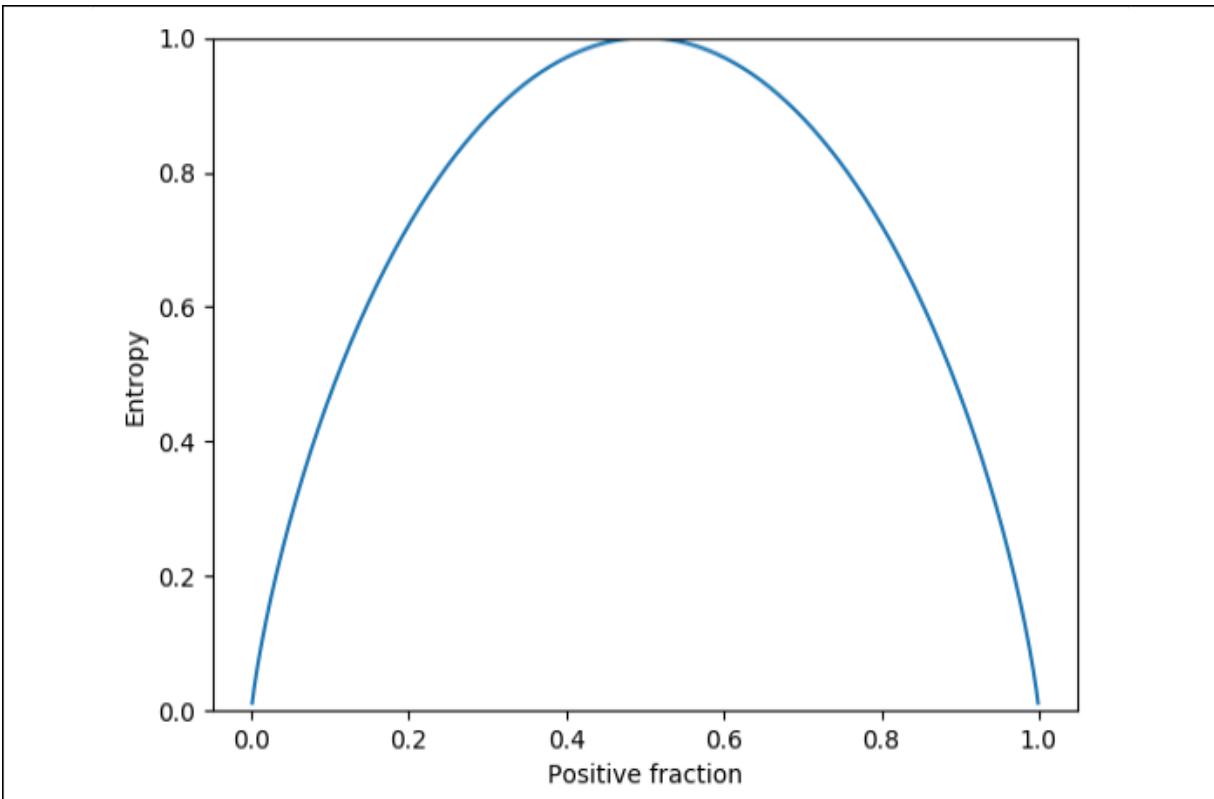


Figure 4.7: Entropy versus positive fraction

As you can see, in binary cases, if the positive fraction is 50%, the entropy will be the highest at 1 ; if the positive fraction is 100% or 0%, it will reach 0 entropy.

Given the labels of a dataset, the `entropy` calculation function can be implemented as follows:

```
>>> def entropy(labels):
...     if not labels:
...         return 0
...     counts = np.unique(labels, return_counts=True)[1]
...     fractions = counts / float(len(labels))
...     return - np.sum(fractions * np.log2(fractions))
```

Test it out with some examples:

```
>>> print(f'{entropy([1, 1, 0, 1, 0]):.4f}')
0.9710
```

```

>>> print(f'{entropy([1, 1, 0, 1, 0, 0]):.4f}')
1.0000
>>> print(f'{entropy([1, 1, 1, 1]):.4f}')
-0.0000

```

Now that you have fully understood entropy, we can look into how Information Gain measures how much uncertainty was reduced after splitting, which is defined as the difference in entropy before a split (parent) and after a split (children):

$$\text{InformationGain} = \text{Entropy}(\text{before}) - \text{Entropy}(\text{after}) = \text{Entropy}(\text{parent}) - \text{Entropy}(\text{children})$$

Entropy after a split is calculated as the weighted sum of the entropy of each child, which is similar to the weighted Gini Impurity.

During the process of constructing a node at a tree, our goal is to search for the splitting point where the maximum Information Gain is obtained. As the entropy of the parent node is unchanged, we just need to measure the entropy of the resulting children due to a split. The best split is the one with the lowest entropy of its resulting children.

To understand this better, let's look at the self-driving car ad example again.

For the first option, the entropy after the split can be calculated as follows:

$$\#1 \text{ entropy} = \frac{3}{5}\left(-\left(\frac{2}{3} * \log_2 \frac{2}{3} + \frac{1}{3} * \log_2 \frac{1}{3}\right)\right) + \frac{2}{5}\left(-\left(\frac{1}{2} * \log_2 \frac{1}{2} + \frac{1}{2} * \log_2 \frac{1}{2}\right)\right) = 0.951$$

The second way of splitting is as follows:

$$\#2 \text{ entropy} = \frac{2}{5}\left(-(1 * \log_2 1 + 0)\right) + \frac{3}{5}\left(-\left(\frac{1}{3} * \log_2 \frac{1}{3} + \frac{2}{3} * \log_2 \frac{2}{3}\right)\right) = 0.551$$

For exploration purposes, we can also calculate the InformationGain by:

$$\text{Entropy before} = -\left(\frac{3}{5} * \log_2 \frac{3}{5} + \frac{2}{5} * \log_2 \frac{2}{5}\right) = 0.971$$

$$\#1 \text{ InformationGain} = 0.971 - 0.951 = 0.020$$

$$\#2 \text{ InformationGain} = 0.971 - 0.551 = 0.420$$

According to the **Information Gain = entropy-based evaluation**, the second split is preferable, which is the conclusion of the Gini Impurity criterion.

In general, the choice of two metrics, Gini Impurity and Information Gain, has little effect on the performance of the trained decision tree. They both measure the weighted impurity of the children after a split. We can combine them into one function to calculate the weighted impurity:

```
>>> criterion_function = {'gini': gini_impurity,
...                      'entropy': entropy}
>>> def weighted_impurity(groups, criterion='gini'):
...     """
...     Calculate weighted impurity of children after a split
...     @param groups: list of children, and a child consists a
...                   list of class labels
...     @param criterion: metric to measure the quality of a spl
...                       'gini' for Gini Impurity or 'entropy'
...                       Information Gain
...     @return: float, weighted impurity
...     """
...     total = sum(len(group) for group in groups)
...     weighted_sum = 0.0
...     for group in groups:
...         weighted_sum += len(group) / float(total) *
...                         criterion_function[criterion](group)
...     return weighted_sum
```

Test it with the example we just hand-calculated, as follows:

```
>>> children_1 = [[1, 0, 1], [0, 1]]
>>> children_2 = [[1, 1], [0, 0, 1]]
>>> print(f"Entropy of #1 split: {weighted_impurity(children_1,
... 'entropy'):.4f}")
Entropy of #1 split: 0.9510
```

```
>>> print('Entropy of #2 split: {:.4f}'.format(weighted_impurity(children[2],  
...                                         'entropy')))  
Entropy of #2 split: 0.5510
```

Now that you have a solid understanding of partitioning evaluation metrics, let's implement the CART tree algorithm from scratch in the next section.

Implementing a decision tree from scratch

We develop the CART tree algorithm by hand on a `toy` dataset as follows:

User interest	User occupation	Click
Tech	Professional	1
Fashion	Student	0
Fashion	Professional	0
Sports	Student	0
Tech	Student	1
Tech	Retired	0
Sports	Professional	1

Figure 4.8: An example of ad data

To begin with, we decide on the first splitting point, the root, by trying out all possible values for each of the two features. We utilize the `weighted_impurity` function we just defined to calculate the weighted Gini Impurity for each possible combination, as follows:

```
Gini(interest, tech) = weighted_impurity([[1, 1, 0],  
[0, 0, 0, 1]]) = 0.405
```

Here, if we partition according to whether the user interest is tech, we have the 1st, 5th, and 6th samples for one group and the remaining samples for another group. Then the classes for the first group are [1, 1, 0], and the classes for the second group are [0, 0, 0, 1] :

```
Gini(interest, Fashion) = weighted_impurity([[0, 0],  
[1, 0, 1, 0, 1]]) = 0.343
```

Here, if we partition according to whether the user's interest is fashion, we have the 2nd and 3rd samples for one group and the remaining samples for another group. Then the classes for the first group are [0, 0], and the classes for the second group are [1, 0, 1, 0, 1] :

```
Gini(interest, Sports) = weighted_impurity([[0, 1],  
[1, 0, 0, 1, 0]]) = 0.486  
Gini(occupation, professional) = weighted_impurity([[0, 0, 1, 0]  
[1, 0, 1]]) = 0.405  
Gini(occupation, student) = weighted_impurity([[0, 0, 1, 0],  
[1, 0, 1]]) = 0.405  
Gini(occupation, retired) = weighted_impurity([[1, 0, 0, 0, 1, 1]  
[1]]) = 0.429
```

The root goes to the user interest feature with the fashion value, as this combination achieves the lowest weighted impurity or the highest Information Gain. We can now build the first level of the tree, as follows:

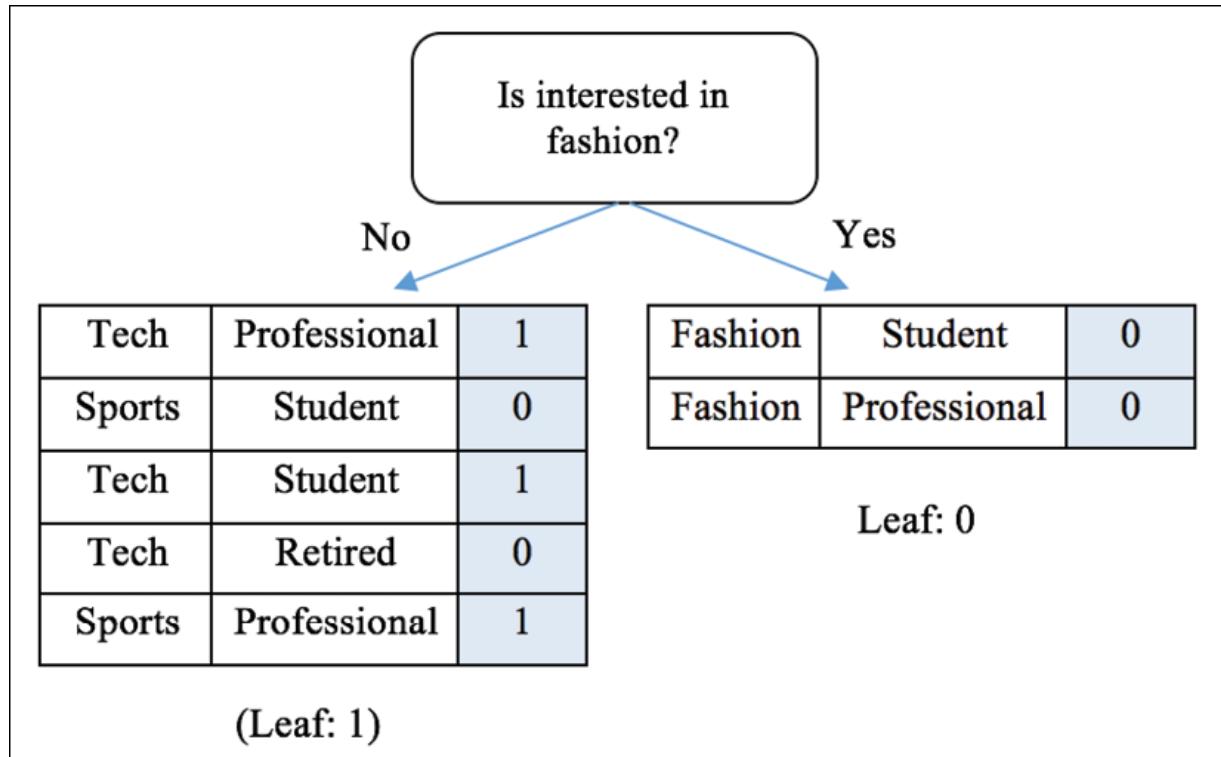


Figure 4.9: Partitioning the data according to is interested in fashion?

If we are satisfied with a one-level-deep tree, we can stop here by assigning the right branch label **0** and the left branch label **1** as the majority class.

Alternatively, we can go further down the road, constructing the second level from the left branch (the right branch cannot be split further):

```

Gini(interest, tech) = weighted_impurity([[0, 1],
[1, 1, 0]]) = 0.467
Gini(interest, Sports) = weighted_impurity([[1, 1, 0],
[0, 1]]) = 0.467
Gini(occupation, professional) = weighted_impurity([[0, 1, 0],
[1, 1]]) = 0.267
Gini(occupation, student) = weighted_impurity([[1, 0, 1],
[0, 1]]) = 0.467
Gini(occupation, retired) = weighted_impurity([[1, 0, 1, 1],
[0]]) = 0.300
    
```

With the second splitting point specified by `(occupation, professional)` with the lowest Gini Impurity, our tree becomes this:

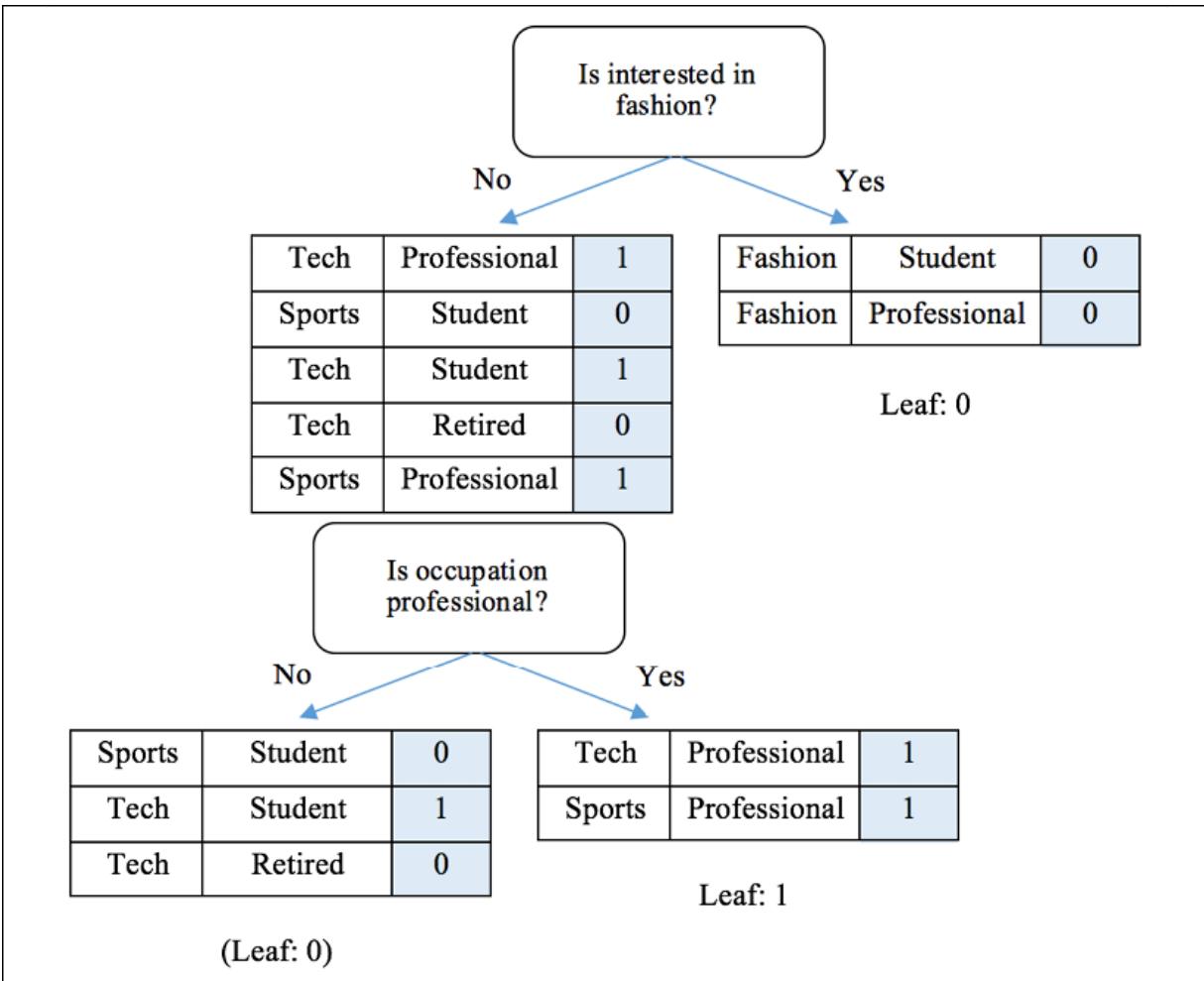


Figure 4.10: Further partitioning of the data according to "is occupation professional?"

We can repeat the splitting process as long as the tree does not exceed the maximum depth and the node contains enough samples.

Now that the process of the tree construction has been made clear, it is time for coding.

We start with the criterion of the best splitting point; the calculation of the weighted impurity of two potential children is what we defined previously, while that of two metrics is slightly different. The inputs now become NumPy arrays for computational efficiency. For Gini Impurity, we have the following:

```

>>> def gini_impurity_np(labels):
...     # When the set is empty, it is also pure
...     if labels.size == 0:
...         return 0
...     # Count the occurrences of each label
...     counts = np.unique(labels, return_counts=True)[1]
...     fractions = counts / float(len(labels))
...     return 1 - np.sum(fractions ** 2)

```

For entropy, we have the following:

```

>>> def entropy_np(labels):
...     # When the set is empty, it is also pure
...     if labels.size == 0:
...         return 0
...     counts = np.unique(labels, return_counts=True)[1]
...     fractions = counts / float(len(labels))
...     return - np.sum(fractions * np.log2(fractions))

```

Also, we update the `weighted_impurity` function, as follows:

```

>>> def weighted_impurity(groups, criterion='gini'):
...     """
...     Calculate weighted impurity of children after a split
...     @param groups: list of children, and a child consists a
...                   of class labels
...     @param criterion: metric to measure the quality of a spl
...                       'gini' for Gini Impurity or
...                       'entropy' for Information Gain
...     @return: float, weighted impurity
...     """
...     total = sum(len(group) for group in groups)
...     weighted_sum = 0.0
...     for group in groups:
...         weighted_sum += len(group) / float(total) *
...                         criterion_function_np[criterion](gr
...     return weighted_sum

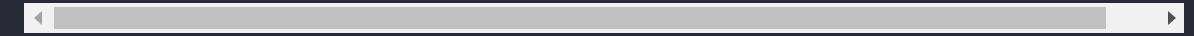
```

Next, we define a utility function to split a node into left and right children based on a feature and a value:

```

>>> def split_node(X, y, index, value):
    """
    ...
    ...     Split dataset X, y based on a feature and a value
    ...     @param X: numpy.ndarray, dataset feature
    ...     @param y: numpy.ndarray, dataset target
    ...     @param index: int, index of the feature used for splitting
    ...     @param value: value of the feature used for splitting
    ...     @return: list, list, left and right child, a child is in
              the format of [X, y]
    """
    ...
    x_index = X[:, index]
    # if this feature is numerical
    if X[0, index].dtype.kind in ['i', 'f']:
        mask = x_index >= value
    # if this feature is categorical
    else:
        mask = x_index == value
    # split into left and right child
    left = [X[~mask, :], y[~mask]]
    right = [X[mask, :], y[mask]]
    return left, right

```



We check whether the feature is numerical or categorical and split the data accordingly.

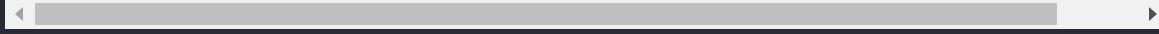
With the splitting measurement and generation functions available, we now define the greedy search function, which tries out all possible splits and returns the best one given a selection criterion, along with the resulting children:

```

>>> def get_best_split(X, y, criterion):
    """
    ...
    ...     Obtain the best splitting point and resulting children for
              the dataset X,
    ...     @param X: numpy.ndarray, dataset feature
    ...     @param y: numpy.ndarray, dataset target
    ...     @param criterion: gini or entropy
    ...     @return: dict {index: index of the feature, value: feature
                   value, children: left and right children}
    """
    ...
    best_index, best_value, best_score, children =
        None, None, 1, None
    for index in range(len(X[0])):
        for value in np.unique(X[:, index]):
            ...

```

```
...         for value in np.sort(np.unique(X[:, index])):
...             groups = split_node(X, y, index, value)
...             impurity = weighted_impurity(
...                 [groups[0][1], groups[1][1]], criter
...             if impurity < best_score:
...                 best_index, best_value, best_score, childre
...                     index, value, impurity, group
...             return {'index': best_index, 'value': best_value,
...                     'children': children}
```



The selection and splitting process occurs in a recursive manner on each of the subsequent children. When a stopping criterion is met, the process stops at a node, and the major label is assigned to this leaf node:

```
>>> def get_leaf(labels):
...     # Obtain the leaf as the majority of the labels
...     return np.bincount(labels).argmax()
```

And, finally, the recursive function links all of them together:

- It assigns a leaf node if one of two child nodes is empty.
- It assigns a leaf node if the current branch depth exceeds the maximum depth allowed.
- It assigns a leaf node if the node does not contain sufficient samples required for a further split.
- Otherwise, it proceeds with a further split with the optimal splitting point.

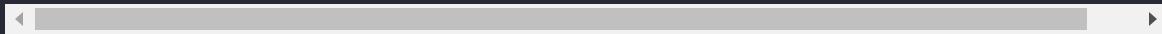
This is done with the following function:

```
>>> def split(node, max_depth, min_size, depth, criterion):
...     """
...         Split children of a node to construct new nodes or assig
...         them terminals
...         @param node: dict, with children info
...         @param max_depth: int, maximal depth of the tree
...         @param min_size: int, minimal samples required to furthe
...                         split a child
...         @param depth: int, current depth of the node
...         @param criterion: function to determine if a node is a leaf
```

```

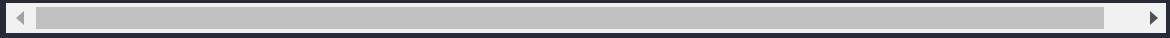
...
    @param criterion: gini or entropy
...
    """
...
    left, right = node['children']
    del (node['children'])
    if left[1].size == 0:
        node['right'] = get_leaf(right[1])
        return
    if right[1].size == 0:
        node['left'] = get_leaf(left[1])
        return
    # Check if the current depth exceeds the maximal depth
    if depth >= max_depth:
        node['left'], node['right'] =
            get_leaf(left[1]), get_leaf(right[1])
        return
    # Check if the left child has enough samples
    if left[1].size <= min_size:
        node['left'] = get_leaf(left[1])
    else:
        # It has enough samples, we further split it
        result = get_best_split(left[0], left[1], criterion)
        result_left, result_right = result['children']
        if result_left[1].size == 0:
            node['left'] = get_leaf(result_right[1])
        elif result_right[1].size == 0:
            node['left'] = get_leaf(result_left[1])
        else:
            node['left'] = result
            split(node['left'], max_depth, min_size,
                  depth + 1, criterion)
    # Check if the right child has enough samples
    if right[1].size <= min_size:
        node['right'] = get_leaf(right[1])
    else:
        # It has enough samples, we further split it
        result = get_best_split(right[0], right[1], criterion)
        result_left, result_right = result['children']
        if result_left[1].size == 0:
            node['right'] = get_leaf(result_right[1])
        elif result_right[1].size == 0:
            node['right'] = get_leaf(result_left[1])
        else:
            node['right'] = result
            split(node['right'], max_depth, min_size,
                  depth + 1, criterion)

```



Finally, the entry point of the tree's construction is as follows:

```
>>> def train_tree(X_train, y_train, max_depth, min_size,
...                  criterion='gini'):
...     """
...     Construction of a tree starts here
...     @param X_train: list of training samples (feature)
...     @param y_train: list of training samples (target)
...     @param max_depth: int, maximal depth of the tree
...     @param min_size: int, minimal samples required to further
...                      split a child
...     @param criterion: gini or entropy
...     """
...     X = np.array(X_train)
...     y = np.array(y_train)
...     root = get_best_split(X, y, criterion)
...     split(root, max_depth, min_size, 1, criterion)
...     return root
```



Now, let's test it with the preceding hand-calculated example:

```
>>> X_train = [['tech', 'professional'],
...              ['fashion', 'student'],
...              ['fashion', 'professional'],
...              ['sports', 'student'],
...              ['tech', 'student'],
...              ['tech', 'retired'],
...              ['sports', 'professional']]
>>> y_train = [1, 0, 0, 0, 1, 0, 1]
>>> tree = train_tree(X_train, y_train, 2, 2)
```

To verify that the resulting tree from the model is identical to what we constructed by hand, we write a function displaying the tree:

```
>>> CONDITION = {'numerical': {'yes': '>=', 'no': '<'},
...                 'categorical': {'yes': 'is', 'no': 'is not'}}
>>> def visualize_tree(node, depth=0):
...     if isinstance(node, dict):
...         if node['value'].dtype.kind in ['i', 'f']:
...             condition = CONDITION['numerical']
...         else:
```

... condition = CONDITION['categorical']

```

...
        condition = CONDITION['categorical']
...
    print('{}|- X{} {} {}'.format(depth * ' ', 
...
        node['index'] + 1, condition['no'], node['value'])
...
    if 'left' in node:
        visualize_tree(node['left'], depth + 1)
    print('{}|- X{} {} {}'.format(depth * ' ', 
...
        node['index'] + 1, condition['yes'], node['value'])
...
    if 'right' in node:
        visualize_tree(node['right'], depth + 1)
...
    else:
        print(f"{depth * ' '}[{node}]")
>>> visualize_tree(tree)
|- X1 is not fashion
| |- X2 is not professional
| | [0]
| |- X2 is professional
| | [1]
|- X1 is fashion
[0]

```

We can test it with a numerical example, as follows:

```

>>> X_train_n = [[6, 7],
...
            [2, 4],
...
            [7, 2],
...
            [3, 6],
...
            [4, 7],
...
            [5, 2],
...
            [1, 6],
...
            [2, 0],
...
            [6, 3],
...
            [4, 1]]
>>> y_train_n = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
>>> tree = train_tree(X_train_n, y_train_n, 2, 2)
>>> visualize_tree(tree)
|- X2 < 4
| |- X1 < 7
| | [1]
| |- X1 >= 7
| | [0]
|- X2 >= 4
| |- X1 < 2
| | [1]
| |- X1 >= 2
| | [0]

```

The resulting trees from our decision tree model are the same as those we hand-crafted.

Now that you have a more solid understanding of decision trees after implementing one from scratch, we can move on with implementing a decision tree with scikit-learn.

Implementing a decision tree with scikit-learn

Here, we use the decision tree module (<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html>), which is already well developed and optimized:

```
>>> from sklearn.tree import DecisionTreeClassifier  
>>> tree_sk = DecisionTreeClassifier(criterion='gini',  
...                                     max_depth=2, min_samples_split=5)  
>>> tree_sk.fit(X_train_n, y_train_n)
```

To visualize the tree we just built, we utilize the built-in `export_graphviz` function, as follows:

```
>>> export_graphviz(tree_sk, out_file='tree.dot',  
...                   feature_names=['X1', 'X2'], impurity=False,  
...                   filled=True, class_names=['0', '1'])
```

Running this will generate a file called `tree.dot`, which can be converted into a PNG image file using **Graphviz** (the introduction and installation instructions can be found at <http://www.graphviz.org>) by running the following command in the terminal:

```
dot -Tpng tree.dot -o tree.png
```

Refer to *Figure 4.11* for the result:

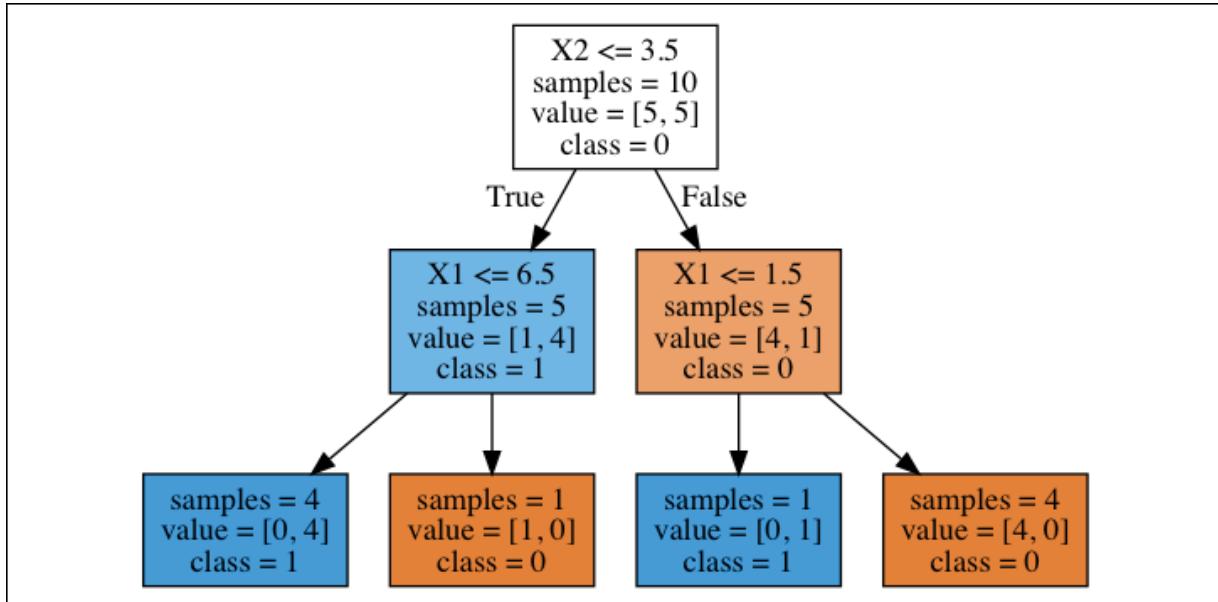


Figure 4.11: Tree visualization

The generated tree is essentially the same as the one we had before.

I know you can't wait to employ a decision tree to predict ad click-through. Let's move on to the next section.

Predicting ad click-through with a decision tree

After several examples, it is now time to predict ad click-through using the decision tree algorithm you have just thoroughly learned about and practiced. We will use the dataset from a Kaggle machine learning competition, *Click-Through Rate Prediction* (<https://www.kaggle.com/c/avazu-ctr-prediction>)

[**prediction**](#)). The dataset can be downloaded from <https://www.kaggle.com/c/avazu-ctr-prediction/data>.



Only the `train.gz` file contains labeled samples, so we only need to download this and unzip it (it will take a while). In this chapter, we will focus on only the first 300,000 samples from the `train` file unzipped from `train.gz`.

The fields in the raw file are as follows:

Field	Description	Example values
id	ad identifier	such as '1000009418151094273', '10000169349117863715'
click	'0' for non-click, '1' for click	0, 1
hour	in the format of YYMMDDHH	'14102100'
C1	anonymized categorical variable	'1005', '1002'
banner_pos	where banner is located	1, 0
site_id	site identifier	'1fbe01fe', 'fe8cc448', 'd6137915'
site_domain	hashed site domain	'bb1ef334', 'f3845767'
site_category	hashed site category	'28905ebd', '28905ebd'
app_id	mobile app identifier	'ecad2386'
app_domain	mobile app domain	'7801e8d9'
app_category	category of app	'07d7df22'
device_id	mobile device identifier	'a99f214a'
device_ip	IP address	'ddd2926e'
device_model	such as iphone 6, Samsung, hashed	'44956a24'
device_type	such as tablet, smartphone, hashed	1
device_conn_type	Wi-Fi or 3G for example, again hashed in the data	0, 2
C14-C21	anonymized categorical variables	

Figure 4.12: Description and example values of the dataset

We take a glance at the head of the file by running the following command:

```
head train | sed 's/,/, ,/g;s/,/, ,/g' | column -s, -t
```

Rather than a simple `head train`, the output is cleaner as all the columns are aligned:

<code>id</code>	<code>click</code>	<code>hour</code>	<code>C1</code>	<code>banner_pos</code>	<code>site_id</code>	<code>site_domain</code>	<code>site_category</code>	<code>app_id</code>	<code>C14</code>	<code>C15</code>
<code>app_domain</code>	<code>app_category</code>	<code>device_id</code>	<code>device_ip</code>	<code>device_model</code>	<code>device_type</code>	<code>device_conn_type</code>				
<code>C16</code>	<code>C17</code>	<code>C18</code>	<code>C19</code>	<code>C20</code>	<code>C21</code>					
1000009418151094273	0	14102100	1005 0		1fbe01fe	f3845767	28905ebd	ecad2386		
7801e8d9	07d7df22	a99f214a	ddd2926e	44956a24	1		2		15706	320
50	1722	0	35	-1	79					
10000169349117863715	0	14102100	1005 0		1fbe01fe	f3845767	28905ebd	ecad2386		
7801e8d9	07d7df22	a99f214a	96809ac8	711ee120	1		0		15704	320
50	1722	0	35	100084	79					
10000371904215119486	0	14102100	1005 0		1fbe01fe	f3845767	28905ebd	ecad2386		
7801e8d9	07d7df22	a99f214a	b3cf8def	8a4875bd	1		0		15704	320
50	1722	0	35	100084	79					
1000064072448083876	0	14102100	1005 0		1fbe01fe	f3845767	28905ebd	ecad2386		
7801e8d9	07d7df22	a99f214a	e8275b8f	6332421a	1		0		15706	320
50	1722	0	35	100084	79					
10000679056417042096	0	14102100	1005 1		fe8cc448	9166c161	0569f928	ecad2386		
7801e8d9	07d7df22	a99f214a	9644d0bf	779d90c2	1		0		18993	320
50	2161	0	35	-1	157					
10000720757801103869	0	14102100	1005 0		d6137915	bb1ef334	f028772b	ecad2386		
7801e8d9	07d7df22	a99f214a	05241af0	8a4875bd	1		0		16920	320
50	1899	0	431	100077	117					
10000724729988544911	0	14102100	1005 0		8fd4644b	25d4cfcd	f028772b	ecad2386		
7801e8d9	07d7df22	a99f214a	b264c159	be6db1d7	1		0		20362	320
50	2333	0	39	-1	157					
10000918755742328737	0	14102100	1005 1		e151e245	7e091613	f028772b	ecad2386		
7801e8d9	07d7df22	a99f214a	e6f67278	be74e6fe	1		0		20632	320
50	2374	3	39	-1	23					
10000949271186029916	1	14102100	1005 0		1fbe01fe	f3845767	28905ebd	ecad2386		
7801e8d9	07d7df22	a99f214a	37e8da74	5db079b5	1		2		15707	320
50	1722	0	35	-1	79					

Figure 4.13: The first few rows of the data

Don't be scared by the anonymized and hashed values. They are categorical features, and each possible value of them corresponds to a real and meaningful value, but it is presented this way due to privacy policy. Possibly, `C1` means user gender, and `1005` and `1002` represent male and female, respectively.

Now, let's start by reading the dataset using `pandas`. That's right, `pandas` is extremely good at handling data in a tabular format:

```
>>> import pandas as pd
>>> n_rows = 300000
>>> df = pd.read_csv("train.csv", nrows=n_rows)
```

The first 300,000 lines of the file are loaded and stored in a DataFrame. Take a quick look at the first five rows of the DataFrame:

```
>>> print(df.head(5))
id    click      hour  C1  banner_pos  site_id ...  C16  C17  C18  C19
0    1.000009e+18      0  14102100  1005          0  1fbe01fe ...  50
1    1.000017e+19      0  14102100  1005          0  1fbe01fe ...  50
2    1.000037e+19      0  14102100  1005          0  1fbe01fe ...  50
3    1.000064e+19      0  14102100  1005          0  1fbe01fe ...  50
4    1.000068e+19      0  14102100  1005          1  fe8cc448 ...  50
```

The target variable is the `click` column:

```
>>> Y = df['click'].values
```

For the remaining columns, there are several columns that should be removed from the features (`id`, `hour`, `device_id`, and `device_ip`) as they do not contain much useful information:

```
>>> X = df.drop(['click', 'id', 'hour', 'device_id', 'device_ip',
   ...:             axis=1].values
>>> print(X.shape)
(300000, 19)
```

Each sample has `19` predictive attributes.

Next, we need to split the data into training and testing sets. Normally, we do this by randomly picking samples. However, in our case, the samples are in chronological order, as indicated in the `hour` field. Obviously, we cannot use future samples to predict the past ones. Hence, we take the first 90% as training samples and the rest as testing samples:

```
>>> n_train = int(n_rows * 0.9)
>>> X_train = X[:n_train]
>>> Y_train = Y[:n_train]
>>> X_test = X[n_train:]
>>> Y_test = Y[n_train:]
```

As mentioned earlier, decision tree models can take in categorical features. However, because the tree-based algorithms in scikit-learn (the current version is 0.22.0 as of 2020) only allow numeric input, we need to transform the categorical features into numerical ones. But note that, in general, we do not need to do this; for example, the decision tree classifier we developed from scratch earlier can directly take in categorical features.

We will now transform string-based categorical features into one-hot encoded vectors using the `OneHotEncoder` module from scikit-learn. One-hot encoding was briefly mentioned in *Chapter 1, Getting Started with Machine Learning and Python*. To recap, it basically converts a categorical feature with k possible values into k binary features. For example, the site category feature with three possible values, `news`, `education`, and `sports`, will be encoded into three binary features, such as `is_news`, `is_education`, and `is_sports`, whose values are either `1` or `0`.

We initialize a `OneHotEncoder` object as follows:

```
>>> from sklearn.preprocessing import OneHotEncoder  
>>> enc = OneHotEncoder(handle_unknown='ignore')
```

We fit it on the training set as follows:

```
>>> X_train_enc = enc.fit_transform(X_train)  
>>> X_train_enc[0]  
<1x8385 sparse matrix of type '<class 'numpy.float64'>'  
with 19 stored elements in Compressed Sparse Row format>  
>>> print(X_train_enc[0])  
(0, 2) 1.0  
(0, 6) 1.0  
(0, 30) 1.0  
(0, 1471) 1.0  
(0, 2743) 1.0  
(0, 3878) 1.0  
(0, 4000) 1.0  
(0, 4048) 1.0  
(0, 6663) 1.0  
(0, 7491) 1.0  
(0, 7494) 1.0
```

```
(0, 7861) 1.0
(0, 8004) 1.
(0, 8008) 1.0
(0, 8085) 1.0
(0, 8158) 1.0
(0, 8163) 1.0
(0, 8202) 1.0
(0, 8383) 1.0
```

Each converted sample is a sparse vector.

We transform the testing set using the trained one-hot encoder as follows:

```
>>> X_test_enc = enc.transform(X_test)
```

Remember, we specified the `handle_unknown='ignore'` parameter in the one-hot encoder earlier. This is to prevent errors due to any unseen categorical values. To use the previous site category example, if there is a sample with the value `movie`, all of the three converted binary features (`is_news`, `is_education`, and `is_sports`) become `0`. If we do not specify `ignore`, an error will be raised.

Next, we will train a decision tree model using grid search, which you learned about in *Chapter 3, Recognizing Faces with Support Vector Machine*. For demonstration purposes, we will only tweak the `max_depth` hyperparameter. Other hyperparameters, such as `min_samples_split` and `class_weight`, are also highly recommended. The classification metric should be AUC of ROC, as it is an imbalanced binary case (only 51,211 out of 300,000 training samples are clicks, which is a 17% positive CTR; I encourage you to figure out the class distribution yourself):

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> parameters = {'max_depth': [3, 10, None]}
```

We pick three options for the maximal depth, `3`, `10`, and unbounded. We initialize a decision tree model with Gini Impurity as the metric and `30` as the minimum number of samples required to split further:

```
>>> decision_tree = DecisionTreeClassifier(criterion='gini',
...                                         min_samples_split=30)
>>> from sklearn.model_selection import GridSearchCV
```

As for grid search, we use three-fold (as there are enough training samples) cross-validation and select the best performing hyperparameter measured by AUC:

```
>>> grid_search = GridSearchCV(decision_tree, parameters,
...                               n_jobs=-1, cv=3, scoring='roc_auc')
```

Note `n_jobs=-1` means that we use all of the available CPU processors:

```
>>> grid_search.fit(X_train, y_train)
>>> print(grid_search.best_params_)
{'max_depth': 10}
```

We use the model with the optimal parameter to predict any future test cases as follows:

```
>>> decision_tree_best = grid_search.best_estimator_
>>> pos_prob = decision_tree_best.predict_proba(X_test)[:, 1]
>>> from sklearn.metrics import roc_auc_score
>>> print(f'The ROC AUC on testing set is: {roc_auc_score(Y_test
...           pos_prob):.3f}')
The ROC AUC on testing set is: 0.719
```

The AUC we can achieve with the optimal decision tree model is 0.72. This does not seem to be very high, but click-through involves many intricate human factors, which is why predicting it is not an easy task. Although we can further optimize the hyperparameters, an AUC of 0.72 is actually pretty good. Randomly selecting 17% of the samples to be clicked on will generate an AUC of `0.496`:

```
>>> pos_prob = np.zeros(len(Y_test))
>>> click_index = np.random.choice(len(Y_test),
...                                 int(len(Y_test) * 51211.0/300000),
...                                 replace=False)
>>> pos_prob[click_index] = 1
>>> print(f'The ROC AUC on testing set is: {roc_auc_score(Y_test
...     pos_prob):.3f}')
The ROC AUC on testing set is: 0.496
```

Looking back, we can see that a decision tree is a sequence of greedy searches for the best splitting point at each step, based on the training dataset. However, this tends to cause overfitting as it is likely that the optimal points only work well for the training samples. Fortunately, ensembling is the technique to correct this, and random forest is an ensemble tree model that usually outperforms a simple decision tree.

Ensembling decision trees – random forest

The **ensemble** technique of **bagging** (which stands for **bootstrap aggregating**), which I briefly mentioned in *Chapter 1, Getting Started with Machine Learning and Python*, can effectively overcome overfitting. To recap, different sets of training samples are randomly drawn with replacement from the original training data; each resulting set is used to fit an individual classification model. The results of these separately trained models are then combined together through a **majority vote** to make the final decision.

Tree bagging, as described in the preceding paragraph, reduces the high variance that a decision tree model suffers from and, hence, in general, performs better than a single tree. However, in some cases, where one or more features are strong indicators, individual trees are constructed largely based on these features and, as a result, become highly correlated. Aggregating multiple correlated trees will not make much difference. To

force each tree to become uncorrelated, random forest only considers a random subset of the features when searching for the best splitting point at each node. Individual trees are now trained based on different sequential sets of features, which guarantees more diversity and better performance. Random forest is a variant of the tree bagging model with additional **feature-based bagging**.

To employ random forest in our click-through prediction project, we can use the package from scikit-learn. Similarly to the way we implemented the decision tree in the preceding section, we only tweak the `max_depth` parameter:

```
>>> from sklearn.ensemble import RandomForestClassifier  
>>> random_forest = RandomForestClassifier(n_estimators=100,  
...                                         criterion='gini', min_samples_split=30,  
...                                         n_jobs=-1)
```

Besides `max_depth`, `min_samples_split`, and `class_weight`, which are important hyperparameters related to a single decision tree, hyperparameters that are related to a random forest (a set of trees) such as `n_estimators` are also highly recommended. We fine-tune the `max_depth` as follows:

```
>>> grid_search = GridSearchCV(random_forest, parameters,  
...                               n_jobs=-1, cv=3, scoring='roc_auc'  
>>> grid_search.fit(X_train, y_train)  
>>> print(grid_search.best_params_)  
{'max_depth': None}
```

We use the model with the optimal parameter `None` for `max_depth` (the nodes are expanded until another stopping criterion is met) to predict any future unseen cases:

```
>>> random_forest_best = grid_search.best_estimator  
>>> pos_prob = random_forest_best.predict_proba(X_test)[:, 1]  
>>> print('The ROC AUC on testing set is:')
```

```
... {0:.3f}").format(roc_auc_score(y_test, pos_prop)))  
The ROC AUC on testing set is: 0.759
```

It turns out that the random forest model gives a substantial lift to the performance.

Let's summarize several critical hyperparameters to tune:

- `max_depth` : This is the deepest individual tree. It tends to overfit if it is too deep or underfit if it is too shallow.
- `min_samples_split` : This hyperparameter represents the minimum number of samples required for further splitting at a node. Too small a value tends to cause overfitting, while too large a value is likely to introduce underfitting. `10` , `30` , and `50` might be good options to start with.

The preceding two hyperparameters are generally related to individual decision trees. The following two parameters are more related to a random forest or collection of trees:

- `max_features` : This parameter represents the number of features to consider for each best splitting point search. Typically, for an m -dimensional dataset, \sqrt{m} (rounded) is a recommended value for `max_features`. This can be specified as `max_features="sqrt"` in scikit-learn. Other options include `log2` , 20%, and 50% of the original features.
- `n_estimators` : This parameter represents the number of trees considered for majority voting. Generally speaking, the more trees, the better the performance but the longer the computation time. It is usually set as `100` , `200` , `500` , and so on.

Next, we'll discuss gradient boosted trees.

Ensembling decision trees – gradient boosted trees

Boosting, which is another ensemble technique, takes an iterative approach instead of combining multiple learners in parallel. In boosted trees, individual trees are no longer trained separately. Specifically, in **gradient boosted trees (GBT)** (also called **gradient boosting machines**), individual trees are trained in succession where a tree aims to correct the errors made by the previous tree. The following two diagrams illustrate the difference between random forest and GBT:

Random forest builds each tree independently using a different subset of the dataset, and then combines the results at the end by majority votes or averaging:

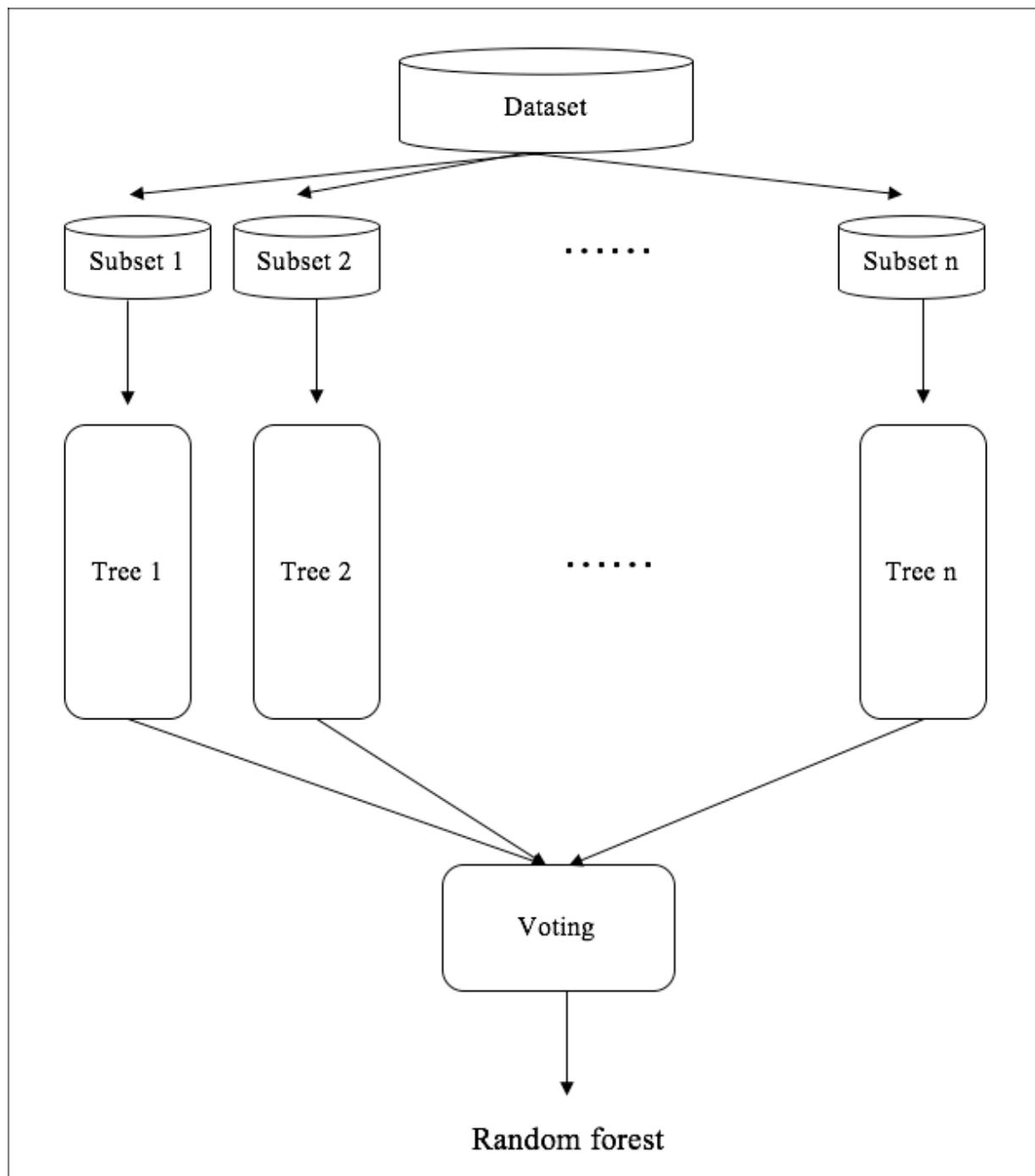


Figure 4.14: The random forest workflow

The GBT model builds one tree at a time and combines the results along the way:

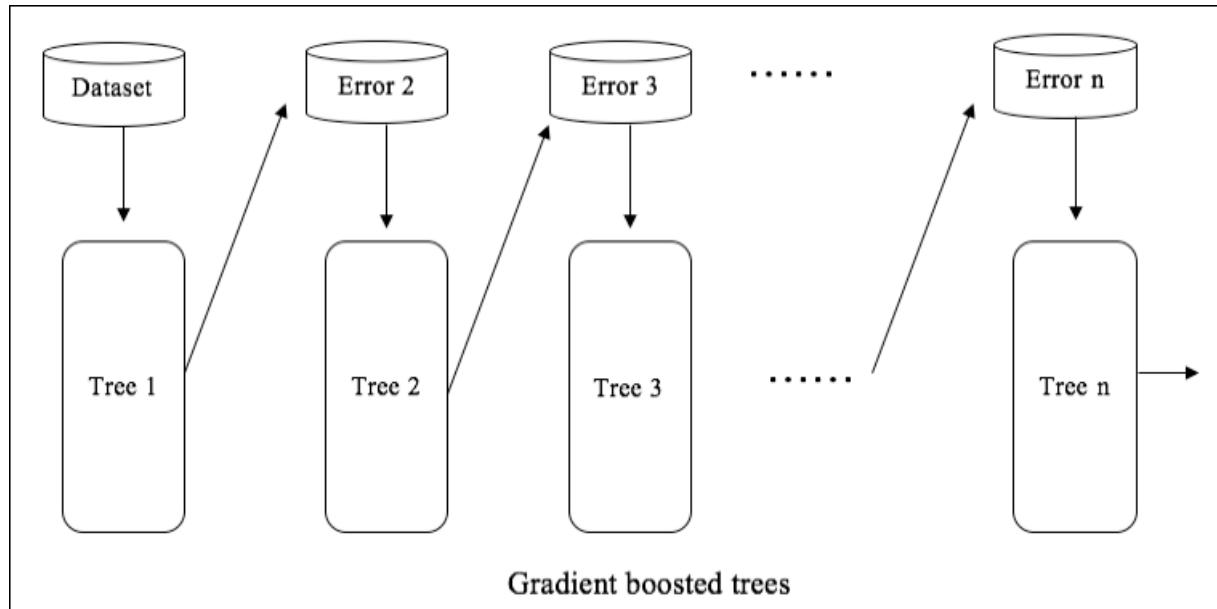


Figure 4.15: The GBT workflow

We will use the XGBoost package (<https://xgboost.readthedocs.io/en/latest/>) to implement GBT. We first install the XGBoost Python API via the following command:

```
pip install xgboost
```

If you run into a problem, please install or upgrade CMake, as follows:

```
pip install CMake
```

Let's now take a look at the following steps. You will see how we predict clicks using GBT:

1. First, we transform the label variable into two dimensions, which means 0 will become [1, 0] and 1 will become [0, 1]:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> le = LabelEncoder()
>>> Y_train_enc = le.fit_transform(Y_train)
```

2. We import XGBoost and initialize a GBT model:

```
>>> import xgboost as xgb  
>>> model = xgb.XGBClassifier(learning_rate=0.1, max_depth=10,  
... n_estimators=1000)
```

We set the learning rate to 0.1, which determines how fast or slow we want to proceed with learning in each step (in each tree, in GBT). We will discuss the learning rate in more detail in *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*. `max_depth` for individual trees is set to 10. Additionally, 1,000 trees will be trained in sequence in our GBT model.

3. Next, we train the GBT model on the training set we prepared previously:

```
>>> model.fit(X_train_enc, Y_train)
```

4. We use the trained model to make predictions on the testing set and calculate the ROC AUC accordingly:

```
>>> pos_prob = model.predict_proba(X_test_enc)[:, 1]  
>>> print(f'The ROC AUC on testing set is: {roc_auc_score(Y_test, pos_prob)}')  
The ROC AUC on testing set is: 0.771
```

We are able to achieve 0.77 AUC using the XGBoost GBT model.

In this section, you learned about another type of tree ensembling, GBT, and applied it to our ad click-through prediction.

Summary

In this chapter, we started with an introduction to a typical machine learning problem, online ad click-through prediction, and its inherent challenges, including categorical features. We then looked at tree-based algorithms that can take in both numerical and categorical features.

Next, we had an in-depth discussion about the decision tree algorithm: its mechanics, its different types, how to construct a tree, and two metrics (Gini Impurity and entropy) that measure the effectiveness of a split at a node. After constructing a tree by hand, we implemented the algorithm from scratch.

You also learned how to use the decision tree package from scikit-learn and applied it to predict the CTR. We continued to improve performance by adopting the feature-based random forest bagging algorithm. Finally, the chapter ended with several ways in which to tune a random forest model, along with a bonus section in which we implemented a GBT model with XGBoost. Bagging and boosting are two approaches to model ensembling that can improve learning performance.

More practice is always good for honing your skills. I recommend that you complete the following exercises before moving on to the next chapter, where we will solve ad click-through prediction using another algorithm: **logistic regression**.

Exercises

1. In the decision tree click-through prediction project, can you also tweak other hyperparameters, such as `min_samples_split` and `class_weight`? What is the highest AUC you are able to achieve?
2. In the random forest-based click-through prediction project, can you also tweak other hyperparameters, such as `min_samples_split`, `max_features`, and `n_estimators`, in scikit-learn? What is the highest AUC you are able to achieve?
3. In the GBT-based click-through prediction project, what hyperparameters can you tweak? What is the highest AUC you are able to achieve? You can read https://xgboost.readthedocs.io/en/latest/python/python_api.html#module-xgboost.sklearn to figure it out.

5

Predicting Online Ad Click-Through with Logistic Regression

In the previous chapter, we predicted ads click-through using tree algorithms. In this chapter, we will continue our journey of tackling the billion-dollar problem. We will focus on learning a very (probably the most) scalable classification model—logistic regression. We will explore what the logistic function is, how to train a logistic regression model, adding regularization to the model, and variants of logistic regression that are applicable to very large datasets. Besides its application in classification, we will also discuss how logistic regression and random forest are used to pick significant features. You won't get bored as there will be lots of implementations from scratch with scikit-learn and TensorFlow.

In this chapter, we will cover the following topics:

- Categorical feature encoding
- The logistic function
- What is logistic regression?
- Gradient descent and stochastic gradient descent
- The implementations of logistic regression
- Click-through prediction with logistic regression
- Logistic regression with L1 and L2 regularization
- Logistic regression for feature selection
- Online learning

- Another way to select features—random forest

Converting categorical features to numerical – one-hot encoding and ordinal encoding

In *Chapter 4, Predicting Online Ad Click-Through with Tree-Based Algorithms*, I mentioned how **one-hot encoding** transforms categorical features to numerical features in order to use them in the tree algorithms in scikit-learn and TensorFlow. If we transform categorical features into numerical ones using one-hot encoding, we don't limit our choice of algorithms to the tree-based ones that can work with categorical features.

The simplest solution we can think of in terms of transforming a categorical feature with k possible values is to map it to a numerical feature with values from 1 to k . For example,

[Tech, Fashion, Fashion, Sports, Tech, Tech, Sports] becomes [1, 2, 2, 3, 1, 1, 3]. However, this will impose an ordinal characteristic, such as Sports being greater than Tech, and a distance property, such as Sports being closer to Fashion than to Tech.

Instead, one-hot encoding converts the categorical feature to k binary features. Each binary feature indicates the presence or absence of a corresponding possible value. Hence, the preceding example becomes the following:

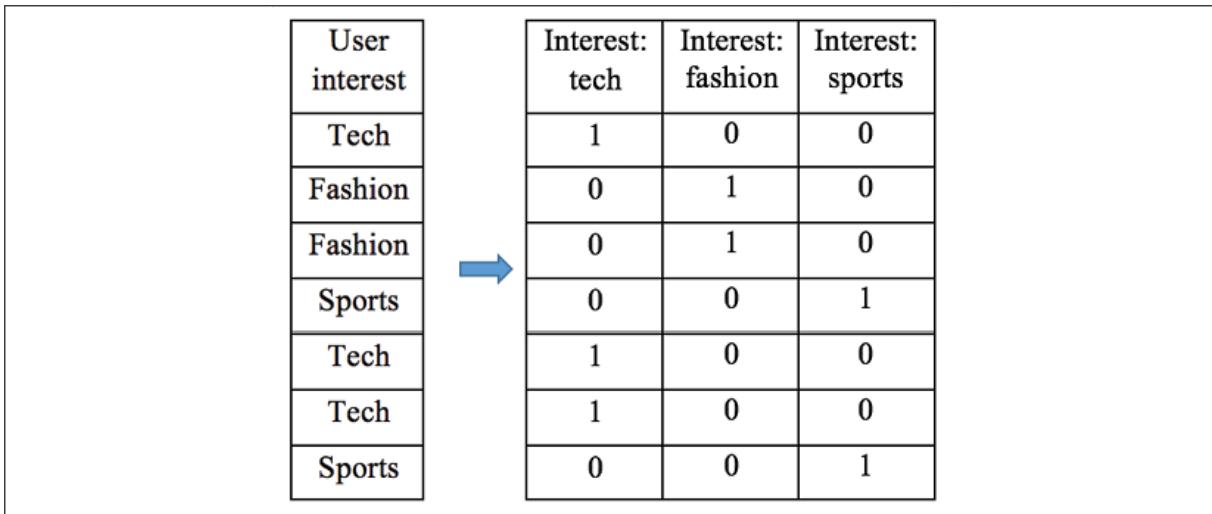


Figure 5.1: Transforming user interest into numerical features with one-hot encoding

Previously, we have used `OneHotEncoder` from scikit-learn to convert a matrix of strings into a binary matrix, but here, let's take a look at another module, `DictVectorizer`, which also provides an efficient conversion. It transforms dictionary objects (categorical feature: value) into one-hot encoded vectors.

For example, take a look at the following code:

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> X_dict = [{'interest': 'tech', 'occupation': 'professional'},
...             {'interest': 'fashion', 'occupation': 'student'},
...             {'interest': 'fashion', 'occupation': 'professional'},
...             {'interest': 'sports', 'occupation': 'student'},
...             {'interest': 'tech', 'occupation': 'student'},
...             {'interest': 'tech', 'occupation': 'retired'},
...             {'interest': 'sports', 'occupation': 'professional'}
>>> dict_one_hot_encoder = DictVectorizer(sparse=False)
>>> X_encoded = dict_one_hot_encoder.fit_transform(X_dict)
>>> print(X_encoded)
[[ 0.  0.  1.  1.  0.  0.]
 [ 1.  0.  0.  0.  0.  1.]
 [ 1.  0.  0.  1.  0.  0.]
 [ 0.  1.  0.  0.  0.  1.]
 [ 0.  0.  1.  0.  0.  1.]
 [ 0.  0.  1.  0.  1.  0.]
 [ 0.  1.  0.  1.  0.  0.]]
```

We can also see the mapping by executing the following:

```
>>> print(dict_one_hot_encoder.vocabulary_)
{'interest=fashion': 0, 'interest=sports': 1,
'occupation=professional': 3, 'interest=tech': 2,
'occupation=retired': 4, 'occupation=student': 5}
```

When it comes to new data, we can transform it with the following:

```
>>> new_dict = [{'interest': 'sports', 'occupation': 'retired'}]
>>> new_encoded = dict_one_hot_encoder.transform(new_dict)
>>> print(new_encoded)
[[ 0.  1.  0.  0.  1.  0.]]
```

We can inversely transform the encoded features back to the original features like this:

```
>>> print(dict_one_hot_encoder.inverse_transform(new_encoded))
[{'interest=sports': 1.0, 'occupation=retired': 1.0}]
```

One important thing to note is that if a new (not seen in training data) category is encountered in new data, it should be ignored (otherwise, the encoder will complain about the unseen categorical value). `DictVectorizer` handles this implicitly (while `OneHotEncoder` needs to specify the parameter `ignore`):

```
>>> new_dict = [{'interest': 'unknown_interest',
                 'occupation': 'retired'},
...                 {'interest': 'tech', 'occupation':
                 'unseen_occupation'}]
>>> new_encoded = dict_one_hot_encoder.transform(new_dict)
>>> print(new_encoded)
[[ 0.  0.  0.  0.  1.  0.]
 [ 0.  0.  1.  0.  0.  0.]]
```

Sometimes, we prefer transforming a categorical feature with k possible values into a numerical feature with values ranging from 1 to k . We

conduct **ordinal encoding** in order to employ ordinal or ranking knowledge in our learning; for example, large, medium, and small become 3, 2, and 1, respectively; good and bad become 1 and 0, while one-hot encoding fails to preserve such useful information. We can realize ordinal encoding easily through the use of `pandas`, for example:

```
>>> import pandas as pd
>>> df = pd.DataFrame({'score': ['low',
...                               'high',
...                               'medium',
...                               'medium',
...                               'low']})
>>> print(df)
   score
0    low
1   high
2  medium
3  medium
4    low
>>> mapping = {'low':1, 'medium':2, 'high':3}
>>> df['score'] = df['score'].replace(mapping)
>>> print(df)
   score
0      1
1      3
2      2
3      2
4      1
```

We convert the string feature into ordinal values based on the mapping we define.

We've covered transforming categorical features into numerical ones. Next, we will talk about logistic regression, a classifier that only takes in numerical features.

Classifying data with logistic regression

In the last chapter, we trained the tree-based models only based on the first 300,000 samples out of 40 million. We did so simply because training a tree on a large dataset is extremely computationally expensive and time-consuming. Since we are now not limited to algorithms directly taking in categorical features thanks to one-hot encoding, we should turn to a new algorithm with high scalability for large datasets. As mentioned, logistic regression is one of the most, or perhaps the most, scalable classification algorithms.

Getting started with the logistic function

Let's start with an introduction to the **logistic function** (which is more commonly referred to as the **sigmoid function**) as the algorithm's core before we dive into the algorithm itself. It basically maps an input to an output of a value between *0* and *1*, and is defined as follows:

$$y(z) = \frac{1}{1 + \exp(-z)}$$

We can visualize what it looks like by performing the following steps:

1. Define the logistic function:

```
>>> import numpy as np
>>> def sigmoid(input):
...     return 1.0 / (1 + np.exp(-input))
```

2. Input variables from `-8` to `8`, and the corresponding output, as follows:

```
>>> z = np.linspace(-8, 8, 1000)
>>> y = sigmoid(z)
>>> import matplotlib.pyplot as plt
```

```

>>> plt.plot(z, y)
>>> plt.axhline(y=0, ls='dotted', color='k')
>>> plt.axhline(y=0.5, ls='dotted', color='k')
>>> plt.axhline(y=1, ls='dotted', color='k')
>>> plt.yticks([0.0, 0.25, 0.5, 0.75, 1.0])
>>> plt.xlabel('z')
>>> plt.ylabel('y(z)')
>>> plt.show()

```

Refer to the following screenshot for the end result:

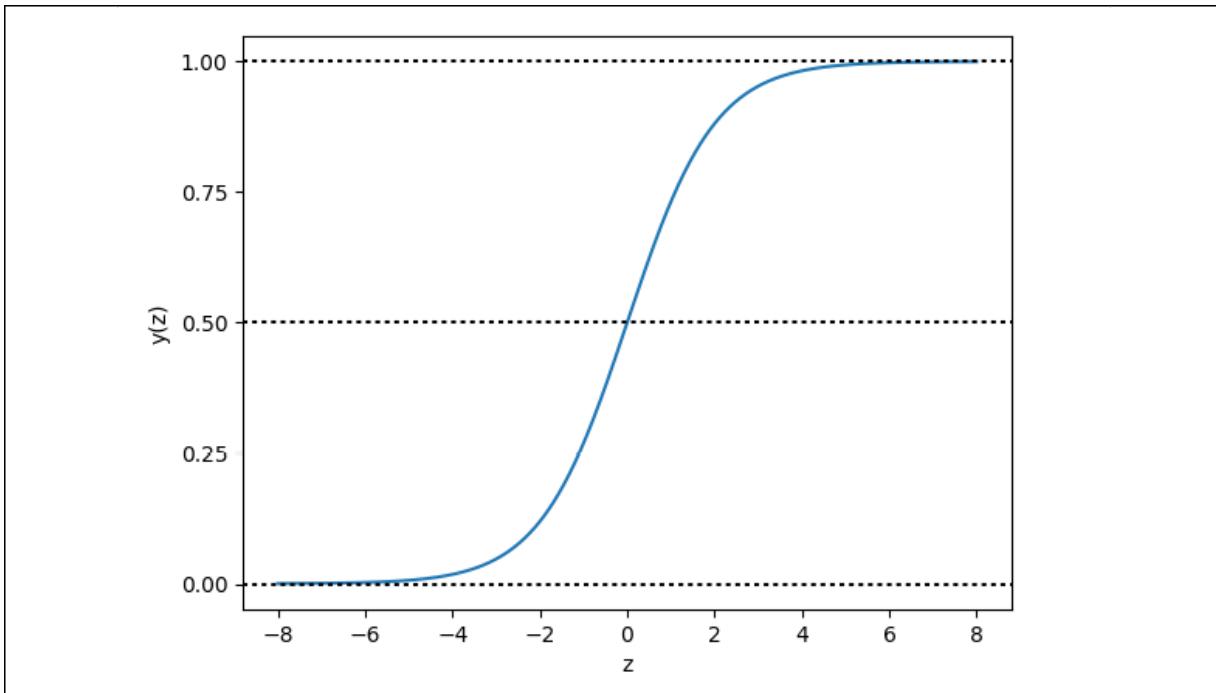


Figure 5.2: The logistic function

In the S-shaped curve, all inputs are transformed into the range from 0 to 1. For positive inputs, a greater value results in an output closer to 1; for negative inputs, a smaller value generates an output closer to 0; when the input is 0, the output is the midpoint, 0.5.

Jumping from the logistic function to logistic regression

Now that you have some knowledge of the logistic function, it is easy to map it to the algorithm that stems from it. In logistic regression, the function input z becomes the weighted sum of features. Given a data sample x with n features, x_1, x_2, \dots, x_n (x represents a feature vector and $x = (x_1, x_2, \dots, x_n)$), and **weights** (also called **coefficients**) of the model w (w represents a vector (w_1, w_2, \dots, w_n)), z is expressed as follows:

$$z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n = w^T x$$

Also, occasionally, the model comes with an **intercept** (also called **bias**), w_0 . In this instance, the preceding linear relationship becomes:

$$z = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_n x_n = w^T x$$

As for the output $y(z)$ in the range of 0 to 1, in the algorithm, it becomes the probability of the target being 1 or the positive class:

$$\hat{y} = P(y = 1 | x) = \frac{1}{1 + \exp(-w^T x)}$$

Hence, logistic regression is a probabilistic classifier, similar to the Naïve Bayes classifier.

A logistic regression model or, more specifically, its weight vector w is learned from the training data, with the goal of predicting a positive sample as close to 1 as possible and predicting a negative sample as close to 0 as possible. In mathematical language, the weights are trained so as to minimize the cost defined as the **mean squared error (MSE)**, which measures the average of squares of the difference between the truth and the prediction. Given m training samples, $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots, (x^{(i)}, y^{(i)}) \dots, (x^{(m)}, y^{(m)})$, where $y^{(i)}$ is either 1 (positive class) or 0 (negative class), the cost function $J(w)$ regarding the weights to be optimized is expressed as follows:

$$J(w) = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (\hat{y}(x^{(i)}) - y^{(i)})^2$$

However, the preceding cost function is **non-convex**, which means that, when searching for the optimal w , many local (suboptimal) optimums are found and the function does not converge to a global optimum.

Examples of the **convex** and **non-convex** functions are plotted respectively below:

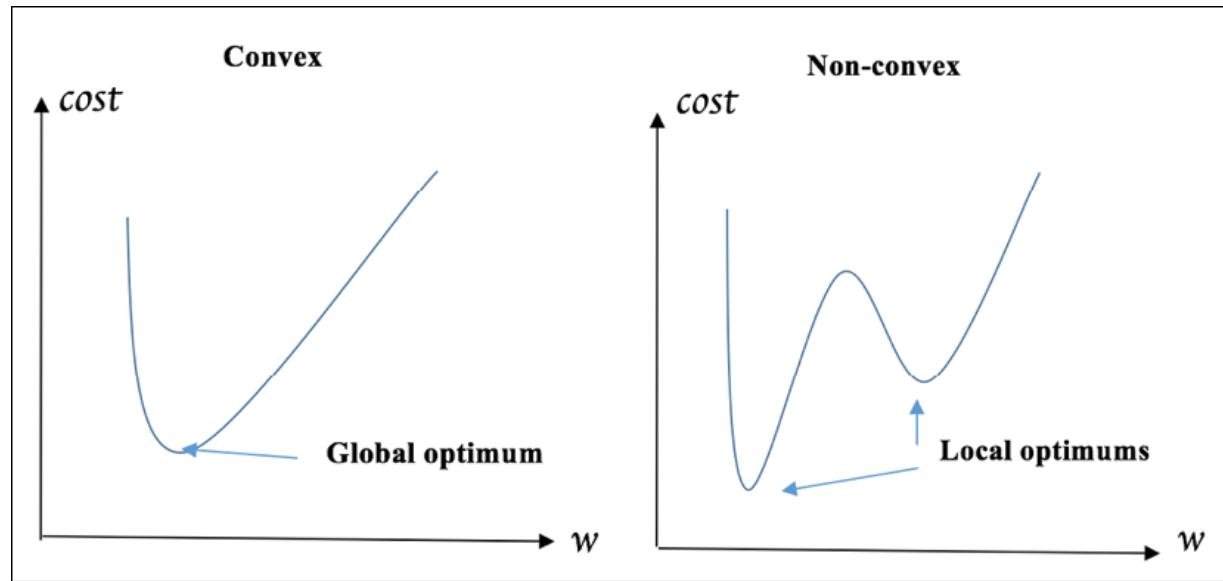


Figure 5.3: Examples of convex and non-convex functions

In the convex example, there is only one global optimum, while there are two optimums in the non-convex example. For more about convex and non-convex functions, feel free to check out

https://en.wikipedia.org/wiki/Convex_function and
<https://web.stanford.edu/class/ee364a/lectures/functions.pdf>.

To overcome this, the cost function in practice is defined as follows:

$$J(w) = \frac{1}{m} \sum_{i=1}^m -[y^{(i)} \log(\hat{y}(x^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)}))]$$

We can take a closer look at the cost of a single training sample:

$$\begin{aligned}
 j(\mathbf{w}) &= -y^{(i)} \log(\hat{y}(\mathbf{x}^{(i)})) - (1 - y^{(i)}) \log(1 - \hat{y}(\mathbf{x}^{(i)})) \\
 &= \begin{cases} -\log(\hat{y}(\mathbf{x}^{(i)})), & \text{if } y^{(i)} = 1 \\ -\log(1 - \hat{y}(\mathbf{x}^{(i)})), & \text{if } y^{(i)} = 0 \end{cases}
 \end{aligned}$$

When the ground truth $y^{(i)} = 1$, if the model predicts correctly with full confidence (the positive class with 100% probability), the sample cost j is 0; the cost j increases when the predicted probability \hat{y} decreases. If the model incorrectly predicts that there is no chance of the positive class, the cost is infinitely high. We can visualize it as follows:

```

>>> y_hat = np.linspace(0, 1, 1000)
>>> cost = -np.log(y_hat)
>>> plt.plot(y_hat, cost)
>>> plt.xlabel('Prediction')
>>> plt.ylabel('Cost')
>>> plt.xlim(0, 1)
>>> plt.ylim(0, 7)
>>> plt.show()

```

Refer to the following graph for the end result:

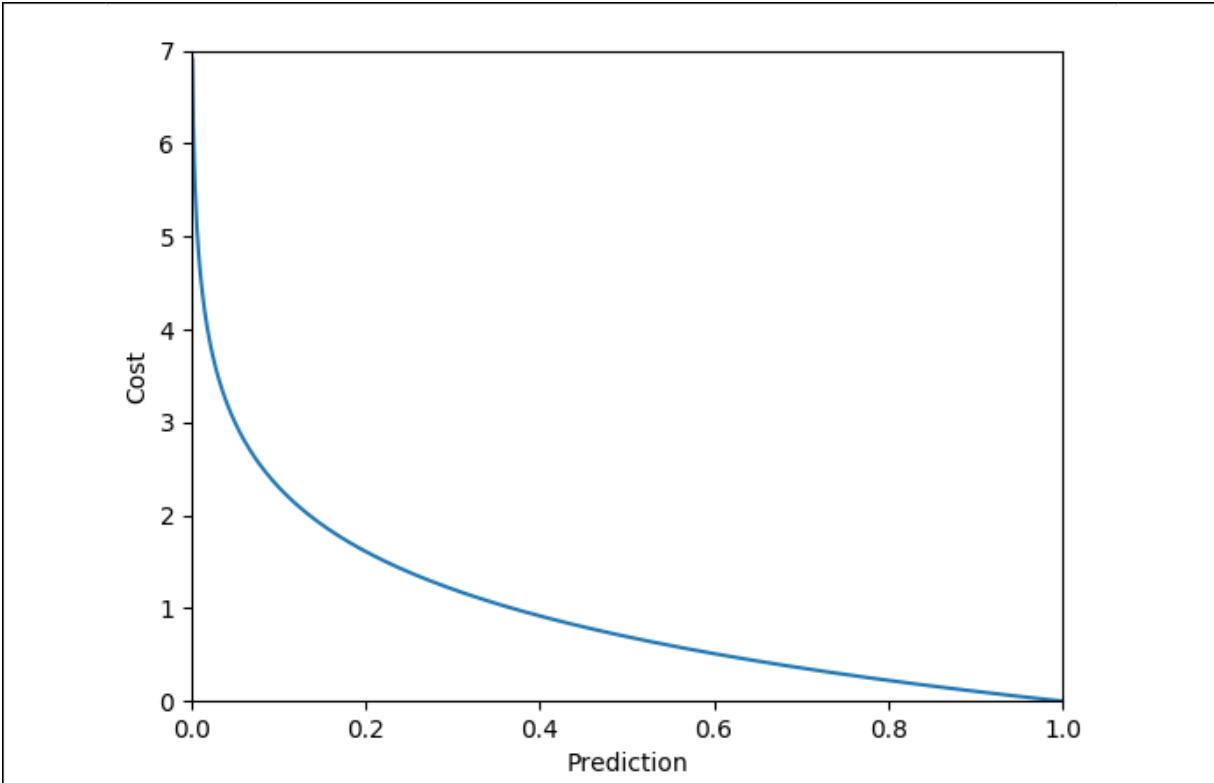


Figure 5.4: Cost function of logistic regression when $y=1$

On the contrary, when the ground truth $y^{(i)} = 0$, if the model predicts correctly with full confidence (the positive class with 0 probability, or the negative class with 100% probability), the sample cost j is 0; the cost j increases when the predicted probability \hat{y} increases. When it incorrectly predicts that there is no chance of the negative class, the cost becomes infinitely high. We can visualize it using the following code:

```
>>> y_hat = np.linspace(0, 1, 1000)
>>> cost = -np.log(1 - y_hat)
>>> plt.plot(y_hat, cost)
>>> plt.xlabel('Prediction')
>>> plt.ylabel('Cost')
>>> plt.xlim(0, 1)
>>> plt.ylim(0, 7)
>>> plt.show()
```

The following graph is the resultant output:

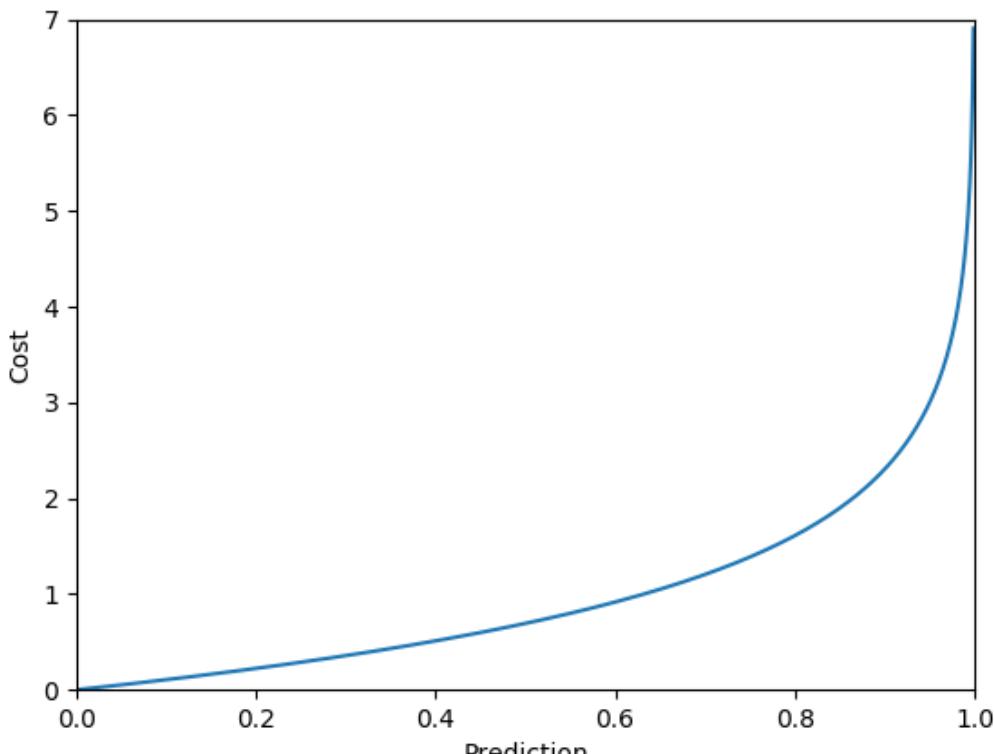


Figure 5.5: Cost function of logistic regression when $y=0$

Minimizing this alternative cost function is actually equivalent to minimizing the MSE-based cost function. The advantages of choosing it over the MSE one include the following:

- Obviously, being convex, so that the optimal model weights can be found
- A summation of the logarithms of prediction $\hat{y}(x^{(i)})$ or $1 - \hat{y}(x^{(i)})$ simplifies the calculation of its derivative with respect to the weights, which we will talk about later

Due to the logarithmic function, the cost function

$$J(w) = \frac{1}{m} \sum_{i=1}^m -[y^{(i)} \log(\hat{y}(x^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)}))]$$

is also called **logarithmic loss**, or simply **log loss**.

Now that we have the cost function ready, how can we train the logistic regression model to minimize the cost function? Let's see in the next section.

Training a logistic regression model

Now, the question is how we can obtain the optimal w such that $J(w)$ is minimized. We can do so using gradient descent.

Training a logistic regression model using gradient descent

Gradient descent (also called **steepest descent**) is a procedure of minimizing an objective function by first-order iterative optimization. In each iteration, it moves a step that is proportional to the negative derivative of the objective function at the current point. This means the to-be-optimal point iteratively moves downhill toward the minimal value of the objective function. The proportion we just mentioned is called the **learning rate**, or **step size**. It can be summarized in a mathematical equation as follows:

$$w := w - \eta \Delta w$$

Here, the left w is the weight vector after a learning step, and the right w is the one before moving, η is the learning rate, and Δw is the first-order derivative, the gradient.

In our case, let's start with the derivative of the cost function $J(w)$ with respect to w . It might require some knowledge of calculus, but don't worry, we will walk through it step by step:

1. We first calculate the derivative of $\hat{y}(z)$ with respect to w . We herein take the j -th weight, w_j , as an example (note $z=w^T x$, and we omit the (i) for simplicity):

$$\begin{aligned}\frac{\partial}{\partial w_j} \hat{y}(z) &= \frac{\partial}{\partial w_j} \frac{1}{1 + \exp(-z)} = \frac{\partial}{\partial z} \frac{1}{1 + \exp(-z)} \frac{\partial}{\partial w_j} z \\ &= \frac{1}{[1 + \exp(-z)]^2} \exp(-z) \frac{\partial}{\partial w_j} z \\ &= \frac{1}{1 + \exp(-z)} \left[1 - \frac{1}{1 + \exp(-z)} \right] \frac{\partial}{\partial w_j} z = \hat{y}(z)(1 - \hat{y}(z)) \frac{\partial}{\partial w_j} z\end{aligned}$$

2. Then, we calculate the derivative of the sample cost $J(w)$ as follows:

$$\begin{aligned}\frac{\partial}{\partial w_j} J(w) &= -y \frac{\partial}{\partial w_j} \log(\hat{y}(z)) + (1 - y) \frac{\partial}{\partial w_j} \log(1 - \hat{y}(z)) \\ &= \left[-y \frac{1}{\hat{y}(z)} + (1 - y) \frac{1}{1 - \hat{y}(z)} \right] \frac{\partial}{\partial w_j} \hat{y}(z) \\ &= \left[-y \frac{1}{\hat{y}(z)} + (1 - y) \frac{1}{1 - \hat{y}(z)} \right] \hat{y}(z)(1 - \hat{y}(z)) \frac{\partial}{\partial w_j} z \\ &= (-y + \hat{y}(z))x_j\end{aligned}$$

3. Finally, we calculate the entire cost over m samples as follows:

$$\Delta w_j = \frac{\delta}{\delta w_j} J(w) = \frac{1}{m} \sum_{i=1}^m -(y^{(i)} - \hat{y}(z^{(i)}))x_j^{(i)}$$

4. We then generalize it to Δw :

$$\Delta w = \frac{1}{m} \sum_{i=1}^m -(y^{(i)} - \hat{y}(z^{(i)}))x^{(i)}$$

5. Combined with the preceding derivations, the weights can be updated as follows:

$$w := w + \eta \frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}(z^{(i)})) x^{(i)}$$

Here, w gets updated in each iteration.

6. After a substantial number of iterations, the learned w and b are then used to classify a new sample x' by means of the following equation:

$$y' = \frac{1}{1 + \exp(-w^T x')}$$

$$\begin{cases} 1, & \text{if } y' \geq 0.5 \\ 0, & \text{if } y' < 0.5 \end{cases}$$

The decision threshold is 0.5 by default, but it definitely can be other values. In a case where a false negative is, by all means, supposed to be avoided, for example, when predicting fire occurrence (the positive class) for alerts, the decision threshold can be lower than 0.5, such as 0.3, depending on how paranoid we are and how proactively we want to prevent the positive event from happening. On the other hand, when the false positive class is the one that should be evaded, for instance, when predicting the product success (the positive class) rate for quality assurance, the decision threshold can be greater than 0.5, such as 0.7, or lower than 0.5, depending on how high a standard you set.

With a thorough understanding of the gradient descent-based training and predicting process, we will now implement the logistic regression algorithm from scratch:

1. We begin by defining the function that computes the prediction $\hat{y}(x)$ with the current weights:

```
>>> def compute_prediction(X, weights):
    """
    Compute the predictions based on our model.
    """
```

```

...     compute the prediction y_hat based on current weight
...
...     """
...     z = np.dot(X, weights)
...     predictions = sigmoid(z)
...     return predictions

```

2. With this, we are able to continue with the function updating the

$w := w + \eta \frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}(z^{(i)}))x^{(i)}$

weights by one step in a gradient descent manner. Take a look at the following code:

```

>>> def update_weights_gd(X_train, y_train, weights,
                           learning_rate):
    """
    Update weights by one step
    """
    predictions = compute_prediction(X_train, weights)
    weights_delta = np.dot(X_train.T, y_train - predictions)
    m = y_train.shape[0]
    weights += learning_rate / float(m) * weights_delta
    return weights

```

3. Then, the function calculating the cost $J(w)$ is implemented as well:

```

>>> def compute_cost(X, y, weights):
    """
    Compute the cost J(w)
    """
    predictions = compute_prediction(X, weights)
    cost = np.mean(-y * np.log(predictions)
                  - (1 - y) * np.log(1 - predictions))
    return cost

```

4. Now, we connect all these functions to the model training function by executing the following:

- Updating the `weights` vector in each iteration
- Printing out the current cost for every `100` (this can be another value) iterations to ensure `cost` is decreasing and that things are on the right track

They are implemented in the following function:

```
>>> def train_logistic_regression(X_train, y_train, max_iter=1000, learning_rate=0.01, fit_intercept=False):
...     """ Train a logistic regression model
...     Args:
...         X_train, y_train (numpy.ndarray, training data)
...         max_iter (int, number of iterations)
...         learning_rate (float)
...         fit_intercept (bool, with an intercept w0 or not)
...     Returns:
...         numpy.ndarray, learned weights
...     """
...     if fit_intercept:
...         intercept = np.ones((X_train.shape[0], 1))
...         X_train = np.hstack((intercept, X_train))
...     weights = np.zeros(X_train.shape[1])
...     for iteration in range(max_iter):
...         weights = update_weights_gd(X_train, y_train, weights, learning_rate)
...         # Check the cost for every 100 (for example) iterations
...         if iteration % 100 == 0:
...             print(compute_cost(X_train, y_train, weights))
...     return weights
```

5. Finally, we predict the results of new inputs using the trained model as follows:

```
>>> def predict(X, weights):
...     if X.shape[1] == weights.shape[0] - 1:
...         intercept = np.ones((X.shape[0], 1))
...         X = np.hstack((intercept, X))
...     return compute_prediction(X, weights)
```

Implementing logistic regression is very simple, as you just saw. Let's now examine it using a toy example:

```
>>> X_train = np.array([[6, 7],
...                     [2, 4],
...                     [3, 6],
...                     [4, 7],
...                     [1, 6],
...                     [5, 2],
...                     [2, 0],
...                     [6, 3],
```

```
...      [4, 1],  
...      [7, 2]])  
>>> y_train = np.array([0,  
...                      0,  
...                      0,  
...                      0,  
...                      0,  
...                      1,  
...                      1,  
...                      1,  
...                      1,  
...                      1])
```

We train a logistic regression model for 1000 iterations, at a learning rate of 0.1 based on intercept-included weights:

```
>>> weights = train_logistic_regression(X_train, y_train,  
...                                         max_iter=1000, learning_rate=0.1, fit_intercept=True)  
0.574404237166  
0.0344602233925  
0.0182655727085  
0.012493458388  
0.00951532913855  
0.00769338806065  
0.00646209433351  
0.00557351184683  
0.00490163225453  
0.00437556774067
```

The decreasing cost means that the model is being optimized over time. We can check the model's performance on new samples as follows:

```
>>> X_test = np.array([[6, 1],  
...                     [1, 3],  
...                     [3, 1],  
...                     [4, 5]])  
>>> predictions = predict(X_test, weights)  
>>> predictions  
array([ 0.9999478 ,  0.00743991,  0.9808652 ,  0.02080847])
```

To visualize this, execute the following code:

```
>>> import matplotlib.pyplot as plt  
>>> plt.scatter(X_train[:,0], X_train[:,1], c=['b']*5+['k']*5,  
                         marker='o')
```

Blue dots are training samples from class 0, while black dots are those from class 1. Use `0.5` as the classification decision threshold:

```
>>> colours = ['k' if prediction >= 0.5 else 'b'  
                           for prediction in predictions]  
>>> plt.scatter(X_test[:,0], X_test[:,1], marker='*', c=colours)
```

Blue stars are testing samples predicted from class 0, while black stars are those predicted from class 1:

