EXPERT INSIGHT

Machine Learning with PyTorch and Scikit-Learn

Develop machine learning and deep learning models with Python

PyTorch book of the bestselling and widely acclaimed *Python Machine Learning* series

Foreword by:
Dmytro Dzhulgakov
PyTorch Core Maintainer

Sebastian Raschka
Yuxi (Hayden) Liu
Vahid Mirjalili

Packt

Machine Learning with PyTorch and Scikit-Learn

Develop machine learning and deep learning models with Python

Sebastian Raschka
Yuxi (Hayden) Liu
Vahid Mirjalili

Packt
BIRMINGHAM—MUMBAI

Machine Learning with PyTorch and Scikit-Learn

Producer: Tushar Gupta
Acquisition Editor – Peer Reviews: Saby Dsilva
Project Editor: Janice Gonsalves
Content Development Editor: Bhavesh Amin
Copy Editor: Safis Editing
Technical Editor: Aniket Shetty
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Presentation Designer: Pranit Padwal

First published: February 2022

Production reference: 5151122

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

ISBN 978-1-80181-931-2

www.packt.com

Foreword

Over recent years, machine learning methods, with their ability to make sense of vast amounts of data and automate decisions, have found widespread applications in healthcare, robotics, biology, physics, consumer products, internet services, and various other industries.

Giant leaps in science usually come from a combination of powerful ideas and great tools. Machine learning is no exception. The success of data-driven learning methods is based on the ingenious ideas of thousands of talented researchers over the field's 60-year history. But their recent popularity is also fueled by the evolution of hardware and software solutions that make them scalable and accessible. The ecosystem of excellent libraries for numeric computing, data analysis, and machine learning built around Python like NumPy and scikit-learn gained wide adoption in research and industry. This has greatly helped propel Python to be the most popular programming language.

Massive improvements in computer vision, text, speech, and other tasks brought by the recent advent of deep learning techniques exemplify this theme. Approaches draw on neural network theory of the last four decades that started working remarkably well in combination with GPUs and highly optimized compute routines.

Our goal with building PyTorch over the past five years has been to give researchers the most flexible tool for expressing deep learning algorithms while taking care of the underlying engineering complexities. We benefited from the excellent Python ecosystem. In turn, we've been fortunate to see the community of very talented people build advanced deep learning models across various domains on top of PyTorch. The authors of this book were among them.

I've known Sebastian within this tight-knit community for a few years now. He has unmatched talent in easily explaining information and making the complex accessible. Sebastian contributed to many widely used machine learning software packages and authored dozens of excellent tutorials on deep learning and data visualization.

Mastery of both ideas and tools is also required to apply machine learning in practice. Getting started might feel intimidating, from making sense of theoretical concepts to figuring out which software packages to install.

Luckily, the book you're holding in your hands does a beautiful job of combining machine learning concepts and practical engineering steps to guide you in this journey. You're in for a delightful ride from the basics of data-driven techniques to the most novel deep learning architectures. Within every chapter, you will find concrete code examples applying the introduced methods to a practical task.

When the first edition came out in 2015, it set a very high bar for the *ML and Python* book category. But the excellence didn't stop there. With every edition, Sebastian and the team kept upgrading and refining the material as the deep learning revolution unfolded in new domains. In this new PyTorch edition, you'll find new chapters on transformer architectures and graph neural networks. These approaches are on the cutting edge of deep learning and have taken the fields of text understanding and molecular structure by storm in the last two years. You will get to practice them using new yet widely popular software packages in the ecosystem like Hugging Face, PyTorch Lightning, and PyTorch Geometric.

The excellent balance of theory and practice this book strikes is no surprise given the authors' combination of advanced research expertise and experience in solving problems hands-on. Sebastian Raschka and Vahid Mirjalili draw from their background in deep learning research for computer vision and computational biology. Hayden Liu brings the experience of applying machine learning methods to event prediction, recommendation systems, and other tasks in the industry. All of the authors share a deep passion for education, and it reflects in the approachable way the book goes from simple to advanced.

I'm confident that you will find this book invaluable both as a broad overview of the exciting field of machine learning and as a treasure of practical insights. I hope it inspires you to apply machine learning for the greater good in your problem area, whatever it might be.

Dmytro Dzhulgakov

PyTorch Core Maintainer

Contributors

About the authors

Dr. Sebastian Raschka is an Asst. Professor of Statistics at the University of Wisconsin-Madison focusing on machine learning and deep learning. His recent research focused on general challenges such as few-shot learning for working with limited data and developing deep neural networks for ordinal targets. Sebastian is also an avid open-source contributor, and in his new role as Lead AI Educator at Grid.ai, he plans to follow his passion for helping people to get into machine learning and AI.

Big thanks to Jitian Zhao and Ben Kaufman, with whom I had the pleasure to work on the new chapters on transformers and graph neural networks. I'm also very grateful for Hayden's and Vahid's help—this book wouldn't have been possible without you. Lastly, I want to thank Andrea Panizza, Tony Gitter, and Adam Bielski for helpful discussions on sections of the manuscript.

Yuxi (Hayden) Liu is a machine learning software engineer at Google and has worked as a machine learning scientist in a variety of data-driven domains. Hayden is the author of a series of ML books. His first book, *Python Machine Learning By Example*, was ranked the #1 bestseller in its category on Amazon in 2017 and 2018 and was translated into many 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 I worked with, especially my co-authors, my editors at Packt, and my reviewers. Without them, this book would be harder to read and to apply to real-world problems. Lastly, I'd like to thank all the readers for their support, which encouraged me to write the PyTorch edition of this bestselling ML book.

Dr. Vahid Mirjalili is a deep learning researcher focusing on computer vision applications. Vahid received a Ph.D. degree in both Mechanical Engineering and Computer Science from Michigan State University. During his Ph.D. journey, he developed novel computer vision algorithms to solve real-world problems and published several research articles that are highly cited in the computer vision community.

Other contributors

Benjamin Kaufman is a Ph.D. candidate at the University of Wisconsin-Madison in Biomedical Data Science. His research focuses on the development and application of machine learning methods for drug discovery. His work in this area has provided a deeper understanding of graph neural networks.

Jitian Zhao is a Ph.D. student at the University of Wisconsin-Madison, where she developed her interest in large-scale language models. She is passionate about deep learning in developing both real-world applications and theoretical support.

I would like to thank my parents for their support. They encouraged me to always pursue my dream and motivated me to be a good person.

About the reviewer

Roman Tezikov is an industrial research engineer and deep learning enthusiast with over four years of experience in advanced computer vision, NLP, and MLOps. As the co-creator of the ML-REPA community, he organized several workshops and meetups about ML reproducibility and pipeline automation. One of his current work challenges involves utilizing computer vision in the fashion industry. Roman was also a core developer of Catalyst – a PyTorch framework for accelerated deep learning.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:

`https://packt.link/MLwPyTorch`

Table of Contents

Chapter 4: Building Good Training Datasets – Data Preprocessing 105

Chapter 5: Compressing Data via Dimensionality Reduction 139

Chapter 14: Classifying Images with Deep Convolutional Neural Networks 451

Chapter 15: Modeling Sequential Data Using Recurrent Neural Networks 499

Preface

Through exposure to the news and social media, you probably are familiar with the fact that machine learning has become one of the most exciting technologies of our time and age. Large companies, such as Microsoft, Google, Meta, Apple, Amazon, IBM, and many more, heavily invest in machine learning research and applications for good reasons. While it may seem that machine learning has become the buzzword of our time and age, it is certainly not hype. This exciting field opens the way to new possibilities and has become indispensable to our daily lives. Talking to the voice assistant on our smartphones, recommending the right product for our customers, preventing credit card fraud, filtering out spam from our e-mail inboxes, detecting and diagnosing medical diseases, the list goes on and on.

If you want to become a machine learning practitioner, a better problem solver, or even consider a career in machine learning research, then this book is for you! However, for a novice, the theoretical concepts behind machine learning can be quite overwhelming. Yet, many practical books that have been published in recent years will help you get started in machine learning by implementing powerful learning algorithms.

Getting exposed to practical code examples and working through example applications of machine learning is a great way to dive into this field. Concrete examples help to illustrate the broader concepts by putting the learned material directly into action. However, remember that with great power comes great responsibility! In addition to offering hands-on experience with machine learning using Python and Python-based machine learning libraries, this book also introduces the mathematical concepts behind machine learning algorithms, which is essential for using machine learning successfully. Thus, this book is different from a purely practical book; it is a book that discusses the necessary details regarding machine learning concepts, offers intuitive yet informative explanations on how machine learning algorithms work, how to use them, and most importantly, how to avoid the most common pitfalls.

In this book, we will embark on an exciting journey that covers all the essential topics and concepts to give you a head start in this field. If you find that your thirst for knowledge is not satisfied, this book references many useful resources that you can use to follow up on the essential breakthroughs in this field.

Who this book is for

This book is the ideal companion for learning how to apply machine learning and deep learning to a wide range of tasks and datasets. If you are a programmer who wants to keep up with the recent trends in technology, this book is definitely for you. Also, if you are a student or considering a career transition, this book will be both your introduction and a comprehensive guide to the world of machine learning.

What this book covers

Chapter 1, *Giving Computers the Ability to Learn from Data*, introduces you to the main subareas of machine learning to tackle various problem tasks. In addition, it discusses the essential steps for creating a typical machine learning model building pipeline that will guide us through the following chapters.

Chapter 2, *Training Simple Machine Learning Algorithms for Classification*, goes back to the origins of machine learning and introduces binary perceptron classifiers and adaptive linear neurons. This chapter is a gentle introduction to the fundamentals of pattern classification and focuses on the interplay of optimization algorithms and machine learning.

Chapter 3, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, describes the essential machine learning algorithms for classification and provides practical examples using one of the most popular and comprehensive open-source machine learning libraries, scikit-learn.

Chapter 4, *Building Good Training Datasets – Data Preprocessing*, discusses how to deal with the most common problems in unprocessed datasets, such as missing data. It also discusses several approaches to identify the most informative features in datasets and teaches you how to prepare variables of different types as proper inputs for machine learning algorithms.

Chapter 5, *Compressing Data via Dimensionality Reduction*, describes the essential techniques to reduce the number of features in a dataset to smaller sets while retaining most of their useful and discriminatory information. It discusses the standard approach to dimensionality reduction via principal component analysis and compares it to supervised and nonlinear transformation techniques.

Chapter 6, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, discusses the do's and don'ts for estimating the performances of predictive models. Moreover, it discusses different metrics for measuring the performance of our models and techniques to fine-tune machine learning algorithms.

Chapter 7, *Combining Different Models for Ensemble Learning*, introduces you to the different concepts of combining multiple learning algorithms effectively. It teaches you how to build ensembles of experts to overcome the weaknesses of individual learners, resulting in more accurate and reliable predictions.

Chapter 8, *Applying Machine Learning to Sentiment Analysis*, discusses the essential steps to transform textual data into meaningful representations for machine learning algorithms to predict the opinions of people based on their writing.

Chapter 9, *Predicting Continuous Target Variables with Regression Analysis*, discusses the essential techniques for modeling linear relationships between target and response variables to make predictions on a continuous scale. After introducing different linear models, it also talks about polynomial regression and tree-based approaches.

Chapter 10, *Working with Unlabeled Data – Clustering Analysis*, shifts the focus to a different subarea of machine learning, unsupervised learning. We apply algorithms from three fundamental families of clustering algorithms to find groups of objects that share a certain degree of similarity.

Chapter 11, *Implementing a Multilayer Artificial Neural Network from Scratch*, extends the concept of gradient-based optimization, which we first introduced in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, to build powerful, multilayer neural networks based on the popular backpropagation algorithm in Python.

Chapter 12, *Parallelizing Neural Network Training with PyTorch*, builds upon the knowledge from the previous chapter to provide you with a practical guide for training neural networks more efficiently. The focus of this chapter is on PyTorch, an open-source Python library that allows us to utilize multiple cores of modern GPUs and construct deep neural networks from common building blocks via a user-friendly and flexible API.

Chapter 13, *Going Deeper – The Mechanics of PyTorch*, picks up where the previous chapter left off and introduces more advanced concepts and functionality of PyTorch. PyTorch is an extraordinarily vast and sophisticated library, and this chapter walks you through concepts such as dynamic computation graphs and automatic differentiation. You will also learn how to use PyTorch's object-oriented API to implement complex neural networks and how PyTorch Lightning helps you with best practices and minimizing boilerplate code.

Chapter 14, *Classifying Images with Deep Convolutional Neural Networks*, introduces **convolutional neural networks** (**CNNs**). A CNN represents a particular type of deep neural network architecture that is particularly well-suited for working with image datasets. Due to their superior performance compared to traditional approaches, CNNs are now widely used in computer vision to achieve state-of-the-art results for various image recognition tasks. Throughout this chapter, you will learn how convolutional layers can be used as powerful feature extractors for image classification.