```
>>> plt.xlabel('x1')  
>>> plt.ylabel('x2')  
>>> plt.show()
```

Refer to the following screenshot for the end result:

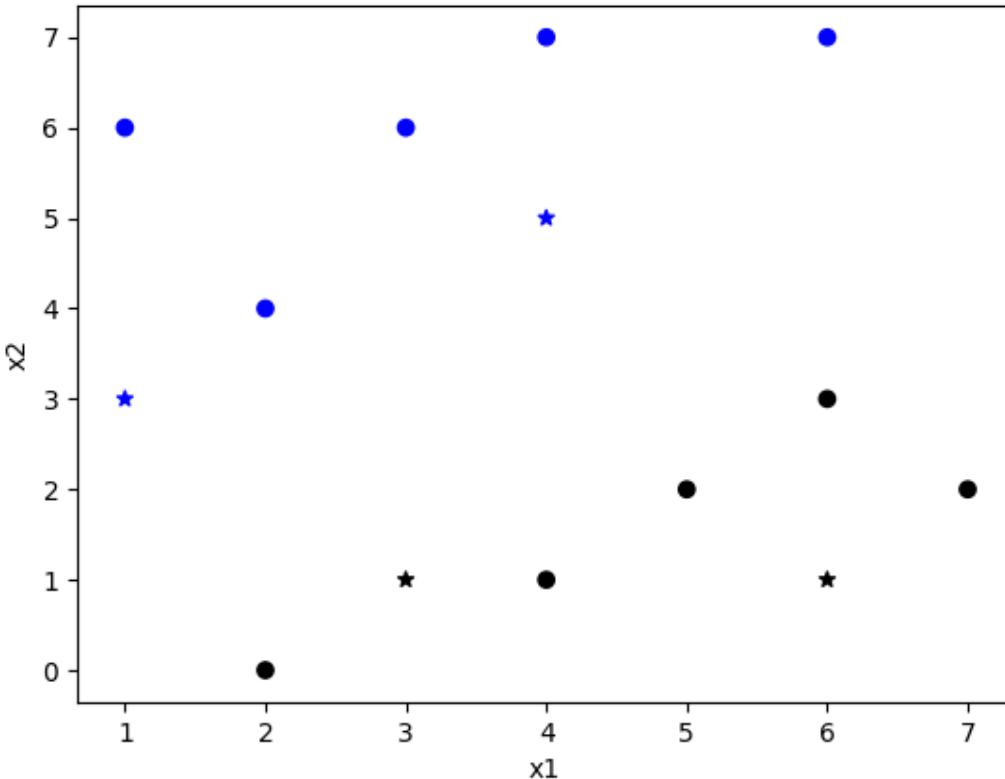


Figure 5.6: Training and testing sets of the toy example

The model we trained correctly predicts classes of new samples (the stars).

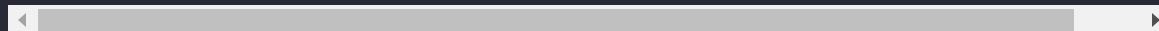
Predicting ad click-through with logistic regression using gradient descent

After this brief example, we will now deploy the algorithm we just developed in our click-through prediction project.

We herein start with only 10,000 training samples (you will soon see why we don't start with 270,000, as we did in the previous chapter):

```
>>> import pandas as pd
>>> n_rows = 300000
>>> df = pd.read_csv("train", nrows=n_rows)
```

```
>>> X = df.drop(['click', 'id', 'hour', 'device_id', 'device_ip',
   axis=1).values
>>> Y = df['click'].values
>>> n_train = 10000
>>> X_train = X[:n_train]
>>> Y_train = Y[:n_train]
>>> X_test = X[n_train:]
>>> Y_test = Y[n_train:]
>>> from sklearn.preprocessing import OneHotEncoder
>>> enc = OneHotEncoder(handle_unknown='ignore')
>>> X_train_enc = enc.fit_transform(X_train)
>>> X_test_enc = enc.transform(X_test)
```



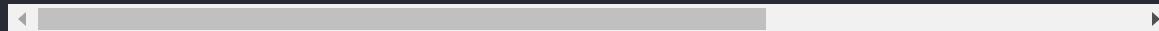
Train a logistic regression model over 10000 iterations, at a learning rate of 0.01 with bias:

```
>>> import timeit
>>> start_time = timeit.default_timer()
>>> weights = train_logistic_regression(X_train_enc.toarray(),
   Y_train, max_iter=10000, learning_rate=0.01,
   fit_intercept=True)
0.6820019456743648
0.4608619713011896
0.4503715555130051
...
...
...
0.41485094023829017
0.41477416506724385
0.41469802145452467
>>> print(f"--- {(timeit.default_timer() - start_time)}.3fs seconds ---")
--- 232.756s seconds ---
```



It takes 232 seconds to optimize the model. The trained model performs on the testing set as follows:

```
>>> pred = predict(X_test_enc.toarray(), weights)
>>> from sklearn.metrics import roc_auc_score
>>> print(f'Training samples: {n_train}, AUC on testing set: {roc_auc_score(Y_test, pred)}')
Training samples: 10000, AUC on testing set: 0.703
```



Now, let's use 100,000 training samples (`n_train = 100000`) and repeat the same process. It will take 5240.4 seconds, which is almost 1.5 hours. It takes 22 times longer to fit data of 10 times the size. As I mentioned at the beginning of the chapter, the logistic regression classifier can be good at training on large datasets. But our testing results seem to contradict this. How could we handle even larger training datasets efficiently, not just 100,000 samples, but millions? Let's look at a more efficient way to train a logistic regression model in the next section.

Training a logistic regression model using stochastic gradient descent

In gradient descent-based logistic regression models, **all** training samples are used to update the weights in every single iteration. Hence, if the number of training samples is large, the whole training process will become very time-consuming and computationally expensive, as you just witnessed in our last example.

Fortunately, a small tweak will make logistic regression suitable for large-sized datasets. For each weight update, **only one** training sample is consumed, instead of the **complete** training set. The model moves a step based on the error calculated by a single training sample. Once all samples are used, one iteration finishes. This advanced version of gradient descent is called **stochastic gradient descent (SGD)**. Expressed in a formula, for each iteration, we do the following:

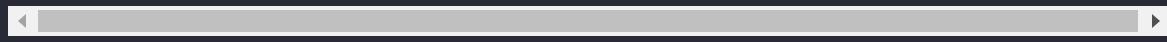
for i in 1 to m:

$$w := w + \eta(y^{(i)} - \hat{y}(z^{(i)}))x^{(i)}$$

SGD generally converges much faster than gradient descent where a large number of iterations is usually needed.

To implement SGD-based logistic regression, we just need to slightly modify the `update_weights_gd` function:

```
>>> def update_weights_sgd(X_train, y_train, weights,
                           learning_rate):
...     """ One weight update iteration: moving weights by one
...         step based on each individual sample
...     Args:
...         X_train, y_train (numpy.ndarray, training data set)
...         weights (numpy.ndarray)
...         learning_rate (float)
...     Returns:
...         numpy.ndarray, updated weights
...
...     """
...     for X_each, y_each in zip(X_train, y_train):
...         prediction = compute_prediction(X_each, weights)
...         weights_delta = X_each.T * (y_each - prediction)
...         weights += learning_rate * weights_delta
...     return weights
```



In the `train_logistic_regression` function, SGD is applied:

```
>>> def train_logistic_regression_sgd(X_train, y_train, max_iter,
                                       learning_rate, fit_intercept=False):
...     """ Train a logistic regression model via SGD
...     Args:
...         X_train, y_train (numpy.ndarray, training data set)
...         max_iter (int, number of iterations)
...         learning_rate (float)
...         fit_intercept (bool, with an intercept w0 or not)
...     Returns:
...         numpy.ndarray, learned weights
...
...     """
...     if fit_intercept:
...         intercept = np.ones((X_train.shape[0], 1))
...         X_train = np.hstack((intercept, X_train))
...     weights = np.zeros(X_train.shape[1])
...     for iteration in range(max_iter):
...         weights = update_weights_sgd(X_train, y_train, weights,
...                                       learning_rate)
...         # Check the cost for every 2 (for example) iterations
...         if iteration % 2 == 0:
...             print(compute_cost(X_train, y_train, weights))
...     return weights
```



Now, let's see how powerful SGD is. We will work with 100,000 training samples and choose 10 as the number of iterations, 0.01 as the learning rate, and print out current costs every other iteration:

```
>>> start_time = timeit.default_timer()
>>> weights = train_logistic_regression_sgd(X_train_enc.toarray(),
...     Y_train, max_iter=10, learning_rate=0.01, fit_intercept=
0.4127864859625796
0.4078504597223988
0.40545733114863264
0.403811787845451
0.4025431351250833
>>> print(f"--- {(timeit.default_timer() - start_time)}.3fs seconds ---")
--- 40.690s seconds ---
>>> pred = predict(X_test_enc.toarray(), weights)
>>> print(f'Training samples: {n_train}, AUC on testing set: {rc}
Training samples: 100000, AUC on testing set: 0.732')
```

The training process finishes in just 40 seconds!

As usual, after successfully implementing the SGD-based logistic regression algorithm from scratch, we implement it using the `SGDClassifier` module of scikit-learn:

```
>>> from sklearn.linear_model import SGDClassifier
>>> sgd_lr = SGDClassifier(loss='log', penalty=None,
...     fit_intercept=True, max_iter=10,
...     learning_rate='constant', eta0=0.01)
```

Here, 'log' for the `loss` parameter indicates that the cost function is log loss, `penalty` is the regularization term to reduce overfitting, which we will discuss further in the next section, `max_iter` is the number of iterations, and the remaining two parameters mean the learning rate is 0.01 and unchanged during the course of training. It should be noted that the default `learning_rate` is 'optimal', where the learning rate slightly decreases as more and more updates are made. This can be beneficial for finding the optimal solution on large datasets.

Now, train the model and test it:

```
>>> sgd_lr.fit(X_train_enc.toarray(), Y_train)
>>> pred = sgd_lr.predict_proba(X_test_enc.toarray())[:, 1]
>>> print(f'Training samples: {n_train}, AUC on testing set: {rc
Training samples: 100000, AUC on testing set: 0.734
```

Quick and easy!

Training a logistic regression model with regularization

As I briefly mentioned in the previous section, the `penalty` parameter in the logistic regression `SGDCclassifier` is related to model **regularization**. There are two basic forms of regularization, **L1** (also called **Lasso**) and **L2** (also called **ridge**). In either way, the regularization is an additional term on top of the original cost function:

$$J(w) = \frac{1}{m} \sum_{i=1}^m -[y^{(i)} \log(\hat{y}(x^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)}))] + \alpha ||w||^q$$

Here, α is the constant that multiplies the regularization term, and q is either 1 or 2 representing L1 or L2 regularization where the following applies:

$$||w||^1 = \sum_{j=1}^n |w_j|$$

Training a logistic regression model is the process of reducing the cost as a function of weights w . If it gets to a point where some weights, such as w_i , w_j , and w_k are considerably large, the whole cost will be determined by these large weights. In this case, the learned model may just memorize the training set and fail to generalize to unseen data. The regularization term herein is introduced in order to penalize large weights, as the weights now become part of the cost to minimize. Regularization as a result eliminates

overfitting. Finally, parameter α provides a trade-off between log loss and generalization. If α is too small, it is not able to compress large weights and the model may suffer from high variance or overfitting; on the other hand, if α is too large, the model may become over generalized and perform poorly in terms of fitting the dataset, which is the syndrome of underfitting. α is an important parameter to tune in order to obtain the best logistic regression model with regularization.

As for choosing between the L1 and L2 form, the rule of thumb is based on whether **feature selection** is expected. In machine learning classification, feature selection is the process of picking a subset of significant features for use in better model construction. In practice, not every feature in a dataset carries information that is useful for discriminating samples; some features are either redundant or irrelevant and hence can be discarded with little loss. In a logistic regression classifier, feature selection can only be achieved with L1 regularization. To understand this, let's consider two weight vectors, $w_1 = (1, 0)$ and $w_2 = (0.5, 0.5)$; supposing they produce the same amount of log loss, the L1 and L2 regularization terms of each weight vector are as follows:

$$|w_1|^1 = |1| + |0| = 1, |w_2|^1 = |0.5| + |0.5| = 1$$

$$|w_1|^2 = 1^2 + 0^2 = 1, |w_2|^2 = 0.5^2 + 0.5^2 = 0.5$$

The L1 term of both vectors is equivalent, while the L2 term of w_2 is less than that of w_1 . This indicates that L2 regularization penalizes weights composed of significantly large and small weights more than L1 regularization does. In other words, L2 regularization favors relatively small values for all weights, and avoids significantly large and small values for any weight, while L1 regularization allows some weights with a significantly small value and some with a significantly large value. Only with L1 regularization can some weights be compressed to close to or exactly 0, which enables feature selection.

In scikit-learn, the regularization type can be specified by the `penalty` parameter with the options `none` (without

regularization), "l1", "l2", and "elasticnet" (a mixture of L1 and L2), and the multiplier α can be specified by the alpha parameter.

Feature selection using L1 regularization

We herein examine L1 regularization for feature selection.

Initialize an SGD logistic regression model with L1 regularization, and train the model based on 10,000 samples:

```
>>> sgd_lr_l1 = SGDClassifier(loss='log', penalty='l1', alpha=0.  
learning_rate='constant', eta0=0.01)  
>>> sgd_lr_l1.fit(X_train_enc.toarray(), Y_train)
```

With the trained model, we obtain the absolute values of its coefficients:

```
>>> coef_abs = np.abs(sgd_lr_l1.coef_)  
>>> print(coef_abs)  
[[0. 0.09963329 0. ... 0. 0. 0.07431834]]
```

The bottom 10 coefficients and their values are printed as follows:

```
>>> print(np.sort(coef_abs)[0][:10])  
[0. 0. 0. 0. 0. 0. 0. 0. 0.]  
>>> bottom_10 = np.argsort(coef_abs)[0][:10]
```

We can see what these 10 features are using the following code:

```
>>> feature_names = enc.get_feature_names()  
>>> print('10 least important features are:\n',  
feature_names[bottom_10])  
10 least important features are:  
['x0_1001' 'x8_851897aa' 'x8_85119990' 'x8_84ebcd4' 'x8_84eb6t'  
'x8_84dda655' 'x8_84c2f017' 'x8_84ace234' 'x8_84a9d4ba' 'x8_849
```

They are `1001` from the `0` column (that is the `c1` column) in `x_train`, `"851897aa"` from the `8` column (that is the `device_model` column), and so on and so forth.

Similarly, the top 10 coefficients and their values can be obtained as follows:

```
>>> print(np.sort(coef_abs)[0][-10:])
[0.67912376 0.70885933 0.79975917 0.8828797 0.98146351 0.9827512
 1.08313767 1.13261091 1.18445527 1.40983505]
>>> top_10 = np.argsort(coef_abs)[0][-10:]
>>> print('10 most important features are:\n', feature_names[top_10])
10 most important features are:
['x7_cef3e649' 'x3_7687a86e' 'x18_61' 'x18_15' 'x5_9c13b419'
 'x5_5e3f096f' 'x2_763a42b5' 'x2_d9750ee7' 'x3_27e3c518'
 'x5_1779deee']
```

They are `"cef3e649"` from the `7` column (that is `app_category`) in `x_train`, `"7687a86e"` from the third column (that is `site_domain`), and so on and so forth.

Training on large datasets with online learning

So far, we have trained our model on no more than 300,000 samples. If we go beyond this figure, memory might be overloaded since it holds too much data, and the program will crash. In this section, we will explore how to train on a large-scale dataset with **online learning**.

SGD evolves from gradient descent by sequentially updating the model with individual training samples one at a time, instead of the complete training set at once. We can scale up SGD further with online learning techniques. In online learning, new data for training is available in sequential order or in real time, as opposed to all at once in an offline

learning environment. A relatively small chunk of data is loaded and preprocessed for training at a time, which releases the memory used to hold the entire large dataset. Besides better computational feasibility, online learning is also used because of its adaptability to cases where new data is generated in real time and is needed for modernizing the model. For instance, stock price prediction models are updated in an online learning manner with timely market data; click-through prediction models need to include the most recent data reflecting users' latest behaviors and tastes; spam email detectors have to be reactive to the ever-changing spammers by considering new features that are dynamically generated.

The existing model trained by previous datasets can now be updated based on the most recently available dataset only, instead of rebuilding it from scratch based on previous and recent datasets together, as is the case in offline learning:

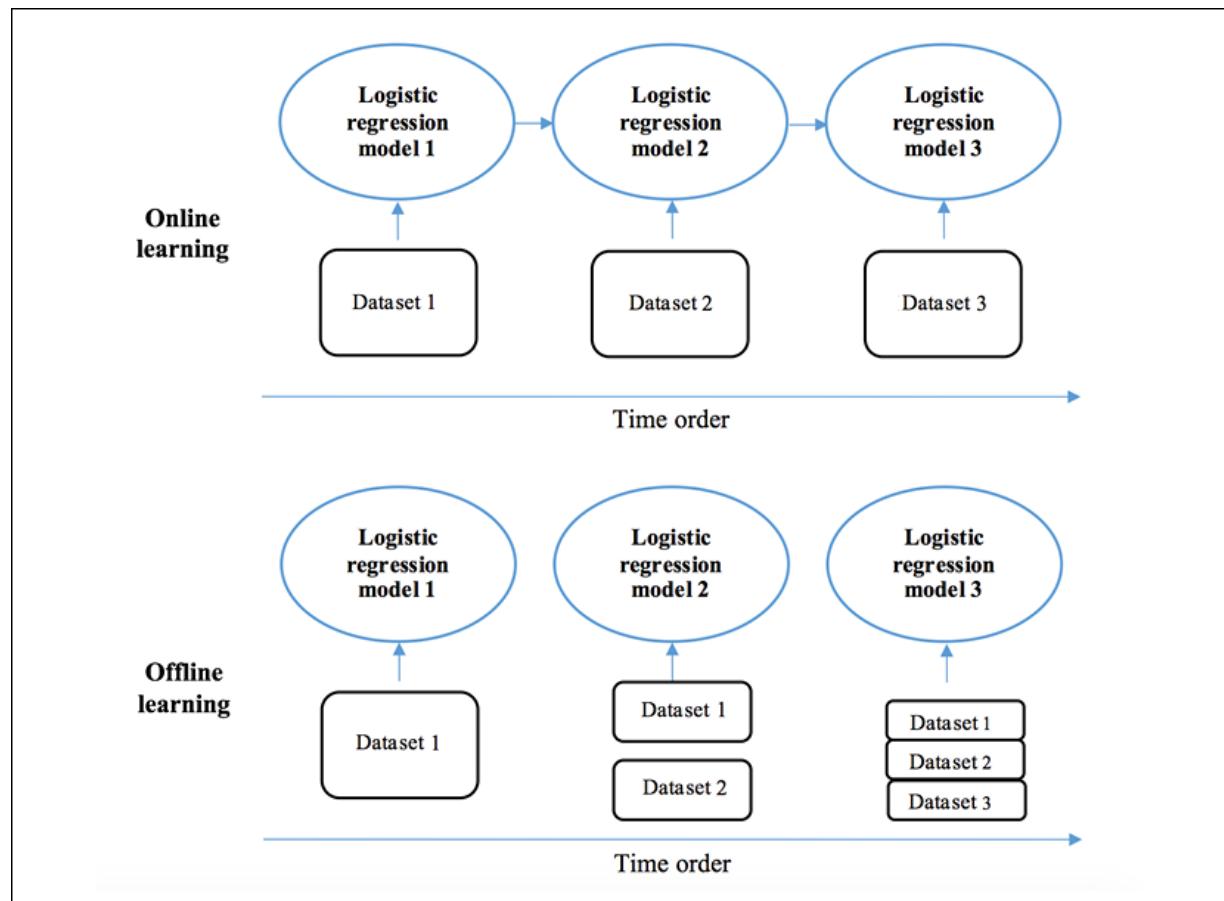


Figure 5.7: Online versus offline learning

In the preceding example, online learning allows the model to continue training with new arriving data. However, in offline learning, we have to retrain the whole model with the new arriving data along with the old data.

The `SGDClassifier` module in scikit-learn implements online learning with the `partial_fit` method (while the `fit` method is applied in offline learning, as you have seen). We will train the model with 1,000,000 samples, where we feed in 100,000 samples at one time to simulate an online learning environment. And we will test the trained model on another 100,000 samples as follows:

```
>>> n_rows = 100000 * 11
>>> df = pd.read_csv("train", nrows=n_rows)
>>> X = df.drop(['click', 'id', 'hour', 'device_id', 'device_ip',
   ...:             axis=1).values
>>> Y = df['click'].values
>>> n_train = 100000 * 10
>>> X_train = X[:n_train]
>>> Y_train = Y[:n_train]
>>> X_test = X[n_train:]
>>> Y_test = Y[n_train:]
```

Fit the encoder on the whole training set as follows:

```
>>> enc = OneHotEncoder(handle_unknown='ignore')
>>> enc.fit(X_train)
```

Initialize an SGD logistic regression model where we set the number of iterations to `1` in order to partially fit the model and enable online learning:

```
>>> sgd_lr_online = SGDClassifier(loss='log', penalty=None,
   ...:                     fit_intercept=True, max_iter=1,
   ...:                     learning_rate='constant', eta0=0.
```

Loop over every `100000` samples and partially fit the model:

```
>>> start_time = timeit.default_timer()
>>> for i in range(10):
...     x_train = X_train[i*100000:(i+1)*100000]
...     y_train = Y_train[i*100000:(i+1)*100000]
...     x_train_enc = enc.transform(x_train)
...     sgd_lr_online.partial_fit(x_train_enc.toarray(), y_train,
...                                classes=[0,
```

Again, we use the `partial_fit` method for online learning. Also, we specify the `classes` parameter, which is required in online learning:

```
>>> print(f"--- {(timeit.default_timer() - start_time)}.3fs seconds ---")
--- 167.399s seconds ---
```

Apply the trained model on the testing set, the next 100,000 samples, as follows:

```
>>> x_test_enc = enc.transform(X_test)
>>> pred = sgd_lr_online.predict_proba(x_test_enc.toarray())[:, 1]
>>> print(f'Training samples: {n_train * 10}, AUC on testing set')
Training samples: 10000000, AUC on testing set: 0.761
```

With online learning, training based on a total of 1 million samples only takes 167 seconds and yields better accuracy.

We have been using logistic regression for binary classification so far. Can we use it for multiclass cases? Yes. However, we do need to make some small tweaks. Let's see this in the next section.

Handling multiclass classification

One last thing worth noting is how logistic regression algorithms deal with multiclass classification. Although we interact with the scikit-learn classifiers in multiclass cases the same way as in binary cases, it is useful to understand how logistic regression works in multiclass classification.

Logistic regression for more than two classes is also called **multinomial logistic regression**, or better known latterly as **softmax regression**. As you have seen in the binary case, the model is represented by one weight vector w , and the probability of the target being 1 or the positive class is written as follows:

$$\hat{y} = P(y = 1 | x) = \frac{1}{1 + \exp(-w^T x)}$$

In the K class case, the model is represented by K weight vectors, w_1, w_2, \dots, w_K , and the probability of the target being class k is written as follows:

$$\hat{y}_k = P(y = k | x) = \frac{\exp(w_k^T x)}{\sum_{j=1}^K \exp(w_j^T x)}$$

$$\sum_{j=1}^K \exp(w_j^T x)$$

Note that the term $\sum_{j=1}^K \exp(w_j^T x)$ normalizes probabilities \hat{y}_k (k from 1 to K) so that they total 1. The cost function in the binary case is expressed as follows:

$$J(w) = \frac{1}{m} \sum_{i=1}^m -[y^{(i)} \log(\hat{y}(x^{(i)})) + (1 - y^{(i)}) \log(1 - \hat{y}(x^{(i)}))] + \alpha ||w||^q$$

Similarly, the cost function in the multiclass case becomes the following:

$$J(w) = \frac{1}{m} \sum_{i=1}^m -[\sum_{j=1}^K 1\{y^{(i)} = j\} \log(\hat{y}_k(x^{(i)}))]$$

Here, function $1\{y^{(i)} = j\}$ is 1 only if $y^{(i)} = j$ is true, otherwise it's 0.

With the cost function defined, we obtain the step Δw_j for the j weight vector in the same way as we derived the step Δw in the binary case:

$$\Delta w_j = \frac{1}{m} \sum_{i=1}^m (-1\{y^{(i)} = j\} + \widehat{y}_k(x^{(i)}))x^{(i)}$$

In a similar manner, all K weight vectors are updated in each iteration. After sufficient iterations, the learned weight vectors, w_1, w_2, \dots, w_K , are then used to classify a new sample x' by means of the following equation:

$$y' = \operatorname{argmax}_k \widehat{y}_k = \operatorname{argmax}_k P(y = k | x')$$

To have a better sense, let's experiment with it with a classic dataset, the handwritten digits for classification:

```
>>> from sklearn import datasets
>>> digits = datasets.load_digits()
>>> n_samples = len(digits.images)
```

As the image data is stored in 8*8 matrices, we need to flatten them, as follows:

```
>>> X = digits.images.reshape((n_samples, -1))
>>> Y = digits.target
```

We then split the data as follows:

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
   test_size=0.2, random_state=
```

We then combine grid search and cross-validation to find the optimal multiclass logistic regression model as follows:

```
>>> from sklearn.model_selection import GridSearchCV
>>> parameters = {'penalty': ['l2', None],
...                 'alpha': [1e-07, 1e-06, 1e-05, 1e-04],
...                 'eta0': [0.01, 0.1, 1, 10]}
>>> sgd_lr = SGDClassifier(loss='log', learning_rate='constant',
...                         eta0=0.01, fit_intercept=True, max_iter=1000,
...                         n_jobs=-1, cv=3)
>>> grid_search = GridSearchCV(sgd_lr, parameters,
...                             n_jobs=-1, cv=3)
>>> grid_search.fit(term_docs_train, label_train)
>>> print(grid_search.best_params_)
{'alpha': 1e-07, 'eta0': 0.1, 'penalty': None}
```

To predict using the optimal model, we apply the following:

```
>>> sgd_lr_best = grid_search.best_estimator_
>>> accuracy = sgd_lr_best.score(term_docs_test, label_test)
>>> print(f'The accuracy on testing set is: {accuracy*100:.1f}%')
The accuracy on testing set is: 94.2%
```

It doesn't look much different from the previous example, since `SGDClassifier` handles multiclass internally. Feel free to compute the confusion matrix as an exercise. It will be interesting to see how the model performs on individual classes.

The next section will be a bonus section where we will implement logistic regression with TensorFlow and use click prediction as an example.

Implementing logistic regression using TensorFlow

We herein use 90% of the first 300,000 samples for training, the remaining 10% for testing, and assume that `x_train_enc`, `Y_train`, `x_test_enc`, and `Y_test` contain the correct data:

1. First, we import TensorFlow, transform `x_train_enc` and `x_test_enc` into a `numpy` array, and cast `x_train_enc`, `Y_train`, `x_test_enc`, and `Y_test` to `float32`:

```
>>> import tensorflow as tf
>>> X_train_enc = X_train_enc.toarray().astype('float32')
>>> X_test_enc = X_test_enc.toarray().astype('float32')
>>> Y_train = Y_train.astype('float32')
>>> Y_test = Y_test.astype('float32')
```

2. We use the `tf.data` API to shuffle and batch data:

```
>>> batch_size = 1000
>>> train_data = tf.data.Dataset.from_tensor_slices((X_train, Y_train))
>>> train_data = train_data.repeat().shuffle(5000).batch(batch_size)
```

For each weight update, only **one batch** of samples is consumed, instead of the one sample or the complete training set. The model moves a step based on the error calculated by a batch of samples. The batch size is 1,000 in this example.

3. Then, we define the weights and bias of the logistic regression model:

```
>>> n_features = int(X_train_enc.shape[1])
>>> W = tf.Variable(tf.zeros([n_features, 1]))
>>> b = tf.Variable(tf.zeros([1]))
```

4. We then create a gradient descent optimizer that searches for the best coefficients by minimizing the loss. We herein use Adam as our optimizer, which is an advanced gradient descent with a learning rate (starting with 0.0008) that is adaptive to gradients:

```
>>> learning_rate = 0.0008
>>> optimizer = tf.optimizers.Adam(learning_rate)
```

5. We define the optimization process where we compute the current prediction and cost and update the model coefficients following the computed gradients:

```
>>> def run_optimization(x, y):
...     with tf.GradientTape() as g:
...         logits = tf.add(tf.matmul(x, W), b)[:, 0]
...         cost =
...             tf.reduce_mean(
```

```
        tf.nn.sigmoid_cross_entropy_with_logits(  
            labels=y, logits=logits))  
...     gradients = g.gradient(cost, [w, b])  
...     optimizer.apply_gradients(zip(gradients, [w, b]))
```

Here, `tf.GradientTape` allows us to track TensorFlow computations and calculate gradients with respect to the given variables.

6. We run the training for 6,000 steps (one step is with one batch of random samples):

```
>>> training_steps = 6000  
>>> for step, (batch_x, batch_y) in  
        enumerate(train_data.take(training_steps), 1)  
...     run_optimization(batch_x, batch_y)  
...     if step % 500 == 0:  
...         logits = tf.add(tf.matmul(batch_x, w), b)[:, 0]  
...         loss =  
...             tf.reduce_mean(  
...                 tf.nn.sigmoid_cross_entropy_with_logits(  
...                     labels=batch_y, logits=logits))  
...         print("step: %i, loss: %f" % (step, loss))  
step: 500, loss: 0.448672  
step: 1000, loss: 0.389186  
step: 1500, loss: 0.413012  
step: 2000, loss: 0.445663  
step: 2500, loss: 0.361000  
step: 3000, loss: 0.417154  
step: 3500, loss: 0.359435  
step: 4000, loss: 0.393363  
step: 4500, loss: 0.402097  
step: 5000, loss: 0.376734  
step: 5500, loss: 0.372981  
step: 6000, loss: 0.406973
```

And for every 500 steps, we compute and print out the current cost to check the training performance. As you can see, the training loss is decreasing overall.

7. After the model is trained, we use it to make predictions on the testing set and report the AUC metric:

```
>>> logits = tf.add(tf.matmul(X_test_enc, w), b)[:, 0]  
>>> pred = tf.nn.sigmoid(logits)  
>>> auc_metric = tf.keras.metrics.AUC()  
>>> auc_metric.update_state(Y_test, pred)
```

```
>>> print(f'AUC on testing set: {auc_metric.result().numpy()}\nAUC on testing set: 0.771
```

We are able to achieve an `AUC` of `0.771` with the TensorFlow-based logistic regression model. You can also tweak the learning rate, the number of training steps, and other hyperparameters to obtain a better performance. This will be a fun exercise at the end of the chapter.

You have seen how feature selection works with L1-regularized logistic regression in the previous section, *Feature selection using L1 regularization*, where weights of unimportant features are compressed to close to, or exactly, 0. Besides L1-regularized logistic regression, random forest is another frequently used feature selection technique. Let's see more in the next section.

Feature selection using random forest

To recap, random forest is bagging over a set of individual decision trees. Each tree considers a random subset of the features when searching for the best splitting point at each node. And, in a decision tree, only those significant features (along with their splitting values) are used to constitute tree nodes. Consider the forest as a whole: the more frequently a feature is used in a tree node, the more important it is. In other words, we can rank the importance of features based on their occurrences in nodes among all trees, and select the top most important ones.

A trained `RandomForestClassifier` module in scikit-learn comes with an attribute, `feature_importances_`, indicating the feature importance, which is calculated as the proportion of occurrences in tree nodes. Again, we will examine feature selection with random forest on the dataset with 100,000 ad click samples:



```
>>> from sklearn.ensemble import RandomForestClassifier  
>>> random_forest = RandomForestClassifier(n_estimators=100,  
                                         criterion='gini', min_samples_split=30, n_jobs=  
>>> random_forest.fit(X_train_enc.toarray(), Y_train)
```

After fitting the random forest model, we obtain the feature importance scores with the following:

```
>>> feature_imp = random_forest.feature_importances_  
>>> print(feature_imp)  
[1.60540750e-05 1.71248082e-03 9.64485853e-04 ... 5.41025913e-04  
 7.78878273e-04 8.24041944e-03]
```

Take a look at the bottom 10 feature scores and the corresponding 10 least important features:

```
>>> feature_names = enc.get_feature_names()  
>>> print(np.sort(feature_imp)[:10])  
[0. 0. 0. 0. 0. 0. 0. 0. 0.]  
>>> bottom_10 = np.argsort(feature_imp)[:10]  
>>> print('10 least important features are:\n', feature_names[bottom_10])  
10 least important features are:  
['x8_ea4912eb' 'x8_c2d34e02' 'x6_2d332391' 'x2_ca9b09d0'  
'x2_0273c5ad' 'x8_92bed2f3' 'x8_eb3f4b48' 'x3_535444a1' 'x8_8741'  
'x8_46cb77e5']
```

And now, take a look at the top 10 feature scores and the corresponding 10 most important features:

```
>>> print(np.sort(feature_imp)[-10:])  
[0.00809279 0.00824042 0.00885188 0.00897925 0.01080301 0.010882  
 0.01270395 0.01392431 0.01532718 0.01810339]  
>>> top_10 = np.argsort(feature_imp)[-10:]  
>>> print('10 most important features are:\n', feature_names[top_10])  
10 most important features are:  
['x17_-1' 'x18_157' 'x12_300' 'x13_250' 'x3_98572c79' 'x8_8a487'
```

In this section, we covered how random forest is used for feature selection.

Summary

In this chapter, we continued working on the online advertising click-through prediction project. This time, we overcame the categorical feature challenge by means of the one-hot encoding technique. We then resorted to a new classification algorithm, logistic regression, for its high scalability to large datasets. The in-depth discussion of the logistic regression algorithm started with the introduction of the logistic function, which led to the mechanics of the algorithm itself. This was followed by how to train a logistic regression model using gradient descent.

After implementing a logistic regression classifier by hand and testing it on our click-through dataset, you learned how to train the logistic regression model in a more advanced manner, using SGD, and we adjusted our algorithm accordingly. We also practiced how to use the SGD-based logistic regression classifier from scikit-learn and applied it to our project.

We then continued to tackle problems we might face in using logistic regression, including L1 and L2 regularization for eliminating overfitting, online learning techniques for training on large-scale datasets, and handling multiclass scenarios. You also learned how to implement logistic regression with TensorFlow. Finally, the chapter ended with applying the random forest model to feature selection, as an alternative to L1-regularized logistic regression.

You might be curious about how we can efficiently train the model on the entire dataset of 40 million samples. In the next chapter, we will utilize tools such as Spark and the `PySpark` module to scale up our solution.

Exercises

1. In the logistic regression-based click-through prediction project, can you also tweak hyperparameters such as `penalty`, `eta0`, and `alpha` in the `SGDClassifier` model? What is the highest testing AUC you are able to achieve?
2. Can you try to use more training samples, for instance, 10 million samples, in the online learning solution?
3. In the TensorFlow-based solution, can you tweak the learning rate, the number of training steps, and other hyperparameters to obtain a better performance?

6

Scaling Up Prediction to Terabyte Click Logs

In the previous chapter, we developed an ad click-through predictor using a logistic regression classifier. We proved that the algorithm is highly scalable by training efficiently on up to 1 million click log samples. In this chapter, we will further boost the scalability of the ad click-through predictor by utilizing a powerful parallel computing (or, more specifically, distributed computing) tool called Apache Spark.

This chapter will demystify how Apache Spark is used to scale up learning on massive data, as opposed to limiting model learning to one single machine. We will also use `Pyspark`, which is the Python API, to explore click log data, to develop classification solutions based on the entire click log dataset, and to evaluate performance, all in a distributed manner. Aside from this, I will introduce two approaches to playing around with categorical features: one is related to hashing in computer science, while the other fuses multiple features. They will be implemented in Spark as well.

In this chapter, we will cover the following topics:

- The main components of Apache Spark
- Spark installation
- Deploying a Spark application
- Fundamental data structures in PySpark
- Core programming in PySpark
- The implementations of ad click-through predictions in PySpark
- Data exploratory analysis in PySpark

- Caching and persistence in Spark
- Feature hashing and its implementations in PySpark
- Feature interaction and its implementations in PySpark

Learning the essentials of Apache Spark

Apache Spark is a distributed cluster computing framework designed for fast and general-purpose computation. It is an open-source technology originally developed by Berkeley's AMPLab at the University of California. It provides an easy-to-use interface for programming interactive queries and stream processing data. What makes it a popular big data analytics tool is its implicit data parallelism, where it automates operations on data in parallel across processors in the computing cluster. Users only need to focus on how they want to manipulate the data, without worrying about how it is distributed among all the computing nodes or which part of the data a node is responsible for.

Bear in mind that this book is mainly about machine learning. Hence, we will only briefly cover the fundamentals of Spark, including its components, installation, deployment, data structure, and core programming.

Breaking down Spark

We will start with the main components of Spark, which are depicted in the following diagram:

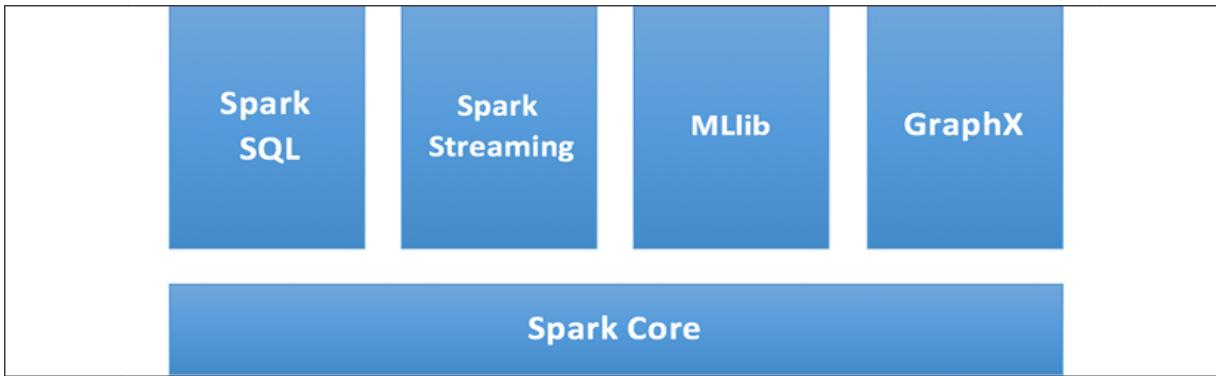


Figure 6.1: The main components of Spark

Let's discuss them in more detail:

- **Spark Core:** This is the foundation and the execution engine of the overall platform. It provides task distribution, scheduling, and in-memory computing. As its name implies, Spark Core is what all the other functionalities are built on top of. It can also be exposed through the APIs of multiple languages, including Python, Java, Scala, and R.
- **Spark SQL:** This is a component built upon Spark Core that introduces a high-level data abstraction called **DataFrames**. We will talk about data structures in Spark later. Spark SQL supports SQL-like data manipulation in Python, Java, and Scala, which works great with structured and semi-structured data. We will be using modules from Spark SQL in this chapter.
- **Spark Streaming:** This performs real-time (or nearly real-time) data analytics by leveraging Spark Core's fast scheduling and in-memory computing capabilities.
- **MLlib:** Short for **machine learning library**, this is a distributed machine learning framework built on top of Spark Core. It allows for learning on large-scale data efficiently thanks to its distributed architecture and in-memory computing capabilities. In in-memory computation, data is kept in the **random access memory (RAM)** if it has sufficient capacity, instead of on disk. This reduces the cost of memory and of reloading data backward and forward during the iterative process. Training a machine learning model is basically an iterative learning process. Hence, the in-memory computing capability

of Spark makes it extremely applicable to machine learning modeling. According to major performance benchmarks, learning using MLlib is nearly 10 times as fast as a disk-based solution. In this chapter, we will be using modules from Spark MLlib.

- **GraphX:** This is another functionality built on top of Spark Core that focuses on distributed graph-based processing. PageRank and Pregel abstraction are two typical use cases.



The main goal of this section is to understand Spark as a distributed cluster computing framework designed for fast computation, which facilitates both data analytics and iterative learning. If you are looking for more detailed information on Spark, there is a lot of useful documentation and tutorials available online, such as <https://spark.apache.org/docs/latest/quick-start.html>.

Installing Spark

For learning purposes, let's now install Spark on the local computer (even though it is more frequently used in a cluster of servers). Full instructions can be found at <https://spark.apache.org/downloads.html>. There are several versions, and we will take version 2.4.5 (Feb 05, 2020) with pre-built for Apache Hadoop 2.7 as an example.



At the time of writing, the latest stable version is 2.4.5. Although there is a preview version, 3.0.0, I think the latest stable version is enough to start with. You won't notice much difference between 3.0 and 2.4.5 going through this chapter. Please note that the module `pyspark.ml.feature.OneHotEncoderEstimator` has been deprecated, and removed in the preview versions (v 3.0.0 and above). Its functionality has been clubbed with `pyspark.ml.feature.OneHotEncoder`.

As illustrated in the following screenshot, after selecting **2.4.5** in *step 1*, we choose the **Pre-built for Apache Hadoop 2.7** option in *step 2*. Then, we click the link in *step 3* to download the **spark-2.4.5-bin-hadoop2.7.tgz** file:

Download Apache Spark™

1. Choose a Spark release: **2.4.5 (Feb 05 2020)**
2. Choose a package type: **Pre-built for Apache Hadoop 2.7**
3. Download Spark: [spark-2.4.5-bin-hadoop2.7.tgz](#)
4. Verify this release using the 2.4.5 [signatures](#), [checksums](#) and [project release KEYS](#).

Figure 6.2: Steps to download Spark

Unzip the downloaded file. The resulting folder contains a complete Spark package; you don't need to do any extra installation.

Before running any Spark program, we need to make sure the following dependencies are installed:

- Java 8+, and that it is included in the system environment variables
- Scala version 2.11

To check whether Spark is installed properly, we run the following tests:

1. First, we approximate the value of π using Spark by typing in the following command in Terminal (note that `bin` is a folder in `spark-2.4.5-bin-hadoop2.7`, so remember to run the following commands inside this folder):

```
./bin/run-example SparkPi 10
```

2. It should print out something similar to the following (the values may differ):

```
Pi is roughly 3.141851141851142
```

This test is actually similar to the following:

```
./bin/spark-submit examples/src/main/python/pi.py 10
```

3. Next, we test the interactive shell with the following command:

```
./bin/pyspark --master local[2]
```

This should open a Python interpreter, as shown in the following screenshot:



Welcome to
Python 2.4.5
Using Python version 3.6.10 (default, Mar 25 2020 18:53:43)
SparkSession available as 'spark'.
>>> |

Figure 6.3: Running Spark in the shell

By now, the Spark program should be installed properly. We will talk about the commands (`pyspark` and `spark-submit`) in the following sections.

Launching and deploying Spark programs

A Spark program can run by itself or over cluster managers. The first option is similar to running a program locally with multiple threads, and one thread is considered one Spark job worker. Of course, there is no parallelism at all, but it is a quick and easy way to launch a Spark application, and we will be deploying it in this mode by way of demonstration throughout this chapter. For example, we can run the following script to launch a Spark application:

```
./bin/spark-submit examples/src/main/python/pi.py
```

This is precisely what we did in the previous section. Or, we can specify the number of threads:

```
./bin/spark-submit --master local[4] examples/src/main/pythc
```

In the previous code, we run Spark locally with four worker threads, or as many cores as there are on the machine, by using the following command:

```
./bin/spark-submit --master local[*] examples/src/main/python/pi
```

Similarly, we can launch the interactive shell by replacing `spark-submit` with `pyspark`:

```
./bin/pyspark --master local[2] examples/src/main/python/pi
```

As for the cluster mode, it (version 2.4.5) currently supports the following approaches:

- **Standalone:** This is the simplest mode to use to launch a Spark application. It means that the master and workers are located on the same machine. Details of how to launch a Spark application in standalone cluster mode can be found at the following link:
<https://spark.apache.org/docs/latest/spark-standalone.html>.
- **Apache Mesos:** As a centralized and fault-tolerant cluster manager, Mesos is designed for managing distributed computing environments. In Spark, when a driver submits tasks for scheduling, Mesos determines which machines handle which tasks. Refer to <https://spark.apache.org/docs/latest/running-on-mesos.html> for further details.
- **Apache Hadoop YARN:** The task scheduler in this approach becomes YARN, as opposed to Mesos in the previous one. **YARN**, which is short for **Yet Another Resource Negotiator**, is the resource manager in Hadoop. With YARN, Spark can be integrated into the Hadoop ecosystem (such as MapReduce, Hive, and File System) more easily. For more information, please go to the following link:
<https://spark.apache.org/docs/latest/running-on-yarn.html>.
- **Kubernetes:** This is an open-source system that provides container-centric infrastructure. It helps automate job deployment and management and has gained in popularity in recent years. Kubernetes

for Spark is still pretty new, but if you are interested, feel free to read more at the following link:

<https://spark.apache.org/docs/latest/running-on-kubernetes.html>.

It's easy to launch and deploy a Spark application. How about coding in PySpark? Let's see in the next section.

Programming in PySpark

This section provides a quick introduction to programming with Python in Spark. We will start with the basic data structures in Spark.

Resilient Distributed Datasets (RDD) is the primary data structure in Spark. It is a distributed collection of objects and has the following three main features:

- **Resilient:** When any node fails, affected partitions will be reassigned to healthy nodes, which makes Spark fault-tolerant
- **Distributed:** Data resides on one or more nodes in a cluster, which can be operated on in parallel
- **Dataset:** This contains a collection of partitioned data with their values or metadata

RDD was the main data structure in Spark before version 2.0. After that, it was replaced by the DataFrame, which is also a distributed collection of data but organized into named columns. DataFrames utilize the optimized execution engine of Spark SQL. Therefore, they are conceptually similar to a table in a relational database or a `DataFrame` object in the Python `pandas` library.



Although the current version of Spark still supports RDD, programming with DataFrames is highly recommended. Hence, we won't spend too much time on programming with RDD. Refer to

<https://spark.apache.org/docs/latest/rdd-programming-guide.html> if you are interested.

The entry point to a Spark program is creating a Spark session, which can be done by using the following lines:

```
>>> from pyspark.sql import SparkSession  
>>> spark = SparkSession \  
...     .builder \  
...     .appName("test") \  
...     .getOrCreate()
```

Note that this is not needed if we run it in a PySpark shell. Right after we spin up a PySpark shell (with `./bin/pyspark`), a Spark session is automatically created. We can check the running Spark application at the following link: `localhost:4040/jobs/`. Refer to the following screenshot for the resulting page:

The screenshot shows the PySparkShell application UI with the title "PySparkShell application UI". At the top, there is a navigation bar with tabs: Jobs, Stages, Storage, Environment, Executors, SQL, and "PySparkShell application UI". The "Jobs" tab is selected. Below the navigation bar, the main content area is titled "Spark Jobs (?)" and displays the following information:
User: hayden
Total Uptime: 2.0 h
Scheduling Mode: FIFO
A blue link labeled "Event Timeline" is visible at the bottom left of the content area.

Figure 6.4: Spark application UI

With a Spark session, `spark`, a DataFrame object can be created by reading a file (which is usually the case) or manual input. In the following example, we will create a DataFrame object, `df`, from a CSV file:

```
>>> df = spark.read.csv("examples/src/main/resources/people.csv"  
                      header=True, sep=';')
```

Columns in the CSV file `people.csv` are separated by `;`.

Once this is done, we will see a completed job in `localhost:4040/jobs/`:

The screenshot shows the 'Spark Jobs' interface. At the top, it displays user information: User: hayden, Total Uptime: 3.5 h, Scheduling Mode: FIFO, and Completed Jobs: 1. Below this is a link to the Event Timeline. The main section is titled 'Completed Jobs (1)' and contains a table with one row. The table has columns: Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. The single row shows Job Id 0, Description 'json at NativeMethodAccessImpl.java:0' (repeated twice), Submitted on 2018/12/02 at 07:42:38, Duration 0.2 s, Stages 1/1, and Tasks 1/1.

Figure 6.5: Completed job list in the Spark application

We can display the contents of the `df` object by using the following command:

```
>>> df.show()
+-----+
| name|age|      job|
+-----+
| Jorge| 30|Developer|
|   Bob| 32|Developer|
+-----+
```

We can count the number of rows by using the following command:

```
>>> df.count()
2
```

The schema of the `df` object can be displayed using the following command:

```
>>> df.printSchema()
root
 |-- name: string (nullable = true)
 |-- age: string (nullable = true)
 |-- job: string (nullable = true)
```

One or more columns can be selected as follows:

```
>>> df.select("name").show()
+---+
| name|
+---+
| Jorge|
| Bob |
+---+
>>> df.select(["name", "job"]).show()
+-----+
| name|      job|
+-----+
| Jorge|Developer|
| Bob |Developer|
+-----+
```

We can filter rows by condition, for instance, by the value of one column, using the following command:

```
>>> df.filter(df['age'] > 31).show()
+-----+
| name|age|      job|
+-----+
| Bob | 32|Developer|
+-----+
```

We will continue programming in PySpark in the next section, where we will use Spark to solve the ad click-through problem.

Learning on massive click logs with Spark

Normally, in order to take advantage of Spark, data is stored using **Hadoop Distributed File System (HDFS)**, which is a distributed file system

designed to store large volumes of data, and computation occurs over multiple nodes on clusters. For demonstration purposes, we will keep the data on a local machine and run Spark locally. This is no different from running it on a distributed computing cluster.

Loading click logs

To train a model on massive click logs, we first need to load the data in Spark. We do so by taking the following steps:

1. We spin up the PySpark shell by using the following command:

```
./bin/pyspark --master local[*] --driver-memory 20G
```

Here, we specify a large driver memory as we are dealing with a dataset of more than 6 GB.



A driver program is responsible for collecting and storing processed results from executors. So, a large driver memory helps complete jobs where lots of data is processed.

2. Next, we start a Spark session with an application named `CTR`:

```
>>> spark = SparkSession\  
...     .builder\  
...     .appName("CTR")\  
...     .getOrCreate()
```

3. Then, we load the click log data from the `train` file into a DataFrame object. Note that the data load function `spark.read.csv` allows custom schemas, which guarantees data is loaded as expected, as opposed to automatically inferring schemas. So, first, we define the schema:

```
>>> from pyspark.sql.types import StructField, StringType,  
        StructType, IntegerType  
>>> schema = StructType([  
...     StructField("id", StringType(), True),  
...     StructField("click", IntegerType(), True),
```

```
...     StructField("hour", IntegerType(), True),
...     StructField("C1", StringType(), True),
...     StructField("banner_pos", StringType(), True),
...     StructField("site_id", StringType(), True),
...     StructField("site_domain", StringType(), True),
...     StructField("site_category", StringType(), True),
...     StructField("app_id", StringType(), True),
...     StructField("app_domain", StringType(), True),
...     StructField("app_category", StringType(), True),
...     StructField("device_id", StringType(), True),
...     StructField("device_ip", StringType(), True),
...     StructField("device_model", StringType(), True),
...     StructField("device_type", StringType(), True),
...     StructField("device_conn_type", StringType(), True)
...     StructField("C14", StringType(), True),
...     StructField("C15", StringType(), True),
...     StructField("C16", StringType(), True),
...     StructField("C17", StringType(), True),
...     StructField("C18", StringType(), True),
...     StructField("C19", StringType(), True),
...     StructField("C20", StringType(), True),
...     StructField("C21", StringType(), True),
...   ])

```

Each field of the schema contains the name of the column (such as `id`, `click`, or `hour`), the data type (such as `integer` or `string`), and whether missing values are allowed (allowed, in this case).

- With the defined schema, we create a DataFrame object, `df`:

```
>>> df = spark.read.csv("file://path_to_file/train", schema=schema)
```



Remember to replace `path_to_file` with the absolute path of where the `train` data file is located.
The `file://` prefix indicates that data is read from a local file. Another prefix, `dbfs://`, is used for data stored in HDFS.

- We now double-check the schema as follows:

```
>>> df.printSchema()
root
 |-- id: string (nullable = true)
```

```
|-- click: integer (nullable = true)
|-- hour: integer (nullable = true)
|-- C1: string (nullable = true)
|-- banner_pos: string (nullable = true)
|-- site_id: string (nullable = true)
|-- site_domain: string (nullable = true)
|-- site_category: string (nullable = true)
|-- app_id: string (nullable = true)
|-- app_domain: string (nullable = true)
|-- app_category: string (nullable = true)
|-- device_id: string (nullable = true)
|-- device_ip: string (nullable = true)
|-- device_model: string (nullable = true)
|-- device_type: string (nullable = true)
|-- device_conn_type: string (nullable = true)
|-- C14: string (nullable = true)
|-- C15: string (nullable = true)
|-- C16: string (nullable = true)
|-- C17: string (nullable = true)
|-- C18: string (nullable = true)
|-- C19: string (nullable = true)
|-- C20: string (nullable = true)
|-- C21: string (nullable = true)
```

6. And the data size is checked as follows:

```
>>> df.count()
40428967
```

7. Also, we need to drop several columns that provide little information. We use the following code to do that:

```
>>> df =
    df.drop('id').drop('hour').drop('device_id').drop('devi
```

8. We rename the column from `click` to `label`, as this will be consumed more often in the downstream operations:

```
>>> df = df.withColumnRenamed("click", "label")
```

9. Let's look at the current columns in the DataFrame object:

```
>>> df.columns
['label', 'C1', 'banner_pos', 'site_id', 'site_domain', 'si
```

After inspecting the input data, we need to split and cache the data.

Splitting and caching the data

Here, we split the data into a training set and testing set, as follows:

```
>>> df_train, df_test = df.randomSplit([0.7, 0.3], 42)
```

In this case, 70% of the samples are used for training and the remaining samples are used for testing, with a random seed specified, as always, for reproduction.

Before we perform any heavy lifting (such as model learning) on the training set, `df_train`, it is good practice to cache the object. In Spark, **caching** and **persistence** are optimization techniques that reduce the computation overhead. This saves the intermediate results of RDD or DataFrame operations in memory and/or on disk. Without caching or persistence, whenever an intermediate DataFrame is needed, it will be recalculated again according to how it was created originally. Depending on the storage level, persistence behaves differently:

- `MEMORY_ONLY` : The object is only stored in memory. If it does not fit in memory, the remaining part will be recomputed each time it is needed.
- `DISK_ONLY` : The object is only kept on disk. A persisted object can be extracted directly from storage without being recalculated.
- `MEMORY_AND_DISK` : The object is stored in memory, and might be on disk as well. If the full object does not fit in memory, the remaining partition will be stored on disk, instead of being recalculated every time it is needed. This is the default mode for caching and persistence in Spark. It takes advantage of both the fast retrieval of in-memory storage and the high accessibility and capacity of disk storage.

In PySpark, caching is simple. All that is required is a `cache` method.

Let's cache both the training and testing DataFrames:

```
>>> df_train.cache()
DataFrame[label: int, C1: string, banner_pos: string, site_id: s
>>> df_train.count()
28297027
>>> df_test.cache()
DataFrame[label: int, C1: string, banner_pos: string, site_id: s
>>> df_test.count()
12131940
```

Now, we have the training and testing data ready for downstream analysis.

One-hot encoding categorical features

Similar to the previous chapter, we need to encode categorical features into sets of multiple binary features by executing the following steps:

1. In our case, the categorical features include the following:

```
>>> categorical = df_train.columns
>>> categorical.remove('label')
>>> print(categorical)
['C1', 'banner_pos', 'site_id', 'site_domain', 'site_catego
```

In PySpark, one-hot encoding is not as direct as it is in scikit-learn (specifically, with the `OneHotEncoder` module).

2. We need to index each categorical column using the `StringIndexer` module:

```
>>> from pyspark.ml.feature import StringIndexer
>>> indexers = [
...     StringIndexer(inputCol=c, outputCol=
...         "{0}_indexed".format(c)).setHandleInvalid("kee
...     for c in categorica
... ]
```

The `setHandleInvalid("keep")` handle makes sure the application won't crash if any new categorical value occurs. Try to omit it; you will see error messages related to unknown values.

3. Then, we perform one-hot encoding on each individual indexed categorical column using the `OneHotEncoderEstimator` module:

```
>>> from pyspark.ml.feature import OneHotEncoderEstimator  
>>> encoder = OneHotEncoderEstimator(  
...     inputCols=[indexer.getOutputCol() for indexer in indexers],  
...     outputCols=["{0}_encoded".format(indexer.getOutputCol())  
                for indexer in indexers],  
... )
```

4. Next, we concatenate all sets of generated binary vectors into a single one using the `VectorAssembler` module:

```
>>> from pyspark.ml.feature import VectorAssembler  
>>> assembler = VectorAssembler(  
...     inputCols=encoder.getOutputCols(),  
...     outputCol="features"  
... )
```

This creates the final encoded vector column called `features`.

5. We chain all these three stages together into a pipeline with the `Pipeline` module in PySpark, which better organizes our one-hot encoding workflow:

```
>>> stages = indexers + [encoder, assembler]  
>>> from pyspark.ml import Pipeline  
>>> pipeline = Pipeline(stages=stages)
```

The variable `stages` is a list of operations needed for encoding.

6. Finally, we can fit the `pipeline` one-hot encoding model over the training set:

```
>>> one_hot_encoder = pipeline.fit(df_train)
```

7. Once this is done, we use the trained encoder to transform both the training and testing sets. For the training set, we use the following code:

```
>>> df_train_encoded = one_hot_encoder.transform(df_train)  
>>> df_train_encoded.show()
```

8. At this point, we skip displaying the results as there are dozens of columns with several additional ones added on top of `df_train`. However, we can see the one we are looking for, the `features` column, which contains the one-hot encoded results. Hence, we only select this column, along with the target variable:

```
>>> df_train_encoded = df_train_encoded.select(  
                    ["label", "features"])  
>>> df_train_encoded.show()  
+-----+  
| label | features |  
+-----+  
| 0 | (31458,[5,7,3527,...]|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,788,4...|  
| 0 | (31458,[5,7,1271,...|  
| 0 | (31458,[5,7,1271,...|  
| 0 | (31458,[5,7,1271,...|  
| 0 | (31458,[5,7,1271,...|  
| 0 | (31458,[5,7,1532,...|  
| 0 | (31458,[5,7,4366,...|  
| 0 | (31458,[5,7,14,45...|  
+-----+  
only showing top 20 rows
```

The feature column contains sparse vectors of size `31458`.

9. Don't forget to cache `df_train_encoded`, as we will be using it to iteratively train our classification model:

```
>>> df_train_encoded.cache()  
DataFrame[label: int, features: vector]
```

10. To release some space, we uncache `df_train`, since we will no longer need it:
-

```
>>> df_train.unpersist()
DataFrame[label: int, C1: string, banner_pos: string, site_
```

11. Now, we repeat the preceding steps for the testing set:

```
>>> df_test_encoded = one_hot_encoder.transform(df_test)
>>> df_test_encoded = df_test_encoded.select(["label", "fea")
>>> df_test_encoded.show()
+-----+
| label | features |
+-----+
| 0|(31458,[5,7,788,4...
| 0|(31458,[5,7,788,4...
| 0|(31458,[5,7,788,4...
| 0|(31458,[5,7,788,4...
| 0|(31458,[5,7,788,4...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,14,45...
| 0|(31458,[5,7,2859,...]
| 0|(31458,[1,7,651,4...
+-----+
only showing top 20 rows
>>> df_test_encoded.cache()
DataFrame[label: int, features: vector]
>>> df_test.unpersist()
DataFrame[label: int, C1: string, banner_pos: string, site_
```

12. If you check the Spark UI `localhost:4040/jobs/` in your browser, you will see several completed jobs, such as the following:

10	countByValue at StringIndexer.scala:140 countByValue at StringIndexer.scala:140	2018/12/03 18:59:12	4 s	2/2	96/96
9	countByValue at StringIndexer.scala:140 countByValue at StringIndexer.scala:140	2018/12/03 18:59:08	4 s	2/2	96/96
8	countByValue at StringIndexer.scala:140 countByValue at StringIndexer.scala:140	2018/12/03 18:59:04	4 s	2/2	96/96
7	countByValue at StringIndexer.scala:140 countByValue at StringIndexer.scala:140	2018/12/03 18:58:59	5 s	2/2	96/96
6	countByValue at StringIndexer.scala:140 countByValue at StringIndexer.scala:140	2018/12/03 18:58:55	4 s	2/2	96/96
5	countByValue at StringIndexer.scala:140 countByValue at StringIndexer.scala:140	2018/12/03 18:58:50	5 s	2/2	96/96
4	countByValue at StringIndexer.scala:140 countByValue at StringIndexer.scala:140	2018/12/03 18:58:46	4 s	2/2	96/96
3	countByValue at StringIndexer.scala:140 countByValue at StringIndexer.scala:140	2018/12/03 18:58:43	3 s	2/2	96/96
2	countByValue at StringIndexer.scala:140 countByValue at StringIndexer.scala:140	2018/12/03 18:58:28	15 s	2/2	96/96

Figure 6.6: List of jobs completed after encoding

With the encoded training and testing sets ready, we can now train our classification model.

Training and testing a logistic regression model

We will use logistic regression as our example, but there are many other classification models supported in PySpark, such as decision tree classifiers, random forests, neural networks (which we will be studying in *Chapter 8, Predicting Stock Prices with Artificial Neural Networks*), linear SVM, and Naïve Bayes. For further details, please refer to the following link:
<https://spark.apache.org/docs/latest/ml-classification-regression.html#classification>.

We can train and test a logistic regression model by using the following steps:

1. We first import the logistic regression module and initialize a model:

```
>>> from pyspark.ml.classification import LogisticRegression
>>> classifier = LogisticRegression(maxIter=20, regParam=0.1,
                                     elasticNetParam=0.001)
```

Here, we set the maximum iterations as 20, and the regularization parameter as 0.001.

2. Now, we fit the model on the encoded training set:

```
>>> lr_model = classifier.fit(df_train_encoded)
```

Be aware that this might take a while. You can check the running or completed jobs in the Spark UI in the meantime. Refer to the following screenshot for some completed jobs:

33	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2018/12/03 19:49:50	20 s	2/2	54/54
32	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2018/12/03 19:49:28	21 s	2/2	54/54
31	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2018/12/03 19:49:07	20 s	2/2	54/54
30	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2018/12/03 19:48:48	19 s	2/2	54/54
29	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2018/12/03 19:48:24	20 s	2/2	54/54
28	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2018/12/03 19:48:01	23 s	2/2	54/54
27	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2018/12/03 19:47:38	23 s	2/2	54/54
26	treeAggregate at RDDLossFunction.scala:61 treeAggregate at RDDLossFunction.scala:61	2018/12/03 19:47:11	26 s	2/2	54/54
25	treeAggregate at LogisticRegression.scala:518 treeAggregate at LogisticRegression.scala:518	2018/12/03 19:28:25	19 min	2/2	54/54

Figure 6.7: List of jobs completed after training

Note that each **RDDLossFunction** represents an iteration of optimizing the logistic regression classifier.

3. After all iterations, we apply the trained model on the testing set:

```
>>> predictions = lr_model.transform(df_test_encoded)
```

4. We cache the prediction results, as we will compute the prediction's performance:

```

>>> predictions.cache()
DataFrame[label: int, features: vector, rawPrediction: vector]
Take a look at the prediction DataFrame:
>>> predictions.show()
+-----+-----+-----+
| label | features | rawPrediction | p |
+-----+-----+-----+
| 0|(31458,[5,7,788,4...|[2.80267740289335...|[0.9428203|
| 0|(31458,[5,7,788,4...|[2.72243908463177...|[0.9383378|
| 0|(31458,[5,7,788,4...|[2.72243908463177...|[0.9383378|
| 0|(31458,[5,7,788,4...|[2.8083664358057...|[0.9437914|
| 0|(31458,[5,7,788,4...|[2.8083664358057...|[0.9437914|
| 0|(31458,[5,7,14,45...|[4.44920221201642...|[0.9884471|
| 0|(31458,[5,7,14,45...|[4.44920221201642...|[0.9884471|
| 0|(31458,[5,7,14,45...|[4.44920221201642...|[0.9884471|
| 0|(31458,[5,7,14,45...|[4.54759977096521...|[0.9895184|
| 0|(31458,[5,7,14,45...|[4.54759977096521...|[0.9895184|
| 0|(31458,[5,7,14,45...|[4.38991492595212...|[0.9877501|
| 0|(31458,[5,7,14,45...|[4.38991492595212...|[0.9877501|
| 0|(31458,[5,7,14,45...|[4.38991492595212...|[0.9877501|
| 0|(31458,[5,7,14,45...|[4.38991492595212...|[0.9877501|
| 0|(31458,[5,7,14,45...|[5.58870435258071...|[0.9962740|
| 0|(31458,[5,7,14,45...|[5.66066729150822...|[0.9965318|
| 0|(31458,[5,7,14,45...|[5.66066729150822...|[0.9965318|
| 0|(31458,[5,7,14,45...|[5.61336061100621...|[0.9963644|
| 0|(31458,[5,7,2859,...|[5.47553763410082...|[0.9958294|
| 0|(31458,[1,7,651,4...|[1.33424801682849...|[0.7915424|
+-----+-----+-----+
only showing top 20 rows

```

This contains the predictive features, the ground truth, the probabilities of the two classes, and the final prediction (with a decision threshold of 0.5).

5. We evaluate the **Area Under Curve (AUC)** of the **Receiver Operating Characteristics (ROC)** on the testing set using the `BinaryClassificationEvaluator` function with the `areaUnderROC` evaluation metric:

```

>>> from pyspark.ml.evaluation import BinaryClassificationEvaluator
>>> ev = BinaryClassificationEvaluator(rawPredictionCol =
                                         "rawPrediction", metricName = "areaUnderROC")
>>> print(ev.evaluate(predictions))
0.7488839207716323

```

We are hereby able to obtain an AUC of 74.89%. Can we do better than this? Let's see in the next section.

Feature engineering on categorical variables with Spark

In this chapter, I have demonstrated how to build an ad click predictor that learns from massive click logs using Spark. Thus far, we have been using one-hot encoding to employ categorical inputs. In this section, we will talk about two popular feature engineering techniques: feature hashing and feature interaction.

Feature hashing is an alternative to one-hot encoding, while feature interaction is a variant of one-hot encoding. **Feature engineering** means generating new features based on domain knowledge or defined rules, in order to improve the learning performance achieved with the existing feature space.

Hashing categorical features

In machine learning, **feature hashing** (also called the **hashing trick**) is an efficient way to encode categorical features. It is based on hashing functions in computer science, which map data of variable sizes to data of a fixed (and usually smaller) size. It is easier to understand feature hashing through an example.

Let's say we have three features: **gender**, **site_domain**, and **device_model**:

gender	site_domain	device_model
male	cnn	samsung

female	abc	iphone
male	nbc	huawei
male	facebook	xiaomi
female	abc	iphone

Table 6.1: Example data of three categorical features

With one-hot encoding, these will become feature vectors of size 9, which comes from 2 (from **gender**) + 4 (from **site_domain**) + 3 (from **device_model**). With feature hashing, we want to obtain a feature vector of size 4. We define a hash function as the sum of Unicode code points for each character, and then divide the result by 4 and take the remainder as the hashed output. Take the first row as an example; we have the following:

$$\begin{aligned} & \text{ord}(m) + \text{ord}(a) + \text{ord}(l) + \text{ord}(e) + \dots + \text{ord}(s) + \text{ord}(u) + \text{ord}(n) + \text{ord}(g) \\ &= \end{aligned}$$

$$109 + 97 + 108 + 101 + \dots + 115 + 117 + 110 + 103 = 1500$$

$1500 \% 4 = 0$, which means we can encode this sample into **[1 0 0 0]**. Similarly, if the remainder is 1, a sample is hashed into **[0, 1, 0, 0]**; **[0, 0, 1, 0]** for a sample with 2 as the remainder; **[0, 0, 0, 1]** for a sample with 3 as the remainder; and so on.

Similarly, for other rows, we have the following:

gender	site_domain	device_model	hash result
male	cnn	samsung	[1 0 0 0]
female	abc	iphone	[0 0 0 1]

male	nbc	huawei	[0 1 0 0]
male	facebook	xiaomi	[1 0 0 0]
female	abc	iphone	[0 0 0 1]

Table 6.2: Hash results of the example data

In the end, we use the four-dimension hashed vectors to represent the original data, instead of the nine-dimension one-hot encoded ones.

There are a few things to note about feature hashing:

- The same input will always be converted into the same output; for instance, the second and fifth rows.
- Two different inputs might be converted into the same output, such as the first and fourth rows. This phenomenon is called **hashing collision**.
- Hence, the choice of the resulting fixed size is important. It will result in serious collision and information loss if the size is too small. If it is too large, it is basically a redundant one-hot encoding. With the correct size, it will make hashing space-efficient and, at the same time, preserve important information, which will further benefit downstream tasks.
- Hashing is one-way, which means we cannot revert the output to its input, while one-hot encoding is a two-way mapping.

Let's now adopt feature hashing for our click prediction project. Recall that the one-hot encoded vectors are of size 31,458. If we choose 10,000 as the fixed hashing size, we will be able to cut the space to less than a third, and reduce the memory consumed by training the classification model. Also, we will see how quick it is to perform feature hashing compared to one-hot encoding, as there is no need to keep track of all unique values across all columns.

It just maps each individual row of string values to a sparse vector through internal hash functions, as follows:

1. We begin by importing the feature hashing module from PySpark and initializing a feature hasher with an output size of 10000:

```
>>> from pyspark.ml.feature import FeatureHasher  
>>> hasher = FeatureHasher(numFeatures=10000,  
                           inputCols=categorical, outputCol="features")
```

2. We use the defined hasher to convert the input DataFrame:

```
>>> hasher.transform(df_train).select("features").show()
+-----+
|      features |
+-----+
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [29, 1228, 1...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [1228, 1289...|
|(10000, [746, 1060, ...|
|(10000, [675, 1228, ...|
|(10000, [1289, 1695...|
+-----+
only showing top 20 rows
```

As you can see, the size of the resulting column, `features`, is 10000. Again, there is no training or fitting in feature hashing. The hasher is a predefined mapping.

3. For better organization of the entire workflow, we chain the hasher and classification model together into a pipeline:

```
>>> stages = [hasher, classifier]
>>> pipeline = Pipeline(stages=stages)
```

4. We fit the pipelined model on the training set as follows:

```
>>> model = pipeline.fit(df_train)
```

5. We apply the trained model on the testing set and record the prediction results:

```
>>> predictions = model.transform(df_test)
>>> predictions.cache()
```

6. We evaluate its performance in terms of the AUC of ROC:

```
>>> ev = BinaryClassificationEvaluator(rawPredictionCol =
                                         "rawPrediction", metricName = "areaUnderROC")
>>> print(ev.evaluate(predictions))
0.7448097180769776
```

We are able to achieve an AUC of 74.48%, which is close to the previous one of 74.89% with one-hot encoding. At the end of the day, we save a substantial amount of computational resources and attain a comparable prediction accuracy. That is a win.



With feature hashing, we lose interpretability but gain a computational advantage.

Combining multiple variables – feature interaction

Among all the features of the click log data, some are very weak signals in themselves. For example, gender itself doesn't tell us much regarding whether someone will click an ad, and the device model itself doesn't provide much information either.

However, by combining multiple features, we will be able to create a stronger synthesized signal. **Feature interaction** (also known as **feature crossing**) will be introduced for this purpose. For numerical features, it usually generates new features by multiplying multiples of them.

We can also define whatever integration rules we want. For example, we can generate an additional feature, **income/person**, from two original features, **household income** and **household size**:

household income	household size	income/person
300,000	2	150,000
100,000	1	100,000
400,000	4	100,000
300,000	5	60,000
200,000	2	100,000

Table 6.3: An example of generating a new numerical feature based on existing ones

For categorical features, feature interaction becomes an **AND** operation on two or more features. In the following example, we are generating an additional feature, **gender:site_domain**, from two original features, **gender** and **site_domain**:

gender	site_domain	gender:site_domain
male	cnn	male:cnn
female	abc	female:abc

male	nbc	male:nbc
male	facebook	male:facebook
female	abc	female:abc

Table 6.4: An example of generating a new categorical feature based on existing ones

We then use one-hot encoding to transform string values. On top of six one-hot encoded features (two from **gender** and four from **site_domain**), feature interaction between **gender** and **site_domain** adds eight further features (two by four).

Let's now adopt feature interaction for our click prediction project. We will take two features, `c14` and `c15`, as an example of an **AND** interaction:

1. First, we import the feature interaction module, `RFormula`, from PySpark:

```
>>> from pyspark.ml.feature import RFormula
```

An `RFormula` model takes in a formula that describes how features interact. For instance, $y \sim a + b$ means it takes in features a and b , and predicts y based on them; $y \sim a + b + a:b$ means it predicts y based on features a , b , and the iteration term, a **AND** b ; $y \sim a + b + c + a:b$ means it predicts y based on features a , b , c , and the iteration term, a **AND** b .

2. We need to define an interaction formula accordingly:

```
>>> cat_inter = ['C14', 'C15']
>>> cat_no_inter = [c for c in categorical if c not in cat_inter]
>>> concat = '+'.join(cat_no_inter)
>>> interaction = ':'.join(cat_inter)
>>> formula = "label ~ " + concat + '+' + interaction
>>> print(formula)
label ~ C1+banner_pos+site_id+site_domain+site_category+app.
```

3. Now, we can initialize a feature interactor with this formula:

```
>>> interactor = RFormula(  
...     formula=formula,  
...     featuresCol="features",  
...     labelCol="label").setHandleInvalid("keep")
```

Again, the `setHandleInvalid("keep")` handle here makes sure it won't crash if any new categorical value occurs.

4. We use the defined feature interactor to fit and transform the input DataFrame:

```
>>> interactor.fit(df_train).transform(df_train).select("fe  
  
+-----+  
|      features|  
+-----+  
|(54930, [5, 7, 3527, ...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 788, 4...|  
|(54930, [5, 7, 1271, ...|  
|(54930, [5, 7, 1271, ...|  
|(54930, [5, 7, 1271, ...|  
|(54930, [5, 7, 1532, ...|  
|(54930, [5, 7, 4366, ...|  
|(54930, [5, 7, 14, 45...|  
+-----+  
only showing top 20 rows
```

More than 20,000 features are added to the feature space due to the interaction term of `c14` and `c15`.

5. Again, we chain the feature interactor and classification model together into a pipeline to organize the entire workflow:

```
>>> classifier = LogisticRegression(maxIter=20, regParam=0.1  
                                     elasticNetParam=0.000)  
>>> stages = [interactor, classifier]  
>>> pipeline = Pipeline(stages=stages)  
>>> model = pipeline.fit(df_train)  
>>> predictions = model.transform(df_test)  
>>> predictions.cache()  
>>> from pyspark.ml.evaluation import BinaryClassificationEvaluator  
>>> ev = BinaryClassificationEvaluator(rawPredictionCol =  
                                         "rawPrediction", metricName = "areaUnderROC")  
>>> print(ev.evaluate(predictions))  
0.7490392990518315
```

An AUC of 74.90%, with additional interaction between features `c14` and `c15`, is a boost from 74.89% without any interaction. Therefore, feature interaction slightly boosts the model's performance.

Summary

In this chapter, we continued working on the online advertising click-through prediction project. This time, we were able to train the classifier on the entire dataset with millions of records, with the help of the parallel computing tool Apache Spark. We discussed the basics of Spark, including its major components, the deployment of Spark programs, the programming essentials of PySpark, and the Python interface of Spark. Then, we programmed using PySpark to explore the click log data.

You learned how to perform one-hot encoding, cache intermediate results, develop classification solutions based on the entire click log dataset, and evaluate performance. In addition, I introduced two feature engineering techniques, feature hashing and feature interaction, in order to improve prediction performance. We had fun implementing them in PySpark as well.

Looking back on our learning journey, we have been working on classification problems since *Chapter 2, Building a Movie Recommendation Engine with Naïve Bayes*. Actually, we have covered all the powerful and

popular classification models in machine learning. We will move on to solving regression problems in the next chapter; regression is the sibling of classification in supervised learning. You will learn about regression models, including linear regression, decision trees for regression, and support vector regression.

Exercises

1. In the one-hot encoding solution, can you use different classifiers supported in PySpark instead of logistic regression, such as decision trees, random forests, or linear SVM?
2. In the feature hashing solution, can you try other hash sizes, such as 5,000 or 20,000? What do you observe?
3. In the feature interaction solution, can you try other interactions, such as `c1` and `c20`?
4. Can you first use feature interaction and then feature hashing in order to lower the expanded dimension? Are you able to obtain a higher AUC?

Predicting Stock Prices with Regression Algorithms

In the previous chapter, we trained a classifier on a large click dataset using Spark. In this chapter, we will be solving a problem that interests everyone—predicting stock prices. Getting wealthy by means of smart investment—who isn't interested?! Stock market movements and stock price predictions have been actively researched by a large number of financial, trading, and even technology corporations. A variety of methods have been developed to predict stock prices using machine learning techniques. Herein, we will be focusing on learning several popular regression algorithms, including linear regression, regression trees and regression forests, and support vector regression, and utilizing them to tackle this billion (or trillion) dollar problem.

We will cover the following topics in this chapter:

- Introducing the stock market and stock prices
- What is regression?
- Stock data acquisition and feature engineering
- The mechanics of linear regression
- Implementing linear regression (from scratch, and using scikit-learn and TensorFlow)
- The mechanics of regression trees
- Implementing regression trees (from scratch and using scikit-learn)
- From regression tree to regression forest

- The mechanics of support vector regression and implementing it with scikit-learn
- Regression performance evaluation
- Predicting stock prices with regression algorithms

A brief overview of the stock market and stock prices

The stock of a corporation signifies ownership in the corporation. A single share of the stock represents a claim on the fractional assets and the earnings of the corporation in proportion to the total number of shares. For example, if an investor owns 50 shares of stock in a company that has, in total, 1,000 outstanding shares, that investor (or shareholder) would own and have a claim on 5% of the company's assets and earnings.

Stocks of a company can be traded between shareholders and other parties via stock exchanges and organizations. Major stock exchanges include New York Stock Exchange, NASDAQ, London Stock Exchange Group, Shanghai Stock Exchange, and Hong Kong Stock Exchange. The prices that a stock is traded at fluctuate essentially due to the law of supply and demand. At any one moment, the supply is the number of shares that are in the hands of public investors, the demand is the number of shares investors want to buy, and the price of the stock moves up and down in order to attain and maintain equilibrium.

In general, investors want to buy low and sell high. This sounds simple enough, but it's very challenging to implement as it's monumentally difficult to say whether a stock price will go up or down. There are two main streams of studies that attempt to understand factors and conditions that lead to price changes or even to forecast future stock prices, **fundamental analysis** and **technical analysis**:

- **Fundamental analysis:** This stream focuses on underlying factors that influence a company's value and business, including overall economy and industry conditions from macro perspectives, the company's financial conditions, management, and competitors from micro perspectives.
- **Technical analysis:** On the other hand, this stream predicts future price movements through the statistical study of past trading activity, including price movement, volume, and market data. Predicting prices via machine learning techniques is an important topic in technical analysis nowadays.

Many quantitative, or quant, trading firms have been using machine learning to empower automated and algorithmic trading. In this chapter, we'll be working as a quantitative analyst/researcher, exploring how to predict stock prices with several typical **machine learning regression** algorithms.

What is regression?

Regression is one of the main types of supervised learning in machine learning. In regression, the training set contains observations (also called features) and their associated **continuous** target values. The process of regression has two phases:

- The first phase is exploring the relationships between the observations and the targets. This is the training phase.
- The second phase is using the patterns from the first phase to generate the target for a future observation. This is the prediction phase.

The overall process is depicted in the following diagram:

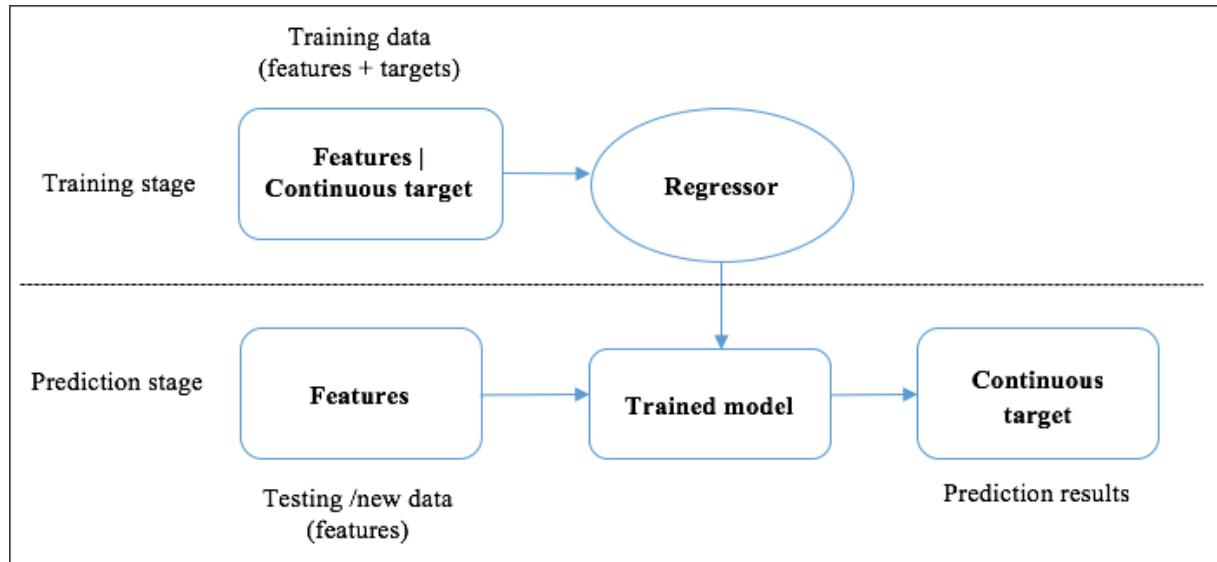


Figure 7.1: Training and prediction phase in regression

The major difference between regression and classification is that the output values in regression are continuous, while in classification they are discrete. This leads to different application areas for these two supervised learning methods. Classification is basically used to determine desired memberships or characteristics, as you've seen in previous chapters, such as email being spam or not, newsgroup topics, or ad click-through. On the other hand, regression mainly involves estimating an outcome or forecasting a response.

An example of estimating continuous targets with linear regression is depicted as follows, where we try to fit a line against a set of two-dimensional data points:

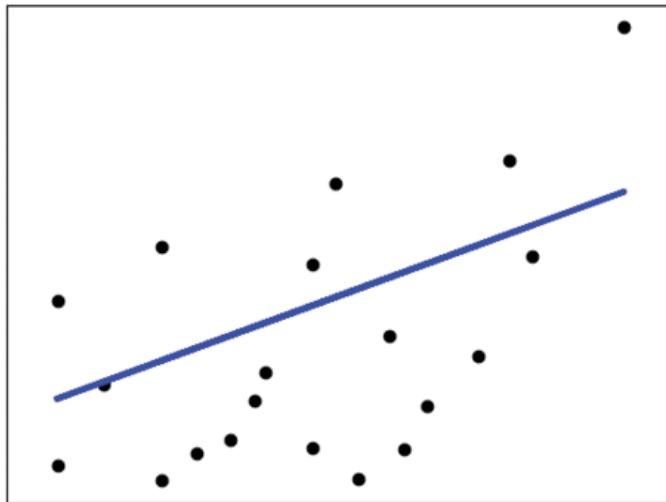


Figure 7.2: Linear regression example

Typical machine learning regression problems include the following:

- Predicting house prices based on location, square footage, number of bedrooms, and bathrooms
- Estimating power consumption based on information about a system's processes and memory
- Forecasting demand in retail
- Predicting stock prices

I've talked about regression in this section and will briefly introduce its use in the stock market and trading in the next one.

Mining stock price data

In theory, we can apply regression techniques to predicting prices of a particular stock. However, it's difficult to ensure the stock we pick is suitable for learning purposes—its price should follow some learnable patterns and it can't have been affected by unprecedented instances or irregular events. Hence, we'll herein be focusing on one of the most

popular **stock indexes** to better illustrate and generalize our price regression approach.

Let's first cover what an index is. A **stock index** is a statistical measure of the value of a portion of the overall stock market. An index includes several stocks that are diverse enough to represent a section of the whole market. And the price of an index is typically computed as the weighted average of the prices of selected stocks.

The **Dow Jones Industrial Average (DJIA)** is one of the longest established and most commonly watched indexes in the world. It consists of 30 of the most significant stocks in the U.S., such as Microsoft, Apple, General Electric, and the Walt Disney Company, and represents around a quarter of the value of the entire U.S. market. You can view its daily prices and performance on Yahoo Finance at <https://finance.yahoo.com/quote/%5EDJI/history?p=%5EDJI>:

The screenshot shows a table of historical price data for the DJIA. The columns include Date, Open, High, Low, Close*, Adj Close**, and Volume. The data spans from April 06, 2020, back to Mar 20, 2020. The table is sorted by date in descending order. The 'Close*' column shows a significant drop starting in March 2020, reflecting the impact of the COVID-19 pandemic.

Date	Open	High	Low	Close*	Adj Close**	Volume
Apr 06, 2020	21,693.63	22,176.79	21,693.63	22,139.61	22,139.61	222,041,710
Apr 03, 2020	21,285.93	21,447.81	20,863.09	21,052.53	21,052.53	450,010,000
Apr 02, 2020	20,819.46	21,477.77	20,735.02	21,413.44	21,413.44	529,540,000
Apr 01, 2020	21,227.38	21,487.24	20,784.43	20,943.51	20,943.51	506,680,000
Mar 31, 2020	22,208.42	22,480.37	21,852.08	21,917.16	21,917.16	571,210,000
Mar 30, 2020	21,678.22	22,378.09	21,522.08	22,327.48	22,327.48	545,540,000
Mar 27, 2020	21,898.47	22,327.57	21,469.27	21,636.78	21,636.78	588,830,000
Mar 26, 2020	21,468.38	22,595.06	21,427.10	22,552.17	22,552.17	705,180,000
Mar 25, 2020	21,050.34	22,019.93	20,538.34	21,200.55	21,200.55	796,320,000
Mar 24, 2020	19,722.19	20,737.70	19,649.25	20,704.91	20,704.91	799,340,000
Mar 23, 2020	19,028.36	19,121.01	18,213.65	18,591.93	18,591.93	787,970,000
Mar 20, 2020	20,253.15	20,531.26	19,094.27	19,173.98	19,173.98	872,290,000

Figure 7.3: Screenshot of daily prices and performance in Yahoo Finance

On each trading day, the price of a stock changes and is recorded in real time. Five values illustrating movements in the price over one unit of time (usually one day, but it can also be one week or one month) are key trading indicators. They are as follows:

- **Open:** The starting price for a given trading day
- **Close:** The final price on that day
- **High:** The highest prices at which the stock traded on that day
- **Low:** The lowest prices at which the stock traded on that day
- **Volume:** The total number of shares traded before the market closed on that day

Other major indexes besides DJIA include the following:

- The S&P 500 (short for Standard & Poor's 500) index is made up of 500 of the most commonly traded stocks in the U.S., representing 80% of the value of the entire U.S. market
(<https://finance.yahoo.com/quote/%5EGSPC/history?p=%5EGSPC>)
- NASDAQ Composite is composed of all stocks traded on NASDAQ
(<https://finance.yahoo.com/quote/%5EIXIC/history?p=%5EIXIC>)
- The Russell 2000 (RUT) index is a collection of the last 2,000 out of 3,000 largest publicly traded companies in the U.S.
(<https://finance.yahoo.com/quote/%5ERUT/history?p=%5ERUT>)
- London FTSE-100 is composed of the top 100 companies in market capitalization listed on the London Stock Exchange
(<https://finance.yahoo.com/quote/%5EFTSE/>)

We will be focusing on DJIA and using its historical prices and performance to predict future prices. In the following sections, we will be exploring how to develop price prediction models, specifically regression models, and what can be used as indicators or predictive features.

Getting started with feature engineering

When it comes to a machine learning algorithm, the first question to ask is usually what features are available or what the predictive variables are.

The driving factors that are used to predict future prices of DJIA, the **close** prices, include historical and current **open** prices as well as historical performance (**high**, **low**, and **volume**). Note that current or same-day performance (**high**, **low**, and **volume**) shouldn't be included because we simply can't foresee the highest and lowest prices at which the stock traded or the total number of shares traded before the market closed on that day.

Predicting the close price with only those preceding four indicators doesn't seem promising and might lead to underfitting. So, we need to think of ways to generate more features in order to increase predictive power. To recap, in machine learning, **feature engineering** is the process of creating domain-specific features based on existing features in order to improve the performance of a machine learning algorithm.

Feature engineering usually requires sufficient domain knowledge and can be very difficult and time-consuming. In reality, features used to solve a machine learning problem are not usually directly available and need to be specifically designed and constructed, for example, term frequency or tf-idf features in spam email detection and newsgroup classification. Hence, feature engineering is essential in machine learning and is usually where we spend the most effort in solving a practical problem.

When making an investment decision, investors usually look at historical prices over a period of time, not just the price the day before. Therefore, in our stock price prediction case, we can compute the average close price over the past week (five trading days), over the past month, and over the past year as three new features. We can also customize the time window to the size we want, such as the past quarter or the past six months. On top of these three averaged price features, we can generate new features associated with the price trend by computing the ratios between each pair of average prices in the three different time frames, for instance, the ratio between the average price over the past week and over the past year.

Besides prices, volume is another important factor that investors analyze. Similarly, we can generate new volume-based features by computing the average volumes in several different time frames and the ratios between each pair of averaged values.

Besides historical averaged values in a time window, investors also greatly consider stock volatility. Volatility describes the degree of variation of prices for a given stock or index over time. In statistical terms, it's basically the standard deviation of the close prices. We can easily generate new sets of features by computing the standard deviation of close prices in a particular time frame, as well as the standard deviation of volumes traded. In a similar manner, ratios between each pair of standard deviation values can be included in our engineered feature pool.

Last but not least, return is a significant financial metric that investors closely watch for. Return is the percentage of gain or loss of close price for a stock/index in a particular period. For example, daily return and annual return are financial terms we frequently hear. They are calculated as follows:

$$\begin{aligned}return_{i:i-1} &= \frac{price_i - price_{i-1}}{price_{i-1}} \\return_{i:i-365} &= \frac{price_i - price_{i-365}}{price_{i-365}}\end{aligned}$$

Here, $price_i$ is the price on the i^{th} day and $price_{i-1}$ is the price on the day before. Weekly and monthly returns can be computed in a similar way. Based on daily returns, we can produce a moving average over a particular number of days.

For instance, given daily returns of the past week, $return_{i:i-1}$, $return_{i-1:i-2}$, $return_{i-2:i-3}$, $return_{i-3:i-4}$, and $return_{i-4:i-5}$, we can calculate the moving average over that week as follows:

$$MovingAvg_{i-5} = \frac{return_{i:i-1} + return_{i-1:i-2} + return_{i-2:i-3} + return_{i-3:i-4} + return_{i-4:i-5}}{5}$$

In summary, we can generate the following predictive variables by applying feature engineering techniques:

AvgPrice_5	The average close price over the past five days
AvgPrice_{30}	The average close price over the past month
AvgPrice_{365}	The average close price over the past year
$\frac{\text{AvgPrice}_5}{\text{AvgPrice}_{30}}$	The ratio between the average price over the past week and that over the past month
$\frac{\text{AvgPrice}_5}{\text{AvgPrice}_{365}}$	The ratio between the average price over the past week and that over the past year
$\frac{\text{AvgPrice}_{30}}{\text{AvgPrice}_{365}}$	The ratio between the average price over the past month and that over the past year
AvgVolume_5	The average volume over the past five days
AvgVolume_{30}	The average volume over the past month
AvgVolume_{365}	The average volume over the past year
$\frac{\text{AvgVolume}_5}{\text{AvgVolume}_{30}}$	The ratio between the average volume over the past week and that over the past month
$\frac{\text{AvgVolume}_5}{\text{AvgVolume}_{365}}$	The ratio between the average volume over the past week and that over the past year
$\frac{\text{AvgVolume}_{30}}{\text{AvgVolume}_{365}}$	The ratio between the average volume over the past month and that over the past year
StdPrice_5	The standard deviation of the close prices over the past five days
StdPrice_{30}	The standard deviation of the close prices over the past month
StdPrice_{365}	The standard deviation of the close prices over the past year

Figure 7.4: Generated features (1)

$\frac{\text{StdPrice}_5}{\text{StdPrice}_{30}}$	The ratio between the standard deviation of the prices over the past week and that over the past month
$\frac{\text{StdPrice}_5}{\text{StdPrice}_{365}}$	The ratio between the standard deviation of the prices over the past week and that over the past year
$\frac{\text{StdPrice}_{30}}{\text{StdPrice}_{365}}$	The ratio between the standard deviation of the prices over the past month and that over the past year
StdVolume_5	The standard deviation of the volumes over the past five days
StdVolume_{30}	The standard deviation of the volumes over the past month
StdVolume_{365}	The standard deviation of the volumes over the past year
$\frac{\text{StdVolume}_5}{\text{StdVolume}_{30}}$	The ratio between the standard deviation of the volumes over the past week and that over the past month
$\frac{\text{StdVolume}_5}{\text{StdVolume}_{365}}$	The ratio between the standard deviation of the volumes over the past week and that over the past year
$\frac{\text{StdVolume}_{30}}{\text{StdVolume}_{365}}$	The ratio between the standard deviation of the volumes over the past month and that over the past year
$\text{return}_{i:i-1}$	Daily return of the past day
$\text{return}_{i:i-5}$	Weekly return of the past week
$\text{return}_{i:i-30}$	Monthly return of the past month
$\text{return}_{i:i-365}$	Yearly return of the past year
MovingAvg_{i_5}	Moving average of the daily returns over the past week
MovingAvg_{i_30}	Moving average of the daily returns over the past month
MovingAvg_{i_365}	Moving average of the daily returns over the past year

Figure 7.5: Generated features (2)

Eventually, we are able to generate in total 31 sets of features, along with the following six original features:

- OpenPrice_i : This feature represents the open price
- OpenPrice_{i-1} : This feature represents the open price on the past day
- ClosePrice_{i-1} : This feature represents the close price on the past day

- HighPrice_{i-1} : This feature represents the highest price on the past day
- LowPrice_{i-1} : This feature represents the lowest price on the past day
- Volume_{i-1} : This feature represents the volume on the past day

Acquiring data and generating features

For easier reference, we will implement the code for generating features here rather than in later sections. We will start by obtaining the dataset we need for our project.

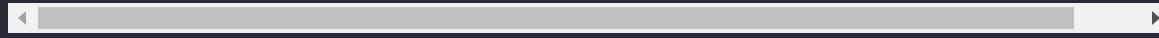
Throughout the project, we will acquire stock index price and performance data from Yahoo Finance. For example, on the Historical Data page, <https://finance.yahoo.com/quote/%5EDJI/history?p=%5EDJI>, we can change the `Time Period` to `Dec 01, 2005 – Dec10, 2005`, select `Historical Prices` in `Show`, and `Daily` in `Frequency` (or open this link directly:

<https://finance.yahoo.com/quote/%5EDJI/history?period1=1133395200&period2=1134172800&interval=1d&filter=history&frequency=1d&includeAdjustedClose=true>), then click on the **Apply** button. Click the **Download data** button to download the data and name the file `20051201_20051210.csv`.

We can load the data we just downloaded as follows:

```
>>> mydata = pd.read_csv('20051201_20051210.csv', index_col='Date')
>>> mydata
          Open        High        Low      Close
Date
2005-12-01  10806.030273  10934.900391  10806.030273  10912.570312
2005-12-02  10912.009766  10921.370117  10861.660156  10877.509766
2005-12-05  10876.950195  10876.950195  10810.669922  10835.009766
2005-12-06  10835.410156  10936.200195  10835.410156  10856.860352
2005-12-07  10856.860352  10868.059570  10764.009766  10810.910156
2005-12-08  10808.429688  10847.250000  10729.669922  10755.120117
2005-12-09  10751.759766  10805.950195  10729.910156  10778.580078
          Volume  Adjusted Close
Date
2005-12-01  1053000000.0  10912.570312
2005-12-02  1053000000.0  10877.509766
2005-12-05  1053000000.0  10835.009766
2005-12-06  1053000000.0  10856.860352
2005-12-07  1053000000.0  10810.910156
2005-12-08  1053000000.0  10755.120117
2005-12-09  1053000000.0  10778.580078
```

```
2005-12-01 256980000.0 10912.570312  
2005-12-02 214900000.0 10877.509766  
2005-12-05 237340000.0 10835.009766  
2005-12-06 264630000.0 10856.860352  
2005-12-07 243490000.0 10810.910156  
2005-12-08 253290000.0 10755.120117  
2005-12-09 238930000.0 10778.580078
```



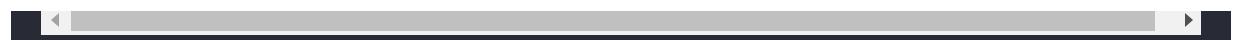
Note the output is a `pandas DataFrame` object. The `Date` column is the index column, and the rest of the columns are the corresponding financial variables. In the following lines of code, you will see how powerful pandas is at simplifying data analysis and transformation on **relational** (or table-like) data.

First, we implement feature generation by starting with a sub-function that directly creates features from the original six features, as follows:

```
>>> def add_original_feature(df, df_new):  
...     df_new['open'] = df['Open']  
...     df_new['open_1'] = df['Open'].shift(1)  
...     df_new['close_1'] = df['Close'].shift(1)  
...     df_new['high_1'] = df['High'].shift(1)  
...     df_new['low_1'] = df['Low'].shift(1)  
...     df_new['volume_1'] = df['Volume'].shift(1)
```

Then we develop a sub-function that generates six features related to average close prices:

```
>>> def add_avg_price(df, df_new):  
...     df_new['avg_price_5'] =  
...         df['Close'].rolling(5).mean().shift(1)  
...     df_new['avg_price_30'] =  
...         df['Close'].rolling(21).mean().shift(1)  
...     df_new['avg_price_365'] =  
...         df['Close'].rolling(252).mean().shift(1)  
...     df_new['ratio_avg_price_5_30'] =  
...         df_new['avg_price_5'] / df_new['avg_price_30']  
...     df_new['ratio_avg_price_5_365'] =  
...         df_new['avg_price_5'] / df_new['avg_price_365']  
...     df_new['ratio_avg_price_30_365'] =  
...         df_new['avg_price_30'] / df_new['avg_price_365']
```



Similarly, a sub-function that generates six features related to average volumes is as follows:

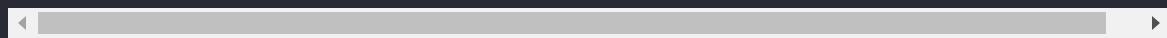
```
>>> def add_avg_volume(df, df_new):
...     df_new['avg_volume_5'] =
...         df['Volume'].rolling(5).mean().shift(1)
...     df_new['avg_volume_30'] =
...         df['Volume'].rolling(21).mean().shift(1)
...     df_new['avg_volume_365'] =
...         df['Volume'].rolling(252).mean().shift(1)
...     df_new['ratio_avg_volume_5_30'] =
...         df_new['avg_volume_5'] / df_new['avg_volume_30']
...     df_new['ratio_avg_volume_5_365'] =
...         df_new['avg_volume_5'] / df_new['avg_volume_365']
...     df_new['ratio_avg_volume_30_365'] =
...         df_new['avg_volume_30'] / df_new['avg_volume_365']
```

As for the standard deviation, we develop the following sub-function for the price-related features:

```
>>> def add_std_price(df, df_new):
...     df_new['std_price_5'] =
...         df['Close'].rolling(5).std().shift(1)
...     df_new['std_price_30'] =
...         df['Close'].rolling(21).std().shift(1)
...     df_new['std_price_365'] =
...         df['Close'].rolling(252).std().shift(1)
...     df_new['ratio_std_price_5_30'] =
...         df_new['std_price_5'] / df_new['std_price_30']
...     df_new['ratio_std_price_5_365'] =
...         df_new['std_price_5'] / df_new['std_price_365']
...     df_new['ratio_std_price_30_365'] =
...         df_new['std_price_30'] / df_new['std_price_365']
```

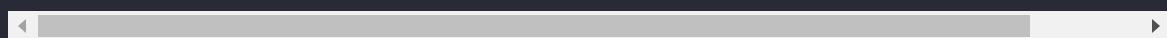
Similarly, a sub-function that generates six volume-based standard deviation features is as follows:

```
>>> def add_std_volume(df, df_new):
...     df_new['std_volume_5'] =
...         df['Volume'].rolling(5).std().shift(1)
...     df_new['std_volume_30'] =
...         df['Volume'].rolling(21).std().shift(1)
...     df_new['std_volume_365'] =
...         df['Volume'].rolling(252).std().shift(1)
...     df_new['ratio_std_volume_5_30'] =
...         df_new['std_volume_5'] / df_new['std_volume_30']
...     df_new['ratio_std_volume_5_365'] =
...         df_new['std_volume_5'] / df_new['std_volume_365']
...     df_new['ratio_std_volume_30_365'] =
...         df_new['std_volume_30'] / df_new['std_volume_365']
```



Seven return-based features are generated using the following sub-function:

```
>>> def add_return_feature(df, df_new):
...     df_new['return_1'] = ((df['Close'] - df['Close'].shift(1))
...                           / df['Close'].shift(1)).shift(1)
...     df_new['return_5'] = ((df['Close'] - df['Close'].shift(5))
...                           / df['Close'].shift(5)).shift(1)
...     df_new['return_30'] = ((df['Close'] -
...                            df['Close'].shift(21)) / df['Close'].shift(21)).shift(1)
...     df_new['return_365'] = ((df['Close'] -
...                             df['Close'].shift(252)) / df['Close'].shift(252)).shift(1)
...     df_new['moving_avg_5'] =
...         df_new['return_1'].rolling(5).mean().shift(1)
...     df_new['moving_avg_30'] =
...         df_new['return_1'].rolling(21).mean().shift(1)
...     df_new['moving_avg_365'] =
...         df_new['return_1'].rolling(252).mean().shift(1)
```

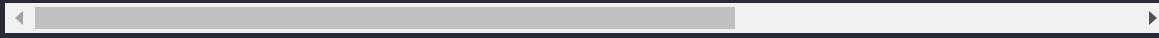


Finally, we put together the main feature generation function that calls all the preceding sub-functions:

```
>>> def generate_features(df):
...     """
...     Generate features for a stock/index based on historical
...     @param df: dataframe with columns "Open", "Close", "High",
...     @return: dataframe, data set with new features
...     """
...     df = add_std_volume(df)
...     df = add_return_feature(df)
```

df = generate_features(df)

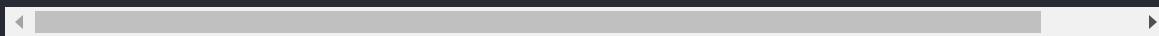
```
...     df_new = pd.DataFrame()
...     # 6 original features
...     add_original_feature(df, df_new)
...     # 31 generated features
...     add_avg_price(df, df_new)
...     add_avg_volume(df, df_new)
...     add_std_price(df, df_new)
...     add_std_volume(df, df_new)
...     add_return_feature(df, df_new)
...     # the target
...     df_new['close'] = df['Close']
...     df_new = df_new.dropna(axis=0)
...     return df_new
```



Note that the window sizes here are 5, 21, and 252, instead of 7, 30, and 365 representing the weekly, monthly, and yearly window. This is because there are 252 (rounded) trading days in a year, 21 trading days in a month, and 5 in a week.

We can apply this feature engineering strategy on the DJIA data queried from 1988 to 2019 as follows (or directly download it from this page: <https://finance.yahoo.com/quote/%5EDJI/history?period1=567993600&period2=1577750400&interval=1d&filter=history&frequency=1d>):

```
>>> data_raw = pd.read_csv('19880101_20191231.csv', index_col='Date')
>>> data = generate_features(data_raw)
```



Take a look at what the data with the new features looks like:

```
>>> print(data.round(decimals=3).head(5))
```

The preceding command line generates the following output:

	open	open_1	close_1	high_1	low_1	volume_1	avg_price_5	...	return_5
Date	return_30	return_365	moving_avg_5	moving_avg_30	moving_avg_365	close		...	
1989-01-04	2153.75	2163.21	2144.64	2168.39	2127.14	17310000.0	2165.000	...	-0.011
	0.020	0.056	0.001	0.001	0.000	2177.68			
1989-01-05	2184.29	2153.75	2177.68	2183.39	2146.61	15710000.0	2168.000	...	0.007
	0.041	0.069	-0.002	0.001	0.000	2190.54			
1989-01-06	2195.89	2184.29	2190.54	2205.18	2173.04	20310000.0	2172.822	...	0.011
	0.031	0.068	0.001	0.002	0.000	2194.29			
1989-01-09	2194.82	2195.89	2194.29	2213.75	2182.32	16500000.0	2175.144	...	0.005
	0.021	0.148	0.002	0.001	0.000	2199.46			
1989-01-10	2205.36	2194.82	2199.46	2209.11	2185.00	18420000.0	2181.322	...	0.014
	0.021	0.131	0.001	0.001	0.001	2193.21			

[5 rows x 38 columns]

Figure 7.6: Screenshot of printing the first five rows of the DataFrame

Since all features and driving factors are ready, we will now focus on regression algorithms that estimate the continuous target variables based on these predictive features.

Estimating with linear regression

The first regression model that comes to our mind is **linear regression**. Does it mean fitting data points using a linear function, as its name implies? Let's explore it.

How does linear regression work?

In simple terms, linear regression tries to fit as many of the data points as possible with a straight line in two-dimensional space or a plane in three-dimensional space. It explores the linear relationship between observations and targets, and the relationship is represented in a linear equation or weighted sum function. Given a data sample x with n features, x_1, x_2, \dots, x_n (x represents a feature vector and $x = (x_1, x_2, \dots, x_n)$), and weights (also

called **coefficients**) of the linear regression model w (w represents a vector (w_1, w_2, \dots, w_n)), the target y is expressed as follows:

$$y = w_1x_1 + w_2x_2 + \dots + w_nx_n = w^T x$$

Also, sometimes the linear regression model comes with an **intercept** (also called **bias**) w_0 , so the preceding linear relationship becomes as follows:

$$y = w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n = w^T x$$

Does it look familiar? The **logistic regression** algorithm you learned in *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*, is just an addition of logistic transformation on top of the linear regression, which maps the continuous weighted sum to the 0 (negative) or 1 (positive) class. Similarly, a linear regression model, or specifically its weight vector, w , is learned from the training data, with the goal of minimizing the estimation error defined as the **mean squared error (MSE)**, which measures the average of squares of difference between the truth and prediction. Given m training samples, $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots (x^{(i)}, y^{(i)}) \dots, (x^{(m)}, y^{(m)})$, the cost function $J(w)$ regarding the weights to be optimized is expressed as follows:

$$J(w) = \frac{1}{m} \sum_{i=1}^m \frac{1}{2} (\hat{y}(x^{(i)}) - y^{(i)})^2$$

Here, $\hat{y}(x^{(i)}) = w^T x^{(i)}$ is the prediction.

Again, we can obtain the optimal w so that $J(w)$ is minimized using gradient descent. The first-order derivative, the gradient Δw , is derived as follows:

$$\Delta w = \frac{1}{m} \sum_{i=1}^m -(y^{(i)} - \hat{y}(x^{(i)})) x^{(i)}$$

Combined with the gradient and learning rate η , the weight vector w can be updated in each step as follows:

$$w := w + \eta \frac{1}{m} \sum_{i=1}^m (y^{(i)} - \hat{y}(x^{(i)})) x^{(i)}$$

After a substantial number of iterations, the learned w is then used to predict a new sample x' as follows:

$$y' = w^T x'$$

After learning about the mathematical theory behind linear regression, let's implement it from scratch in the next section.

Implementing linear regression from scratch

Now that you have a thorough understanding of gradient-descent-based linear regression, we'll implement it from scratch.

We start by defining the function computing the prediction, $\hat{y}(x)$, with the current weights:

```
>>> def compute_prediction(X, weights):
    """
    ...
    Compute the prediction y_hat based on current weights
    ...
    predictions = np.dot(X, weights)
    ...
    return predictions
```

Then, we continue with the function updating the weight, w , with one step in a gradient descent manner, as follows:

```
>>> def update_weights_gd(X_train, y_train, weights,
learning_rate):
    """
    ...
    Update weights by one step and return updated weights
    ...
    predictions = compute_prediction(X_train, weights)
    weights_delta = np.dot(X_train.T, y_train - predictions)
    m = y_train.shape[0]
```

```
...     weights += learning_rate / float(m) * weights_delta
...
...     return weights
```

Next, we add the function that calculates the cost $J(w)$ as well:

```
>>> def compute_cost(X, y, weights):
...     """
...     Compute the cost J(w)
...     """
...     predictions = compute_prediction(X, weights)
...     cost = np.mean((predictions - y) ** 2 / 2.0)
...     return cost
```

Now, put all functions together with a model training function by performing the following tasks:

1. Update the weight vector in each iteration
2. Print out the current cost for every 100 (or it can be any number) iterations to ensure cost is decreasing and things are on the right track

Let's see how it's done by executing the following commands:

```
>>> def train_linear_regression(X_train, y_train, max_iter, learn
...     """
...     Train a linear regression model with gradient descent, a
...     """
...     if fit_intercept:
...         intercept = np.ones((X_train.shape[0], 1))
...         X_train = np.hstack((intercept, X_train))
...     weights = np.zeros(X_train.shape[1])
...     for iteration in range(max_iter):
...         weights = update_weights_gd(X_train, y_train,
...                                     weights, learning_rate)
...         # Check the cost for every 100 (for example) iterations
...         if iteration % 100 == 0:
...             print(compute_cost(X_train, y_train, weights))
...     return weights
```

Finally, predict the results of new input values using the trained model as follows:

```
>>> def predict(X, weights):
...     if X.shape[1] == weights.shape[0] - 1:
...         intercept = np.ones((X.shape[0], 1))
...         X = np.hstack((intercept, X))
...     return compute_prediction(X, weights)
```

Implementing linear regression is very similar to logistic regression, as you just saw. Let's examine it with a small example:

```
>>> X_train = np.array([[6], [2], [3], [4], [1],
...                     [5], [2], [6], [4], [7]])
>>> y_train = np.array([5.5, 1.6, 2.2, 3.7, 0.8,
...                     5.2, 1.5, 5.3, 4.4, 6.8])
```

Train a linear regression model with 100 iterations, at a learning rate of 0.01 based on intercept-included weights:

```
>>> weights = train_linear_regression(X_train, y_train,
...                                     max_iter=100, learning_rate=0.01, fit_intercept=True
```

Check the model's performance on new samples as follows:

```
>>> X_test = np.array([[1.3], [3.5], [5.2], [2.8]])
>>> predictions = predict(X_test, weights)
>>> import matplotlib.pyplot as plt
>>> plt.scatter(X_train[:, 0], y_train, marker='o', c='b')
>>> plt.scatter(X_test[:, 0], predictions, marker='*', c='k')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.show()
```

Refer to the following screenshot for the end result:

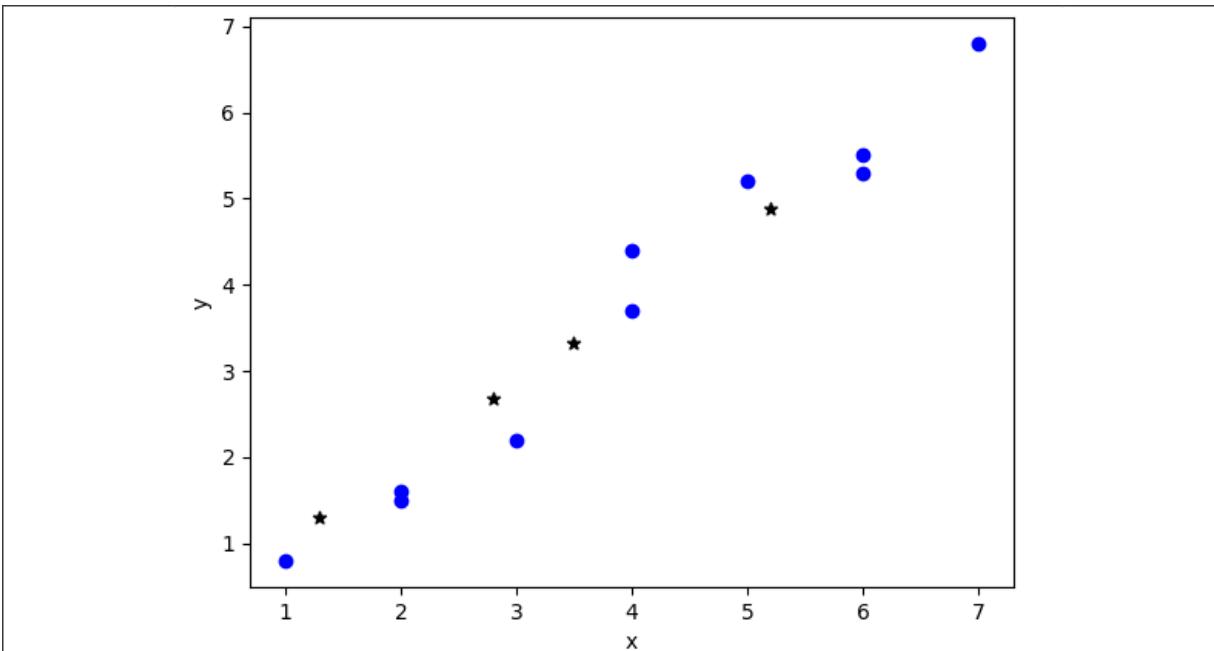


Figure 7.7: Linear regression on a toy dataset

The model we trained correctly predicts new samples (depicted by the stars).

Let's try it on another dataset, the diabetes dataset from scikit-learn:

```
>>> from sklearn import datasets
>>> diabetes = datasets.load_diabetes()
>>> print(diabetes.data.shape)
(442, 10)
>>> num_test = 30
>>> X_train = diabetes.data[:-num_test, :]
>>> y_train = diabetes.target[:-num_test]
```

Train a linear regression model with 5000 iterations, at a learning rate of 1 based on intercept-included weights (the cost is displayed every 500 iterations):

```
>>> weights = train_linear_regression(X_train, y_train,
                                     max_iter=5000, learning_rate=1, fit_intercept=True
2960.1229915
1539.55080927
```

```
1487.02495658
1480.27644342
1479.01567047
1478.57496091
1478.29639883
1478.06282572
1477.84756968
1477.64304737
>>> X_test = diabetes.data[-num_test:, :]
>>> y_test = diabetes.target[-num_test:]
>>> predictions = predict(X_test, weights)
>>> print(predictions)
[ 232.22305668 123.87481969 166.12805033 170.23901231
 228.12868839 154.95746522 101.09058779 87.33631249
 143.68332296 190.29353122 198.00676871 149.63039042
 169.56066651 109.01983998 161.98477191 133.00870377
 260.1831988 101.52551082 115.76677836 120.7338523
 219.62602446 62.21227353 136.29989073 122.27908721
 55.14492975 191.50339388 105.685612 126.25915035
 208.99755875 47.66517424]
>>> print(y_test)
[ 261. 113. 131. 174. 257. 55. 84. 42. 146. 212. 233.
 91. 111. 152. 120. 67. 310. 94. 183. 66. 173. 72.
 49. 64. 48. 178. 104. 132. 220. 57.]
```

The estimate is pretty close to the ground truth.

Next, let's utilize scikit-learn to implement linear regression.

Implementing linear regression with scikit-learn

So far, we have been using gradient descent in weight optimization but, like with logistic regression, linear regression is also open to **stochastic gradient descent (SGD)**. To use it, we can simply replace the `update_weights_gd` function with the `update_weights_sgd` function we created in *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*.

We can also directly use the SGD-based regression algorithm, `SGDRegressor`, from scikit-learn:

```
>>> from sklearn.linear_model import SGDRegressor  
>>> regressor = SGDRegressor(loss='squared_loss', penalty='l2',  
alpha=0.0001, learning_rate='constant', eta0=0.01, max_iter=100
```

Here, `'squared_loss'` for the `loss` parameter indicates that the cost function is MSE; `penalty` is the regularization term and it can be `None`, `l1`, or `l2`, which is similar to `SGDClassifier` in *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*, in order to reduce overfitting; `max_iter` is the number of iterations; and the remaining two parameters mean the learning rate is `0.01` and unchanged during the course of training. Train the model and output predictions on the testing set as follows:

```
>>> regressor.fit(X_train, y_train)  
>>> predictions = regressor.predict(X_test)  
>>> print(predictions)  
[ 231.03333725 124.94418254 168.20510142 170.7056729  
 226.52019503 154.85011364 103.82492496 89.376184  
 145.69862538 190.89270871 197.0996725 151.46200981  
 170.12673917 108.50103463 164.35815989 134.10002755  
 259.29203744 103.09764563 117.6254098 122.24330421  
 219.0996765 65.40121381 137.46448687 123.25363156  
 57.34965405 191.0600674 109.21594994 128.29546226  
 207.09606669 51.10475455]
```

You can also implement linear regression with TensorFlow. Let's see this in the next section.

Implementing linear regression with TensorFlow

First, we import TensorFlow and construct the model:

```
>>> import tensorflow as tf  
>>> layer0 = tf.keras.layers.Dense(units=1,  
                                 input_shape=[X_train.shape[1]])  
>>> model = tf.keras.Sequential(layer0)
```

It uses a linear layer (or you can think of it as a linear function) to connect the input in the `X_train.shape[1]` dimension and the output in `1` dimension.

Next, we specify the loss function, the MSE, and a gradient descent optimizer `Adam` with a learning rate of `1`:

```
>>> model.compile(loss='mean_squared_error',  
                  optimizer=tf.keras.optimizers.Adam(1))
```

Now we train the model for 100 iterations:

```
>>> model.fit(X_train, y_train, epochs=100, verbose=True)  
Epoch 1/100  
412/412 [=====] - 1s 2ms/sample - loss:  
Epoch 2/100  
412/412 [=====] - 0s 44us/sample - loss  
Epoch 3/100  
412/412 [=====] - 0s 47us/sample - loss  
Epoch 4/100  
412/412 [=====] - 0s 51us/sample - loss  
Epoch 5/100  
412/412 [=====] - 0s 44us/sample - loss  
.....  
Epoch 96/100  
412/412 [=====] - 0s 55us/sample - loss  
Epoch 97/100  
412/412 [=====] - 0s 44us/sample - loss  
Epoch 98/100  
412/412 [=====] - 0s 52us/sample - loss  
Epoch 99/100  
412/412 [=====] - 0s 47us/sample - loss  
Epoch 100/100  
412/412 [=====] - 0s 46us/sample - loss
```

This also prints out the loss for every iteration. Finally, we make predictions using the trained model:

```
>>> predictions = model.predict(X_test)[:, 0]
>>> print(predictions)
[231.52155 124.17711 166.71492 171.3975 227.70126 152.0252
 103.01532 91.79277 151.07457 190.01042 190.60373 152.5227
 168.92166 106.18033 167.02473 133.37477 259.24756 101.5125
 119.43106 120.893005 219.37921 64.873634 138.43217 123.6656
 56.33039 189.27441 108.67446 129.12535 205.06857 47.9946
```

The next regression algorithm you will be learning about is decision tree regression.

Estimating with decision tree regression

Decision tree regression is also called **regression tree**. It is easy to understand a regression tree by comparing it with its sibling, the classification tree, which you are already familiar with.

Transitioning from classification trees to regression trees

In classification, a decision tree is constructed by recursive binary splitting and growing each node into left and right children. In each partition, it greedily searches for the most significant combination of features and its value as the optimal splitting point. The quality of separation is measured by the weighted purity of labels of the two resulting children, specifically via Gini Impurity or Information Gain. In regression, the tree construction process is almost identical to the classification one, with only two differences due to the fact that the target becomes continuous:

- The quality of the splitting point is now measured by the weighted MSE of two children; the MSE of a child is equivalent to the variance of all target values, and the smaller the weighted MSE, the better the split
- The **average** value of targets in a terminal node becomes the leaf value, instead of the majority of labels in the classification tree

To make sure you understand regression trees, let's work on a small example of house price estimation using the features **house type** and **number of bedrooms**:

Type	Number of bedrooms	Price (thousand)
Semi	3	600
Detached	2	700
Detached	3	800
Semi	2	400
Semi	4	700

Figure 7.8: Toy dataset of house prices

We first define the MSE and weighted MSE computation functions that will be used in our calculation:

```
>>> def mse(targets):
...     # When the set is empty
...     if targets.size == 0:
...         return 0
...     return np.var(targets)
```

Then we define the weighted MSE after a split in a node:

```
>>> def weighted_mse(groups):
...     """
...     Calculate weighted MSE of children after a split
...     """
...     # Add code here
```

```

...
    total = sum(len(group) for group in groups)
...
    weighted_sum = 0.0
...
    for group in groups:
        weighted_sum += len(group) / float(total) * mse(group)
...
    return weighted_sum

```

Test things out by executing the following commands:

```

>>> print(f'{mse(np.array([1, 2, 3])):.4f}')
0.6667
>>> print(f'{weighted_mse([np.array([1, 2, 3]), np.array([1, 2])]):.4f}')
0.5000

```

To build the house price regression tree, we first exhaust all possible pairs of feature and value, and we compute the corresponding MSE:

```

MSE(type, semi) = weighted_mse([[600, 400, 700], [700, 800]]) =
MSE(bedroom, 2) = weighted_mse([[700, 400], [600, 800, 700]]) =
MSE(bedroom, 3) = weighted_mse([[600, 800], [700, 400, 700]]) =
MSE(bedroom, 4) = weighted_mse([[700], [600, 700, 800, 400]]) =

```

The lowest MSE is achieved with the `type, semi` pair, and the root node is then formed by this splitting point. The result of this partition is as follows:

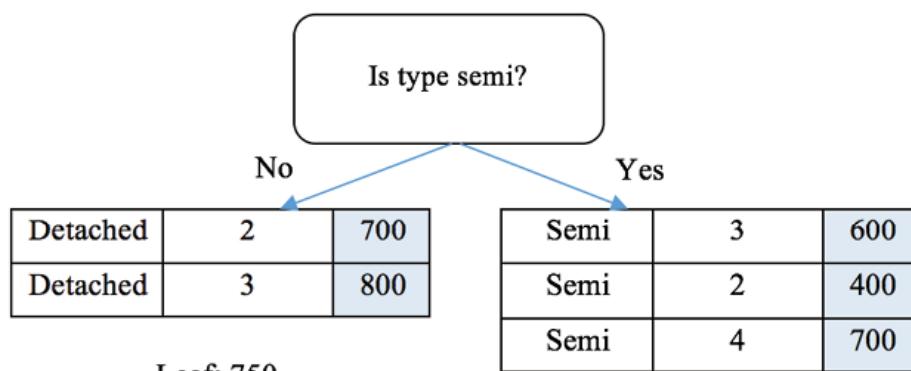


Figure 7.9: Splitting using (type=semi)

If we are satisfied with a one-level regression tree, we can stop here by assigning both branches as leaf nodes with the value as the average of targets of the samples included. Alternatively, we can go further down the road by constructing the second level from the right branch (the left branch can't be split further):

```
MSE(bedroom, 2) = weighted_mse([], [600, 400, 700]) = 15556
MSE(bedroom, 3) = weighted_mse([[400], [600, 700]]) = 1667
MSE(bedroom, 4) = weighted_mse([[400, 600], [700]]) = 6667
```

With the second splitting point specified by the `bedroom, 3` pair (whether it has at least three bedrooms or not) with the lowest MSE, our tree becomes as shown in the following diagram:

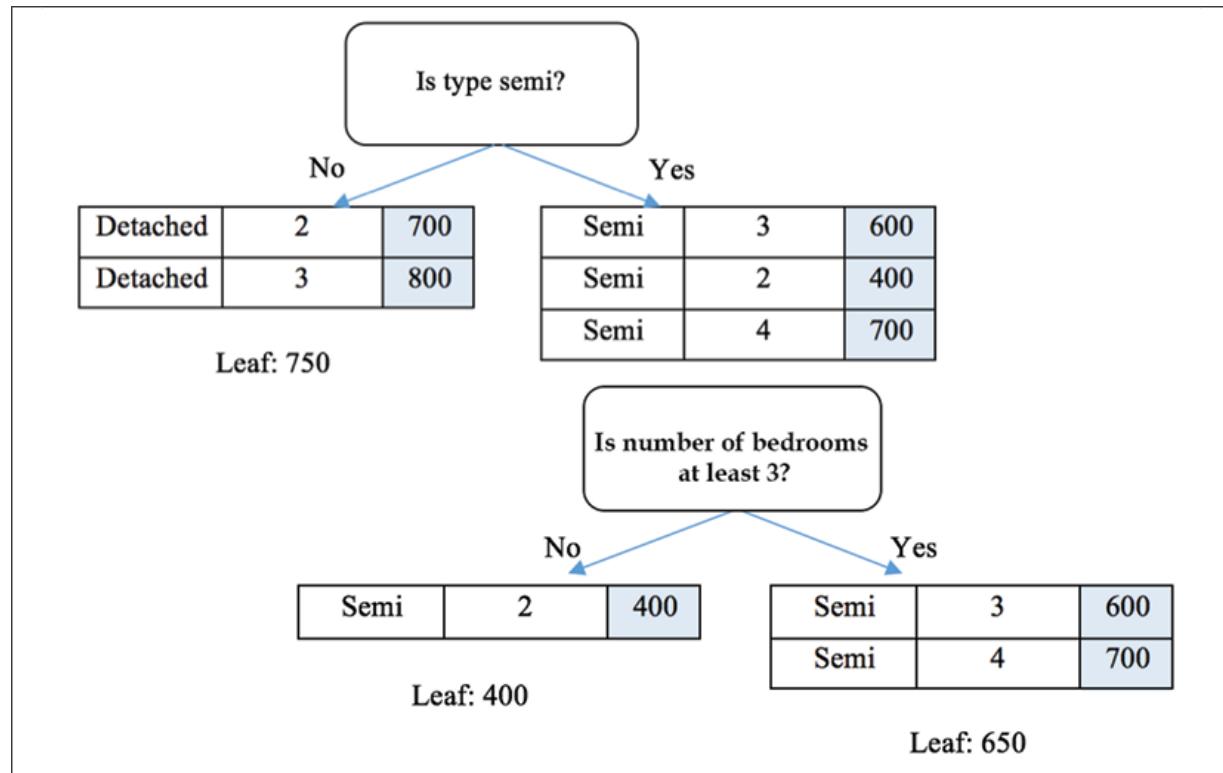


Figure 7.10: Splitting using $(\text{bedroom} \geq 3)$

We can finish up the tree by assigning average values to both leaf nodes.

Implementing decision tree regression

Now that you're clear about the regression tree construction process, it's time for coding.

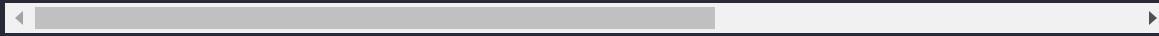
The node splitting utility function we will define in this section is identical to what we used in *Chapter 4, Predicting Online Ad Click-Through with Tree-Based Algorithms*, which separates samples in a node into left and right branches based on a feature and value pair:

```
>>> def split_node(X, y, index, value):
...     """
...     Split data set X, y based on a feature and a value
...     @param index: index of the feature used for splitting
...     @param value: value of the feature used for splitting
...     @return: left and right child, a child is in the format
...             """
...     x_index = X[:, index]
...     # if this feature is numerical
...     if type(X[0, index]) in [int, float]:
...         mask = x_index >= value
...     # if this feature is categorical
...     else:
...         mask = x_index == value
...     # split into left and right child
...     left = [X[~mask, :], y[~mask]]
...     right = [X[mask, :], y[mask]]
...     return left, right
```

Next, we define the greedy search function, trying out all possible splits and returning the one with the least weighted MSE:

```
>>> def get_best_split(X, y):
...     """
...     Obtain the best splitting point and resulting children f
...     @return: {index: index of the feature, value: feature va
...             """
...     best_index, best_value, best_score, children =
...             None, None, 1e10, None
...     for index in range(len(X[0])):
...         for value in np.sort(np.unique(X[:, index])):
...             score = split_node(X, y, index, value)
```

```
...     groups = split_node(x, y, index, value)
...     impurity = weighted_mse(
...         [groups[0][1], groups[1][1]])
...     if impurity < best_score:
...         best_index, best_value, best_score, children
...             = index, value, impurity, groups
...     return {'index': best_index, 'value': best_value,
...             'children': children}
```



The preceding selection and splitting process occurs in a recursive manner on each of the subsequent children. When a stopping criterion is met, the process at a node stops, and the mean value of the sample `targets` will be assigned to this terminal node:

```
>>> def get_leaf(targets):
...     # Obtain the leaf as the mean of the targets
...     return np.mean(targets)
```

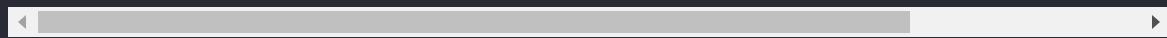
And finally, here is the recursive function, `split`, that links it all together. It checks whether any stopping criteria are met and assigns the leaf node if so, or proceeds with further separation otherwise:

```
>>> def split(node, max_depth, min_size, depth):
...     """
...     Split children of a node to construct new nodes or assign leaf
...     @param node: dict, with children info
...     @param max_depth: maximal depth of the tree
...     @param min_size: minimal samples required to further split
...     @param depth: current depth of the node
...     """
...     left, right = node['children']
...     del (node['children'])
...     if left[1].size == 0:
...         node['right'] = get_leaf(right[1])
...         return
...     if right[1].size == 0:
...         node['left'] = get_leaf(left[1])
...         return
...     # Check if the current depth exceeds the maximal depth
...     if depth >= max_depth:
...         node['left'], node['right'] = get_leaf(
...             left[1]), get_leaf(right[1])
...         return
```

```

...
    return
...
    # Check if the left child has enough samples
    if left[1].size <= min_size:
        node['left'] = get_leaf(left[1])
    else:
        # It has enough samples, we further split it
        result = get_best_split(left[0], left[1])
        result_left, result_right = result['children']
        if result_left[1].size == 0:
            node['left'] = get_leaf(result_right[1])
        elif result_right[1].size == 0:
            node['left'] = get_leaf(result_left[1])
        else:
            node['left'] = result
            split(node['left'], max_depth, min_size, depth + 1)
    # Check if the right child has enough samples
    if right[1].size <= min_size:
        node['right'] = get_leaf(right[1])
    else:
        # It has enough samples, we further split it
        result = get_best_split(right[0], right[1])
        result_left, result_right = result['children']
        if result_left[1].size == 0:
            node['right'] = get_leaf(result_right[1])
        elif result_right[1].size == 0:
            node['right'] = get_leaf(result_left[1])
        else:
            node['right'] = result
            split(node['right'], max_depth, min_size,
                  depth + 1)

```



The entry point of the regression tree construction is as follows:

```

>>> def train_tree(X_train, y_train, max_depth, min_size):
...     root = get_best_split(X_train, y_train)
...     split(root, max_depth, min_size, 1)
...     return root

```

Now, let's test it with a hand-calculated example:

```

>>> X_train = np.array([[semi', 3],
...                      ['detached', 2],
...                      ['detached', 3],

```

```

...
['semi', 2],
['semi', 4]], dtype=object)
>>> y_train = np.array([600, 700, 800, 400, 700])
>>> tree = train_tree(X_train, y_train, 2, 2)

```

To verify the trained tree is identical to what we constructed by hand, we write a function displaying the tree:

```

>>> CONDITION = {'numerical': {'yes': '>=', 'no': '<'},
...                 'categorical': {'yes': 'is', 'no': 'is not'}}
>>> def visualize_tree(node, depth=0):
...     if isinstance(node, dict):
...         if type(node['value']) in [int, float]:
...             condition = CONDITION['numerical']
...         else:
...             condition = CONDITION['categorical']
...         print('{}|- X{} {} {}'.format(depth * ' ', 
...                                         node['index'] + 1, condition['no'],
...                                         node['value']))
...         if 'left' in node:
...             visualize_tree(node['left'], depth + 1)
...             print('{}|- X{} {} {}'.format(depth * ' ', 
...                                         node['index'] + 1, condition['yes'],
...                                         node['value']))
...         if 'right' in node:
...             visualize_tree(node['right'], depth + 1)
...     else:
...         print('{}[{}]'.format(depth * ' ', node))
>>> visualize_tree(tree)
|- X1 is not detached
 |- X2 < 3
   [400.0]
 |- X2 >= 3
   [650.0]
|- X1 is detached
 [750.0]

```

Now that you have a better understanding of the regression tree after implementing it from scratch, we can directly use the `DecisionTreeRegressor` package (<https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeRegressor.html>) from scikit-learn. Let's apply it on an example of predicting Boston house prices as follows:

```
>>> boston = datasets.load_boston()
>>> num_test = 10 # the last 10 samples as testing set
>>> X_train = boston.data[:-num_test, :]
>>> y_train = boston.target[:-num_test]
>>> X_test = boston.data[-num_test:, :]
>>> y_test = boston.target[-num_test:]
>>> from sklearn.tree import DecisionTreeRegressor
>>> regressor = DecisionTreeRegressor(max_depth=10,
                                         min_samples_split=3)
>>> regressor.fit(X_train, y_train)
>>> predictions = regressor.predict(X_test)
>>> print(predictions)
[12.7 20.9 20.9 20.2 20.9 30.8
 20.73076923 24.3 28.2 20.73076923]
```

Compare predictions with the ground truth as follows:

```
>>> print(y_test)
[ 19.7  18.3  21.2  17.5  16.8  22.4  20.6  23.9  22.  11.9]
```

We have implemented a regression tree in this section. Is there an ensemble version of the regression tree? Let's see next.

Implementing a regression forest

In *Chapter 4, Predicting Online Ad Click-Through with Tree-Based Algorithms*, we explored **random forests** as an ensemble learning method by combining multiple decision trees that are separately trained and randomly subsampling training features in each node of a tree. In classification, a random forest makes a final decision by a majority vote of all tree decisions. Applied to regression, a random forest regression model (also called a **regression forest**) assigns the average of regression results from all decision trees to the final decision.

Here, we will use the regression forest package `RandomForestRegressor` from scikit-learn and deploy it in our Boston house price prediction example:

```

>>> from sklearn.ensemble import RandomForestRegressor
>>> regressor = RandomForestRegressor(n_estimators=100,
                                     max_depth=10, min_samples_split=3)
>>> regressor.fit(X_train, y_train)
>>> predictions = regressor.predict(X_test)
>>> print(predictions)
[ 19.34404351 20.93928947 21.66535354 19.99581433 20.873871
 25.52030056 21.33196685 28.34961905 27.54088571 21.32508585]

```

The third regression algorithm that we want to explore is **support vector regression (SVR)**.

Estimating with support vector regression

As the name implies, SVR is part of the support vector family and a sibling of the **support vector machine (SVM)** for classification (or we can just call it **SVC**) that you learned about in *Chapter 3, Recognizing Faces with Support Vector Machine*.

To recap, SVC seeks an optimal hyperplane that best segregates observations from different classes. Suppose a hyperplane is determined by a slope vector w and intercept b , and the optimal hyperplane is picked so

1
that the distance (which can be expressed as $\frac{1}{||w||}$) from the nearest points in each of the segregated spaces to the hyperplane is maximized. The optimal w and b can be learned and solved with the following optimization problem:

- Minimizing $||w||$
- Subject to $y^{(i)}(wx^{(i)} + b) \geq 1$, for a training set of $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$, ... $(x^{(i)}, y^{(i)})$..., $(x^{(m)}, y^{(m)})$

In SVR, our goal is to find a decision hyperplane (defined by a slope vector w and intercept b) so that two hyperplanes $wx+b=-\varepsilon$ (negative hyperplane) and $wx+b=\varepsilon$ (positive hyperplane) can cover most training data. In other words, most of the data points are bounded in the ε bands of the optimal hyperplane. And at the same time, the optimal hyperplane is as flat as possible, which means w is as small as possible, as shown in the following diagram:

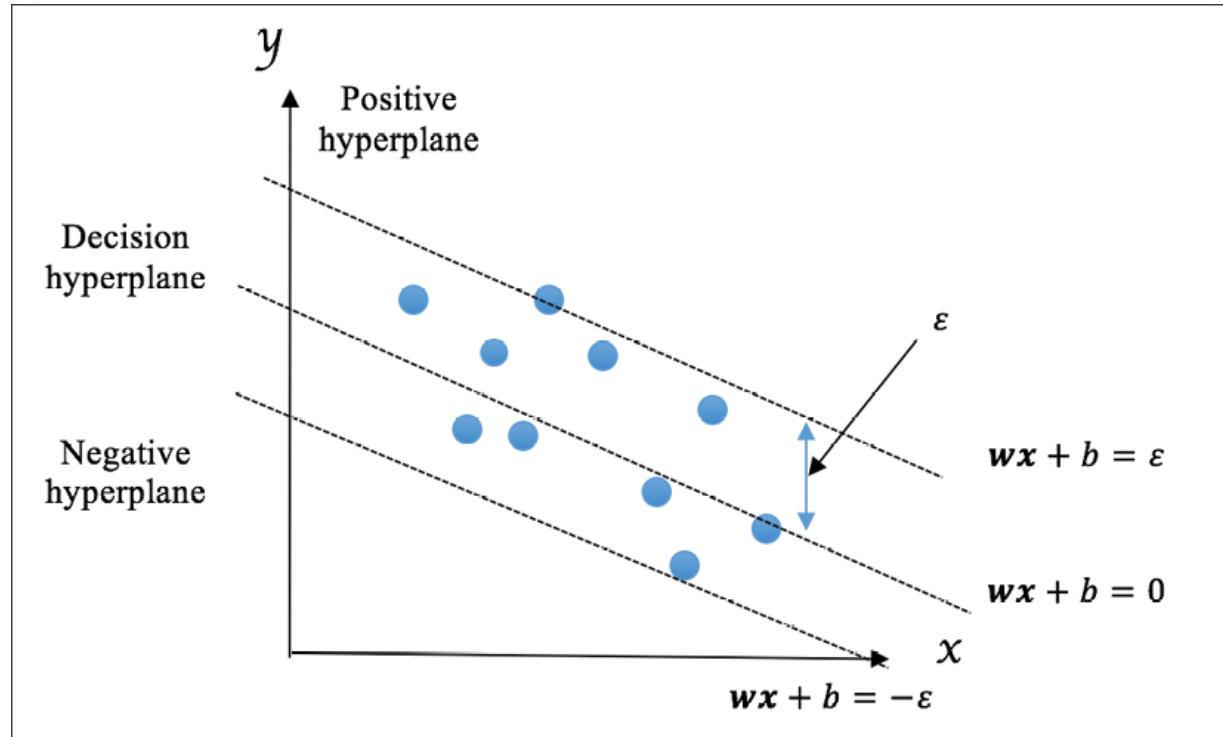


Figure 7.11: Finding the decision hyperplane in SVR

This translates into deriving the optimal w and b by solving the following optimization problem:

- Minimizing $\|w\|$
- Subject to $|y^{(i)} - (wx^{(i)} + b)| \leq \varepsilon$, given a training set of $(x^{(1)}, y^{(1)})$, $(x^{(2)}, y^{(2)})$, ... $(x^{(i)}, y^{(i)})$, ..., $(x^{(m)}, y^{(m)})$

The theory behind SVR is very similar to SVM. In the next section, let's see the implementation of SVR.

Implementing SVR

Again, to solve the preceding optimization problem, we need to resort to quadratic programming techniques, which are beyond the scope of our learning journey. Therefore, we won't cover the computation methods in detail and will implement the regression algorithm using the `SVR` package (<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html>) from scikit-learn.

Important techniques used in SVM, such as penalty as a trade off between bias and variance, and the kernel (RBF, for example) handling linear non-separation, are transferable to SVR. The `SVR` package from scikit-learn also supports these techniques.

Let's solve the previous house price prediction problem with `SVR` this time:

```
>>> from sklearn.svm import SVR
>>> regressor = SVR(C=0.1, epsilon=0.02, kernel='linear')
>>> regressor.fit(X_train, y_train)
>>> predictions = regressor.predict(X_test)
>>> print(predictions)
[ 14.59908201 19.32323741 21.16739294 18.53822876 20.1960847
 23.74076575 22.65713954 26.98366295 25.75795682 22.69805145]
```

You've learned about three (or four) regression algorithms. So, how should we evaluate regression performance? Let's find out in the next section.

Evaluating regression performance

So far, we've covered three popular regression algorithms in depth and implemented them from scratch by using several prominent libraries. Instead of judging how well a model works on testing sets by printing out

the prediction, we need to evaluate its performance with the following metrics, which give us better insights:

- The MSE, as I mentioned, measures the squared loss corresponding to the expected value. Sometimes the square root is taken on top of the MSE in order to convert the value back into the original scale of the target variable being estimated. This yields the **root mean squared error (RMSE)**. Also, the RMSE has the benefit of penalizing large errors more since we first calculate the square of an error.
- The **mean absolute error (MAE)** on the other hand measures the absolute loss. It uses the same scale as the target variable and gives us an idea of how close the predictions are to the actual values.



For both the MSE and MAE, the smaller the value, the better the regression model.

- R^2 (pronounced **r squared**) indicates the goodness of the fit of a regression model. It is the fraction of the dependent variable variation that a regression model is able to explain. It ranges from 0 to 1, representing from no fit to a perfect prediction. There is a variant of R^2 called **adjusted R²**. It adjusts for the number of features in a model relative to the number of data points.

Let's compute these three measurements on a linear regression model using corresponding functions from scikit-learn:

1. We will work on the diabetes dataset again and fine-tune the parameters of the linear regression model using the grid search technique:

```
>>> diabetes = datasets.load_diabetes()
>>> num_test = 30 # the last 30 samples as testing set
>>> X_train = diabetes.data[:-num_test, :]
>>> y_train = diabetes.target[:-num_test]
>>> X_test = diabetes.data[-num_test:, :]
>>> y_test = diabetes.target[-num_test:]
>>> param_grid = {
...     "alpha": [1e-07, 1e-06, 1e-05],
```

```
...     "penalty": [None, "l2"],
...     "eta0": [0.03, 0.05, 0.1],
...     "max_iter": [500, 1000]
... }
>>> from sklearn.model_selection import GridSearchCV
>>> regressor = SGDRegressor(loss='squared_loss',
                               learning_rate='constant',
                               random_state=42)
>>> grid_search = GridSearchCV(regressor, param_grid, cv=3)
```

2. We obtain the optimal set of parameters:

```
>>> grid_search.fit(X_train, y_train)
>>> print(grid_search.best_params_)
{'alpha': 1e-07, 'eta0': 0.05, 'max_iter': 500, 'penalty': l
>>> regressor_best = grid_search.best_estimator_
```

3. We predict the testing set with the optimal model:

```
>>> predictions = regressor_best.predict(X_test)
```

4. We evaluate the performance on testing sets based on the MSE, MAE, and R² metrics:

```
>>> from sklearn.metrics import mean_squared_error,
      mean_absolute_error, r2_score
>>> mean_squared_error(y_test, predictions)
1933.3953304460413
>>> mean_absolute_error(y_test, predictions)
35.48299900764652
>>> r2_score(y_test, predictions)
0.6247444629690868
```

Now that you've learned about three (or four, you could say) commonly used and powerful regression algorithms and performance evaluation metrics, let's utilize each of them to solve our stock price prediction problem.

Predicting stock prices with the three regression algorithms

Here are the steps to predict the stock price:

1. Earlier, we generated features based on data from 1988 to 2019, and we will now continue with constructing the training set with data from 1988 to 2018 and the testing set with data from 2019:

```
>>> data_raw = pd.read_csv('19880101_20191231.csv', index_col=0)
>>> data = generate_features(data_raw)
>>> start_train = '1988-01-01'
>>> end_train = '2018-12-31'
>>> start_test = '2019-01-01'
>>> end_test = '2019-12-31'
>>> data_train = data.loc[start_train:end_train]
>>> X_train = data_train.drop('close', axis=1).values
>>> y_train = data_train['close'].values
>>> print(X_train.shape)
(7558, 37)
>>> print(y_train.shape)
(7558,)
```

All fields in the `dataframe` data except `'close'` are feature columns, and `'close'` is the target column. We have 7,558 training samples and each sample is 37-dimensional. We also have 251 testing samples:

```
>>> print(X_test.shape)
(251, 37)
```

2. We will first experiment with SGD-based linear regression. Before we train the model, you should realize that SGD-based algorithms are sensitive to data with features at very different scales; for example, in our case, the average value of the `open` feature is around 8,856, while that of the `moving_avg_365` feature is 0.00037 or so. Hence, we need to normalize features into the same or a comparable scale. We do so by removing the mean and rescaling to unit variance:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
```

3. We rescale both sets with `scaler` taught by the training set:

```
>>> X_scaled_train = scaler.fit_transform(X_train)
>>> X_scaled_test = scaler.transform(X_test)
```

4. Now we can search for the SGD-based linear regression with the optimal set of parameters. We specify `l2` regularization and 1,000 iterations, and tune the regularization term multiplier, `alpha`, and initial learning rate, `eta0`:

```
>>> param_grid = {
...     "alpha": [1e-4, 3e-4, 1e-3],
...     "eta0": [0.01, 0.03, 0.1],
... }
>>> lr = SGDRegressor(penalty='l2', max_iter=1000, random_state=42)
>>> grid_search = GridSearchCV(lr, param_grid, cv=5, scoring='neg_mean_squared_error')
>>> grid_search.fit(X_scaled_train, y_train)
```

5. Select the best linear regression model and make predictions of the testing samples:

```
>>> print(grid_search.best_params_)
{'alpha': 0.0001, 'eta0': 0.03}
>>> lr_best = grid_search.best_estimator_
>>> predictions_lr = lr_best.predict(X_scaled_test)
```

6. Measure the prediction performance via the MSE, MAE, and R2:

```
>>> print(f'MSE: {mean_squared_error(y_test, predictions_lr)}')
MSE: 41631.128
>>> print(f'MAE: {mean_absolute_error(y_test, predictions_lr)}')
MAE: 154.989
>>> print(f'R^2: {r2_score(y_test, predictions_lr):.3f}')
R^2: 0.964
```

We achieve an R^2 of 0.964 with a fine-tuned linear regression model.

7. Similarly, let's experiment with a random forest. We specify 100 trees to ensemble and tune the maximum depth of the tree, `max_depth`; the minimum number of samples required to further split a

node, `min_samples_split`; and the number of features used for each tree, as well as the following:

```
>>> param_grid = {  
...     'max_depth': [30, 50],  
...     'min_samples_split': [2, 5, 10],  
...     'min_samples_leaf': [3, 5]  
... }  
>>> rf = RandomForestRegressor(n_estimators=100, n_jobs=-1,  
>>> grid_search = GridSearchCV(rf, param_grid, cv=5,  
...                             scoring='r2', n_jobs=-1)  
>>> grid_search.fit(X_train, y_train)
```

Note this may take a while, hence we use all available CPU cores for training.

8. Select the best regression forest model and make predictions of the testing samples:

```
>>> print(grid_search.best_params_)  
{'max_depth': 30, 'min_samples_leaf': 3, 'min_samples_split': 5}  
>>> rf_best = grid_search.best_estimator_  
>>> predictions_rf = rf_best.predict(X_test)
```

9. Measure the prediction performance as follows:

```
>>> print(f'MSE: {mean_squared_error(y_test, predictions_rf)}')  
MSE: 404310.522  
>>> print(f'MAE: {mean_absolute_error(y_test, predictions_rf)}')  
MAE: 419.398  
>>> print(f'R^2: {r2_score(y_test, predictions_rf):.3f}')  
R^2: 0.647
```

An R^2 of `0.647` is obtained with a tweaked forest regressor.

10. Next, we work with SVR with a linear and RBF kernel and leave the penalty hyperparameters `C` and `ε`, as well as the kernel coefficient of RBF, for fine-tuning. Similar to SGD-based algorithms, SVR doesn't work well on data with feature scale disparity:

```
>>> param_grid = [  
...     {'kernel': ['linear'], 'C': [100, 300, 500],  
...     'epsilon': [0.0001, 0.001, 0.01, 0.1, 1, 10, 100]}]
```

```
        'epsilon': [0.00003, 0.0001]},  
...     {'kernel': ['rbf'], 'gamma': [1e-3, 1e-4],  
...      'C': [10, 100, 1000], 'epsilon': [0.00003, 0.0001]}  
... ]
```

11. Again, to work around this, we use the rescaled data to train the `SVR` model:

```
>>> svr = SVR()  
>>> grid_search = GridSearchCV(svr, param_grid, cv=5, scoring='neg_mean_squared_error')  
>>> grid_search.fit(X_scaled_train, y_train)
```

12. Select the best `SVR` model and make predictions of the testing samples:

```
>>> print(grid_search.best_params_)  
{'C': 500, 'epsilon': 0.0001, 'kernel': 'linear'}  
>>> svr_best = grid_search.best_estimator_  
>>> predictions_svr = svr_best.predict(X_scaled_test)  
>>> print(f'MSE: {mean_squared_error(y_test, predictions_svr)}')  
MSE: 29999.827  
>>> print(f'MAE: {mean_absolute_error(y_test, predictions_svr)}')  
MAE: 123.566  
>>> print(f'R^2: {r2_score(y_test, predictions_svr):.3f}')  
R^2: 0.974
```

- With SVR, we're able to achieve an R^2 of 0.974 on the testing set.
13. We also plot the prediction generated by each of the three algorithms, along with the ground truth:

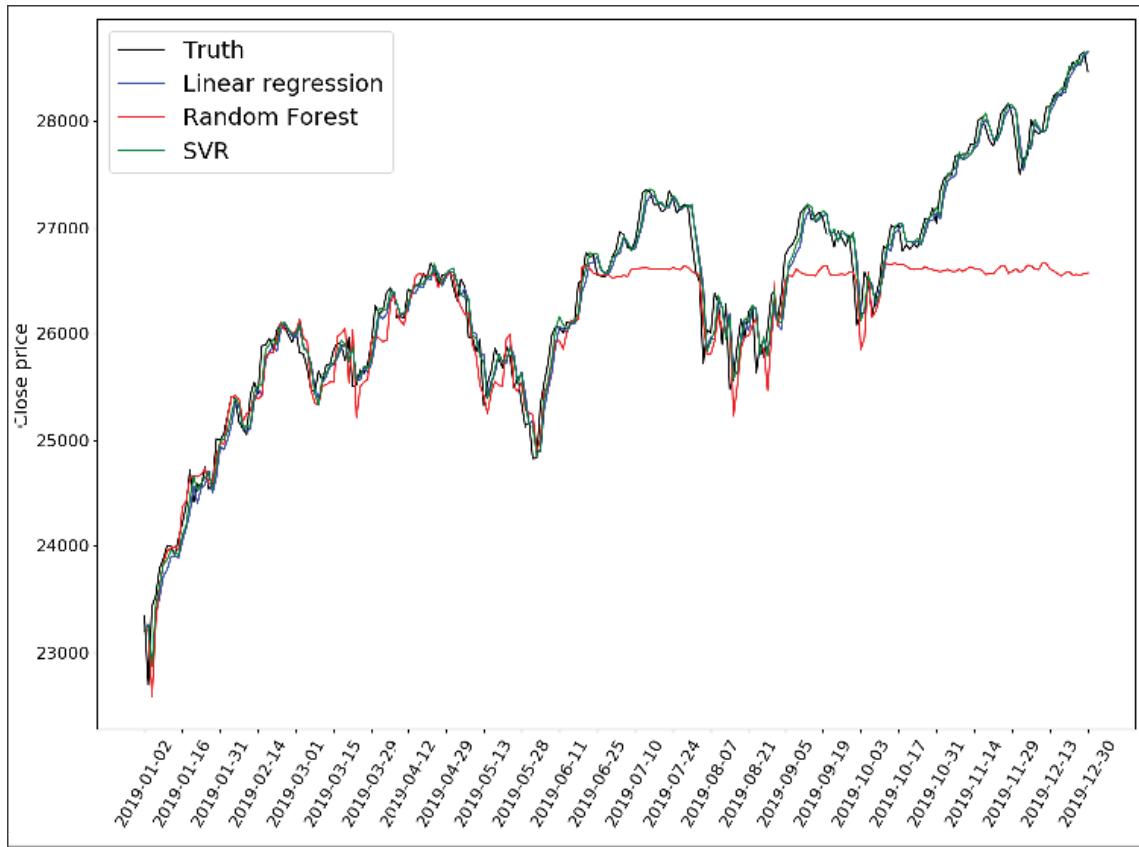


Figure 7.12: Predictions using the three algorithms versus the ground truth

The visualization is produced by the following code:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(data_test.index, y_test, c='k')
>>> plt.plot(data_test.index, predictions_lr, c='b')
>>> plt.plot(data_test.index, predictions_rf, c='r')
>>> plt.plot(data_test.index, predictions_svr, c='g')
>>> plt.xticks(range(0, 252, 10), rotation=60)
>>> plt.xlabel('Date')
>>> plt.ylabel('Close price')
>>> plt.legend(['Truth', 'Linear regression', 'Random Forest', 'SVR'])
>>> plt.show()
```

We've built a stock predictor using three regression algorithms individually in this section. Overall, SVR outperforms the other two algorithms.

Summary

In this chapter, we worked on the last project in this book, predicting stock (specifically stock index) prices using machine learning regression techniques. We started with a short introduction to the stock market and factors that influence trading prices. To tackle this billion-dollar problem, we investigated machine learning regression, which estimates a continuous target variable, as opposed to discrete output in classification. We followed this with an in-depth discussion of three popular regression algorithms, linear regression, regression trees and regression forests, and SVR. We covered their definitions, mechanics, and implementations from scratch with several popular frameworks, including scikit-learn and TensorFlow, along with applications on toy datasets. You also learned the metrics used to evaluate a regression model. Finally, we applied what was covered in this whole chapter to solve our stock price prediction problem.

In the next chapter, we will continue working on the stock price prediction project, but with powerful **neural networks**. We will see whether they can beat what we have achieved with the three regression models in this chapter.

Exercises

1. As mentioned, can you add more signals to our stock prediction system, such as the performance of other major indexes? Does this improve prediction?
2. Recall that I briefly mentioned several major stock indexes besides DJIA. Is it possible to improve on the DJIA price prediction model we just developed by considering the historical prices and performances of these major indexes? It's highly likely! The idea behind this is that no stock or index is isolated and that there are weak or strong influences between stocks and different financial markets. This should be intriguing to explore.

3. Can you try to ensemble linear regression and SVR, for example, averaging the prediction, and see if you can improve the prediction?

Predicting Stock Prices with Artificial Neural Networks

Continuing the same project of stock price prediction from the last chapter, in this chapter I will introduce and explain neural network models in depth. We will start by building the simplest neural network and go deeper by adding more layers to it. We will cover neural network building blocks and other important concepts, including activation functions, feedforward, and backpropagation. We will also implement neural networks from scratch with scikit-learn and TensorFlow. We will pay attention to how to learn with neural networks efficiently without overfitting, utilizing dropout and early stopping techniques. Finally, we will train a neural network to predict stock prices and see whether it can beat what we achieved with the three regression algorithms in the previous chapter.

We will cover the following topics in this chapter:

- Demystifying neural networks
- From shallow neural networks to deep learning
- Implementation of neural networks from scratch
- Implementation of neural networks with scikit-learn
- Implementation of neural networks with TensorFlow
- Activation functions
- Dropout
- Early stopping
- Predicting stock prices with neural networks
- Fine-tuning a neural network

Demystifying neural networks

Here comes probably the most frequently mentioned model in the media, **artificial neural networks (ANNs)**; more often we just call them **neural networks**. Interestingly, the neural network has been (falsely) considered equivalent to machine learning or artificial intelligence by the general public.



The ANN is just one type of algorithm among many in machine learning. And machine learning is a branch of artificial intelligence. It is one of the ways we achieve general artificial intelligence.

Regardless, it is one of the most important machine learning models and has been rapidly evolving along with the revolution of **deep learning (DL)**. Let's first understand how neural networks work.

Starting with a single-layer neural network

I will first talk about different layers in a network, then the activation function, and finally training a network with backpropagation.

Layers in neural networks

A simple neural network is composed of three layers: the **input layer**, **hidden layer**, and **output layer**, as shown in the following diagram:

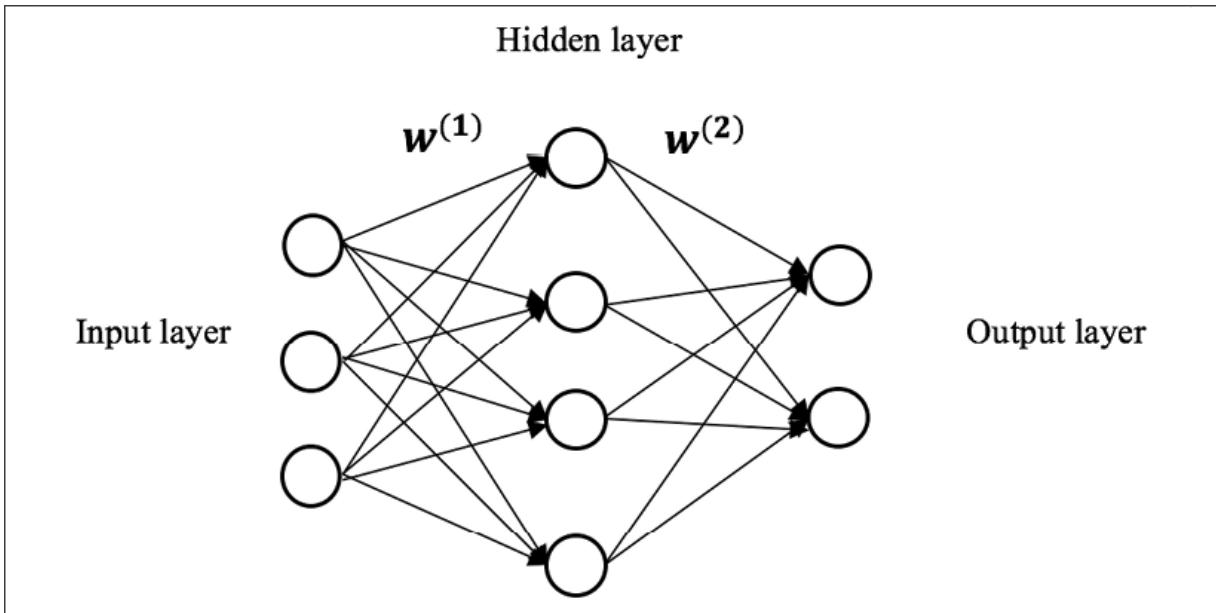


Figure 8.1: A simple shallow neural network

A **layer** is a conceptual collection of **nodes** (also called **units**), which simulate neurons in a biological brain. The input layer represents the input features, x , and each node is a predictive feature, x . The output layer represents the target variable(s).

In binary classification, the output layer contains only one node, whose value is the probability of the positive class. In multiclass classification, the output layer consists of n nodes, where n is the number of possible classes and the value of each node is the probability of predicting that class. In regression, the output layer contains only one node, the value of which is the prediction result.

The hidden layer can be considered a composition of latent information extracted from the previous layer. There can be more than one hidden layer. Learning with a neural network with two or more hidden layers is called **DL**. In this chapter, we will focus on one hidden layer to begin with.

Two adjacent layers are connected by conceptual edges (sort of like the synapses in a biological brain), which transmit signals from one neuron in a layer to another neuron in the next layer. The **edges** are parameterized by the weights, W , of the model. For example, $W^{(1)}$ in the preceding diagram

connects the input and hidden layers and $W^{(2)}$ connects the hidden and output layers.

In a standard neural network, data are conveyed only from the input layer to the output layer, through a hidden layer(s). Hence, this kind of network is called a **feedforward** neural network. Basically, logistic regression is a feedforward neural network with no hidden layer where the output layer connects directly with the input layer. Neural networks with one or more hidden layers between the input and output layer should be able to learn more about the underlying relationship between the input data and the target.

Activation functions

Suppose the input, x , is of n dimensions and the hidden layer is composed of H hidden units. The weight matrix, $W^{(1)}$, connecting the input and hidden layer is of size n by H , where each column, $w_h^{(1)}$, represents the coefficients associating the input with the h -th hidden unit. The output (also called **activation**) of the hidden layer can be expressed mathematically as follows:

$$a^{(2)} = f(z^{(2)}) = f(W^{(1)}x)$$

Here, $f(z)$ is an activation function. As its name implies, the activation function checks how activated each neuron is, simulating the way our brains work. Typical activation functions include the logistic function (more often called the **sigmoid** function in neural networks) and the **tanh** function, which is considered a re-scaled version of the logistic function, as well as **ReLU** (short for **Rectified Linear Unit**), which is often used in DL:

$$\text{sigmoid}(z) = \frac{1}{1 + e^{-z}}$$

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{2}{1 + e^{-2z}} - 1$$

$$relu(z) = z^+ = \max(0, z)$$

We plot these three activation functions as follows:

- The **logistic (sigmoid)** function where the output value is in the range of (0, 1):

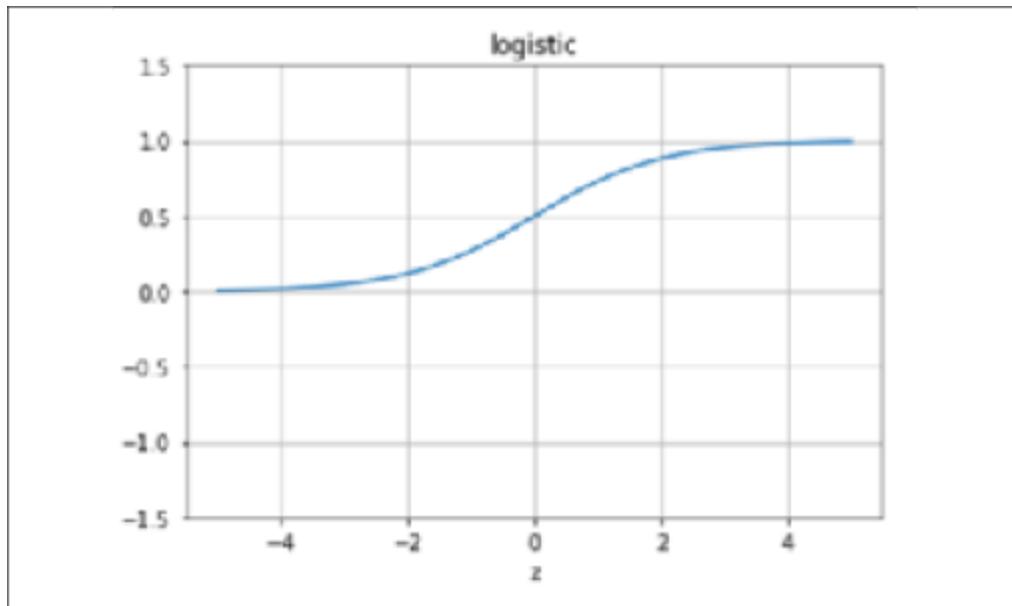


Figure 8.2: The logistic function

- The **tanh** function plot where the output value is in the range of (-1, 1):

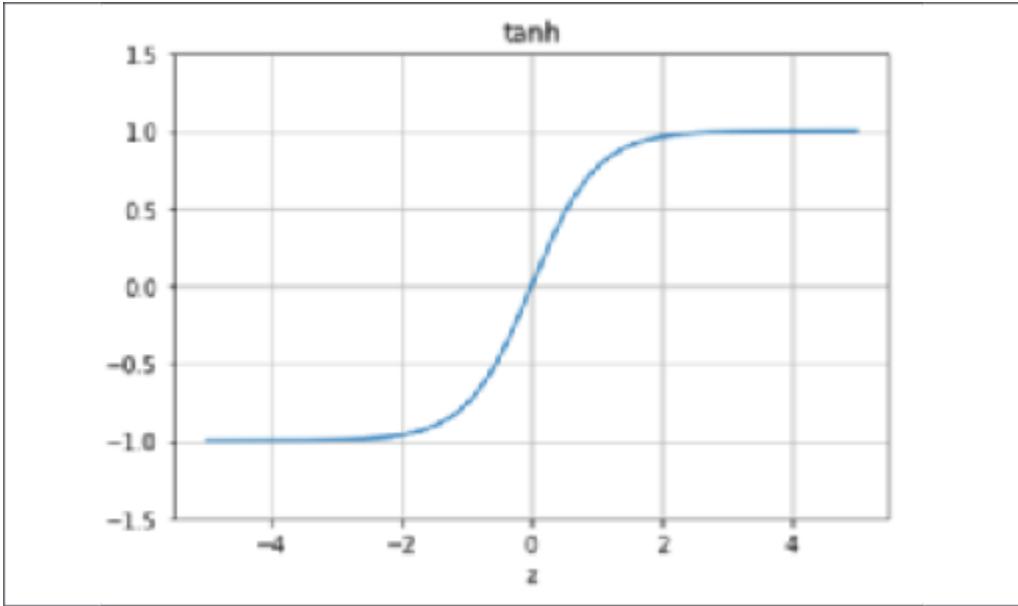


Figure 8.3: The tanh function

- The **ReLU** function plot where the output value is in the range of $(0, +\infty)$:

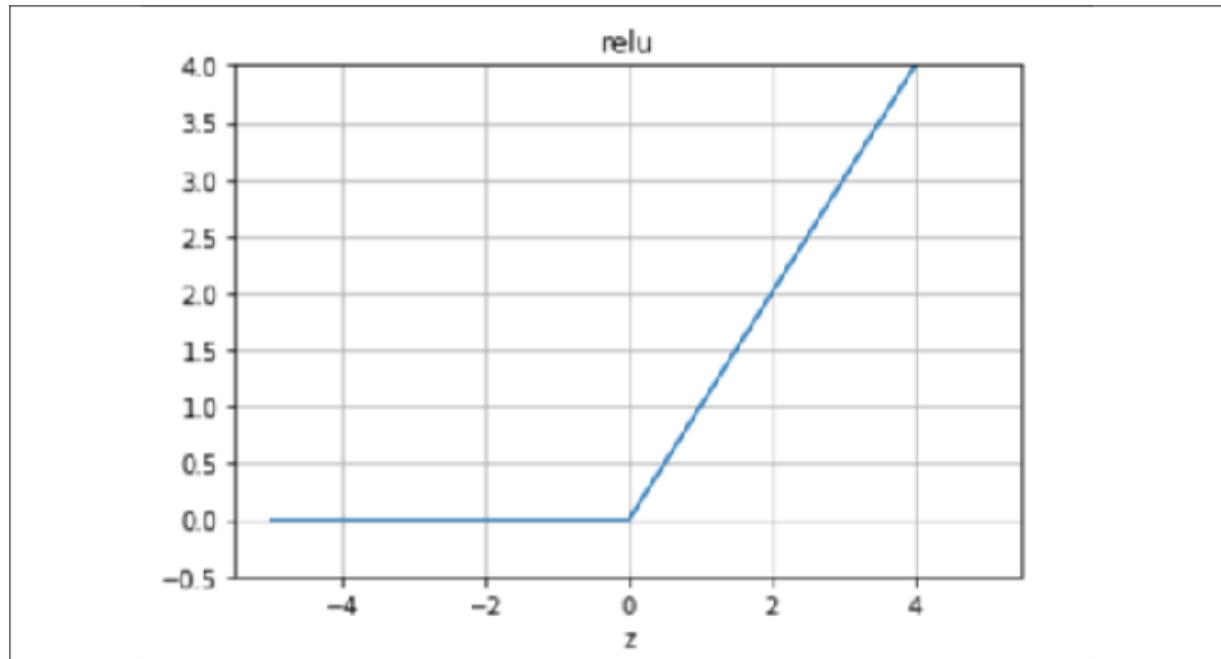


Figure 8.4: The ReLU function

As for the output layer, let's assume there is one output unit (regression or binary classification) and the weight matrix, $W^{(2)}$, connecting the hidden layer to the output layer is of size H by 1. In regression, the output can be expressed mathematically as follows (for consistency, I here denote it as $a^{(3)}$ instead of y):

$$a^{(3)} = f(z^{(3)}) = W^{(2)}a^{(2)}$$

Backpropagation

So, how can we obtain the optimal weights, $W = \{W(1), W(2)\}$, of the model? Similar to logistic regression, we can learn all weights using gradient descent with the goal of minimizing the **mean squared error (MSE)** cost, $J(W)$. The difference is that the gradients, ΔW , are computed through **backpropagation**. After each forward pass through a network, a backward pass is performed to adjust the model's parameters.

As the word *back* in the name implies, the computation of the gradient proceeds backward: the gradient of the final layer is computed first and the gradient of the first layer is computed last. As for *propagation*, it means that partial computations of the gradient on one layer are reused in the computation of the gradient on the previous layer. Error information is propagated layer by layer, instead of being calculated separately.

In a single-layer network, the detailed steps of backpropagation are as follows:

1. We travel through the network from the input to output and compute the output values, $a^{(2)}$, of the hidden layer as well as the output layer, $a^{(3)}$. This is the feedforward step.
2. For the last layer, we calculate the derivative of the cost function with regard to the input to the output layer:

$$\delta^{(3)} = \frac{\partial}{\partial z^{(3)}} J(W) = -(y - a^{(3)}) \cdot f'(z^{(3)}) = a^{(3)} - y$$

3. For the hidden layer, we compute the derivative of the cost function with regard to the input to the hidden layer:

$$\delta^{(2)} = \frac{\partial}{\partial z^{(2)}} J(W) = \frac{\partial z^{(3)}}{\partial z^{(2)}} \frac{\partial}{\partial z^{(3)}} J(W) = ((W^{(2)})\delta^{(3)}) \cdot f'(z^{(2)})$$

4. We compute the gradients by applying the **chain rule**:

$$\Delta W^{(2)} = \frac{\partial J(W)}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial W^{(2)}} = \delta^{(3)} a^{(2)}$$

$$\Delta W^{(1)} = \frac{\partial J(W)}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(1)}} = \delta^{(2)} x$$

5. We update the weights with the computed gradients and learning rate, α :

$$W^{(1)} := W^{(1)} - \frac{1}{m} \alpha \Delta W^{(1)}$$

$$W^{(2)} := W^{(2)} - \frac{1}{m} \alpha \Delta W^{(2)}$$

Here, m is the number of samples.

We repeatedly update all the weights by taking these steps with the latest weights until the cost function converges or the model goes through enough iterations.

This might not be easy to digest at first glance, so right after the next section, we will implement it from scratch, which will help you to understand neural networks better.

Adding more layers to a neural network: DL

In real applications, a neural network usually comes with multiple hidden layers. That is how DL got its name—learning using neural networks with

"stacked" hidden layers. An example of a DL model follows:

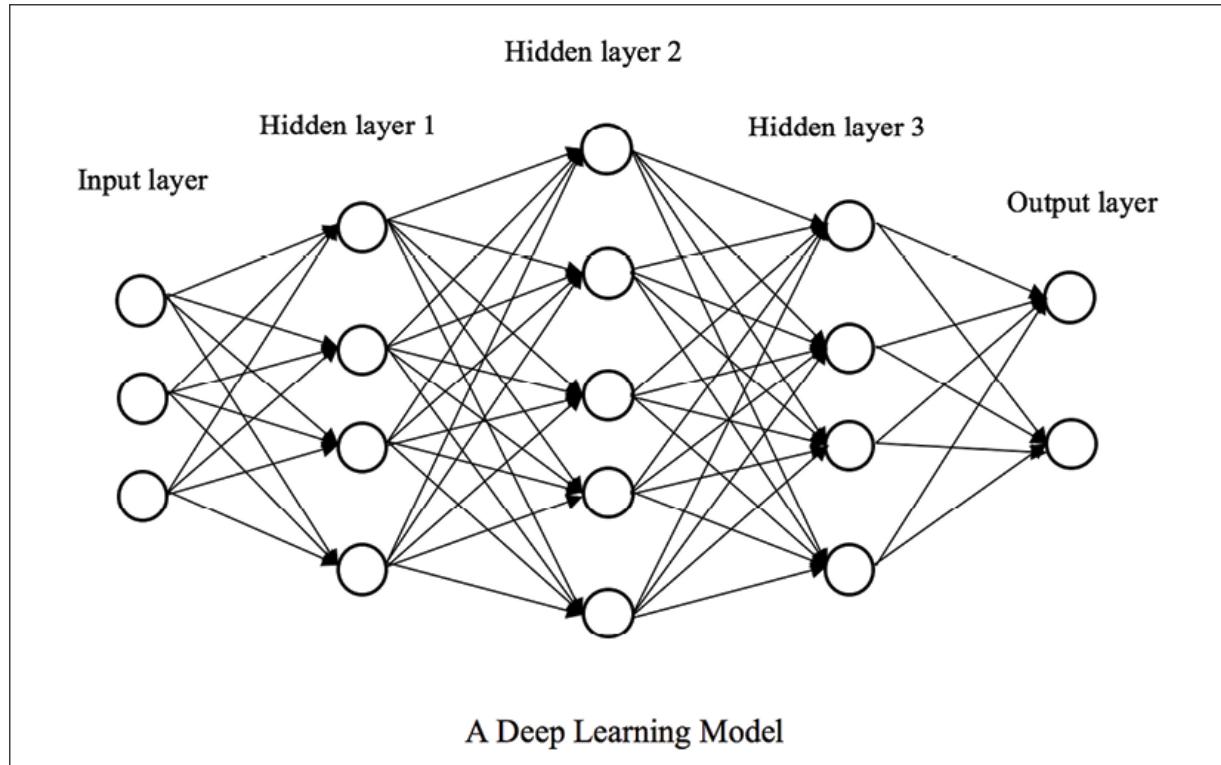


Figure 8.5: A deep neural network

In a stack of multiple hidden layers, the input of one hidden layer is the output of its previous layer, as you can see from *Figure 8.5*. Features (signals) are extracted from each hidden layer. Features from different layers represent patterns from different levels. Going beyond shallow neural networks (usually with only one hidden layer), a DL model (usually with two or more hidden layers) with the right network architectures and parameters can better learn complex non-linear relationships from data.

Let's see some typical applications of DL so that you will be more motivated to get started with upcoming DL projects.

Computer vision is widely considered the area with massive breakthroughs in DL. You will learn more about this in *Chapter 12, Categorizing Images of Clothing with Convolutional Neural Networks*. For now, here is a list of common applications in computer vision:

- Image recognition, such as face recognition and handwritten digit recognition. Handwritten digit recognition, along with the common evaluation dataset MNIST (<http://yann.lecun.com/exdb/mnist/>), has become a "Hello, World!" project in DL.
- Image-based search engines heavily utilize DL techniques in their image classification and image similarity encoding components.
- Machine vision, which is a critical part of autonomous vehicles, perceives camera views to make real-time decisions.
- Color restoration from black and white photos and art transfer that ingeniously blends two images of different styles. The artificial masterpieces in Google Arts & Culture (<https://artsandculture.google.com/>) are impressive.

Natural language processing (NLP) is another field where you can see the dominant use of DL in its modern solutions. You will learn more about this in *Chapter 13, Making Predictions with Sequences Using Recurrent Neural Networks*. But let's quickly look at some examples now:

- Machine translation, where DL has dramatically improved accuracy and fluency, for example, the sentence-based **Google Neural Machine Translation (GNMT)** system.
- Text generation, which reproduces text by learning the intricate relationships between words in sentences and paragraphs with deep neural networks. You can become a virtual J. K. Rowling or a virtual Shakespeare if you train a model right on their works.
- Image captioning, also known as image to text, leverages deep neural networks to detect and recognize objects in images, and "describe" those objects in a comprehensible sentence. It couples recent breakthroughs in computer vision and NLP. Examples can be found at <http://cs.stanford.edu/people/karpathy/deepimagegenerationdemo/> (developed by Andrej Karpathy from Stanford University).
- In other common NLP tasks such as sentiment analysis and information retrieval and extraction, DL models have achieved state-

of-the-art performance.

Similar to shallow networks, we learn all the weights in a deep neural network using gradient descent with the goal of minimizing the MSE cost, $J(W)$. And gradients, ΔW , are computed through backpropagation. The difference is that we backpropagate more than one hidden layer. In the next section, we will implement neural networks by starting with shallow networks then moving on to deep ones.

Building neural networks

This practical section will start with implementing a shallow network from scratch, followed by a deep network with two layers using scikit-learn. We will then implement a deep network with TensorFlow and Keras.

Implementing neural networks from scratch

We will use sigmoid as the activation function in this example.

We first define the `sigmoid` function and its derivative function:

```
>>> def sigmoid(z):
...     return 1.0 / (1 + np.exp(-z))
>>> def sigmoid_derivative(z):
...     return sigmoid(z) * (1.0 - sigmoid(z))
```

You can derive the derivative yourself if you are interested in verifying it.

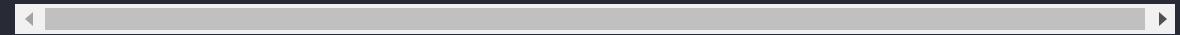
We then define the training function, which takes in the training dataset, the number of units in the hidden layer (we will only use one hidden layer as an example), and the number of iterations:

```
>>> def train(X, y, n_hidden, learning_rate, n_iter):
...     m, n_input = X.shape
```

```

...
    w1 = np.random.randn(n_input, n_hidden)
...
    b1 = np.zeros((1, n_hidden))
...
    w2 = np.random.randn(n_hidden, 1)
...
    b2 = np.zeros((1, 1))
    for i in range(1, n_iter+1):
        Z2 = np.matmul(X, w1) + b1
        A2 = sigmoid(Z2)
        Z3 = np.matmul(A2, w2) + b2
        A3 = Z3
        dZ3 = A3 - y
        dw2 = np.matmul(A2.T, dZ3)
        db2 = np.sum(dZ3, axis=0, keepdims=True)
        dZ2 = np.matmul(dZ3, w2.T) * sigmoid_derivative(Z2)
        dw1 = np.matmul(X.T, dZ2)
        db1 = np.sum(dZ2, axis=0)
        w2 = w2 - learning_rate * dw2 / m
        b2 = b2 - learning_rate * db2 / m
        w1 = w1 - learning_rate * dw1 / m
        b1 = b1 - learning_rate * db1 / m
        if i % 100 == 0:
            cost = np.mean((y - A3) ** 2)
            print('Iteration %i, training loss: %f' %
                  (i, cost))
...
    model = {'w1': w1, 'b1': b1, 'w2': w2, 'b2': b2}
...
    return model

```



Note that besides weights, W , we also employ bias, b . Before training, we first randomly initialize weights and biases. In each iteration, we feed all layers of the network with the latest weights and biases, then calculate the gradients using the backpropagation algorithm, and finally update the weights and biases with the resulting gradients. For training performance inspection, we print out the loss and the MSE for every 100 iterations.

To test the model, we will use Boston house prices as the toy dataset. As a reminder, data normalization is usually recommended whenever gradient descent is used. Hence, we will standardize the input data by removing the mean and scaling to unit variance:

```

>>> boston = datasets.load_boston()
>>> num_test = 10 # the last 10 samples as testing set
>>> from sklearn import preprocessing
>>> scaler = preprocessing.StandardScaler()

```

```
>>> X_train = boston.data[:-num_test, :]
>>> X_train = scaler.fit_transform(X_train)
>>> y_train = boston.target[:-num_test].reshape(-1, 1)
>>> X_test = boston.data[-num_test:, :]
>>> X_test = scaler.transform(X_test)
>>> y_test = boston.target[-num_test:]
```

With the scaled dataset, we can now train a one-layer neural network with 20 hidden units, a 0.1 learning rate, and 2000 iterations:

```
>>> n_hidden = 20
>>> learning_rate = 0.1
>>> n_iter = 2000
>>> model = train(X_train, y_train, n_hidden, learning_rate, n_iter)
Iteration 100, training loss: 13.500649
Iteration 200, training loss: 9.721267
Iteration 300, training loss: 8.309366
Iteration 400, training loss: 7.417523
Iteration 500, training loss: 6.720618
Iteration 600, training loss: 6.172355
Iteration 700, training loss: 5.748484
Iteration 800, training loss: 5.397459
Iteration 900, training loss: 5.069072
Iteration 1000, training loss: 4.787303
Iteration 1100, training loss: 4.544623
Iteration 1200, training loss: 4.330923
Iteration 1300, training loss: 4.141120
Iteration 1400, training loss: 3.970357
Iteration 1500, training loss: 3.814482
Iteration 1600, training loss: 3.673037
Iteration 1700, training loss: 3.547397
Iteration 1800, training loss: 3.437391
Iteration 1900, training loss: 3.341110
Iteration 2000, training loss: 3.255750
```

Then, we define a prediction function, which will take in a model and produce the regression results:

```
>>> def predict(x, model):
...     w1 = model['W1']
...     b1 = model['b1']
...     w2 = model['W2']
```

```
...     b2 = model['b2']
...     A2 = sigmoid(np.matmul(x, w1) + b1)
...     A3 = np.matmul(A2, w2) + b2
...     return A3
```

Finally, we apply the trained model on the testing set:

```
>>> predictions = predict(x_test, model)
```

Print out the predictions and their ground truths to compare them:

```
>>> print(predictions)
[[16.28103034]
 [19.98591039]
 [22.17811179]
 [19.37515137]
 [20.5675095 ]
 [24.90457042]
 [22.92777643]
 [26.03651277]
 [25.35493394]
 [23.38112184]]
>>> print(y_test)
[19.7 18.3 21.2 17.5 16.8 22.4 20.6 23.9 22. 11.9]
```

After successfully building a neural network model from scratch, we will move on to the implementation with `scikit-learn`.