Chapter 15, *Modeling Sequential Data Using Recurrent Neural Networks*, introduces another popular neural network architecture for deep learning that is especially well suited for working with text and other types of sequential data and time series data. As a warm-up exercise, this chapter introduces recurrent neural networks for predicting the sentiment of movie reviews. Then, we will teach recurrent networks to digest information from books in order to generate entirely new text.

Chapter 16, *Transformers – Improving Natural Language Processing with Attention Mechanisms*, focuses on the latest trends in natural language processing and explains how attention mechanisms help with modeling complex relationships in long sequences. In particular, this chapter describes the influential transformer architecture and state-of-the-art transformer models such as BERT and GPT.

Chapter 17, *Generative Adversarial Networks for Synthesizing New Data*, introduces a popular adversarial training regime for neural networks that can be used to generate new, realistic-looking images. The chapter starts with a brief introduction to autoencoders, which is a particular type of neural network architecture that can be used for data compression. The chapter then shows you how to combine the decoder part of an autoencoder with a second neural network that can distinguish between real and synthesized images. By letting two neural networks compete with each other in an adversarial training approach, you will implement a generative adversarial network that generates new handwritten digits.

Chapter 18, *Graph Neural Networks for Capturing Dependencies in Graph Structured Data*, goes beyond working with tabular datasets, images, and text. This chapter introduces graph neural networks that operate on graph-structured data, such as social media networks and molecules. After explaining the fundamentals of graph convolutions, this chapter includes a tutorial showing you how to implement predictive models for molecular data.

Chapter 19, *Reinforcement Learning for Decision Making in Complex Environments*, covers a subcategory of machine learning that is commonly used for training robots and other autonomous systems. This chapter starts by introducing the basics of **reinforcement learning** (**RL**) to become familiar with the agent/environment interactions, the reward process of RL systems, and the concept of learning from experience. After learning about the main categories of RL, you will implement and train an agent that can navigate in a grid world environment using the Q-learning algorithm. Finally, this chapter introduces the deep Q-learning algorithm, which is a variant of Q-learning that uses deep neural networks.

To get the most out of this book

Ideally, you are already comfortable with programming in Python to follow along with the code examples we provide to both illustrate and apply various algorithms and models. To get the most out of this book, a firm grasp of mathematical notation will be helpful as well.

A common laptop or desktop computer should be sufficient for running most of the code in this book, and we provide instructions for your Python environment in the first chapter. Later chapters will introduce additional libraries and installation recommendations when the need arises.

A recent **graphics processing unit** (**GPU**) can accelerate the code runtimes in the later deep learning chapters. However, a GPU is not required, and we also provide instructions for using free cloud resources.

Download the example code files

All code examples are available for download through GitHub at `https://github.com/rasbt/machine-learning-book`. We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

While we recommend using Jupyter Notebook for executing code interactively, all code examples are available in both a Python script (for example, `ch02/ch02.py`) and a Jupyter Notebook format (for example, `ch02/ch02.ipynb`). Furthermore, we recommend viewing the `README.md` file that accompanies each individual chapter for additional information and updates

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/9781801819312_ColorImages.pdf`. In addition, lower resolution color images are embedded in the code notebooks of this book that come bundled with the example code files.

Conventions

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

Here are some examples of these styles and an explanation of their meaning. Code words in text are shown as follows: "And already installed packages can be updated via the `--upgrade` flag."

A block of code is set as follows:

```
def __init__(self, eta=0.01, n_iter=50, random_state=1):
    self.eta = eta
    self.n_iter = n_iter
    self.random_state = random_state
```

Any input in the Python interpreter is written as follows (notice the >>> symbol). The expected output will be shown without the >>> symbol:

```
>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...           np.linalg.norm(v2)))
0.0
```

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

```
pip install gym==0.20
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Clicking the **Next** button moves you to the next screen."

Warnings or important notes appear in a box 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`.

Share your thoughts

Once you've read *Machine Learning with PyTorch and Scikit-Learn*, we'd love to hear your thoughts! Please `click here to go straight to the Amazon review page` for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a Free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?
Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781801819312`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1

Giving Computers the Ability to Learn from Data

In my opinion, **machine learning**, the application and science of algorithms that make sense of data, is the most exciting field of all the computer sciences! We are living in an age where data comes in abundance; using self-learning algorithms from the field of machine learning, we can turn this data into knowledge. Thanks to the many powerful open-source libraries that have been developed in recent years, there has probably never been a better time to break into the machine learning field and learn how to utilize powerful algorithms to spot patterns in data and make predictions about future events.

In this chapter, you will learn about the main concepts and different types of machine learning. Together with a basic introduction to the relevant terminology, we will lay the groundwork for successfully using machine learning techniques for practical problem solving.

In this chapter, we will cover the following topics:

- The general concepts of machine learning
- The three types of learning and basic terminology
- The building blocks for successfully designing machine learning systems
- Installing and setting up Python for data analysis and machine learning

Building intelligent machines to transform data into knowledge

In this age of modern technology, there is one resource that we have in abundance: a large amount of structured and unstructured data. In the second half of the 20th century, machine learning evolved as a subfield of **artificial intelligence** (**AI**) involving self-learning algorithms that derive knowledge from data to make predictions.

Instead of requiring humans to manually derive rules and build models from analyzing large amounts of data, machine learning offers a more efficient alternative for capturing the knowledge in data to gradually improve the performance of predictive models and make data-driven decisions.

Not only is machine learning becoming increasingly important in computer science research, but it is also playing an ever-greater role in our everyday lives. Thanks to machine learning, we enjoy robust email spam filters, convenient text and voice recognition software, reliable web search engines, recommendations on entertaining movies to watch, mobile check deposits, estimated meal delivery times, and much more. Hopefully, soon, we will add safe and efficient self-driving cars to this list. Also, notable progress has been made in medical applications; for example, researchers demonstrated that deep learning models can detect skin cancer with near-human accuracy (`https://www.nature.com/articles/nature21056`). Another milestone was recently achieved by researchers at DeepMind, who used deep learning to predict 3D protein structures, outperforming physics-based approaches by a substantial margin (`https://deepmind.com/blog/article/alphafold-a-solution-to-a-50-year-old-grand-challenge-in-biology`). While accurate 3D protein structure prediction plays an essential role in biological and pharmaceutical research, there have been many other important applications of machine learning in healthcare recently. For instance, researchers designed systems for predicting the oxygen needs of COVID-19 patients up to four days in advance to help hospitals allocate resources for those in need (`https://ai.facebook.com/blog/new-ai-research-to-help-predict-covid-19-resource-needs-from-a-series-of-x-rays/`). Another important topic of our day and age is climate change, which presents one of the biggest and most critical challenges. Today, many efforts are being directed toward developing intelligent systems to combat it (`https://www.forbes.com/sites/robtoews/2021/06/20/these-are-the-startups-applying-ai-to-tackle-climate-change`). One of the many approaches to tackling climate change is the emergent field of precision agriculture. Here, researchers aim to design computer vision-based machine learning systems to optimize resource deployment to minimize the use and waste of fertilizers.

The three different types of machine learning

In this section, we will take a look at the three types of machine learning: **supervised learning**, **unsupervised learning**, and **reinforcement learning**. We will learn about the fundamental differences between the three different learning types and, using conceptual examples, we will develop an understanding of the practical problem domains where they can be applied:

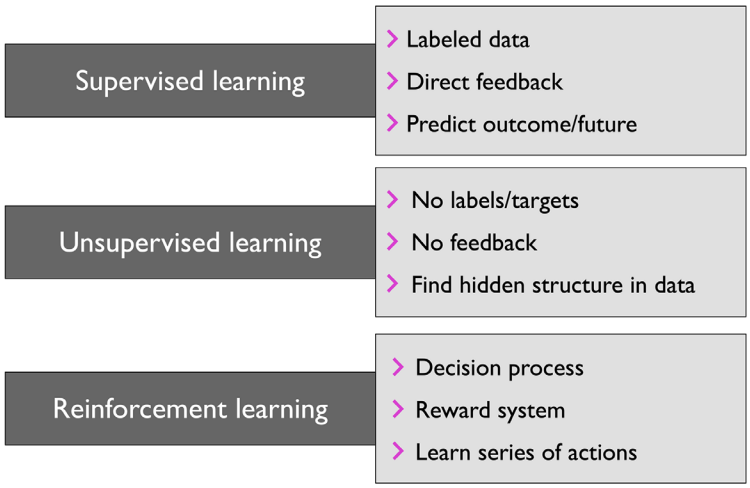


Figure 1.1: The three different types of machine learning

Making predictions about the future with supervised learning

The main goal in supervised learning is to learn a model from labeled training data that allows us to make predictions about unseen or future data. Here, the term "supervised" refers to a set of training examples (data inputs) where the desired output signals (labels) are already known. Supervised learning is then the process of modeling the relationship between the data inputs and the labels. Thus, we can also think of supervised learning as "label learning."

Figure 1.2 summarizes a typical supervised learning workflow, where the labeled training data is passed to a machine learning algorithm for fitting a predictive model that can make predictions on new, unlabeled data inputs:

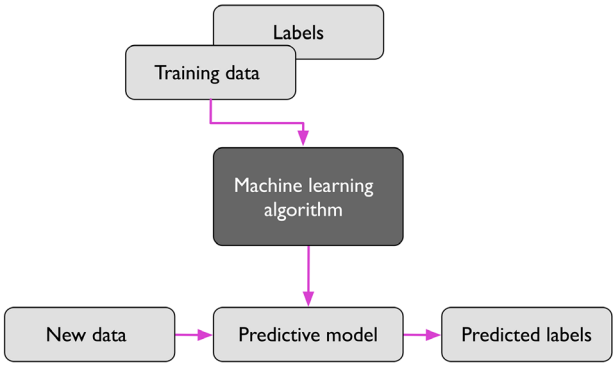


Figure 1.2: Supervised learning process

Considering the example of email spam filtering, we can train a model using a supervised machine learning algorithm on a corpus of labeled emails, which are correctly marked as spam or non-spam, to predict whether a new email belongs to either of the two categories. A supervised learning task with discrete class labels, such as in the previous email spam filtering example, is also called a **classification task**. Another subcategory of supervised learning is **regression**, where the outcome signal is a continuous value.

Classification for predicting class labels

Classification is a subcategory of supervised learning where the goal is to predict the categorical class labels of new instances or data points based on past observations. Those class labels are discrete, unordered values that can be understood as the group memberships of the data points. The previously mentioned example of email spam detection represents a typical example of a binary classification task, where the machine learning algorithm learns a set of rules to distinguish between two possible classes: spam and non-spam emails.

Figure 1.3 illustrates the concept of a binary classification task given 30 training examples; 15 training examples are labeled as class A and 15 training examples are labeled as class B. In this scenario, our dataset is two-dimensional, which means that each example has two values associated with it: x_1 and x_2. Now, we can use a supervised machine learning algorithm to learn a rule—the decision boundary represented as a dashed line—that can separate those two classes and classify new data into each of those two categories given its x_1 and x_2 values:

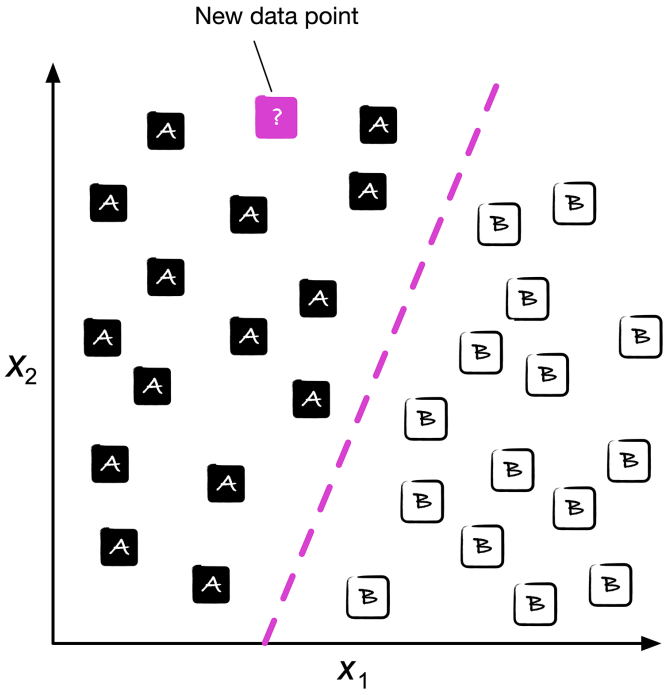


Figure 1.3: Classifying a new data point

However, the set of class labels does not have to be of a binary nature. The predictive model learned by a supervised learning algorithm can assign any class label that was presented in the training dataset to a new, unlabeled data point or instance.

A typical example of a **multiclass classification** task is handwritten character recognition. We can collect a training dataset that consists of multiple handwritten examples of each letter in the alphabet. The letters ("A," "B," "C," and so on) will represent the different unordered categories or class labels that we want to predict. Now, if a user provides a new handwritten character via an input device, our predictive model will be able to predict the correct letter in the alphabet with certain accuracy. However, our machine learning system will be unable to correctly recognize any of the digits between 0 and 9, for example, if they were not part of the training dataset.

Regression for predicting continuous outcomes

We learned in the previous section that the task of classification is to assign categorical, unordered labels to instances. A second type of supervised learning is the prediction of continuous outcomes, which is also called **regression analysis**. In regression analysis, we are given a number of predictor (**explanatory**) variables and a continuous response variable (**outcome**), and we try to find a relationship between those variables that allows us to predict an outcome.

Note that in the field of machine learning, the predictor variables are commonly called "features," and the response variables are usually referred to as "target variables." We will adopt these conventions throughout this book.

For example, let's assume that we are interested in predicting the math SAT scores of students. (The SAT is a standardized test frequently used for college admissions in the United States.) If there is a relationship between the time spent studying for the test and the final scores, we could use it as training data to learn a model that uses the study time to predict the test scores of future students who are planning to take this test.

Regression toward the mean

The term "regression" was devised by Francis Galton in his article *Regression towards Mediocrity in Hereditary Stature* in 1886. Galton described the biological phenomenon that the variance of height in a population does not increase over time.

He observed that the height of parents is not passed on to their children, but instead, their children's height regresses toward the population mean.

Figure 1.4 illustrates the concept of linear regression. Given a feature variable, x, and a target variable, y, we fit a straight line to this data that minimizes the distance—most commonly the average squared distance—between the data points and the fitted line.

We can now use the intercept and slope learned from this data to predict the target variable of new data:

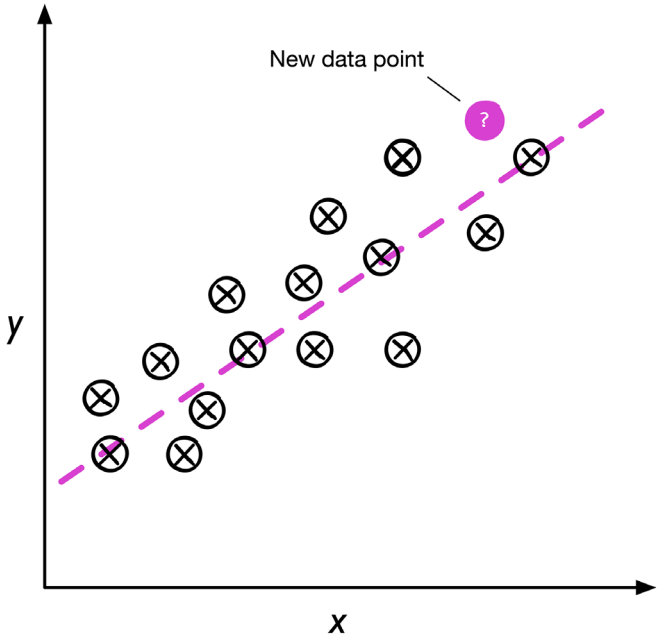


Figure 1.4: A linear regression example

Solving interactive problems with reinforcement learning

Another type of machine learning is **reinforcement learning**. In reinforcement learning, the goal is to develop a system (**agent**) that improves its performance based on interactions with the environment. Since the information about the current state of the environment typically also includes a so-called **reward signal**, we can think of reinforcement learning as a field related to supervised learning. However, in reinforcement learning, this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a reward function. Through its interaction with the environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward via an exploratory trial-and-error approach or deliberative planning.

A popular example of reinforcement learning is a chess program. Here, the agent decides upon a series of moves depending on the state of the board (the environment), and the reward can be defined as **win** or **lose** at the end of the game:

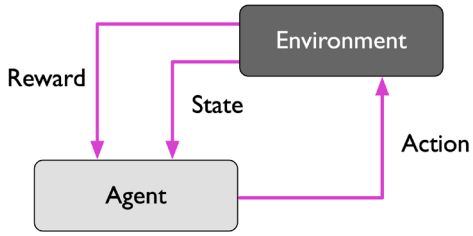


Figure 1.5: Reinforcement learning process

There are many different subtypes of reinforcement learning. However, a general scheme is that the agent in reinforcement learning tries to maximize the reward through a series of interactions with the environment. Each state can be associated with a positive or negative reward, and a reward can be defined as accomplishing an overall goal, such as winning or losing a game of chess. For instance, in chess, the outcome of each move can be thought of as a different state of the environment.

To explore the chess example further, let's think of visiting certain configurations on the chessboard as being associated with states that will more likely lead to winning—for instance, removing an opponent's chess piece from the board or threatening the queen. Other positions, however, are associated with states that will more likely result in losing the game, such as losing a chess piece to the opponent in the following turn. Now, in the game of chess, the reward (either positive for winning or negative for losing the game) will not be given until the end of the game. In addition, the final reward will also depend on how the opponent plays. For example, the opponent may sacrifice the queen but eventually win the game.

In sum, reinforcement learning is concerned with learning to choose a series of actions that maximizes the total reward, which could be earned either immediately after taking an action or via *delayed* feedback.

Discovering hidden structures with unsupervised learning

In supervised learning, we know the right answer (the label or target variable) beforehand when we train a model, and in reinforcement learning, we define a measure of reward for particular actions carried out by the agent. In unsupervised learning, however, we are dealing with unlabeled data or data of an unknown structure. Using unsupervised learning techniques, we are able to explore the structure of our data to extract meaningful information without the guidance of a known outcome variable or reward function.

Finding subgroups with clustering

Clustering is an exploratory data analysis or pattern discovery technique that allows us to organize a pile of information into meaningful subgroups (**clusters**) without having any prior knowledge of their group memberships. Each cluster that arises during the analysis defines a group of objects that share a certain degree of similarity but are more dissimilar to objects in other clusters, which is why clustering is also sometimes called **unsupervised classification**. Clustering is a great technique for structuring information and deriving meaningful relationships from data. For example, it allows marketers to discover customer groups based on their interests, in order to develop distinct marketing programs.

Figure 1.6 illustrates how clustering can be applied to organizing unlabeled data into three distinct groups or clusters (A, B, and C, in arbitrary order) based on the similarity of their features, x_1 and x_2:

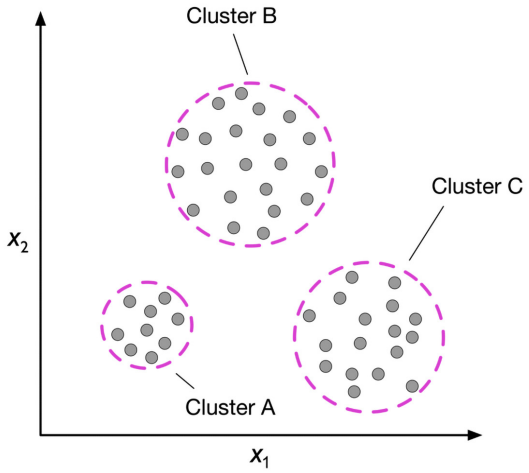


Figure 1.6: How clustering works

Dimensionality reduction for data compression

Another subfield of unsupervised learning is **dimensionality reduction**. Often, we are working with data of high dimensionality—each observation comes with a high number of measurements—that can present a challenge for limited storage space and the computational performance of machine learning algorithms. Unsupervised dimensionality reduction is a commonly used approach in feature preprocessing to remove noise from data, which can degrade the predictive performance of certain algorithms. Dimensionality reduction compresses the data onto a smaller dimensional subspace while retaining most of the relevant information.

Sometimes, dimensionality reduction can also be useful for visualizing data; for example, a high-dimensional feature set can be projected onto one-, two-, or three-dimensional feature spaces to visualize it via 2D or 3D scatterplots or histograms. *Figure 1.7* shows an example where nonlinear dimensionality reduction was applied to compress a 3D Swiss roll onto a new 2D feature subspace:

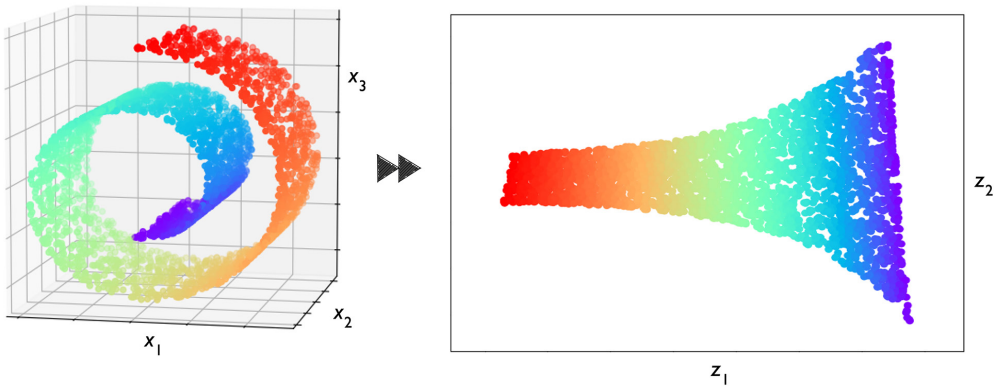


Figure 1.7: An example of dimensionality reduction from three to two dimensions

Introduction to the basic terminology and notations

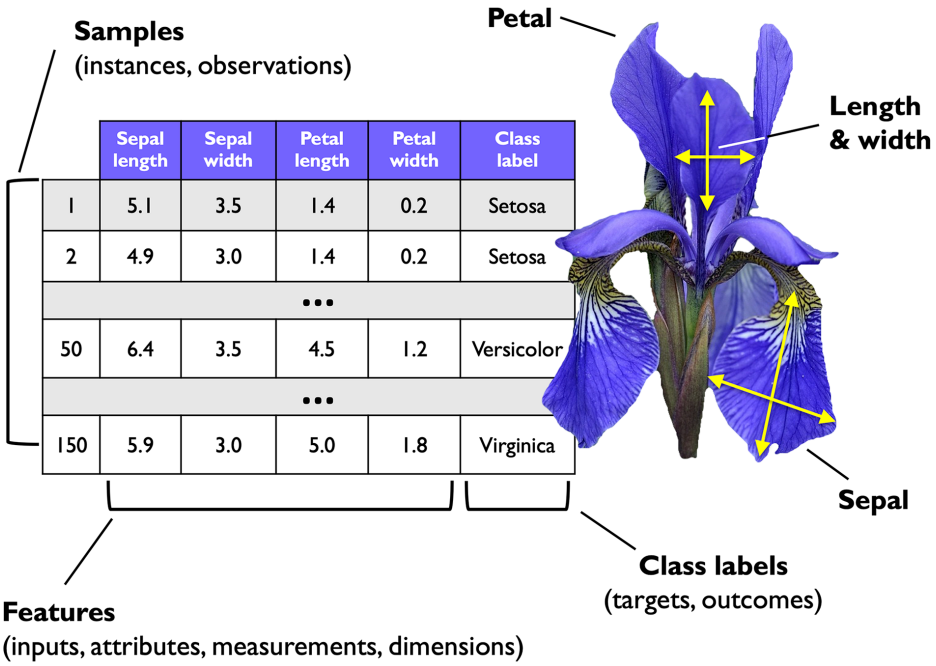
Now that we have discussed the three broad categories of machine learning—supervised, unsupervised, and reinforcement learning—let's have a look at the basic terminology that we will be using throughout this book. The following subsection covers the common terms we will be using when referring to different aspects of a dataset, as well as the mathematical notation to communicate more precisely and efficiently.

As machine learning is a vast field and very interdisciplinary, you are guaranteed to encounter many different terms that refer to the same concepts sooner rather than later. The second subsection collects many of the most commonly used terms that are found in machine learning literature, which may be useful to you as a reference section when reading machine learning publications.

Notation and conventions used in this book

Figure 1.8 depicts an excerpt of the Iris dataset, which is a classic example in the field of machine learning (more information can be found at `https://archive.ics.uci.edu/ml/datasets/iris`). The Iris dataset contains the measurements of 150 Iris flowers from three different species—Setosa, Versicolor, and Virginica.

Here, each flower example represents one row in our dataset, and the flower measurements in centimeters are stored as columns, which we also call the **features** of the dataset:



	Sepal length	Sepal width	Petal length	Petal width	Class label
1	5.1	3.5	1.4	0.2	Setosa
2	4.9	3.0	1.4	0.2	Setosa
...					
50	6.4	3.5	4.5	1.2	Versicolor
...					
150	5.9	3.0	5.0	1.8	Virginica

Figure 1.8: The Iris dataset

To keep the notation and implementation simple yet efficient, we will make use of some of the basics of linear algebra. In the following chapters, we will use a matrix notation to refer to our data. We will follow the common convention to represent each example as a separate row in a feature matrix, $\boldsymbol{X}$, where each feature is stored as a separate column.

The Iris dataset, consisting of 150 examples and four features, can then be written as a 150×4 matrix, formally denoted as $\boldsymbol{X} \in \mathbb{R}^{150\times 4}$:

$$\begin{bmatrix} x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & x_4^{(1)} \\ x_1^{(2)} & x_2^{(2)} & x_3^{(2)} & x_4^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(150)} & x_2^{(150)} & x_3^{(150)} & x_4^{(150)} \end{bmatrix}$$

Notational conventions

For most parts of this book, unless noted otherwise, we will use the superscript *i* to refer to the *i*th training example, and the subscript *j* to refer to the *j*th dimension of the training dataset.

We will use lowercase, bold-face letters to refer to vectors ($\boldsymbol{x} \in \mathbb{R}^{n\times 1}$) and uppercase, bold-face letters to refer to matrices ($\boldsymbol{X} \in \mathbb{R}^{n\times m}$). To refer to single elements in a vector or matrix, we will write the letters in italics ($x^{(n)}$ or $x_m^{(n)}$, respectively).

For example, $x_1^{(150)}$ refers to the first dimension of flower example 150, the sepal length. Each row in matrix $\boldsymbol{X}$ represents one flower instance and can be written as a four-dimensional row vector, $\boldsymbol{x}^{(i)} \in \mathbb{R}^{1\times 4}$:

$$\boldsymbol{X}^{(i)} = \begin{bmatrix} x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & x_4^{(i)} \end{bmatrix}$$

And each feature dimension is a 150-dimensional column vector, $\boldsymbol{X}^{(i)} \in \mathbb{R}^{150\times 1}$. For example:

$$\boldsymbol{x_j} = \begin{bmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \cdots \\ x_j^{(150)} \end{bmatrix}$$

Similarly, we can represent the target variables (here, class labels) as a 150-dimensional column vector:

$$\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ \cdots \\ y^{(150)} \end{bmatrix}, \text{ where } y^{(i)} \in \{\text{Setosa, Versicolor, Virginica}\}$$

Machine learning terminology

Machine learning is a vast field and also very interdisciplinary as it brings together many scientists from other areas of research. As it happens, many terms and concepts have been rediscovered or redefined and may already be familiar to you but appear under different names. For your convenience, in the following list, you can find a selection of commonly used terms and their synonyms that you may find useful when reading this book and machine learning literature in general:

- **Training example**: A row in a table representing the dataset and synonymous with an observation, record, instance, or sample (in most contexts, sample refers to a collection of training examples).
- **Training**: Model fitting, for parametric models similar to parameter estimation.

- **Feature, abbrev. *x***: A column in a data table or data (design) matrix. Synonymous with predictor, variable, input, attribute, or covariate.
- **Target, abbrev. *y***: Synonymous with outcome, output, response variable, dependent variable, (class) label, and ground truth.
- **Loss function**: Often used synonymously with a *cost* function. Sometimes the loss function is also called an *error* function. In some literature, the term "loss" refers to the loss measured for a single data point, and the cost is a measurement that computes the loss (average or summed) over the entire dataset.

A roadmap for building machine learning systems

In previous sections, we discussed the basic concepts of machine learning and the three different types of learning. In this section, we will discuss the other important parts of a machine learning system accompanying the learning algorithm.

Figure 1.9 shows a typical workflow for using machine learning in predictive modeling, which we will discuss in the following subsections:

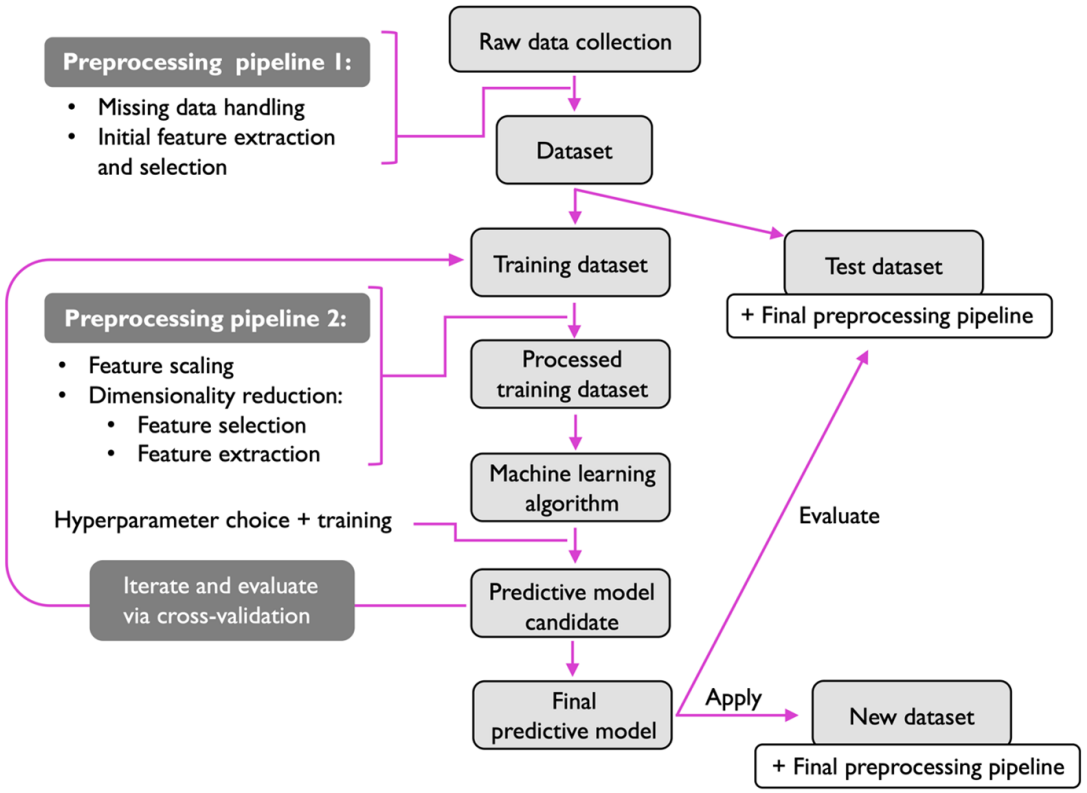


Figure 1.9: Predictive modeling workflow

Preprocessing – getting data into shape

Let's begin by discussing the roadmap for building machine learning systems. Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm. Thus, the preprocessing of the data is one of the most crucial steps in any machine learning application.

If we take the Iris flower dataset from the previous section as an example, we can think of the raw data as a series of flower images from which we want to extract meaningful features. Useful features could be centered around the color of the flowers or the height, length, and width of the flowers.

Many machine learning algorithms also require that the selected features are on the same scale for optimal performance, which is often achieved by transforming the features in the range [0, 1] or a standard normal distribution with zero mean and unit variance, as we will see in later chapters.

Some of the selected features may be highly correlated and therefore redundant to a certain degree. In those cases, dimensionality reduction techniques are useful for compressing the features onto a lower-dimensional subspace. Reducing the dimensionality of our feature space has the advantage that less storage space is required, and the learning algorithm can run much faster. In certain cases, dimensionality reduction can also improve the predictive performance of a model if the dataset contains a large number of irrelevant features (or noise); that is, if the dataset has a low signal-to-noise ratio.

To determine whether our machine learning algorithm not only performs well on the training dataset but also generalizes well to new data, we also want to randomly divide the dataset into separate training and test datasets. We use the training dataset to train and optimize our machine learning model, while we keep the test dataset until the very end to evaluate the final model.

Training and selecting a predictive model

As you will see in later chapters, many different machine learning algorithms have been developed to solve different problem tasks. An important point that can be summarized from David Wolpert's famous *No free lunch theorems* is that we can't get learning "for free" (*The Lack of A Priori Distinctions Between Learning Algorithms*, D.H. Wolpert, 1996; *No free lunch theorems for optimization*, D.H. Wolpert and W.G. Macready, 1997). We can relate this concept to the popular saying, *I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail* (Abraham Maslow, 1966). For example, each classification algorithm has its inherent biases, and no single classification model enjoys superiority if we don't make any assumptions about the task. In practice, it is therefore essential to compare at least a handful of different learning algorithms in order to train and select the best performing model. But before we can compare different models, we first have to decide upon a metric to measure performance. One commonly used metric is classification accuracy, which is defined as the proportion of correctly classified instances.

One legitimate question to ask is this: how do we know which model performs well on the final test dataset and real-world data if we don't use this test dataset for the model selection, but keep it for the final model evaluation? To address the issue embedded in this question, different techniques summarized as "cross-validation" can be used. In cross-validation, we further divide a dataset into training and validation subsets in order to estimate the generalization performance of the model.

Finally, we also cannot expect that the default parameters of the different learning algorithms provided by software libraries are optimal for our specific problem task. Therefore, we will make frequent use of hyperparameter optimization techniques that help us to fine-tune the performance of our model in later chapters.

We can think of those hyperparameters as parameters that are not learned from the data but represent the knobs of a model that we can turn to improve its performance. This will become much clearer in later chapters when we see actual examples.

Evaluating models and predicting unseen data instances

After we have selected a model that has been fitted on the training dataset, we can use the test dataset to estimate how well it performs on this unseen data to estimate the so-called *generalization error*. If we are satisfied with its performance, we can now use this model to predict new, future data. It is important to note that the parameters for the previously mentioned procedures, such as feature scaling and dimensionality reduction, are solely obtained from the training dataset, and the same parameters are later reapplied to transform the test dataset, as well as any new data instances—the performance measured on the test data may be overly optimistic otherwise.

Using Python for machine learning

Python is one of the most popular programming languages for data science, and thanks to its very active developer and open-source community, a large number of useful libraries for scientific computing and machine learning have been developed.

Although the performance of interpreted languages, such as Python, for computation-intensive tasks is inferior to lower-level programming languages, extension libraries such as NumPy and SciPy have been developed that build upon lower-layer Fortran and C implementations for fast vectorized operations on multidimensional arrays.

For machine learning programming tasks, we will mostly refer to the scikit-learn library, which is currently one of the most popular and accessible open-source machine learning libraries. In the later chapters, when we focus on a subfield of machine learning called *deep learning*, we will use the latest version of the PyTorch library, which specializes in training so-called *deep neural network* models very efficiently by utilizing graphics cards.

Installing Python and packages from the Python Package Index

Python is available for all three major operating systems—Microsoft Windows, macOS, and Linux—and the installer, as well as the documentation, can be downloaded from the official Python website: `https://www.python.org`.

The code examples provided in this book have been written for and tested in Python 3.9, and we generally recommend that you use the most recent version of Python 3 that is available. Some of the code may also be compatible with Python 2.7, but as the official support for Python 2.7 ended in 2019, and the majority of open-source libraries have already stopped supporting Python 2.7 (`https://python3statement.org`), we strongly advise that you use Python 3.9 or newer.

You can check your Python version by executing