Implementing neural networks with scikit-learn

We will utilize the `MLPRegressor` class (**MLP** stands for **multi-layer perceptron**, a nickname for neural networks):

```
...           learning_rate_init=0.001,
...           random_state=42, max_iter=2000)
```

The `hidden_layer_sizes` hyperparameter represents the number of hidden neurons. In this example, the network contains two hidden layers with `16` and `8` nodes, respectively. ReLU activation is used.

We fit the neural network model on the training set and predict on the testing data:

```
>>> nn_scikit.fit(X_train, y_train)
>>> predictions = nn_scikit.predict(X_test)
>>> print(predictions)
[16.79582331 18.55538023 21.07961496 19.21362606 18.50955771 23.
```

And we calculate the MSE on the prediction:

```
>>> print(np.mean((y_test - predictions) ** 2))
13.933482332708781
```

We've implemented a neural network with scikit-learn. Let's do so with TensorFlow in the next section.

Implementing neural networks with TensorFlow

In the industry, neural networks are often implemented with TensorFlow. Other popular DL (multilayer neural network) frameworks include PyTorch (<https://pytorch.org/>), which we will use in *Chapter 14, Making Decisions in Complex Environments with Reinforcement Learning*, and Keras (<https://keras.io/>), which is already included in TensorFlow 2.x. Now let's implement neural networks with TensorFlow by following these steps:

1. First, we import the necessary modules and set a random seed, which is recommended for reproducible modeling:

```
>>> import tensorflow as tf  
>>> from tensorflow import keras  
>>> tf.random.set_seed(42)
```

2. Next, we create a Keras Sequential model by passing a list of layer instances to the constructor, including two fully connected hidden layers with 20 nodes and 8 nodes, respectively. And again, ReLU activation is used:

```
>>> model = keras.Sequential([  
...     keras.layers.Dense(units=20, activation='relu'),  
...     keras.layers.Dense(units=8, activation='relu'),  
...     keras.layers.Dense(units=1)  
... ])
```

3. And we compile the model by using Adam as the optimizer with a learning rate of 0.02 and MSE as the learning goal:

```
>>> model.compile(loss='mean_squared_error',  
...                 optimizer=tf.keras.optimizers.Adam(0.02))
```

The Adam optimizer is a replacement for the stochastic gradient descent algorithm. It updates the gradients adaptively based on training data. For more information about Adam, check out the paper at <https://arxiv.org/abs/1412.6980>.

4. After defining the model, we now train it against the training set:

```
>>> model.fit(X_train, y_train, epochs=300)  
Train on 496 samples  
Epoch 1/300  
496/496 [=====] - 1s 2ms/sample -  
Epoch 2/300  
496/496 [=====] - 0s 76us/sample -  
Epoch 3/300  
496/496 [=====] - 0s 62us/sample -  
.....  
.....  
Epoch 298/300  
496/496 [=====] - 0s 60us/sample -  
Epoch 299/300
```

```
496/496 [=====] - 0s 60us/sample -
Epoch 300/300
496/496 [=====] - 0s 56us/sample -
```

We fit the model with 300 iterations. In each iteration, the training loss (MSE) is displayed.

- Finally, we use the trained model to predict the testing cases and print out the predictions and their MSE:

```
>>> predictions = model.predict(X_test)[:, 0]
>>> print(predictions)
[18.078342 17.279167 19.802671 17.54534  16.193192 24.76933
>>> print(np.mean((y_test - predictions) ** 2))
15.72498178190508
```

As you can see, we add layer by layer to the neural network model in the TensorFlow Keras API. We start from the first hidden layer (with 20 nodes), then the second hidden layer (with eight nodes), and finally the output layer (with one unit, the target variable). It is quite similar to building LEGOs. Next, we will look at how to choose the right activation functions.

Picking the right activation functions

So far, we have used the ReLU and sigmoid activation functions in our implementations. You may wonder how to pick the right activation function for your neural networks. Detailed answers to when to choose a particular activation function are given next:

- **Linear:** $f(z) = z$. You can interpret this as no activation function. We usually use it in the output layer in regression networks as we don't need any transformation to the outputs.
- **Sigmoid (logistic)** transforms the output of a layer to a range between 0 and 1. You can interpret it as the probability of an output prediction.

Therefore, we usually use it in the output layer in **binary classification** networks. Besides that, we sometimes use it in hidden layers. However, it should be noted that the sigmoid function is monotonic but its derivative is not. Hence, the neural network may get stuck at a suboptimal solution.

- **Softmax.** As was mentioned in *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*, softmax is a generalized logistic function used for multiclass classification. Hence, we use it in the output layer in **multiclass classification** networks.
- **Tanh** is a better version of the sigmoid function with stronger gradients. As you can see in the plots, the derivatives in the tanh function are steeper than those for the sigmoid function. It has a range of -1 to 1. It is common to use the tanh function in hidden layers.
- **ReLU** is probably the most frequently used activation function nowadays. It is the "default" one in hidden layers in feedforward networks. Its range is from 0 to infinity, and both the function itself and its derivative are monotonic. One drawback of the ReLU function is the inability to appropriately map the negative part of the input where all negative inputs are transformed to zero. To fix the "dying negative" problem in ReLU, **Leaky ReLU** was invented to introduce a small slope in the negative part. When $z < 0$, $f(z) = az$, where a is usually a small value, such as 0.01.

To recap, ReLU is usually in hidden layer activation. You can try Leaky ReLU if ReLU doesn't work well. Sigmoid and tanh can be used in hidden layers but are not recommended in deep networks with many layers. For the output layer, linear activation (or no activation) is used in the regression network; sigmoid is for the binary classification network and softmax is for the multiple classification case.

Picking the right activation is important, and so is avoiding overfitting in neural networks. Let's see how to do this in the next section.

Preventing overfitting in neural networks

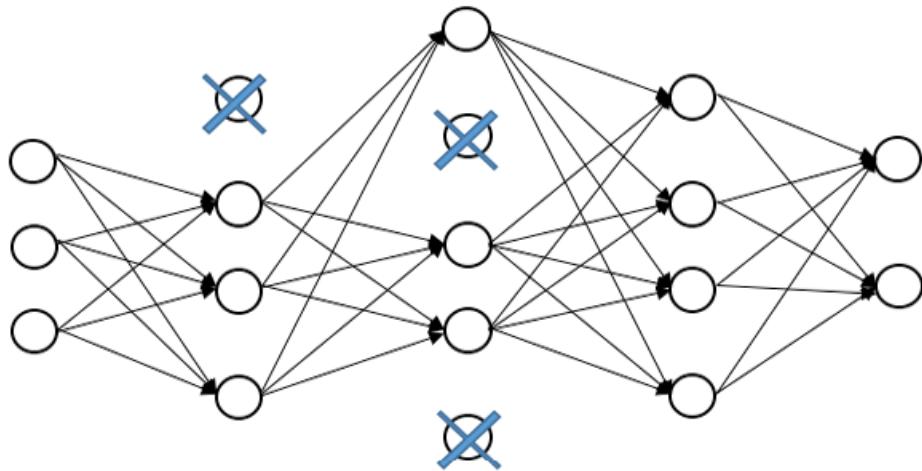
A neural network is powerful as it can derive hierarchical features from data with the right architecture (the right number of hidden layers and hidden nodes). It offers a great deal of flexibility and can fit a complex dataset. However, this advantage will become a weakness if the network is not given enough control over the learning process. Specifically, it may lead to overfitting if a network is only good at fitting to the training set but is not able to generalize to unseen data. Hence, preventing overfitting is essential to the success of a neural network model.

There are mainly three ways to impose restrictions on our neural networks: L1/L2 regularization, dropout, and early stopping. We practiced the first method in *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*, and will discuss another two in this section.

Dropout

Dropout means ignoring a certain set of hidden nodes during the learning phase of a neural network. And those hidden nodes are chosen randomly given a specified probability. In the forward pass during a training iteration, the randomly selected nodes are temporarily not used in calculating the loss; in the backward pass, the randomly selected nodes are not updated temporarily.

In the following diagram, we choose three nodes in the network to ignore during training:



A neural network with dropout

Figure 8.6: Three nodes to ignore

Recall that a regular layer has nodes fully connected to nodes from the previous layer and the following layer. It will lead to overfitting if a large network develops and memorizes the co-dependency between individual pairs of nodes. Dropout breaks this co-dependency by temporarily deactivating certain nodes in each iteration. Therefore, it effectively reduces overfitting and won't disrupt learning at the same time.

The fraction of nodes being randomly chosen in each iteration is also called the dropout rate. In practice, we usually set a dropout rate no greater than 50%. In TensorFlow, we use the `tf.keras.layers.Dropout` module to add dropout to a layer. An example is as follows:

```
>>> model = keras.Sequential([
...     keras.layers.Dense(units=32, activation='relu'),
...     tf.keras.layers.Dropout(0.5)
...     keras.layers.Dense(units=1)
```

In the preceding example, 50% of nodes randomly picked from the 16-node layer are ignored in an iteration during training.





Keep in mind that dropout only occurs in the training phase. In the prediction phase, all nodes are fully connected again.

Early stopping

As the name implies, training a network with **early stopping** will end if the model performance doesn't improve for a certain number of iterations. The model performance is measured on a validation set that is different from the training set, in order to assess how well it generalizes. During training, if the performance degrades after several (let's say 50) iterations, it means the model is overfitting and not able to generalize well anymore. Hence, stopping the learning early in this case helps prevent overfitting.

In TensorFlow, we use the `tf.keras.callbacks.EarlyStopping` module to incorporate early stopping. I will demonstrate how to use it later in this chapter.

Now that you've learned about neural networks and their implementation, let's utilize them to solve our stock price prediction problem.

Predicting stock prices with neural networks

We will build the stock predictor with TensorFlow in this section. We will start with feature generation and data preparation, followed by network building and training. After that, we will fine-tune the network and incorporate early stopping to boost the stock predictor.

Training a simple neural network

We prepare data and train a simple neural work with the following steps:

1. We load the stock data, generate features, and label the `generate_features` function we developed in *Chapter 7, Predicting Stock Prices with Regression Algorithms*:

```
>>> data_raw = pd.read_csv('19880101_20191231.csv', index_col=0)
>>> data = generate_features(data_raw)
```



2. We construct the training set using data from 1988 to 2018 and the testing set using data from 2019:

```
>>> start_train = '1988-01-01'
>>> end_train = '2018-12-31'
>>> start_test = '2019-01-01'
>>> end_test = '2019-12-31'
>>> data_train = data.loc[start_train:end_train]
>>> X_train = data_train.drop('close', axis=1).values
>>> y_train = data_train['close'].values
>>> data_test = data.loc[start_test:end_test]
>>> X_test = data_test.drop('close', axis=1).values
>>> y_test = data_test['close'].values
```

3. We need to normalize features into the same or a comparable scale. We do so by removing the mean and rescaling to unit variance:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
```

We rescale both sets with the scaler taught by the training set:

```
>>> X_scaled_train = scaler.fit_transform(X_train)
>>> X_scaled_test = scaler.transform(X_test)
```

4. We now build a neural network model using the Keras Sequential API:

```
>>> from tensorflow.keras import Sequential
>>> from tensorflow.keras.layers import Dense
>>> model = Sequential([
...     Dense(units=32, activation='relu'),
...     Dense(units=1),
... ])
```

The network we begin with has one hidden layer with 32 nodes followed by a ReLU function.

5. And we compile the model by using Adam as the optimizer with a learning rate of `0.1` and MSE as the learning goal:

```
>>> model.compile(loss='mean_squared_error',
...                  optimizer=tf.keras.optimizers.Adam(0.1))
```

6. After defining the model, we now train it against the training set:

```
>>> model.fit(X_scaled_train, y_train, epochs=100, verbose=1)
Train on 7558 samples
Epoch 1/100
7558/7558 [=====] - 1s 175us/sample
Epoch 2/100
7558/7558 [=====] - 0s 58us/sample
Epoch 3/100
7558/7558 [=====] - 0s 56us/sample
.....
.....
Epoch 98/100
7558/7558 [=====] - 0s 56us/sample
Epoch 99/100
7558/7558 [=====] - 0s 55us/sample
Epoch 100/100
7558/7558 [=====] - 0s 52us/sample
```

7. Finally, we use the trained model to predict the testing data and display metrics:

```
>>> from sklearn.metrics import mean_squared_error, mean_absolute_error
>>> print(f'MSE: {mean_squared_error(y_test, predictions):.3f}')
MSE: 43212.312
>>> print(f'MAE: {mean_absolute_error(y_test, predictions):.3f}')
MAE: 160.936
>>> print(f'R^2: {r2_score(y_test, predictions):.3f}')
R^2: 0.962
```

We achieve `0.962 R 2` with a simple neural network model.

Fine-tuning the neural network

Can we do better? Of course, we haven't fine-tuned the hyperparameters yet. We perform model fine-tuning in TensorFlow with the following steps:

1. We rely on the `hparams` module in TensorFlow, so we import it first:

```
>>> from tensorflow.plugins.hparams import api as hp
```

2. We want to tweak the number of hidden nodes in the hidden layer (again, we are using one hidden layer for this example), the number of training iterations, and the learning rate. We pick the following values of hyperparameters to experiment on:

```
>>> HP_HIDDEN = hp.HParam('hidden_size', hp.Discrete([64, 32, 16]))
>>> HP_EPOCHS = hp.HParam('epochs', hp.Discrete([300, 1000]))
>>> HP_LEARNING_RATE = hp.HParam('learning_rate', hp.RealIn-
```

Here, we experiment with three options for the number of hidden nodes (discrete value), `16`, `32`, and `64`; we use two options for the number of iterations (discrete value), `300` and `1000`; and we use the range of `0.01` to `4` for the learning rate (continuous value).

3. After initializing the hyperparameters to optimize, we now create a function to train and validate the model that will take the hyperparameters as arguments, and output the performance:

```
>>> def train_test_model(hparams, logdir):
...     model = Sequential([
...         Dense(units=hparams[HP_HIDDEN], activation='relu'),
...         Dense(units=1)
...     ])
...     model.compile(loss='mean_squared_error',
...                   optimizer=tf.keras.optimizers.Adam(
...                     hparams[HP_LEARNING_RATE]),
...                   metrics=['mean_squared_error'])
...     model.fit(X_scaled_train, y_train,
...               validation_data=(X_scaled_test, y_test),
...               epochs=hparams[HP_EPOCHS], verbose=False,
...               callbacks=[
...                 tf.keras.callbacks.TensorBoard(logdir),
...                 hp.KerasCallback(logdir, hparams),
...                 tf.keras.callbacks.EarlyStopping(
...                   monitor='val_loss', min_delta=0,
...                   patience=200, verbose=0, mode='au-
```

```
...         ],
...
...     )
...     _, mse = model.evaluate(X_scaled_test, y_test)
...     pred = model.predict(X_scaled_test)
...     r2 = r2_score(y_test, pred)
...     return mse, r2
```

Here, we build, compile, and fit a neural network model based on the given hyperparameters, including the number of hidden nodes, the learning rate, and the number of training iterations. There's nothing much different here from what we did before. But when we train the model, we also run several callback functions, including updating TensorBoard using `tf.keras.callbacks.TensorBoard(logdir)`, logging hyperparameters and metrics using `hp.KerasCallback(logdir, hparams)`, and early stopping using `tf.keras.callbacks.EarlyStopping(...)`.

The TensorBoard callback function is straightforward. It provides visualization for the model graph and metrics during training and validation.

The hyperparameters logging callback logs the hyperparameters and metrics.

The early stopping callback monitors the performance on the validation set, which is the testing set in our case. If the MSE doesn't decrease after 200 epochs, it stops the training process.

At the end of this function, we output the MSE and R^2 of the prediction on the testing set.

4. Next, we develop a function to initiate a training process with a combination of hyperparameters to be assessed and to write a summary with the metrics for MSE and R^2 returned by the `train_test_model` function:

```
>>> def run(hparams, logdir):
...     with tf.summary.create_file_writer(logdir).as_default():
...         hp.hparams_config(
...             hparams=[HP_HIDDEN, HP_EPOCHS, HP_LEARNING_RATE],
...             metrics=[hp.Metric('mean_squared_error',
...                               display_name='mse'),
...                      hp.Metric('r2', display_name='r2')])
... 
```

```
...         mse, r2 = train_test_model(hparams, logdir)
...         tf.summary.scalar('mean_squared_error', mse, step=step)
...         tf.summary.scalar('r2', r2, step=step)
```

5. We now train the model for each different combination of the hyperparameters in a gridsearch manner:

```
>>> for hidden in HP_HIDDEN.domain.values:
...     for epochs in HP_EPOCHS.domain.values:
...         for learning_rate in
...             tf.linspace(HP_LEARNING_RATE.domain.min_value
...                         HP_LEARNING_RATE.domain.max_value
...         hparams = {
...             HP_HIDDEN: hidden,
...             HP_EPOCHS: epochs,
...             HP_LEARNING_RATE:
...                 float("%.2f"%float(learning_rate))
...         }
...         run_name = "run-%d" % session_num
...         print('--- Starting trial: %s' % run_name)
...         print({h.name: hparams[h] for h in hparams})
...         run(hparams, 'logs/hparam_tuning/' + run_name)
...         session_num += 1
```

For each experiment, a discrete value (the number of hidden nodes and iterations) is picked from the predefined value pool and a continuous value (the learning rate) is chosen from five evenly spaced values over the interval (from `0.01` to `0.4`). It will take a few minutes to run these experiments. You will see the following output:

```
--- Starting trial: run-0
{'hidden_size': 16, 'epochs': 300, 'learning_rate': 0.01}
2020-04-29 08:06:43.149021: I tensorflow/core/profiler/lib/
.....
=====] - 0s 42us
.....
.....
.....
--- Starting trial: run-29
{'hidden_size': 64, 'epochs': 1000, 'learning_rate': 0.4}
2020-04-29 08:28:03.036671: I tensorflow/core/profiler/lib/
.....
=====] - 0s 54us
```

6. You will notice that a new folder, logs, is created after the experiments start. It contains the training and validation performance for each experiment. After 30 experiments finish, it's time to launch TensorBoard. We use the following command:

```
tensorboard --logdir ls/hparam_tuning
Serving TensorBoard on localhost; to expose to the network,
TensorBoard 2.0.0 at http://localhost:6006/ (Press CTRL+C to stop)
```

Once it is launched, you will see the beautiful dashboard at <http://localhost:6006/>. See the screenshot of the expected result here:

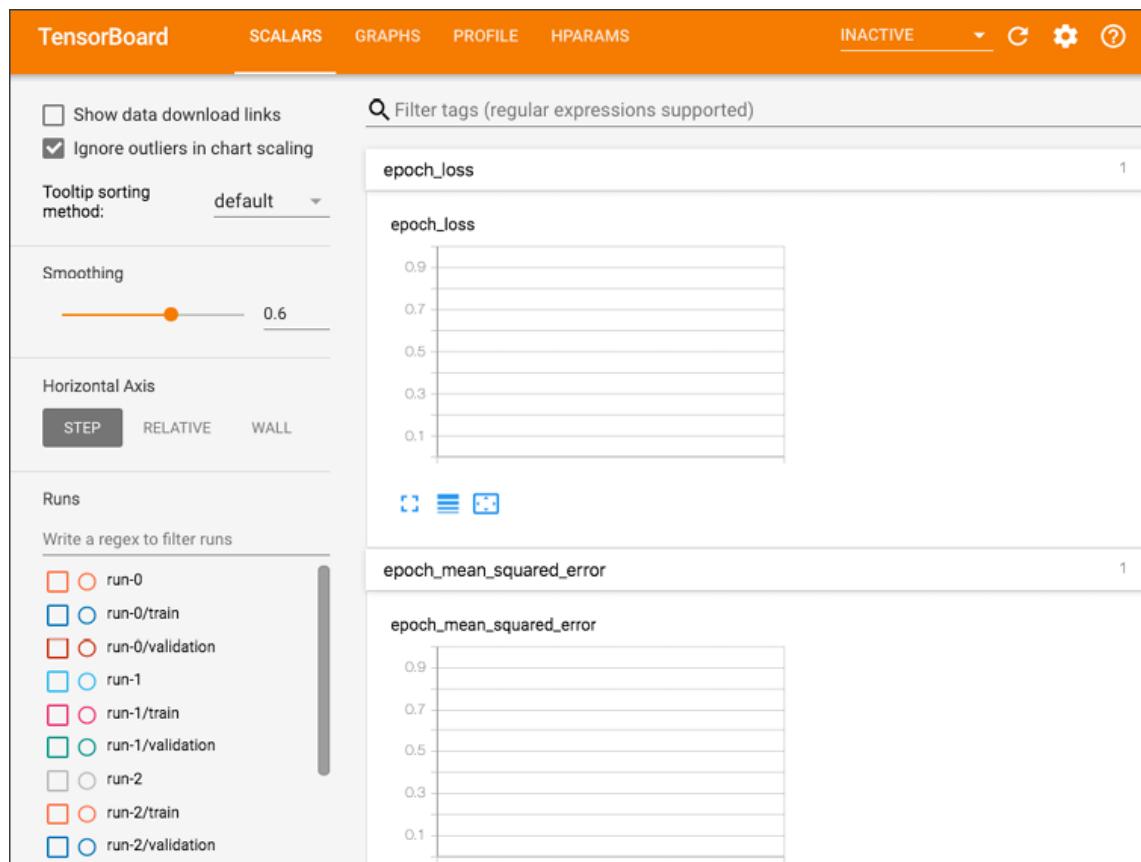


Figure 8.7: Screenshot of TensorBoard

Click on the **HPARAMS** tab to see the hyperparameter logs. You can see all the hyperparameter combinations and the respective metrics (MSE and R^2) displayed in a table, as shown here:

TensorBoard		SCALARS	GRAPHS	PROFILE	HPARAMS	INACTIVE	
Hyperparameters		TABLE VIEW		PARALLEL COORDINATES VIEW			SCATTER PLOT MATRIX VIEW
Trial ID	Show Metrics	hidden_size	epochs	learning_rate	mse	r2	
040324d52ab617...	<input type="checkbox"/>	64.000	300.00	0.21000	36284	0.96836	
0979d01d62d519...	<input type="checkbox"/>	16.000	300.00	0.40000	36948	0.96778	
1a60de513b2319...	<input type="checkbox"/>	32.000	300.00	0.21000	34755	0.96969	
485f4ee17ecb588...	<input type="checkbox"/>	64.000	1000.0	0.11000	95550	0.91668	
6552154cd58ee14...	<input type="checkbox"/>	32.000	1000.0	0.21000	40258	0.96489	
6cb44a3856f106e...	<input type="checkbox"/>	32.000	1000.0	0.30000	61809	0.94610	
7040706da3e91e...	<input type="checkbox"/>	16.000	300.00	0.30000	38412	0.96650	
75440855f618f11...	<input type="checkbox"/>	16.000	1000.0	0.21000	33007	0.97122	
795049416b5fa81...	<input type="checkbox"/>	64.000	1000.0	0.40000	59099	0.94846	
7b5f9905ea11db8...	<input type="checkbox"/>	16.000	1000.0	0.010000	34735	0.96971	
80a1dbb4fb82af9...	<input type="checkbox"/>	64.000	1000.0	0.30000	65738	0.94267	
82fecc9de850858...	<input type="checkbox"/>	32.000	300.00	0.40000	35779	0.96880	
8791a9be31c112...	<input type="checkbox"/>	16.000	1000.0	0.40000	58809	0.94872	
b4385335083a2f...	<input type="checkbox"/>	16.000	300.00	0.11000	45370	0.96044	
b2d7ffaae826dc9...	<input type="checkbox"/>	64.000	300.00	0.40000	37284	0.96749	
8f89d44a41663e3...	<input type="checkbox"/>	16.000	1000.0	0.11000	84439	0.92637	
982dce79d07f59d...	<input type="checkbox"/>	32.000	1000.0	0.11000	66802	0.94175	
a13fd6949f5727b...	<input type="checkbox"/>	64.000	300.00	0.30000	55400	0.95169	
a5b9a7e44d9012...	<input type="checkbox"/>	32.000	1000.0	0.010000	38066	0.96680	
bf45446cad23c5e...	<input type="checkbox"/>	16.000	1000.0	0.30000	42437	0.96299	
d28a4bbb534c83f...	<input type="checkbox"/>	16.000	300.00	0.21000	36929	0.96780	
d4a2e530db443d...	<input type="checkbox"/>	16.000	300.00	0.010000	55866	0.95128	
d8b1aea9727190...	<input type="checkbox"/>	32.000	300.00	0.30000	34349	0.97005	
df1fd0e02334141...	<input type="checkbox"/>	64.000	1000.0	0.010000	45241	0.96055	

Figure 8.8: Screenshot of TensorBoard for hyperparameter tuning

The combination of (**hidden_size=16**, **epochs=1000**, **learning_rate=0.21**) is the best performing one, with which we achieve an R^2 of **0.97122**.

7. Finally, we use the optimal model to make predictions:

```
>>> model = Sequential([
...     Dense(units=16, activation='relu'),
...     Dense(units=1)
... ])
>>> model.compile(loss='mean_squared_error',
...                  optimizer=tf.keras.optimizers.Adam(0.21))
>>> model.fit(X_scaled_train, y_train, epochs=1000, verbose=0)
>>> predictions = model.predict(X_scaled_test)[:, 0]
```

8. Plot the prediction along with the ground truth as follows:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(data_test.index, y_test, c='k')
>>> plt.plot(data_test.index, predictions, c='b')
>>> plt.plot(data_test.index, predictions, c='r')
>>> plt.plot(data_test.index, predictions, c='g')
>>> plt.xticks(range(0, 252, 10), rotation=60)
>>> plt.xlabel('Date')
>>> plt.ylabel('Close price')
>>> plt.legend(['Truth', 'Neural network prediction'])
>>> plt.show()
```

Refer to the following screenshot for the end result:

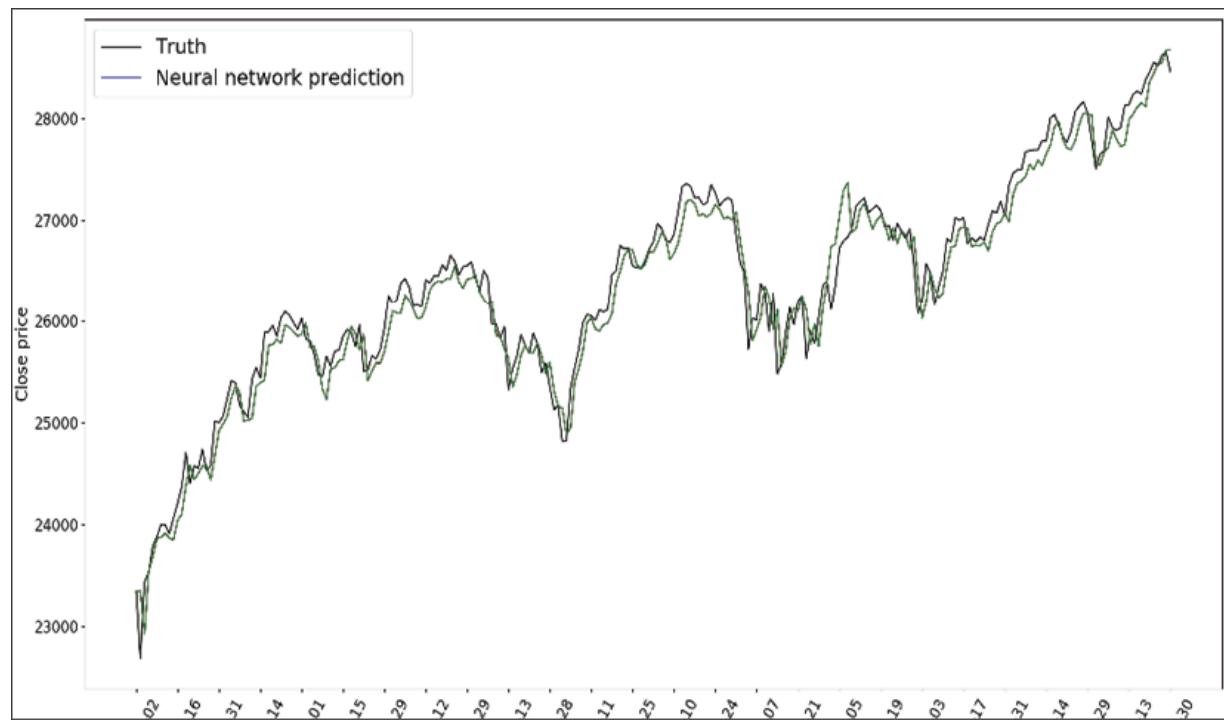


Figure 8.9: Prediction and ground truth of stock prices

The fine-tuned neural network does a good job of predicting stock prices.

In this section, we further improved the neural network stock predictor by utilizing the `hparams` module in TensorFlow. Feel free to use more hidden layers and re-run model fine-tuning to see whether you can get a better result.

Summary

In this chapter, we worked on the stock prediction project again, but with neural networks this time. We started with a detailed explanation of neural networks, including the essential components (layers, activations, feedforward, and backpropagation), and transitioned to DL. We moved on to implementations from scratch with scikit-learn and TensorFlow. You also learned about ways to avoid overfitting, such as dropout and early stopping. Finally, we applied what we covered in this chapter to solve our stock price prediction problem.

In the next chapter, we will explore NLP techniques and unsupervised learning.

Exercise

1. As mentioned, can you use more hidden layers in the neural network stock predictor and re-run the model fine-tuning? Can you get a better result, maybe using dropout and early stopping?

Mining the 20 Newsgroups Dataset with Text Analysis Techniques

In previous chapters, we went through a bunch of fundamental machine learning concepts and supervised learning algorithms. Starting from this chapter, as the second step of our learning journey, we will be covering in detail several important unsupervised learning algorithms and techniques. To make our journey more interesting, we will start with a **natural language processing (NLP)** problem—exploring newsgroups data. You will gain hands-on experience in working with text data, especially how to convert words and phrases into machine-readable values and how to clean up words with little meaning. We will also visualize text data by mapping it into a two-dimensional space in an unsupervised learning manner.

We will go into detail on each of the following topics:

- NLP fundamentals and applications
- Touring Python NLP libraries
- Tokenization, stemming, and lemmatization
- Getting and exploring the newsgroups data
- Data visualization using `seaborn` and `matplotlib`
- The **Bag of words (BoW)** model
- Text preprocessing
- Dimensionality reduction
- T-SNE and T-SNE for text visualization

How computers understand language – NLP

In *Chapter 1, Getting Started with Machine Learning and Python*, I mentioned that machine learning-driven programs or computers are good at discovering event patterns by processing and working with data. When the data is well structured or well defined, such

as in a Microsoft Excel spreadsheet table or a relational database table, it is intuitively obvious why machine learning is better at dealing with it than humans. Computers read such data the same way as humans, for example, `revenue: 5,000,000` as the revenue being 5 million and `age: 30` as the age being 30; then computers crunch assorted data and generate insights in a faster way than humans. However, when the data is unstructured, such as words with which humans communicate, news articles, or someone's speech in French, it seems that computers cannot understand words as well as humans do (yet).

What is NLP?

There is a lot of information in the world about words or raw text, or, broadly speaking, **natural language**. This refers to any language that humans use to communicate with each other. Natural language can take various forms, including, but not limited to, the following:

- Text, such as a web page, SMS, email, and menus
- Audio, such as speech and commands to Siri
- Signs and gestures
- Many other forms, such as songs, sheet music, and Morse code

The list is endless, and we are all surrounded by natural language all of the time (that's right, right now as you are reading this book). Given the importance of this type of unstructured data, natural language data, we must have methods to get computers to understand and reason with natural language and to extract data from it. Programs equipped with NLP throughout techniques can already do a lot in certain areas, which already seems magical!

NLP is a significant subfield of machine learning that deals with the interactions between machines (computers) and human (natural) languages. The data for NLP tasks can be in different forms, for example, text from social media posts, web pages, even medical prescriptions, or audio from voice mails, commands to control systems, or even a favorite song or movie. Nowadays, NLP is broadly involved in our daily lives: we cannot live without machine translation; weather forecast scripts are automatically generated; we find voice search convenient; we get the answer to a question (such as what is the population of Canada) quickly thanks to intelligent question-answering systems; speech-to-text technology helps people with special needs.

The history of NLP

If machines are able to understand language like humans do, we consider them intelligent. In 1950, the famous mathematician Alan Turing proposed in an article, *Computing Machinery and Intelligence*, a test as a criterion of machine intelligence. It's now called the **Turing test** (<https://plato.stanford.edu/entries/turing-test/>), and its goal is to examine whether a computer is able to adequately understand languages so as to fool humans into thinking that this machine is another human. It is probably no surprise to you that no computer has passed the Turing test yet, but the 1950s is considered to be when the history of NLP started.

Understanding language might be difficult, but would it be easier to automatically translate texts from one language to another? In my first ever programming course, the lab booklet had the algorithm for coarse-grained machine translation. This type of translation involves looking up in dictionaries and generating text in a new language. A more practically feasible approach would be to gather texts that are already translated by humans and train a computer program on these texts. In 1954, in the Georgetown–IBM experiment

(https://en.wikipedia.org/wiki/Georgetown%E2%80%93IBM_experiment), scientists claimed that machine translation would be solved in three to five years. Unfortunately, a machine translation system that can beat human expert translators does not exist yet. But machine translation has been greatly evolving since the introduction of deep learning and has seen incredible achievements in certain areas, for example, social media (Facebook open sourced a neural machine translation system, <https://ai.facebook.com/tools/translate/>), real-time conversation (Skype, SwiftKey Keyboard, and Google Pixel Buds), and image-based translation, such as Google Translate.

Conversational agents, or chatbots, are another hot topic in NLP. The fact that computers are able to have a conversation with us has reshaped the way businesses are run. In 2016, **Microsoft's AI chatbot, Tay** (<https://blogs.microsoft.com/blog/2016/03/25/learning-tays-introduction/>), was unleashed to mimic a teenage girl and converse with users on Twitter in real time. She learned how to speak from all the things users posted and commented on Twitter. However, she was overwhelmed by tweets from trolls, and automatically learned their bad behaviors and started to output inappropriate things on her feeds. She ended up being terminated within 24 hours.

NLP applications

There are also several tasks that attempt to organize knowledge and concepts in such a way that they become easier for computer programs to manipulate. The way we organize and represent concepts is called **ontology**. An ontology defines concepts and relationships between concepts. For instance, we can have a so-called triple, such as ("python",

"language", "is-a") representing the relationship between two concepts, such as *Python is a language*.

An important use case for NLP at a much lower level, compared to the previous cases, is **part-of-speech (PoS) tagging**. A PoS is a grammatical word category such as a noun or verb. PoS tagging tries to determine the appropriate tag for each word in a sentence or a larger document. The following table gives examples of English PoS:

Part of speech	Examples
Noun	David, machine
Pronoun	They, her
Adjective	Awesome, amazing
Verb	Read, write
Adverb	Very, quite
Preposition	Out, at
Conjunction	And, but
Interjection	Phew, oops
Article	A, the

Table 9.1: PoS examples

There are a variety of real-world NLP applications involving supervised learning, such as PoS tagging mentioned earlier. A typical example is identifying news sentiment, which could be positive or negative in the binary case, or positive, neutral, or negative in multiclass classification. News sentiment analysis provides a significant signal to trading in the stock market.

Another example we can easily think of is news topic classification, where classes may or may not be mutually exclusive. In the newsgroup example that we just discussed, classes are mutually exclusive (despite slight overlapping), such as technology, sports, and religion. It is, however, good to realize that a news article can be occasionally assigned

multiple categories (multi-label classification). For example, an article about the Olympic Games may be labeled sports and politics if there is unexpected political involvement.

Finally, an interesting application that is perhaps unexpected is **named entity recognition (NER)**. Named entities are phrases of definitive categories, such as names of persons, companies, geographic locations, dates and times, quantities and monetary values. NER is an important subtask of information extraction to seek and identify such entities. For example, we can conduct NER on the following sentence: SpaceX[Organization], a California[Location]-based company founded by a famous tech entrepreneur Elon Musk[Person], announced that it would manufacture the next-generation, 9[Quantity]-meter-diameter launch vehicle and spaceship for the first orbital flight in 2020[Date].

In the next chapter, we will discuss how unsupervised learning, including clustering and topic modeling, is applied to text data. We will begin by covering NLP basics in the upcoming sections in this chapter.

Touring popular NLP libraries and picking up NLP basics

Now that we have covered a short list of real-world applications of NLP, we will be touring the essential stack of Python NLP libraries. These packages handle a wide range of NLP tasks as mentioned previously, including sentiment analysis, text classification, and NER.

Installing famous NLP libraries

The most famous NLP libraries in Python include the **Natural Language Toolkit (NLTK)**, **spaCy**, **Gensim**, and **TextBlob**. The scikit-learn library also has impressive NLP-related features. Let's take a look at them in more detail:

- `nltk`: This library (<http://www.nltk.org/>) was originally developed for educational purposes and is now being widely used in industry as well. It is said that you can't talk about NLP without mentioning NLTK. It is one of the most famous and leading platforms for building Python-based NLP applications. You can install it simply by running the following command line in the terminal:

```
sudo pip install -U nltk
```

If you're using `conda`, execute the following command line:

```
conda install nltk
```

- `spacy` : This library (<https://spacy.io/>) is a more powerful toolkit in the industry than NLTK. This is mainly for two reasons: one, `spacy` is written in Cython, which is much more memory-optimized (now you can see where the `cy` in `spacy` comes from) and excels in NLP tasks; second, `spacy` uses state-of-the-art algorithms for core NLP problems, such as **convolutional neural network (CNN)** models for tagging and NER. However, it could seem advanced for beginners. In case you're interested, here are the installation instructions.

Run the following command line in the terminal:

```
pip install -U spacy
```

For `conda`, execute the following command line:

```
conda install -c conda-forge spacy
```

- `Gensim` : This library (<https://radimrehurek.com/gensim/>), developed by Radim Rehurek, has been gaining popularity over recent years. It was initially designed in 2008 to generate a list of similar articles given an article, hence the name of this library (`generate similar` —> `Gensim`). It was later drastically improved by Radim Rehurek in terms of its efficiency and scalability. Again, you can easily install it via `pip` by running the following command line:

```
pip install --upgrade gensim
```

In the case of `conda`, you can perform the following command line in the terminal:

```
conda install -c conda-forge gensim
```



You should make sure that the dependencies, NumPy and SciPy, are already installed before `gensim`.

- `TextBlob` : This library (<https://textblob.readthedocs.io/en/dev/>) is a relatively new one built on top of NLTK. It simplifies NLP and text analysis with easy-to-use built-in functions and methods, as well as wrappers around common tasks. We can install `TextBlob` by running the following command line in the terminal:

```
pip install -U textblob
```

`TextBlob` has some useful features that are not available in NLTK (currently), such as spellchecking and correction, language detection, and translation.

Corpora

As of 2018, NLTK comes with over 100 collections of large and well-structured text datasets, which are called **corpora** in NLP. Corpora can be used as dictionaries for checking word occurrences and as training pools for model learning and validating. Some useful and interesting corpora include Web Text corpus, Twitter samples, Shakespeare corpus, Sentiment Polarity, Names corpus (it contains lists of popular names, which we will be exploring very shortly), WordNet, and the Reuters benchmark corpus. The full list can be found at http://www.nltk.org/nltk_data. Before using any of these corpus resources, we need to first download them by running the following code in the Python interpreter:

```
>>> import nltk
>>> nltk.download()
```

A new window will pop up and ask you which collections (the **Collections** tab in the following screenshot) or corpus (the **Corpora** tab in the following screenshot) to download, and where to keep the data:

The screenshot shows a software interface for managing NLTK data. At the top, there are four tabs: **Collections** (selected), **Corpora**, **Models**, and **All Packages**. Below the tabs is a table with columns: **Identifier**, **Name**, **Size**, and **Status**. The table rows are as follows:

Identifier	Name	Size	Status
all	All packages	n/a	out of date
all-corpora	All the corpora	n/a	out of date
all-nltk	All packages available on nltk_data gh-pages branch	n/a	out of date
book	Everything used in the NLTK Book	n/a	out of date
popular	Popular packages	n/a	out of date
tests	Packages for running tests	n/a	out of date
third-party	Third-party data packages	n/a	not installed

At the bottom of the window, there are two buttons: **Download** and **Refresh**. Below these buttons are two text input fields: **Server Index:** https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml and **Download Directory:** /Users/hayden/nltk_data.

Figure 9.1: Collections tab in the NLTK installation

Installing the whole popular package is the quick solution, since it contains all important corpora needed for your current study and future research. Installing a particular corpus, as shown in the following screenshot, is also fine:

Collections	Corpora	Models	All Packages
Identifier	Name	Size	Status
lin_thesaurus	Lin's Dependency Thesaurus	85.0 MB	installed
mac_morpho	MAC-MORPHO: Brazilian Portuguese news text with part-of-s	2.9 MB	installed
machado	Machado de Assis -- Obra Completa	5.9 MB	installed
masc_tagged	MASC Tagged Corpus	1.5 MB	installed
movie_reviews	Sentiment Polarity Dataset Version 2.0	3.8 MB	installed
mte_teip5	MULTEXT-East 1984 annotated corpus 4.0	14.1 MB	installed
names	Names Corpus, Version 1.3 (1994-03-29)	20.8 KB	installed
nombank.1.0	NomBank Corpus 1.0	6.4 MB	installed
nonbreaking_prefixes	Non-Breaking Prefixes (Moses Decoder)	24.8 KB	out of date
nps_chat	NPS Chat	294.3 KB	installed
omw	Open Multilingual Wordnet	11.5 MB	out of date
opinion_lexicon	Opinion Lexicon	24.4 KB	installed
panlex_swadesh	PanLex Swadesh Corpora	2.7 MB	out of date
paradigms	Paradigm Corpus	24.3 KB	installed
pe08	Cross-Framework and Cross-Domain Parser Evaluation Shared	78.8 KB	not installed
pil	The Patient Information Leaflet (PIL) Corpus	1.4 MB	installed

Server Index: https://raw.githubusercontent.com/nltk/nltk_data/gh-pages/index.xml

Download Directory: /Users/hayden/nltk_data

Figure 9.2: Corpora tab in the NLTK installation

Once the package or corpus you want to explore is installed, you can now take a look at the **Names** corpus (make sure the `names` corpus is installed).

First, import the `names` corpus:

```
>>> from nltk.corpus import names
```

We can check out the first `10` names in the list:

```
>>> print(names.words()[:10])
['Abagael', 'Abagail', 'Abbe', 'Abbey', 'Abbi', 'Abbie',
'Abby', 'Abigael', 'Abigail', 'Abigale']
```

There are, in total, `7944` names, as shown in the following output derived by executing the following command:

```
>>> print(len(names.words()))
7944
```

Other corpora are also fun to explore.

Besides the easy-to-use and abundant corpora pool, more importantly, NLTK is also good at many NLP and text analysis tasks, including tokenization, PoS tagging, NER, word stemming, and lemmatization.

Tokenization

Given a text sequence, **tokenization** is the task of breaking it into fragments, which can be words, characters, or sentences. Certain characters are usually removed, such as punctuation marks, digits, and emoticons. The remaining fragments are the so-called **tokens** used for further processing. Moreover, tokens composed of one word are also called **unigrams** in computational linguistics; **bigrams** are composed of two consecutive words; **trigrams** of three consecutive words; and **n-grams** of n consecutive words. Here is an example of tokenization:

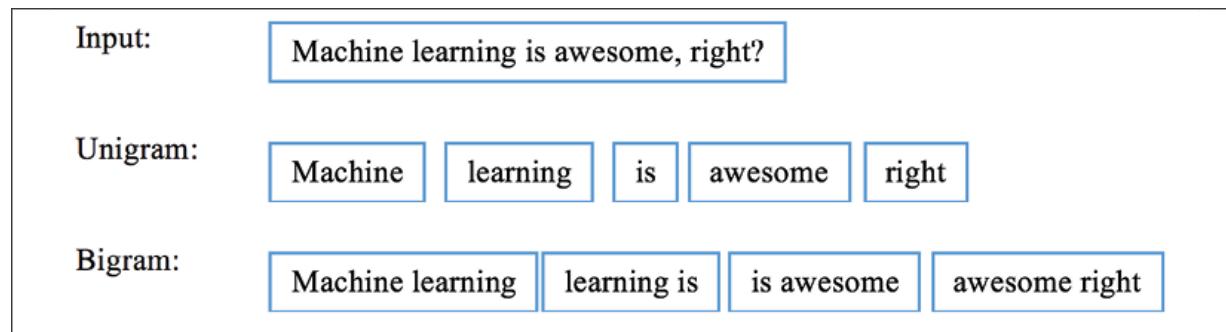


Figure 9.3: Tokenization example

We can implement word-based tokenization using the `word_tokenize` function in NLTK. We will use the input text `'''I am reading a book., It is Python Machine Learning By Example, , then 3rd edition.'''`, as an example, as shown in the following commands:

```
>>> from nltk.tokenize import word_tokenize
>>> sent = '''I am reading a book.
...           It is Python Machine Learning By Example,
...           3rd edition.'''
>>> print(word_tokenize(sent))
['I', 'am', 'reading', 'a', 'book', '.', 'It', 'is', 'Python', 'Machine', 'Le
```

Word tokens are obtained.



The `word_tokenize` function keeps punctuation marks and digits, and only discards whitespaces and newlines.

You might think word tokenization is simply splitting a sentence by space and punctuation. Here's an interesting example showing that tokenization is more complex than you think:

```
>>> sent2 = 'I have been to U.K. and U.S.A.'
>>> print(word_tokenize(sent2))
['I', 'have', 'been', 'to', 'U.K.', 'and', 'U.S.A', '.']
```

The tokenizer accurately recognizes the words 'U.K.' and 'U.S.A' as tokens instead of 'U' and '.' followed by 'K', for example.

`spacy` also has an outstanding tokenization feature. It uses an accurately trained model that is constantly updated. To install it, we can run the following command:

```
python -m spacy download en_core_web_sm
```

Then, we'll load the `en_core_web_sm` model and parse the sentence using this model:

```
>>> import spacy
>>> nlp = spacy.load('en_core_web_sm')
>>> tokens2 = nlp(sent2)
>>> print([token.text for token in tokens2])
['I', 'have', 'been', 'to', 'U.K.', 'and', 'U.S.A.']
```

We can also segment text based on sentences. For example, on the same input text, using the `sent_tokenize` function from NLTK, we have the following commands:

```
>>> from nltk.tokenize import sent_tokenize
>>> print(sent_tokenize(sent))
['I am reading a book.', 'It\'s Python Machine Learning By Example,\n            3rd edition.']
```

Two sentence-based tokens are returned, as there are two sentences in the input text.

PoS tagging

We can apply an off-the-shelf tagger from NLTK or combine multiple taggers to customize the tagging process. It is easy to directly use the built-in tagging function, `pos_tag`, as in `pos_tag(input_tokens)`, for instance. But behind the scenes, it is actually a prediction from a pre-built supervised learning model. The model is trained based on a large corpus composed of words that are correctly tagged.

Reusing an earlier example, we can perform PoS tagging as follows:

```
>>> import nltk
>>> tokens = word_tokenize(sent)
```

```
>>> print(nltk.pos_tag(tokens))
[('I', 'PRP'), ('am', 'VBP'), ('reading', 'VBG'), ('a', 'DT'), ('book', 'NN')]
```

The PoS tag following each token is returned. We can check the meaning of a tag using the `help` function. Looking up `PRP` and `VBP`, for example, gives us the following output:

```
>>> nltk.help.upenn_tagset('PRP')
PRP: pronoun, personal
    hers herself him himself hisself it itself me myself one oneself ours ours
>>> nltk.help.upenn_tagset('VBP')
VBP: verb, present tense, not 3rd person singular
    predominate wrap resort sue twist spill cure lengthen brush terminate appe
```

In `spacy`, getting a PoS tag is also easy. The `token` object parsed from an input sentence has an attribute called `pos_`, which is the tag we are looking for. Let's print the `pos_` for each token, as follows:

```
>>> print([(token.text, token.pos_) for token in tokens2])
[('I', 'PRON'), ('have', 'VERB'), ('been', 'VERB'), ('to', 'ADP'), ('U.K.', 'NOUN')]
```

We have just played around with PoS tagging with NLP packages. What about NER? Let's see in the next section.

NER

Given a text sequence, the NER task is to locate and identify words or phrases that are of definitive categories, such as names of persons, companies, locations, and dates. Let's take a peep at an example of using `spacy` for NER.

First, tokenize an input sentence, `The book written by Hayden Liu in 2020 was sold at $30 in America`, as usual, as shown in the following command:

```
>>> tokens3 = nlp('The book written by Hayden Liu in 2020 was sold at $30 in America')
```

The resultant `tokens3` object contains an attribute called `ents`, which are the named entities. We can extract the tagging for each recognized named entity as follows:

```
>>> print([(token_ent.text, token_ent.label_) for token_ent in tokens3.ents])
[('Hayden Liu', 'PERSON'), ('2018', 'DATE'), ('$30', 'MONEY'), ('America', 'GPE')]
```

We can see from the results that `Hayden Liu` is `PERSON`, `2018` is `DATE`, `30` is `MONEY`, and `America` is `GPE` (country). Please refer to <https://spacy.io/api/annotation#section-named-entities> for a full list of named entity tags.

Stemming and lemmatization

Word **stemming** is a process of reverting an inflected or derived word to its root form. For instance, *machine* is the stem of *machines*, and *learning* and *learned* are generated from *learn* as their stem.

The word **lemmatization** is a cautious version of stemming. It considers the PoS of a word when conducting stemming. Also, it traces back to the lemma of the word. We will discuss these two text preprocessing techniques, stemming and lemmatization, in further detail shortly. For now, let's take a quick look at how they're implemented respectively in NLTK by performing the following steps:

1. Import `porter` as one of the three built-in stemming algorithms (`LancasterStemmer` and `SnowballStemmer` are the other two) and initialize the stemmer as follows:

```
>>> from nltk.stem.porter import PorterStemmer  
>>> porter_stemmer = PorterStemmer()
```

2. Stem `machines` and `learning`, as shown in the following codes:

```
>>> porter_stemmer.stem('machines')  
'machin'  
>>> porter_stemmer.stem('learning')  
'learn'
```



Stemming sometimes involves the chopping of letters if necessary, as you can see in `machin` in the preceding command output.

3. Now, import a lemmatization algorithm based on the built-in WordNet corpus and initialize a `lemmatizer`:

```
>>> from nltk.stem import WordNetLemmatizer  
>>> lemmatizer = WordNetLemmatizer()
```

Similar to stemming, we lemmatize `machines`, and `learning`:

```
>>> lemmatizer.lemmatize('machines')  
'machine'  
>>> lemmatizer.lemmatize('learning')  
'learning'
```

Why is `learning` unchanged? It turns out that this algorithm only lemmatizes on nouns by default.

Semantics and topic modeling

Gensim is famous for its powerful semantic and topic modeling algorithms. Topic modeling is a typical text mining task of discovering the hidden semantic structures in a document. A semantic structure in plain English is the distribution of word occurrences. It is obviously an unsupervised learning task. What we need to do is to feed in plain text and let the model figure out the abstract topics. We will study topic modeling in detail in *Chapter 10, Discovering Underlying Topics in the Newsgroups Dataset with Clustering and Topic Modeling*.

In addition to robust semantic modeling methods, gensim also provides the following functionalities:

- **Word embedding:** Also known as **word vectorization**, this is an innovative way to represent words while preserving words' co-occurrence features. We will study word embedding in detail in *Chapter 11, Machine Learning Best Practices*.
- **Similarity querying:** This functionality retrieves objects that are similar to the given query object. It's a feature built on top of word embedding.
- **Distributed computing:** This functionality makes it possible to efficiently learn from millions of documents.

Last but not least, as mentioned in the first chapter, *Getting Started with Machine Learning and Python*, scikit-learn is the main package we have used throughout this entire book. Luckily, it provides all the text processing features we need, such as tokenization, along with comprehensive machine learning functionalities. Plus, it comes with a built-in loader for the 20 newsgroups dataset.

Now that the tools are available and properly installed, what about the data?

Getting the newsgroups data

The project in this chapter is about the 20 newsgroups dataset. It's composed of text taken from newsgroup articles, as its name implies. It was originally collected by Ken Lang and now has been widely used for experiments in text applications of machine learning techniques, specifically NLP techniques.

The data contains approximately 20,000 documents across 20 online newsgroups. A newsgroup is a place on the internet where people can ask and answer questions about a certain topic. The data is already cleaned to a certain degree and already split into training and testing sets. The cutoff point is at a certain date.

The original data comes from <http://qwone.com/~jason/20Newsgroups/>, with 20 different topics listed, as follows:

- `comp.graphics`
- `comp.os.ms-windows.misc`
- `comp.sys.ibm.pc.hardware`
- `comp.sys.mac.hardware`
- `comp.windows.x`
- `rec.autos`
- `rec.motorcycles`
- `rec.sport.baseball`
- `rec.sport.hockey`
- `sci.crypt`
- `sci.electronics`
- `sci.med`
- `sci.space`
- `misc.forsale`
- `talk.politics.misc`
- `talk.politics.guns`
- `talk.politics.mideast`
- `talk.religion.misc`
- `alt.atheism`
- `soc.religion.christian`

All of the documents in the dataset are in English. And we can easily deduce the topics from the newsgroups' names.

The dataset is labeled and each document is composed of text data and a group label. This also makes it a perfect fit for supervised learning, such as text classification. At the end of the chapter, feel free to practice classification on this dataset using what you've learned so far in this book.

Some of the newsgroups are closely related or even overlapping, for instance, those five computer groups (`comp.graphics`, `comp.os.ms-`

`windows.misc`, `comp.sys.ibm.pc.hardware`, `comp.sys.mac.hardware`, and `comp.windows.x`), while some are not closely related to each other, such as Christian (`soc.religion.christian`) and baseball (`rec.sport.baseball`).

Hence, it's a perfect use case for unsupervised learning such as clustering, with which we can see whether similar topics are grouped together and unrelated ones are far apart. Moreover, we can even discover abstract topics beyond the original 20 labels using topic modeling techniques.

For now, let's focus on exploring and analyzing the text data. We will get started with acquiring the data.

It is possible to download the dataset manually from the original website or many other online repositories. However, there are also many versions of the dataset—some are cleaned in a certain way and some are in raw form. To avoid confusion, it is best to use a consistent acquisition method. The scikit-learn library provides a utility function that loads the dataset. Once the dataset is downloaded, it's automatically cached. We don't need to download the same dataset twice.



In most cases, caching the dataset, especially for a relatively small one, is considered a good practice. Other Python libraries also provide data download utilities, but not all of them implement automatic caching. This is another reason why we love scikit-learn.

As always, we first import the loader function for the 20 newsgroups data, as follows:

```
>>> from sklearn.datasets import fetch_20newsgroups
```

Then, we download the dataset with all the default parameters, as follows:

```
>>> groups = fetch_20newsgroups()
Downloading 20news dataset. This may take a few minutes.
Downloading dataset from https://ndownloader.figshare.com/files/5975967 (14 M
```

We can also specify one or more certain topic groups and particular sections (training, testing, or both) and just load such a subset of data in the program. The full list of parameters and options for the loader function is summarized in the following table:

Parameter	Default value	Example values	Description
<code>subset</code>	'train'	'train','test','all'	The dataset to load: the training set,

			the testing set, or both.
<code>data_home</code>	<code>~/scikit_learn_data</code>	<code>~/myfolder</code>	Directory where the files are stored and cached
<code>categories</code>	None	<code>['sci.space', 'alt.atheism']</code>	List of newsgroups to load. If None, all newsgroups will be loaded.
<code>shuffle</code>	True	True, False	Boolean indicating whether to shuffle the data
<code>random_state</code>	42	7, 43	Random seed integer used to shuffle the data
<code>remove</code>	0	<code>('headers','footers','quotes')</code>	Tuple indicating the part(s) among header, footer, and quote of each newsgroup post to omit. Nothing is removed by default.
<code>download_if_missing</code>	True	True, False	Boolean indicating whether to download the data if it is not found locally

Table 9.2: List of parameters of the `fetch_20newsgroups()` function

Remember that `random_state` is useful for the purpose of reproducibility. You are able to get the same dataset every time you run the script. Otherwise, working on datasets shuffled under different orders might bring in unnecessary variations.

In this section, we loaded the newsgroups data. Let's explore it next.

Exploring the newsgroups data

After we download the 20 newsgroups dataset by whatever means we prefer, the `data` object of `groups` is now cached in memory. The `data` object is in the form of a key-value dictionary. Its keys are as follows:

```
>>> groups.keys()
dict_keys(['data', 'filenames', 'target_names', 'target', 'DESCR'])
```

The `target_names` key gives the newsgroups names:

```
>>> groups['target_names']
['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.
```

The `target` key corresponds to a newsgroup, but is encoded as an integer:

```
>>> groups.target
array([7, 4, 4, ..., 3, 1, 8])
```

Then, what are the distinct values for these integers? We can use the `unique` function from NumPy to figure it out:

```
>>> import numpy as np
>>> np.unique(groups.target)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

They range from `0` to `19`, representing the 1st, 2nd, 3rd, ..., 20th newsgroup topics in `groups['target_names']`.

In the context of multiple topics or categories, it is important to know what the distribution of topics is. A balanced class distribution is the easiest to deal with, because there are no

under-represented or over-represented categories. However, frequently we have a skewed distribution with one or more categories dominating.

We will use the `seaborn` package (<https://seaborn.pydata.org/>) to compute the histogram of categories and plot it utilizing the `matplotlib` package (<https://matplotlib.org/>). We can install both packages via `pip` as follows:

```
python -m pip install -U matplotlib  
pip install seaborn
```

In the case of `conda`, you can execute the following command line:

```
conda install -c conda-forge matplotlib  
conda install seaborn
```

Remember to install `matplotlib` before `seaborn` as `matplotlib` is one of the dependencies of the `seaborn` package.

Now, let's display the distribution of the classes, as follows:

```
>>> import seaborn as sns  
>>> import matplotlib.pyplot as plt  
>>> sns.distplot(groups.target)  
<matplotlib.axes._subplots.AxesSubplot object at 0x108ada6a0>  
>>> plt.show()
```

Refer to the following screenshot for the end result:

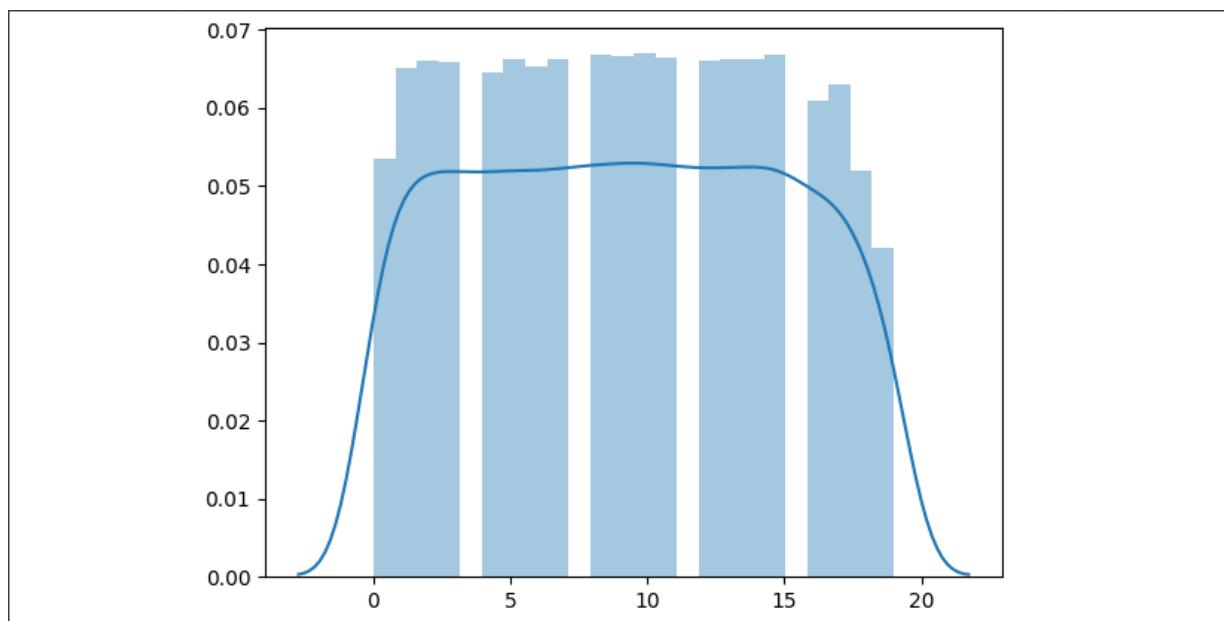


Figure 9.4: Distribution of newsgroup classes

As you can see, the distribution is approximately uniform so that's one less thing to worry about.



It's good to visualize data to get a general idea of how the data is structured, what possible issues may arise, and whether there are any irregularities that we have to take care of.

Other keys are quite self-explanatory: `data` contains all newsgroups documents and `filenames` store the path where each document is located in your filesystem.

Now, let's now have a look at the first document and its topic number and name by executing the following command:

```
>>> groups.data[0]
"From: lerxst@wam.umd.edu (where's my thing)\nSubject: WHAT car is this!?\nNr
>>> groups.target[0]
7
>>> groups.target_names[groups.target[0]]
'rec.autos'
```



If `random_state` isn't fixed (42 by default), you may get different results running the preceding scripts.

As you can see, the first document is from the `rec.autos` newsgroup, which was assigned the number `7`. Reading this post, we can easily figure out that it's about cars. The word `car` actually occurs a number of times in the document. Words such as `bumper` also seem very car-oriented. However, words such as `doors` may not necessarily be car related, as they may also be associated with home improvement or another topic.

As a side note, it makes sense to not distinguish between `doors` and `door`, or the same word with different capitalization, such as `Doors`. There are some rare cases where capitalization does matter, for instance, if we're trying to find out whether a document is about the band called `The Doors` or the more common concept, `the doors` (in wood).

Thinking about features for text data

From the preceding analysis, we can safely conclude that, if we want to figure out whether a document was from the `rec.autos` newsgroup, the presence or absence of words such as `car`, `doors`, and `bumper` can be very useful features. The presence or not of a word is

a Boolean variable, and we can also look at the count of certain words. For instance, `car` occurs multiple times in the document. Maybe the more times such a word is found in a text, the more likely it is that the document has something to do with cars.

Counting the occurrence of each word token

It seems that we are only interested in the occurrence of certain words, their count, or a related measure, and not in the order of the words. We can therefore view a text as a collection of words. This is called the **Bag of Words (BoW)** model. This is a very basic model, but it works pretty well in practice. We can optionally define a more complex model that takes into account the order of words and PoS tags. However, such a model is going to be more computationally expensive and more difficult to program. In reality, the basic BoW model in most cases suffices. We can give it a shot and see whether the BoW model makes sense.

We begin by converting documents into a matrix where each row represents each newsgroup document and each column represents a word token, or specifically, a unigram to begin with. And the value of each element in the matrix is the number of times the word (column) occurs in the document (row). We are utilizing the `CountVectorizer` class from scikit-learn to do the work:

```
>>> from sklearn.feature_extraction.text import CountVectorizer
```

The important parameters and options for the count conversion function are summarized in the following table:

Constructor parameter	Default value	Example values	Description
<code>ngram_range</code>	(1,1)	(1,2), (2,2)	Lower and upper bound of the n-grams to be extracted in the input text, for example (1,1) means unigram, (1,2) means unigram and bigram
<code>stop_words</code>	None	'english', or list ['a','the','of'] or None	Which stop word list to use, can be "english" referring to the built in list, or a customized input list. If None, no words will be removed.
<code>lowercase</code>	True	True, False	Whether or not converting all characters to lowercase
<code>max_features</code>	None	None, 200, 500	The number of top (most frequent) tokens to consider, or all tokens if None

<code>binary</code>	False	True, False	If true, all non-zero counts becomes 1s
---------------------	-------	-------------	---

Table 9.3: List of parameters of the CountVectorizer() function

We first initialize the count vectorizer with `500` top features (500 most frequent tokens):

```
>>> count_vector = CountVectorizer(max_features=500)
```

Use it to fit on the raw text data as follows:

```
>>> data_count = count_vector.fit_transform(groups.data)
```

Now the count vectorizer captures the top 500 features and generates a token count matrix out of the original text input:

```
>>> data_count
<11314x500 sparse matrix of type '<class 'numpy.int64'>'>
      with 798221 stored elements in Compressed Sparse Row format>
>>> data_count[0]
<1x500 sparse matrix of type '<class 'numpy.int64'>'>
      with 53 stored elements in Compressed Sparse Row format>
```

The resulting count matrix is a sparse matrix where each row only stores non-zero elements (hence, only `798,221` elements instead of `11314 * 500 = 5,657,000`). For example, the first document is converted into a sparse vector composed of 53 non-zero elements. If you are interested in seeing the whole matrix, feel free to run the following:

```
>>> data_count.toarray()
```

If you just want the first row, run the following:

```
>>> data_count.toarray()[0]
```

Let's take a look at the following output derived from the preceding command:

Figure 9.5: Output of count vectorization

So, what are those 500 top features? They can be found in the following output:

```
>>> print(count_vector.get_feature_names())
['00', '000', '0d', '0t', '10', '100', '11', '12', '13', '14', '145', '15', '...
.....
.....
.....
'user', 'using', 'usually', 'uucp', 've', 'version', 'video', 'view', 'virgi
```

Our first trial doesn't look perfect. Obviously, the most popular tokens are numbers, or letters with numbers such as `a86`, which do not convey important information. Moreover, there are many words that have no actual meaning, such as `you`, `the`, `them`, and `then`.

Also, some words contain identical information, for example, `tell` and `told`, `use` and `used`, and `time` and `times`. Let's tackle these issues.

Text preprocessing

We begin by retaining letter-only words so that numbers such as `00` and `000` and combinations of letters and numbers such as `b8f` will be removed. The filter function is defined as follows:

```
>>> data_cleaned = []
>>> for doc in groups.data:
...     doc_cleaned = ' '.join(word for word in doc.split()
...                           if word.isalpha())
...     data_cleaned.append(doc_cleaned)
```

This will generate a cleaned version of the newsgroups data.

Dropping stop words

We didn't talk about `stop_words` as an important parameter in `CountVectorizer`. **Stop words** are those common words that provide little value in helping to differentiate documents. In general, stop words add noise to the BoW model and can be removed.

There's no universal list of stop words. Hence, depending on the tools or packages you are using, you will remove different sets of stop words. Take scikit-learn as an example—you can check the list as follows:

```
>>> from sklearn.feature_extraction import stop_words
>>> print(stop_words.ENGLISH_STOP_WORDS)
frozenset({'most', 'three', 'between', 'anyway', 'made', 'mine', 'none', 'col
.....
.....
'him', 'somehow', 'or', 'per', 'nowhere', 'fifteen', 'via', 'must', 'someone'
```

To drop stop words from the newsgroups data, we simply just need to specify the `stop_words` parameter:

```
>>> count_vector_sw = CountVectorizer(stop_words="english", max_features=500)
```

Besides stop words, you may notice that names are included in the top features, such as `andrew`. We can filter names with the `Name` corpus from NLTK we just worked with.

Reducing inflectional and derivational forms of words

As mentioned earlier, we have two basic strategies to deal with words from the same root—stemming and lemmatization. Stemming is a quicker approach that involves, if necessary, chopping off letters; for example, *words* becomes *word* after stemming. The result of stemming doesn't have to be a valid word. For instance, *trying* and *try* become *tri*. Lemmatizing, on the other hand, is slower but more accurate. It performs a dictionary lookup and guarantees to return a valid word. Recall that we implemented both stemming and lemmatization using NLTK in a previous section.

Putting all of these (preprocessing, dropping stop words, lemmatizing, and count vectorizing) together, we obtain the following:

```
>>> from nltk.corpus import names
>>> all_names = set(names.words())
>>> count_vector_sw = CountVectorizer(stop_words="english", max_features=500)
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> data_cleaned = []
>>> for doc in groups.data:
...     doc = doc.lower()
...     doc_cleaned = ' '.join(lemmatizer.lemmatize(word)
...                           for word in doc.split()
...                           if word.isalpha() and
...                           word not in all_names)
...     data_cleaned.append(doc_cleaned)
>>> data_cleaned_count = count_vector_sw.fit_transform(data_cleaned)
```

Now the features are much more meaningful:

```
>>> print(count_vector_sw.get_feature_names())
['able', 'accept', 'access', 'according', 'act', 'action', 'actually', 'add',
.....
.....
'short', 'shot', 'similar', 'simple', 'simply', 'single', 'site', 'situation',
'talk', 'talking', 'tape', 'tax', 'team', 'technical', 'technology', 'tell',
```

We have just converted text from each raw newsgroup document into a sparse vector of size 500. For a vector from a document, each element represents the number of times a word token occurs in this document. Also, these 500-word tokens are selected based on their overall occurrences after text preprocessing, the removal of stop words, and

lemmatization. Now you may ask questions such as, is such an occurrence vector representative enough, or does such an occurrence vector convey enough information that can be used to differentiate the document from documents on other topics? You will see the answer in the next section.

Visualizing the newsgroups data with t-SNE

We can answer these questions easily by visualizing those representation vectors. If we can see the document vectors from the same topic form a cluster, we did a good job mapping the documents into vectors. But how? They are of 500 dimensions, while we can visualize data of **at most** three dimensions. We can resort to t-SNE for dimensionality reduction.

What is dimensionality reduction?

Dimensionality reduction is an important machine learning technique that reduces the number of features and, at the same time, retains as much information as possible. It is usually performed by obtaining a set of new principal features.

As mentioned before, it is difficult to visualize data of high dimension. Given a three-dimensional plot, we sometimes don't find it straightforward to observe any findings, not to mention 10, 100, or 1,000 dimensions. Moreover, some of the features in high dimensional data may be correlated and, as a result, bring in redundancy. This is why we need dimensionality reduction.

Dimensionality reduction is not simply taking out a pair of two features from the original feature space. It is transforming the original feature space to a new space of fewer dimensions. The data transformation can be linear, such as the famous one, **principal component analysis (PCA)**, which maps the data in a higher dimensional space to a lower dimensional space where the variance of the data is maximized, as mentioned in *Chapter 3, Recognizing Faces with Support Vector Machine*, or nonlinear, such as neural networks and t-SNE, which is coming up shortly. **Non-negative matrix factorization (NMF)** is another powerful algorithm, which we will study in detail in *Chapter 10, Discovering Underlying Topics in the Newsgroups Dataset with Clustering and Topic Modeling*.

At the end of the day, most dimensionality reduction algorithms are in the family of **unsupervised learning** as the target or label information (if available) is not used in data transformation.

t-SNE for dimensionality reduction

t-SNE stands for **t-distributed Stochastic Neighbor Embedding**. It is a nonlinear dimensionality reduction technique developed by Laurens van der Maaten and Geoffrey Hinton. t-SNE has been widely used for data visualization in various domains, including computer vision, NLP, bioinformatics, and computational genomics.

As its name implies, t-SNE embeds high-dimensional data into a low-dimensional (usually two-dimensional or three-dimensional) space where similarity among data samples (neighbor information) is preserved. It first models a probability distribution over neighbors around data points by assigning a high probability to similar data points and an extremely small probability to dissimilar ones. Note that similarity and neighbor distances are measured by Euclidean distance or other metrics. Then, t-SNE constructs a projection onto a low-dimensional space where the divergence between the input distribution and output distribution is minimized. The original high-dimensional space is modeled as a Gaussian distribution, while the output low-dimensional space is modeled as t-distribution.

We'll herein implement t-SNE using the `TSNE` class from scikit-learn:

```
>>> from sklearn.manifold import TSNE
```

Now, let's use t-SNE to verify our count vector representation.

We pick three distinct topics, `talk.religion.misc`, `comp.graphics`, and `sci.space`, and visualize document vectors from these three topics.

First, just load documents of these three labels, as follows:

```
>>> categories_3 = ['talk.religion.misc', 'comp.graphics', 'sci.space']
>>> groups_3 = fetch_20newsgroups(categories=categories_3)
```

We go through the same process and generate a count matrix, `data_cleaned_count_3`, with 500 features from the input, `groups_3`. You can refer to steps in previous sections as you just need to repeat the same code.

Next, we apply t-SNE to reduce the 500-dimensional matrix to a two-dimensional matrix:

```
>>> tsne_model = TSNE(n_components=2, perplexity=40,
                      random_state=42, learning_rate=500)
>>> data_tsne = tsne_model.fit_transform(data_cleaned_count_3.toarray())
```

The parameters we specify in the `TSNE` object are as follows:

- `n_components` : The output dimension
- `perplexity` : The number of nearest data points considered neighbors in the algorithm with a typical value of between 5 and 50
- `random_state` : The random seed for program reproducibility
- `learning_rate` : The factor affecting the process of finding the optimal mapping space with a typical value of between 10 and 1,000

Note that the `TSNE` object only takes in a dense matrix, hence we convert the sparse matrix, `data_cleaned_count_3`, into a dense one using `toarray()`.

We just successfully reduced the input dimension from 500 to 2. Finally, we can easily visualize it in a two-dimensional scatter plot where the `x` axis is the first dimension, the `y` axis is the second dimension, and the color, `c`, is based on the topic label of each original document:

```
>>> import matplotlib.pyplot as plt
>>> plt.scatter(data_tsne[:, 0], data_tsne[:, 1], c=groups_3.target)
>>> plt.show()
```

Refer to the following screenshot for the end result:

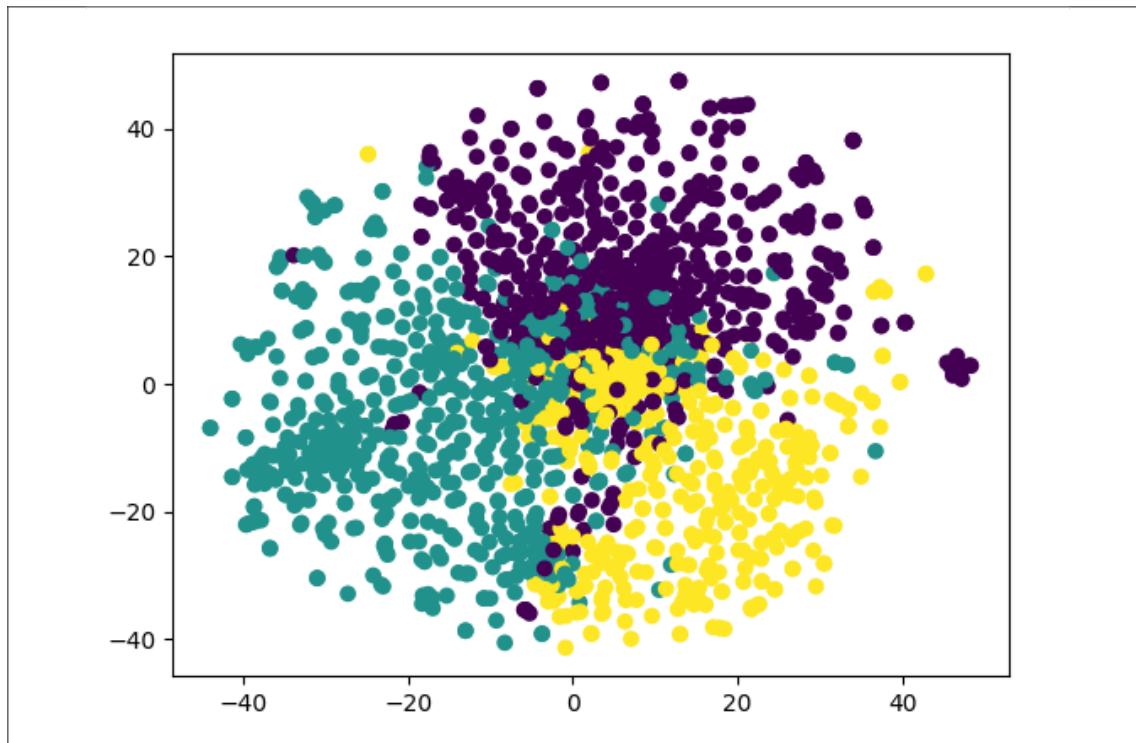


Figure 9.6: Applying t-SNE to data from three different topics

Data points from the three topics are in different colors, such as green, purple, and yellow. We can observe three clear clusters. Data points from the same topic are close to each other, while those from different topics are far away. Obviously, count vectors are great representations for original text data as they preserve the distinction among three different topics.

You can also play around with the parameters and see whether you can obtain a nicer plot where the three clusters are better separated.

Count vectorization does well in keeping document disparity. How about maintaining similarity? We can also check that using documents from overlapping topics, such as these five topics: `comp.graphics`, `comp.os.ms-windows.misc`, `comp.sys.ibm.pc.hardware`, `comp.sys.mac.hardware`, and `comp.windows.x`:

```
>>> categories_5 = ['comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware', 'comp.sys.mac.hardware', 'comp.windows.x']
```

Similar processes (including text clean-up, count vectorization, and t-SNE) are repeated and the resulting plot is displayed as follows:

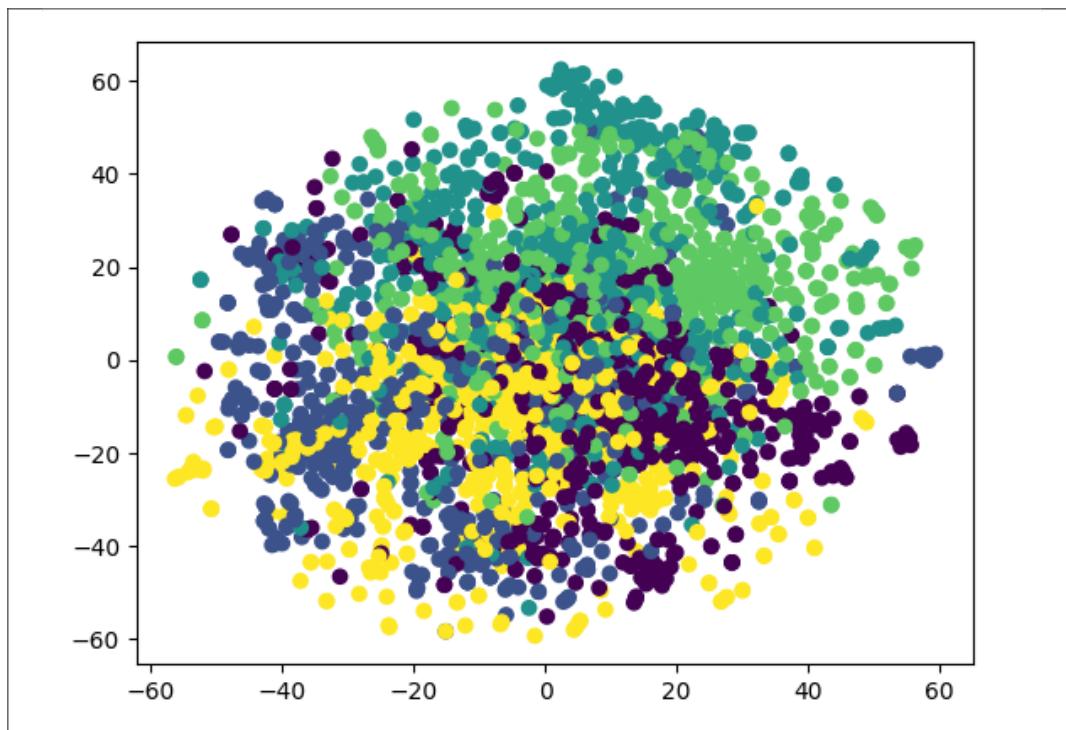


Figure 9.7: Applying t-SNE to data from five similar topics

Data points from those five computer-related topics are all over the place, which means they are contextually similar. To conclude, count vectors are great representations for original text data as they are also good at preserving similarity among related topics.

Summary

In this chapter, you learned the fundamental concepts of NLP as an important subfield in machine learning, including tokenization, stemming and lemmatization, and PoS tagging. We also explored three powerful NLP packages and worked on some common tasks using NLTK and `spacy`. Then, we continued with the main project exploring newsgroups data. We began by extracting features with tokenization techniques and went through text preprocessing, stop word removal, and stemming and lemmatization. We then performed dimensionality reduction and visualization with t-SNE and proved that count vectorization is a good representation for text data.

We had some fun mining the newsgroups data using dimensionality reduction as an unsupervised approach. Moving forward, in the next chapter, we'll be continuing our unsupervised learning journey, specifically looking at topic modeling and clustering.

Exercises

1. Do you think all of the top 500-word tokens contain valuable information? If not, can you impose another list of stop words?
2. Can you use stemming instead of lemmatization to process the newsgroups data?
3. Can you increase `max_features` in `CountVectorizer` from `500` to `5000` and see how the t-SNE visualization will be affected?
4. Try visualizing documents from six topics (similar or dissimilar) and tweak parameters so that the formed clusters look reasonable.

Discovering Underlying Topics in the Newsgroups Dataset with Clustering and Topic Modeling

In the previous chapter, we went through a text visualization using t-SNE. T-SNE, or any dimensionality reduction algorithm, is a type of unsupervised learning. Moving forward, in this chapter, we will be continuing our unsupervised learning journey, specifically focusing on clustering and topic modeling. We will start with how unsupervised learning learns without guidance and how it is good at discovering hidden information underneath data.

Next, we will talk about clustering as an important branch of unsupervised learning, which identifies different groups of observations from data. For instance, clustering is useful for market segmentation, where consumers of similar behaviors are grouped into one segment for marketing purposes. We will perform clustering on the 20 newsgroups text dataset and see what clusters will be produced.

Another unsupervised learning route we will take is topic modeling, which is the process of extracting themes hidden in the dataset. You will be amused by how many interesting themes we are able to mine from the 20 newsgroups dataset.

We will cover the following topics:

- What is unsupervised learning?
- Types of unsupervised learning

- What is k-means clustering and how does it work?
- Implementing k-means clustering from scratch
- Implementing k-means with scikit-learn
- Optimizing k-means clustering models
- Term frequency-inverse document frequency
- Clustering newsgroups data using k-means
- What is topic modeling?
- Non-negative matrix factorization for topic modeling
- Latent Dirichlet allocation for topic modeling
- Topic modeling on the newsgroups data

Learning without guidance – unsupervised learning

In the previous chapter, we applied t-SNE to visualize the newsgroup text data, reduced to 2 dimensions. T-SNE, or dimensionality reduction in general, is a type of **unsupervised learning**. Instead of having a teacher educating on what particular output to produce, such as a class or membership (classification), and a continuous value (regression), unsupervised learning identifies inherent structures or commonalities in the input data. Since there is no guidance from the "*teacher*" in unsupervised learning, there is no clear answer on what is a right or wrong result. Unsupervised learning has the freedom to discover hidden information underneath input data.

An easy way to understand unsupervised learning is to think of going through many practice questions for an exam. In supervised learning, you are given answers to those practice questions. You basically figure out the relationship between the questions and answers and learn how to map the questions to the answers. Hopefully, you will do well in the actual exam in the end by giving the correct answers. However, in unsupervised learning,

you are not provided with the answers to those practice questions. What you might do in this instance could include the following:

- Grouping similar practice questions so that you can later study related questions together at one time
- Finding questions that are highly repetitive so that you will not waste time on them
- Spotting rare questions so that you can be better prepared for them
- Extracting the key chunk of each question by removing boilerplate so you can cut to the point

You will notice that the outcomes of all these tasks are pretty open-ended. They are correct as long as they are able to describe the commonality and the structure underneath the data.

Practice questions are the **features** in machine learning, which are also often called **attributes**, **observations**, or **predictive variables**. Answers to questions are the labels in machine learning, which are also called **targets** or **target variables**. Practice questions with answers provided are **labeled data**, while practice questions without answers are **unlabeled data**. Unsupervised learning works with unlabeled data and acts on that information without guidance.

Unsupervised learning can include the following types:

- **Clustering:** This means grouping data based on commonality, which is often used for exploratory data analysis. Grouping similar practice questions, as mentioned earlier, is an example of clustering. Clustering techniques are widely used in customer segmentation or for grouping similar online behaviors for a marketing campaign.
- **Association:** This explores the co-occurrence of particular values of two or more features. Outlier detection (also called anomaly detection) is a typical case, where rare observations are identified. Spotting rare questions in the preceding example can be achieved using outlier detection techniques.

- **Projection:** This maps the original feature space to a reduced dimensional space retaining or extracting a set of principal variables. Extracting the key chunk of practice questions is an example projection, or specifically a dimensionality reduction.

Unsupervised learning is extensively employed in the area of NLP mainly because of the difficulty of obtaining labeled text data. Unlike numerical data (such as house prices, stock data, and online click streams), labeling text can sometimes be subjective, manual, and tedious. Unsupervised learning algorithms that do not require labels become effective when it comes to mining text data.

In *Chapter 9, Mining the 20 Newsgroups Dataset with Text Analysis Techniques*, you experienced using t-SNE to reduce the dimensionality of text data. Now, let's explore text mining with clustering algorithms and topic modeling techniques. We will start with clustering the newsgroups data.

Clustering newsgroups data using k-means

The newsgroups data comes with labels, which are the categories of the newsgroups, and a number of categories that are closely related or even overlapping, for instance, the five computer groups: `comp.graphics`, `comp.os.ms-windows.misc`, `comp.sys.ibm.pc.hardware`, `comp.sys.mac.hardware`, and `comp.windows.x`, and the two religion-related ones: `alt.atheism` and `talk.religion.misc`.

Let's now pretend we don't know those labels or they don't exist. Will samples from related topics be clustered together? We will now resort to the k-means clustering algorithm.

How does k-means clustering work?

The goal of the k-means algorithm is to partition the data into k groups based on feature similarities. K is a predefined property of a k-means clustering model. Each of the k clusters is specified by a centroid (center of a cluster) and each data sample belongs to the cluster with the nearest centroid. During training, the algorithm iteratively updates the k centroids based on the data provided. Specifically, it involves the following steps:

1. **Specifying k:** The algorithm needs to know how many clusters to generate as an end result.
2. **Initializing centroids:** The algorithm starts with randomly selecting k samples from the dataset as centroids.
3. **Assigning clusters:** Now that we have k centroids, samples that share the same closest centroid constitute one cluster. K clusters are created as a result. Note that closeness is usually measured by the **Euclidean distance**. Other metrics can also be used, such as the **Manhattan distance** and **Chebyshev distance**, which are listed in the following table:

Given two 2-dimension data points (x_1, y_1) and (x_2, y_2)	
Distance metric	Calculation
Euclidean distance	$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
Manhattan distance	$ x_1 - x_2 + y_1 - y_2 $
Chebyshev distance	$\max(x_1 - x_2 , y_1 - y_2)$

Figure 10.1: Distance metrics

4. **Updating centroids:** For each cluster, we need to recalculate its center point, which is the mean of all the samples in the cluster. K centroids are updated to be the means of corresponding clusters. This is why the algorithm is called **k-means**.
5. **Repeating steps 3 and 4:** We keep repeating assigning clusters and updating centroids until the model converges when no or a small enough update of centroids can be done, or enough iterations have been completed.

The outputs of a trained k-means clustering model include the following:

- The cluster ID of each training sample, ranging from 1 to k
- K centroids, which can be used to cluster new samples—a new sample will belong to the cluster of the closest centroid

It is very easy to understand the k-means clustering algorithm and its implementation is also straightforward, as you will discover next.

Implementing k-means from scratch

We will use the `iris` dataset from scikit-learn as an example. Let's first load the data and visualize it. We herein only use two features out of the original four for simplicity:

```
>>> from sklearn import datasets  
>>> iris = datasets.load_iris()  
>>> X = iris.data[:, 2:4]  
>>> y = iris.target
```

Since the dataset contains three iris classes, we plot it in three different colors, as follows:

```
>>> import numpy as np  
>>> from matplotlib import pyplot as plt  
>>> plt.scatter(X[:,0], X[:,1], c=y)  
>>> plt.show()
```

This will give us the following output for the original data plot:

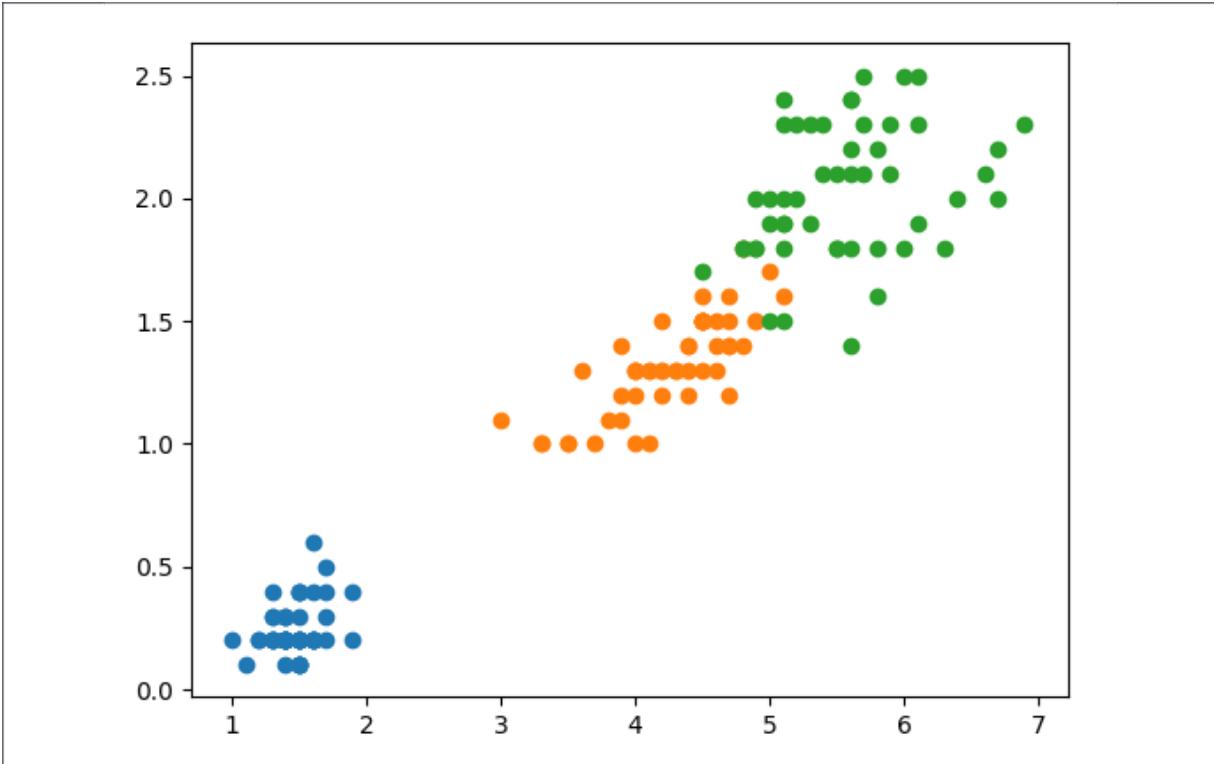


Figure 10.2: Plot of the original iris dataset

Assuming we know nothing about the label y , we try to cluster the data into three groups, as there seem to be three clusters in the preceding plot (or you might say two, which we will come back to later). Let's perform *step 1, specifying k* , and *step 2, initializing centroids*, by randomly selecting three samples as initial `centroids`:

```
>>> k = 3
>>> random_index = np.random.choice(range(len(X)), k)
>>> centroids = X[random_index]
```

We visualize the data (without labels any more) along with the initial random centroids:

```
>>> def visualize_centroids(X, centroids):
...     plt.scatter(X[:, 0], X[:, 1])
...     plt.scatter(centroids[:, 0], centroids[:, 1], marker='*',
...                 s=200, c='#050505')
```

```
...     plt.show()
>>> visualize_centroids(X, centroids)
```

Refer to the following screenshot for the data, along with the initial random centroids:

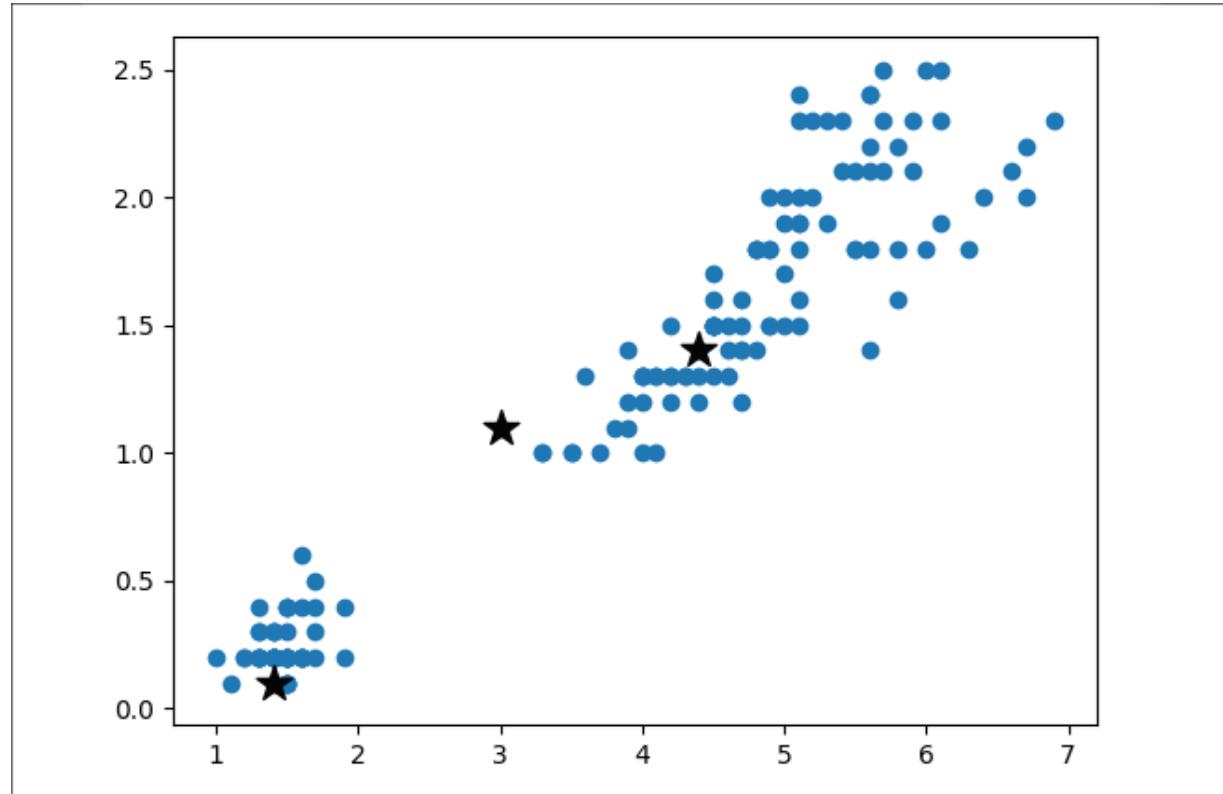


Figure 10.3: Data points with random centroids

Now we perform *step 3*, which entails assigning clusters based on the nearest centroids. First, we need to define a function calculating distance that is measured by the Euclidean distance, as demonstrated herein:

```
>>> def dist(a, b):
...     return np.linalg.norm(a - b, axis=1)
```

Then, we develop a function that assigns a sample to the cluster of the nearest centroid:

```
>>> def assign_cluster(x, centroids):
...     distances = dist(x, centroids)
...     cluster = np.argmin(distances)
...     return cluster
```

With the clusters assigned, we perform *step 4*, which involves updating the centroids to the mean of all samples in the individual `clusters`:

```
>>> def update_centroids(X, centroids, clusters):
...     for i in range(k):
...         cluster_i = np.where(clusters == i)
...         centroids[i] = np.mean(X[cluster_i], axis=0)
```

Finally, we have *step 5*, which involves repeating *step 3* and *step 4* until the model converges and whichever of the following occurs:

- Centroids move less than the pre-specified threshold
- Sufficient iterations have been taken

We set the tolerance of the first condition and the maximum number of iterations as follows:

```
>>> tol = 0.0001
>>> max_iter = 100
```

Initialize the clusters' starting values, along with the starting clusters for all samples as follows:

```
>>> iter = 0
>>> centroids_diff = 100000
>>> clusters = np.zeros(len(X))
```

With all the components ready, we can train the model iteration by iteration where it first checks convergence, before performing *steps 3* and *4*, and then visualizes the latest centroids:

```
>>> from copy import deepcopy
>>> while iter < max_iter and centroids_diff > tol:
...     for i in range(len(X)):
...         clusters[i] = assign_cluster(X[i], centroids)
...     centroids_prev = deepcopy(centroids)
...     update_centroids(X, centroids, clusters)
...     iter += 1
...     centroids_diff = np.linalg.norm(centroids -
...                                     centroids_prev)
...     print('Iteration:', str(iter))
...     print('Centroids:\n', centroids)
...     print('Centroids move: {:.5f}'.format(centroids_diff))
...     visualize_centroids(X, centroids)
```



Let's take a look at the following outputs generated from the preceding commands:

- **Iteration 1:** Take a look at the following output of iteration 1:

```
Iteration: 1
Centroids:
[[5.01827957 1.72258065]
 [3.41428571 1.05714286]
 [1.464       0.244 ]]
Centroids move: 0.8274
```

The plot of centroids after iteration 1 is as follows:

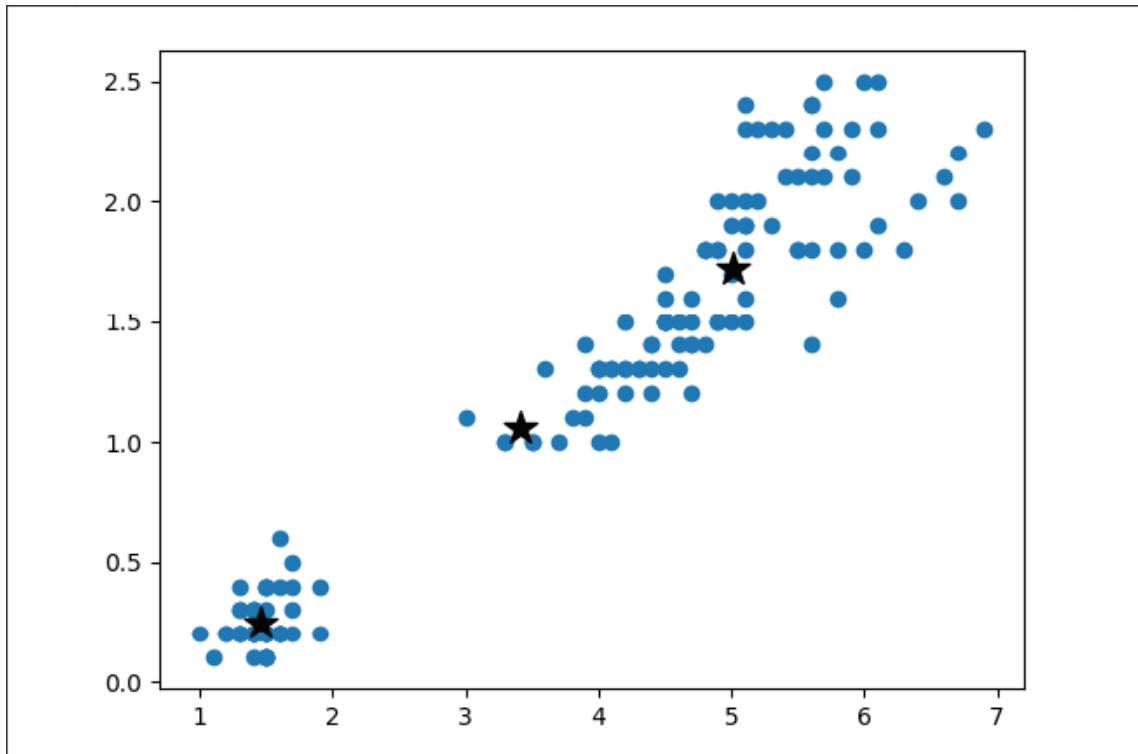


Figure 10.4: k-means clustering result after the first round

- **Iteration 2:** Take a look at the following output of iteration 2:

```
Iteration: 2
Centroids:
[[5.20897436 1.81923077]
 [3.83181818 1.16818182]
 [1.464      0.244 ]]
Centroids move: 0.4820
```

The plot of centroids after iteration 2 is as follows:

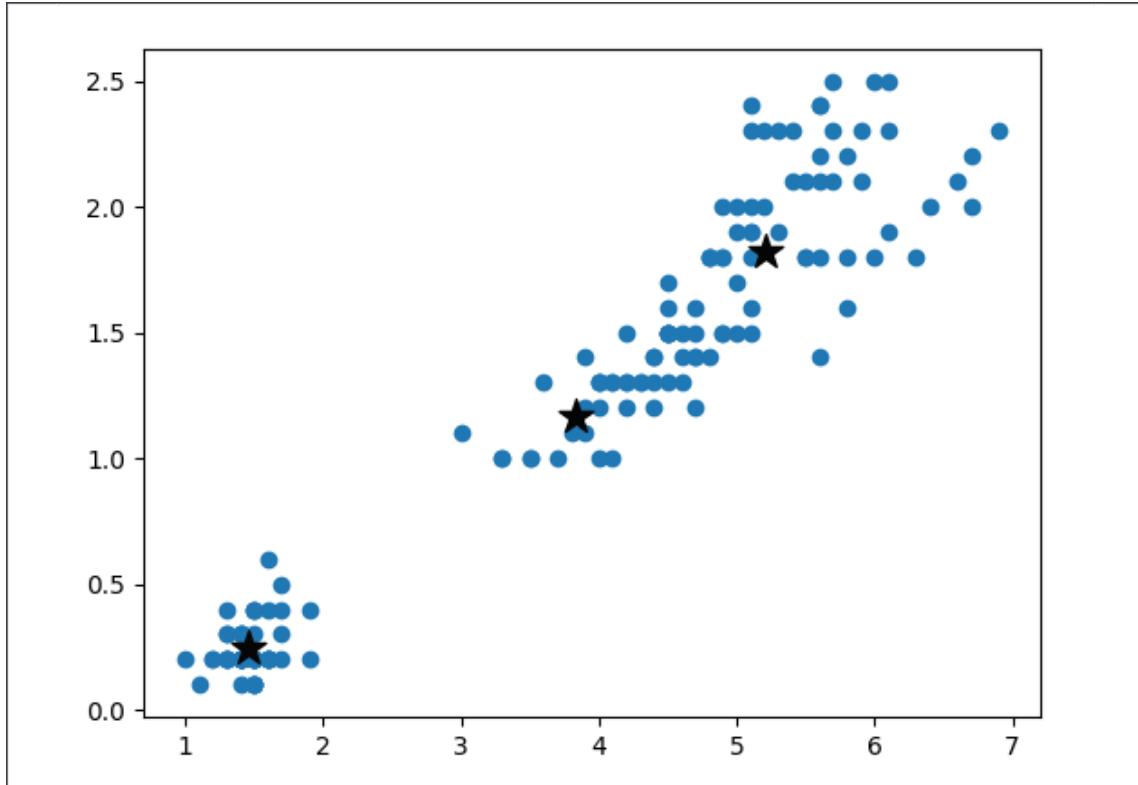


Figure 10.5: k-means clustering result after the second round

- **Iteration 3:** Take a look at the following output of iteration 3:

```
Iteration: 3
Centroids:
[[5.3796875  1.9125 ]
 [4.06388889 1.25555556]
 [1.464       0.244 ]]
Centroids move: 0.3152
```

The plot of centroids after iteration 3 is as follows:

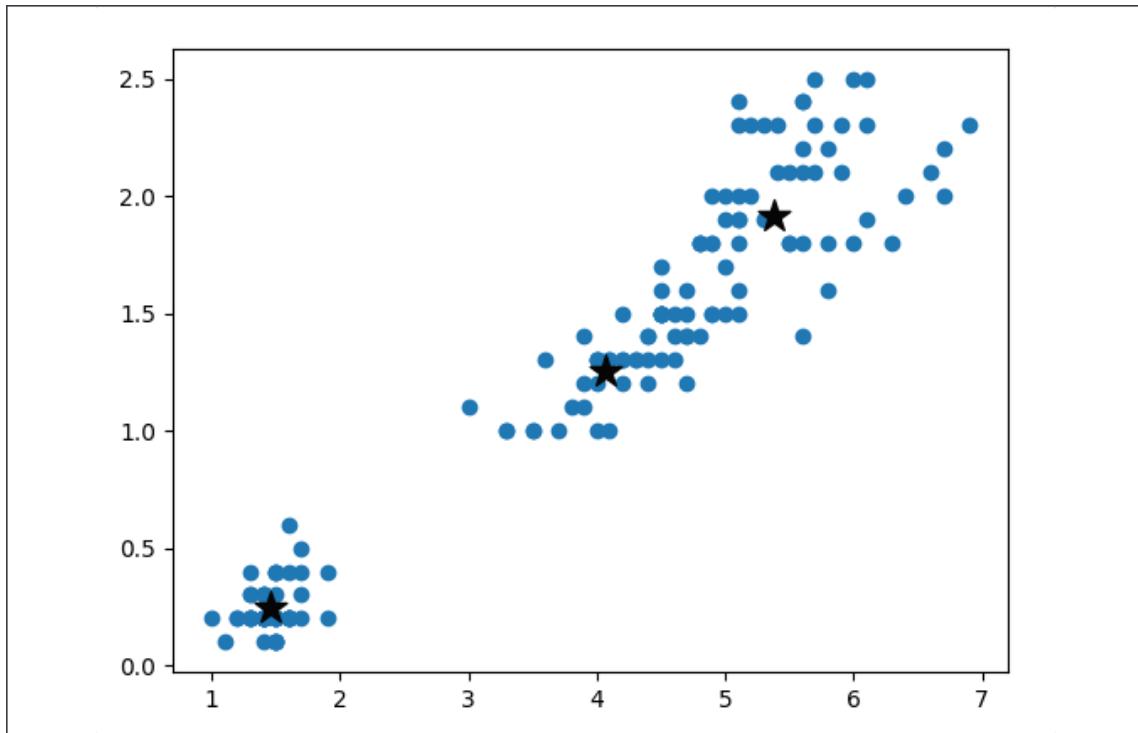


Figure 10.6: k-means clustering result after the third round

- **Iteration 4:** Take a look at the following output of iteration 4:

```
Iteration: 4
Centroids:
[[5.51481481 1.99444444]
 [4.19130435 1.30217391]
 [1.464       0.244 ]]
Centroids move: 0.2083
```

The plot of centroids after iteration 4 is as follows:

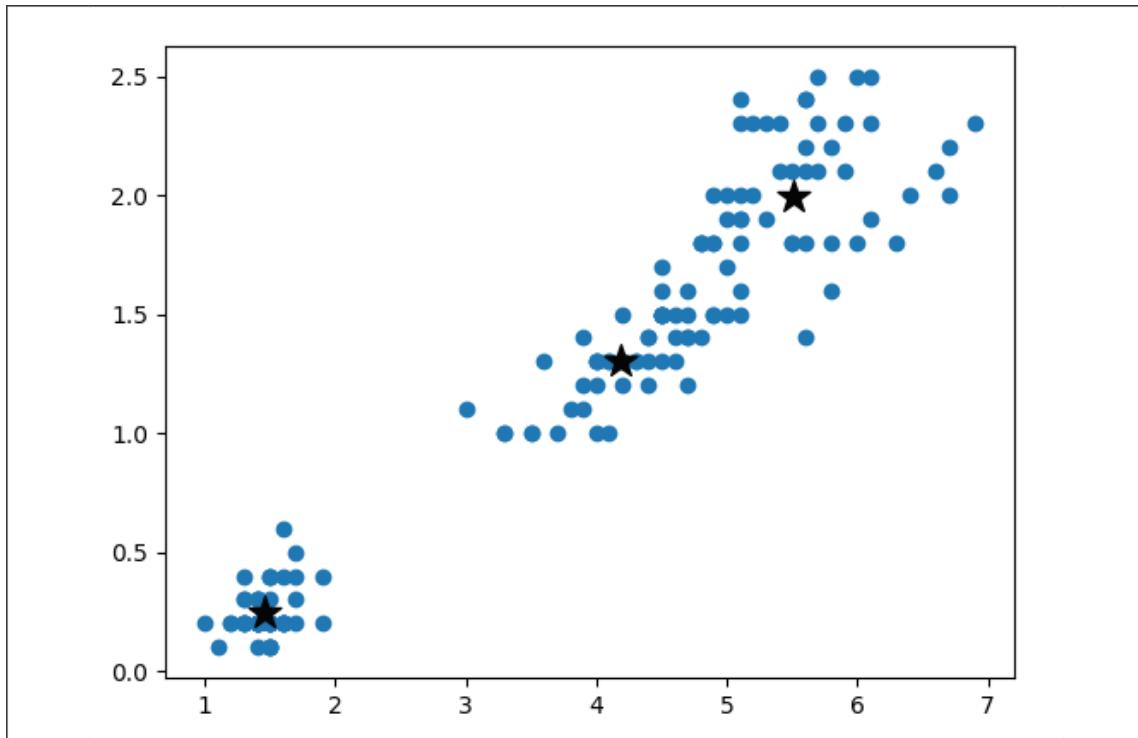


Figure 10.7: k-means clustering result after the fourth round

- **Iteration 5:** Take a look at the following output of iteration 5:

```
Iteration: 5
Centroids:
[[5.53846154 2.01346154]
 [4.22083333 1.31041667]
 [1.464      0.244 ]]
Centroids move: 0.0431
```

The plot of centroids after iteration 5 is as follows:

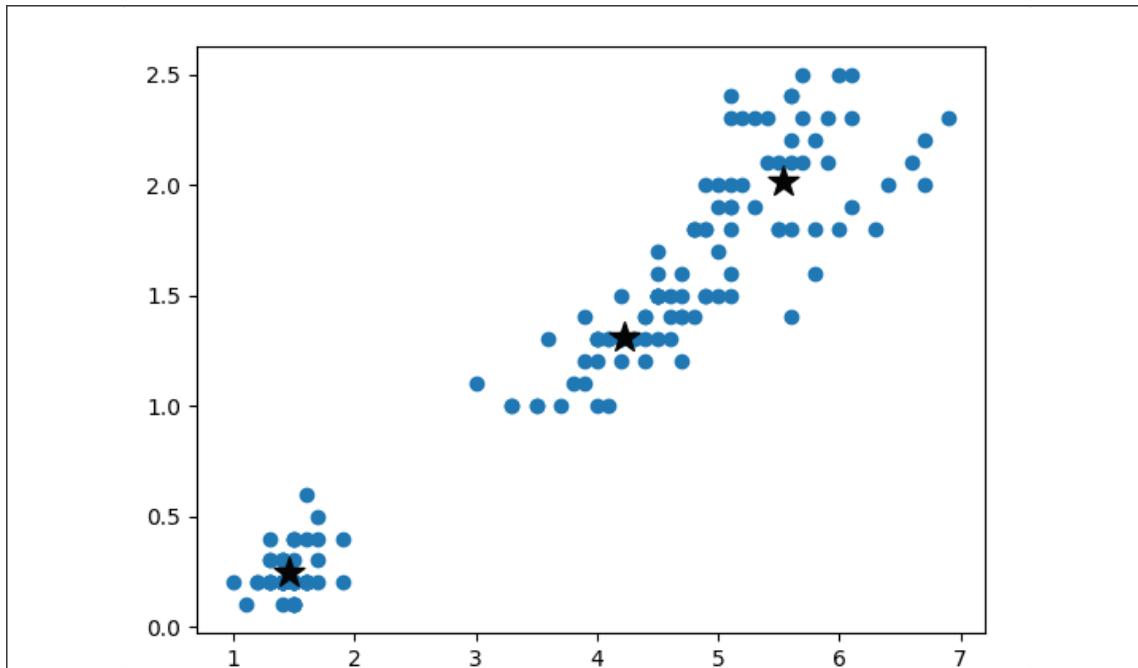


Figure 10.8: k-means clustering result after the fifth round

- **Iteration 6:** Take a look at the following output of iteration 6:

```
Iteration: 6
Centroids:
[[5.58367347 2.02653061]
 [4.25490196 1.33921569]
 [1.464 0.244 ]]
Centroids move: 0.0648
```

The plot of centroids after iteration 6 is as follows:

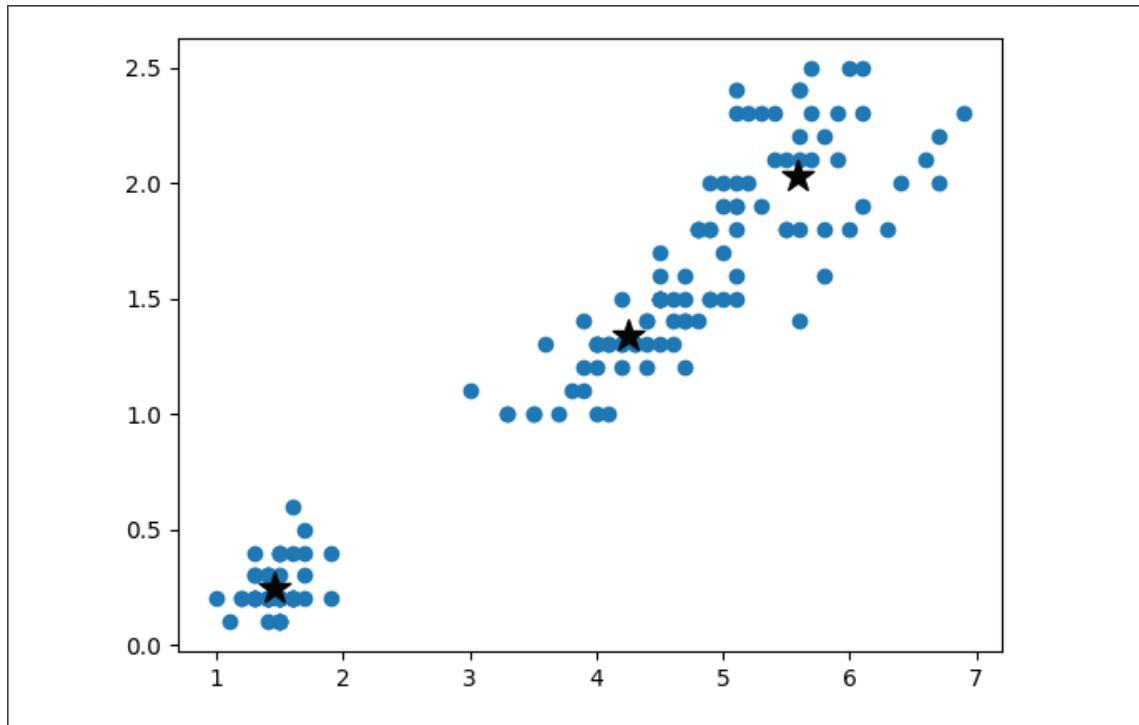


Figure 10.9: k-means clustering result after the sixth round

- **Iteration 7:** Take a look at the following output of iteration 7:

```
Iteration: 7
Centroids:
[[5.59583333 2.0375 ]
 [4.26923077 1.34230769]
 [1.464 0.244 ]]
Centroids move: 0.0220
```

The plot of centroids after iteration 7 is as follows:

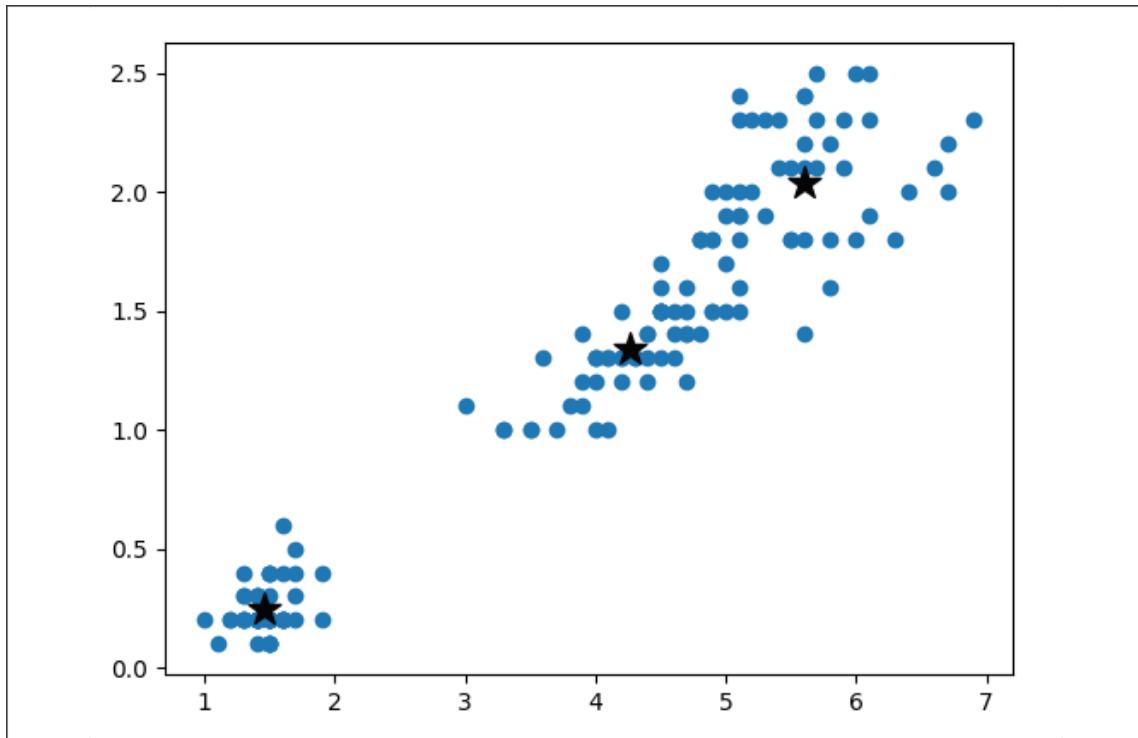


Figure 10.10: k-means clustering result after the seventh round

- **Iteration 8:** Take a look at the following output of iteration 8:

```
Iteration: 8
Centroids:
[[5.59583333 2.0375 ]
 [4.26923077 1.34230769]
 [1.464 0.244 ]]
Centroids move: 0.0000
```

The plot of centroids after iteration 8 is as follows:

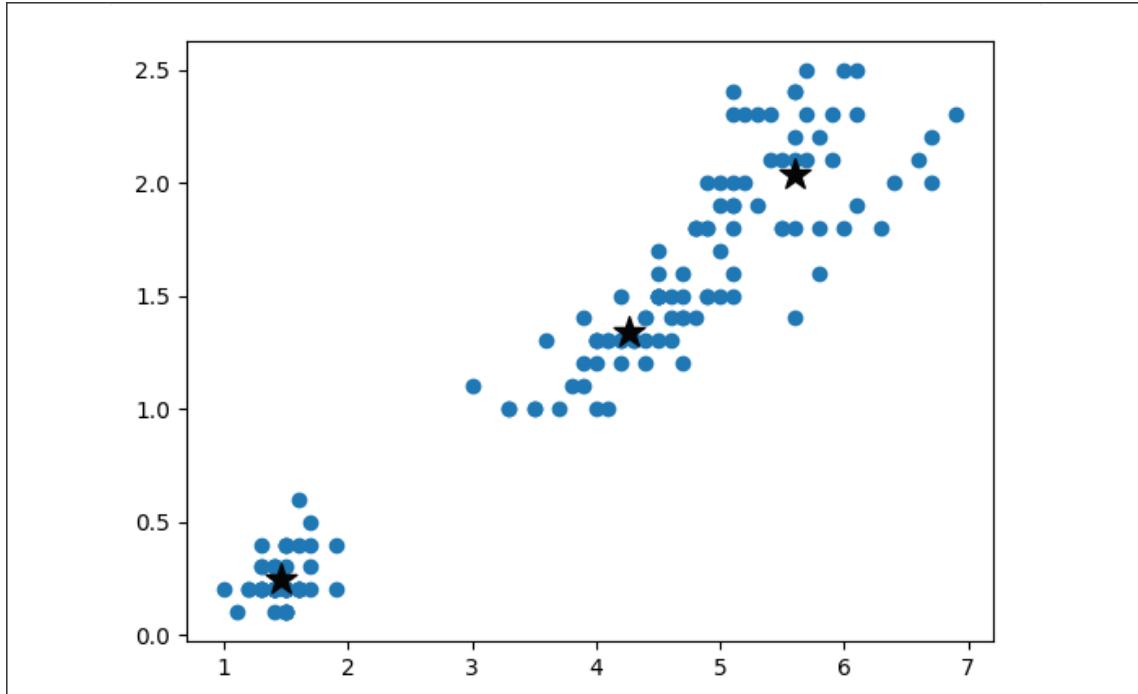


Figure 10.11: k-means clustering result after the eighth round

The model converges after eight iterations. The resulting centroids look promising, and we can also plot the clusters:

```
>>> plt.scatter(X[:, 0], X[:, 1], c=clusters)
>>> plt.scatter(centroids[:, 0], centroids[:, 1], marker='*',
               s=200, c='#050505')
>>> plt.show()
```

Refer to the following screenshot for the end result:

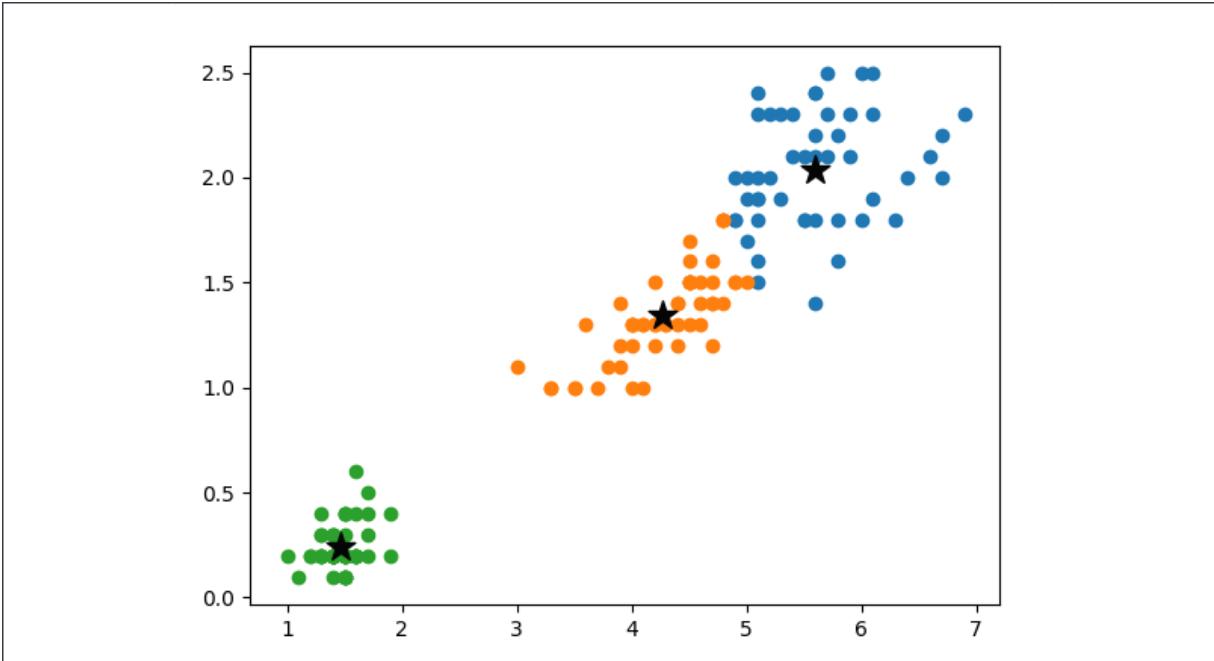


Figure 10.12: Data samples along with learned cluster centroids

As you can see, samples around the same centroid form a cluster. After eight iterations (you might see slightly more or less iterations in your case), the model converges and the centroids will no longer be updated.

Implementing k-means with scikit-learn

Having developed our own k-means clustering model, we will now discuss how to use scikit-learn for a quicker solution by performing the following steps:

1. First, import the `KMeans` class and initialize a model with three clusters, as follows:

```
>>> from sklearn.cluster import KMeans
>>> kmeans_sk = KMeans(n_clusters=3, random_state=42)
```

The `KMeans` class takes in the following important parameters:

Constructor parameter	Default value	Example values	Description
<code>n_clusters</code>	8	3, 5, 10	K clusters

<code>max_iter</code>	<code>300</code>	<code>10 , 100 , 500</code>	Maximum number of iterations
<code>tol</code>	<code>1e-4</code>	<code>1e-5 , 1e-8</code>	Tolerance to declare convergence
<code>random_state</code>	<code>None</code>	<code>0, 42</code>	Random seed for program reproducibility

Table 10.1: Parameters of the KMeans class

2. We then fit the model on the data:

```
>>> kmeans_sk.fit(X)
```

3. After that, we can obtain the clustering results, including the clusters for data samples and centroids of individual clusters:

```
>>> clusters_sk = kmeans_sk.labels_
>>> centroids_sk = kmeans_sk.cluster_centers_
```

4. Similarly, we plot the clusters along with the centroids:

```
>>> plt.scatter(X[:, 0], X[:, 1], c=clusters_sk)
>>> plt.scatter(centroids_sk[:, 0], centroids_sk[:, 1],
                 marker='*', s=200, c='#050505')
>>> plt.show()
```

This will result in the following output:

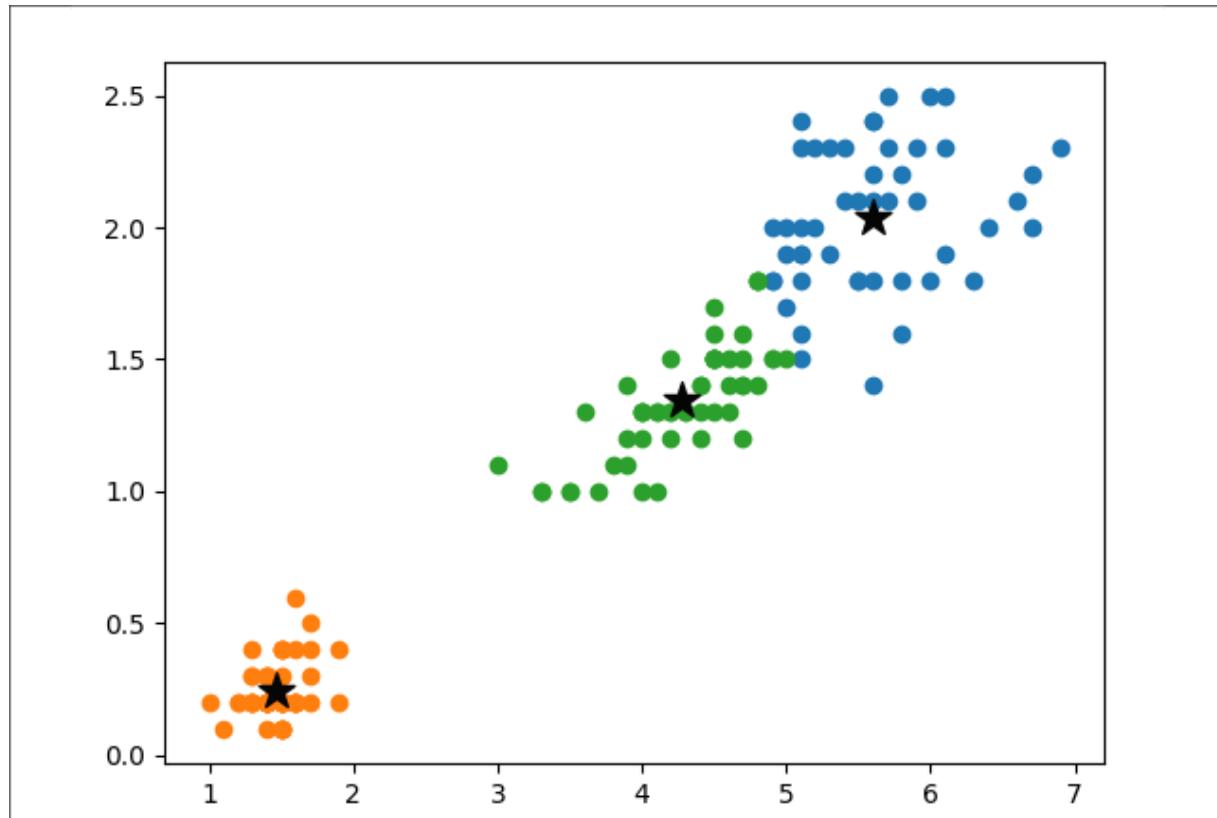


Figure 10.13: Data samples along with learned cluster centroids using scikit-learn

We get similar results to the previous one using the model we implemented from scratch.

Choosing the value of k

Let's return to our earlier discussion on what the right value for k is. In the preceding example, it is more intuitive to set it to 3 since we know there are three classes in total. However, in most cases, we don't know how many groups are sufficient or efficient, and meanwhile, the algorithm needs a specific value of k to start with. So, how can we choose the value for k ? There is a famous approach called the **Elbow method**.

In the Elbow method, different values of k are chosen and corresponding models are trained; for each trained model, the **sum of squared errors**, or **SSE** (also called the **sum of within-cluster distances**) of centroids is calculated and is plotted against k . Note that for one cluster, the squared

error (or the within-cluster distance) is computed as the sum of the squared distances from individual samples in the cluster to the centroid. The optimal k is chosen where the marginal drop of SSE starts to decrease dramatically. This means that further clustering does not provide any substantial gain.

Let's apply the Elbow method to the example we covered in the previous section (learning by examples is what this book is all about). We perform k -means clustering under different values of k on the `iris` data:

```
>>> iris = datasets.load_iris()
>>> X = iris.data
>>> y = iris.target
>>> k_list = list(range(1, 7))
>>> sse_list = [0] * len(k_list)
```

We use the whole feature space and k ranges from 1 to 6. Then, we train individual models and record the resulting SSE, respectively:

```
>>> for k_ind, k in enumerate(k_list):
...     kmeans = KMeans(n_clusters=k, random_state=42)
...     kmeans.fit(X)
...     clusters = kmeans.labels_
...     centroids = kmeans.cluster_centers_
...     sse = 0
...     for i in range(k):
...         cluster_i = np.where(clusters == i)
...         sse += np.linalg.norm(X[cluster_i] - centroids[i])
...     print('k={}, SSE={}'.format(k, sse))
...     sse_list[k_ind] = sse
k=1, SSE=26.103076447039722
k=2, SSE=16.469773740281195
k=3, SSE=15.089477089696558
k=4, SSE=15.0307321707491
k=5, SSE=14.858930749063735
k=6, SSE=14.883090350867239
```

Finally, we plot the SSE versus the various k ranges, as follows:

```
>>> plt.plot(k_list, sse_list)
>>> plt.show()
```

This will result in the following output:

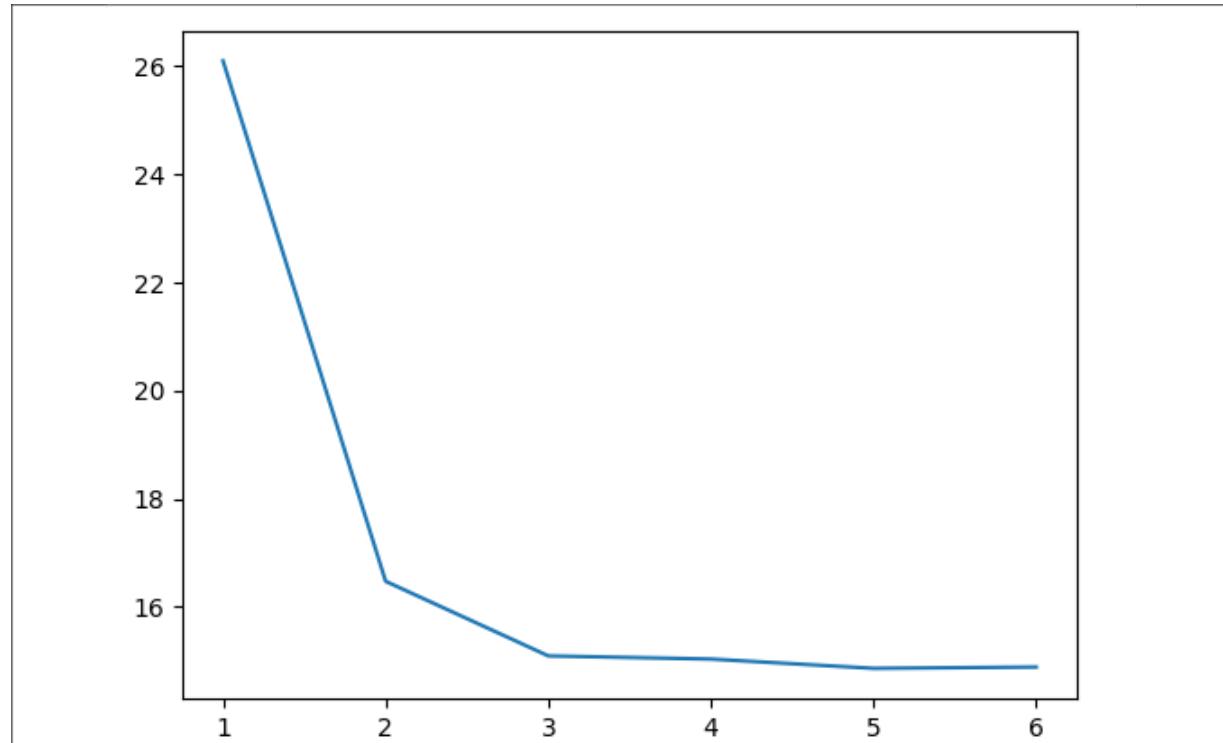


Figure 10.14: k-means elbow: SSE versus k

Apparently, the Elbow point is `k=3`, since the drop in SSE slows down dramatically right after `3`. Hence, `k=3` is an optimal solution in this case, which is consistent with the fact that there are three classes of flowers.

Clustering newsgroups data using k-means

You should now be very familiar with k-means clustering. Next, let's see what we are able to mine from the newsgroups dataset using this algorithm. We will use all data from four categories, `'alt.atheism'`, `'talk.religion.misc'`, `'comp.graphics'`, and `'sci.space'`, as an example.

We first load the data from those newsgroups and preprocess it as we did in *Chapter 9, Mining the 20 Newsgroups Dataset with Text Analysis Techniques*:

```

>>> from sklearn.datasets import fetch_20newsgroups
>>> categories = [
...     'alt.atheism',
...     'talk.religion.misc',
...     'comp.graphics',
...     'sci.space',
... ]
>>> groups = fetch_20newsgroups(subset='all',
...                               categories=categories)
>>> labels = groups.target
>>> label_names = groups.target_names
>>> def is_letter_only(word):
...     for char in word:
...         if not char.isalpha():
...             return False
...     return True
>>> from nltk.corpus import names
>>> all_names = set(names.words())
>>> from nltk.stem import WordNetLemmatizer
>>> lemmatizer = WordNetLemmatizer()
>>> data_cleaned = []
>>> for doc in groups.data:
...     doc = doc.lower()
...     doc_cleaned = ' '.join(lemmatizer.lemmatize(word) for
...                           word in doc.split() if word.isalpha()
...                           and word not in all_names)
...     data_cleaned.append(doc_cleaned)

```

We then convert the cleaned text data into count vectors using `CountVectorizer` of scikit-learn:

```

>>> from sklearn.feature_extraction.text import CountVectorizer
>>> count_vector = CountVectorizer(stop_words="english",
...                                 max_features=None, max_df=0.5, min_df=2)
>>> data = count_vector.fit_transform(data_cleaned)

```

Note that the vectorizer we use here does not limit the number of features (word tokens), but the minimum and maximum document frequency, which are 2 and 50% of the dataset, respectively. **Document frequency** of a word is measured by the fraction of documents (samples) in the dataset that contain this word.

With the input data ready, we will now try to cluster them into four groups as follows:

```
>>> from sklearn.cluster import KMeans  
>>> k = 4  
>>> kmeans = KMeans(n_clusters=k, random_state=42)  
>>> kmeans.fit(data)
```

Let's do a quick check on the sizes of the resulting clusters:

```
>>> clusters = kmeans.labels_  
>>> from collections import Counter  
>>> print(Counter(clusters))  
Counter({3: 3360, 0: 17, 1: 7, 2: 3})
```

The clusters don't look absolutely correct, with most samples (3360 samples) congested in one big cluster (cluster 3). What could have gone wrong? It turns out that our count-based features are not sufficiently representative. A better numerical representation for text data is the **term frequency-inverse document frequency (tf-idf)**. Instead of simply using the token count, or the so-called **term frequency (tf)**, it assigns each term frequency a weighting factor that is inversely proportional to the document frequency. In practice, the **idf** factor of a term t in documents D is calculated as follows:

$$idf(t, D) = \log \frac{n_D}{1 + n_t}$$

Here, n_D is the total number of documents, n_t is the number of documents containing the term t , and the 1 is added to avoid division by zero.

With the **idf** factor incorporated, the **tf-idf** representation diminishes the weight of common terms (such as *get* and *make*) and emphasizes terms that rarely occur, but that convey an important meaning.

To use the tf-idf representation, we just need to replace `CountVectorizer` with `TfidfVectorizer` from scikit-learn as follows:

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer  
>>> tfidf_vector = TfidfVectorizer(stop_words='english',  
                                max_features=None, max_df=0.5, min_df=2)
```

Now, redo feature extraction using the `tf-idf` vectorizer and the k-means clustering algorithm on the resulting feature space:

```
>>> data = tfidf_vector.fit_transform(data_cleaned)  
>>> kmeans.fit(data)  
>>> clusters = kmeans.labels_  
print(Counter(clusters))  
Counter({1: 1560, 2: 686, 3: 646, 0: 495})
```

The clustering result becomes more reasonable.

We also take a closer look at the clusters by examining what they contain and the top 10 terms (the terms with the 10 highest tf-idf) representing each cluster:

```
>>> cluster_label = {i: labels[np.where(clusters == i)] for i in  
                     range(k)}  
>>> terms = tfidf_vector.get_feature_names()  
>>> centroids = kmeans.cluster_centers_  
>>> for cluster, index_list in cluster_label.items():  
...     counter = Counter(cluster_label[cluster])  
...     print('cluster_{}: {} samples'.format(cluster, len(index_list)))  
...     for label_index, count in sorted(counter.items(),  
                                      key=lambda x: x[1], reverse=True):  
...         print('{}: {}'.format(label_names[label_index], count))  
...     print('Top 10 terms:')...     for ind in centroids[cluster].argsort()[-10:]:  
...         print(' %s' % terms[ind], end="")  
...     print()  
cluster_0: 495 samples  
sci.space: 494 samples  
comp.graphics: 1 samples  
Top 10 terms:  
toronto moon zoology nasa hst mission wa launch shuttle space  
cluster_1: 1560 samples  
sci.space: 459 samples  
alt.atheism: 430 samples
```

```
talk.religion.misc: 352 samples
comp.graphics: 319 samples
Top 10 terms:
people new think know like ha just university article wa
cluster_2: 686 samples
comp.graphics: 651 samples
sci.space: 32 samples
alt.atheism: 2 samples
talk.religion.misc: 1 samples
Top 10 terms:
know thanks need format looking university program file graphic
cluster_3: 646 samples
alt.atheism: 367 samples
talk.religion.misc: 275 samples
sci.space: 2 samples
comp.graphics: 2 samples
Top 10 terms:
moral article morality think jesus people say christian wa god
```

From what we observe in the preceding results:

- `cluster_0` is obviously about space and includes almost all `sci.space` samples and related terms such as `moon`, `nasa`, `launch`, `shuttle`, and `space`
- `cluster_1` is more of a generic topic
- `cluster_2` is more about computer graphics and related terms, such as `format`, `program`, `file`, `graphic`, and `image`
- `cluster_3` is an interesting one, which successfully brings together two overlapping topics, atheism and religion, with key terms including `moral`, `morality`, `jesus`, `christian`, and `god`



Feel free to try different values of `k`, or use the Elbow method to find the optimal one (this is actually an exercise for this chapter).

It is quite interesting to find key terms for each text group via clustering. Topic modeling is another approach for doing so, but in a much more direct way. It does not simply search for the key terms in individual clusters

generated beforehand. What it does is directly extract collections of key terms from documents. You will see how this works in the next section.

Discovering underlying topics in newsgroups

A **topic model** is a type of statistical model for discovering the probability distributions of words linked to the topic. The topic in topic modeling does not exactly match the dictionary definition, but corresponds to a nebulous statistical concept, which is an abstraction that occurs in a collection of documents.

When we read a document, we expect certain words appearing in the title or the body of the text to capture the semantic context of the document. An article about Python programming will have words such as *class* and *function*, while a story about snakes will have words such as *eggs* and *afraid*. Documents usually have multiple topics; for instance, this recipe is about three things: topic modeling, non-negative matrix factorization, and latent Dirichlet allocation, which we will discuss shortly. We can therefore define an additive model for topics by assigning different weights to topics.

Topic modeling is widely used for mining hidden semantic structures in given text data. There are two popular topic modeling algorithms—non-negative matrix factorization and latent Dirichlet allocation. We will go through both of these in the next two sections.

Topic modeling using NMF

Non-negative matrix factorization (NMF) relies heavily on linear algebra. It factorizes an input matrix, \mathbf{V} , into a product of two smaller matrices, \mathbf{W} and \mathbf{H} , in such a way that these three matrices have no negative values. In the context of NLP, these three matrices have the following meanings:

- The input matrix \mathbf{V} is the term count or tf-idf matrix of size $n * m$, where n is the number of documents or samples, and m is the number of terms.
- The first decomposition output matrix \mathbf{W} is the feature matrix of size $t * m$, where t is the number of topics specified. Each row of \mathbf{W} represents a topic with each element in the row representing the rank of a term in the topic.
- The second decomposition output matrix \mathbf{H} is the coefficient matrix of size $n * t$. Each row of \mathbf{H} represents a document, with each element in the row representing the weight of a topic within the document.

How to derive the computation of \mathbf{W} and \mathbf{H} is beyond the scope of this book. However, you can refer to the following diagram to get a better sense of how NMF works:

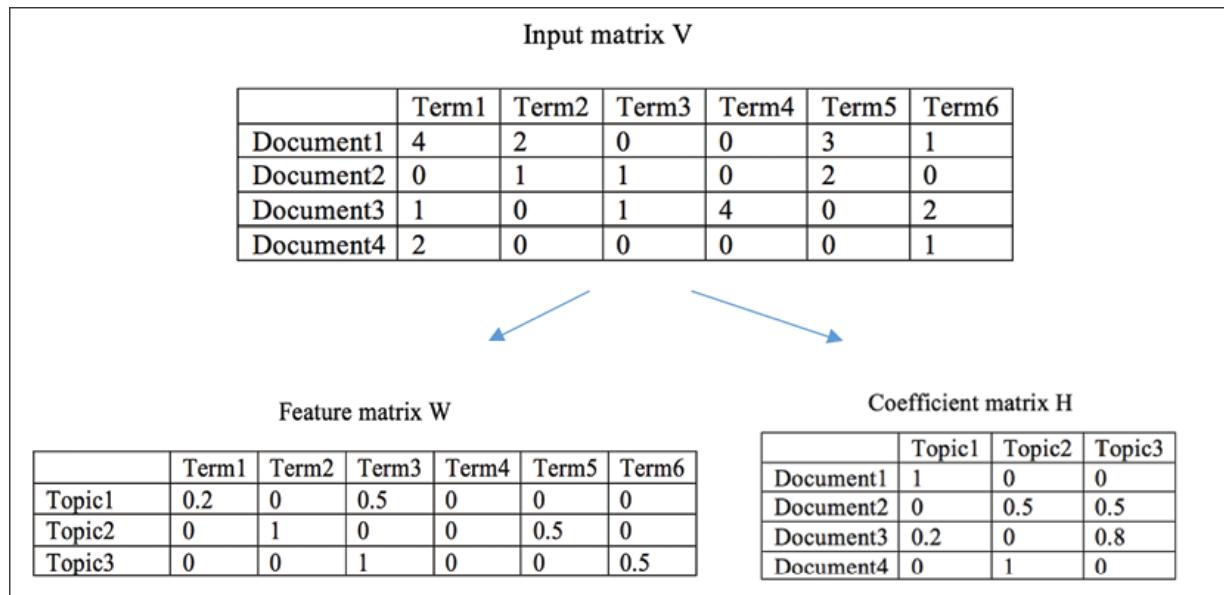


Figure 10.15: Example of matrix \mathbf{W} and matrix \mathbf{H} derived from an input matrix \mathbf{V}

If you are interested in reading more about NMF, feel free to check out the original paper *Generalized Nonnegative Matrix Approximations with Bregman Divergences*, by Inderjit S. Dhillon and Suvrit Sra, in *NIPS 2005*.

Let's now apply NMF to our newsgroups data. Scikit-learn has a nice module for decomposition that includes NMF:

```
>>> from sklearn.decomposition import NMF
>>> t = 20
>>> nmf = NMF(n_components=t, random_state=42)
```

We specify 20 topics (`n_components`) as an example. Important parameters of the model are included in the following table:

Constructor parameter	Default value	Example values	Description
<code>n_components</code>	<code>None</code>	<code>5</code> , <code>10</code> , <code>20</code>	Number of components — in the context of topic modeling, this corresponds to the number of topics. If <code>None</code> , it becomes the number of input features.
<code>max_iter</code>	<code>200</code>	<code>100</code> , <code>200</code>	Maximum number of iterations
<code>tol</code>	<code>1e-4</code>	<code>1e-5</code> , <code>1e-8</code>	Tolerance to declare convergence

Table 10.2: Parameters of the NMF class

We used the term matrix as input to the NMF model, but you could also use the tf-idf one instead. Here, we will reuse `count_vector`, as defined previously:

```
>>> data = count_vector.fit_transform(data_cleaned)
```

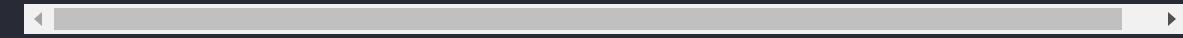
Now, fit the NMF model `nmf` on the term matrix `data`:

```
>>> nmf.fit(data)
```

We can obtain the resulting topic-feature rank **W** after the model is trained:



```
>>> nmf.components_
[[0.0000000e+00 0.0000000e+00 0.0000000e+00 ... 0.0000000e+0
 0.0000000e+00 1.81952400e-04]
[0.0000000e+00 0.0000000e+00 0.0000000e+00 ... 0.0000000e+00
 7.35497518e-04 3.65665719e-03]
[0.0000000e+00 0.0000000e+00 0.0000000e+00 ... 0.0000000e+00
 0.0000000e+00 0.0000000e+00]
...
[0.0000000e+00 0.0000000e+00 0.0000000e+00 ... 2.69725134e-02
 0.0000000e+00 0.0000000e+00]
[0.0000000e+00 0.0000000e+00 0.0000000e+00 ... 0.0000000e+00
 0.0000000e+00 4.26844886e-05]
[0.0000000e+00 0.0000000e+00 0.0000000e+00 ... 0.0000000e+00
 0.0000000e+00 0.0000000e+00]]
```



For each topic, we display the top 10 terms based on their ranks:

```
>>> terms = count_vector.get_feature_names()
>>> for topic_idx, topic in enumerate(nmf.components_):
...     print("Topic {}:".format(topic_idx))
...     print(" ".join([terms[i] for i in topic.argsort()[-1:0:-1]])
Topic 0:
available quality program free color version gif file image jpeg
Topic 1:
ha article make know doe say like just people think
Topic 2:
include available analysis user software ha processing data tool
Topic 3:
atmosphere kilometer surface ha earth wa planet moon spacecraft
Topic 4:
communication technology venture service market ha commercial sp
Topic 5:
verse wa jesus father mormon shall unto mcconkie lord god
Topic 6:
format message server object image mail file ray send graphic
Topic 7:
christian people doe atheism believe religion belief religious c
Topic 8:
file graphic grass program ha package ftp available image data
Topic 9:
speed material unified star larson book universe theory physicis
Topic 10:
planetary station program group astronaut center mission shuttle
Topic 11:
```

... [REDACTED]

```
infrared high astronomical center acronym observatory satellite  
Topic 12:  
used occurs true form ha ad premise conclusion argument fallacy  
Topic 13:  
gospel people day psalm prophecy christian ha matthew wa jesus  
Topic 14:  
doe word hanging say greek matthew mr act wa juda  
Topic 15:  
siggraph graphic file information format isbn data image ftp ava  
Topic 16:  
venera mar lunar surface space venus soviet mission wa probe  
Topic 17:  
april book like year time people new did article wa  
Topic 18:  
site retrieve ftp software data information client database goplr  
Topic 19:  
use look xv color make program correction bit gamma image
```

There are a number of interesting topics, for instance, computer graphics-related topics, such as 0, 2, 6, and 8, space-related ones, such as 3, 4, and 9, and religion-related ones, such as 5, 7, and 13. There are also two topics, 1 and 12, that are hard to interpret, which is totally fine since topic modeling is a kind of free-form learning.

Topic modeling using LDA

Let's explore another popular topic modeling algorithm, **latent Dirichlet allocation (LDA)**. LDA is a generative probabilistic graphical model that explains each input document by means of a mixture of topics with certain probabilities. Again, **topic** in topic modeling means a collection of words with a certain connection. In other words, LDA basically deals with two probability values, $P(\text{term} \vee \text{topic})$ and $P(\text{topic} \vee \text{document})$. This can be difficult to understand at the beginning. So, let's start from the bottom, the end result of an LDA model.

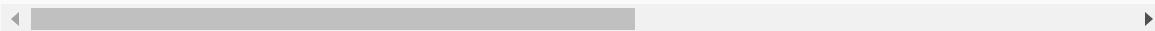
Let's take a look at the following set of documents:

```
Document 1: This restaurant is famous for fish and chips.  
Document 2: I had fish and rice for lunch.
```

```
Document 3: My sister bought me a cute kitten.  
Document 4: Some research shows eating too much rice is bad.  
Document 5: I always forget to feed fish to my cat.
```

Now, let's say we want two topics. The topics derived from these documents may appear as follows:

```
Topic 1: 30% fish, 20% chip, 30% rice, 10% lunch, 10% restaurant  
Topic 2: 40% cute, 40% cat, 10% fish, 10% feed (which we can int
```



Therefore, we find how each document is represented by these two topics:

```
Documents 1: 85% Topic 1, 15% Topic 2  
Documents 2: 88% Topic 1, 12% Topic 2  
Documents 3: 100% Topic 2  
Documents 4: 100% Topic 1  
Documents 5: 33% Topic 1, 67% Topic 2
```

After seeing a dummy example, we come back to its learning procedure:

1. Specify the number of topics, T . Now we have topic 1, 2, ..., and T .
2. For each document, randomly assign one of the topics to each term in the document.
3. For each document, calculate $P(topic = t \vee document)$, which is the proportion of terms in the document that are assigned to the topic t .
4. For each topic, calculate $P(term = w \vee topic)$, which is the proportion of term w among all terms that are assigned to the topic.
5. For each term w , reassign its topic based on the latest probabilities $P(topic = t \vee document)$ and $P(term = w \vee topic = t)$.
6. Repeat steps 3 to 5 under the latest topic distributions for each iteration. The training stops if the model converges or reaches the maximum number of iterations.

LDA is trained in a generative manner, where it tries to abstract from the documents a set of hidden topics that are likely to generate a certain

collection of words.

With all this in mind, let's see LDA in action. The LDA model is also included in scikit-learn:

```
>>> from sklearn.decomposition import LatentDirichletAllocation  
>>> t = 20  
>>> lda = LatentDirichletAllocation(n_components=t,  
                                     learning_method='batch', random_state=42)
```

Again, we specify 20 topics (`n_components`). The key parameters of the model are included in the following table:

Constructor parameter	Default value	Example values	Description
<code>n_components</code>	10	5, 10, 20	Number of components – in the context of topic modeling, this corresponds to the number of topics.
<code>learning_method</code>	"batch"	"online", "batch"	In batch mode, all training data is used for each update. In online mode, a mini-batch of training data is used for each update. In general, if the data size is large, the online mode is faster.
<code>max_iter</code>	10	10, 20	Maximum number of iterations.
<code>random_state</code>	None	0, 42	Seed used by the random number generator.

Table 10.3: Parameters of the `LatentDirichletAllocation` class

For the input data to LDA, remember that LDA only takes in term counts as it is a probabilistic graphical model. This is unlike NMF, which can work with both the term count matrix and the tf-idf matrix as long as they are

non-negative data. Again, we use the term matrix defined previously as input to the LDA model:

```
>>> data = count_vector.fit_transform(data_cleaned)
```

Now, fit the LDA model on the term matrix, `data`:

```
>>> lda.fit(data)
```

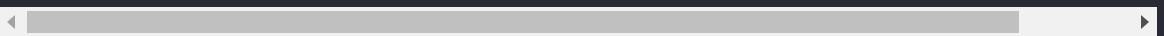
We can obtain the resulting topic-term rank after the model is trained:

```
>>> lda.components_
[[0.05    2.05    2.05    ...   0.05    0.05    0.05 ],
 [0.05    0.05    0.05    ...   0.05    0.05    0.05 ],
 [0.05    0.05    0.05    ...   4.0336285 0.05    0.05 ],
 ...
 [0.05    0.05    0.05    ...   0.05    0.05    0.05 ],
 [0.05    0.05    0.05    ...   0.05    0.05    0.05 ],
 [0.05    0.05    0.05    ...   0.05    0.05    3.05 ]]
```

Similarly, for each topic, we display the top 10 terms based on their ranks as follows:

```
>>> terms = count_vector.get_feature_names()
>>> for topic_idx, topic in enumerate(lda.components_):
...     print("Topic {}:".format(topic_idx))
...     print(" ".join([terms[i] for i in
...                   topic.argsort()[-10:]]))
Topic 0:
atheist doe ha believe say jesus people christian wa god
Topic 1:
moment just adobe want know ha wa hacker article radius
Topic 2:
center point ha wa available research computer data graphic hst
Topic 3:
objective argument just thing doe people wa think say article
Topic 4:
time like brian ha good life want know just wa
Topic 5:
computer graphic think know need university just article wa like
Topic 6:
```

```
topic 6:  
free program color doe use version gif jpeg file image  
Topic 7:  
gamma ray did know university ha just like article wa  
Topic 8:  
tool ha processing using data software color program bit image  
Topic 9:  
apr men know ha think woman just university article wa  
Topic 10:  
jpl propulsion mission april mar jet command data spacecraft wa  
Topic 11:  
russian like ha university redesign point option article space s  
Topic 12:  
ha van book star material physicist universe physical theory wa  
Topic 13:  
bank doe book law wa article rushdie muslim islam islamic  
Topic 14:  
think gopher routine point polygon book university article know  
Topic 15:  
ha rocket new lunar mission satellite shuttle nasa launch space  
Topic 16:  
want right article ha make like just think people wa  
Topic 17:  
just light space henry wa like zoology sky article toronto  
Topic 18:  
comet venus solar moon orbit planet earth probe ha wa  
Topic 19:  
site format image mail program available ftp send file graphic
```



There are a number of interesting topics that we just mined, for instance, computer graphics-related topics, such as 2, 5, 6, 8, and 19, space-related ones, such as 10, 11, 12, and 15, and religion-related ones, such as 0 and 13. There are also topics involving noise, for example, 9 and 16, which may require some imagination to interpret. Again, this is not surprising at all, since LDA, or topic modeling in general, as mentioned earlier, is a kind of free-form learning.

Summary

The project in this chapter was about finding hidden similarity underneath newsgroups data, be it semantic groups, themes, or word clouds. We started with what unsupervised learning does and the typical types of unsupervised learning algorithms. We then introduced unsupervised learning clustering and studied a popular clustering algorithm, k-means, in detail.

We also talked about tf-idf as a more efficient feature extraction tool for text data. After that, we performed k-means clustering on the newsgroups data and obtained four meaningful clusters. After examining the key terms in each resulting cluster, we went straight to extracting representative terms among original documents using topic modeling techniques. Two powerful topic modeling approaches, NMF and LDA, were discussed and implemented. Finally, we had some fun interpreting the topics we obtained from both methods.

Hitherto, we have covered all the main categories of unsupervised learning, including dimensionality reduction, clustering, and topic modeling, which is also dimensionality reduction in a way.

In the next chapter, we will review what you have learned so far in this book and provide best practices of real-world machine learning. The chapter aims to foolproof your learning and get you ready for the entire machine learning workflow and productionization. This will be a wrap-up of general machine learning techniques before we move on to more complex topics in the final three chapters.

Exercises

1. Perform k-means clustering on newsgroups data using different values of k , or use the Elbow method to find the optimal one. See if you get better grouping results.
2. Try different numbers of topics, in either NMF or LDA, and see which one produces more meaningful topics in the end. This should be a fun exercise.

3. Can you experiment with NMF or LDA on the entire 20 groups of newsgroups data? Are the resulting topics full of noise or gems?

11

Machine Learning Best Practices

After working on multiple projects covering important machine learning concepts, techniques, and widely used algorithms, you have a broad picture of the machine learning ecosystem, as well as solid experience in tackling practical problems using machine learning algorithms and Python.

However, there will be issues once we start working on projects from scratch in the real world. This chapter aims to get us ready for it with 21 best practices to follow throughout the entire machine learning solution workflow.

We will cover the following topics in this chapter:

- Machine learning solution workflow
- Tasks in the data preparation stage
- Tasks in the training sets generation stage
- Tasks in the algorithm training, evaluation, and selection stage
- Tasks in the system deployment and monitoring stage
- Best practices in the data preparation stage
- Best practices in the training sets generation stage
- Word embedding
- Best practices in the model training, evaluation, and selection stage
- Best practices in the system deployment and monitoring stage

Machine learning solution workflow

In general, the main tasks involved in solving a machine learning problem can be summarized into four areas, as follows:

- Data preparation
- Training sets generation
- Model training, evaluation, and selection
- Deployment and monitoring

Starting from data sources and ending with the final machine learning system, a machine learning solution basically follows the paradigm shown here:

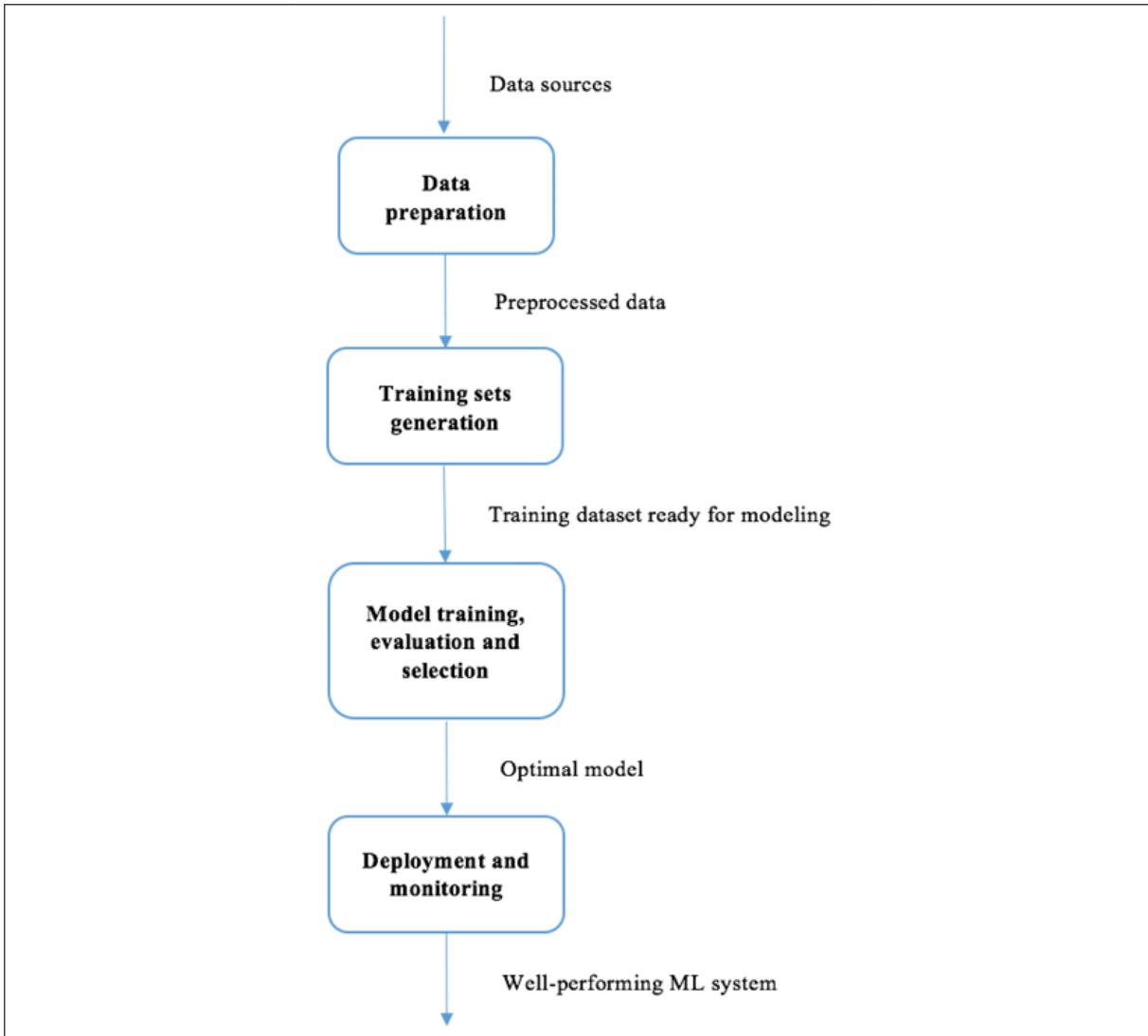


Figure 11.1: The life cycle of a machine learning solution

In the following sections, we will be learning about the typical tasks, common challenges, and best practices for each of these four stages.

Best practices in the data preparation stage

No machine learning system can be built without data. Therefore, **data collection** should be our first focus.

Best practice 1 – Completely understanding the project goal

Before starting to collect data, we should make sure that the goal of the project and the business problem is completely understood, as this will guide us on what data sources to look into, and where sufficient domain knowledge and expertise is also required. For example, in a previous chapter, *Chapter 7, Predicting Stock Prices with Regression Algorithms*, our goal was to predict the future prices of the DJIA index, so we first collected data of its past performance, instead of the past performance of an irrelevant European stock. In *Chapter 4, Predicting Online Ad Click-Through with Tree-Based Algorithms*, for example, the business problem was to optimize advertising targeting efficiency measured by click-through rate, so we collected the clickstream data of who clicked or did not click on what ad on what page, instead of merely using how many ads were displayed in a web domain.

Best practice 2 – Collecting all fields that are relevant

With a set goal in mind, we can narrow down potential data sources to investigate. Now the question becomes: is it necessary to collect the data of all fields available in a data source, or is a subset of attributes enough? It would be perfect if we knew in advance which attributes were key indicators or key predictive factors. However, it is in fact very difficult to ensure that the attributes hand-picked by a domain expert will yield the best prediction results. Hence, for each data source, it is recommended to collect all of the fields that are related to the project, especially in cases where recollecting the data is time-consuming, or even impossible.

For example, in the stock price prediction example, we collected the data of all fields including **open**, **high**, **low**, and **volume**, even though we were initially not certain of how useful **high** and **low** predictions would be.

Retrieving the stock data is quick and easy, however. In another example, if we ever want to collect data ourselves by scraping online articles for topic classification, we should store as much information as possible. Otherwise, if any piece of information is not collected but is later found to be valuable, such as hyperlinks in an article, the article might already have been removed from the web page; if it still exists, rescraping those pages can be costly.

After collecting the datasets that we think are useful, we need to assure the data quality by inspecting its **consistency** and **completeness**. Consistency refers to how the distribution of data is changing over time. Completeness means how much data is present across fields and samples. They are explained in detail in the following two practices.

Best practice 3 – Maintaining the consistency of field values

In a dataset that already exists, or in one we collect from scratch, often we see different values representing the same meaning. For example, we see *American*, *US*, and *U.S.A* in the country field, and *male* and *M* in the gender field. It is necessary to unify or standardize the values in a field. For example, we should keep only the three options of *M*, *F*, and *gender-diverse* in the gender field, and replace other alternative values. Otherwise, it will mess up the algorithms in later stages as different feature values will be treated differently even if they have the same meaning. It is also a great practice to keep track of what values are mapped to the default value of a field.

In addition, the format of values in the same field should also be consistent. For instance, in the *age* field, there could be true age values, such as 21 and 35, and incorrect age values, such as 1990 and 1978; in the *rating* field, both cardinal numbers and English numerals could be found, such as 1, 2,

and 3, and *one*, *two*, and *three*. Transformation and reformatting should be conducted in order to ensure data consistency.

Best practice 4 – Dealing with missing data

Due to various reasons, datasets in the real world are rarely completely clean and often contain missing or corrupted values. They are usually presented as blanks, *Null*, *-1*, *999999*, *unknown*, or any other placeholder. Samples with missing data not only provide incomplete predictive information, but also confuse the machine learning model as it cannot tell whether *-1* or *unknown* holds a meaning. It is important to pinpoint and deal with missing data in order to avoid jeopardizing the performance of models in the later stages.

Here are three basic strategies that we can use to tackle the missing data issue:

- Discarding samples containing any missing value.
- Discarding fields containing missing values in any sample.
- Inferring the missing values based on the known part from the attribute. This process is called **missing data imputation**. Typical imputation methods include replacing missing values with the mean or median value of the field across all samples, or the most frequent value for categorical data.

The first two strategies are simple to implement; however, they come at the expense of the data lost, especially when the original dataset is not large enough. The third strategy doesn't abandon any data, but does try to fill in the blanks.

Let's look at how each strategy is applied in an example where we have a dataset (age, income) consisting of six samples: (30, 100), (20, 50), (35, *unknown*), (25, 80), (30, 70), and (40, 60):

- If we process this dataset using the first strategy, it becomes (30, 100), (20, 50), (25, 80), (30, 70), and (40, 60).

- If we employ the second strategy, the dataset becomes (30), (20), (35), (25), (30), and (40), where only the first field remains.
- If we decide to complete the unknown value instead of skipping it, the sample (35, *unknown*) can be transformed into (35, 72) with the mean of the rest of the values in the second field, or (35, 70), with the median value in the second field.

In scikit-learn, the `SimpleImputer` class (<https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html>) provides a nicely written imputation transformer. We herein use it for the following small example:

```
>>> import numpy as np
>>> from sklearn.impute import SimpleImputer
```

Represent the unknown value by `np.nan` in `numpy`, as detailed in the following:

```
>>> data_origin = [[30, 100],
...                 [20, 50],
...                 [35, np.nan],
...                 [25, 80],
...                 [30, 70],
...                 [40, 60]]
```

Initialize the imputation transformer with the mean value and obtain the mean value from the original data:

```
>>> imp_mean = SimpleImputer(missing_values=np.nan, strategy='me
>>> imp_mean.fit(data_origin)
```

Complete the missing value as follows:

```
>>> data_mean_imp = imp_mean.transform(data_origin)
>>> print(data_mean_imp)
[[ 30. 100.]
```

```
[ 20. 50.]  
[ 35. 72.]  
[ 25. 80.]  
[ 30. 70.]  
[ 40. 60.]]
```

Similarly, initialize the imputation transformer with the median value, as detailed in the following:

```
>>> imp_median = SimpleImputer(missing_values=np.nan, strategy='median')  
>>> imp_median.fit(data_origin)  
>>> data_median_imp = imp_median.transform(data_origin)  
>>> print(data_median_imp)  
[[ 30. 100.]  
 [ 20. 50.]  
 [ 35. 70.]  
 [ 25. 80.]  
 [ 30. 70.]  
 [ 40. 60.]]
```

When new samples come in, the missing values (in any attribute) can be imputed using the trained transformer, for example, with the mean value, as shown here:

```
>>> new = [[20, np.nan],  
...          [30, np.nan],  
...          [np.nan, 70],  
...          [np.nan, np.nan]]  
>>> new_mean_imp = imp_mean.transform(new)  
>>> print(new_mean_imp)  
[[ 20. 72.]  
 [ 30. 72.]  
 [ 30. 70.]  
 [ 30. 72.]]
```

Note that `30` in the age field is the mean of those six age values in the original dataset.

Now that we have seen how imputation works, as well as its implementation, let's explore how the strategy of imputing missing values

and discarding missing data affects the prediction results through the following example:

1. First, we load the diabetes dataset, as shown here:

```
>>> from sklearn import datasets  
>>> dataset = datasets.load_diabetes()  
>>> X_full, y = dataset.data, dataset.target
```

2. Simulate a corrupted dataset by adding 25% missing values:

```
>>> m, n = X_full.shape  
>>> m_missing = int(m * 0.25)  
>>> print(m, m_missing)  
442 110
```

3. Randomly select the `m_missing` samples, as follows:

```
>>> np.random.seed(42)  
>>> missing_samples = np.array([True] * m_missing +  
                           [False] * (m - m_missing))  
>>> np.random.shuffle(missing_samples)
```

4. For each missing sample, randomly select 1 out of `n` features:

```
>>> missing_features = np.random.randint(low=0, high=n,  
                                         size=m_missing)
```

5. Represent missing values by `nan`, as shown here:

```
>>> X_missing = X_full.copy()  
>>> X_missing[np.where(missing_samples)[0], missing_features] = np.nan
```

6. Then we deal with this corrupted dataset by discarding the samples containing a missing value:

```
>>> X_rm_missing = X_missing[~missing_samples, :]  
>>> y_rm_missing = y[~missing_samples]
```

7. Measure the effects of using this strategy by estimating the averaged regression score, R^2 , with a regression forest model in a cross-

validation manner. Estimate R^2 on the dataset with the missing samples removed, as follows:

```
>>> from sklearn.ensemble import RandomForestRegressor
>>> from sklearn.model_selection import cross_val_score
>>> regressor = RandomForestRegressor(random_state=42,
                                         max_depth=10, n_estimators=100)
>>> score_rm_missing = cross_val_score(regressor, X_rm_missing,
                                         y_rm_missing).mean()
>>> print(f'Score with the data set with missing samples removed: {score_rm_missing:.2f}')
Score with the data set with missing samples removed: 0.38
```

- Now we approach the corrupted dataset differently by imputing missing values with the mean, as shown here:

```
>>> imp_mean = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> X_mean_imp = imp_mean.fit_transform(X_missing)
```

- Similarly, measure the effects of using this strategy by estimating the averaged R^2 , as follows:

```
>>> regressor = RandomForestRegressor(random_state=42,
                                         max_depth=10,
                                         n_estimators=100)
>>> score_mean_imp = cross_val_score(regressor, X_mean_imp,
                                         y).mean()
>>> print(f'Score with the data set with missing values replaced by mean: {score_mean_imp:.2f}')
Score with the data set with missing values replaced by mean: 0.42
```

- An imputation strategy works better than discarding in this case. So, how far is the imputed dataset from the original full one? We can check it again by estimating the averaged regression score on the original dataset, as follows:

```
>>> regressor = RandomForestRegressor(random_state=42,
                                         max_depth=10,
                                         n_estimators=500)
>>> score_full = cross_val_score(regressor, X_full, y).mean()
>>> print(f'Score with the full data set: {score_full:.2f}')
Score with the full data set: 0.42
```

It turns out that little information is compromised in the imputed dataset.

However, there is no guarantee that an imputation strategy always works better, and sometimes dropping samples with missing values can be more effective. Hence, it is a great practice to compare the performances of different strategies via cross-validation as we have done previously.

Best practice 5 – Storing large-scale data

With the ever-growing size of data, oftentimes we can't simply fit the data in our single local machine and need to store it on the cloud or distributed file systems. As this is mainly a book on machine learning with Python, we will just touch on some basic areas that you can look into. The two main strategies of storing big data are **scale-up** and **scale-out**:

- A scale-up approach increases storage capacity if data exceeds the current system capacity, such as by adding more disks. This is useful in fast-access platforms.
- In a scale-out approach, storage capacity grows incrementally with additional nodes in a storage cluster. Apache Hadoop (<https://hadoop.apache.org/>) is used to store and process big data in scale-out clusters, where data is spread across hundreds or even thousands of nodes. Also, there are cloud-based distributed file services, such as S3 in Amazon Web Services (<https://aws.amazon.com/s3/>), Google Cloud Storage in Google Cloud (<https://cloud.google.com/storage/>), and Storage in Microsoft Azure (<https://azure.microsoft.com/en-us/services/storage/>). They are massively scalable and are designed for secure and durable storage.

With well-prepared data, it is safe to move on to the training sets generation stage. Let's see the next section.

Best practices in the training sets generation stage

Typical tasks in this stage can be summarized into two major categories: **data preprocessing** and **feature engineering**.

To begin, data preprocessing usually involves categorical feature encoding, feature scaling, feature selection, and dimensionality reduction.

Best practice 6 – Identifying categorical features with numerical values

In general, categorical features are easy to spot, as they convey qualitative information, such as risk level, occupation, and interests. However, it gets tricky if the feature takes on a discrete and countable (limited) number of numerical values, for instance, 1 to 12 representing months of the year, and 1 and 0 indicating true and false. The key to identifying whether such a feature is categorical or numerical is whether it provides a mathematical or ranking implication; if so, it is a numerical feature, such as a product rating from 1 to 5; otherwise, it is categorical, such as the month, or day of the week.

Best practice 7 – Deciding whether to encode categorical features

If a feature is considered categorical, we need to decide whether we should encode it. It depends on what prediction algorithm(s) we will use in later stages. Naïve Bayes and tree-based algorithms can directly work with categorical features, while other algorithms in general cannot, in which case encoding is essential.

As the output of the feature generation stage is the input of the model training stage, *steps taken in the feature generation stage should be compatible with the prediction algorithm*. Therefore, we should look at the two stages of feature generation and predictive model training as a whole, instead of two isolated components. The next two practical tips also reinforce this point.

Best practice 8 – Deciding whether to select features, and if so, how to do so

You have seen in *Chapter 5, Predicting Online Ad Click-Through with Logistic Regression*, how feature selection can be performed using L1-based regularized logistic regression and random forest. The benefits of feature selection include the following:

- Reducing the training time of prediction models as redundant or irrelevant features are eliminated
- Reducing overfitting for the same preceding reason
- Likely improving performance, as prediction models will learn from data with more significant features

Note we used the word *likely* because there is no absolute certainty that feature selection will increase prediction accuracy. It is therefore good practice to compare the performances of conducting feature selection and not doing so via cross-validation. For example, by executing the following steps, we can measure the effects of feature selection by estimating the averaged classification accuracy with an `svc` model in a cross-validation manner:

1. First, we load the handwritten digits dataset from scikit-learn, as follows:

```
>>> from sklearn.datasets import load_digits  
>>> dataset = load_digits()  
>>> X, y = dataset.data, dataset.target  
>>> print(X.shape)  
(1797, 64)
```

2. Next, estimate the accuracy of the original dataset, which is 64-dimensional, as detailed here:

```
>>> from sklearn.svm import SVC
>>> from sklearn.model_selection import cross_val_score
>>> classifier = SVC(gamma=0.005, random_state=42)
>>> score = cross_val_score(classifier, X, y).mean()
>>> print(f'Score with the original data set: {score:.2f}')
Score with the original data set: 0.90
```

3. Then conduct feature selection based on random forest and sort the features based on their importance scores:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> random_forest = RandomForestClassifier(n_estimators=100)
>>> random_forest.fit(X, y)
>>> feature_sorted =
    np.argsort(random_forest.feature_importances_)
```

4. Now select a different number of top features to construct a new dataset, and estimate the accuracy on each dataset, as follows:

```
>>> K = [10, 15, 25, 35, 45]
>>> for k in K:
...     top_K_features = feature_sorted[-k:]
...     X_k_selected = X[:, top_K_features]
...     # Estimate accuracy on the data set with k
...     selected features
...     classifier = SVC(gamma=0.005)
...     score_k_features = cross_val_score(classifier,
...                                         X_k_selected, y).mean()
...     print(f'Score with the dataset of top {k} features:
...           {score_k_features:.2f}')
...
Score with the dataset of top 10 features: 0.86
Score with the dataset of top 15 features: 0.92
Score with the dataset of top 25 features: 0.95
Score with the dataset of top 35 features: 0.93
Score with the dataset of top 45 features: 0.90
```

If we use the top 25 features selected by the random forest, the SVM classification performance can increase from 0.9 to 0.95.

Best practice 9 – Deciding whether to reduce dimensionality, and if so, how to do so

Feature selection and dimensionality are different in the sense that the former chooses features from the original data space, while the latter does so from a projected space from the original space. Dimensionality reduction has the following advantages that are similar to feature selection, as follows:

- Reducing the training time of prediction models, as redundant or correlated features are merged into new ones
- Reducing overfitting for the same reason
- Likely improving performance, as prediction models will learn from data with less redundant or correlated features

Again, it is not guaranteed that dimensionality reduction will yield better prediction results. In order to examine its effects, integrating dimensionality reduction in the model training stage is recommended. Reusing the preceding handwritten digits example, we can measure the effects of **principal component analysis (PCA)**-based dimensionality reduction, where we keep a different number of top components to construct a new dataset, and estimate the accuracy on each dataset:

```
>>> from sklearn.decomposition import PCA
>>> # Keep different number of top components
>>> N = [10, 15, 25, 35, 45]
>>> for n in N:
...     pca = PCA(n_components=n)
...     X_n_kept = pca.fit_transform(X)
...     # Estimate accuracy on the data set with top n component
...     classifier = SVC(gamma=0.005)
...     score_n_components =
...         cross_val_score(classifier, X_n_kept, y).mean
...     print(f'Score with the dataset of top {n} components: {score_n_components}')
```

```
{score_n_components:.2f}"))
Score with the dataset of top 10 components: 0.94
Score with the dataset of top 15 components: 0.95
Score with the dataset of top 25 components: 0.93
Score with the dataset of top 35 components: 0.91
Score with the dataset of top 45 components: 0.90
```

If we use the top 15 features generated by PCA, the SVM classification performance can increase from 0.9 to 0.95.

Best practice 10 – Deciding whether to rescale features

As seen in *Chapter 7, Predicting Stock Prices with Regression Algorithms*, and *Chapter 8, Predicting Stock Prices with Artificial Neural Networks*, SGD-based linear regression, SVR, and the neural network model require features to be standardized by removing the mean and scaling to unit variance. So, when is feature scaling needed and when is it not?

In general, Naïve Bayes and tree-based algorithms are not sensitive to features at different scales, as they look at each feature independently.

In most cases, an algorithm that involves any form of distance (or separation in spaces) of samples in learning requires scaled/standardized inputs, such as SVC, SVR, k-means clustering, and **k-nearest neighbors (KNN)** algorithms. Feature scaling is also a must for any algorithm using SGD for optimization, such as linear or logistic regression with gradient descent, and neural networks.

We have so far covered tips regarding data preprocessing and will next discuss best practices of feature engineering as another major aspect of training sets generation. We will do so from two perspectives.

Best practice 11 – Performing feature engineering with domain expertise

If we are lucky enough to possess sufficient domain knowledge, we can apply it in creating domain-specific features; we utilize our business experience and insights to identify what is in the data and to formulate new data that correlates to the prediction target. For example, in *Chapter 7, Predicting Stock Prices with Regression Algorithms*, we designed and constructed feature sets for the prediction of stock prices based on factors that investors usually look at when making investment decisions.

While particular domain knowledge is required, sometimes we can still apply some general tips in this category. For example, in fields related to customer analytics, such as marketing and advertising, the time of the day, day of the week, and month are usually important signals. Given a data point with the value *2020/09/01* in the date column and *14:34:21* in the time column, we can create new features including *afternoon*, *Tuesday*, and *September*. In retail, information covering a period of time is usually aggregated to provide better insights. The number of times a customer visits a store for the past three months, or the average number of products purchased weekly for the previous year, for instance, can be good predictive indicators for customer behavior prediction.

Best practice 12 – Performing feature engineering without domain expertise

If, unfortunately, we have very little domain knowledge, how can we generate features? Don't panic. There are several generic approaches that you can follow, such as binarization, discretization, interaction, and polynomial transformation.

Binarization

This is the process of converting a numerical feature to a binary one with a preset threshold. For example, in spam email detection, for the feature (or term) *prize*, we can generate a new feature `whether_term_prize_occurs`: any term frequency value greater than 1 becomes 1; otherwise, it is 0. The

feature *number of visits per week* can be used to produce a new feature, `is_frequent_visitor`, by judging whether the value is greater than or equal to 3. We implement such binarization using scikit-learn, as follows:

```
>>> from sklearn.preprocessing import Binarizer
>>> X = [[4], [1], [3], [0]]
>>> binarizer = Binarizer(threshold=2.9)
>>> X_new = binarizer.fit_transform(X)
>>> print(X_new)
[[1]
 [0]
 [1]
 [0]]
```

Discretization

This is the process of converting a numerical feature to a categorical feature with limited possible values. Binarization can be viewed as a special case of discretization. For example, we can generate an *age group* feature: "18-24" for age from 18 to 24, "25-34" for age from 25 to 34, "34-54", and "55+".

Interaction

This includes the sum, multiplication, or any operations of two numerical features, and joint condition check of two categorical features. For example, *the number of visits per week* and *the number of products purchased per week* can be used to generate *the number of products purchased per visit* feature; *interest and occupation*, such as *sports* and *engineer*, can form *occupation AND interest*, such as *engineer interested in sports*.

Polynomial transformation

This is the process of generating polynomial and interaction features. For two features, a and b , the two degrees of polynomial features generated are a^2 , ab , and b^2 . In scikit-learn, we can use the `PolynomialFeatures` class (<https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html>) to perform polynomial transformation, as follows:

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> X = [[2, 4],
...       [1, 3],
...       [3, 2],
...       [0, 3]]
>>> poly = PolynomialFeatures(degree=2)
>>> X_new = poly.fit_transform(X)
>>> print(X_new)
[[ 1.  2.  4.  4.  8.  16.]
 [ 1.  1.  3.  1.  3.  9.]
 [ 1.  3.  2.  9.  6.  4.]
 [ 1.  0.  3.  0.  0.  9.]]
```

Note the resulting new features consist of 1 (bias, intercept), a , b , a^2 , ab , and b^2 .