```
python --version
```

or

```
python3 --version
```

in your terminal (or PowerShell if you are using Windows).

The additional packages that we will be using throughout this book can be installed via the `pip` installer program, which has been part of the Python Standard Library since Python 3.3. More information about `pip` can be found at `https://docs.python.org/3/installing/index.html`.

After we have successfully installed Python, we can execute `pip` from the terminal to install additional Python packages:

```
pip install SomePackage
```

Already installed packages can be updated via the `--upgrade` flag:

```
pip install SomePackage --upgrade
```

Using the Anaconda Python distribution and package manager

A highly recommended open-source package management system for installing Python for scientific computing contexts is conda by Continuum Analytics. Conda is free and licensed under a permissive open-source license. Its goal is to help with the installation and version management of Python packages for data science, math, and engineering across different operating systems. If you want to use conda, it comes in different flavors, namely Anaconda, Miniconda, and Miniforge:

- Anaconda comes with many scientific computing packages pre-installed. The Anaconda installer can be downloaded at `https://docs.anaconda.com/anaconda/install/`, and an Anaconda quick start guide is available at `https://docs.anaconda.com/anaconda/user-guide/getting-started/`.
- Miniconda is a leaner alternative to Anaconda (`https://docs.conda.io/en/latest/miniconda.html`). Essentially, it is similar to Anaconda but without any packages pre-installed, which many people (including the authors) prefer.
- Miniforge is similar to Miniconda but community-maintained and uses a different package repository (conda-forge) from Miniconda and Anaconda. We found that Miniforge is a great alternative to Miniconda. Download and installation instructions can be found in the GitHub repository at `https://github.com/conda-forge/miniforge`.

After successfully installing conda through either Anaconda, Miniconda, or Miniforge, we can install new Python packages using the following command:

```
conda install SomePackage
```

Existing packages can be updated using the following command:

```
conda update SomePackage
```

Packages that are not available through the official conda channel might be available via the community-supported conda-forge project (`https://conda-forge.org`), which can be specified via the `--channel conda-forge` flag. For example:

```
conda install SomePackage --channel conda-forge
```

Packages that are not available through the default conda channel or conda-forge can be installed via `pip` as explained earlier. For example:

```
pip install SomePackage
```

Packages for scientific computing, data science, and machine learning

Throughout the first half of this book, we will mainly use NumPy's multidimensional arrays to store and manipulate data. Occasionally, we will make use of pandas, which is a library built on top of NumPy that provides additional higher-level data manipulation tools that make working with tabular data even more convenient. To augment your learning experience and visualize quantitative data, which is often extremely useful to make sense of it, we will use the very customizable Matplotlib library.

The main machine learning library used in this book is scikit-learn (*Chapters 3* to *11*). *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*, will then introduce the PyTorch library for deep learning.

The version numbers of the major Python packages that were used to write this book are mentioned in the following list. Please make sure that the version numbers of your installed packages are, ideally, equal to these version numbers to ensure that the code examples run correctly:

- NumPy 1.21.2
- SciPy 1.7.0
- Scikit-learn 1.0
- Matplotlib 3.4.3
- pandas 1.3.2

After installing these packages, you can double-check the installed version by importing the package in Python and accessing its `__version__` attribute, for example:

```
>>> import numpy
>>> numpy.__version__
'1.21.2'
```

For your convenience, we included a `python-environment-check.py` script in this book's complimentary code repository at `https://github.com/rasbt/machine-learning-book` so that you can check both your Python version and the package versions by executing this script.

Certain chapters will require additional packages and will provide information about the installations. For instance, do not worry about installing PyTorch at this point. *Chapter 12* will provide tips and instructions when you need them.

If you encounter errors even though your code matches the code in the chapter exactly, we recommend you first check the version numbers of the underlying packages before spending more time on debugging or reaching out to the publisher or authors. Sometimes, newer versions of libraries introduce backward-incompatible changes that could explain these errors.

If you do not want to change the package version in your main Python installation, we recommend using a virtual environment for installing the packages used in this book. If you use Python without the conda manager, you can use the `venv` library to create a new virtual environment. For example, you can create and activate the virtual environment via the following two commands:

```
python3 -m venv /Users/sebastian/Desktop/pyml-book
source /Users/sebastian/Desktop/pyml-book/bin/activate
```

Note that you need to activate the virtual environment every time you open a new terminal or PowerShell. You can find more information about `venv` at `https://docs.python.org/3/library/venv.html`.

If you are using Anaconda with the conda package manager, you can create and activate a virtual environment as follows:

```
conda create -n pyml python=3.9
conda activate pyml
```

Summary

In this chapter, we explored machine learning at a very high level and familiarized ourselves with the big picture and major concepts that we are going to explore in the following chapters in more detail. We learned that supervised learning is composed of two important subfields: classification and regression. While classification models allow us to categorize objects into known classes, we can use regression analysis to predict the continuous outcomes of target variables. Unsupervised learning not only offers useful techniques for discovering structures in unlabeled data, but it can also be useful for data compression in feature preprocessing steps.

We briefly went over the typical roadmap for applying machine learning to problem tasks, which we will use as a foundation for deeper discussions and hands-on examples in the following chapters. Finally, we set up our Python environment and installed and updated the required packages to get ready to see machine learning in action.

Later in this book, in addition to machine learning itself, we will introduce different techniques to preprocess a dataset, which will help you to get the best performance out of different machine learning algorithms. While we will cover classification algorithms quite extensively throughout the book, we will also explore different techniques for regression analysis and clustering.

We have an exciting journey ahead, covering many powerful techniques in the vast field of machine learning. However, we will approach machine learning one step at a time, building upon our knowledge gradually throughout the chapters of this book. In the following chapter, we will start this journey by implementing one of the earliest machine learning algorithms for classification, which will prepare us for *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, where we will cover more advanced machine learning algorithms using the scikit-learn open-source machine learning library.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`

2

Training Simple Machine Learning Algorithms for Classification

In this chapter, we will make use of two of the first algorithmically described machine learning algorithms for classification: the perceptron and adaptive linear neurons. We will start by implementing a perceptron step by step in Python and training it to classify different flower species in the Iris dataset. This will help us to understand the concept of machine learning algorithms for classification and how they can be efficiently implemented in Python.

Discussing the basics of optimization using adaptive linear neurons will then lay the groundwork for using more sophisticated classifiers via the scikit-learn machine learning library in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*.

The topics that we will cover in this chapter are as follows:

- Building an understanding of machine learning algorithms
- Using pandas, NumPy, and Matplotlib to read in, process, and visualize data
- Implementing linear classifiers for 2-class problems in Python

Artificial neurons – a brief glimpse into the early history of machine learning

Before we discuss the perceptron and related algorithms in more detail, let's take a brief tour of the beginnings of machine learning. Trying to understand how the biological brain works in order to design an **artificial intelligence** (**AI**), Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell, the so-called **McCulloch-Pitts** (**MCP**) neuron, in 1943 (*A Logical Calculus of the Ideas Immanent in Nervous Activity* by *W. S. McCulloch* and *W. Pitts*, *Bulletin of Mathematical Biophysics*, 5(4): 115-133, 1943).

Biological neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in *Figure 2.1*:

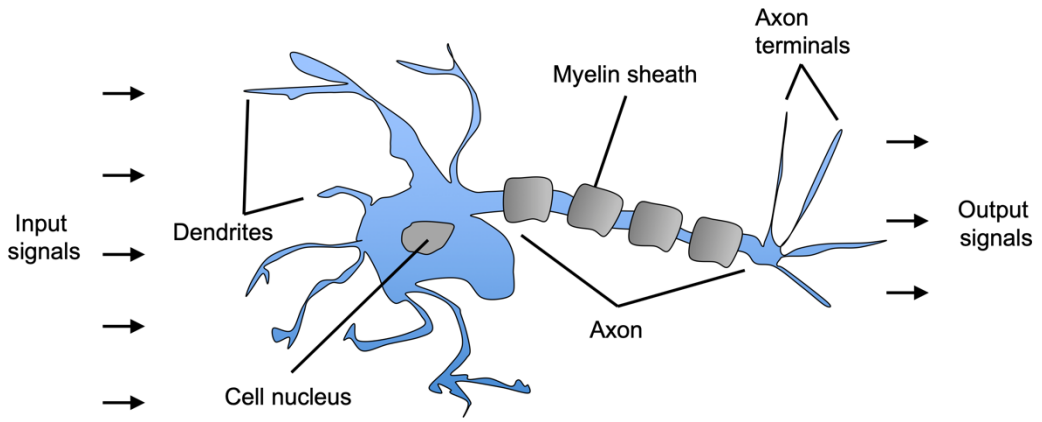


Figure 2.1: A neuron processing chemical and electrical signals

McCulloch and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, they are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (*The Perceptron: A Perceiving and Recognizing Automaton* by *F. Rosenblatt*, *Cornell Aeronautical Laboratory*, 1957). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that would then be multiplied with the input features in order to make the decision of whether a neuron fires (transmits a signal) or not. In the context of supervised learning and classification, such an algorithm could then be used to predict whether a new data point belongs to one class or the other.

The formal definition of an artificial neuron

More formally, we can put the idea behind **artificial neurons** into the context of a binary classification task with two classes: 0 and 1. We can then define a decision function, $\sigma(z)$, that takes a linear combination of certain input values, x, and a corresponding weight vector, w, where z is the so-called net input $z = w_1x_1 + w_2x_2 + \ldots + w_mx_m$:

$$\boldsymbol{w} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

Now, if the net input of a particular example, $\boldsymbol{x}^{(i)}$, is greater than a defined threshold, θ, we predict class 1, and class 0 otherwise. In the perceptron algorithm, the decision function, $\sigma(\cdot)$, is a variant of a **unit step function**:

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

To simplify the code implementation later, we can modify this setup via a couple of steps. First, we move the threshold, θ, to the left side of the equation:

$$z \geq \theta$$
$$z - \theta \geq 0$$

Second, we define a so-called *bias unit* as $b = -\theta$ and make it part of the net input:

$$z = w_1x_1 + \ldots + w_mx_m + b = \boldsymbol{w}^T\boldsymbol{x} + b$$

Third, given the introduction of the bias unit and the redefinition of the net input z above, we can redefine the decision function as follows:

$$\sigma(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Linear algebra basics: dot product and matrix transpose

In the following sections, we will often make use of basic notations from linear algebra. For example, we will abbreviate the sum of the products of the values in $\boldsymbol{x}$ and $\boldsymbol{w}$ using a vector dot product, whereas the superscript T stands for transpose, which is an operation that transforms a column vector into a row vector and vice versa. For example, assume we have the following two column vectors:

$$\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}, \quad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Then, we can write the transpose of vector $\boldsymbol{a}$ as $\boldsymbol{a}^T = [a_1\ a_2\ a_3]$ and write the dot product as

$$\boldsymbol{a}^T\boldsymbol{b} = \sum_i a_i b_i = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

Furthermore, the transpose operation can also be applied to matrices to reflect it over its diagonal, for example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Please note that the transpose operation is strictly only defined for matrices; however, in the context of machine learning, we refer to $n \times 1$ or $1 \times m$ matrices when we use the term "vector."

In this book, we will only use very basic concepts from linear algebra; however, if you need a quick refresher, please take a look at Zico Kolter's excellent *Linear Algebra Review and Reference*, which is freely available at `http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf`.

Figure 2.2 illustrates how the net input $z = \boldsymbol{w}^T\boldsymbol{x} + b$ is squashed into a binary output (0 or 1) by the decision function of the perceptron (left subfigure) and how it can be used to discriminate between two classes separable by a linear decision boundary (right subfigure):

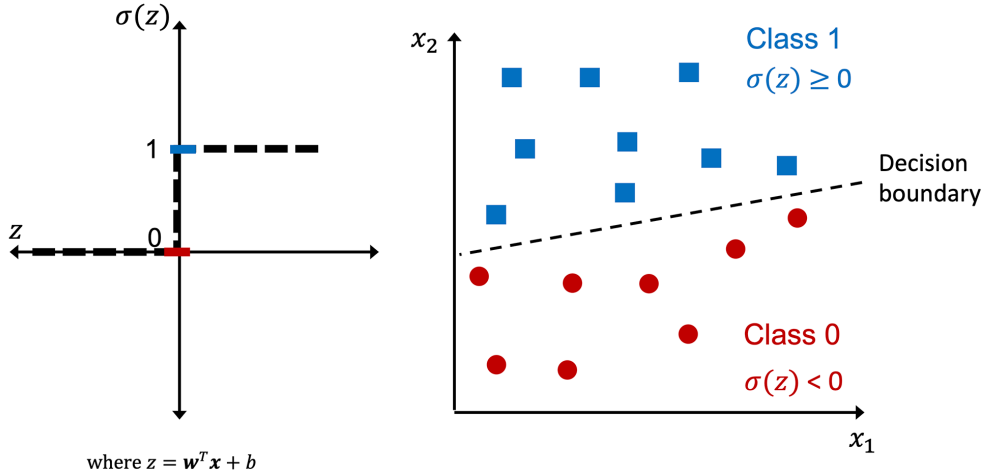


Figure 2.2: A threshold function producing a linear decision boundary for a binary classification problem

The perceptron learning rule

The whole idea behind the MCP neuron and Rosenblatt's *thresholded* perceptron model is to use a reductionist approach to mimic how a single neuron in the brain works: it either *fires* or it doesn't. Thus, Rosenblatt's classic perceptron rule is fairly simple, and the perceptron algorithm can be summarized by the following steps:

1. Initialize the weights and bias unit to 0 or small random numbers
2. For each training example, $\boldsymbol{x}^{(i)}$:
    a. Compute the output value, $\hat{y}^{(i)}$
    b. Update the weights and bias unit

Here, the output value is the class label predicted by the unit step function that we defined earlier, and the simultaneous update of the bias unit and each weight, w_j, in the weight vector, $\boldsymbol{w}$, can be more formally written as:

$$w_j := w_j + \Delta w_j$$
$$\text{and} \quad b := b + \Delta b$$

The update values ("deltas") are computed as follows:

$$\Delta w_j = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)x_j^{(i)}$$
$$\text{and} \quad \Delta b = \eta\left(y^{(i)} - \hat{y}^{(i)}\right)$$

Note that unlike the bias unit, each weight, w_j, corresponds to a feature, x_j, in the dataset, which is involved in determining the update value, Δw_j, defined above. Furthermore, η is the **learning rate** (typically a constant between 0.0 and 1.0), $y^{(i)}$ is the **true class label** of the ith training example, and $\hat{y}^{(i)}$ is the **predicted class label**. It is important to note that the bias unit and all weights in the weight vector are being updated simultaneously, which means that we don't recompute the predicted label, $\hat{y}^{(i)}$, before the bias unit and all of the weights are updated via the respective update values, Δw_j and Δb. Concretely, for a two-dimensional dataset, we would write the update as:

$$\begin{aligned}\Delta w_1 &= \eta(y^{(i)} - \text{output}^{(i)})x_1^{(i)};\\ \Delta w_2 &= \eta(y^{(i)} - \text{output}^{(i)})x_2^{(i)};\\ \Delta b &= \eta(y^{(i)} - \text{output}^{(i)})\end{aligned}$$

Before we implement the perceptron rule in Python, let's go through a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the bias unit and weights remain unchanged, since the update values are 0:

(1) $y^{(i)} = 0, \quad \hat{y}^{(i)} = 0, \quad \Delta w_j = \eta(0-0)x_j^{(i)} = 0, \quad \Delta b = \eta(0-0) = 0$

(2) $y^{(i)} = 1, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(1-1)x_j^{(i)} = 0, \quad \Delta b = \eta(1-1) = 0$

However, in the case of a wrong prediction, the weights are being pushed toward the direction of the positive or negative target class:

(3) $y^{(i)} = 1, \quad \hat{y}^{(i)} = 0, \quad \Delta w_j = \eta(1-0)x_j^{(i)} = \eta x_j^{(i)}, \quad \Delta b = \eta(1-0) = \eta$

(4) $y^{(i)} = 0, \quad \hat{y}^{(i)} = 1, \quad \Delta w_j = \eta(0-1)x_j^{(i)} = -\eta x_j^{(i)}, \quad \Delta b = \eta(0-1) = -\eta$

To get a better understanding of the feature value as a multiplicative factor, $x_j^{(i)}$, let's go through another simple example, where:

$$y^{(i)} = 1, \quad \hat{y}^{(i)} = 0, \quad \eta = 1$$

Let's assume that $x_j^{(i)} = 1.5$ and we misclassify this example as *class 0*. In this case, we would increase the corresponding weight by 2.5 in total so that the net input, $z = x_j^{(i)} \times w_j + b$, would be more positive the next time we encounter this example, and thus be more likely to be above the threshold of the unit step function to classify the example as *class 1*:

$$\Delta w_j = (1-0)1.5 = 1.5, \quad \Delta b = (1-0) = 1$$

The weight update, Δw_j, is proportional to the value of $x_j^{(i)}$. For instance, if we have another example, $x_j^{(i)} = 2$, that is incorrectly classified as *class 0*, we will push the decision boundary by an even larger extent to classify this example correctly the next time:

$$\Delta w_j = (1-0)2 = 2, \quad \Delta b = (1-0) = 1$$

It is important to note that the convergence of the perceptron is only guaranteed if the two classes are linearly separable, which means that the two classes can be perfectly separated by a linear decision boundary. (Interested readers can find the convergence proof in my lecture notes: `https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf`). *Figure 2.3* shows visual examples of linearly separable and linearly inseparable scenarios:

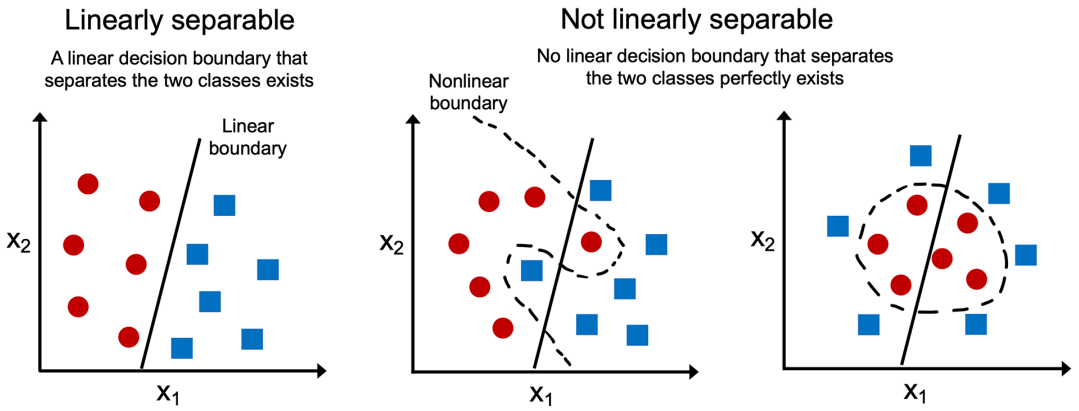


Figure 2.3: Examples of linearly and nonlinearly separable classes

If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (**epochs**) and/or a threshold for the number of tolerated misclassifications—the perceptron would never stop updating the weights otherwise. Later in this chapter, we will cover the Adaline algorithm that produces linear decision boundaries and converges even if the classes are not perfectly linearly separable. In *Chapter 3*, we will learn about algorithms that can produce nonlinear decision boundaries.

Downloading the example code

If you bought this book directly from Packt, you can download the example code files from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can download all code examples and datasets directly from `https://github.com/rasbt/machine-learning-book`.

Now, before we jump into the implementation in the next section, what you just learned can be summarized in a simple diagram that illustrates the general concept of the perceptron:

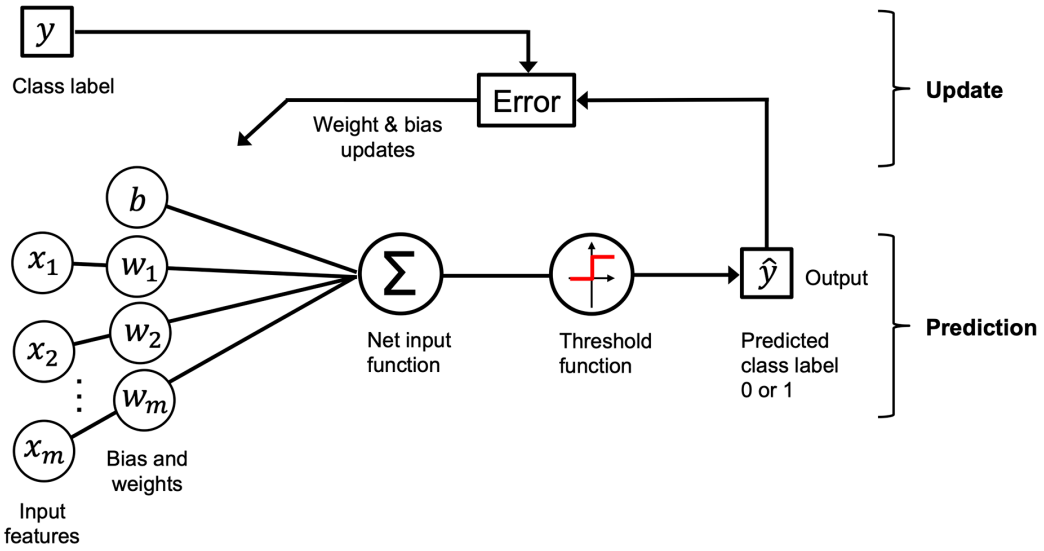


Figure 2.4: Weights and bias of the model are updated based on the error function

The preceding diagram illustrates how the perceptron receives the inputs of an example (x) and combines them with the bias unit (b) and weights (w) to compute the net input. The net input is then passed on to the threshold function, which generates a binary output of 0 or 1—the predicted class label of the example. During the learning phase, this output is used to calculate the error of the prediction and update the weights and bias unit.

Implementing a perceptron learning algorithm in Python

In the previous section, we learned how Rosenblatt's perceptron rule works; let's now implement it in Python and apply it to the Iris dataset that we introduced in *Chapter 1*, *Giving Computers the Ability to Learn from Data*.

An object-oriented perceptron API

We will take an object-oriented approach to defining the perceptron interface as a Python class, which will allow us to initialize new `Perceptron` objects that can learn from data via a `fit` method and make predictions via a separate `predict` method. As a convention, we append an underscore (_) to attributes that are not created upon the initialization of the object, but we do this by calling the object's other methods, for example, `self.w_`.

Additional resources for Python's scientific computing stack

If you are not yet familiar with Python's scientific libraries or need a refresher, please see the following resources:

- **NumPy**: `https://sebastianraschka.com/blog/2020/numpy-intro.html`
- **pandas**: `https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html`
- **Matplotlib**: `https://matplotlib.org/stable/tutorials/introductory/usage.html`

The following is the implementation of a perceptron in Python:

```
import numpy as np
class Perceptron:
    """Perceptron classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    random_state : int
      Random number generator seed for random weight
      initialization.

    Attributes
    -----------
    w_ : 1d-array
      Weights after fitting.
    b_ : Scalar
      Bias unit after fitting.

    errors_ : list
      Number of misclassifications (updates) in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```

```
    def fit(self, X, y):
        """Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
          Training vectors, where n_examples is the number of
          examples and n_features is the number of features.
        y : array-like, shape = [n_examples]
          Target values.

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=X.shape[1])
        self.b_ = np.float_(0.)
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.eta * (target - self.predict(xi))
                self.w_ += update * xi
                self.b_ += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.net_input(X) >= 0.0, 1, 0)
```

Using this perceptron implementation, we can now initialize new `Perceptron` objects with a given learning rate, `eta` (η), and the number of epochs, `n_iter` (passes over the training dataset).

Via the `fit` method, we initialize the bias `self.b_` to an initial value 0 and the weights in `self.w_` to a vector, $\mathbb{R}^m$, where m stands for the number of dimensions (features) in the dataset.

Notice that the initial weight vector contains small random numbers drawn from a normal distribution with a standard deviation of 0.01 via `rgen.normal(loc=0.0, scale=0.01, size=1 + X.shape[1])`, where `rgen` is a NumPy random number generator that we seeded with a user-specified random seed so that we can reproduce previous results if desired.

Technically, we could initialize the weights to zero (in fact, this is done in the original perceptron algorithm). However, if we did that, then the learning rate η (`eta`) would have no effect on the decision boundary. If all the weights are initialized to zero, the learning rate parameter, `eta`, affects only the scale of the weight vector, not the direction. If you are familiar with trigonometry, consider a vector, $v1$ =[1 2 3], where the angle between $v1$ and a vector, $v2 = 0.5 \times v1$, would be exactly zero, as demonstrated by the following code snippet:

```
>>> v1 = np.array([1, 2, 3])
>>> v2 = 0.5 * v1
>>> np.arccos(v1.dot(v2) / (np.linalg.norm(v1) *
...           np.linalg.norm(v2)))
0.0
```

Here, `np.arccos` is the trigonometric inverse cosine, and `np.linalg.norm` is a function that computes the length of a vector. (Our decision to draw the random numbers from a random normal distribution—for example, instead of from a uniform distribution—and to use a standard deviation of `0.01` was arbitrary; remember, we are just interested in small random values to avoid the properties of all-zero vectors, as discussed earlier.)

As an optional exercise after reading this chapter, you can change `self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])` to `self.w_ = np.zeros(X.shape[1])` and run the perceptron training code presented in the next section with different values for `eta`. You will observe that the decision boundary does not change.

NumPy array indexing

NumPy indexing for one-dimensional arrays works similarly to Python lists using the square-bracket (`[]`) notation. For two-dimensional arrays, the first indexer refers to the row number and the second indexer to the column number. For example, we would use `X[2, 3]` to select the third row and fourth column of a two-dimensional array, `X`.

After the weights have been initialized, the `fit` method loops over all individual examples in the training dataset and updates the weights according to the perceptron learning rule that we discussed in the previous section.

The class labels are predicted by the `predict` method, which is called in the `fit` method during training to get the class label for the weight update; but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the `self.errors_` list so that we can later analyze how well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product, $\boldsymbol{w}^T\boldsymbol{x} + b$.

Vectorization: Replacing for loops with vectorized code

Instead of using NumPy to calculate the vector dot product between two arrays, `a` and `b`, via `a.dot(b)` or `np.dot(a, b)`, we could also perform the calculation in pure Python via `sum([i * j for i, j in zip(a, b)])`. However, the advantage of using NumPy over classic Python `for` loop structures is that its arithmetic operations are vectorized. Vectorization means that an elemental arithmetic operation is automatically applied to all elements in an array. By formulating our arithmetic operations as a sequence of instructions on an array, rather than performing a set of operations for each element at a time, we can make better use of our modern **central processing unit** (**CPU**) architectures with **single instruction, multiple data** (**SIMD**) support. Furthermore, NumPy uses highly optimized linear algebra libraries, such as **Basic Linear Algebra Subprograms** (**BLAS**) and **Linear Algebra Package** (**LAPACK**), that have been written in C or Fortran. Lastly, NumPy also allows us to write our code in a more compact and intuitive way using the basics of linear algebra, such as vector and matrix dot products.

Training a perceptron model on the Iris dataset

To test our perceptron implementation, we will restrict the following analyses and examples in the remainder of this chapter to two feature variables (dimensions). Although the perceptron rule is not restricted to two dimensions, considering only two features, sepal length and petal length, will allow us to visualize the decision regions of the trained model in a scatterplot for learning purposes.

Note that we will also only consider two flower classes, setosa and versicolor, from the Iris dataset for practical reasons—remember, the perceptron is a binary classifier. However, the perceptron algorithm can be extended to multi-class classification—for example, the **one-versus-all** (**OvA**) technique.

The OvA method for multi-class classification

OvA, which is sometimes also called **one-versus-rest** (**OvR**), is a technique that allows us to extend any binary classifier to multi-class problems. Using OvA, we can train one classifier per class, where the particular class is treated as the positive class and the examples from all other classes are considered negative classes. If we were to classify a new, unlabeled data instance, we would use our *n* classifiers, where *n* is the number of class labels, and assign the class label with the highest confidence to the particular instance we want to classify. In the case of the perceptron, we would use OvA to choose the class label that is associated with the largest absolute net input value.

First, we will use the `pandas` library to load the Iris dataset directly from the *UCI Machine Learning Repository* into a `DataFrame` object and print the last five lines via the `tail` method to check that the data was loaded correctly:

```
>>> import os
>>> import pandas as pd
>>> s = 'https://archive.ics.uci.edu/ml/'\
...     'machine-learning-databases/iris/iris.data'
>>> print('From URL:', s)
From URL: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.
data
>>> df = pd.read_csv(s,
...                  header=None,
...                  encoding='utf-8')
>>> df.tail()
```

After executing the previous code, we should see the following output, which shows the last five lines of the Iris dataset:

	0	1	2	3	4
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

Figure 2.5: The last five lines of the Iris dataset

Loading the Iris dataset

You can find a copy of the Iris dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or if the UCI server at `https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data` is temporarily unavailable. For instance, to load the Iris dataset from a local directory, you can replace this line,

```
df = pd.read_csv(
  'https://archive.ics.uci.edu/ml/'
  'machine-learning-databases/iris/iris.data',
  header=None, encoding='utf-8')
```

with the following one:

```
df = pd.read_csv(
  'your/local/path/to/iris.data',
  header=None, encoding='utf-8')
```

Next, we extract the first 100 class labels that correspond to the 50 Iris-setosa and 50 Iris-versicolor flowers and convert the class labels into the two integer class labels, `1` (versicolor) and `0` (setosa), that we assign to a vector, `y`, where the `values` method of a pandas `DataFrame` yields the corresponding NumPy representation.

Similarly, we extract the first feature column (sepal length) and the third feature column (petal length) of those 100 training examples and assign them to a feature matrix, `X`, which we can visualize via a two-dimensional scatterplot:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> # select setosa and versicolor
>>> y = df.iloc[0:100, 4].values
>>> y = np.where(y == 'Iris-setosa', 0, 1)
>>> # extract sepal length and petal length
>>> X = df.iloc[0:100, [0, 2]].values
>>> # plot data
>>> plt.scatter(X[:50, 0], X[:50, 1],
...             color='red', marker='o', label='Setosa')
>>> plt.scatter(X[50:100, 0], X[50:100, 1],
...             color='blue', marker='s', label='Versicolor')
>>> plt.xlabel('Sepal length [cm]')
>>> plt.ylabel('Petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should see the following scatterplot:

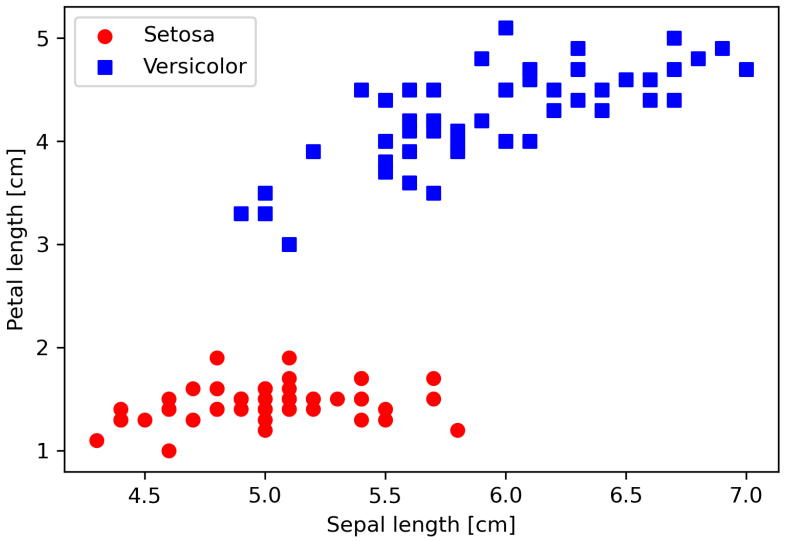


Figure 2.6: Scatterplot of setosa and versicolor flowers by sepal and petal length

Figure 2.6 shows the distribution of flower examples in the Iris dataset along the two feature axes: petal length and sepal length (measured in centimeters). In this two-dimensional feature subspace, we can see that a linear decision boundary should be sufficient to separate setosa from versicolor flowers. Thus, a linear classifier such as the perceptron should be able to classify the flowers in this dataset perfectly.

Now, it's time to train our perceptron algorithm on the Iris data subset that we just extracted. Also, we will plot the misclassification error for each epoch to check whether the algorithm converged and found a decision boundary that separates the two Iris flower classes:

```
>>> ppn = Perceptron(eta=0.1, n_iter=10)
>>> ppn.fit(X, y)
>>> plt.plot(range(1, len(ppn.errors_) + 1),
...          ppn.errors_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Number of updates')
>>> plt.show()
```

Note that the number of misclassification errors and the number of updates is the same, since the perceptron weights and bias are updated each time it misclassifies an example. After executing the preceding code, we should see the plot of the misclassification errors versus the number of epochs, as shown in *Figure 2.7*:

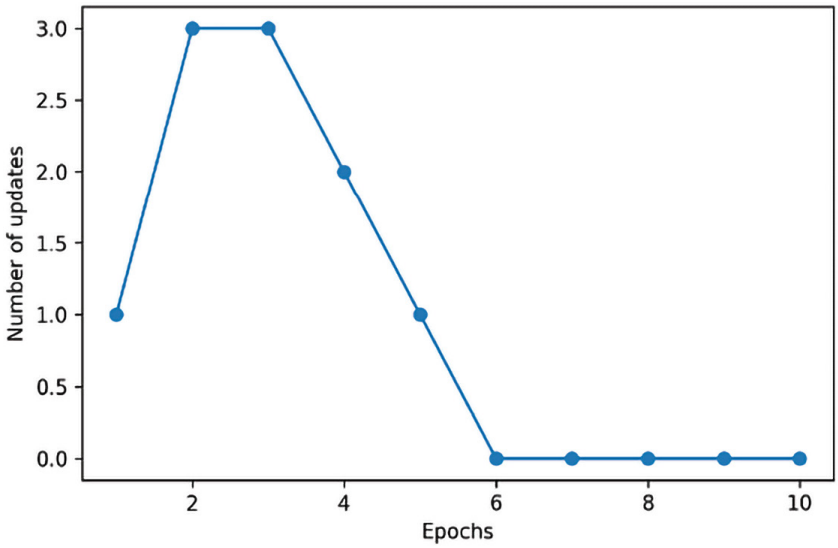


Figure 2.7: A plot of the misclassification errors against the number of epochs

As we can see in *Figure 2.7*, our perceptron converged after the sixth epoch and should now be able to classify the training examples perfectly. Let's implement a small convenience function to visualize the decision boundaries for two-dimensional datasets:

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, resolution=0.02):
    # setup marker generator and color map
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())
```

```
    # plot class examples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')
```

First, we define a number of `colors` and `markers` and create a colormap from the list of colors via `ListedColormap`. Then, we determine the minimum and maximum values for the two features and use those feature vectors to create a pair of grid arrays, `xx1` and `xx2`, via the NumPy `meshgrid` function. Since we trained our perceptron classifier on two feature dimensions, we need to flatten the grid arrays and create a matrix that has the same number of columns as the Iris training subset so that we can use the `predict` method to predict the class labels, `lab`, of the corresponding grid points.

After reshaping the predicted class labels, `lab`, into a grid with the same dimensions as `xx1` and `xx2`, we can now draw a contour plot via Matplotlib's `contourf` function, which maps the different decision regions to different colors for each predicted class in the grid array:

```
>>> plot_decision_regions(X, y, classifier=ppn)
>>> plt.xlabel('Sepal length [cm]')
>>> plt.ylabel('Petal length [cm]')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

After executing the preceding code example, we should now see a plot of the decision regions, as shown in *Figure 2.8*:

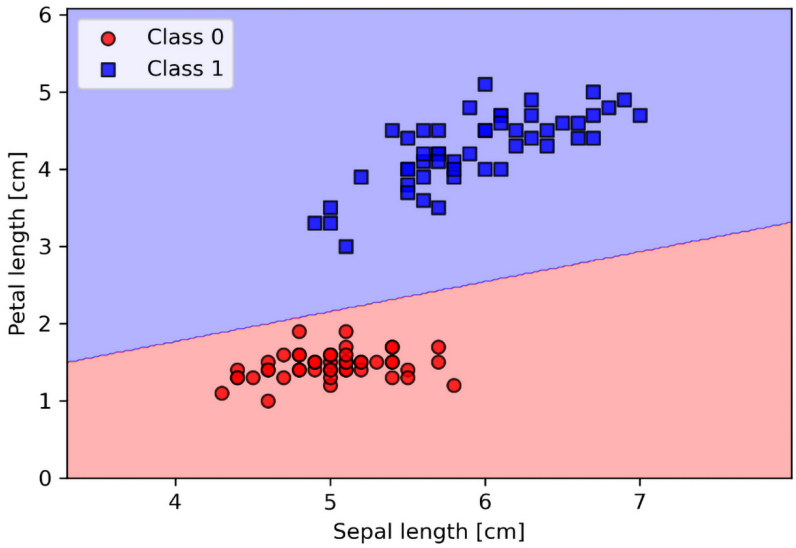


Figure 2.8: A plot of the perceptron's decision regions

As we can see in the plot, the perceptron learned a decision boundary that can classify all flower examples in the Iris training subset perfectly.

Perceptron convergence

Although the perceptron classified the two Iris flower classes perfectly, convergence is one of the biggest problems of the perceptron. Rosenblatt proved mathematically that the perceptron learning rule converges if the two classes can be separated by a linear hyperplane. However, if the classes cannot be separated perfectly by such a linear decision boundary, the weights will never stop updating unless we set a maximum number of epochs. Interested readers can find a summary of the proof in my lecture notes at `https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L03_perceptron_slides.pdf`.

Adaptive linear neurons and the convergence of learning

In this section, we will take a look at another type of single-layer **neural network** (**NN**): **ADAptive LInear NEuron** (**Adaline**). Adaline was published by Bernard Widrow and his doctoral student Tedd Hoff only a few years after Rosenblatt's perceptron algorithm, and it can be considered an improvement on the latter (*An Adaptive "Adaline" Neuron Using Chemical "Memistors"*, *Technical Report Number 1553-2* by *B. Widrow and colleagues*, *Stanford Electron Labs*, Stanford, CA, *October* 1960).

The Adaline algorithm is particularly interesting because it illustrates the key concepts of defining and minimizing continuous loss functions. This lays the groundwork for understanding other machine learning algorithms for classification, such as logistic regression, support vector machines, and multilayer neural networks, as well as linear regression models, which we will discuss in future chapters.

The key difference between the Adaline rule (also known as the **Widrow-Hoff rule**) and Rosenblatt's perceptron is that the weights are updated based on a linear activation function rather than a unit step function like in the perceptron. In Adaline, this linear activation function, $\sigma(z)$, is simply the identity function of the net input, so that $\sigma(z) = z$.

While the linear activation function is used for learning the weights, we still use a threshold function to make the final prediction, which is similar to the unit step function that we covered earlier.

The main differences between the perceptron and Adaline algorithm are highlighted in *Figure 2.9*:

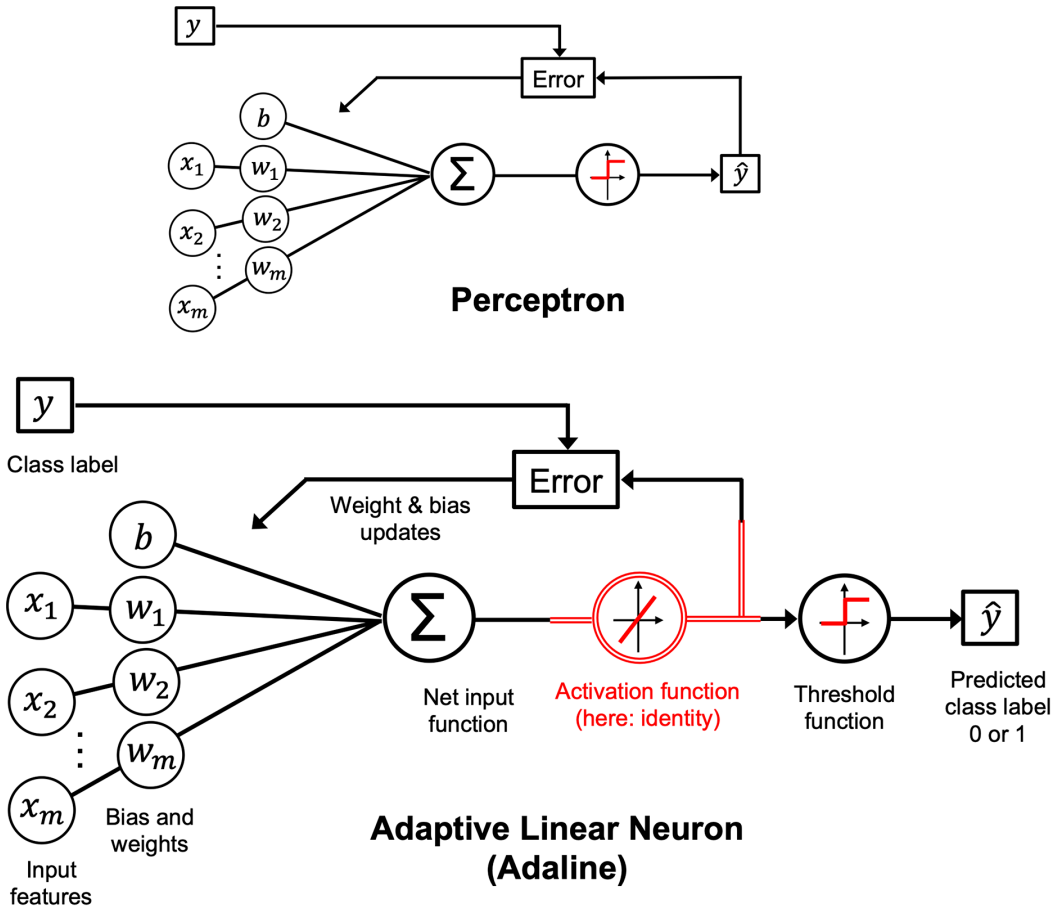


Figure 2.9: A comparison between a perceptron and the Adaline algorithm

As *Figure 2.9* indicates, the Adaline algorithm compares the true class labels with the linear activation function's continuous valued output to compute the model error and update the weights. In contrast, the perceptron compares the true class labels to the predicted class labels.