Best practice 13 – Documenting how each feature is generated

We have covered the rules of feature engineering with domain knowledge, and in general, there is one more thing worth noting: documenting how each feature is generated. It sounds trivial, but oftentimes we just forget about how a feature is obtained or created. We usually need to go back to this stage after some failed trials in the model training stage and attempt to create more features with the hope of improving performance. We have to be clear on what and how features are generated, in order to remove those that do not quite work out, and to add new ones that have more potential.

Best practice 14 – Extracting features from text data

We will start with a traditional approach to extract features from text, tf, and tf-idf. Then we will continue with a modern approach: word embedding. And finally, we will look at word embedding using pre-trained models.

Tf and tf-idf

We have worked intensively with text data in *Chapter 9, Mining the 20 Newsgroups Dataset with Text Analysis Techniques*, and *Chapter 10, Discovering Underlying Topics in the Newsgroups Dataset with Clustering and Topic Modeling*, where we extracted features from text based on **term frequency (tf)** and **term frequency-inverse document frequency (tf-idf)**. Both methods consider each document of words (terms) a collection of words, or a **bag of words (BoW)**, disregarding the order of the words, but keeping multiplicity. A tf approach simply uses the counts of tokens, while tf-idf extends tf by assigning each tf a weighting factor that is inversely proportional to the document frequency. With the idf factor incorporated, tf-idf diminishes the weight of common terms (such as *get*, *make*) that occur frequently, and emphasizes terms that rarely occur but convey important meaning. Hence, oftentimes features extracted from tf-idf are more representative than those from tf.

As you may remember, a document is represented by a very sparse vector where only present terms have non-zero values. And the vector's dimensionality is usually high, which is determined by the size of the vocabulary and the number of unique terms. Also, such a one-hot encoding approach treats each term as an independent item and does not consider the relationship across words (referred to as "context" in linguistics).

Word embedding

On the contrary, another approach, called **word embedding**, is able to capture the meanings of words and their context. In this approach, a word is represented by a vector of float numbers. Its dimensionality is a lot lower than the size of the vocabulary and is usually several hundred only. For example, the word **machine** can be represented as [1.4, 2.1, 10.3, 0.2, 6.81]. So, how can we embed a word into a vector? One solution is word2vec, which trains a shallow neural network to predict a word given the other words around it (called **Continuous Bag of Words (CBOW)**), or to predict the other words around a word (called **skip-gram**). The coefficients of the trained neural network are the embedding vectors for the corresponding words.

Given the sentence *I love reading Python machine learning by example* in a corpus, and 5 as the size of the word window, we can have the following training sets for the CBOW neural network:

Input of neural network	Output of neural network
(I, love, python, machine)	(reading)
(love, reading, machine, learning)	(python)
(reading, python, learning, by)	(machine)
(python, machine, by, example)	(learning)

Table 11.1: Input and output of the neural network for CBOW

Of course, the inputs and outputs of the neural network are one-hot encoding vectors, where values are either 1 for present words, or 0 for absent words. And we can have millions of training samples constructed from a corpus, sentence by sentence. After the network is trained, the weights that connect the input layer and hidden layer embed individual input words. A skip-gram-based neural network embeds words in a similar way. But its input and output are an inverse version of CBOW. Given the

same sentence, *I love reading Python machine learning by example*, and 5 as the size of the word window, we can have the following training sets for the skip-gram neural network:

Input of neural network	Output of neural network
(reading)	(i)
(reading)	(love)
(reading)	(python)
(reading)	(machine)
(python)	(love)
(python)	(reading)
(python)	(machine)
(python)	(learning)
(machine)	(reading)
(machine)	(python)
(machine)	(learning)
(machine)	(by)
(learning)	(python)

(learning)	(machine)
(learning)	(by)
(learning)	(example)

Table 11.2: Input and output of the neural network for skip-gram

The embedding vectors are of real values, where each dimension encodes an aspect of meaning for the words in the vocabulary. This helps preserve the semantic information of the words, as opposed to discarding it as in the dummy one-hot encoding approach using tf or td-idf. An interesting phenomenon is that vectors from semantically similar words are proximate to each other in geometric space. For example, both the words *clustering* and *grouping* refer to unsupervised clustering in the context of machine learning, hence their embedding vectors are close together.

Word embedding with pre-trained models

Training a word embedding neural network can be time-consuming and computationally expensive. Fortunately, there are several big tech companies that have trained word embedding models on different kinds of corpora and open sourced them. We can simply use these **pre-trained** models to map words to vectors. Some popular pre-trained word embedding models are as follows:

Name	fasttext-wiki-news-subwords-300	
Corpus	Wikipedia 2017	
Vector size	300	
Vocabulary size	1 million	
File size	958 MB	
More information	https://fasttext.cc/docs/en/english-vectors.html	
<hr/>		
Name	glove-twitter-100	glove-twitter-25
Corpus	Twitter (2 billion tweets)	
Vector size	100	25
Vocabulary size	1.2 million	
File size	387 MB	104 MB
More information	https://nlp.stanford.edu/projects/glove/	
<hr/>		
Name	word2vec-google-news-300	
Corpus	Google News (about 100 billion words)	
Vector size	300	
Vocabulary size	3 million	
File size	1662 MB	
More information	https://code.google.com/archive/p/word2vec/	

Figure 11.2: Configurations of popular pre-trained word embedding models

Once we have embedding vectors for individual words, we can represent a document sample by averaging all of the vectors of present words in this document. The resulting vectors of document samples are then consumed by downstream predictive tasks, such as classification, similarity ranking in search engines, and clustering.

Now let's play around with `gensim`, a popular NLP package with powerful word embedding modules. If you have not installed the package in *Chapter*

9, *Mining the 20 Newsgroups Dataset with Text Analysis Techniques*, you can do so using `pip`.

First, we import the package and load a pre-trained model, `glove-twitter-25`, as follows:

```
>>> import gensim.downloader as api  
>>> model = api.load("glove-twitter-25")  
[=====] 100.0%  
104.8/104.8MB downloaded
```

You will see the process bar if you run this line of code. The `glove-twitter-25` model is one of the smallest ones so the download will not take very long.

We can obtain the embedding vector for a word (`computer`, for example), as follows:

```
>>> vector = model.wv['computer']  
>>> print('Word computer is embedded into:\n', vector)  
Word computer is embedded into:  
[ 0.64005 -0.019514 0.70148 -0.66123 1.1723 -0.58859 0.25917  
-0.81541 1.1708 1.1413 -0.15405 -0.11369 -3.8414 -0.87233  
0.47489 1.1541 0.97678 1.1107 -0.14572 -0.52013 -0.52234  
-0.92349 0.34651 0.061939 -0.57375 ]
```

The result is a 25-dimension float vector as expected.

We can also get the top 10 words that are most contextually relevant to `computer` using the `most_similar` method, as follows:

```
>>> similar_words = model.most_similar("computer")  
>>> print('Top ten words most contextually relevant to computer:  
        similar_words)  
Top ten words most contextually relevant to computer:  
[('camera', 0.907833456993103), ('cell', 0.891890287399292), ('
```

The result looks promising.

Finally, we demonstrate how to generate representing vectors for a document with a simple example, as follows:

```
>>> doc_sample = ['i', 'love', 'reading', 'python', 'machine',
                  'learning', 'by', 'example']
>>> import numpy as np
>>> doc_vector = np.mean([model.wv[word] for word in doc_sample],
                        axis=0)
>>> print('The document sample is embedded into:\n', doc_vector)
The document sample is embedded into:
[-0.17100249 0.1388764 0.10616798 0.200275 0.1159925 -0.1515975
 1.1621187 -0.4241785 0.2912 -0.28199488 -0.31453252 0.43692702
 -3.95395 -0.35544625 0.073975 0.1408525 0.20736426 0.17444688
 0.10602863 -0.04121475 -0.34942 -0.2736689 -0.47526264 -0.1184
 -0.16284864]
```

The resulting vector is the average of embedding vectors of eight input words.

In traditional NLP applications, such as text classification and topic modeling, tf, or td-idf, is still an outstanding solution for feature extraction. In more complicated areas, such as text summarization, machine translation, named entity resolution, question answering, and information retrieval, word embedding is used extensively and extracts far better features than the two traditional approaches.

Now that you have reviewed the best practices for data and feature generation, let's look at model training next.

Best practices in the model training, evaluation, and selection stage

Given a supervised machine learning problem, the first question many people ask is usually *what is the best classification or regression algorithm to solve it?* However, there is no one-size-fits-all solution, and no free lunch. No one could know which algorithm will work best before trying multiple ones and fine-tuning the optimal one. We will be looking into best practices around this in the following sections.

Best practice 15 – Choosing the right algorithm(s) to start with

Due to the fact that there are several parameters to tune for an algorithm, exhausting all algorithms and fine-tuning each one can be extremely time-consuming and computationally expensive. We should instead shortlist one to three algorithms to start with using the general guidelines that follow (note we herein focus on classification, but the theory transcends to regression, and there is usually a counterpart algorithm in regression).

There are several things we need to be clear about before shortlisting potential algorithms, as described in the following:

- The size of the training dataset
- The dimensionality of the dataset
- Whether the data is linearly separable
- Whether features are independent
- Tolerance and trade-off of bias and variance
- Whether online learning is required

Now, let's look at how we choose the right algorithm to start with, taking into account the aforementioned perspectives.

Naïve Bayes

This is a very simple algorithm. For a relatively small training dataset, if features are independent, Naïve Bayes will usually perform well. For a

large dataset, Naïve Bayes will still work well as feature independence can be assumed in this case, regardless of the truth. The training of Naïve Bayes is usually faster than any other algorithm due to its computational simplicity. However, this may lead to a high bias (but low variance).

Logistic regression

This is probably the most widely used classification algorithm, and the first algorithm that a machine learning practitioner usually tries when given a classification problem. It performs well when data is linearly separable or approximately **linearly separable**. Even if it is not linearly separable, it might be possible to convert the linearly non-separable features into separable ones and apply logistic regression afterward.

In the following instance, data in the original space is not linearly separable, but it becomes separable in a transformed space created from the interaction of two features:

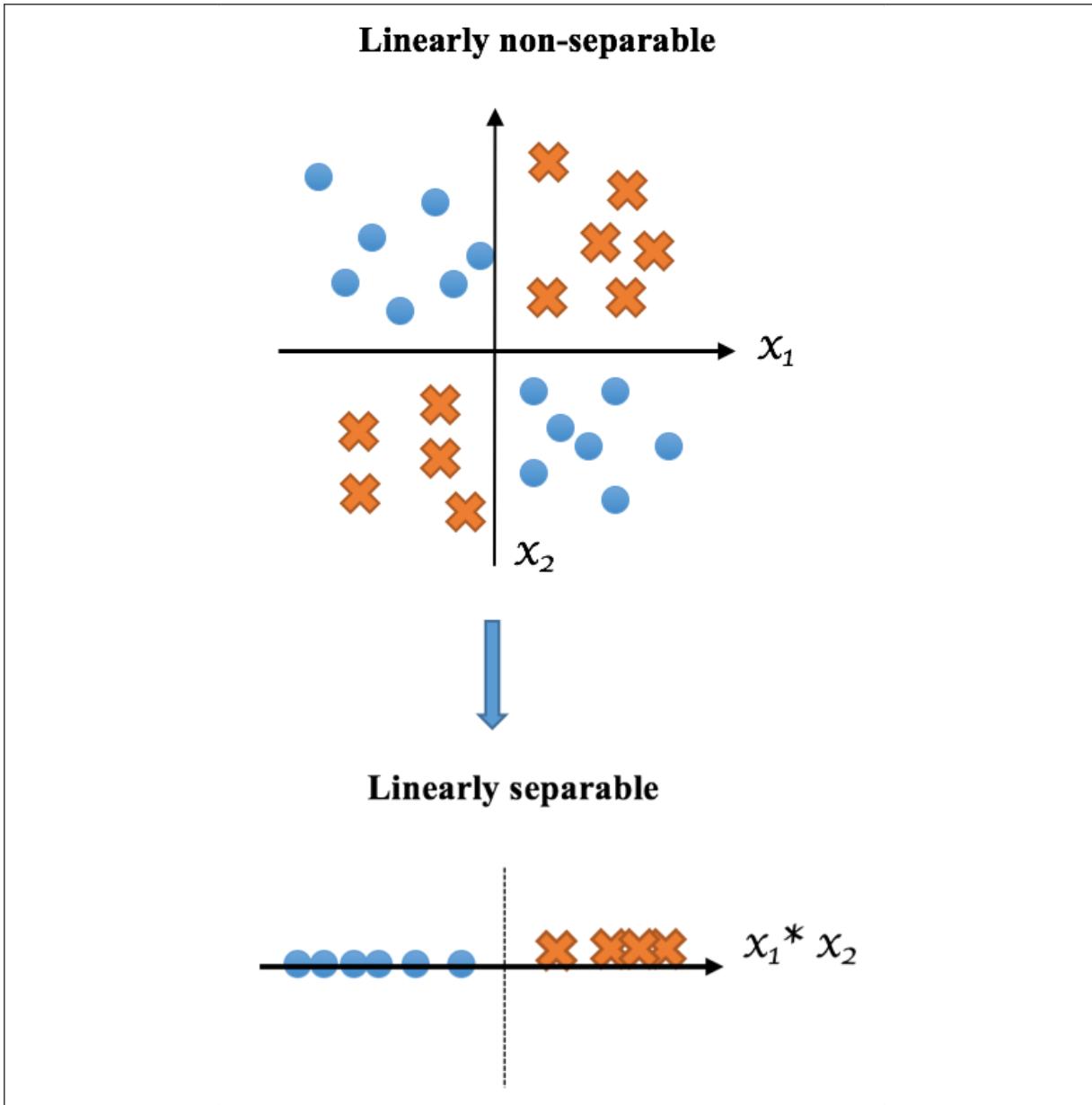


Figure 11.3: Transforming features from linearly non-separable to separable

Also, logistic regression is extremely scalable to large datasets with SGD optimization, which makes it efficient in solving big data problems. Plus, it makes online learning feasible. Although logistic regression is a low-bias, high-variance algorithm, we overcome the potential overfitting by adding L1, L2, or a mix of the two regularizations.

SVM

This is versatile enough to adapt to the linear separability of data. For a separable dataset, SVM with linear kernel performs comparably to logistic regression. Beyond this, SVM also works well for a non-separable dataset if equipped with a non-linear kernel, such as RBF. For a high-dimensional dataset, the performance of logistic regression is usually compromised, while SVM still performs well. A good example of this can be in news classification, where the feature dimensionality is in the tens of thousands. In general, very high accuracy can be achieved by SVM with the right kernel and parameters. However, this might be at the expense of intense computation and high memory consumption.

Random forest (or decision tree)

The linear separability of the data does not matter to the algorithm, and it works directly with categorical features without encoding, which provides great ease of use. Also, the trained model is very easy to interpret and explain to non-machine learning practitioners, which cannot be achieved with most other algorithms. Additionally, random forest boosts the decision tree algorithm, which can reduce overfitting by ensembling a collection of separate trees. Its performance is comparable to SVM, while fine-tuning a random forest model is less difficult compared to SVM and neural networks.

Neural networks

These are extremely powerful, especially with the development of deep learning. However, finding the right topology (layers, nodes, activation functions, and so on) is not easy, not to mention the time-consuming model of training and tuning. Hence, they are not recommended as an algorithm to start with for general machine learning problems. However, for computer vision and many NLP tasks, the neural network is still the go-to model.

Best practice 16 – Reducing overfitting

We touched on ways to avoid overfitting when discussing the pros and cons of algorithms in the last practice. We herein formally summarize them, as follows:

- **Cross-validation:** A good habit that we have built over all of the chapters in this book.
- **Regularization:** This adds penalty terms to reduce the error caused by fitting the model perfectly on the given training set.
- **Simplification, if possible:** The more complex the model is, the higher chance of overfitting. Complex models include a tree or forest with excessive depth, a linear regression with a high degree of polynomial transformation, and an SVM with a complicated kernel.
- **Ensemble learning:** This involves combining a collection of weak models to form a stronger one.

So, how can we tell whether a model suffers from overfitting, or the other extreme, underfitting? Let's see the next section.

Best practice 17 – Diagnosing overfitting and underfitting

A **learning curve** is usually used to evaluate the bias and variance of a model. A learning curve is a graph that compares the cross-validated training and testing scores over a given number of training samples.

For a model that fits well on the training samples, the performance of the training samples should be beyond what's desired. Ideally, as the number of training samples increases, the model performance on the testing samples will improve; eventually, the performance on the testing samples will become close to that on the training samples.

When the performance on the testing samples converges at a value much lower than that of the training performance, overfitting can be concluded. In this case, the model fails to generalize to instances that have not been seen.

For a model that does not even fit well on the training samples, underfitting is easily spotted: both performances on the training and testing samples are below the desired performance in the learning curve.

Here is an example of the learning curve in an ideal case:

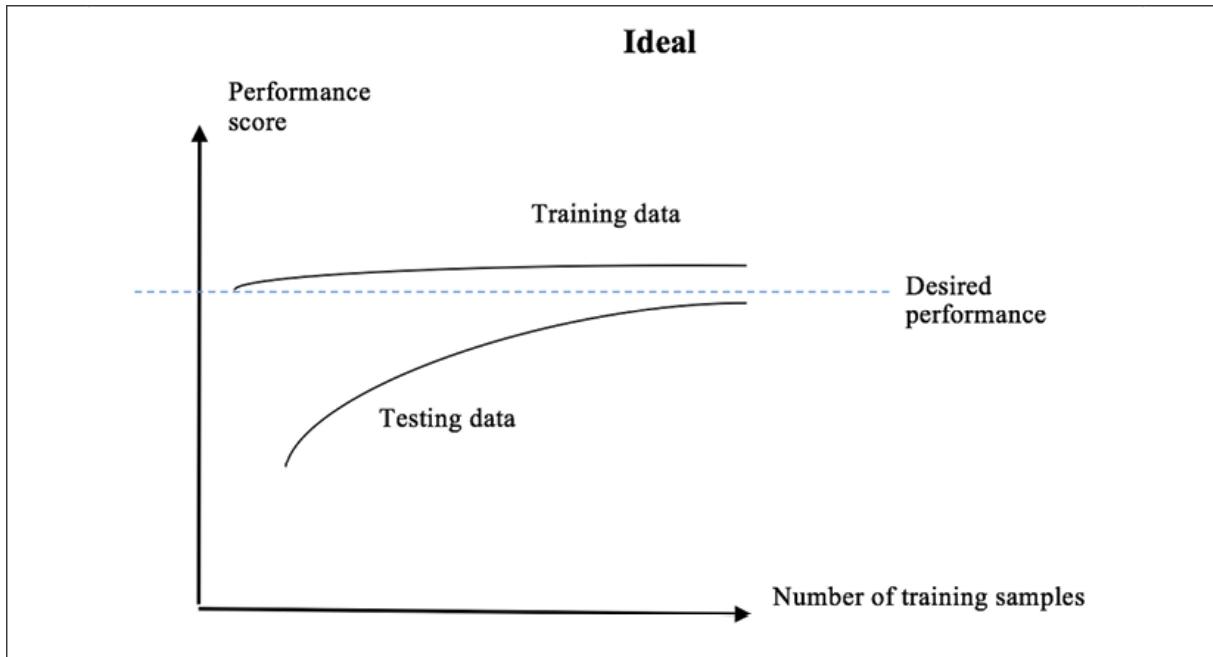


Figure 11.4: Ideal learning curve

An example of the learning curve for an overfitted model is shown in the following diagram:

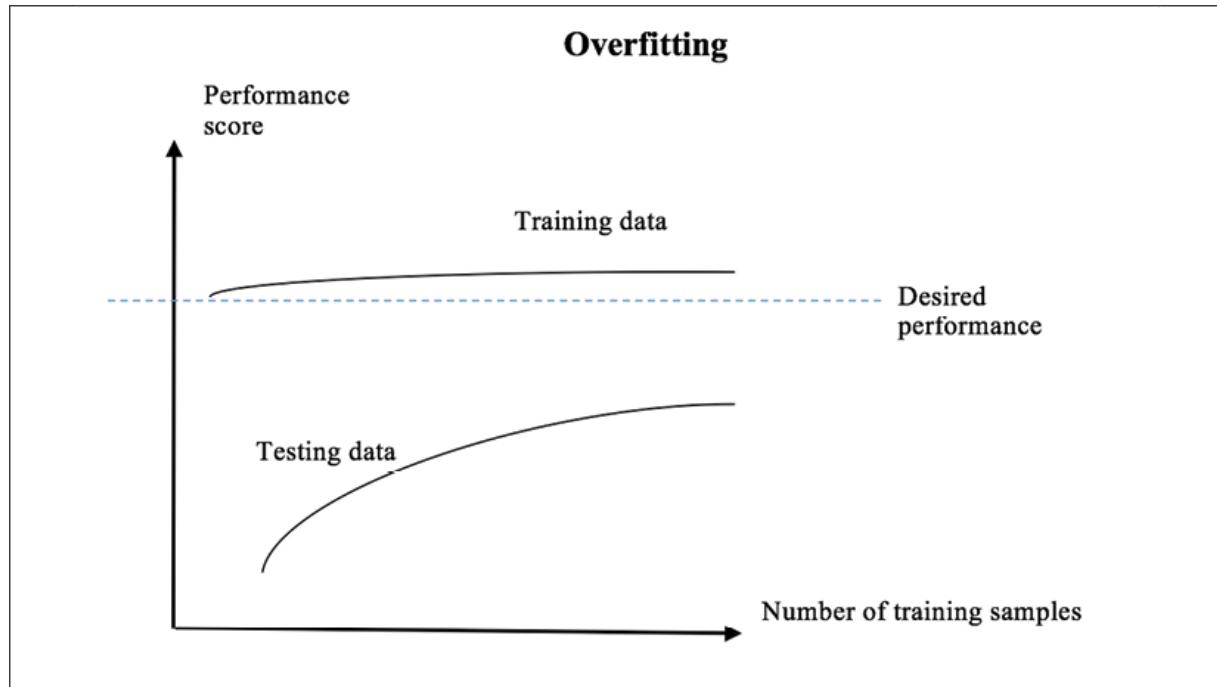


Figure 11.5: Overfitting learning curve

The learning curve for an underfitted model may look like the following diagram:

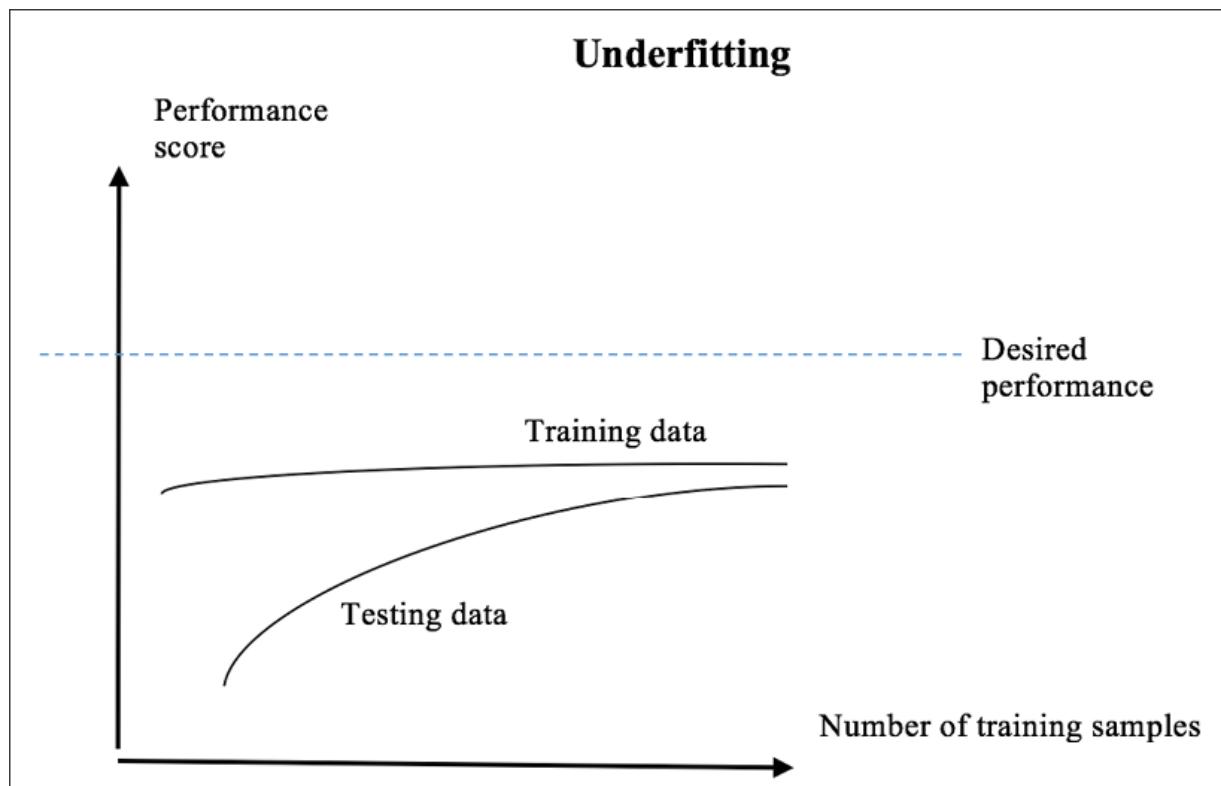


Figure 11.6: Underfitting learning curve

To generate the learning curve, you can utilize the `learning_curve` module (https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.learning_curve.html#sklearn.model_selection.learning_curve) from scikit-learn, and the `plot_learning_curve` function defined in https://scikit-learn.org/stable/auto_examples/model_selection/plot_learning_curve.html.

Best practice 18 – Modeling on large-scale datasets

We have gained experience working with large datasets in *Chapter 6, Scaling Up Prediction to Terabyte Click Logs*. There are a few tips that can help you model on large-scale data more efficiently.

First, start with a small subset, for instance, a subset that can fit on your local machine. This can help speed up early experimentation. Obviously, you don't want to train on the entire dataset just to find out whether SVM or random forest works better. Instead, you can randomly sample data points and quickly run a few models on the selected set.

The second tip is choosing scalable algorithms, such as logistic regression, linear SVM, and SGD-based optimization. This is quite intuitive.

Once you figure out which model works best, you can fine-tune it using more data points and eventually train on the entire dataset. After that, don't forget to save the trained model. This is the third tip. Training on a large dataset takes a long time, which you would want to avoid redoing, if possible. We will explore saving and loading models in detail in *Best practice 19 – Saving, loading, and reusing models*, which is a part of the deployment and monitoring stage.

Best practices in the deployment and monitoring stage

After performing all of the processes in the previous three stages, we now have a well-established data preprocessing pipeline and a correctly trained prediction model. The last stage of a machine learning system involves saving those resulting models from previous stages and deploying them on new data, as well as monitoring their performance and updating the prediction models regularly.

Best practice 19 – Saving, loading, and reusing models

When machine learning is deployed, new data should go through the same data preprocessing procedures (scaling, feature engineering, feature selection, dimensionality reduction, and so on) as in the previous stages. The preprocessed data is then fed in the trained model. We simply cannot rerun the entire process and retrain the model every time new data comes in. Instead, we should save the established preprocessing models and trained prediction models after the corresponding stages have been completed. In deployment mode, these models are loaded in advance and are used to produce prediction results from the new data.

Saving and restoring models using pickle

This can be illustrated via the diabetes example, where we standardize the data and employ an `SVR` model, as follows:

```
>>> dataset = datasets.load_diabetes()
>>> X, y = dataset.data, dataset.target
>>> num_new = 30 # the last 30 samples as new data set
>>> X_train = X[:-num_new, :]
>>> y_train = y[:-num_new]
>>> X_new = X[-num_new:, :]
>>> y_new = y[-num_new:]
```

Preprocess the training data with scaling, as shown in the following commands:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> scaler.fit(X_train)
```

Now save the established standardizer, the `scaler` object with `pickle`, as follows:

```
>>> import pickle
>>> pickle.dump(scaler, open("scaler.p", "wb"))
```

This generates a `scaler.p` file.

Move on to training an `SVR` model on the scaled data, as follows:

```
>>> X_scaled_train = scaler.transform(X_train)
>>> from sklearn.svm import SVR
>>> regressor = SVR(C=20)
>>> regressor.fit(X_scaled_train, y_train)
```

Save the trained `regressor` object with `pickle`, as follows:

```
>>> pickle.dump(regressor, open("regressor.p", "wb"))
```

This generates a `regressor.p` file.

In the deployment stage, we first load the saved standardizer and the `regressor` object from the preceding two files, as follows:

```
>>> my_scaler = pickle.load(open("scaler.p", "rb"))
>>> my_regressor = pickle.load(open("regressor.p", "rb"))
```

Then we preprocess the new data using the standardizer and make a prediction with the `regressor` object just loaded, as follows:

```
>>> X_scaled_new = my_scaler.transform(X_new)
>>> predictions = my_regressor.predict(X_scaled_new)
```

Saving and restoring models in TensorFlow

I will also demonstrate how to save and restore models in TensorFlow as a bonus session in this section. As an example, we will train a simple logistic regression model on the cancer dataset, save the trained model, and reload it in the following steps:

1. Import the necessary TensorFlow modules and load the cancer dataset from scikit-learn:

```
>>> import tensorflow as tf
>>> from tensorflow import keras
>>> from sklearn import datasets
>>> cancer_data = datasets.load_breast_cancer()
>>> X = cancer_data.data
>>> Y = cancer_data.target
```

2. Build a simple logistic regression model using the Keras Sequential API, along with several specified parameters:

```
>>> learning_rate = 0.005
>>> n_iter = 10
>>> tf.random.set_seed(42)
>>> model = keras.Sequential([
...     keras.layers.Dense(units=1, activation='sigmoid')
... ])
>>> model.compile(loss='binary_crossentropy',
...                 optimizer=tf.keras.optimizers.Adam(learning_rate))
```

3. Train the TensorFlow model against the data:

```
>>> model.fit(X, Y, epochs=n_iter)
```

4. Display the model's architecture:

```
>>> model.summary()
Model: "sequential"

Layer (type)                  Output Shape             Params
=====
dense (Dense)                multiple               31
=====
Total params: 31
Trainable params: 31
Non-trainable params: 0
```

We will see if we can retrieve the same model later.

5. Hopefully, the previous steps look familiar to you. If not, feel free to review our TensorFlow implementation. Now we save the model to a

path:

```
>>> path = './model_tf'  
>>> model.save(path)
```

After this, you will see that a folder called `model_tf` is created. The folder contains the trained model's architecture, weights, and training configuration.

6. Finally, we load the model from the previous path and display the loaded model's path:

```
>>> new_model = tf.keras.models.load_model(path)  
>>> new_model.summary()  
Model: "sequential"  
  
Layer (type)          Output Shape       Parameters  
=====            ======           =====  
dense (Dense)        multiple           31  
  
Total params: 31  
Trainable params: 31  
Non-trainable params: 0
```

We just loaded back the exact same model.

Best practice 20 – Monitoring model performance

The machine learning system is now up and running. To make sure everything is on the right track, we need to conduct performance checks on a regular basis. To do so, besides making a prediction in real time, we should also record the ground truth at the same time.

Continuing with the diabetes example from earlier in the chapter, we conduct a performance check as follows:

```
>>> from sklearn.metrics import r2_score  
>>> print(f'Health check on the model, R^2: {r2_score(y_new,
```

```
predictions):.3f}')  
Health check on the model, R^2: 0.613
```

We should log the performance and set up an alert for any decayed performance.

Best practice 21 – Updating models regularly

If the performance is getting worse, chances are that the pattern of data has changed. We can work around this by updating the model. Depending on whether online learning is feasible or not with the model, the model can be modernized with the new set of data (online updating), or retrained completely with the most recent data.

Summary

The purpose of this chapter is to prepare you for real-world machine learning problems. We started with the general workflow that a machine learning solution follows: data preparation, training sets generation, algorithm training, evaluation and selection, and finally, system deployment and monitoring. We then went in depth through the typical tasks, common challenges, and best practices for each of these four stages.

Practice makes perfect. The most important best practice is practice itself. Get started with a real-world project to deepen your understanding and apply what you have learned so far.

In the next chapter, we will discuss categorizing images of clothing using Convolutional Neural Networks.

Exercises

1. Can you use word embedding to extract text features and develop a multiclass classifier to classify the newsgroup data? (Note that you might not be able to get better results with word embedding than tf-idf, but it is good practice.)
2. Can you find several challenges in Kaggle (www.kaggle.com) and practice what you have learned throughout the entire book?

12

Categorizing Images of Clothing with Convolutional Neural Networks

The previous chapter wrapped up our coverage of the best practices for general and traditional machine learning. Starting from this chapter, we will dive into the more advanced topics of deep learning and reinforcement learning.

When we deal with image classification, we usually flatten the images and get vectors of pixels and feed them to a neural network (or another model). Although this might do the job, we lose critical spatial information. In this chapter, we will use **Convolutional Neural Networks (CNNs)** to extract rich and distinguishable representations from images. You will see how CNN representations make a "9" a "9", a "4" a "4", a cat a cat, or a dog a dog.

We will start with exploring individual building blocks in the CNN architecture. Then, we will develop a CNN classifier in TensorFlow to categorize clothing images and demystify the convolutional mechanism. Finally, we will introduce data augmentation to boost the performance of CNN models.

We will cover the following topics in this chapter:

- CNN building blocks
- CNNs for classification
- Implementation of CNNs with TensorFlow and Keras

- Classifying clothing images with CNNs
- Visualization of convolutional filters
- Data augmentation and implementation

Getting started with CNN building blocks

Although regular hidden layers (the fully connected layers we have seen so far) do a good job of extracting features from data at certain levels, these representations might be not useful in differentiating images of different classes. CNNs can be used to extract richer and more distinguishable representations that, for example, make a car a car, a plane a plane, or the handwritten letters "y" a "y", "z" a "z", and so on. CNNs are a type of neural network that is biologically inspired by the human visual cortex. To demystify CNNs, I will start by introducing the components of a typical CNN, including the convolutional layer, the nonlinear layer, and the pooling layer.

The convolutional layer

The **convolutional layer** is the first layer in a CNN, or the first few layers in a CNN if it has multiple convolutional layers. It takes in input images or matrices and simulates the way neuronal cells respond to receptive fields by applying a convolutional operation to the input. Mathematically, it computes the **dot product** between the nodes of the convolutional layer and individual small regions in the input layer. The small region is the receptive field, and the nodes of the convolutional layer can be viewed as the values on a filter. As the filter moves along on the input layer, the dot product between the filter and current receptive field (sub-region) is computed. A new layer called the **feature map** is obtained after the filter has convolved over all the sub-regions. Let's look at a simple example, as follows:

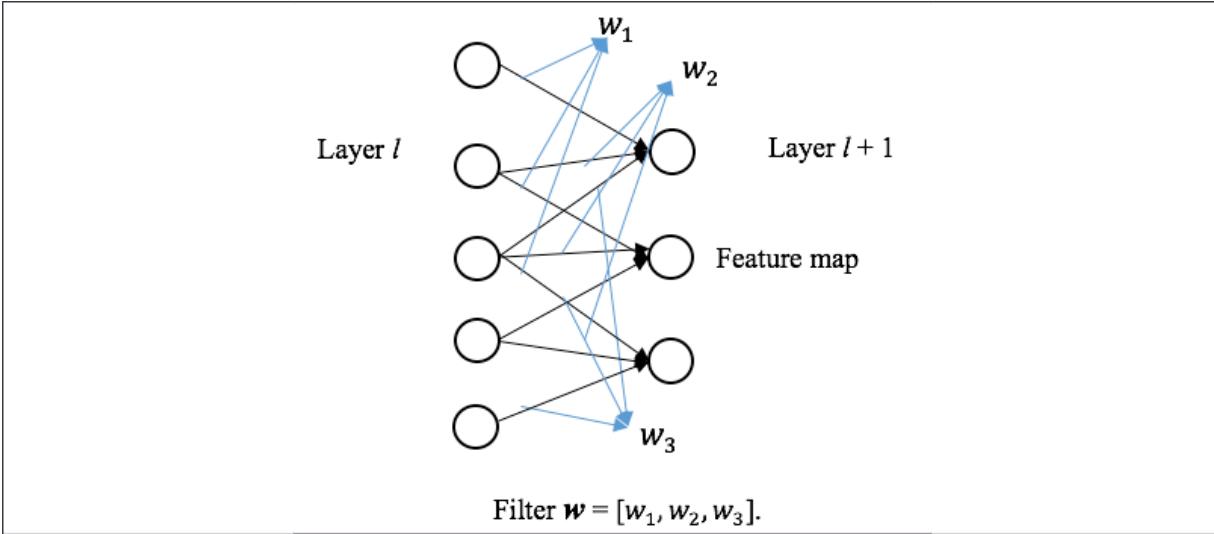


Figure 12.1: How a feature map is generated

In this example, layer l has 5 nodes and the filter is composed of 3 nodes $[w_1, w_2, w_3]$. We first compute the dot product between the filter and the first three nodes in layer l and obtain the first node in the output feature map; then, we compute the dot product between the filter and the middle three nodes and generate the second node in the output feature map; finally, the third node is generated from the convolution on the last three nodes in layer l .

Now, we take a closer look at how convolution works in the following example:

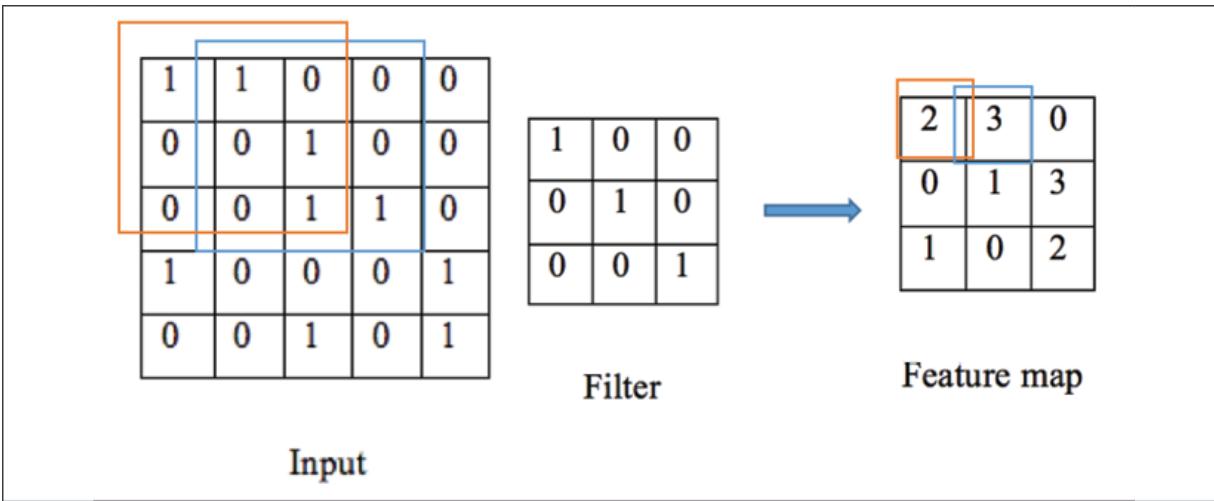


Figure 12.2: How convolution works

In this example, a 3*3 filter is sliding around a 5*5 input matrix from the top left sub-region to the bottom right sub-region. For each sub-region, the dot product is computed using the filter. Take the top left sub-region (in the orange rectangle) as an example: we have $1 * 1 + 1 * 0 + 1 * 1 = 2$, therefore the top left node (in the upper-left orange rectangle) in the feature map is of value 2. For the next leftmost sub-region (in the blue rectangle), we calculate the convolution as $1 * 1 + 1 * 1 + 1 * 1 = 3$, so the value of the next node (in the upper-middle blue rectangle) in the resulting feature map becomes 3. At the end, a 3*3 feature map is generated as a result.

So what do we use convolutional layers for? They are actually used to extract features such as edges and curves. The pixel in the output feature map will be of high value if the corresponding receptive field contains an edge or curve that is recognized by the filter. For instance, in the preceding example, the filter portrays a backslash-shape "\ diagonal edge; the receptive field in the blue rectangle contains a similar curve and hence the highest intensity 3 is created. However, the receptive field at the top-right corner does not contain such a backslash shape, hence it results in a pixel of value 0 in the output feature map. The convolutional layer acts as a curve detector or a shape detector.

Also, a convolutional layer usually has multiple filters detecting different curves and shapes. In the simple preceding example, we only apply one filter and generate one feature map, which indicates how well the shape in the input image resembles the curve represented in the filter. In order to detect more patterns from the input data, we can employ more filters, such as horizontal, vertical curve, 30-degree, and right-angle shape.

Additionally, we can stack several convolutional layers to produce higher-level representations such as the overall shape and contour. Chaining more layers will result in larger receptive fields that are able to capture more global patterns.

In reality, the CNNs, specifically their convolutional layers, mimic the way our visual cells work, as follows:

- Our visual cortex has a set of complex neuronal cells that are sensitive to specific sub-regions of the visual field and that are called **receptive fields**. For instance, some cells only respond in the presence of vertical edges; some cells fire only when they are exposed to horizontal edges; some react stronger when they are shown edges of a certain orientation. These cells are organized together to produce the entire visual perception, with each cell being specialized in a specific component. A convolutional layer in a CNN is composed of a set of filters that act as those cells in humans' visual cortices.
- A simple cell only responds when the edge-like patterns are presented within its receptive sub-regions. A more complex cell is sensitive to larger sub-regions, and as a result, can respond to edge-like patterns across the entire visual field. A stack of convolutional layers is a bunch of complex cells that can detect patterns in a bigger scope.

Right after each convolutional layer, we often apply a nonlinear layer.

The nonlinear layer

The nonlinear layer is basically the activation layer we have seen in *Chapter 8, Predicting Stock Prices with Artificial Neural Networks*. It is used to introduce non-linearity, obviously. Recall that in the convolutional layer, we only perform linear operations (multiplication and addition). And no matter how many linear hidden layers a neural network has, it will just behave as a single-layer perceptron. Hence, we need a nonlinear activation right after the convolutional layer. Again, ReLU is the most popular candidate for the nonlinear layer in deep neural networks.

The pooling layer

Normally after one or more convolutional layers (along with nonlinear activation), we can directly use the derived features for classification. For example, we can apply a softmax layer in the multiclass classification case. But let's do some math first.

Given $28 * 28$ input images, supposing that we apply $20 5 * 5$ filters in the first convolutional layer, we will obtain 20 output feature maps and each feature map layer will be of size $(28 - 5 + 1) * (28 - 5 + 1) = 24 * 24 = 576$. This means that the number of features as inputs for the next layer increases to $11,520 (20 * 576)$ from $784 (28 * 28)$. We then apply $50 5 * 5$ filters in the second convolutional layer. The size of the output grows to $50 * 20 * (24 - 5 + 1) * (24 - 5 + 1) = 400,000$. This is a lot higher than our initial size of 784 . We can see that the dimensionality increases dramatically with every convolutional layer before the final softmax layer. This can be problematic as it leads to overfitting easily, not to mention the cost of training such a large number of weights.

To address the issue of drastically growing dimensionality, we often employ a **pooling layer** after the convolutional and nonlinear layer. The pooling layer is also called the **downsampling layer**. As you can imagine, it reduces the dimensions of the feature maps. This is done by aggregating the statistics of features over sub-regions. Typical pooling methods include:

- Max pooling, which takes the max values over all non-overlapping sub-regions
- Mean pooling, which takes the mean values over all non-overlapping sub-regions

In the following example, we apply a $2 * 2$ max-pooling filter on a $4 * 4$ feature map and output a $2 * 2$ one:

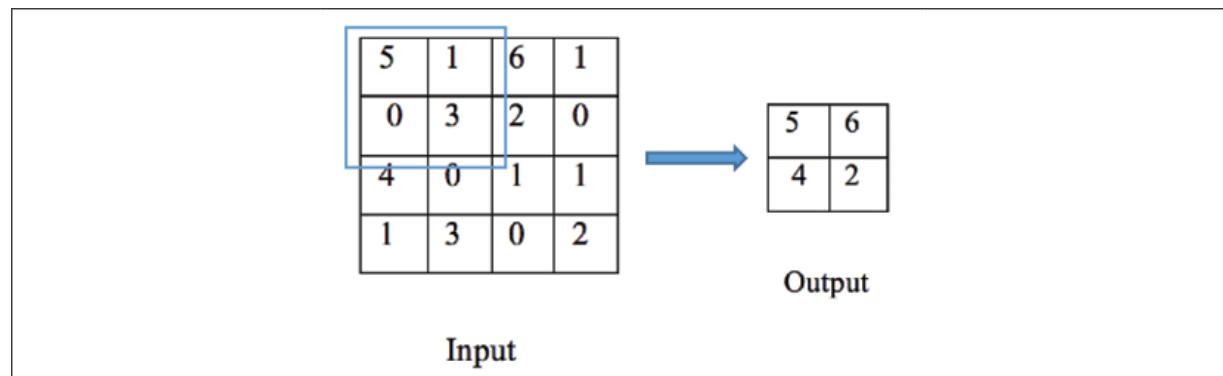


Figure 12.3: How max pooling works

Besides dimensionality reduction, the pooling layer has another advantage: translation invariance. This means that its output doesn't change even if the input matrix undergoes a small amount of translation. For example, if we shift the input image a couple of pixels to the left or right, as long as the highest pixels remain the same in the sub-regions, the output of the max-pooling layer will still be the same. In other words, the prediction becomes less position-sensitive with pooling layers. The following example illustrates how max pooling achieves translation invariance.

Here is the $4 * 4$ original image, along with the output from max pooling with a $2 * 2$ filter:

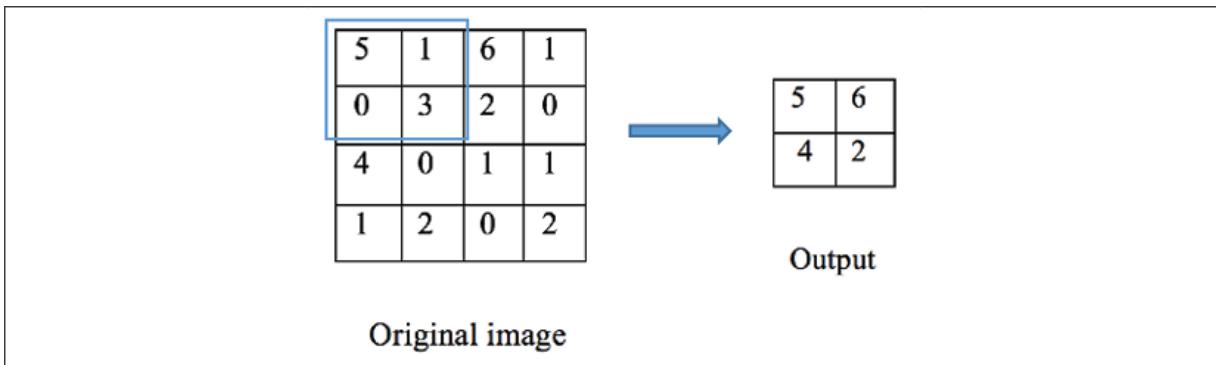


Figure 12.4: The original image and the output from max pooling

And if we shift the image 1 pixel to the right, we have the following shifted image and the corresponding output:

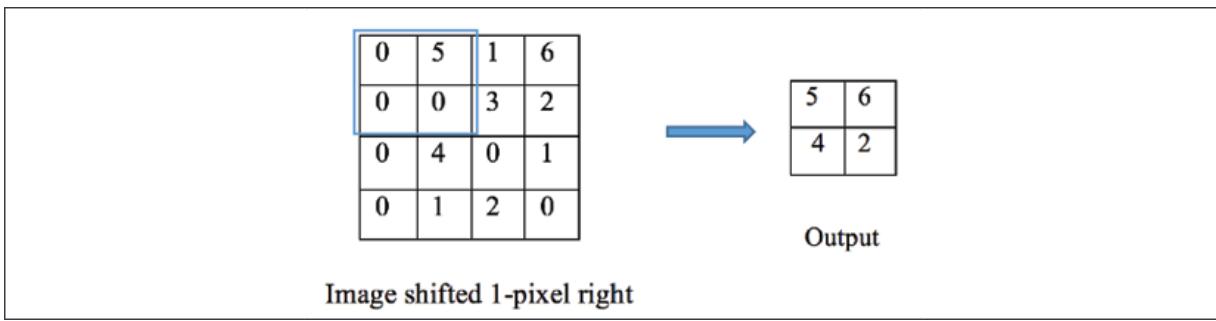


Figure 12.5: The shifted image and the output

We have the same output even if we horizontally move the input image. Pooling layers increase the robustness of image translation.

You've now learned all of the components of a CNN. It was easier than you thought, right? Let's see how they compose a CNN next.

Architecting a CNN for classification

Putting the three types of convolutional-related layers together, along with the fully connected layer(s), we can structure the CNN model for classification as follows:

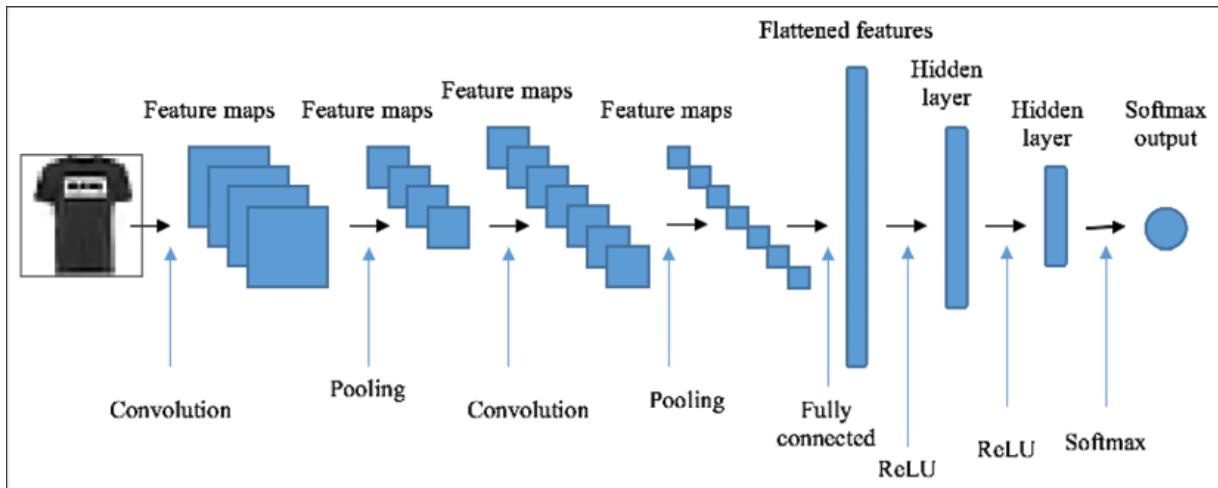


Figure 12.6: CNN architecture

In this example, the input images are first fed into a convolutional layer (with ReLU activation) composed of a bunch of filters. The coefficients of the convolutional filters are trainable. A well-trained initial convolutional layer is able to derive good low-level representations of the input images, which will be critical to downstream convolutional layers if there are any, and also downstream classification tasks. Each resulting feature map is then downsampled by the pooling layer.

Next, the aggregated feature maps are fed into the second convolutional layer. Similarly, the second pooling layer reduces the size of the output

feature maps. You can chain as many pairs of convolutional and pooling layers as you want. The second (or more, if any) convolutional layer tries to compose high-level representations, such as the overall shape and contour, through a series of low-level representations derived from previous layers.

Up until this point, the feature maps are matrices. We need to flatten them into a vector before performing any downstream classification. The flattened features are just treated as the input to one or more fully-connected hidden layers. We can think of a CNN as a hierarchical feature extractor on top of a regular neural network. CNNs are well suited to exploit strong and unique features that differentiate images.

The network ends up with a logistic function if we deal with a binary classification problem, a softmax function for a multiclass case, or a set of logistic functions for multi-label cases.

By now you should have a good understanding of CNNs, and should be ready to solve the clothing image classification problem. Let's start by exploring the dataset.

Exploring the clothing image dataset

The clothing Fashion-MNIST

(<https://github.com/zalandoresearch/fashion-mnist>) is a dataset of images from Zalando (Europe's biggest online fashion retailer). It consists of 60,000 training samples and 10,000 test samples. Each sample is a 28 * 28 grayscale image, associated with a label from the following 10 classes, each representing articles of clothing:

- 0: T-shirt/top
- 1: Trouser
- 2: Pullover

- 3: Dress
- 4: Coat
- 5: Sandal
- 6: Shirt
- 7: Sneaker
- 8: Bag
- 9: Ankle boot

Zalando seeks to make the dataset as popular as the handwritten digits MNIST dataset (<http://yann.lecun.com/exdb/mnist/>) for benchmarking algorithms, and hence calls it Fashion-MNIST.

You can download the dataset from the direct links in the *Get the data* section using the GitHub link, or simply import it from Keras, which already includes the dataset and its API. We will take the latter approach, as follows:

```
>>> import tensorflow as tf
>>> fashion_mnist = tf.keras.datasets.fashion_mnist
>>> (train_images, train_labels), (test_images, test_labels) = f
```

We just import TensorFlow and load the Fashion-MNIST from the Keras module. We now have the training images and their labels, along with the test images and their labels. Feel free to print a few samples from these four arrays, for example, the training labels as follows:

```
>>> print(train_labels)
[9 0 0 ... 3 0 5]
```

The label arrays do not include class names. Hence, we define them as follows and will use them for plotting later on:

```
>>> class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress'
```

Take a look at the format of the image data as follows:

```
>>> print(train_images.shape)
(60000, 28, 28)
```

There are 60,000 training samples and each is represented as 28 * 28 pixels.

Similarly for the 10,000 testing samples, we check the format as follows:

```
>>> print(test_images.shape)
(10000, 28, 28)
```

Let's now inspect a random training sample as follows:

```
>>> import matplotlib.pyplot as plt
>>> plt.figure()
>>> plt.imshow(train_images[42])
>>> plt.colorbar()
>>> plt.grid(False)
>>> plt.title(class_names[train_labels[42]])
>>> plt.show()
```

Refer to the following image as the end result:

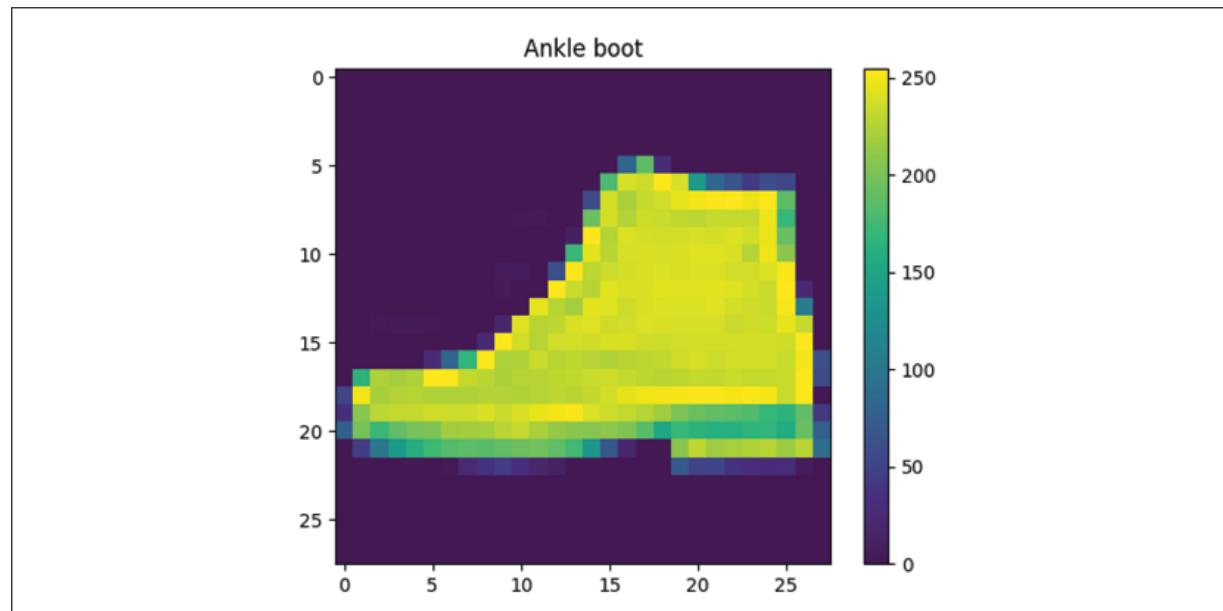


Figure 12.7: A training sample from Fashion-MNIST



You may run into an error similar to the following:

```
OMP: Error #15: Initializing libiomp5.dylib, bu
OMP: Hint This means that multiple copies of th
Abort trap: 6
```

If so, please add the following code at the beginning of your code:

```
>>> import os
>>> os.environ['KMP_DUPLICATE_LIB_OK'] = 'True'
```

In the ankle boot sample, the pixel values are in the range of 0 to 255. Hence, we need to rescale the data to a range of 0 to 1 before feeding it to the neural network. We divide the values of both training samples and test samples by 255 as follows:

```
>>> train_images = train_images / 255.0
>>> test_images = test_images / 255.0
```

Now we display the first 16 training samples after the preprocessing, as follows:

```
>>> for i in range(16):
...     plt.subplot(4, 4, i + 1)
...     plt.subplots_adjust(hspace=.3)
...     plt.xticks([])
...     plt.yticks([])
...     plt.grid(False)
...     plt.imshow(train_images[i], cmap=plt.cm.binary)
...     plt.title(class_names[train_labels[i]])
... plt.show()
```

Refer to the following image of the end result:



Figure 12.8: The end result

In the next section, we will be building our CNN model to classify these clothing images.

Classifying clothing images with CNNs

As mentioned, the CNN model has two main components: the feature extractor composed of a set of convolutional and pooling layers, and the classifier backend similar to a regular neural network.

Architecting the CNN model

As the convolutional layer in Keras only takes in individual samples in three dimensions, we need to first reshape the data into four dimensions as follows:

```
>>> X_train = train_images.reshape((train_images.shape[0], 28, 28, 1))
>>> X_test = test_images.reshape((test_images.shape[0], 28, 28, 1))
>>> print(X_train.shape)
(60000, 28, 28, 1)
```

The first dimension is the number of samples, and the fourth dimension is the appended one representing the grayscale images.

Before we develop the CNN model, let's specify the random seed in TensorFlow for reproducibility:

```
>>> tf.random.set_seed(42)
```

We now import the necessary modules from Keras and initialize a Keras-based model:

```
>>> from tensorflow.keras import datasets, layers, models, losses
```

```
>>> model = models.Sequential()
```

For the convolutional extractor, we are going to use three convolutional layers. We start with the first convolutional layer with 32 small-sized 3×3 filters. This is implemented by the following code:

```
>>> model.add(layers.Conv2D(32, (3, 3), activation='relu', input
```

Note that we use ReLU as the activation function.

The convolutional layer is followed by a max-pooling layer with a 2×2 filter:

```
>>> model.add(layers.MaxPooling2D((2, 2)))
```

Here comes the second convolutional layer. It has $64 \times 3 \times 3$ filters and comes with a ReLU activation function as well:

```
>>> model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

The second convolutional layer is followed by another max-pooling layer with a 2×2 filter:

```
>>> model.add(layers.MaxPooling2D((2, 2)))
```

We continue adding the third convolutional layer. It has $128 \times 3 \times 3$ filters at this time:

```
>>> model.add(layers.Conv2D(128, (3, 3), activation='relu'))
```

The resulting filter maps are then flattened to provide features to the downstream classifier backend:

```
>>> model.add(layers.Flatten())
```

For the classifier backend, we just use one hidden layer with 64 nodes:

```
>>> model.add(layers.Dense(64, activation='relu'))
```

The hidden layer here is the regular fully-connected dense layer, with ReLU as the activation function.

And finally, the output layer has 10 nodes representing 10 different classes in our case, along with a softmax activation:

```
>>> model.add(layers.Dense(10, activation='softmax'))
```

Now we compile the model with Adam as the optimizer, cross-entropy as the loss function, and classification accuracy as the metric:

```
>>> model.compile(optimizer='adam',
...                  loss=losses.sparse_categorical_crossentropy,
...                  metrics=['accuracy'])
```

Let's take a look at the model summary as follows:

```
>>> model.summary()
Model: "sequential"

Layer (type)                 Output Shape              Param #
=====
conv2d (Conv2D)             (None, 26, 26, 32)        320
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
conv2d_1 (Conv2D)            (None, 11, 11, 64)       18496
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
conv2d_2 (Conv2D)            (None, 3, 3, 128)        73856
flatten (Flatten)           (None, 1152)             0
dense (Dense)               (None, 64)                73792
dense_1 (Dense)              (None, 10)                650
=====
Total params: 167,114
Trainable params: 167,114
Non-trainable params: 0
```

It displays each layer in the model, the shape of its single output, and the number of its trainable parameters. As you may notice, the output from a convolutional layer is three-dimensional, where the first two are the dimensions of the feature maps and the third is the number of filters used in the convolutional layer. The size (the first two dimensions) of the max-pooling output is half of its input feature map in the example. Feature maps

are downsampled by the pooling layer. You may want to see how many parameters there would be to be trained if you take out all the pooling layers. Actually, it is 4,058,314! So, the benefits of applying pooling are obvious: avoiding overfitting and reducing training cost.

You may wonder why the numbers of convolutional filters keep increasing over the layers. Recall that each convolutional layer attempts to capture patterns of a specific hierarchy. The first convolutional layer captures low-level patterns, such as edges, dots, and curves. Then the subsequent layers combine those patterns extracted in previous layers to form high-level patterns, such as shapes and contours. As we move forward in these convolutional layers, there are more and more combinations of patterns to capture in most cases. As a result, we need to keep increasing (or at least not decreasing) the number of filters in the convolutional layers.

Fitting the CNN model

Now it's time to train the model we just built. We train it for 10 iterations and evaluate it using the testing samples:

```
>>> model.fit(X_train, train_labels, validation_data=(X_test, te
```

Note that the batch size is 32 by default. Here is how the training progresses:

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [=====] - 68s 1ms/sample -
Epoch 2/10
60000/60000 [=====] - 68s 1ms/sample -
Epoch 3/10
60000/60000 [=====] - 69s 1ms/sample -
Epoch 4/10
60000/60000 [=====] - 69s 1ms/sample -
Epoch 5/10
60000/60000 [=====] - 69s 1ms/sample -
Epoch 6/10
60000/60000 [=====] - 69s 1ms/sample -
Epoch 7/10
60000/60000 [=====] - 69s 1ms/sample -
Epoch 8/10
60000/60000 [=====] - 69s 1ms/sample -
Epoch 9/10
60000/60000 [=====] - 69s 1ms/sample -
Epoch 10/10
60000/60000 [=====] - 69s 1ms/sample -
```

```
60000/60000 [=====] - 71s 1ms/sample -
Epoch 7/10
60000/60000 [=====] - 68s 1ms/sample -
Epoch 8/10
60000/60000 [=====] - 70s 1ms/sample -
Epoch 9/10
60000/60000 [=====] - 69s 1ms/sample -
Epoch 10/10
60000/60000 [=====] - 68s 1ms/sample -
10000/1 - 5s - loss: 0.2933 - accuracy: 0.9100
```

We are able to achieve an accuracy of around 96% on the training set and 91% on the test set.

If you want to double-check the performance on the test set, you can do the following:

```
>>> test_loss, test_acc = model.evaluate(X_test, test_labels, verbose=0)
>>> print('Accuracy on test set:', test_acc)
Accuracy on test set: 0.91
```

Now that we have a well-trained model, we can make predictions on the test set using the following code:

```
>>> predictions = model.predict(X_test)
```

Take a look at the first sample; we have the prediction as follows:

```
>>> print(predictions[0])
[1.8473367e-11 1.1924335e-07 1.0303306e-13 1.2061150e-12 3.19379e-13
 3.5260896e-07 6.2364621e-13 9.1853758e-07 4.0739218e-11 9.99998e-14]
```

We have the predicted probabilities for this sample. To obtain the predicted label, we do the following:

```
>>> import numpy as np
>>> print('Predicted label for the first test sample: ', np.argmax(predictions))
Predicted label for the first test sample: 9
```

And we do a fact check as follows:

```
>>> print('True label for the first test sample: ', test_labels[0])
True label for the first test sample: 9
```

We take one step further by plotting the sample image and the prediction results, including the probabilities of 10 possible classes:

```
>>> def plot_image_prediction(i, images, predictions, labels, class_names):
...     plt.subplot(1,2,1)
...     plt.imshow(images[i], cmap=plt.cm.binary)
...     prediction = np.argmax(predictions[i])
...     color = 'blue' if prediction == labels[i] else 'red'
...     plt.title(f'{class_names[labels[i]]} (predicted {class_names[prediction]}', color=color)
...     plt.subplot(1,2,2)
...     plt.grid(False)
...     plt.xticks(range(10))
...     plot = plt.bar(range(10), predictions[i], color="#777777")
...     plt.ylim([0, 1])
...     plot[prediction].set_color('red')
...     plot[labels[i]].set_color('blue')
...     plt.show()
```

The original image (on the left) will have the title *<true label> (predicted <predicted label>)* in blue if the prediction matches the label, or in red if not. The predicted probability (on the right) will be a blue bar on the true label, or a red bar on the predicted label if the predicted label is not the same as the true label.

Let's try it with the first test sample:

```
>>> plot_image_prediction(0, test_images, predictions, test_labels)
```



Refer to the following screenshot for the end result:

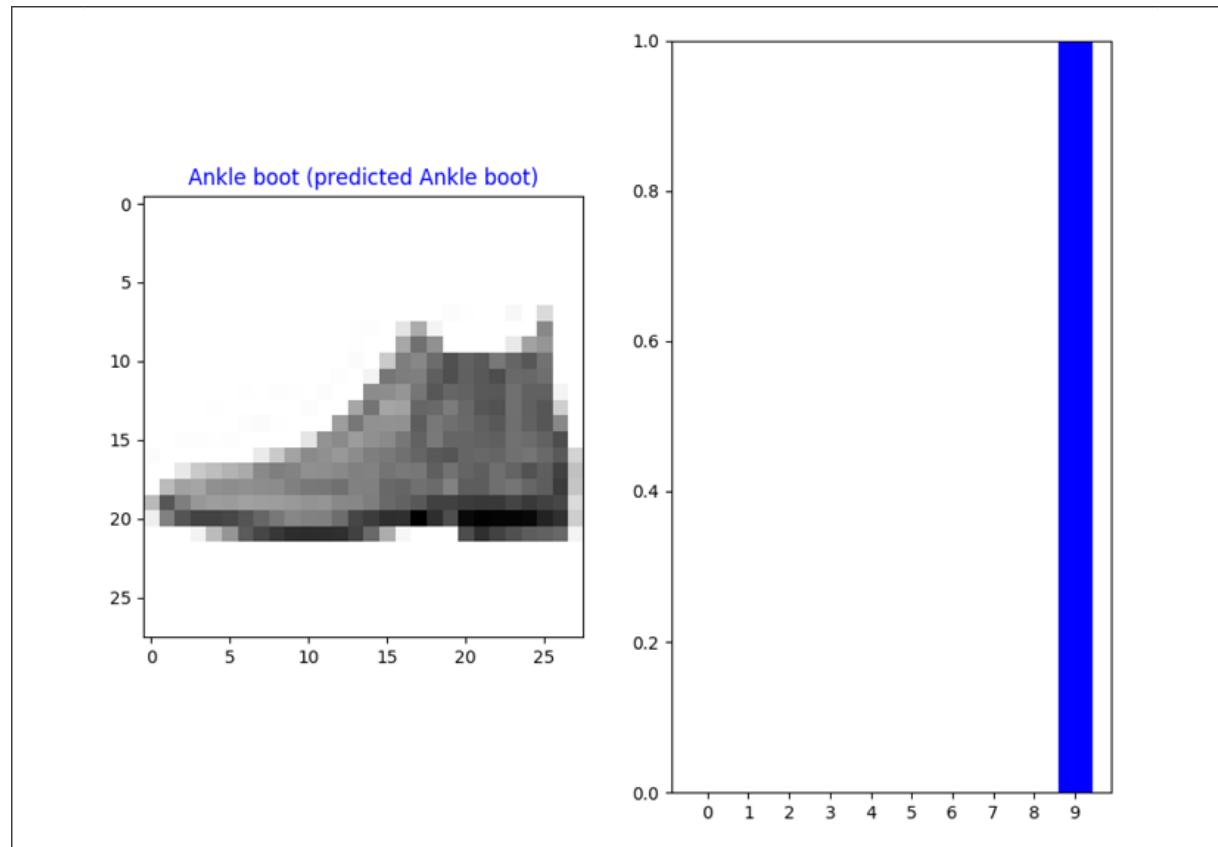


Figure 12.9: A sample of the original image with its prediction result

Feel free to play around with other samples, especially those that aren't predicted accurately, such as item 17.

You have seen how the trained model performs, and you may wonder what the learned convolutional filters look like. You will find the answer in the next section.

Visualizing the convolutional filters

We extract the convolutional filters from the trained model and visualize them with the following steps:

1. From the model summary, we know that the layers of indexes 0, 2, and 4 in the model are convolutional layers. Using the second convolutional layer as an example, we obtain its filters as follows:

```
>>> filters, _ = model.layers[2].get_weights()
```

2. Next, we normalize the filter values to the range of 0 to 1 so we can visualize them more easily:

```
>>> f_min, f_max = filters.min(), filters.max()  
>>> filters = (filters - f_min) / (f_max - f_min)
```

3. Recall we have 64 filters in this convolutional layer. We visualize the first 16 filters in four rows and four columns:

```
>>> n_filters = 16  
>>> for i in range(n_filters):  
...     filter = filters[:, :, :, i]  
...     plt.subplot(4, 4, i+1)  
...     plt.xticks([])  
...     plt.yticks([])  
...     plt.imshow(filter[:, :, 0], cmap='gray')  
... plt.show()
```

Refer to the following screenshot for the end result:

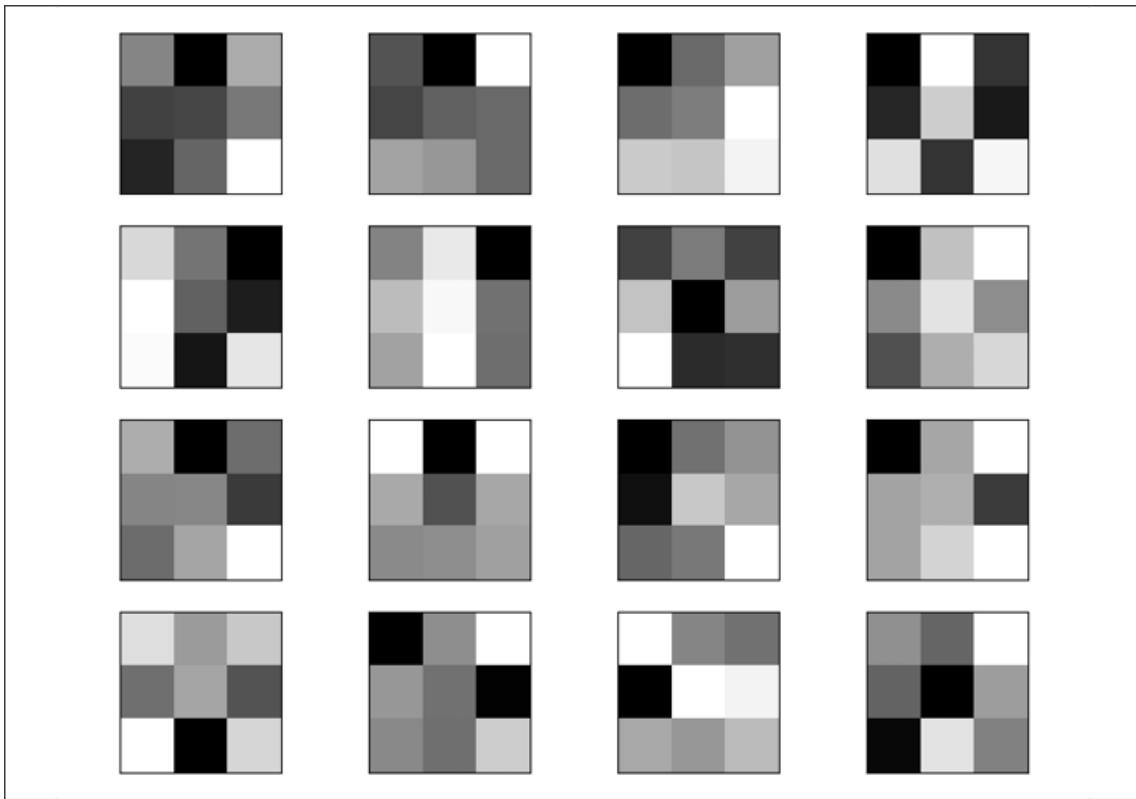


Figure 12.10: Trained convolutional filters

In a convolutional filter, the dark squares represent small weights and the white squares indicate large weights. Based on this intuition, we can see that the second filter in the second row detects the vertical line in a receptive field, while the third filter in the first row detects a gradient from light in the bottom right to dark in the top left.

In the previous example, we trained the clothing image classifier with 60,000 labeled samples. However, it is not easy to gather such a big labeled dataset in reality. Specifically, image labeling is expensive and time-consuming. How can we effectively train an image classifier with a limited number of samples? One solution is data augmentation.


```
...                         save_format='jpeg'):
...
...             i += 1
...             if i > 2:
...                 break
...
...             plt.subplot(2, 2, 1, xticks=[], yticks[])
...             plt.imshow(original_img)
...             plt.title("Original")
...
...             i = 1
...             for file in os.listdir(folder):
...                 if file.startswith(save_prefix):
...                     plt.subplot(2, 2, i + 1, xticks=[], yticks[])
...                     aug_img = load_img(folder + "/" + file)
...                     plt.imshow(aug_img)
...                     plt.title(f"Augmented {i}")
...
...                     i += 1
...
...             plt.show()
```

The generator first randomly generates three (in this example) images given the original image and the augmentation condition. The function then plots the original image along with three artificial images. The generated images are also stored in the local disk in the folder named `aug_images`.

Let's try it out with our `horizontal_flip` generator using the first training image (feel free to use any other image) as follows:

```
>>> generate_plot_pics(datagen, train_images[0], 'horizontal_fli
```

Refer to the following screenshot for the end result:

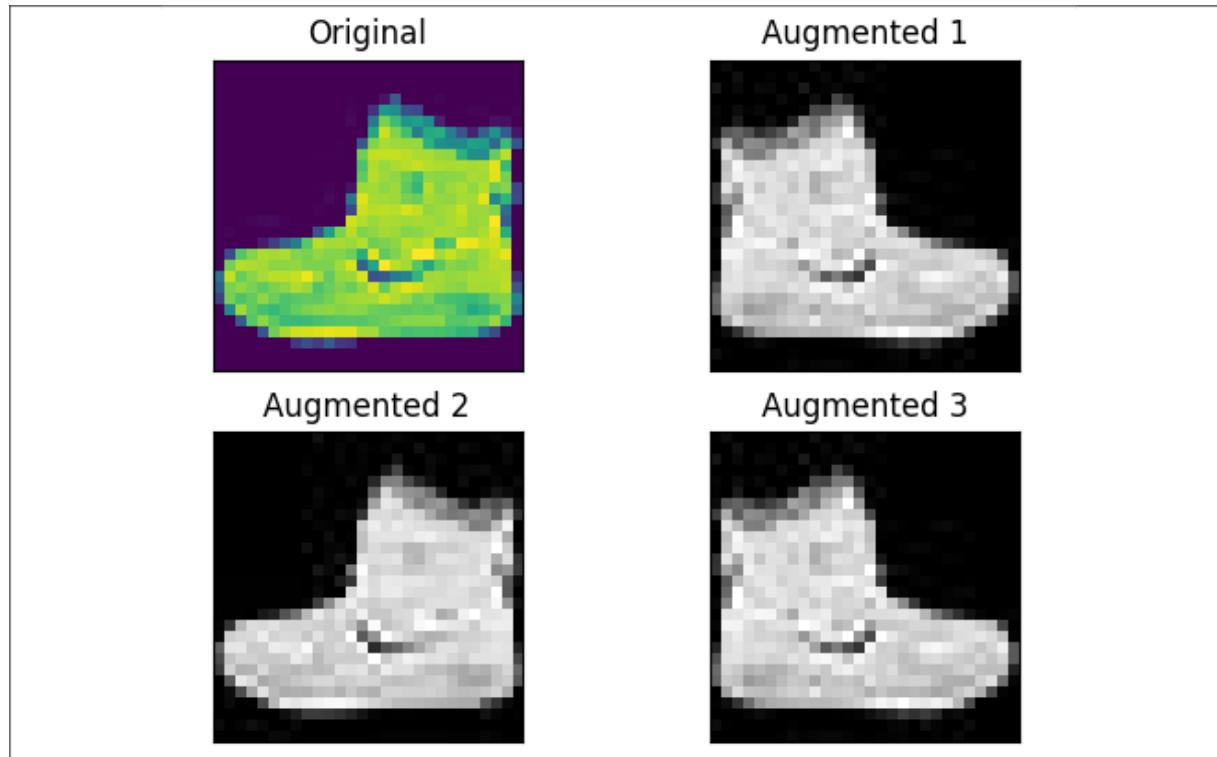


Figure 12.11: Horizontally flipped images for data augmentation

As you can see, the generated images are either horizontally flipped or not flipped. Why don't we try one with both horizontally and vertically flips simultaneously? We can do so as follows:

```
>>> datagen = ImageDataGenerator(horizontal_flip=True,  
...                                vertical_flip=True)  
>>> generate_plot_pics(datagen, train_images[0], 'hv_flip')
```

Refer to the following screenshot for the end result:

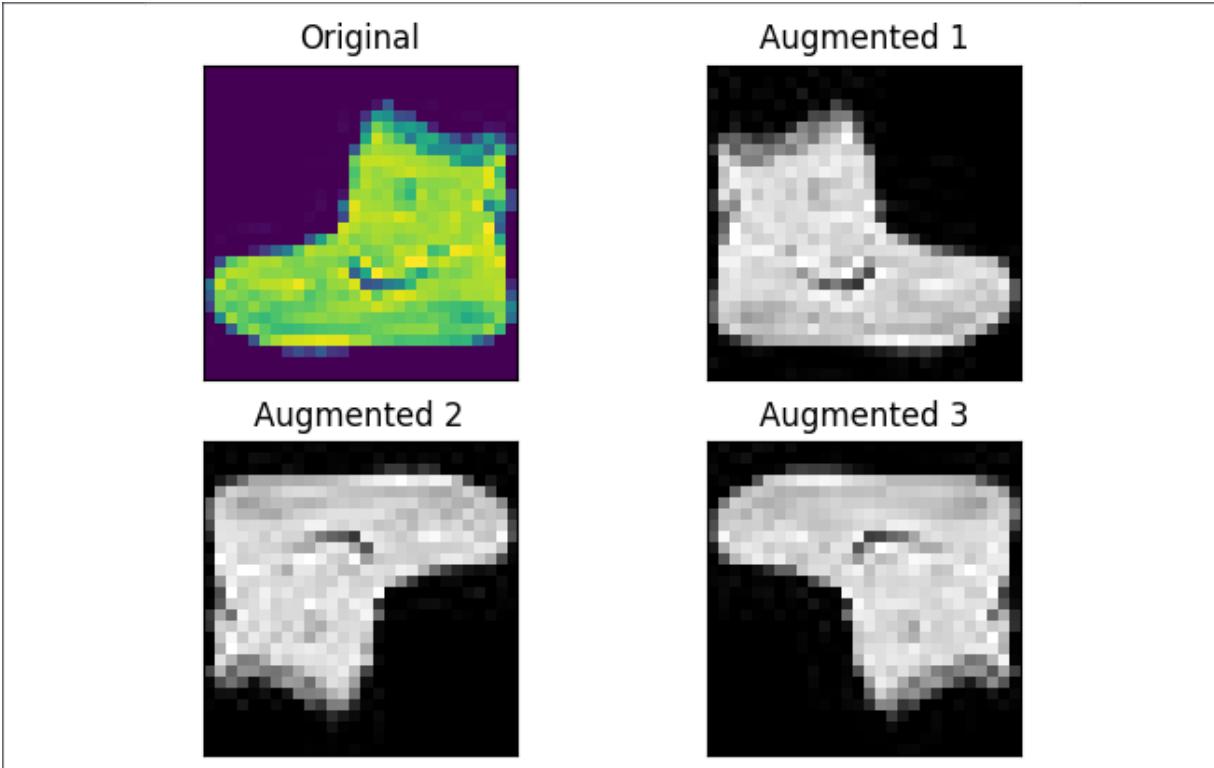


Figure 12.12: Horizontally and vertically flipped images for data augmentation

Besides being horizontally flipped or not, the generated images are either vertically flipped or not flipped.



In general, the horizontally flipped images convey the same message as the original ones. Vertically flipped images are not frequently seen. It is also worth noting that flipping only works in orientation-insensitive cases, such as classifying cats and dogs or recognizing parts of cars. On the contrary, it is dangerous to do so in cases where orientation matters, such as classifying between right and left turn signs.

Rotation for data augmentation

Instead of rotating every 90 degrees as in horizontal or vertical flipping, a small-to-medium degree rotation can also be applied in image data augmentation. Let's see rotation in the following example:

```
>>> datagen = ImageDataGenerator(rotation_range=30)
>>> generate_plot_pics(datagen, train_images[0], 'rotation')
```

Refer to the following screenshot for the end result:

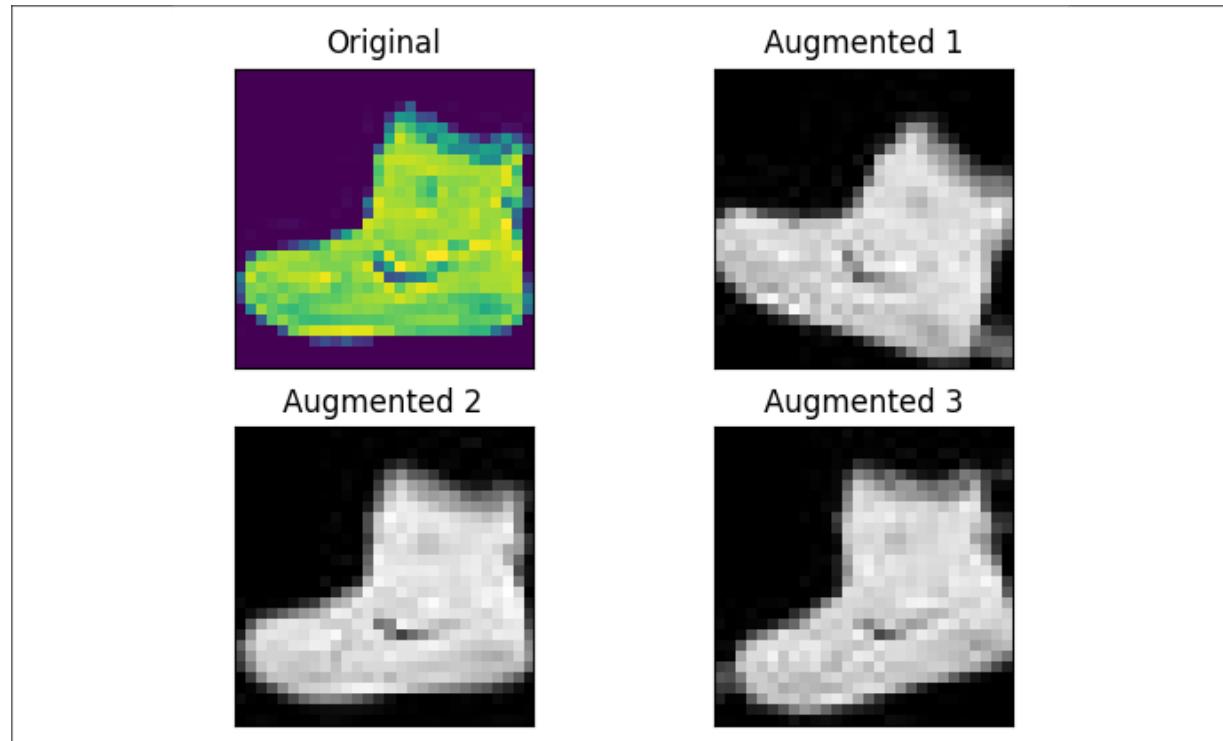


Figure 12.13: Rotated images for data augmentation

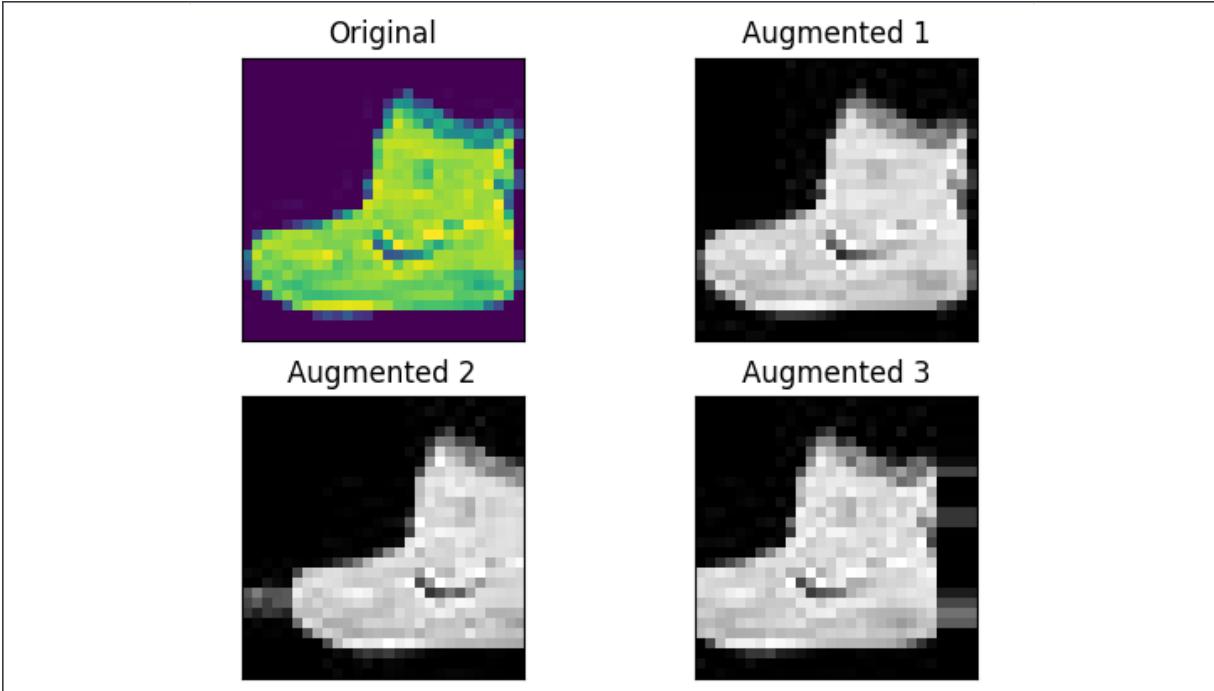
In the preceding example, the image is rotated by any degree ranging from -30 (counterclockwise) to 30 (clockwise).

Shifting for data augmentation

Shifting is another commonly used augmentation method. It generates new images by moving the original image horizontally or vertically by a small number of pixels. In TensorFlow, you can either specify a maximal number of pixels the image will be shifted by, or a maximal portion of the weight or height. Let's take a look at the following example where we shift the image horizontally by at most 8 pixels:

```
>>> datagen = ImageDataGenerator(width_shift_range=8)
>>> generate_plot_pics(datagen, train_images[0], 'width_shift')
```

Refer to the following screenshot for the end result:



12.14: Horizontally shifted images for data augmentation

As you can see, the generated images are horizontally shifted by no more than 8 pixels. Let's now try shifting both horizontally and vertically at the same time:

```
>>> datagen = ImageDataGenerator(width_shift_range=8,
...                               height_shift_range=8)
>>> generate_plot_pics(datagen, train_images[0], 'width_height_s')
```

Refer to the following screenshot for the end result:

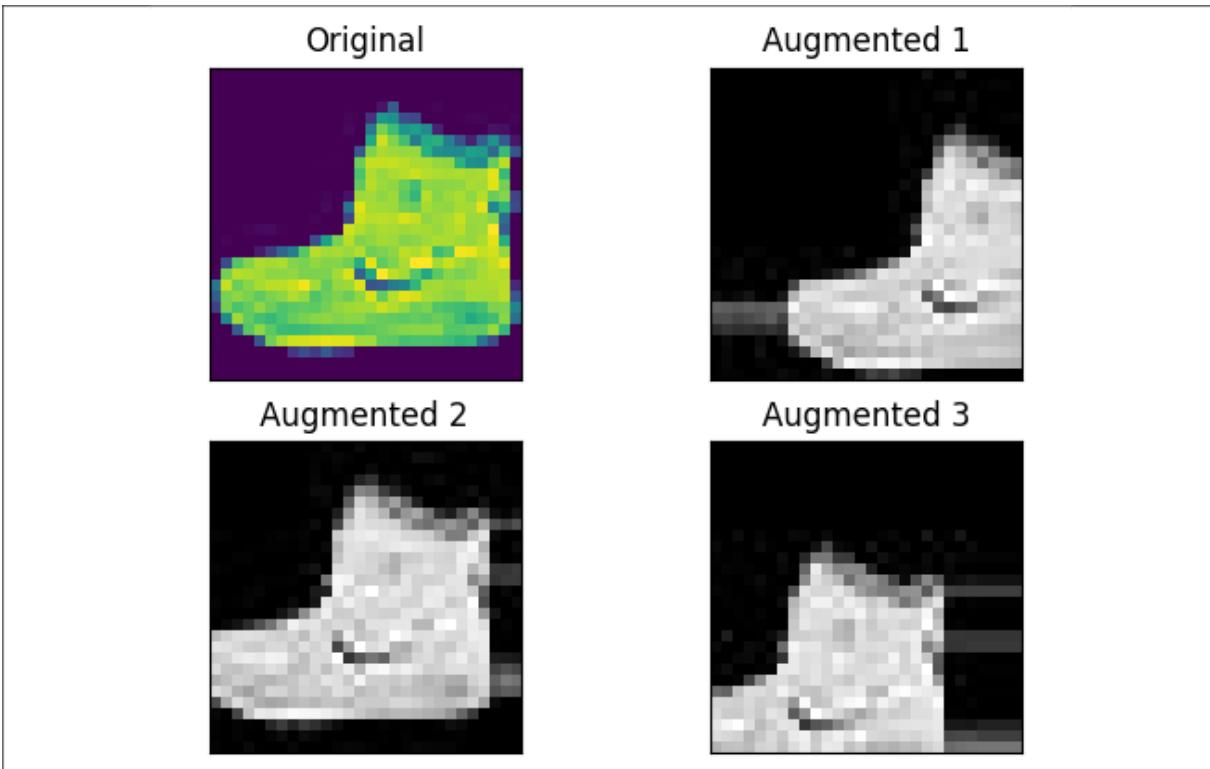


Figure 12.15: Horizontally and vertically shifted images for data augmentation

Improving the clothing image classifier with data augmentation

Armed with several common augmentation methods, we now apply them to train our image classifier on a small dataset in the following steps:

1. We start by constructing a small training set:

```
>>> n_small = 500
>>> X_train = X_train[:n_small]
>>> train_labels = train_labels[:n_small]
>>> print(X_train.shape)
(500, 28, 28, 1)
```

We only use 500 samples for training.

2. We architect the CNN model using the Keras Sequential API:

```
>>> model = models.Sequential()  
>>> model.add(layers.Conv2D(32, (3, 3), activation='relu',  
>>> model.add(layers.MaxPooling2D((2, 2)))  
>>> model.add(layers.Conv2D(64, (3, 3), activation='relu'))  
>>> model.add(layers.Flatten())  
>>> model.add(layers.Dense(32, activation='relu'))  
>>> model.add(layers.Dense(10, activation='softmax'))
```

As we have training data of a small size, we use only two convolutional layers and adjust the size of the hidden layer accordingly: the first convolutional layer has 32 small-sized 3×3 filters, the second convolutional layer has 64 filters of the same size, and the fully-connected hidden layer has 32 nodes.

3. We compile the model with Adam as the optimizer, cross-entropy as the loss function, and classification accuracy as the metric:

```
>>> model.compile(optimizer='adam',  
...                 loss=losses.sparse_categorical_crossentropy,  
...                 metrics=['accuracy'])
```

4. We first train the model without data augmentation:

```
>>> model.fit(X_train, train_labels, validation_data=(X_test,  
Train on 500 samples, validate on 10000 samples  
Epoch 1/20  
500/500 [=====] - 6s 11ms/sample -  
Epoch 2/20  
500/500 [=====] - 4s 8ms/sample -  
Epoch 3/20  
500/500 [=====] - 4s 9ms/sample -  
.....  
.....  
Epoch 18/20  
500/500 [=====] - 5s 10ms/sample -  
accuracy: 0.7947  
Epoch 19/20  
500/500 [=====] - 5s 10ms/sample -  
Epoch 20/20  
500/500 [=====] - 5s 10ms/sample -
```

We train the model for 20 iterations.

5. Let's see how it performs on the test set:

```
>>> test_loss, test_acc = model.evaluate(X_test, test_labels)
>>> print('Accuracy on test set:', test_acc)
    Accuracy on test set: 0.7924
```

The model without data augmentation has a classification accuracy of 79.24% on the test set.

6. Now we work on the data augmentation and see if it can boost the performance. We first define the augmented data generator:

```
>>> datagen = ImageDataGenerator(height_shift_range=3,
...                                horizontal_flip=True
...)
```

We herein apply horizontal flipping and vertical shifting. We notice that none of the clothing images are upside down, hence vertical flipping won't provide any normal-looking images. Also, most clothing images are perfectly horizontally centered, so we are not going to perform any width shift. To put it simply, we try to avoid creating augmented images that will look different from the original ones.

7. We clone the CNN model we used previously:

```
>>> model_aug = tf.keras.models.clone_model(model)
```

It only copies the CNN architecture and creates new weights instead of sharing the weights of the existing model.

We compile the cloned model as before, with Adam as the optimizer, cross-entropy as the loss function, and classification accuracy as the metric:

```
>>> model_aug.compile(optimizer='adam',
...                      loss=losses.sparse_categorical_crossentropy,
...                      metrics=['accuracy'])
```

8. Finally, we fit this CNN model on data with real-time augmentation:

```
>>> train_generator = datagen.flow(X_train, train_labels, ...
>>> model_aug.fit(train_generator, epochs=50, validation_data=...
```

```
Epoch 1/50
13/13 [=====] - 5s 374ms/step - loss: 0.6971 - acc: 0.7924
.....  
.....  
Epoch 48/50
13/13 [=====] - 4s 300ms/step - loss: 0.6971 - acc: 0.7924
Epoch 49/50
13/13 [=====] - 4s 304ms/step - loss: 0.6971 - acc: 0.7924
Epoch 50/50
13/13 [=====] - 4s 306ms/step - loss: 0.6971 - acc: 0.7924
```

During the training process, augmented images are randomly generated on the fly to feed the model. We train the model with data augmentation for 50 iterations this time, as it takes more iterations for the model to learn the patterns.

9. Let's see how it performs on the test set:

```
>>> test_loss, test_acc = model_aug.evaluate(X_test, test_labels)
>>> print('Accuracy on test set:', test_acc)
Accuracy on test set: 0.8109
```

The accuracy increases to 81.09% from 79.24% with data augmentation.

Feel free to fine-tune the hyperparameters as we did in *Chapter 8, Predicting Stock Prices with Artificial Neural Networks*, and see if you can further improve the classification performance.

Summary

In this chapter, we worked on classifying clothing images using CNNs. We started with a detailed explanation of individual components of a CNN model and learned how CNNs are inspired by the way our visual cells work. We then developed a CNN model to categorize fashion-MNIST clothing images from Zalando. We also talked about data augmentation and several popular image augmentation methods. We practiced implementing deep learning models again with the Keras module in TensorFlow.

In the next chapter, we will focus on another type of deep learning networks: **Recurrent Neural Networks (RNNs)**. CNNs and RNNs are the two most powerful deep neural networks that make deep learning so popular nowadays.

Exercises

1. As mentioned before, can you try to fine-tune the CNN image classifier and see if you can beat what we have achieved?
2. Can you also employ dropout and early stopping techniques?

13

Making Predictions with Sequences Using Recurrent Neural Networks

In the previous chapter, we focused on **convolutional neural networks** (CNNs) and used them to deal with image-related tasks. In this chapter, we will explore **recurrent neural networks** (RNNs), which are suitable for sequential data and time-dependent data, such as daily temperature, DNA sequences, and customers' shopping transactions over time. You will learn how the recurrent architecture works and see variants of the model. We will then work on their applications, including sentiment analysis and text generation. Finally, as a bonus section, we will cover a recent state-of-the-art sequential learning model: the Transformer.

We will cover the following topics in this chapter:

- Sequential learning by RNNs
- Mechanisms and training of RNNs
- Different types of RNNs
- Long Short-Term Memory RNNs
- RNNs for sentiment analysis
- RNNs for text generation
- Self-attention and the Transformer model

Introducing sequential learning

The machine learning problems we have solved so far in this book have been time-independent. For example, ad click-through doesn't depend on the user's historical ad clicks under our previous approach; in face classification, the model only takes in the current face image, not previous ones. However, there are many cases in life that depend on time. For example, in financial fraud detection, we can't just look at the present transaction; we should also consider previous transactions so that we can model based on their discrepancy. Another example is **part-of-speech (PoS)** tagging, where we assign a PoS (verb, noun, adverb, and so on) to a word. Instead of solely focusing on the given word, we must look at some previous words, and sometimes the next words too.

In time-dependent cases like those just mentioned, the current output is dependent on not only the current input, but also the previous inputs; note that the length of the previous inputs is not fixed. Using machine learning to solve such problems is called **sequence learning**, or **sequence modeling**. And obviously, the time-dependent event is called a **sequence**. Besides events that occur in disjoint time intervals (such as financial transactions, phone calls, and so on), text, speech, and video are also sequential data.

You may be wondering why we can't just model the sequential data in a regular fashion by feeding in the entire sequence. This can be quite limiting as we have to fix the input size. One problem is that we will lose information if an important event lies outside of the fixed window. But can we just use a very large time window? Note that the feature space grows along with the window size. The feature space will become excessive if we want to cover enough events in a certain time window. Hence, overfitting can be another problem.

I hope you now see why we need to model sequential data in a different way. In the next section, we will talk about the model used for modern sequence learning: RNNs.

Learning the RNN architecture by example

As you can imagine, RNNs stand out because of their recurrent mechanism. We will start with a detailed explanation of this in the next section. We will talk about different types of RNNs after that, along with some typical applications.

Recurrent mechanism

Recall that in feedforward networks (such as vanilla neural networks and CNNs), data moves one way, from the input layer to the output layer. In RNNs, the recurrent architecture allows data to circle back to the input layer. This means that data is not limited to a feedforward direction. Specifically, in a hidden layer of an RNN, the output from the previous time point will become part of the input for the current time point. The following diagram illustrates how data flows in an RNN in general:

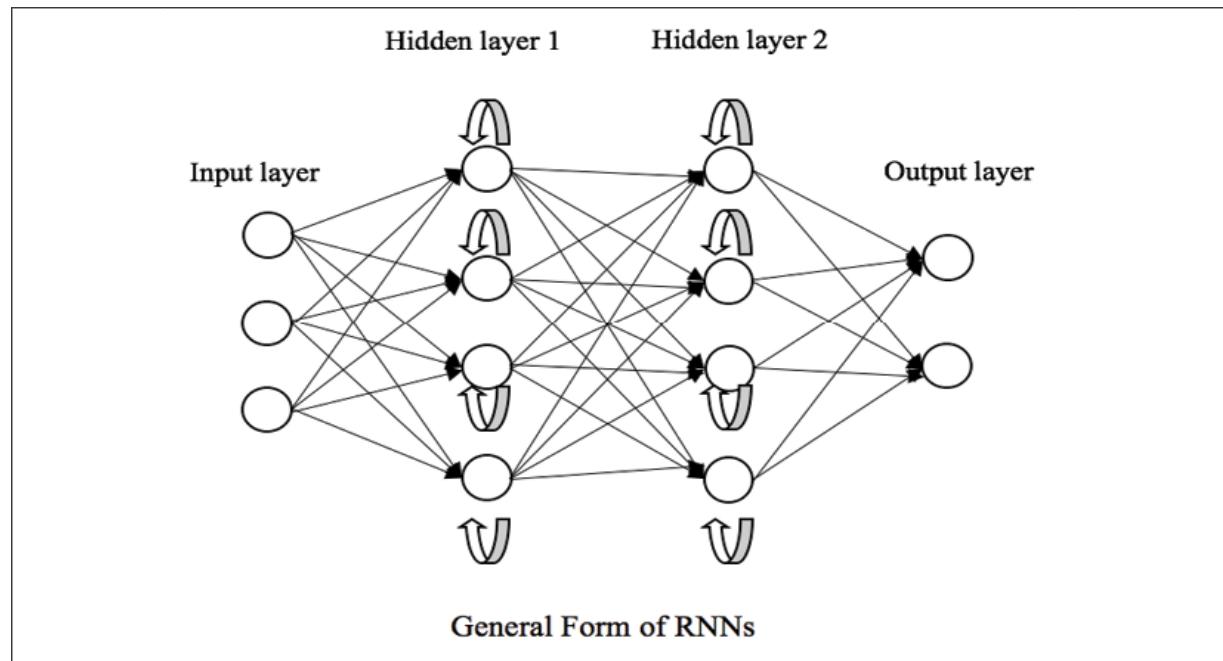


Figure 13.1: The general form of an RNN

Such a recurrent architecture makes RNNs work well with sequential data, including time series (such as daily temperatures, daily product sales, and clinical EEG recordings) and general consecutive data with order (such as words in a sentence, DNA sequences, and so on). Take a financial fraud detector as an example; the output features from the previous transaction go into the training for the current transaction. In the end, the prediction for one transaction depends on all of its previous transactions. Let me explain the recurrent mechanism in a mathematical and visual way.

Suppose we have some inputs, x_t . Here, t represents a time step or a sequential order. In a feedforward neural network, we simply assume that inputs at different t are independent of each other. We denote the output of a hidden layer at a time step, t , as $h_t = f(x_t)$, where f is the abstract of the hidden layer.

This is depicted in the following diagram:

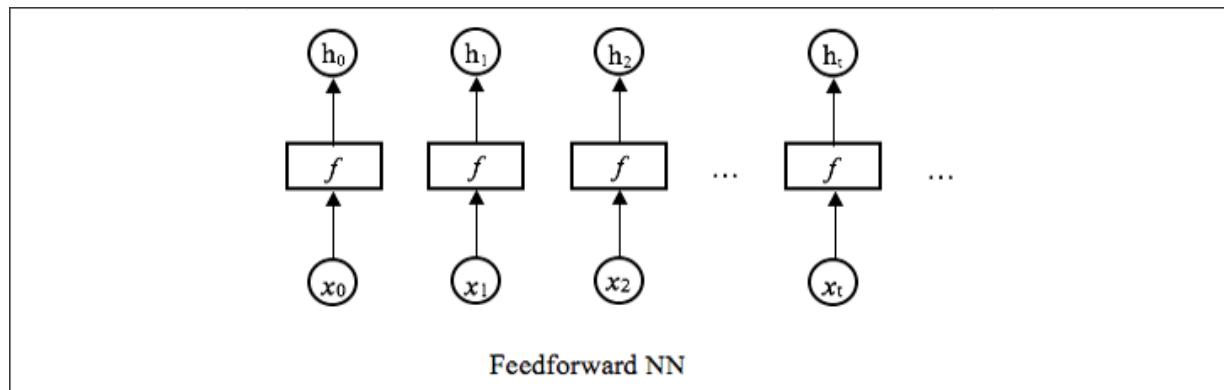


Figure 13.2: General form of a feedforward neural network

On the contrary, the feedback loop in an RNN feeds the information of the previous state to the current state. The output of a hidden layer of an RNN at a time step, t , can be expressed as $h_t = f(h_{t-1}, x_t)$. This is depicted in the following diagram:

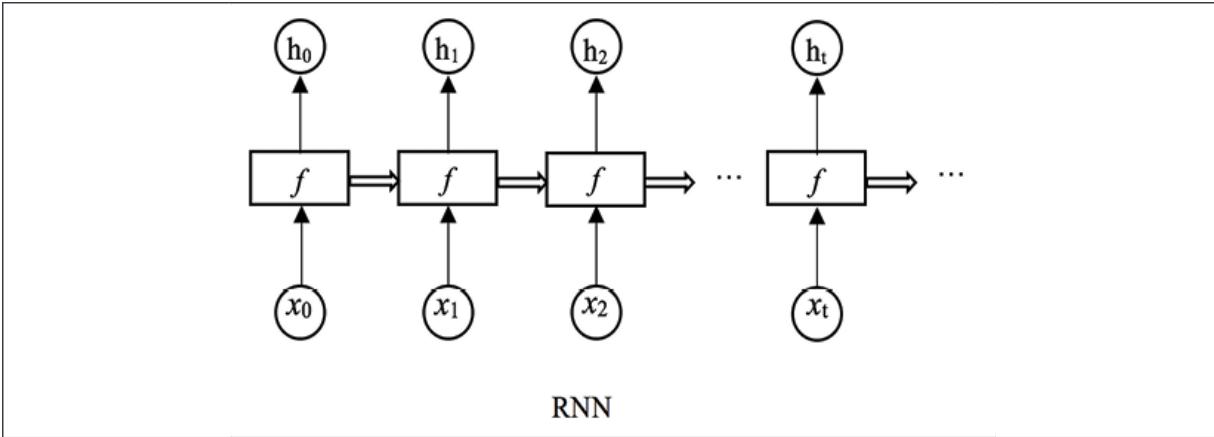


Figure 13.3: Unfolded recurrent layer over time steps

The same task, f , is performed on each element of the sequence, and the output, h_t , is dependent on the output that's generated from previous computations, h_{t-1} . The chain-like architecture captures the "memory" that has been calculated so far. This is what makes RNNs so successful in dealing with sequential data.

Moreover, thanks to the recurrent architecture, RNNs also have great flexibility in dealing with different combinations of input sequences and/or output sequences. In the next section, we will talk about different categories of RNNs based on input and output, including the following:

- Many-to-one
- One-to-many
- Many-to-many (synced)
- Many-to-many (unsynced)

We will start by looking at many-to-one RNNs.

Many-to-one RNNs

The most intuitive type of RNN is probably **many-to-one**. A many-to-one RNN can have input sequences with as many time steps as you want, but it

only produces one output after going through the entire sequence. The following diagram depicts the general structure of a many-to-one RNN:

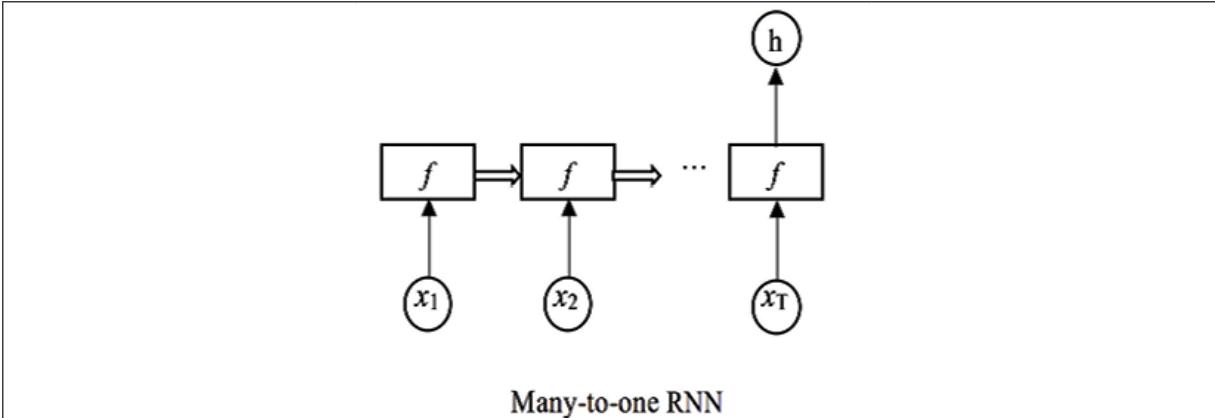


Figure 13.4: General form of a many-to-one RNN

Here, f represents one or more recurrent hidden layers, where an individual layer takes in its own output from the previous time step. Here is an example of three hidden layers stacking up:

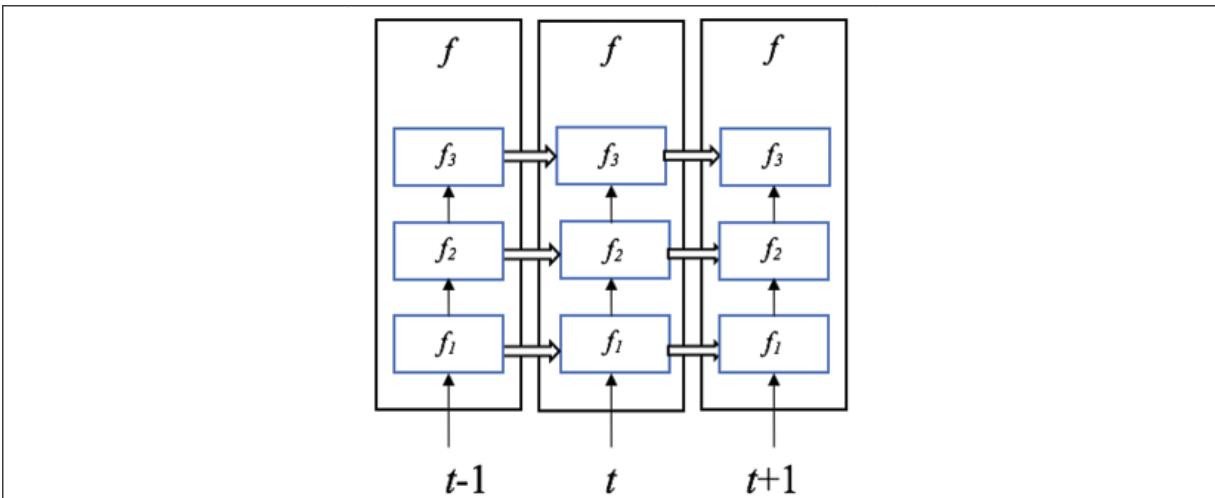


Figure 13.5: Example of three recurrent layers stacking up

Many-to-one RNNs are widely used for classifying sequential data. Sentiment analysis is a good example of this and is where the RNN reads the entire customer review, for instance, and assigns a sentiment score (positive, neutral, or negative sentiment). Similarly, we can also use RNNs of this kind in the topic classification of news articles. Identifying the genre

of a song is another application as the model can read the entire audio stream. We can also use many-to-one RNNs to determine whether a patient is having a seizure based on an EEG trace.

One-to-many RNNs

One-to-many RNNs are the exact opposite of many-to-one RNNs. They take in only one input (not a sequence) and generate a sequence of outputs. A typical one-to-many RNN is presented in the following diagram:

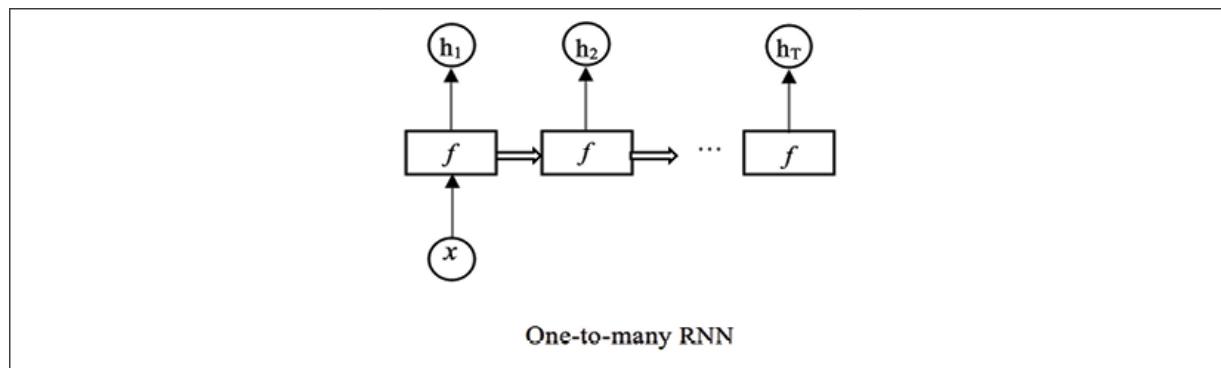


Figure 13.6: General form of a one-to-many RNN

Again, f represents one or more recurrent hidden layers.

Note that "one" here doesn't mean that there is only one input feature. It means the input is from one time step, or it is time-independent.

One-to-many RNNs are commonly used as sequence generators. For example, we can generate a piece of music given a starting note or/and a genre. Similarly, we can write a movie script like a professional screenwriter using one-to-many RNNs with a starting word we specify. Image captioning is another interesting application: the RNN takes in an image and outputs the description (a sentence of words) of the image.

Many-to-many (synced) RNNs

The third type of RNN, many-to-many (synced), allows each element in the input sequence to have an output. Let's look at how data flows in the following many-to-many (synced) RNN:

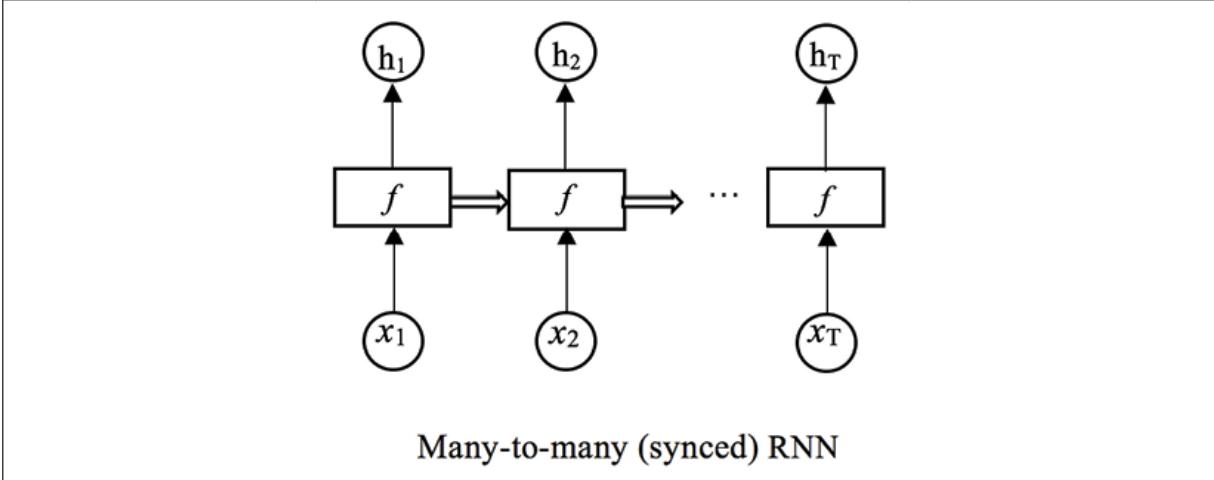


Figure 13.7: General form of a many-to-many (synced) RNN

As you can see, each output is calculated based on its corresponding input and all the previous outputs.

One common use case for this type of RNN is time series forecasting, where we want to perform rolling prediction at every time step based on the current and previously observed data. Here are some examples of time series forecasting where we can leverage synced many-to-many RNNs:

- Product sales each day for a store
- Daily closing price of a stock
- Power consumption of a factory each hour

They are also widely used in solving NLP problems, including PoS tagging, named-entity recognition, and real-time speech recognition.

Many-to-many (unsynced) RNNs

Sometimes, we only want to generate the output sequence *after* we've processed the entire input sequence. This is the **unsynced** version of a

many-to-many RNN.

Refer to the following diagram for the general structure of a many-to-many (unsynced) RNN:

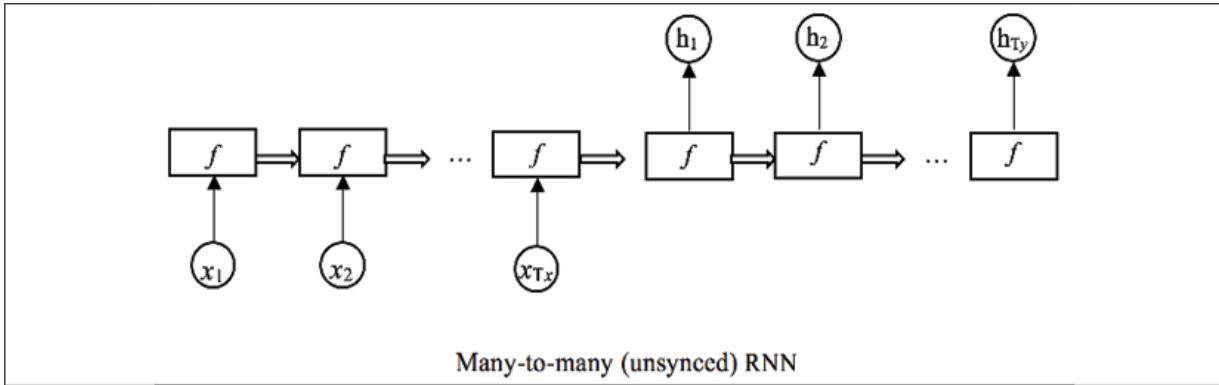


Figure 13.8: General form of a many-to-many (unsynced) RNN

Note that the length of the output sequence (T_y in the preceding diagram) can be different from that of the input sequence (T_x in the preceding diagram). This provides us with some flexibility.

This type of RNN is the go-to model for machine translation. In French-English translation, for example, the model first reads a complete sentence in French and then produces a translated sentence in English. Multi-step ahead forecasting is another popular example: sometimes, we are asked to predict sales for multiple days in the future when given data from the past month.

You have now learned about four types of RNN based on the model's input and output.



Wait, what about one-to-one RNNs? There is no such thing. One-to-one is just a regular feedforward model.

We will be applying some of these types of RNN to solve projects, including sentiment analysis and word generation, later in this chapter. Now, let's figure out how an RNN model is trained.

Training an RNN model

To explain how we optimize the weights (parameters) of an RNN, we first annotate the weights and the data on the network, as follows:

- U denotes the weights connecting the input layer and the hidden layer.
- V denotes the weights between the hidden layer and the output layer.
Note here that we use only one recurrent layer for simplicity.
- W denotes the weights of the recurrent layer; that is, the feedback layer.
- x_t denotes the inputs at time step t .
- s_t denotes the hidden state at time step t .
- h_t denotes the outputs at time step t .

Next, we unfold the simple RNN model over three time steps: $t - 1$, t , and $t + 1$, as follows:

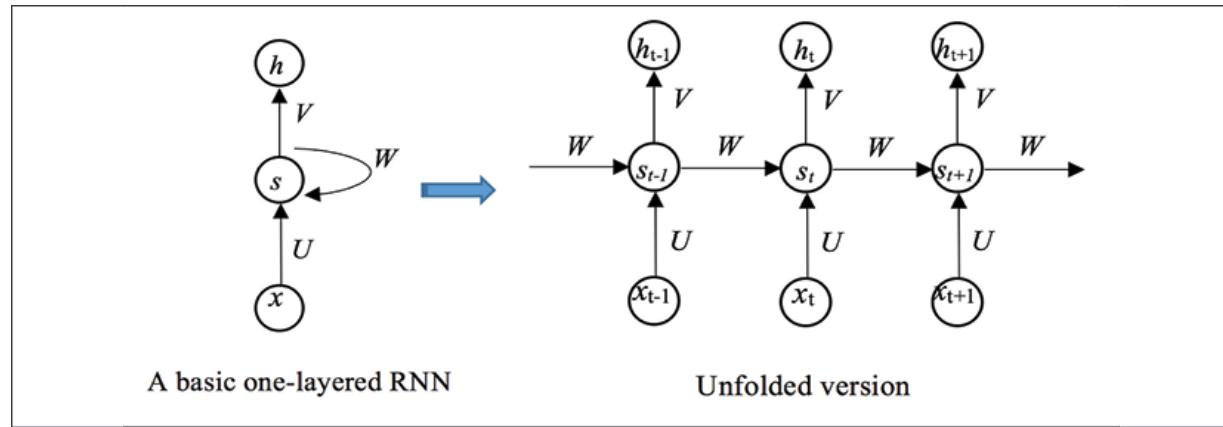


Figure 13.9: Unfolding a recurrent layer

We describe the mathematical relationships between the layers as follows:

- We let a denote the activation function for the hidden layer. In RNNs, we usually choose tanh or ReLU as the activation function for the hidden layers.

- Given the current input, x_t , and the previous hidden state, s_{t-1} , we compute the current hidden state, s_t , by $s_t = a(Ux_t + Ws_{t-1})$. Feel free to read *Chapter 8, Predicting Stock Prices with Artificial Neural Networks* again to brush up on your knowledge of neural networks.
- In a similar manner, we compute s_{t-1} based on $s_{t-2}: s_{t-1} = a(Ux_{t-1} + Ws_{t-2})$. We repeat this until s_1 , which depends on $s_0: s_1 = a(Ux_1 + Ws_0)$. We usually set s_0 to all zeros.
- We let g denote the activation function for the output layer. It can be a sigmoid function if we want to perform binary classification, a softmax function for multi-class classification, and a simple linear function (that is, no activation) for regression.
- Finally, we compute the output at time step t , $h_t: h_t = g(Vs_t)$.

With the dependency in hidden states over time steps (that is, s_t depends on s_{t-1} , s_{t-1} depends on s_{t-2} , and so on), the recurrent layer brings memory to the network, which captures and retains information from all the previous time steps.

As we did for traditional neural networks, we apply the backpropagation algorithm to optimize all the weights, U , V , and W , in RNNs. However, as you may have noticed, the output at a time step is indirectly dependent on all the previous time steps (h_t depends on s_t , while s_t depends on all the previous ones). Hence, we need to compute the loss over all previous $t-1$ time steps, besides the current time step. Consequently, the gradients of the weights are calculated this way. For example, if we want to compute the gradients at time step $t = 4$, we need to backpropagate the previous four time steps ($t = 3, t = 2, t = 1, t = 0$) and sum up the gradients over these five time steps. This version of the backpropagation algorithm is called **Backpropagation Through Time (BPTT)**.

The recurrent architecture enables RNNs to capture information from the very beginning of the input sequence. This advances the predictive capability of sequence learning. You may be wondering whether vanilla RNNs can handle long sequences. They can in theory, but not in practice due to the **vanishing gradient** problem. Vanishing gradient means the

gradient will become vanishingly small over long time steps, which prevents the weight from updating. I will explain this in detail in the next section, as well as introducing a variant architecture, Long Short-Term Memory, that helps solve this issue.

Overcoming long-term dependencies with Long Short-Term Memory

Let's start with the vanishing gradient issue in vanilla RNNs. Where does it come from? Recall that during backpropagation, the gradient decays along with each time step in the RNN (that is, $s_t = a(Ux_t + Ws_{t-1})$); early elements in a long input sequence will have little contribution to the computation of the current gradient. This means that vanilla RNNs can only capture the temporal dependencies within a short time window. However, dependencies between time steps that are far away are sometimes critical signals to the prediction. RNN variants, including **Long Short-Term Memory (LSTM)** and **Gated Recurrent Unit (GRU)**, are specifically designed to solve problems that require learning long-term dependencies.



We will be focusing on LSTM in this book as it is a lot more popular than GRU. LSTM was introduced a decade earlier and is more mature than GRU. If you are interested in learning more about GRU and its applications, feel free to check out *Chapter 6, Recurrent Neural Networks*, of *Hands-On Deep Learning Architectures with Python* by Yuxi Hayden Liu (Packt Publishing).

In LSTM, we use a gating mechanism to handle long-term dependencies. Its magic comes from a memory unit and three information gates built on top of the recurrent cell. The word "gate" is taken from the logic gate in a circuit (https://en.wikipedia.org/wiki/Logic_gate). It is basically a sigmoid function whose output value ranges from 0 to 1. 0 represents the "off" logic, while 1 represents the "on" logic.

The LSTM version of the recurrent cell is depicted in the following diagram, right after the vanilla version for comparison:

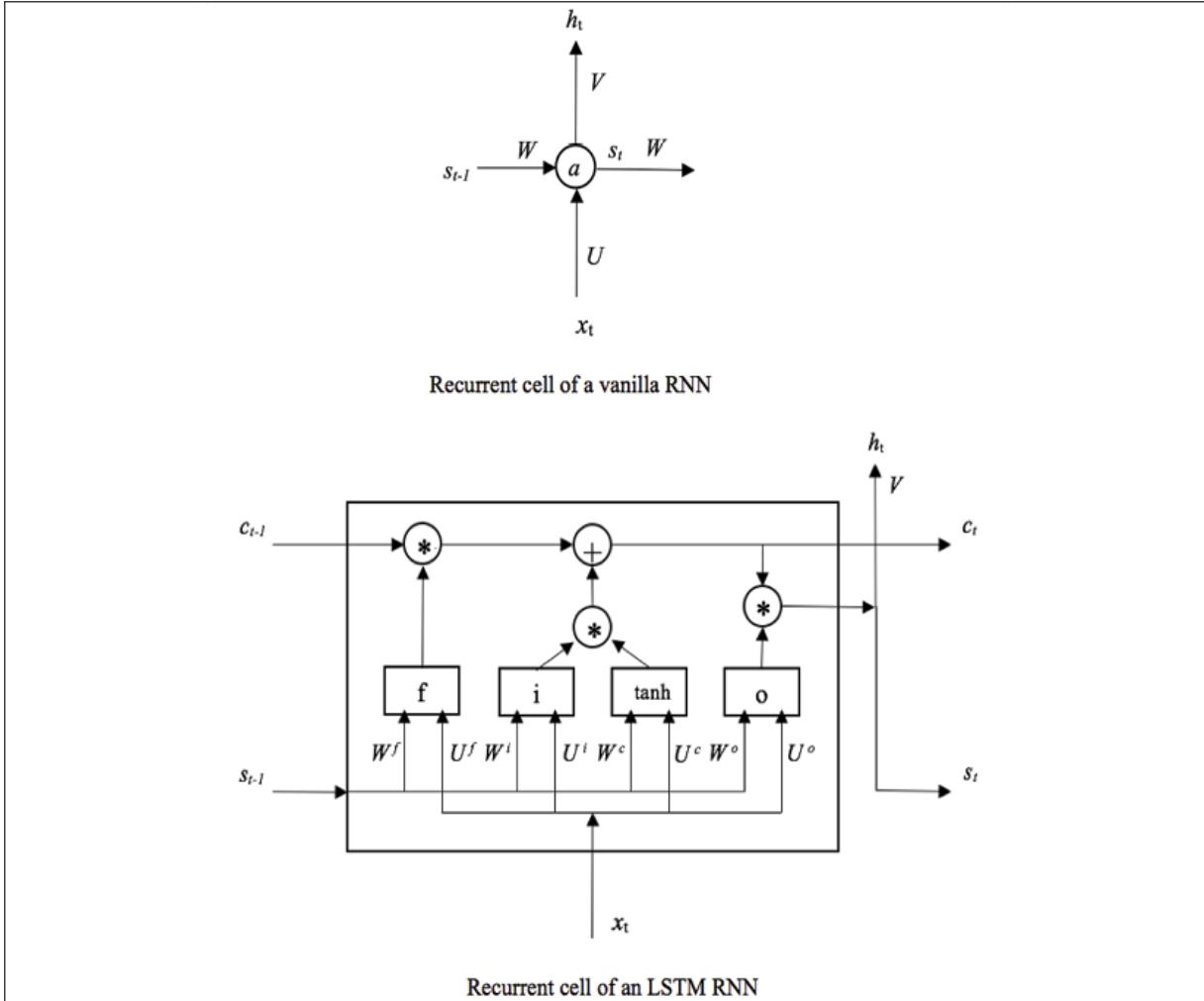


Figure 13.10: Recurrent cell in vanilla RNNs versus LSTM RNNs

Let's look at the LSTM recurrent cell in detail from left to right:

- c_t is the **memory unit**. It memorizes information from the very beginning of the input sequence.
- "f" stands for the **forget gate**. It determines how much information from the previous memory state, c_{t-1} , to forget, or in other words, how much information to pass forward. Let W^f denote the weights between

the forget gate and the previous hidden state, s_{t-1} , and U^f denote the weights between the forget gate and the current input, x_t .

- "i" represents the **input gate**. It controls how much information from the current input to put through. W^i and U^i are the weights connecting the input gate to the previous hidden state, s_{t-1} , and the current input, x_t , respectively.
- The "tanh" is simply the activation function for the hidden state. It acts as the "a" in the vanilla RNN. Its output is computed based on the current input, x_t , along with the associated weights, U^c , the previous hidden state, s_{t-1} , and the corresponding weights, W^c .
- "o" serves as the **output gate**. It defines how much information is extracted from the internal memory for the output of the entire recurrent cell. As always, W^o and U^o are the associated weights for the previous hidden state and current input, respectively.

We describe the relationship between these components as follows:

- The output of the forget gate, f , at time step t is computed as $f = \text{sigmoid}(W^f s_{t-1} + U^f x_t)$.
- The output of the input gate, i , at time step t is computed as $i = \text{sigmoid}(W^i s_{t-1} + U^i x_t)$.
- The output of the tanh activation, c' , at time step t is computed as $c' = \tanh(W^c s_{t-1} + U^c x_t)$.
- The output of the output gate, o , at time step t is computed as $o = \text{sigmoid}(W^o s_{t-1} + U^o x_t)$.
- The memory unit, c_t , at time step t is updated using $c_t = f.* c_{t-1} + i.* c'$ (here, the operator $.*$ denotes element-wise multiplication). Again, the output of a sigmoid function has a value from 0 to 1. Hence, the forget gate, f , and input gate, i , control how much of the previous memory, c_{t-1} , and the current memory input, c' , to carry forward, respectively.
- Finally, we update the hidden state, s_t , at time step t by $s_t = o.* c_t$. Here, the output gate, o , governs how much of the updated memory

unit, c_t , will be used as the output of the entire cell.

As always, we apply the **BPTT** algorithm to train all the weights in LSTM RNNs, including four sets each of weights, U and W , associated with three gates and the tanh activation function. By learning these weights, the LSTM network explicitly models long-term dependencies in an efficient way. Hence, LSTM is the go-to or default RNN model in practice. Next, you will learn how to use LSTM RNNs to solve real-world problems. We will start by categorizing movie review sentiment.

Analyzing movie review sentiment with RNNs

So, here comes our first RNN project: movie review sentiment. We'll use the IMDb (<https://www.imdb.com/>) movie review dataset (<https://ai.stanford.edu/~amaas/data/sentiment/>) as an example. It contains 25,000 highly polar movie reviews for training, and another 25,000 for testing. Each review is labeled as 1 (positive) or 0 (negative). We'll build our RNN-based movie sentiment classifier in the following three sections: *Analyzing and preprocessing the movie review data*, *Developing a simple LSTM network*, and *Boosting the performance with multiple LSTM layers*.

Analyzing and preprocessing the data

We'll start with data analysis and preprocessing, as follows:

1. We import all necessary modules from TensorFlow:

```
>>> import tensorflow as tf
>>> from tensorflow.keras.datasets import imdb
>>> from tensorflow.keras import layers, models, losses, optimizers
>>> from tensorflow.keras.preprocessing.sequence import pad_sequences
```

-
2. Keras has a built-in IMDb dataset, so first, we load the dataset:

```
>>> vocab_size = 5000
>>> (X_train, y_train), (X_test, y_test) = \
    imdb.load_data(num_words=vocab_size)
```

Here, we set the vocabulary size and only keep this many most frequent words. In this example, this is the top 5,000 words that occur most frequently in the dataset. If `num_words` is `None`, all the words will be kept.

3. Take a look at the training and testing data we just loaded:

```
>>> print('Number of training samples:', len(y_train))
Number of training samples: 25000
>>> print('Number of positive samples', sum(y_train))
Number of positive samples 12500
>>> print('Number of test samples:', len(y_test))
Number of test samples: 25000
```

The training set is perfectly balanced, with the same number of positive and negative samples.

4. Print a training sample, as follows:

```
>>> print(X_train[0])
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66
```

As you can see, the raw text has already been transformed into a bag of words and each word is represented by an integer. And for convenience, the value of the integer indicates how frequently the word occurs in the dataset. For instance, "1" represents the most frequent word ("the", as you can imagine), while "10" represents the 10th most frequent word. Can we find out what the words are? Let's see in the next step.

5. We use the word dictionary to map the integer back to the word it represents:

```
>>> word_index = imdb.get_word_index()
>>> index_word = {index: word for word, index in word_index}
```

Take the first review as an example:

```
>>> print([index_word.get(i, ' ') for i in X_train[0]])  
['the', 'as', 'you', 'with', 'out', 'themselves', 'powerful']
```

6. Next, we analyze the length of each sample (the number of words in each review, for example). We do so because all the input sequences to an RNN model must be the same length:

```
>>> review_lengths = [len(x) for x in X_train]
```

Plot the distribution of these document lengths, as follows:

```
>>> import matplotlib.pyplot as plt  
>>> plt.hist(review_lengths, bins=10)  
>>> plt.show()
```

Refer to the following diagram for the distribution result:

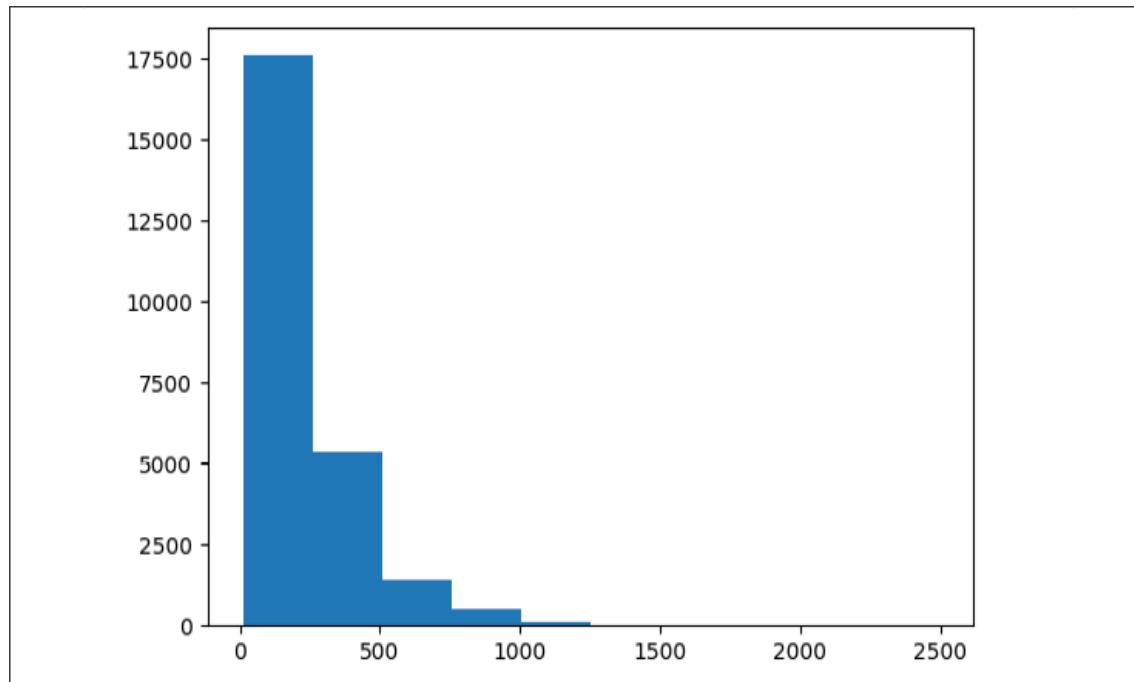


Figure 13.11: Review length distribution

7. As you can see, the majority of the reviews are around 200 words long. Next, we set 200 as the universal sequence length by padding shorter reviews with zeros and truncating longer reviews. We use the `pad_sequences` function from Keras to accomplish this:

```
>>> maxlen = 200
>>> X_train = pad_sequences(X_train, maxlen=maxlen)
>>> X_test = pad_sequences(X_test, maxlen=maxlen)
```

Let's look at the shape of the input sequences after this:

```
>>> print('X_train shape after padding:', X_train.shape)
X_train shape after padding: (25000, 200)
>>> print('X_test shape after padding:', X_test.shape)
X_test shape after padding: (25000, 200)
```

Let's move on to building an LSTM network.

Building a simple LSTM network

Now that the training and testing datasets are ready, we can build our first RNN model:

1. First, we fix the random seed and initiate a Keras Sequential model:

```
>>> tf.random.set_seed(42)
>>> model = models.Sequential()
```

2. Since our input sequences are word indices that are equivalent to one-hot encoded vectors, we need to embed them in dense vectors using the `Embedding` layer from Keras:

```
>>> embedding_size = 32
>>> model.add(layers.Embedding(vocab_size, embedding_size))
```

Here, we embed the input sequences that are made of up `vocab_size=5000` unique word tokens into dense vectors of size 32.

Feel free to reread *Best practice 14 – Extracting features from text data using word embedding with neural networks* from *Chapter 11, Machine Learning Best Practices*.

3. Now here comes the recurrent layer, the LSTM layer specifically:

```
>>> model.add(layers.LSTM(50))
```

Here, we only use one recurrent layer with 50 nodes.

- After that, we add the output layer, along with a sigmoid activation function, since we are working on a binary classification problem:

```
>>> model.add(layers.Dense(1, activation='sigmoid'))
```

- Display the model summary to double-check the layers:

```
>>> print(model.summary())
Model: "sequential"

Layer (type)                 Output Shape            Parameters
=====
embedding (Embedding)        (None, None, 32)         1600
lstm (LSTM)                  (None, 50)              1660
dense (Dense)                (None, 1)               51
=====
Total params: 176,651
Trainable params: 176,651
Non-trainable params: 0
```

- Next, we compile the model with the Adam optimizer and use binary cross-entropy as the optimization target:

```
>>> model.compile(loss='binary_crossentropy',
...                  optimizer='adam',
...                  metrics=['accuracy'])
```

- Finally, we train the model with batches of size 64 for three epochs:

```
>>> batch_size = 64
>>> n_epoch = 3
>>> model.fit(X_train, y_train,
...             batch_size=batch_size,
...             epochs=n_epoch,
...             validation_data=(X_test, y_test))
Train on 25000 samples, validate on 25000 samples
Epoch 1/3
391/391 [=====] - 70s 178ms/step -
Epoch 2/3
391/391 [=====] - 69s 176ms/step -
Epoch 3/3
391/391 [=====] - 69s 176ms/step -
```

```

Epoch 3/3
391/391 [=====] - 69s 177ms/step -

```

8. Using the trained model, we evaluate the classification accuracy on the testing set:

```

>>> acc = model.evaluate(X_test, y_test, verbose = 0)[1]
>>> print('Test accuracy:', acc)
Test accuracy: 0.8705199956893921

```

We obtained a test accuracy of 87.05%.

Stacking multiple LSTM layers

Now, let's try to stack two recurrent layers. The following diagram shows how two recurrent layers can be stacked:

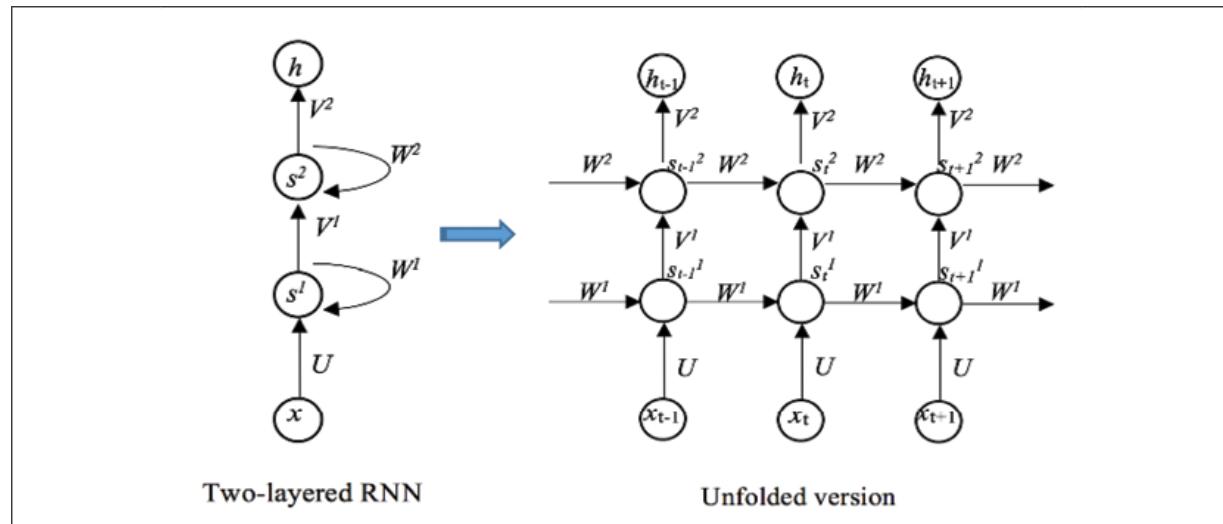


Figure 13.12: Unfolding two stacked recurrent layers

Let's see whether we can beat the previous accuracy by following these steps to build a multi-layer RNN model:

1. Initiate a new model and add an embedding layer, two LSTM layers, and an output layer:

```
>>> model = models.Sequential()
>>> model.add(layers.Embedding(vocab_size, embedding_size))
>>> model.add(layers.LSTM(50, return_sequences=True, dropout=0.2))
>>> model.add(layers.LSTM(50, dropout=0.2))
>>> model.add(layers.Dense(1, activation='sigmoid'))
```

Here, the first LSTM layer comes with `return_sequences=True` as we need to feed its entire output sequence to the second LSTM layer. We also add 20% dropout to both LSTM layers to reduce overfitting since we will have more parameters to train:

```
>>> print(model.summary())
Model: "sequential_1"

Layer (type)                 Output Shape            Parameters
=====
embedding_1 (Embedding)       (None, None, 32)        16000
lstm_1 (LSTM)                (None, None, 50)        16600
lstm_2 (LSTM)                (None, 50)             20200
dense_1 (Dense)              (None, 1)              51
=====
Total params: 196,851
Trainable params: 196,851
Non-trainable params: 0
=====
None
```

- Similarly, we compile the model with the Adam optimizer at a 0.003 learning rate:

```
>>> optimizer = optimizers.Adam(lr=0.003)
>>> model.compile(loss='binary_crossentropy',
...                  optimizer=optimizer,
...                  metrics=['accuracy'])
```

- Then, we train the stacked model for 7 epochs:

```
>>> n_epoch = 7
>>> model.fit(X_train, y_train,
...             batch_size=batch_size,
...             epochs=n_epoch,
```

```
...           validation_data=(X_test, y_test))
Train on 25000 samples, validate on 25000 samples
Epoch 1/7
391/391 [=====] - 139s 356ms/step
Epoch 2/7
391/391 [=====] - 140s 357ms/step
Epoch 3/7
391/391 [=====] - 137s 350ms/step
Epoch 4/7
391/391 [=====] - 136s 349ms/step
Epoch 5/7
391/391 [=====] - 137s 350ms/step
Epoch 6/7
391/391 [=====] - 137s 349ms/step
Epoch 7/7
391/391 [=====] - 138s 354ms/step
```

4. Finally, we verify the test accuracy:

```
>>> acc = model.evaluate(X_test, y_test, verbose=0)[1]
>>> print('Test accuracy with stacked LSTM:', acc)
Test accuracy with stacked LSTM: 0.8755999803543091
```

We obtained a better test accuracy of 87.56%.

With that, we've just finished the review sentiment classification project using RNNs. The RNNs were in the many-to-one structure. In the next project, we will develop an RNN under the many-to-many structure to write a "novel."

Writing your own War and Peace with RNNs

In this project, we'll work on an interesting language modeling problem—text generation.

An RNN-based text generator can write anything, depending on what text we feed it. The training text can be from a novel such as *A Game of*

Thrones, a poem from Shakespeare, or the movie scripts for *The Matrix*. The artificial text that's generated should read similar (but not identical) to the original one if the model is well-trained. In this section, we are going to write our own *War and Peace* with RNNs, a novel written by the Russian author Leo Tolstoy. Feel free to train your own RNNs on any of your favorite books.

We will start with data acquisition and analysis before constructing the training set. After that, we will build and train an RNN model for text generation.

Acquiring and analyzing the training data

I recommend downloading text data for training from books that are not currently protected by copyright. Project Gutenberg (www.gutenberg.org) is a great place for this. It provides over 60,000 free eBooks whose copyright has expired.

The original work, *War and Peace*, can be downloaded from <http://www.gutenberg.org/ebooks/2600>, but note that there will be some cleanup, such as removing the extra beginning section "The Project Gutenberg EBook," the table of contents, and the extra appendix "End of the Project Gutenberg EBook of War and Peace" of the plain text UTF-8 file (<http://www.gutenberg.org/files/2600/2600-0.txt>) required. So, instead of doing this, we will download the cleaned text file directly from https://cs.stanford.edu/people/karpathy/char-rnn/warpeace_input.txt. Let's get started:

1. First, we read the file and convert the text into lowercase:

```
>>> training_file = 'warpeace_input.txt'  
>>> raw_text = open(training_file, 'r').read()  
>>> raw_text = raw_text.lower()
```

2. Then, we take a quick look at the training text data by printing out the first 200 characters:
-

```
>>> print(raw_text[:200])
"well, prince, so genoa and lucca are now just family estates
buonapartes. but i warn you, if you don't tell me that this
if you still try to defend the infamies and horrors perpetr
```

3. Next, we count the number of unique words:

```
>>> all_words = raw_text.split()
>>> unique_words = list(set(all_words))
>>> print(f'Number of unique words: {len(unique_words)}')
Number of unique words: 39830
```

And then, we count the total number of characters:

```
>>> n_chars = len(raw_text)
>>> print(f'Total characters: {n_chars}')
Total characters: 3196213
```

4. From these 3 million characters, we obtain the unique characters, as follows:

```
>>> chars = sorted(list(set(raw_text)))
>>> n_vocab = len(chars)
>>> print(f'Total vocabulary (unique characters): {n_vocab}')
Total vocabulary (unique characters): 57
>>> print(chars)
['\n', ' ', '!', '"', "'", '(', ')', '*', ',', '-', '.', '/
```

The raw training text is made up of 57 unique characters and made up of close to 40,000 unique words. Generating words, which requires computing 40,000 probabilities at one step, is far more difficult than generating characters, which requires computing only 57 probabilities at one step. Hence, we treat a character as a token, and the vocabulary here is composed of 57 characters.

So, how can we feed the characters to the RNN model and generate output characters? Let's see in the next section.

Constructing the training set for the RNN text generator

Recall that in a synced "many-to-many" RNN, the network takes in a sequence and simultaneously produces a sequence; the model captures the relationships among the elements in a sequence and reproduces a new sequence based on the learned patterns. As for our text generator, we can feed in fixed-length sequences of characters and let it generate sequences of the same length, where each output sequence is one character shifted from its input sequence. The following example will help you understand this better:

Say that we have a raw text sample, "learning," and we want the sequence length to be 5. Here, we can have an input sequence, "learn," and an output sequence, "earni." We can put them into the network as follows:

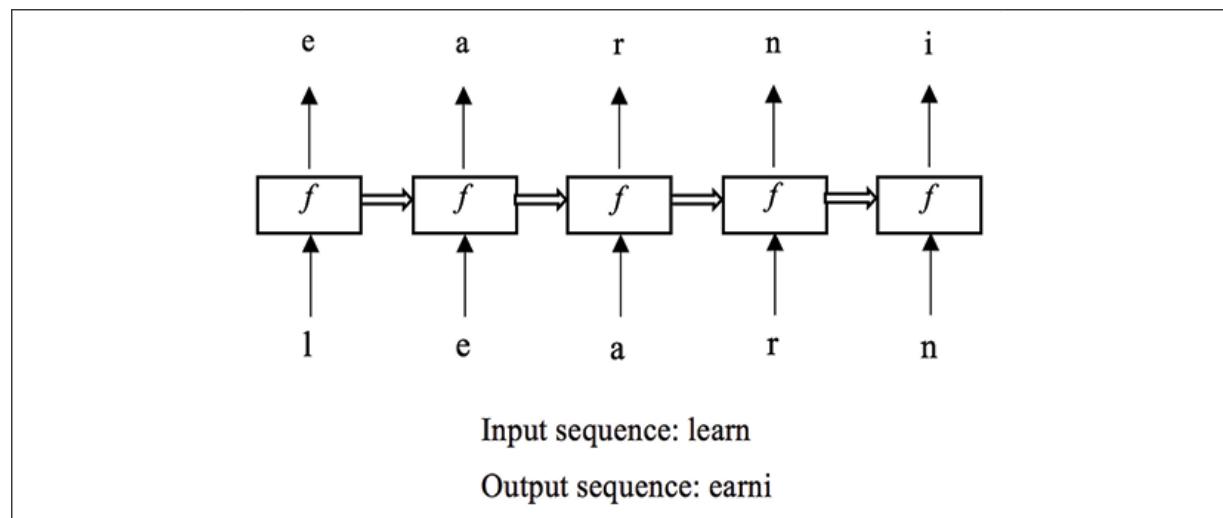


Figure 13.13: Feeding a training set ("learn," "earni") to the RNN

We've just constructed a training sample ("learn," "earni"). Similarly, to construct training samples from the entire original text, first, we need to split the original text into fixed-length sequences, X ; then, we need to ignore the first character of the original text and split shift it into sequences of the same length, Y . A sequence from X is the input of a training sample, while the corresponding sequence from Y is the output of the sample. Let's

say we have a raw text sample, "machine learning by example," and we set sequence length to 5. We will construct the following training samples:

Input	Output
machi	achin
ne□le	e□lea
armin	rning
g□by□	□by□e
examp	xampl

Figure 13.14: Training samples constructed from "machine learning by example"

Here, □ denotes space. Note that the remaining subsequence, "le", is not long enough, so we simply ditch it.

We also need to one-hot encode the input and output characters since neural network models only take in numerical data. We simply map the 57 unique characters to indices from 0 to 56, as follows:

```
>>> index_to_char = dict((i, c) for i, c in enumerate(chars))
>>> char_to_index = dict((c, i) for i, c in enumerate(chars))
>>> print(char_to_index)
{'\n': 0, ' ': 1, '!': 2, "'": 3, '"': 4, '(': 5, ')': 6, '*': 7}
```

For instance, the character "c" becomes a vector of length 57 with "1" in index 28 and "0"s in all other indices; the character "h" becomes a vector of length 57 with "1" in index 33 and "0"s in all other indices.

Now that the character lookup dictionary is ready, we can construct the entire training set, as follows:

```
>>> import numpy as np
>>> seq_length = 160
>>> n_seq = int(n_chars / seq_length)
```

Here, we set the sequence length to `160` and obtain `n_seq` training samples. Next, we initialize the training inputs and outputs, which are both of the shape (number of samples, sequence length, feature dimension):

```
>>> X = np.zeros((n_seq, seq_length, n_vocab))
>>> Y = np.zeros((n_seq, seq_length, n_vocab))
```



RNN models in Keras require the shape of the input and output sequences to be in the shape (number of samples, sequence length, feature dimension).

Now, for each of the `n_seq` samples, we assign "`1`" to the indices of the input and output vectors where the corresponding characters exist:

```
>>> for i in range(n_seq):
...     x_sequence = raw_text[i * seq_length :
...                           (i + 1) * seq_length]
...     x_sequence_ohe = np.zeros((seq_length, n_vocab))
...     for j in range(seq_length):
...         char = x_sequence[j]
...         index = char_to_index[char]
...         x_sequence_ohe[j][index] = 1.
...     X[i] = x_sequence_ohe
...     y_sequence = raw_text[i * seq_length + 1 : (i + 1) *
...                           seq_length + 1]
...     y_sequence_ohe = np.zeros((seq_length, n_vocab))
...     for j in range(seq_length):
...         char = y_sequence[j]
...         index = char_to_index[char]
...         y_sequence_ohe[j][index] = 1.
...     Y[i] = y_sequence_ohe
```

Next, take a look at the shapes of the constructed input and output samples:

```
>>> X.shape
(19976, 160, 57)
>>> Y.shape
(19976, 160, 57)
```

Again, each sample (input or output sequence) is composed of 160 elements. Each element is a 57-dimension one-hot encoded vector.

We finally got the training set ready and it is time to build and fit the RNN model. Let's do this in the next two sections.

Building an RNN text generator

In this section, we will build an RNN with two stacked recurrent layers. This has more predictive power than an RNN with a single recurrent layer for complicated problems such as text generation. Let's get started:

1. First, we import all the necessary modules and fix a random seed:

```
>>> import tensorflow as tf  
>>> from tensorflow.keras import layers, models, losses, optimizers  
>>> tf.random.set_seed(42)
```

2. Each recurrent layer contains 700 units, with a 0.4 dropout ratio and a tanh activation function:

```
>>> hidden_units = 700  
>>> dropout = 0.4
```

3. We specify other hyperparameters, including the batch size, 100, and the number of epochs, 300:

```
>>> batch_size = 100  
>>> n_epoch= 300
```

4. Now, we create the RNN model, as follows:

```
>>> model = models.Sequential()  
>>> model.add(layers.LSTM(hidden_units, input_shape=(None, 160),  
>>> model.add(layers.LSTM(hidden_units, return_sequences=True))  
>>> model.add(layers.TimeDistributed(layers.Dense(n_vocab, activation='softmax')))
```

There are a few things worth looking into:

- `return_sequences=True` for the first recurrent layer: The output of the first recurrent layer is a sequence so that we can stack the second recurrent layer on top.
- `return_sequences=True` for the second recurrent layer: The output of the second recurrent layer is a sequence, which enables the many-to-many structure.
- `Dense(n_vocab, activation='softmax')`: Each element of the output sequence is a one-hot encoded vector, so softmax activation is used to compute the probabilities for individual characters.
- `TimeDistributed`: Since the output of the recurrent layers is a sequence and the `Dense` layer does not take in a sequential input, `TimeDistributed` is used as an adapter so that the `Dense` layer can be applied to every element of the input sequence.

5. Next, we compile the network. As for the optimizer, we choose `RMSprop` with a learning rate of 0.001:

```
>>> optimizer = optimizers.RMSprop(lr=0.001)
>>> model.compile(loss="categorical_crossentropy",
                  optimizer=optimizer)
```

Here, the loss function is multiclass cross-entropy.

6. Let's summarize the model we just built:

```
>>> print(model.summary())
Model: "sequential"

Layer (type)                 Output Shape            Params
=====
lstm (LSTM)                  (None, None, 700)        2122
lstm_1 (LSTM)                 (None, None, 700)        3922
time_distributed (TimeDistr... (None, None, 57)        3995
=====
Total params: 6,085,157
Trainable params: 6,085,157
Non-trainable params: 0
```

With that, we've just finished building and are ready to train the model. We'll do this in the next section.

Training the RNN text generator

As shown in the model summary, we have more than 6 million parameters to train. Hence, it is recommended to train the model on a GPU. If you don't have a GPU in-house, you can use the free GPU provided by Google Colab. You can set it up by following the tutorial at <https://ml-book.now.sh/free-gpu-for-deep-learning/>.

Also, for a deep learning model that requires long training, it is good practice to set up some callbacks in order to keep track of the internal states and performance of the model during training. In our project, we employ the following callbacks:

- **Model checkpoint:** This saves the model after each epoch. If anything goes wrong unexpectedly during training, you don't have to retrain the model. You can simply load the saved model and resume training from there.
- **Early stopping:** We covered this in *Chapter 8, Predicting Stock Prices with Artificial Neural Networks*.
- **Generating text with the latest model on a regular basis:** By doing this, we can see how reasonable the generated text is on the fly.

We employ these three callbacks to train our RNN model as follows:

1. First, we import the necessary modules:

```
>>> from tensorflow.keras.callbacks import Callback, ModelC
```

2. Then, we define the model checkpoint callback:

```
>>> file_path =
        "weights/weights_{epoch:03d}_{loss:.4f}.h5"
>>> checkpoint = ModelCheckpoint(file_path, monitor='loss',
                                verbose=1, save_best_only=True, mode='mi
```

The model checkpoints will be saved with filenames made up of the epoch number and training loss.

- After that, we create an early stopping callback to halt the training if the validation loss doesn't decrease for 50 successive epochs:

```
>>> early_stop = EarlyStopping(monitor='loss', min_delta=0,
                                patience=50, verbose=1, mode:
```

- Next, we develop a helper function that generates text of any length, given a model:

```
>>> def generate_text(model, gen_length, n_vocab, index_to_
...     """
...     Generating text using the RNN model
...     @param model: current RNN model
...     @param gen_length: number of characters we want to generate
...     @param n_vocab: number of unique characters
...     @param index_to_char: index to character mapping
...     @return: string of text generated
...     """
...     # Start with a randomly picked character
...     index = np.random.randint(n_vocab)
...     y_char = [index_to_char[index]]
...     X = np.zeros((1, gen_length, n_vocab))
...     for i in range(gen_length):
...         X[0, i, index] = 1.
...         indices = np.argmax(model.predict(
...             X[:, max(0, i - seq_length - 1):i + 1, :])
...         index = indices[-1]
...         y_char.append(index_to_char[index])
...     return ''.join(y_char)
```

It starts with a randomly picked character. Then, the input model predicts each of the remaining `gen_length-1` characters based on its previously generated characters.

- Now, we define the callback class that generates text with the `generate_text` `util` function for every `N` epochs:

```
>>> class ResultChecker(Callback):
...     def __init__(self, model, N, gen_length):
...         self.model = model
```

```
...     self.N = N
...
...     self.gen_length = gen_length
...
...     def on_epoch_end(self, epoch, logs={}):
...         if epoch % self.N == 0:
...             result = generate_text(self.model,
...                                   self.gen_length, n_vocab, index_to_
...             print('\nMy War and Peace:\n' + result)
...
```

Next, we initiate a text generation checker callback:

```
>>> result_checker = ResultChecker(model, 10, 500)
```

The model will generate text of 500 characters for every 10 epochs.

6. Now that all the callback components are ready, we can start training the model:

```
>>> model.fit(X, Y, batch_size=batch_size,
    verbose=1, epochs=n_epoch, callbacks=[  
        result_checker, checkpoint, early_stop])
```

I will only demonstrate the results for epochs 1, 51, 101, and 291 here:

Epoch 1:

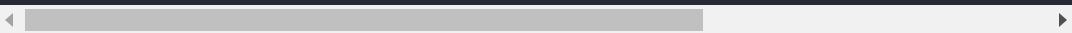
Epoch 51:

```
Epoch 51/300
200/200 [=====] - ETA: 0s - loss: 1.74298
My War and Peace:
re and the same time the same time the same time he had not
Epoch 00051: loss improved from 1.74371 to 1.74298, saving 1
200/200 [=====] - 64s 321ms/step -
```

Epoch 101:

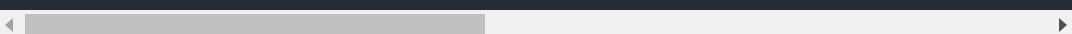
```
Epoch 101/300
200/200 [=====] - ETA: 0s - loss: 0.0000 - acc: 1.0000
My War and Peace:
```

```
's and the same time and the same sense of his life and her  
"what is it?" asked Natasha. "I have not the post and the sa-  
her and will not be able to say something to her and went to  
the door.  
"what is it?" asked Natasha. "I have not the post and the sa-  
and that I shall not be able to say something to her and  
went on to the door.  
"what a strange in the morning, I am so all to say something  
said Prince Andrew, "I have not the post and the  
same  
Epoch 00101: loss did not improve from 1.68711  
200/200 [=====] - 64s 321ms/step -
```



Epoch 291:

```
Epoch 291/300  
200/200 [=====] - ETA: 0s - loss: 1.61188  
My War and Peace:  
à to the countess, who was sitting in the same way the sound  
"what are you doing?" said the officer, turning to the prince  
a smile.  
"I don't know what to say and want to see you."  
"yes, yes," said Prince Andrew, "I have not been the first  
and you will be a little better than you are and we will be  
married. what a sin I want to see you."  
"yes, yes," said Prince Andrew, "I have not been the first  
Epoch 00291: loss did not improve from 1.61188  
200/200 [=====] - 65s 323ms/step -
```



Each epoch takes around 60 seconds on a Tesla K80 GPU. After a couple of hours of training, the RNN-based text generator can write a realistic and interesting version of *War and Peace*. With that, we've successfully used a many-to-many type of RNN to generate text.

An RNN with a many-to-many structure is a type of sequence-to-sequence (seq2seq) model that takes in a sequence and outputs another sequence. A typical example is machine translation, where a sequence of words from one language is transformed into a sequence in another language. The state-of-the-art seq2seq model is the Transformer model, and it was developed by Google Brain. We will briefly discuss it in the next section.

Advancing language understanding with the Transformer model

The `Transformer` model was first proposed in *Attention Is All You Need* (<https://arxiv.org/abs/1706.03762>). It can effectively handle long-term dependencies, which are still challenging in LSTM. In this section, we will go through the Transformer's architecture and building blocks, as well as its most crucial part: the self-attention layer.

Exploring the Transformer's architecture

We'll start by looking at the high-level architecture of the Transformer model (image taken from *Attention Is All You Need*):

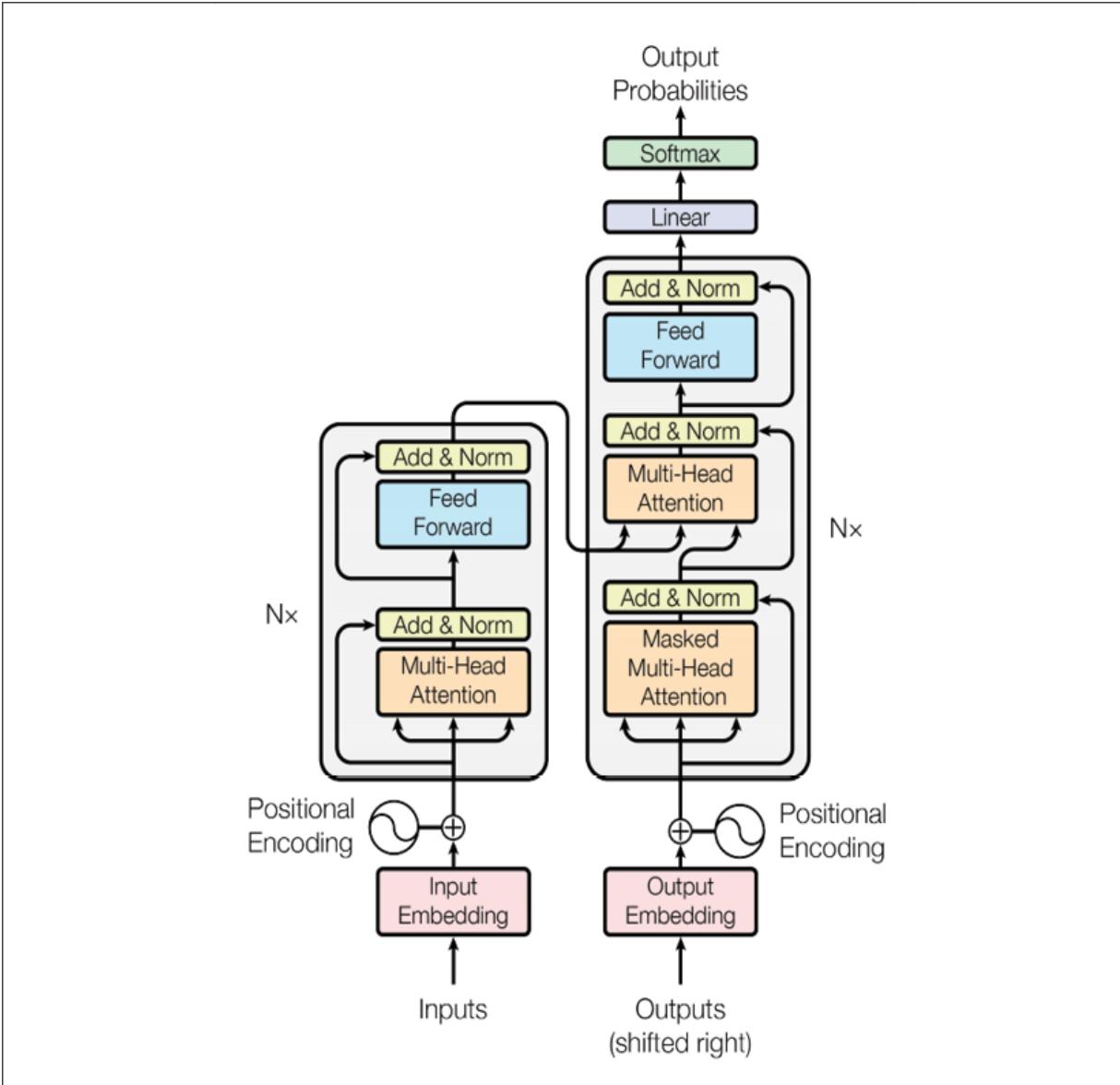


Figure 13.15: Transformer architecture

As you can see, the Transformer consists of two parts: the **encoder** (the big rectangle on the left-hand side) and the **decoder** (the big rectangle on the right-hand side). The encoder encrypts the input sequence. It has a multi-head attention layer (we will talk about this next) and a regular feedforward layer. On the other hand, the decoder generates the output sequence. It has a masked multi-head attention layer, along with a multi-head attention layer and a regular feedforward layer.

At step t , the Transformer model takes in input steps x_1, x_2, \dots, x_t and output steps y_1, y_2, \dots, y_{t-1} . It then predicts y_t . This is no different from the many-to-many RNN model.

The multi-head attention layer is probably the only thing that looks strange to you, so we'll take a look at it in the next section.

Understanding self-attention

Let's discuss how the **self-attention** layer plays a key role in the Transformer in the following example:

"I read Python Machine Learning by Example and it is indeed a great book." Apparently, it refers to *Python Machine Learning by Example*. When the Transformer model processes this sentence, self-attention will associate it with *Python Machine Learning by Example*. Given a word in an input sequence, self-attention allows the model to look at the other words in the sequence at different attention levels, which boosts language understanding and learning in `seq2seq` tasks.

Now, let's see how we calculate the attention score.

As shown by the architecture diagram, there are three input vectors to the attention layer:

- The query vector, Q , which represents the query word (that is, the current word) in the sequence
- The key vector, K , which represents individual words in the sequence
- The value vector, V , which also represents individual words in the sequence

These three vectors are trained during training.

The output of the attention layer is calculated as follows:

$$\text{attention}(Q, K, V) = V \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right)$$

Here, d_k is the dimension of the key vector. Take the sequence *python machine learning by example* as an example; we take the following steps to calculate the self-attention for the first word, *python*:

1. We calculate the dot products between each word in the sequence and the word *python*. They are $q_1, k_1, q_1, k_2, q_1, k_3, q_1, k_4$, and q_1, k_5 . Here, q_1 is the query vector for the first word and k_1 to k_5 are the key vectors for the five words, respectively.
2. We normalize the resulting dot products with division and softmax activation:

$$\begin{aligned}s_{11} &= \text{softmax} \left(\frac{q_1 \cdot k_1}{\sqrt{d_k}} \right) \\ s_{12} &= \text{softmax} \left(\frac{q_1 \cdot k_2}{\sqrt{d_k}} \right) \\ &\dots \\ s_{15} &= \text{softmax} \left(\frac{q_1 \cdot k_5}{\sqrt{d_k}} \right)\end{aligned}$$

3. Then, we multiply the resulting softmax vectors by the value vectors, v_1, v_2, \dots, v_5 , and sum up the results:

$$z_1 = s_{11} \cdot v_1 + s_{12} \cdot v_2 + s_{13} \cdot v_3 + s_{14} \cdot v_4 + s_{15} \cdot v_5$$

z_1 is the self-attention score for the first word, *python*, in the sequence. We repeat this process for each remaining word in the sequence to obtain its attention score. Now, you should understand why this is called **multi-head attention**: self-attention is not computed for just one word (one step), but for all words (all steps).

All output attention scores are then concatenated and fed into the downstream regular feedforward layer.

In this section, we have covered the main concepts of the Transformer model. It has become the model of choice for many complicated problems in NLP, such as speech to text, text summarization, and question answering. With added attention mechanisms, the Transformer model can effectively handle long-term dependencies in sequential learning. Moreover, it allows parallelization during training since self-attention can be computed independently for individual steps.

If you are interested in reading more, here are some recent developments that have been made using Transformer:

- **Bidirectional Encoder Representations from Transformers (BERT)**, developed by Google (<https://arxiv.org/abs/1810.04805v2>)
- **Generative Pre-training Transformer (GPT)**, proposed by OpenAI (https://s3-us-west-2.amazonaws.com/openai-assets/research-covers/language-unsupervised/language_understanding_paper.pdf)
- Object detection with Transformers (<https://ai.facebook.com/blog/end-to-end-object-detection-with-transformers/>)

Summary

In this chapter, we worked on two NLP projects: sentiment analysis and text generation using RNNs. We started with a detailed explanation of the recurrent mechanism and different RNN structures for different forms of input and output sequences. You also learned how LSTM improves vanilla RNNs. Finally, as a bonus section, we covered the Transformer, a recent state-of-the-art sequential learning model.

In the next chapter, we will focus on the third type of machine learning problem: reinforcement learning. You will learn how the reinforcement

learning model learns by interacting with the environment to reach the learning goal.

Exercises

1. Use a bi-directional recurrent layer (it is easy enough to learn about it by yourself) and apply it to the sentiment analysis project. Can you beat what we achieved? Read https://www.tensorflow.org/api_docs/python/tf/keras/layers/Bidirectional if you want to see an example.
2. Feel free to fine-tune the hyperparameters, as we did in *Chapter 8, Predicting Stock Prices with Artificial Neural Networks*, and see whether you can improve the classification performance further.

14

Making Decisions in Complex Environments with Reinforcement Learning

In the previous chapter, we focused on RNNs for sequential learning. The last chapter of the book will be about reinforcement learning, which is the third type of machine learning task mentioned at the beginning of the book. You will see how learning from experience and learning by interacting with the environment differs from previously covered supervised and unsupervised learning.

We will cover the following topics in this chapter:

- Setting up a workspace for reinforcement learning
- Basics of reinforcement learning
- Simulation of OpenAI Gym environments
- Value iteration and policy iteration algorithms
- Monte Carlo methods for policy evaluation and control
- The Q-learning algorithm

Setting up the working environment

Let's get started with setting up the working environment, including PyTorch as the main framework, and OpenAI Gym, the toolkit that gives you a variety of environments to develop your learning algorithms on.

Installing PyTorch

PyTorch (<https://pytorch.org/>) is a trendy machine learning library developed on top of Torch (<http://torch.ch/>) by Facebook's AI lab. It provides powerful computational graphs and high compatibility to GPUs, as well as a simple and friendly interface. PyTorch is expanding quickly in academia and it has seen heavy adoption by more and more companies. The following chart (taken from <http://horace.io/pytorch-vs-tensorflow/>) shows the growth of PyTorch at top machine learning conferences:

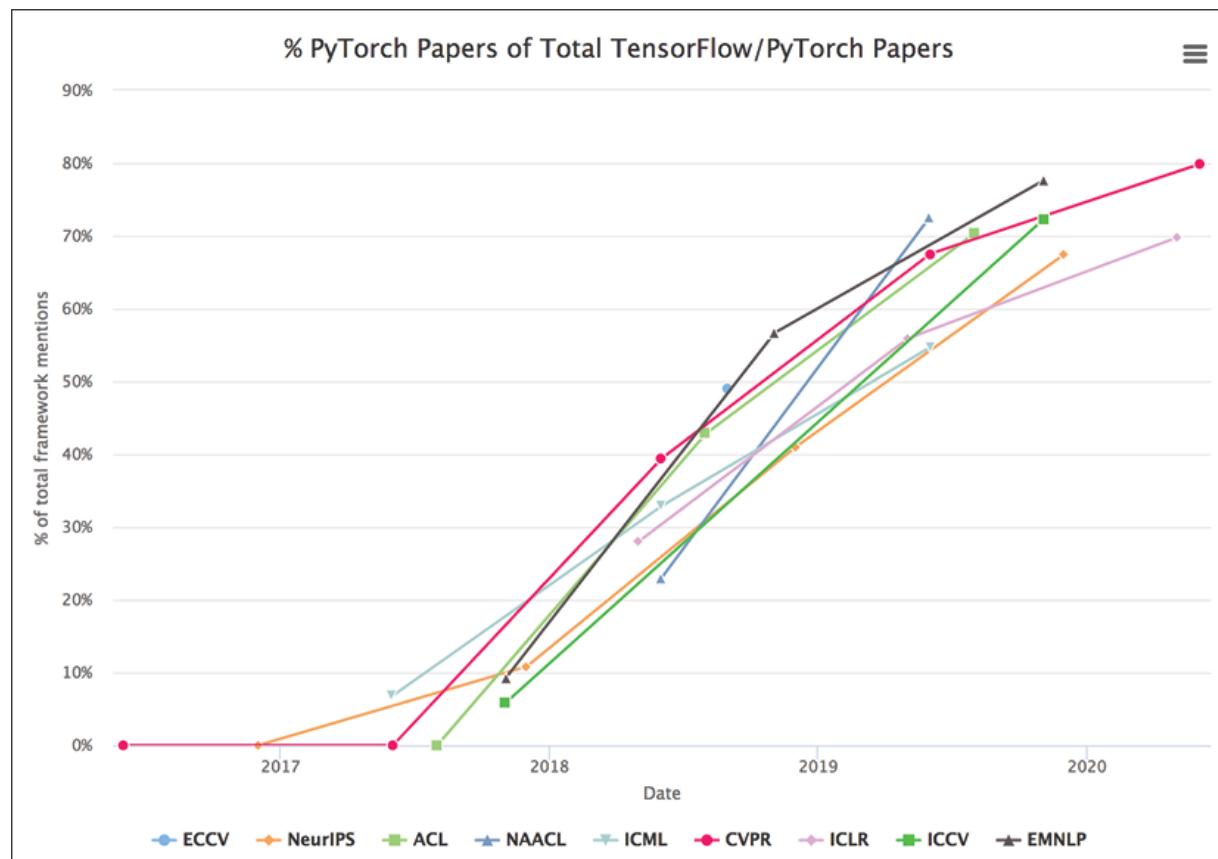


Figure 14.1: Number of PyTorch papers in top machine learning conferences

In the past year, there have been more mentions of PyTorch than TensorFlow at those conferences. Hopefully, you are motivated enough to work with PyTorch. Now let's see how to properly install it.

Firstly, in the following table on the page <https://pytorch.org/get-started/locally/>, you can pick the right configurations for your environment:

PyTorch Build	Stable (1.5.1)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python			C++ / Java
CUDA	9.2	10.1	10.2	None
Run this Command:	conda install pytorch torchvision -c pytorch			

Figure 14.2: Installing PyTorch with system configurations

Here, I use Mac, Conda, and Python 3.7 running locally (no CUDA) as an example, and run the suggested command line:

```
conda install pytorch torchvision -c pytorch
```

Next, you can run the following lines of code in Python to confirm correct installation:

```
>>> import torch
>>> x = torch.empty(3, 4)
>>> print(x)
tensor([[7.8534e+34, 4.7418e+30, 5.9663e-02, 7.0374e+22],
       [3.5788e+01, 4.5825e-41, 4.0272e+01, 4.5825e-41],
       [0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00]])
```

Here, the tensor in PyTorch is similar to the ndarrays in NumPy, or the tensor in TensorFlow. We have just created a tensor of size 3 * 4. It is an

empty matrix with a bunch of meaningless placeholder floats. Again, this is very similar to NumPy's empty array.

If you want to get more familiar with PyTorch, you can go through the *Getting Started* sections in the official tutorial

<https://pytorch.org/tutorials/#getting-started>. I recommend you at least finish these two:

- What is PyTorch?:
https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py
- Learning PyTorch with examples:
https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

By now, we have successfully set up PyTorch. Let's look at installing OpenAI Gym in the next section.



You are not limited to PyTorch for reinforcement learning. TensorFlow is always a good option. It's just beneficial to learn the trending framework PyTorch in the last chapter of this book.

Installing OpenAI Gym

OpenAI Gym (<https://gym.openai.com/>) is a powerful open source toolkit for developing and comparing reinforcement learning algorithms. It provides a variety of environments to develop your reinforcement learning algorithms on. It is developed by **OpenAI** (<https://openai.com/>), a non-profit research company focused on building safe and beneficial **Artificial General Intelligence (AGI)**.

There are two ways to install Gym. The first one is via `pip`, as follows:

```
pip install gym
```

Another approach is to build from source by cloning the package from its Git repository and installing it from there:

```
git clone https://github.com/openai/gym  
cd gym  
pip install -e .
```

After the installation, you can check the available Gym environment by running the following code:

```
>>> from gym import envs  
>>> print(envs.registry.all())  
dict_values([EnvSpec(Copy-v0), EnvSpec(RepeatCopy-v0), EnvSpec(F
```



You can see the full list of environments at <https://gym.openai.com/envs/>, including walking, moon landing, car racing, and Atari games. Feel free to play around with Gym.

When benchmarking different reinforcement learning algorithms, we need to apply them in a standardized environment. Gym is a perfect place with a number of versatile environments. This is similar to using datasets such as MNIST, ImageNet, and Thomson Reuters News as benchmarks in supervised and unsupervised learning.

Gym has an easy-to-use interface for the reinforcement learning environments, which we can write **agents** to interact with. So what's reinforcement learning? What's an agent? Let's see in the next section.

Introducing reinforcement learning with examples

In this chapter, I will first introduce the elements of reinforcement learning along with an interesting example, then will move on to how we measure feedback from the environment, and follow with the fundamental approaches to solve reinforcement learning problems.

Elements of reinforcement learning

You may have played Super Mario (or Sonic) when you were young. During the video game, you control Mario to collect coins and avoid obstacles at the same time. The game ends if Mario hits an obstacle or falls in a gap. And you try to get as many coins as possible before the game ends.

Reinforcement learning is very similar to the Super Mario game. Reinforcement learning is about learning what to do. It observes the situations in the environment and determines the right actions in order to maximize a numerical reward. Here is the list of elements in a reinforcement learning task (I also link each element to Super Mario and other examples so it's easier to understand):

- **Environment:** The environment is a task or simulation. In the Super Mario game, the game itself is the environment. In self-driving, the road and traffic are the environment. In AlphaGo playing chess, the board is the environment. The inputs to the environment are the actions sent from the **agent** and the outputs are **states** and **rewards** sent to the agent.
- **Agent:** The agent is the component that takes **actions** according to the reinforcement learning model. It interacts with the environment and observes the states to feed into the model. The goal of the agent is to solve the environment—finding the best set of actions to maximize the rewards. The agent in the Super Mario game is Mario, and the autonomous vehicle is for self-driving.
- **Action:** This is the possible movement of the agent. It is usually random in a reinforcement learning task at the beginning when the model starts to learn about the environment. Possible actions for Mario include moving left and right, jumping, and crouching.

- **States:** The states are the observations from the environment. They describe the situation in a numerical way at every time step. For the chess game, the state is the positions of all the pieces on the board. For Super Mario, the state includes the coordinates of Mario and other elements in the time frame. For a robot learning to walk, the position of its two legs is the state.
- **Rewards:** Every time the agent takes an action, it receives numerical feedback from the environment. The feedback is called the **reward**. It can be positive, negative, or zero. The reward in the Super Mario game can be, for example, +1 if Mario collects a coin, +2 if he avoids an obstacle, -10 if he hits an obstacle, or 0 for other cases.

The following diagram summarizes the process of reinforcement learning:

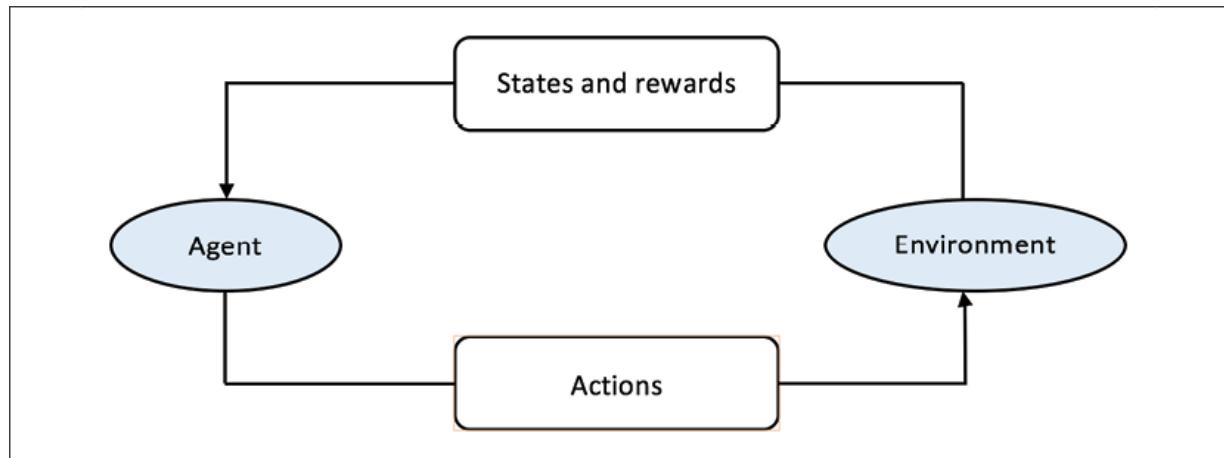


Figure 14.3: Reinforcement learning process

The reinforcement learning process is an iterative loop. At the beginning, the agent observes the initial state, s_0 , from the environment. Then the agent takes an action, a_0 , according to the model. After the agent moves, the environment is now in a new state, s_1 , and it gives a feedback reward, R_1 . The agent then takes an action, a_1 , as computed by the model with inputs s_1 and R_1 . This process continues until termination, completion, or for forever.

The goal of the reinforcement learning model is to maximize the total reward. So how can we calculate the total reward? Is it simply by summing up rewards at all the time steps? Let's see in the next section.

Cumulative rewards

At time step t , the **cumulative rewards** (also called **returns**) G_t can be written as:

$$G_t = \sum_{k=0}^T R_{t+k+1}$$

Here, T is the termination time step or infinity. G_t means the total future reward after taking an action a_t at time t . At each time step t , the reinforcement learning model attempts to learn the best possible action in order to maximize G_t .

However, in many real-world cases, things don't work this way where we simply sum up all future rewards. Take a look at the following example:

Stock A rises 6 dollars at the end of day 1 and falls 5 dollars at the end of day 2. Stock B falls 5 dollars on day 1 and rises 6 dollars on day 2. After two days, both stocks rise 1 dollar. So which one will you buy at the beginning of day 1? Obviously stock A because you won't lose money and can even profit 6 dollars if you sell it at the beginning of day 2.

Both stocks have the same total reward but we favor stock A as we care more about immediate return than distant return. Similarly in reinforcement learning, we discount rewards in the distant future and the discount factor is associated with the time horizon. Longer time horizons should have less impact on the cumulative rewards. This is because longer time horizons include more irrelevant information and consequently are of higher variance.

We define a discount factor γ with a value between 0 and 1. We rewrite the cumulative rewards incorporating the discount factor:

$$G_t = \sum_{k=0}^T \gamma^k R_{t+k+1}$$

As you can see, the larger the γ , the smaller the discount and vice versa. If $\gamma = 1$, there is literally no discount and the model evaluates an action based on the sum total of all future rewards. If $\gamma = 0$, the model only focuses on the immediate reward R_{t+1} .

Now that we know how to calculate the cumulative reward, the next thing to talk about is how to maximize it.

Approaches to reinforcement learning

There are mainly two approaches to solving reinforcement learning problems, which are about finding the optimal actions to maximize the cumulative rewards. One is a policy-based approach and another one is value-based.

A **policy** is a function π that maps each input state to an action:

$$a = \pi(s)$$

It can be either deterministic or stochastic:

- **Deterministic:** There is one-to-one mapping from the input state to the output action
- **Stochastic:** It gives a probability distribution over all possible actions $P(A = a|s)$

In the **policy-based** approach, the model learns the optimal policy that maps each input state to the best action.

The **value** V of a state is defined as the expected future cumulative reward to collect from the state:

$$V(s) = E \left[\sum_{k=0}^T \gamma^k R_{t+k+1} | s_t = s \right]$$

In the **value-based** approach, the model learns the optimal value function that maximizes the value of the input state. In other words, the agent takes an action to reach the state that achieves the largest value.

In a policy-based algorithm, the model starts with a random policy. It then computes the value function of that policy. This step is called the **policy evaluation step**. After this, it finds a new and better policy based on the value function. This is the policy improvement step. These two steps repeat until the optimal policy is found. Whereas in a value-based algorithm, the model starts with a random value function. It then finds a new and improved value function in an iterative manner, until it reaches the optimal value function.

We've learned there are two main approaches to solve reinforcement learning problems. In the next section, let's see how to solve a concrete reinforcement learning example (FrozenLake) using a concrete algorithm, the dynamic programming method, in a policy-based and value-based way respectively.

Solving the FrozenLake environment with dynamic programming

We will focus on the policy-based and value-based dynamic programming algorithms in this section. But let's start with simulating the FrozenLake environment.

Simulating the FrozenLake environment

FrozenLake is a typical OpenAI Gym environment with **discrete** states. It is about moving the agent from the starting tile to the destination tile in a grid,

and at the same time avoiding traps. The grid is either $4 * 4$ (<https://gym.openai.com/envs/FrozenLake-v0/>), or $8 * 8$ (<https://gym.openai.com/envs/FrozenLake8x8-v0/>). There are four types of tiles in the grid:

- **S**: The starting tile. This is state 0, and it comes with 0 reward.
- **G**: The goal tile. It is state 15 in the $4 * 4$ grid. It gives +1 reward and terminates an episode.
- **F**: The frozen tile. In the $4 * 4$ grid, states 1, 2, 3, 4, 6, 8, 9, 10, 13, and 14 are walkable tiles. It gives 0 reward.
- **H**: The hole tile. In the $4 * 4$ grid, states 5, 7, 11, and 12 are hole tiles. It gives 0 reward and terminates an episode.

Here, an **episode** means a simulation of a reinforcement learning environment. It contains a list of states from the initial state to the terminal state, a list of actions and rewards. In the $4 * 4$ FrozenLake environment, there are 16 possible states as the agent can move to any of the 16 tiles. And there are four possible actions: moving left (0), down (1), right (2), and up (3).

The tricky part of this environment is that, as the ice surface is slippery, the agent won't always move in the direction it intends and can move in any other walkable directions or stay unmoved at some probabilities. For example, it may move to the right even though it intends to move up.

Now let's simulate the $4 * 4$ FrozenLake environment by following these steps:

1. To simulate any OpenAI Gym environment, we need to first look up its name in the table <https://github.com/openai/gym/wiki/Table-of-environments>. We get "FrozenLake-v0" in our case.
2. We import the Gym library and create a FrozenLake instance:

```
>>> import gym  
>>> env = gym.make("FrozenLake-v0")  
>>> n_state = env.observation_space.n  
>>> print(n_state)
```

```
16
>>> n_action = env.action_space.n
>>> print(n_action)
4
```

We also obtain the dimensions of the environment.

3. Every time we run a new episode, we need to reset the environment:

```
>>> env.reset()
0
```

It means that the agent starts with state 0. Again, there are 16 possible states, 0, 1, ..., 15.

4. We render the environment to display it:

```
>>> env.render()
```

You will see a $4 * 4$ matrix representing the FrozenLake grid and the tile (state 0) where the agent is located:

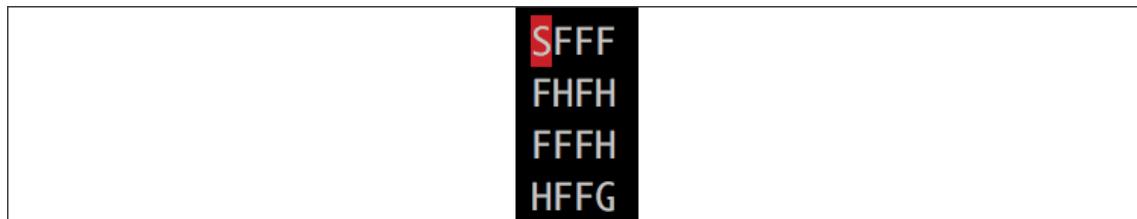


Figure 14.4: Initial state of FrozenLake

5. Let's take a right action since it is walkable:

```
>>> new_state, reward, is_done, info = env.step(2)
>>> print(new_state)
1
>>> print(reward)
0.0
>>> print(is_done)
False
>>> print(info)
{'prob': 0.3333333333333333}
```

The agent moves right to state 1, at a probability of 33.33%, and gets 0 reward since the episode is not done yet. Also see the render result:

```
>>> env.render()
```

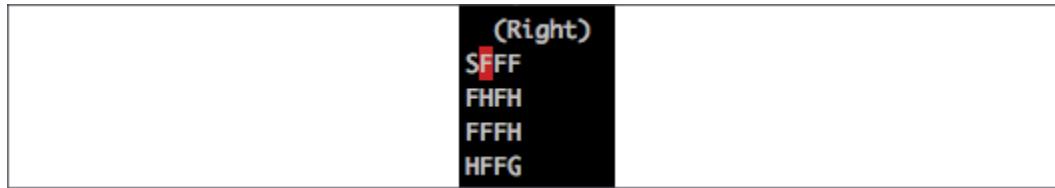


Figure 14.5: Result of the agent moving right

You may get a completely different result as the agent can move down to state 4 at a probability of 33.33%, or stay at state 0 at a probability of 33.33%.

6. Next, we define a function that simulates a FrozenLake episode under a given policy and returns the total reward (as an easy start, let's just assume discount factor $\gamma = 1$):

```
>>> def run_episode(env, policy):
...     state = env.reset()
...     total_reward = 0
...     is_done = False
...     while not is_done:
...         action = policy[state].item()
...         state, reward, is_done, info = env.step(action)
...         total_reward += reward
...         if is_done:
...             break
...     return total_reward
```

Here, `policy` is a PyTorch tensor, and `.item()` extracts the value of an element on the tensor.

7. Now let's play around with the environment using a random policy. We will implement a random policy (where random actions are taken) and calculate the average total reward over 1,000 episodes:

```
>>> n_episode = 1000
>>> total_rewards = []
>>> for episode in range(n_episode):
...     random_policy = torch.randint(high=n_action, size=(1))
...     total_reward = run_episode(env, random_policy)
...     total_rewards.append(total_reward)
...
>>> print(f'Average total reward under random policy: {sum(total_rewards) / n_episode}')
Average total reward under random policy: 0.014
```

On average, there is a 1.4% chance that the agent can reach the goal if we take random actions. This tells us it is not as easy to solve the FrozenLake environment as you might think.

8. As a bonus step, you can look into the transition matrix. The **transition matrix** $T(s, a, s')$ contains probabilities of taking action a from state s then reaching s' . Take state 6 as an example:

```
>>> print(env.env.P[6])
{0: [(0.3333333333333333, 2, 0.0, False), (0.3333333333333333,
```

The keys of the returning dictionary 0, 1, 2, 3 represent four possible actions. The value of a key is a list of tuples associated with the action. The tuple is in the format of (transition probability, new state, reward, is terminal state or not). For example, if the agent intends to take action 1 (down) from state 6, it will move to state 5 (H) with 33.33% probability and receive 0 reward and the episode will end consequently; it will move to state 10 with 33.33% probability and receive 0 reward; it will move to state 7 (H) with 33.33% probability and receive 0 reward and terminate the episode.

We've experimented with the random policy in this section, and we only succeeded 1.4% of the time. But this gets you ready for the next section where we will find the optimal policy using the value-based dynamic programming algorithm, called the **value iteration algorithm**.

Solving FrozenLake with the value iteration algorithm

Value iteration is an iterative algorithm. It starts with random policy values, V , and then iteratively updates the values based on the **Bellman optimality equation** (https://en.wikipedia.org/wiki/Bellman_equation) until the values converge.



It is usually difficult for the values to completely converge. Hence, there are two criteria of convergence. One is passing a fixed number of iterations, such as 1,000 or 10,000. Another one is specifying a threshold (such as 0.0001, or 0.00001) and we terminate the process if the changes of all values are less than the threshold.

Importantly, in each iteration, instead of taking the expectation (average) of values across all actions, it picks the action that maximizes the policy values. The iteration process can be expressed as follows:

$$V^*(s) := \max_a \left[R(s, a, s') + \gamma \sum_{s'} T(s, a, s') V^*(s') \right]$$

Here, $V^*(s)$ is the optimal value function; $T(s, a, s')$ denotes the transition probability of moving to state s' from state s by taking action a ; and $R(s, a, s')$ is the reward provided in state s' by taking action a .

Once we obtain the optimal values, we can easily compute the optimal policy accordingly:

$$\pi^*(s) := \operatorname{argmax}_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

Let's solve the FrozenLake environment using the value iteration algorithm as follows:

1. First we set 0.99 as the discount factor, and 0.0001 as the convergence threshold:

```
>>> gamma = 0.99
>>> threshold = 0.0001
```

2. We develop the value iteration algorithm, which computes the optimal values:

```
>>> def value_iteration(env, gamma, threshold):
...     """
...     Solve a given environment with value iteration algo
... 
```

```

...
    @param env: OpenAI Gym environment
    @param gamma: discount factor
    @param threshold: the evaluation will stop once value
    @return: values of the optimal policy for the given
    """
...
    n_state = env.observation_space.n
    n_action = env.action_space.n
    V = torch.zeros(n_state)
    while True:
        V_temp = torch.empty(n_state)
        for state in range(n_state):
            v_actions = torch.zeros(n_action)
            for action in range(n_action):
                for trans_prob, new_state, reward, _ in
                    env.env.P[state][action]:
                    v_actions[action] += trans_prob * (
                        reward + gamma * V[new_
...
                    V_temp[state] = torch.max(v_actions)
                    max_delta = torch.max(torch.abs(V - V_temp))
                    V = V_temp.clone()
                    if max_delta <= threshold:
                        break
    return V

```

The `value_iteration` function does the following tasks:

- Starting with policy values as all 0s
- Updating the values based on the Bellman optimality equation
- Computing the maximal change of the values across all states
- Continuing updating the values, if the maximal change is greater than the convergence threshold
- Otherwise, terminating the iteration process and returning the last values as the optimal values

3. We apply the algorithm to solve the FrozenLake environment along with the specified parameters:

```

>>> V_optimal = value_iteration(env, gamma, threshold)
Take a look at the resulting optimal values:
>>> print('Optimal values:\n', V_optimal)
Optimal values:
tensor([0.5404, 0.4966, 0.4681, 0.4541, 0.5569, 0.0000, 0.31

```

4. Since we have the optimal values, we can extract the optimal policy from the values. We develop the following function to do this:

```
>>> def extract_optimal_policy(env, V_optimal, gamma):
...     """
...     Obtain the optimal policy based on the optimal values
...     @param env: OpenAI Gym environment
...     @param V_optimal: optimal values
...     @param gamma: discount factor
...     @return: optimal policy
...     """
...     n_state = env.observation_space.n
...     n_action = env.action_space.n
...     optimal_policy = torch.zeros(n_state)
...     for state in range(n_state):
...         v_actions = torch.zeros(n_action)
...         for action in range(n_action):
...             for trans_prob, new_state, reward, _ in
...                 env.env.P[state][action]
...             v_actions[action] += trans_prob * (
...                 reward + gamma * V_optimal[new_s])
...     optimal_policy[state] = torch.argmax(v_actions)
...     return optimal_policy
```

5. Then we obtain the optimal policy based on the optimal values:

```
>>> optimal_policy = extract_optimal_policy(env, V_optimal,
```

Take a look at the resulting optimal policy:

```
>>> print('Optimal policy:\n', optimal_policy)
```

This means the optimal action in state 0 is 0 (left), 3 (up) in state 1, etc. This doesn't look very intuitive if you look at the grid. But remember that the grid is slippery and the agent can move in another direction than the desired one.

6. If you doubt that it is the optimal policy, you can run 1,000 episodes with the policy and gauge how good it is by checking the average reward, as follows:

```

>>> n_episode = 1000
>>> total_rewards = []
>>> for episode in range(n_episode):
...     total_reward = run_episode(env, optimal_policy)
...     total_rewards.append(total_reward)

```

Here, we reuse the `run_episode` function we defined in the previous section. Then we print out the average reward:

```

>>> print('Average total reward under the optimal policy:',
Average total reward under the optimal policy: 0.75

```

Under the optimal policy computed by the value iteration algorithm, the agent reaches the goal tile 75% of the time. Can we do something similar with the policy-based approach? Let's see in the next section.

Solving FrozenLake with the policy iteration algorithm

The **policy iteration** algorithm has two components, policy evaluation and policy improvement. Similar to value iteration, it starts with an arbitrary policy and follows with a bunch of iterations.

In the policy evaluation step in each iteration, we first compute the values of the latest policy, based on the Bellman expectation equation:

$$V(s) := \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')]$$

In the policy improvement step, we derive an improved policy based on the latest policy values, again based on the Bellman optimality equation:

$$\pi(s) := \operatorname{argmax}_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')]$$

These two steps repeat until the policy converges. At convergence, the latest policy and its value are the optimal policy and the optimal value.

Let's develop the policy iteration algorithm and use it to solve the FrozenLake environment as follows:

1. We start with the `policy_evaluation` function that computes the values of a given policy:

```
>>> def policy_evaluation(env, policy, gamma, threshold):
...     """
...         Perform policy evaluation
...         @param env: OpenAI Gym environment
...         @param policy: policy matrix containing actions and
...             their probability in each state
...         @param gamma: discount factor
...         @param threshold: the evaluation will stop once val
...             for all states are less than the threshold
...         @return: values of the given policy
...
...     """
...     n_state = policy.shape[0]
...     V = torch.zeros(n_state)
...     while True:
...         V_temp = torch.zeros(n_state)
...         for state in range(n_state):
...             action = policy[state].item()
...             for trans_prob, new_state, reward, _ in \
...                 env.env.P[state][action]:
...                 V_temp[state] += trans_prob * (
...                     reward + gamma * V[new_
...             max_delta = torch.max(torch.abs(V - V_temp)))
...             V = V_temp.clone()
...             if max_delta <= threshold:
...                 break
...     return V
```

The function does the following tasks:

- Initializing the policy values with all 0s
- Updating the values based on the Bellman expectation equation
- Computing the maximal change of the values across all states
- If the maximal change is greater than the threshold, keeping updating the values

- Otherwise, terminating the evaluation process and returning the latest values
2. Next, we develop the second component, the policy improvement, in the following function:

```
>>> def policy_improvement(env, V, gamma):
...     """
...         Obtain an improved policy based on the values
...         @param env: OpenAI Gym environment
...         @param V: policy values
...         @param gamma: discount factor
...         @return: the policy
...
...     """
...     n_state = env.observation_space.n
...     n_action = env.action_space.n
...     policy = torch.zeros(n_state)
...     for state in range(n_state):
...         v_actions = torch.zeros(n_action)
...         for action in range(n_action):
...             for trans_prob, new_state, reward, _ in
...                 env.env.P[state][action]:
...                 v_actions[action] += trans_prob * (
...                     reward + gamma * V[new_state])
...         policy[state] = torch.argmax(v_actions)
...     return policy
```

It derives a new and better policy from the input policy values, based on the Bellman optimality equation.

3. With both components ready, we now develop the whole policy iteration algorithm:

```
>>> def policy_iteration(env, gamma, threshold):
...     """
...         Solve a given environment with policy iteration alg
...         @param env: OpenAI Gym environment
...         @param gamma: discount factor
...         @param threshold: the evaluation will stop once val
...         @return: optimal values and the optimal policy for
...
...     """
...     n_state = env.observation_space.n
...     n_action = env.action_space.n
...     policy = torch.randint(high=n_action,
...                           size=(n_state,)).float()
...     while True:
```

```
...     v = policy_evaluation(env, policy, gamma, threshold)
...     policy_improved = policy_improvement(env, V, gain)
...     if torch.equal(policy_improved, policy):
...         return V, policy_improved
...     policy = policy_improved
```

This function does the following tasks:

- Initializing a random policy
 - Performing policy evaluation to update the policy values
 - Performing policy improvement to generate a new policy
 - If the new policy is different from the old one, updating the policy and running another iteration of policy evaluation and improvement
 - Otherwise, terminating the iteration process and returning the latest policy and its values

4. Next, we use policy iteration to solve the FrozenLake environment:

```
>>> V_optimal, optimal_policy = policy_iteration(env, gamma)
```

5. Finally, we display the optimal policy and its values:

```
>>> print('Optimal values\n', V_optimal)
Optimal values:
tensor([0.5404, 0.4966, 0.4681, 0.4541, 0.5569, 0.0000, 0.31
>>> print('Optimal policy\n', optimal_policy)
Optimal policy:
tensor([0., 3., 3., 3., 0., 3., 2., 3., 3., 1., 0., 3., 3.,
```

We got the same results as the value iteration algorithm.

We have just solved the FrozenLake environment with the policy iteration algorithm. You may wonder how to choose between the value iteration and policy iteration algorithms. Take a look at the following table:

Scenario	Preference	Reason
A large number of actions	Policy iteration	Policy iteration can converge faster
A small number of actions	Value iteration	Less computation in value iteration
A fair policy exists (obtained either by intuition or domain knowledge)	Policy iteration	Policy iteration from a fair policy can converge faster
Others	No preference	Policy iteration and value iteration are comparable

Table 14.1: Choosing between the policy iteration and value iteration algorithms

We solved a reinforcement learning problem using dynamic programming methods. They require a fully known transition matrix and reward matrix of an environment. And they have limited scalability for environments with many states. In the next section, we will continue our learning journey with the Monte Carlo method, which has no requirement of prior knowledge of the environment and is much more scalable.

Performing Monte Carlo learning

Monte Carlo (MC)-based reinforcement learning is a **model-free** approach, which means it doesn't need a known transition matrix and reward matrix. In this section, you will learn about MC policy evaluation on the Blackjack environment, and solve the environment with MC Control algorithms. Blackjack is a typical environment with an unknown transition matrix. Let's first simulate the Blackjack environment.

Simulating the Blackjack environment

Blackjack is a popular card game. The game has the following rules:

- The player competes against a dealer and wins if the total value of their cards is higher and doesn't exceed 21.
- Cards from 2 to 10 have values from 2 to 10.
- Cards J, K, and Q have a value of 10.
- The value of an ace can be either 1 or 11 (called a "usable" ace).
- At the beginning, both parties are given two random cards, but only one of the dealer's cards is revealed to the player. The player can request additional cards (called **hit**) or stop having any more cards (called **stick**). Before the player calls stick, the player will lose if the sum of their cards exceeds 21 (called **bust**). After the player sticks, the dealer keeps drawing cards until the sum of cards reaches 17. If the sum of the dealer's cards exceeds 21, the player will win. If neither of the two parties busts, the one with higher points will win or it may be a draw.

The Blackjack environment

(https://github.com/openai/gym/blob/master/gym/envs/toy_text/blackjack.py) in Gym is formulated as follows:

- An episode of the environment starts with two cards for each party, and only one from the dealer's cards is observed.
- An episode ends if there is a win or draw.
- The final reward of an episode is +1 if the player wins, -1 if the player loses, or 0 if there is a draw.
- In each round, the player can take any of the two actions, hit (1) and stick (0)

Now let's simulate the Blackjack environment and explore its states and actions:

1. First create a `Blackjack` instance:

```
>>> env = gym.make('Blackjack'v0')
```

2. Reset the environment:

```
>>> env.reset()  
(7, 10, False)
```

It returns the initial state (a 3-dimensional vector):

- Player's current points (7 in this example)
- The points of the dealer's revealed card (10 in this example)
- Having a usable ace or not (False in this example)

The usable ace variable is `True` only if the player has an ace that can be counted as 11 without causing a bust. If the player doesn't have an ace, or has an ace but it busts, this state variable will become `False`.

For another state example (18, 6, True), it means that the player has an ace counted as 11 and a 7, and that the dealer's revealed card is value 6.

3. Let's now take some actions to see how the environment works. First, we take a hit action since we only have 7 points:

```
>>> env.step(1)  
((13, 10, False), 0.0, False, {})
```

It returns a state (13, 10, False), a 0 reward, and the episode not being done (as in `False`).

4. Let's take another hit since we only have 13 points:

```
>>> env.step(1)  
((19, 10, False), 0.0, False, {})
```

5. We have 19 points and think it is good enough. Then we stop drawing cards by taking action stick (0):

```
>>> env.step(0)  
((19, 10, False), 1.0, True, {})
```

The dealer gets some cards and gets a bust. So the player wins and gets a +1 reward. The episode ends.

Feel free to play around with the Blackjack environment. Once you feel comfortable with the environment, you can move on to the next section, MC policy evaluation on a simple policy.

Performing Monte Carlo policy evaluation

In the previous section, we applied dynamic programming to perform policy evaluation, which is the value function of a policy. However, it won't work in most real-life situations where the transition matrix is not known beforehand. In this case, we can evaluate the value function using the MC method.

To estimate the value function, the MC method uses empirical mean return instead of expected return (as in dynamic programming). There are two approaches to compute the empirical mean return. One is first-visit, which averages returns **only** for the **first occurrence** of a state s among all episodes. Another one is every-visit, which averages returns for **every occurrence** of a state s among all episodes. Obviously, the first-visit approach has a lot less computation and is therefore more commonly used. And I will only cover the first-visit approach in this chapter.

In this section, we experiment with a simple policy, where we keep adding new cards until the total value reaches 18 (or 19, or 20 if you like). We perform first-visit MC evaluation on the simple policy as follows:

1. We first need to define a function that simulates a Blackjack episode under the simple policy:

```
>>> def run_episode(env, hold_score):
...     state = env.reset()
...     rewards = []
...     states = [state]
...     while True:
...         action = 1 if state[0] < hold_score else 0
...         state, reward, is_done, info = env.step(action)
...         states.append(state)
...         rewards.append(reward)
...         if is_done:
...             break
...     return states, rewards
```

In each round of an episode, the agent takes a hit if the current score is less than `hold_score` or a stick otherwise.

2. In the MC settings, we need to keep track of states and rewards over all steps. And in first-visit value evaluation, we average returns only for the first occurrence of a state among all episodes. We define a function that evaluates the simple Blackjack policy with first-visit MC:

```
>>> from collections import defaultdict
>>> def mc_prediction_first_visit(env, hold_score, gamma, n_episode):
...     V = defaultdict(float)
...     N = defaultdict(int)
...     for episode in range(n_episode):
...         states_t, rewards_t = run_episode(env, hold_score)
...         return_t = 0
...         G = {}
...         for state_t, reward_t in zip(
...             states_t[1:-1], rewards_t[1:-1]):
...             return_t = gamma * return_t + reward_t
...             G[state_t] = return_t
...         for state, return_t in G.items():
...             if state[0] <= 21:
...                 V[state] += return_t
...                 N[state] += 1
...         for state in V:
...             V[state] = V[state] / N[state]
...     return V
```

The function performs the following tasks:

- Running `n_episode` episodes under the simple Blackjack policy with function `run_episode`
- For each episode, computing the returns `G` for the first visit of each state
- For each state, obtaining the value by averaging its first returns from all episodes
- Returning the resulting values

Note that here we ignore states where the player busts, since we know their values are -1.

3. We specify the `hold_score` as `18`, the discount rate as `1` as a Blackjack episode is short enough, and will simulate 500,000 episodes:

```
>>> hold_score = 18
>>> gamma = 1
```

```
>>> n_episode = 500000
```

4. Now we plug in all variables to perform MC first-visit evaluation:

```
>>> value = mc_prediction_first_visit(env, hold_score, gamma, n_episode)
```

We then print the resulting values:

```
>>> print(value)
defaultdict(<class 'flat'>, {(20, 6, False): 0.692348565351,
.....  
.....  
(5, 9, False): -0.20612244897959184, (12, 7, True): 0.05882151515151515}
```

We have just computed the values for all possible 280 states:

```
>>> print('Number of stat's:', len(value))
Number of states: 280
```

We have just experienced computing the values of 280 states under a simple policy in the Blackjack environment using the MC method. The transition matrix of the Blackjack environment is not known beforehand. Moreover, obtaining the transition matrix (size 280 * 280) will be extremely costly if we go with the dynamic programming approach. In the MC-based solution, we just need to simulate a bunch of episodes and compute the empirical average values. In a similar manner, we will search for the optimal policy in the next section.

Performing on-policy Monte Carlo control

MC control is used to find the optimal policy for environments with unknown transition matrices. There are two types of MC control, on-policy and off-policy. In the **on-policy approach**, we execute the policy and evaluate and improve it iteratively; whereas in the off-policy approach, we train the optimal policy using data generated by another policy.

In this section, we focus on the on-policy approach. The way it works is very similar to the policy iteration method. It iterates between the following

two phases, evaluation and improvement, until convergence:

- In the evaluation phase, instead of evaluating the state value, we evaluate the **action-value**, which is commonly called the **Q-value**. Q-value $Q(s, a)$ is the value of a state-action pair (s, a) when taking the action a in state s under a given policy. The evaluation can be conducted in a first-visit or an every-visit manner.
- In the improvement phase, we update the policy by assigning the optimal action in each state:

$$\pi(s) = \operatorname{argmax}_a Q(s, a)$$

Let's now search for the optimal Blackjack policy with on-policy MC control by following the steps below:

1. We start with developing a function that executes an episode by taking the best actions under the given Q-values:

```
>>> def run_episode(env, Q, n_action):
    """
    Run a episode given Q-values
    @param env: OpenAI Gym environment
    @param Q: Q-values
    @param n_action: action space
    @return: resulting states, actions and rewards for
    """
    state = env.reset()
    rewards = []
    actions = []
    states = []
    action = torch.randint(0, n_action, [1]).item()
    while True:
        actions.append(action)
        states.append(state)
        state, reward, is_done, info = env.step(action)
        rewards.append(reward)
        if is_done:
            break
        action = torch.argmax(Q[state]).item()
    return states, actions, rewards
```

This serves as the improvement phase. Specifically, it does the following tasks:

- Initializing an episode
- Taking a random action as an exploring start
- After the first action, taking actions based on the given Q-value table, that is $a = \arg\max_a Q(s, a)$
- Storing the states, actions, and rewards for all steps in the episode, which will be used for evaluation

2. Next, we develop the on-policy MC control algorithm:

```
>>> def mc_control_on_policy(env, gamma, n_episode):  
...     """  
...     Obtain the optimal policy with on-policy MC control  
...     @param env: OpenAI Gym environment  
...     @param gamma: discount factor  
...     @param n_episode: number of episodes  
...     @return: the optimal Q-function, and the optimal po  
"""  
...     G_sum = defaultdict(float)  
...     N = defaultdict(int)  
...     Q = defaultdict(lambda: torch.empty(env.action_space))  
...     for episode in range(n_episode):  
...         states_t, actions_t, rewards_t =  
...             run_episode(env, Q, env.action_space)  
...         return_t = 0  
...         G = {}  
...         for state_t, action_t, reward_t in zip(  
...             states_t[::-1], actions_t[::-1],  
...             return_t = gamma * return_t + reward_t  
...             G[(state_t, action_t)] = return_t  
...         for state_action, return_t in G.items():  
...             state, action = state_action  
...             if state[0] <= 21:  
...                 G_sum[state_action] += return_t  
...                 N[state_action] += 1  
...                 Q[state][action] =  
...                     G_sum[state_action] / N[state_acti  
...     policy = {}  
...     for state, actions in Q.items():  
...         policy[state] = torch.argmax(actions).item()  
...     return Q, policy
```

This function does the following tasks:

- Randomly initializing the Q-values
- Running `n_episode` episodes
- For each episode, performing policy improvement and obtaining the training data; performing first-visit policy evaluation on the resulting states, actions, and rewards, and updating the Q-values
- In the end, finalizing the optimal Q-values and the optimal policy

3. Now that the MC control function is ready, we compute the optimal policy:

```
>>> gamma = 1
>>> n_episode = 500000
>>> optimal_Q, optimal_policy = mc_control_on_policy(env, g
```

Take a look at the optimal policy:

```
>>> print(optimal_policy)
{(16, 8, True): 1, (11, 2, False): 1, (15, 5, True): 1, (14
0, (12, 9, False): 0, (21, 2, True): 0, (16, 10, False): 1,
.....  
.....  
1, (18, 6, True): 0, (12, 2, True): 1, (8, 3, False): 1, (1
```

You may wonder if this optimal policy is really optimal and better than the previous simple policy (hold at 18 points). Let's simulate 100,000 Blackjack episodes under the optimal policy and the simple policy respectively:

1. We start with the function that simulates an episode under the simple policy:

```
>>> def simulate_hold_episode(env, hold_score):
...     state = env.reset()
...     while True:
...         action = 1 if state[0] < hold_score else 0
...         state, reward, is_done, _ = env.step(action)
...         if is_done:
...             return reward
```

2. Next, we work on the simulation function under the optimal policy:

```
>>> def simulate_episode(env, policy):
...     state = env.reset()
...     while True:
...         action = policy[state]
...         state, reward, is_done, _ = env.step(action)
...         if is_done:
...             return reward
```

3. We then run 100,000 episodes for both policies and keep track of their winning times:

```
>>> n_episode = 100000
>>> hold_score = 18
>>> n_win_opt = 0
>>> n_win_hold = 0
>>> for _ in range(n_episode):
...     reward = simulate_episode(env, optimal_policy)
...     if reward == 1:
...         n_win_opt += 1
...     reward = simulate_hold_episode(env, hold_score)
...     if reward == 1:
...         n_win_hold += 1
```

We print out the results as follows:

```
>>> print(f'Winning probability:\nUnder the simple policy:\nWinning probability:\nUnder the simple policy: 0.39955\nUnder the optimal policy: 0.42779')
```

Playing under the optimal policy has a 43% chance of winning, while playing under the simple policy has only 40% chance.

In this section, we solved the Blackjack environment with a model-free algorithm, MC learning. In MC learning, the Q-values are updated until the end of an episode. This could be problematic for long processes. In the next section, we will talk about Q-learning, which updates the Q-values for every step in an episode. You will see how it increases learning efficiency.

Solving the Taxi problem with the Q-learning algorithm

Q-learning is also a model-free learning algorithm. It updates the Q-function for every step in an episode. We will demonstrate how Q-learning is used to solve the Taxi environment. It is a typical environment with relatively long episodes. So let's first simulate the Taxi environment.

Simulating the Taxi environment

In the Taxi environment (<https://gym.openai.com/envs/Taxi-v3/>) the agent acts as a taxi driver to pick up the passenger from one location and drop off the passenger at the destination.

All subjects are on a 5 * 5 grid. Take a look at the following example:

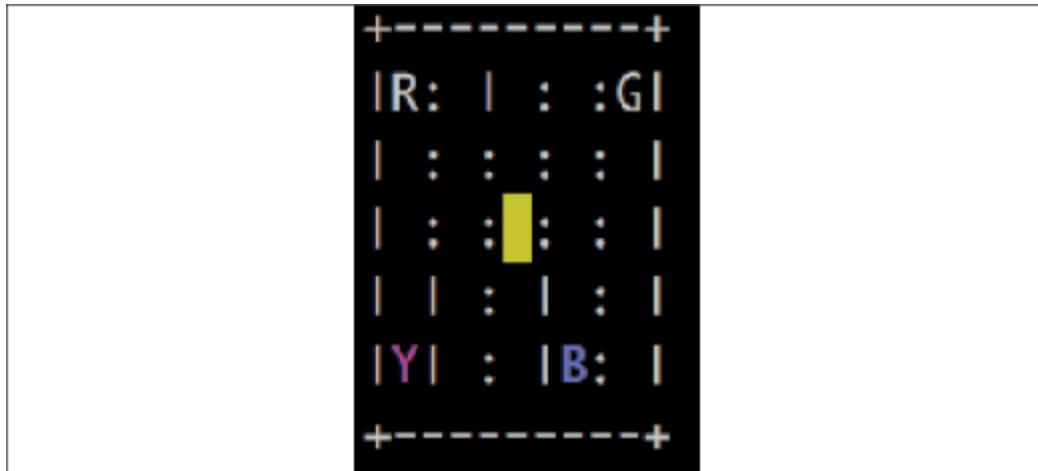


Figure 14.6: Example of the Taxi environment

Tiles in certain colors have the following meanings:

- **Yellow:** The location of the empty taxi (without the passenger)
- **Blue:** The passenger's location
- **Purple:** The passenger's destination

- **Green:** The location of the taxi with the passenger

The starting positions of the empty taxi and the passenger and the passenger's destination are randomly assigned in each episode.

The four letters R, Y, B, and G are the only four locations that allow passenger pick-up and drop-off. The one in purple is the destination, and the one in blue is the passenger's location.

The taxi can take any of the following six actions:

- 0: Moving south
- 1: Moving north
- 2: Moving east
- 3: Moving west
- 4: Picking up the passenger
- 5: Dropping off the passenger

There is a pillar "|" between two tiles, which prevents the taxi from moving between two tiles.

In each step, the reward follows these rules:

- +20 for driving the passenger to the destination. An episode will end in this situation. And another situation in which an episode will end is when there are 200 steps.
- -10 for trying to pick up or drop off illegally (not on any of R, Y, B, or G).
- -1 otherwise.

Last but not least, there are actually 500 possible states: obviously the taxi can be on any of the 25 tiles, the passenger can be on any of R, Y, B, G or inside the taxi, and the destination can be any of R, Y, B, G; hence, we have $25 * 5 * 4 = 500$ possible states.

Now let's play around with the environment as follows:

1. First we create an instance of the Taxi environment:

```
>>> env = gym.make('Taxi-v3')
>>> n_state = env.observation_space.n
>>> print(n_state)
500
>>> n_action = env.action_space.n
>>> print(n_action)
6
```

We also know that the state is represented by an integer ranging from 0 to 499, and there are 6 possible actions.

2. We reset the environment and render it:

```
>>> env.reset()
262
>>> env.render()
```

You will see a 5 * 5 grid similar to the following one:

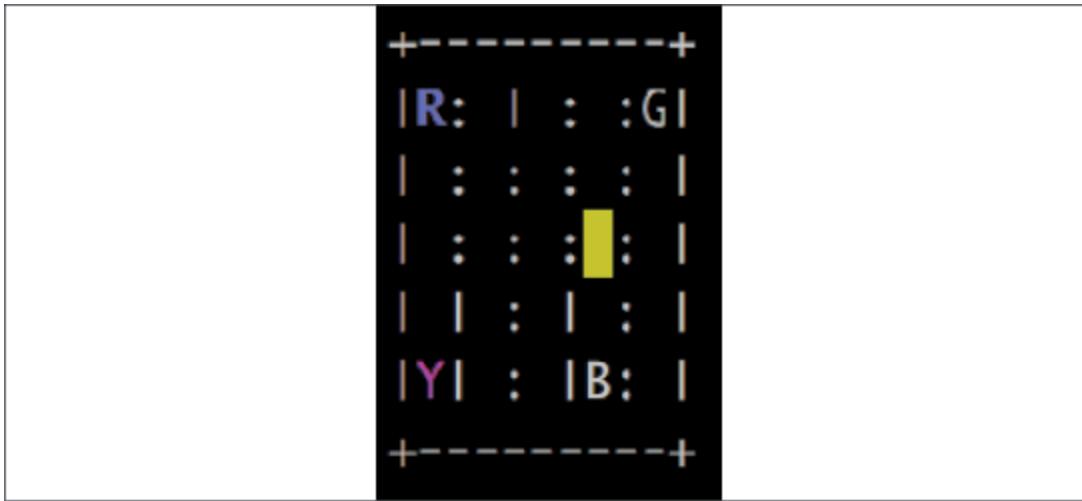


Figure 14.7: Example starting step of the Taxi environment

The passenger is on the blue R tile, and the destination is on the purple Y.

3. Now let's go pick up the passenger by heading west for three tiles and north for two tiles (you will need to take different actions according to your initial state) then take the "pick-up" action:

```
>>> print(env.step(3))
(242, -1, False, {'prob': 1.0})
```

```
>>> print(env.step(3))
(222, -1, False, {'prob': 1.0})
>>> print(env.step(3))
(202, -1, False, {'prob': 1.0})
>>> print(env.step(1))
(102, -1, False, {'prob': 1.0})
>>> print(env.step(1))
(2, -1, False, {'prob': 1.0})
>>> print(env.step(4))
(18, -1, False, {'prob': 1.0})
```

Render the environment:

```
>>> env.render()
```



Figure 14.8: Example of a state where the passenger is inside the taxi
The taxi turns green, meaning the passenger is inside the taxi.

4. Now let's head to the destination by taking the "down" action four times (again, you will need to take your own set of actions) then executing a "drop-off":

```
>>> print(env.step(0))
(118, -1, False, {'prob': 1.0})
>>> print(env.step(0))
(218, -1, False, {'prob': 1.0})
>>> print(env.step(0))
(318, -1, False, {'prob': 1.0})
>>> print(env.step(0))
(418, -1, False, {'prob': 1.0})
```

```
>>> print(env.step(5))
(410, 20, True, {'prob': 1.0})
```

A +20 reward is granted in the end for a successful drop-off.

5. We render the environment finally:

```
>>> env.render()
```

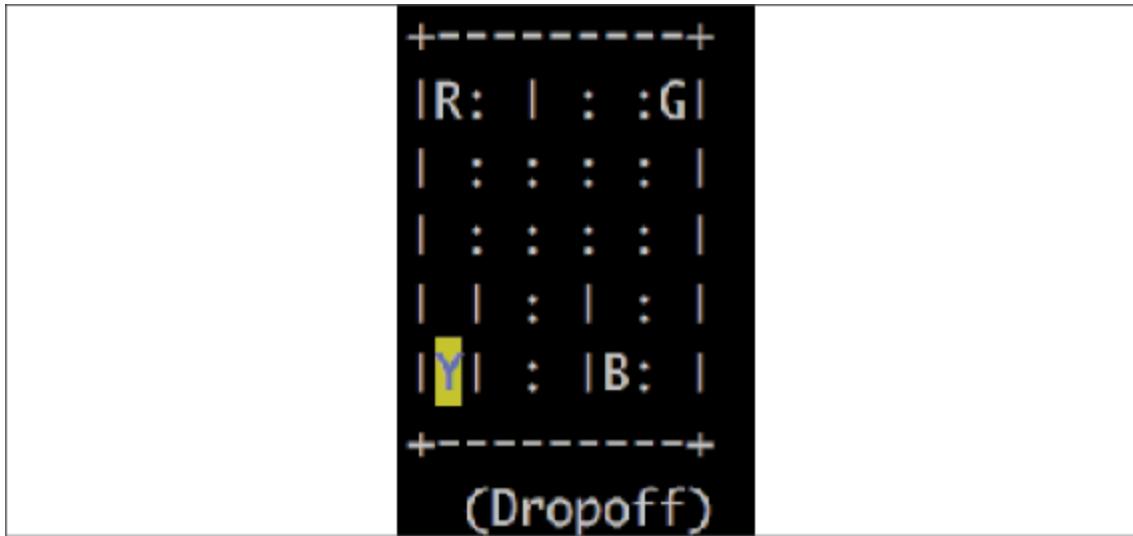


Figure 14.9: Example of a state where the passenger arrives at the destination

You can take some random actions and see how difficult it is for a model to solve the environment. We will discuss the Q-learning algorithm in the next section.

Developing the Q-learning algorithm

Q-learning is an off-policy learning algorithm that optimizes the Q-values based on data generated by a behavior policy. The behavior policy is a greedy policy where it takes actions that achieve the highest returns for given states. The behavior policy generates learning data and the target policy (the policy we attempt to optimize) updates the Q-values based on the following equation:

$$Q(s, a) := Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

Here, s' is the resulting state after taking action a from state s and r is the associated reward. $\max_{a'} Q(s', a')$ means that the behavior policy generates the highest Q-value given state s' . Finally, hyperparameters α and γ are the learning rate and discount factor respectively.

Learning from experience generated by another policy enables Q-learning to optimize its Q-values in every single step in an episode. We gain the information from a greedy policy and use this information to update the target values right away.

One more thing to note is that the target policy is epsilon-greedy, meaning it takes a random action with a probability of ε (value from 0 to 1) and takes a greedy action with a probability of $1 - \varepsilon$. The epsilon-greedy policy combines **exploitation** and **exploration**: it exploits the best action while exploring different actions.

Now it is time to develop the Q-learning algorithm to solve the Taxi environment:

1. We start with defining the epsilon-greedy policy:

```
>>> def gen_epsilon_greedy_policy(n_action, epsilon):
...     def policy_function(state, Q):
...         probs = torch.ones(n_action) * epsilon / n_acti...
...         best_action = torch.argmax(Q[state]).item()
...         probs[best_action] += 1.0 - epsilon
...         action = torch.multinomial(probs, 1).item()
...         return action
...     return policy_function
```

Given $|A|$ possible actions, each action is taken with a probability ε/A , and the action with the highest state-action value is chosen with an additional probability $1 - \varepsilon$.

2. Now we create an instance of the epsilon-greedy-policy:

```
>>> epsilon = 0.1
>>> epsilon_greedy_policy = gen_epsilon_greedy_policy(env.a...
```

Here, $\varepsilon = 0.1$, which is the exploration ratio.

3. Next, we develop the Q-learning algorithm:

```
>>> def q_learning(env, gamma, n_episode, alpha):
...     """
...     Obtain the optimal policy with off-policy Q-learning
...     @param env: OpenAI Gym environment
...     @param gamma: discount factor
...     @param n_episode: number of episodes
...     @return: the optimal Q-function, and the optimal po
...
...     n_action = env.action_space.n
...     Q = defaultdict(lambda: torch.zeros(n_action))
...     for episode in range(n_episode):
...         state = env.reset()
...         is_done = False
...         while not is_done:
...             action = epsilon_greedy_policy(state, Q)
...             next_state, reward, is_done, info =
...                 env.step(action)
...             delta = reward + gamma * torch.max(Q[next_s
...                 Q[state][action]
...             Q[state][action] += alpha * delta
...             length_episode[episode] += 1
...             total_reward_episode[episode] += reward
...             if is_done:
...                 break
...             state = next_state
...     policy = {}
...     for state, actions in Q.items():
...         policy[state] = torch.argmax(actions).item()
...     return Q, policy
```

We first initialize the Q-table. Then in each episode, we let the agent take actions following the epsilon-greedy policy, and update the Q function for each step based on the off-policy learning equation. We run `n_episode` episodes and finally obtain the optimal policy and Q-values.

4. We then initiate two variables to store the performance of each of 1,000 episodes, the episode length (number of steps in an episode), and total reward:

```
>>> n_episode = 1000
>>> length_episode = [0] * n_episode
>>> total_reward_episode = [0] * n_episode
```

5. Finally, we perform Q-learning to obtain the optimal policy for the Taxi problem:

```
>>> gamma = 1
>>> alpha = 0.4
>>> optimal_Q, optimal_policy = q_learning(env, gamma, n_episodes)
```

Here, discount rate $\gamma = 1$, and learning rate $\alpha = 0.4$.

6. After 1,000 episodes of learning, we plot the total rewards over episodes as follows:

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(total_reward_episode)
>>> plt.title('Episode reward over time')
>>> plt.xlabel('Episode')
>>> plt.ylabel('Total reward')
>>> plt.ylim([-200, 20])
>>> plt.show()
```

Refer to the following screenshot for the end result:

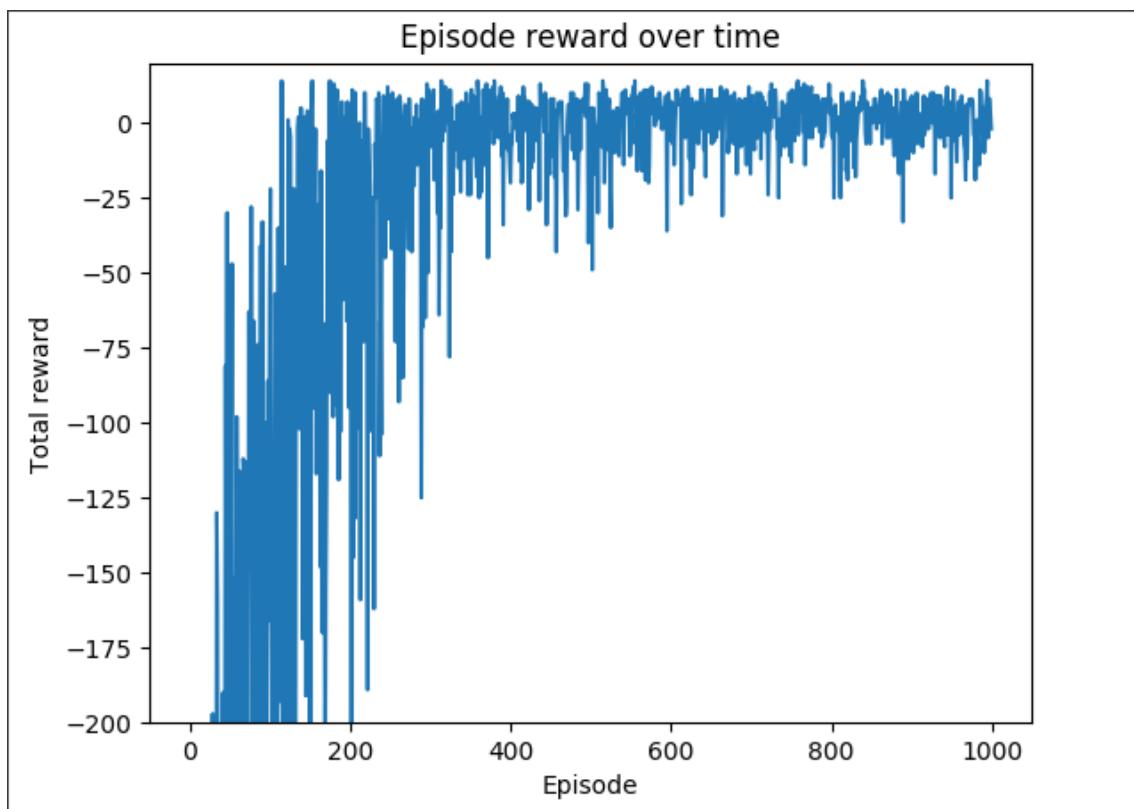


Figure 14.10: Total rewards over episodes

The total rewards keep improving during learning. And they stay around +5 after 600 episodes.

7. We also plot the lengths over episodes as follows:

```
>>> plt.plot(length_episode)
>>> plt.title('Episode length over time')
>>> plt.xlabel('Episode')
>>> plt.ylabel('Length')
>>> plt.show()
```

Refer to the following screenshot for the end result:

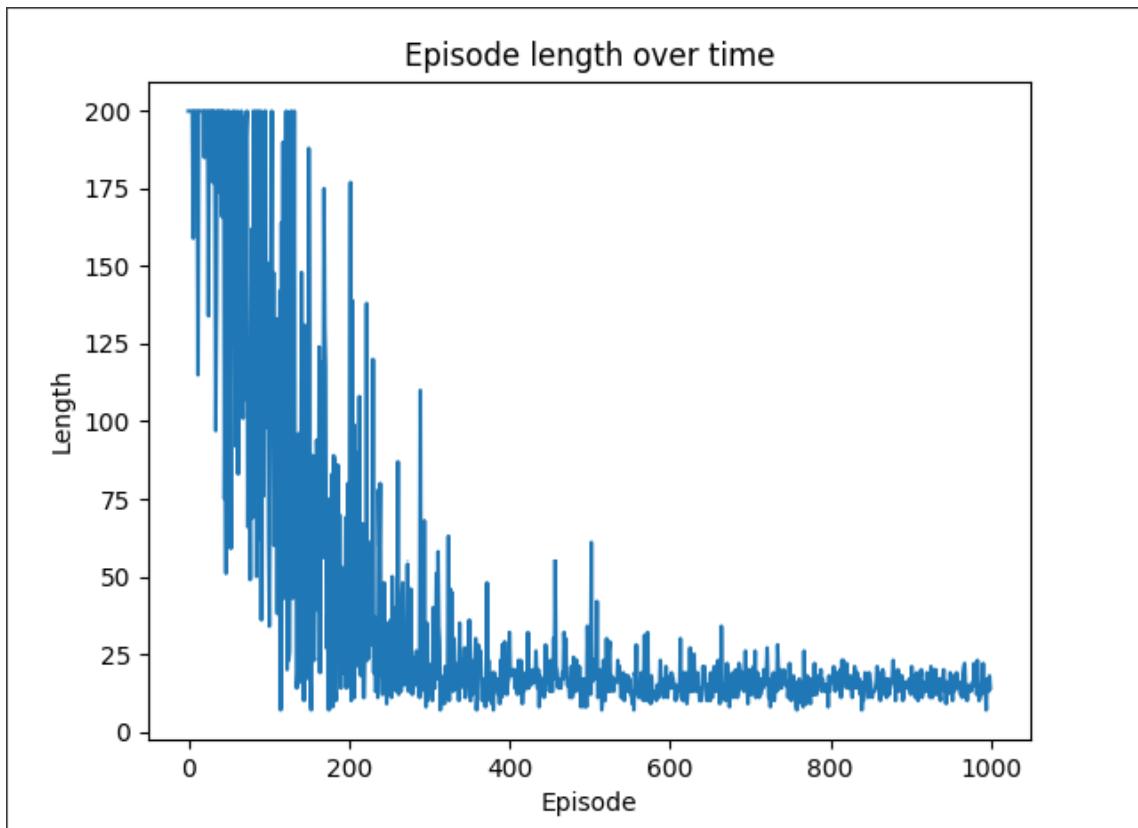


Figure 14.11: Episode lengths over episodes

As you can see, the episode lengths decrease from the maximum 200 to around 10, and the model converges around 600 episodes. It means after training, the model is able to solve the problem in around 10 steps.

In this section, we solved the Taxi problem with off-policy Q-learning. The algorithm optimizes the Q-values in every single step by learning from the experience generated by a greedy policy.

Summary

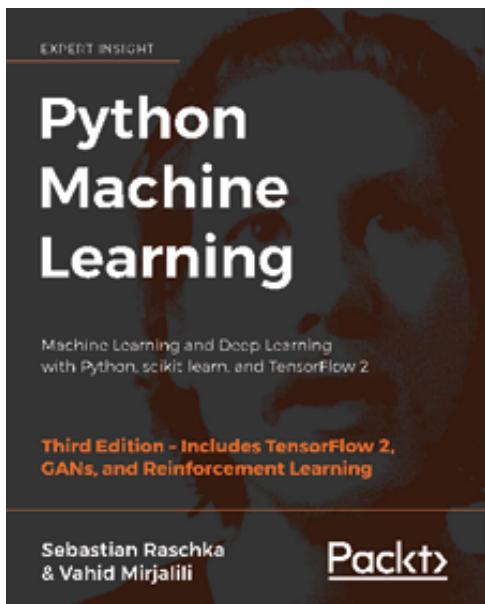
We started the chapter by setting up the working environment. After that, we studied the fundamentals of reinforcement learning along with a few examples. After exploring the FrozenLake environment, we solved it with two dynamic programming algorithms, value iteration and policy iteration. We talked about Monte Carlo learning and used it for value approximation and control in the Blackjack environment. Lastly, we developed the Q-learning algorithm and solved the Taxi problem.

Exercises

1. Can you try to solve the $8 * 8$ FrozenLake environment with the value iteration or policy iteration algorithm?
2. Can you implement the every-visit MC policy evaluation algorithm?
3. Can you use a different exploration ratio ε in the Q-learning algorithm and see how things change?

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



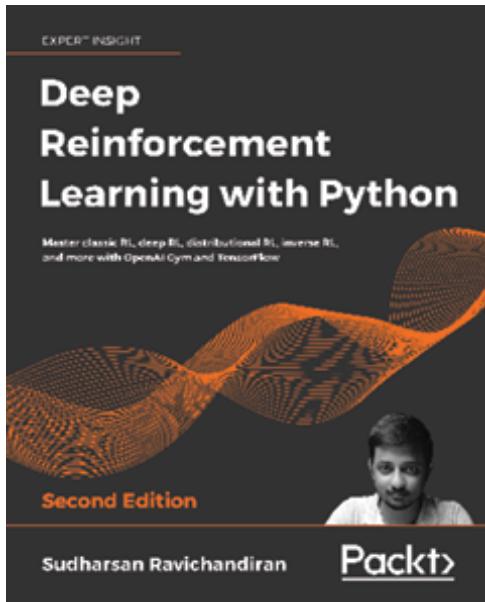
Python Machine Learning - Third Edition

Sebastian Raschka, Vahid Mirjalili

ISBN: 978-1-78995-575-0

- Master the frameworks, models, and techniques that enable machines to 'learn' from data
- Use scikit-learn for machine learning and TensorFlow for deep learning
- Apply machine learning to image classification, sentiment analysis, intelligent web applications, and more
- Build and train neural networks, GANs, and other models
- Discover best practices for evaluating and tuning models

- Predict continuous target outcomes using regression analysis
- Dig deeper into textual and social media data using sentiment analysis



Deep Reinforcement Learning with Python – Second Edition

Sudharsan Ravichandiran

ISBN: 978-1-83921-068-6

- Understand core RL concepts including the methodologies, math, and code
- Train an agent to solve Blackjack, FrozenLake, and many other problems using OpenAI Gym
- Train an agent to play Ms Pac-Man using a Deep Q Network
- Learn policy-based, value-based, and actor-critic methods
- Master the math behind DDPG, TD3, TRPO, PPO, and many others
- Explore new avenues such as the distributional RL, meta RL, and inverse RL
- Use Stable Baselines to train an agent to walk and play Atari games

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

Symbols

A

actions [453](#), [454](#)

action-value [473](#)

activation function [256](#)

activation functions

linear [268](#)

ReLU [268](#)

sigmoid [268](#)

softmax [268](#)

tanh [268](#)

ad click-through

predicting, with logistic regression [165](#), [166](#)

ad click-through prediction [110](#)

with decision tree [134](#), [136](#), [137](#), [138](#), [139](#), [140](#)

adjusted R² [245](#)

agent [453](#), [454](#)

AI-based assistance [4](#)

AI plus human intelligence [4](#)

AlphaGo [3](#)

Anaconda [38](#)

reference link [37](#)

Apache Hadoop

URL [355](#)

Arcene Dataset [97](#)

area under the curve (AUC) [68](#)

Artificial General Intelligence (AGI) [452](#)

Artificial Intelligence (AI) [8](#)

artificial masterpieces, Google Arts & Culture

reference link [261](#)

artificial neural networks (ANNs) [11](#), [254](#)

association [315](#)

attributes [315](#)

automation

versus machine learning [5](#)

averaging [32](#)

B

backpropagation [258](#), [259](#)

Backpropagation Through Time (BPTT) [420](#)

bagging [32](#), [140](#)

bag of words (BoW) [362](#)

Bag of Words (BoW) model [301](#)

basic linear algebra

reference link [8](#)

Bayes [48](#)

Bayes' theorem

example [49](#), [50](#), [51](#)

Bellman optimality equation

reference link [460](#)

bias [14](#), [154](#), [227](#)

bias-variance trade-off [17](#), [18](#)

Bidirectional Encoder Representations from Transformers (BERT) [448](#)

bigrams [289](#)

binarization [360](#)

binary classification [45](#), [268](#)

binning [31](#)

Blackjack environment

reference link [469](#)

simulating [468](#), [470](#)

boosting [34](#), [36](#), [142](#)

bootstrap aggregating [140](#)

bootstrapping [32](#)

Box-Cox transformation [31](#)

C

C4.5 [116](#)

categorical features [111](#), [112](#)

converting, to numerical features [148](#), [150](#), [151](#)

categorical variables

combining [207](#), [209](#), [210](#)

categories [44](#)

chain rule [259](#)

Chebyshev distance [316](#)

Chi-squared Automatic Interaction Detector (CHAID) [116](#)

classes [44](#)

classification [10](#), [44](#)

binary classification [45](#)

multiclass classification [46](#), [47](#)

multi-label classification [47](#), [48](#)

Classification and Regression Tree (CART) [116](#)

classification performance

evaluating [65](#), [66](#), [67](#), [68](#), [70](#)

click-through rate (CTR) [110](#)

clothing Fashion-MNIST

reference link [388](#)

clothing image classifier

improving, with data augmentation [406](#), [407](#), [408](#), [409](#)

clothing image dataset [388](#), [389](#), [391](#)

clothing images, classifying with CNNs [392](#)

CNN model, architecting [392](#), [393](#), [394](#)

CNN model, fitting [395](#), [396](#), [397](#), [398](#)

convolutional filters, visualizing [398](#), [399](#), [400](#)

clustering [315](#)

CNN [382](#)

architecting, for classification [387](#), [388](#)

convolutional layer [382](#), [383](#), [384](#)

nonlinear layer [384](#)

pooling layer [385](#), [386](#)

CNN classifier

boosting, with data augmentation [400](#)

coefficients [153](#), [227](#)

color restoration [261](#)

computation graphs [40](#)

computer vision [260](#)

conda [37](#)

confusion matrix [66](#)

Continuous Bag of Words (CBOW) [363](#)

convex function [154](#)

reference link [155](#)

convolutional layer [382](#), [383](#), [384](#)

Corpora [287](#), [288](#), [289](#)

cost function [9](#), [155](#), [157](#), [158](#)

Cross-Industry Standard Process for Data Mining (CRISP-DM) [25](#)

business understanding [26](#)

data preparation [26](#)

data understanding [26](#)

deployment phase [26](#)

evaluation phase [26](#)

modeling phase [26](#)

URL [25](#)

cross-validation

used, for avoiding overfitting [19](#), [20](#), [21](#)

used, for tuning models [70](#), [72](#), [73](#)

cumulative rewards [455](#)

D

data

acquiring [222](#), [223](#), [224](#), [225](#), [226](#)

classifying, with logistic regression [151](#)

data augmentation

clothing image classifier, improving [406](#), [407](#), [408](#), [409](#)

CNN classifier, boosting [400](#)

DataFrames [185](#)

data preparation stage

best practices [349](#), [350](#), [351](#), [352](#), [353](#), [354](#), [355](#)

data preprocessing [355](#)

data technology (DT) [6](#)

decision hyperplane [78](#)

decision tree

ad click-through prediction [134](#), [136](#), [137](#), [138](#), [139](#), [140](#)

constructing [115](#), [116](#)

ensembling [140](#), [142](#), [143](#), [144](#), [145](#)

exploring [112](#), [113](#), [114](#)
implementing [124](#), [125](#), [127](#), [128](#), [129](#), [131](#), [132](#)
implementing, with scikit-learn [133](#), [134](#)

decision tree module

reference link [133](#)

decision tree regression

estimating with [234](#)
implementing [237](#), [238](#), [240](#), [241](#)

decoder [446](#)

deep learning [11](#)

deep learning (DL) [254](#)

deep neural networks [30](#)

deployment and monitoring stage

best practices [374](#), [375](#), [376](#), [377](#), [378](#)

dimensionality reduction [25](#), [307](#), [308](#)

used, for avoiding overfitting [24](#)

discretization [361](#)

distributed computing [294](#)

document frequency [334](#)

Dorothea Dataset [97](#)

dot product [382](#)

Dow Jones Industrial Average (DJIA) [217](#)

downsampling layer [385](#)

dropout [269](#), [270](#)

dynamic programming

FrozenLake environment, solving [457](#)

E

early stopping [24](#), [270](#)

edges [255](#)

Elbow method [331](#)

encoder [446](#)

entropy [120](#), [121](#), [122](#)

environment [453](#)

episode [457](#)

epsilon-greedy policy [482](#)

Euclidean distance [316](#)

evidence [52](#)

exploitation [482](#)

exploration [482](#)

exploration phase [26](#)

F

f1 score [66](#)

face image dataset

exploring [98](#), [99](#)

face images

classifying, with SVMs [98](#)

feature [24](#)

feature-based bagging [141](#)

feature crossing. See also **feature interaction**

feature engineering [30](#), [204](#), [218](#), [219](#), [220](#), [221](#), [355](#)

on categorical variables, with Spark [203](#)

feature hashing. See also **hashing trick**

feature interaction [207](#), [209](#), [210](#)

feature map [382](#)

feature projection [25](#)

features [44](#), [315](#)

generating [222](#), [223](#), [224](#), [225](#), [226](#)

feature selection [170](#)

L1 regularization, examining for [170](#), [171](#)

used, for avoiding overfitting [24](#)

with random forest [180](#), [181](#)

feedforward neural network [256](#)

fetal state classification

on cardiotocography [104](#), [105](#), [106](#)

forget gate [422](#)

FrozenLake

solving, with policy iteration algorithm [464](#), [465](#), [466](#), [467](#), [468](#)

solving, with value iteration algorithm [460](#), [461](#), [462](#), [463](#), [464](#)

FrozenLake environment

simulating [457](#), [458](#), [459](#), [460](#)

solving, with dynamic programming [457](#)

fundamental analysis [214](#)

G

Gated Recurrent Unit (GRU) [420](#)

Gaussian kernel [93](#)

generalization [13](#), [14](#)

Generative Pre-training Transformer (GPT) [448](#)

genetic algorithms (GA) [11](#)

Gensim [285](#), [294](#)

URL [286](#)

Georgetown-IBM experiment

reference link [283](#)

Gini Impurity [117](#), [118](#), [119](#), [120](#)

Google Cloud Storage

reference link [355](#)

Google Neural Machine Translation (GNMT) [261](#)

gradient boosted trees (GBT) [142](#), [144](#), [145](#)

gradient boosting machines [142](#)

gradient descent [158](#)

ad click-through, predicting with logistic regression [165](#), [166](#)

logistic regression model, training [158](#), [159](#), [160](#), [161](#), [163](#), [164](#)

gradients [41](#)

Graphical Processing Units (GPUs) [11](#)

Graphviz

URL [133](#)

GraphX [185](#)

H

Hadoop Distributed File System (HDFS) [192](#)

handwritten digit recognition [46](#)

handwritten digits MNIST dataset

reference link [388](#)

harmonic mean [66](#)

hashing categorical

features [204](#), [206](#), [207](#)

hashing collision [205](#)

hashing trick [204](#)

Heterogeneity Activity Recognition Dataset [97](#)

HIGGS Dataset [97](#)

high-order polynomial function [22](#)

high variance [15](#)

holdout method [21](#)

horizontal flipping

for data augmentation [400](#), [401](#), [402](#), [403](#)

hyperplane [76](#)

I

image-based search engines [261](#)

image classification performance

boosting, with PCA [103](#), [104](#)

ImageDataGenerator module

reference link [400](#)

image recognition [261](#)

IMDb

URL [423](#)

imputing [27](#)

Information Gain [120](#), [121](#), [122](#)

inner cross-validation [21](#)

input gate [422](#)

interaction [30](#)

intercept [154](#)

Internet of Things (IoT) [6](#)

interquartile range [29](#)

Iterative Dichotomiser 3 (ID3) [116](#)

K

k

value, selecting [331](#), [332](#), [333](#)

Kaggle

URL [8](#)

k equal-sized folds [20](#)

Keras

URL [266](#)

kernel coefficient [93](#)

kernel function [93](#)

kernels

linearly non-separable problems, solving [91](#), [92](#), [93](#), [94](#), [96](#)

k-fold cross-validation [20](#)

k-means

implementing [317](#), [318](#), [319](#), [320](#), [321](#), [322](#), [323](#), [324](#), [325](#), [326](#), [327](#),
[328](#), [329](#)

implementing, with scikit-learn [329](#), [330](#), [331](#)

used, for clustering newsgroups data [316](#), [333](#), [334](#), [335](#), [336](#), [337](#)

k-means clustering

working [316](#), [317](#)

k-nearest neighbors (KNN) [359](#)

L

L1 regularization [169](#)

examining, for feature selection [170](#), [171](#)

L2 regularization [169](#)

labeled data [315](#)

Labeled Faces in the Wild (LFW) people dataset

reference link [98](#)

label encoding [28](#)

labels [44](#)

Laplace smoothing [54](#)

Lasso [169](#)

latent Dirichlet allocation (LDA)

using, for topic modeling [342](#), [343](#), [344](#), [345](#)

layer [255](#)

layers

adding, to neural network [260](#)

leaf [112](#)

Leaky ReLU [268](#)

learning_curve module

reference link [373](#)

learning rate [158](#)

Leave-One-Out-Cross-Validation (LOOCV) [20](#)

lemmatization [293](#), [305](#)

liblinear

reference link [80](#)

libsvm

reference link [80](#)

likelihood [52](#)

linear function [268](#)

linear kernel [96](#)

linearly non-separable problems

solving, with kernels [91](#), [92](#), [93](#), [94](#), [96](#)

linear regression

estimating with [226](#)

example [216](#)

implementing [228](#), [229](#), [230](#), [231](#), [232](#)

implementing, with scikit-learn [232](#)

implementing, with TensorFlow [233](#), [234](#)

working [227](#), [228](#)

LinearSVC

reference link [102](#)

logarithmic loss [158](#)

logic gate

reference link [421](#)

logistic function [152](#), [153](#), [256](#)

logistic regression [153](#), [154](#), [368](#)

ad click-through, predicting [165](#), [166](#)
data, classifying [151](#)
implementing, with TensorFlow [178](#), [180](#)

logistic regression model

testing [201](#), [203](#)
training [158](#), [201](#), [203](#)
training, with gradient descent [158](#), [159](#), [160](#), [161](#), [163](#), [164](#)
training, with regularization [169](#), [170](#)
training, with stochastic gradient descent [166](#), [168](#), [169](#)

log loss [158](#)

London FTSE-100

reference link [218](#)

Long Short-Term Memory

long-term dependencies, overcoming [420](#), [421](#)

Long Short-Term Memory (LSTM) [420](#)

loss function [9](#)

low bias [14](#)

LSTM recurrent cell

forget gate [422](#)
input gate [422](#)
memory unit [422](#)
output gate [422](#)

M

machine [363](#)

machine learning [2](#)

applications [6](#), [7](#)

core [13](#)

need for [2](#), [3](#), [4](#)

prerequisites [7](#)

reinforcement learning [9](#)

supervised learning [9](#)

types [8](#)

unsupervised learning [9](#)

versus automation [5](#)

versus traditional programming [5](#)

machine learning algorithms

development history [11](#), [12](#)

machine learning library (MLlib) [185](#)

machine learning regression

problems [216](#)

machine learning solution

workflow [348](#), [349](#)

machine learning tasks [10](#), [11](#)

machine vision [261](#)

Manhattan distance [316](#)

many-to-many (synced) RNNs [416](#), [417](#)

many-to-many (unsynced) RNNs [417](#), [418](#)

many-to-one RNNs [415](#), [416](#)

margin [78](#)

massive click logs

data, caching [196](#)

data, splitting [195](#), [196](#)

learning, with Spark [192](#)

loading [192](#), [193](#), [194](#), [195](#)

Massive Open Online Courses (MOOCs) [8](#)

Matplotlib [40](#)

matplotlib package

reference link [299](#)

maximum-margin [79](#)

mean absolute error (MAE) [245](#)

mean squared error (MSE) [18](#), [154](#), [227](#), [258](#)

memory unit [422](#)

Miniconda [37](#)

reference link [37](#)

missing data imputation [351](#)

missing values

dealing with [27](#)

MNIST (Modified National Institute of Standards and Technology) [46](#)

model-free approach [468](#)

models

combining [31](#)

tuning, with cross-validation [70](#), [72](#), [73](#)

model training, evaluation, and selection stage

best practices [367](#), [369](#), [370](#), [371](#), [372](#), [373](#), [374](#)

Monte Carlo learning

performing [468](#)

Monte Carlo policy evaluation

performing [470](#), [472](#), [473](#)

Moore's law [12](#)

MovieLens

URL [60](#)

movie rating dataset

reference link [60](#)

movie recommender

building, with Naïve Bayes [60](#), [62](#), [63](#), [64](#), [65](#)

movie review sentiment, analyzing with RNNs [423](#)

data analysis [423](#), [424](#), [425](#), [426](#)

data preprocessing [423](#), [424](#), [425](#), [426](#)

multiple LSTM layers, stacking [429](#), [430](#), [431](#)

simple LSTM network, building [426](#), [428](#)

multiclass classification [46](#), [47](#), [268](#)

handling [175](#), [176](#), [177](#)

multi-head attention [447](#)

multi-label classification [47](#), [48](#)

multi-layer perceptron (MLP) [265](#)

multinomial classification [46](#)

multinomial logistic regression [175](#)

multiple classes

dealing with [85](#), [87](#), [88](#), [89](#), [90](#), [91](#)

N

Naïve [48](#)

Naïve Bayes [48](#), [368](#)

implementing [55](#), [56](#), [58](#), [59](#)

implementing, with sci-kit learn [59](#)

mechanics [52](#), [53](#), [54](#), [55](#)

movie recommender, building [60](#), [62](#), [63](#), [64](#), [65](#)

named entities [285](#)

named entity recognition (NER) [285](#)

NASDAQ Composite

reference link [218](#)

natural language [282](#)

natural language processing (NLP) [261](#), [282](#), [283](#)

applications [284](#), [285](#)

history [283](#)

Natural Language Toolkit (NLTK) [285](#)

negative hyperplane [78](#)

NER [292](#)

nested cross-validation [21](#)

neural machine translation system, Facebook

reference link [283](#)

neural networks [370](#)

building [262](#)

demystifying [254](#)

fine-tuning [273](#), [274](#), [275](#), [276](#), [277](#), [278](#), [279](#)

hidden layer [254](#), [255](#)

implementing [262](#), [263](#), [264](#), [265](#)

implementing, with scikit-learn [265](#)

implementing, with TensorFlow [266](#), [267](#)

input layer [254](#), [255](#)
layers [254](#), [255](#)
layers, adding [260](#)
output layer [254](#), [255](#)
overfitting, preventing [269](#)
stock prices, predicting [271](#)
training [271](#), [273](#)

newsgroups

underlying topics, discovering [337](#)

newsgroups data

clustering, with k-means [316](#), [333](#), [334](#), [335](#), [336](#), [337](#)
exploring [298](#), [299](#), [300](#)
obtaining [294](#), [295](#), [296](#), [297](#)
visualizing, with t-SNE [307](#)

n-grams [289](#)

NLP libraries

installing [285](#), [286](#), [287](#)

nltk

URL [286](#)

NLTK [40](#)

node [112](#)

nodes [255](#)

no free lunch theorem

reference link [8](#)

non-convex function [154](#)

reference link [155](#)

non-exhaustive scheme [20](#)

nonlinear layer [384](#)

non-negative matrix factorization (NMF) [308](#)

used, for topic modeling [338](#), [339](#), [340](#), [341](#)

numerical features [111](#), [112](#)

categorical features, converting to [148](#), [150](#), [151](#)

NumPy [39](#)

URL [38](#)

O

observations [44](#), [315](#)

one-hot encoding [28](#), [148](#)

one-hot encoding categorical features [196](#), [198](#), [199](#), [200](#)

one-to-many RNNs [416](#)

online learning

large datasets, training [172](#), [174](#), [175](#)

on-policy approach [473](#)

on-policy Monte Carlo control

performing [473](#), [474](#), [475](#), [476](#), [477](#)

ontology [284](#)

OpenAI

URL [452](#)

OpenAI Gym

installing [452](#), [453](#)

URL [452](#)

optimal hyperplane

determining [78](#), [79](#), [80](#), [81](#)

ordinal encoding [148](#), [150](#)

ordinal feature [111](#)

outer cross-validation [21](#)

outliers

handling [82](#), [83](#)

output gate [422](#)

overfitting [14](#), [15](#)

avoiding, with cross-validation [19](#), [20](#), [21](#)

avoiding, with dimensionality reduction [24](#)

avoiding, with feature selection [24](#)

avoiding, with regularization [22](#), [24](#)
preventing, in neural networks [269](#)

P

pandas library [39](#)

part-of-speech (PoS) tagging [291](#), [412](#)

pickle

models, restoring [374](#), [375](#)

models, saving [374](#), [375](#)

plot_learning_curve function

reference link [373](#)

policy [456](#)

policy evaluation step [456](#)

policy iteration algorithm

FrozenLake, solving [464](#), [465](#), [466](#), [467](#), [468](#)

polynomial transformation [30](#), [361](#)

pooling layer [385](#), [386](#)

positive hyperplane [78](#)

posterior [52](#)

Power transforms [30](#)

precision [66](#)

predictive variables [44](#), [315](#)

preprocessing phase [26](#), [27](#)

principal component analysis (PCA) [308](#), [358](#)

image classification performance, boosting [103](#), [104](#)

reference link [103](#)

prior [52](#)

probability [101](#)

reference link [8](#)

Project Gutenberg

URL [432](#)

projection [315](#)

PySpark [40](#)

programming [189](#), [190](#), [191](#), [192](#)

Python [36](#)

setting up [37](#)

Python Imaging Library (PIL) [99](#)

Python packages

installing [38](#)

PyTorch [40](#)

installing [450](#), [451](#)

references [451](#)

URL [266](#), [450](#)

Q

Q-learning algorithm

developing [482](#), [483](#), [484](#), [485](#), [486](#)

Taxi problem, solving [477](#)

qualitative features [111](#)

quantitative features [112](#)

Q-value [473](#)

R

R² [245](#)

radial basis function (RBF) kernel [93](#)

random access memory (RAM) [185](#)

random forest [141](#), [370](#)

using, for feature selection [180](#), [181](#)

RBF kernel [96](#)

recall [66](#)

receiver operating characteristic (ROC) [68](#)

receptive fields [384](#)

Rectified Linear Unit (ReLU) [256](#), [268](#)

recurrent mechanism [413](#), [414](#)

recurrent neural networks (RNNs) [412](#)

many-to-many (synced) RNNs [416](#), [417](#)

many-to-many (unsynced) RNNs [417](#), [418](#)

many-to-one RNNs [415](#), [416](#)

one-to-many RNNs [416](#)

regression [10](#), [215](#)

regression algorithms

stock prices, predicting [246](#), [247](#), [248](#), [249](#), [250](#)

regression forest

implementing [242](#)

regression performance

estimating [244](#), [245](#), [246](#)

regression trees [234](#), [235](#), [236](#), [237](#)

regularization

used, for avoiding overfitting [22](#), [24](#)

used, for training logistic regression model [169](#), [170](#)

reinforcement learning [9](#), [453](#)

approaches [456](#)

deterministic [456](#)

policy-based approach [456](#)

stochastic [456](#)

value-based approach [456](#)

reinforcement learning, elements

action [454](#)

agent [454](#)

environment [453](#)

rewards [454](#)

states [454](#)

ReLU function [258](#)

Resilient Distributed Datasets (RDD) [189](#)

reference link [189](#)

returns [455](#)

rewards [453, 454](#)

ridge [169](#)

RNN architecture

learning [412](#)

RNN model

training [418, 419, 420](#)

RNN text generator

building [436, 437, 438](#)

training [438, 439, 440, 441, 444](#)

root [112](#)

root mean squared error (RMSE) [245](#)

rotation

for data augmentation [404](#)

Russell 2000 (RUT) index

reference link [218](#)

S

S3, Amazon Web Services

reference link [355](#)

scaling [29](#)

scikit-learn

decision tree, implementing [133](#), [134](#)

k-means, implementing with [329](#), [330](#), [331](#)

linear regression, implementing [232](#)

Naïve Bayes, implementing [59](#)

neural networks, implementing [265](#)

URL [38](#)

scikit-learn library [40](#)

SciPy [39](#)

Seaborn [40](#)

seaborn package

reference link [299](#)

self-attention [446](#)

semantics [294](#)

semi-supervised learning [10](#)

separating boundary

finding, with SVM [76](#)

separating hyperplane

identifying [77](#)

sequence [412](#)

sequence modeling [412](#)

sequential learning [412](#)

shifting

for data augmentation [405](#)

sigmoid function [152](#), [256](#), [268](#)

similarity querying [294](#)

SimpleImputer class

reference link [351](#)

single-layer neural network [254](#)

skip-gram [363](#)

softmax function [268](#)

softmax regression [175](#)

S&P 500 index

reference link [218](#)

spaCy [285, 290](#)

URL [286](#)

Spark

download link [186](#)

fundamentals [184](#)

installing [186, 187](#)

massive click logs, learning with [192](#)

used, for feature engineering on categorical variables [203](#)

Spark, cluster mode approaches

Apache Hadoop YARN [188](#)

Apache Mesos [188](#)

Kubernetes [188](#)

standalone cluster mode [188](#)

Spark, components [184](#)

GraphX [185](#)

MLlib [185](#)

Spark Core [185](#)

Spark SQL [185](#)

Spark Streaming [185](#)

Spark Core [185](#)

Spark, documentation and tutorials

reference link [185](#)

Spark programs

deploying [187](#)

launching [187](#)

Spark SQL [185](#)

Spark Streaming [185](#)

stacking [36](#)

states [453](#), [454](#)

statistical learning [11](#)

steepest descent [158](#)

stemming [292](#), [293](#), [305](#)

step size [158](#)

stochastic gradient descent

used, for training logistic regression model [166](#), [168](#), [169](#)

stochastic gradient descent (SGD) [232](#)

stock index [217](#)

stock market [214](#)

stock price data

mining [216](#), [217](#)

stock prices [214](#)

predicting, with neural networks [271](#)

predicting, with regression algorithms [246](#), [247](#), [248](#), [249](#), [250](#)

stop words

dropping [304](#), [305](#)

Storage, in Microsoft Azure

reference link [355](#)

sum of squared errors (SSE) [332](#)

sum of within-cluster distances [332](#)

supervised learning [9](#)

support vector machine (SVM) [48](#), [242](#)

support vector regression

estimating with [242](#), [244](#)

support vectors [76](#), [79](#)

SVM [370](#)

face images, classifying [98](#)

implementing [84](#), [85](#)

separating boundary, finding [76](#)

SVM-based image classifier

building [100](#), [101](#), [102](#)

SVR

implementing [244](#)

T

tanh function [256](#), [257](#), [268](#)

targets [315](#)

target variables [315](#)

Taxi environment

reference link [477](#)

simulating [477](#), [478](#), [479](#), [480](#), [481](#), [482](#)

Taxi problem

solving, with Q-learning algorithm [477](#)

Tay

reference link [284](#)

t-distributed Stochastic Neighbor Embedding (t-SNE)

for dimensionality reduction [308](#), [309](#), [310](#), [311](#)

newsgroups data, visualizing [307](#)

technical analysis

TensorFlow

linear regression, implementing [233](#), [234](#)

logistic regression, implementing [178](#), [180](#)

models, restoring [376](#), [377](#)

models, saving [376](#), [377](#)

neural networks, implementing [266](#), [267](#)

URL [38](#)

TensorFlow 2 [40](#)

term frequency-inverse document frequency (tf-idf) [335](#), [362](#)

term frequency (tf) [335](#), [362](#)

terminal node [112](#)

testing samples [13](#)

testing sets [13](#)

TextBlob [285](#)

URL [287](#)

text data, features [301](#)

inflectional and derivational forms of words, reducing [305](#), [306](#), [307](#)

occurrence, counting of word token [301](#), [302](#), [303](#), [304](#)

stop words, dropping [304](#), [305](#)

text preprocessing [304](#)

text datasets, NLTK

reference link [287](#)

text preprocessing [304](#)

tokenization [289](#), [290](#)

tokens [289](#)

topic [342](#)

topic model [337](#)

topic modeling [294](#), [338](#)

with latent Dirichlet allocation (LDA) [342](#), [343](#), [344](#), [345](#)

with non-negative matrix factorization (NMF) [338](#), [339](#), [340](#), [341](#)

Torch

URL [450](#)

traditional programming

versus machine learning [5](#)

training samples [13](#)

training sets [13](#)

training sets generation stage

best practices [355](#), [356](#), [357](#), [358](#), [359](#), [360](#), [361](#), [362](#), [363](#), [364](#)

Transformer model [444](#)

architecture [444](#), [446](#)

transition matrix [460](#)

true positive rate [66](#)

Turing test

reference link [283](#)

U

underfitting [16](#)

unigrams [289](#)

units [255](#)

unlabeled data [9](#), [315](#)

unsupervised learning [9](#), [314](#)

 association [315](#)

 clustering [315](#)

 projection [315](#)

 types [315](#)

unsupervised learning [308](#)

URL Reputation Dataset [97](#)

V

validation samples [13](#)

validation sets [13](#)

value iteration algorithm [460](#)

 FrozenLake, solving [460](#), [461](#), [462](#), [463](#), [464](#)

vanishing gradient problem [420](#)

variance [17](#)

voting [32](#)

W

War and Peace, writing with RNNs [431](#)

 RNN text generator, building [436](#), [437](#), [438](#)

RNN text generator, training [438](#), [439](#), [440](#), [442](#), [443](#), [444](#)
training data, acquiring [432](#), [433](#)
training data, analyzing [432](#), [433](#)
training set, constructing for RNN text generator [433](#), [435](#), [436](#)

weak learners [34](#)

weights [153](#)

word embedding [294](#), [363](#)

with pre-trained models [364](#), [365](#), [366](#), [367](#)

word token

occurrence, counting [301](#), [302](#), [303](#), [304](#)

word_tokenize function [290](#)

word vectorization [294](#)

working environment

setting up [450](#)

X

XGBoost package

reference link [144](#)

XOR gate

reference link [96](#)

Y

Yet Another Resource Negotiator (YARN) [188](#)

YouTube Multiview Video Games Dataset [97](#)