Minimizing loss functions with gradient descent

One of the key ingredients of supervised machine learning algorithms is a defined **objective function** that is to be optimized during the learning process. This objective function is often a loss or cost function that we want to minimize. In the case of Adaline, we can define the loss function, L, to learn the model parameters as the **mean squared error** (**MSE**) between the calculated outcome and the true class label:

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)^2$$

The main advantage of this continuous linear activation function, in contrast to the unit step function, is that the loss function becomes differentiable. Another nice property of this loss function is that it is convex; thus, we can use a very simple yet powerful optimization algorithm called **gradient descent** to find the weights that minimize our loss function to classify the examples in the Iris dataset.

As illustrated in *Figure 2.10*, we can describe the main idea behind gradient descent as *climbing down a hill* until a local or global loss minimum is reached. In each iteration, we take a step in the opposite direction of the gradient, where the step size is determined by the value of the learning rate, as well as the slope of the gradient (for simplicity, the following figure visualizes this only for a single weight, w):

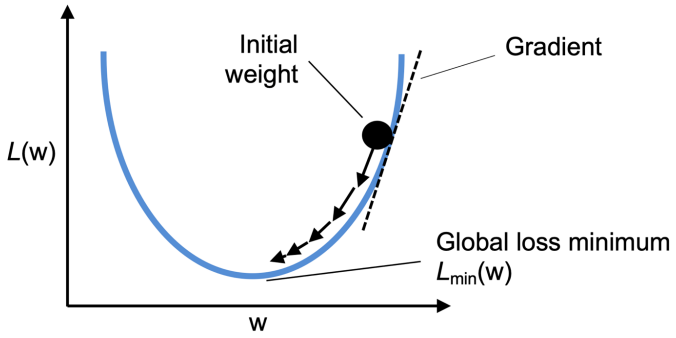


Figure 2.10: How gradient descent works

Using gradient descent, we can now update the model parameters by taking a step in the opposite direction of the gradient, $\nabla L(\boldsymbol{w}, b)$, of our loss function, $L(\boldsymbol{w}, b)$:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w}, \quad b := b + \Delta b$$

The parameter changes, $\Delta \boldsymbol{w}$ and Δb, are defined as the negative gradient multiplied by the learning rate, η:

$$\Delta \boldsymbol{w} = -\eta \nabla_w L(\boldsymbol{w}, b), \quad \Delta b = -\eta \nabla_b L(\boldsymbol{w}, b)$$

To compute the gradient of the loss function, we need to compute the partial derivative of the loss function with respect to each weight, w_j:

$$\frac{\partial L}{\partial w_j} = -\frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) x_j^{(i)}$$

Similarly, we compute the partial derivative of the loss with respect to the bias as:

$$\frac{\partial L}{\partial b} = -\frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right)$$

Please note that the 2 in the numerator above is merely a constant scaling factor, and we could omit it without affecting the algorithm. Removing the scaling factor has the same effect as changing the learning rate by a factor of 2. The following information box explains where this scaling factor originates.

So we can write the weight update as:

$$\Delta w_j = -\eta \frac{\partial L}{\partial w_j} \text{ and } \Delta b = -\eta \frac{\partial L}{\partial b}$$

Since we update all parameters simultaneously, our Adaline learning rule becomes:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w}, \quad b := b + \Delta b$$

The mean squared error derivative

If you are familiar with calculus, the partial derivative of the MSE loss function with respect to the *j*th weight can be obtained as follows:

$$\begin{aligned}
\frac{\partial L}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right)^2 = \frac{1}{n} \frac{\partial}{\partial w_j} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right)^2 \\
&= \frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) \\
&= \frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) \frac{\partial}{\partial w_j} \left(y^{(i)} - \sum_j \left(w_j x_j^{(i)} + b \right) \right) \\
&= \frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) \left(-x_j^{(i)} \right) = -\frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right) x_j^{(i)}
\end{aligned}$$

The same approach can be used to find partial derivative $\frac{\partial L}{\partial b}$ except that $\frac{\partial}{\partial b}\left(y^{(i)} - \sum_i \left(w_j^{(i)} x_j^{(i)} + b \right) \right)$ is equal to –1 and thus the last step simplifies to $-\frac{2}{n} \sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right) \right)$.

Although the Adaline learning rule looks identical to the perceptron rule, we should note that $\sigma(z^{(i)})$ with $z^{(i)} = \boldsymbol{w}^T \boldsymbol{x}^{(i)} + b$ is a real number and not an integer class label. Furthermore, the weight update is calculated based on all examples in the training dataset (instead of updating the parameters incrementally after each training example), which is why this approach is also referred to as **batch gradient descent**. To be more explicit and avoid confusion when talking about related concepts later in this chapter and this book, we will refer to this process as **full batch gradient descent**.

Implementing Adaline in Python

Since the perceptron rule and Adaline are very similar, we will take the perceptron implementation that we defined earlier and change the `fit` method so that the weight and bias parameters are now updated by minimizing the loss function via gradient descent:

```
class AdalineGD:
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    random_state : int
        Random number generator seed for random weight initialization.

    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    b_ : Scalar
        Bias unit after fitting.
    losses_ : list
      Mean squared error loss function values in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.
```

```
        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
            Training vectors, where n_examples
            is the number of examples and
            n_features is the number of features.
        y : array-like, shape = [n_examples]
            Target values.

        Returns
        -------
        self : object

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01,
                              size=X.shape[1])
        self.b_ = np.float_(0.)
        self.losses_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (errors**2).mean()
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def activation(self, X):
        """Compute linear activation"""
        return X

    def predict(self, X):
        """Return class label after unit step"""
```

```
        return np.where(self.activation(self.net_input(X))
                        >= 0.5, 1, 0)
```

Instead of updating the weights after evaluating each individual training example, as in the perceptron, we calculate the gradient based on the whole training dataset. For the bias unit, this is done via `self.eta * 2.0 * errors.mean()`, where `errors` is an array containing the partial derivative values $\frac{\partial}{\partial b}$. Similarly, we update the weights. However note that the weight updates via the partial derivatives $\frac{\partial L}{\partial w_j}$ involve the feature values x_j, which we can compute by multiplying `errors` with each feature value for each weight:

```
            for w_j in range(self.w_.shape[0]):
                self.w_[w_j] += self.eta *
                    (2.0 * (X[:, w_j]*errors)).mean()
```

To implement the weight update more efficiently without using a `for` loop, we can use a matrix-vector multiplication between our feature matrix and the error vector instead:

```
self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
```

Please note that the `activation` method has no effect on the code since it is simply an identity function. Here, we added the activation function (computed via the `activation` method) to illustrate the general concept with regard to how information flows through a single-layer NN: features from the input data, net input, activation, and output.

In the next chapter, we will learn about a logistic regression classifier that uses a non-identity, nonlinear activation function. We will see that a logistic regression model is closely related to Adaline, with the only difference being its activation and loss function.

Now, similar to the previous perceptron implementation, we collect the loss values in a `self.losses_` list to check whether the algorithm converged after training.

Matrix multiplication

Performing a matrix multiplication is similar to calculating a vector dot-product where each row in the matrix is treated as a single row vector. This vectorized approach represents a more compact notation and results in a more efficient computation using NumPy. For example:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 2 \times 8 + 3 \times 9 \\ 4 \times 7 + 5 \times 8 + 6 \times 9 \end{bmatrix} = \begin{bmatrix} 50 \\ 122 \end{bmatrix}$$

Please note that in the preceding equation, we are multiplying a matrix with a vector, which is mathematically not defined. However, remember that we use the convention that this preceding vector is regarded as a 3×1 matrix.

In practice, it often requires some experimentation to find a good learning rate, η, for optimal convergence. So, let's choose two different learning rates, $\eta = 0.1$ and $\eta = 0.0001$, to start with and plot the loss functions versus the number of epochs to see how well the Adaline implementation learns from the training data.

Hyperparameters

The learning rate, η (`eta`), as well as the number of epochs (`n_iter`), are the so-called hyperparameters (or tuning parameters) of the perceptron and Adaline learning algorithms. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will take a look at different techniques to automatically find the values of different hyperparameters that yield optimal performance of the classification model.

Let's now plot the loss against the number of epochs for the two different learning rates:

```
>>> fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(10, 4))
>>> ada1 = AdalineGD(n_iter=15, eta=0.1).fit(X, y)
>>> ax[0].plot(range(1, len(ada1.losses_) + 1),
...            np.log10(ada1.losses_), marker='o')
>>> ax[0].set_xlabel('Epochs')
>>> ax[0].set_ylabel('log(Mean squared error)')
>>> ax[0].set_title('Adaline - Learning rate 0.1')
>>> ada2 = AdalineGD(n_iter=15, eta=0.0001).fit(X, y)
>>> ax[1].plot(range(1, len(ada2.losses_) + 1),
...            ada2.losses_, marker='o')
>>> ax[1].set_xlabel('Epochs')
>>> ax[1].set_ylabel('Mean squared error')
>>> ax[1].set_title('Adaline - Learning rate 0.0001')
>>> plt.show()
```

As we can see in the resulting loss function plots, we encountered two different types of problems. The left chart shows what could happen if we choose a learning rate that is too large. Instead of minimizing the loss function, the MSE becomes larger in every epoch, because we *overshoot* the global minimum. On the other hand, we can see that the loss decreases on the right plot, but the chosen learning rate, $\eta = 0.0001$, is so small that the algorithm would require a very large number of epochs to converge to the global loss minimum:

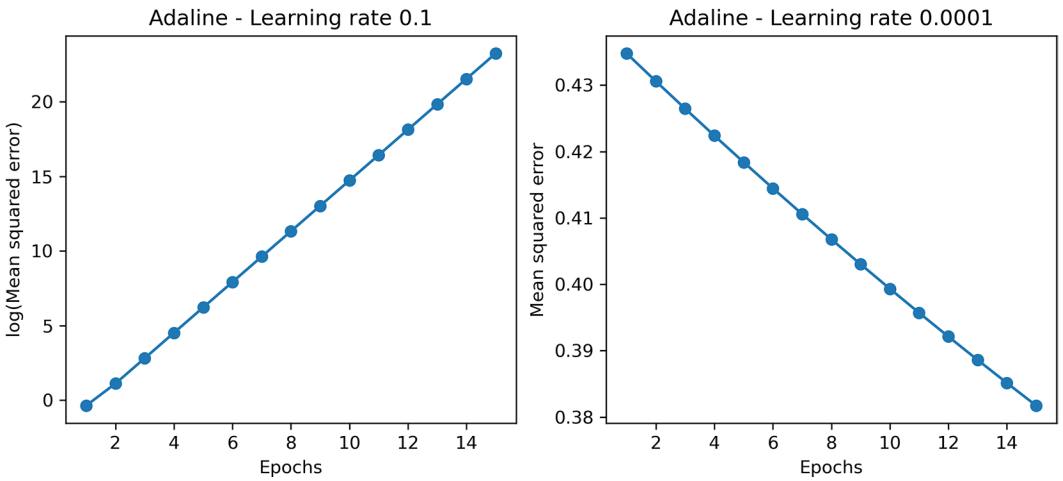


Figure 2.11: Error plots for suboptimal learning rates

Figure 2.12 illustrates what might happen if we change the value of a particular weight parameter to minimize the loss function, *L*. The left subfigure illustrates the case of a well-chosen learning rate, where the loss decreases gradually, moving in the direction of the global minimum.

The subfigure on the right, however, illustrates what happens if we choose a learning rate that is too large—we overshoot the global minimum:

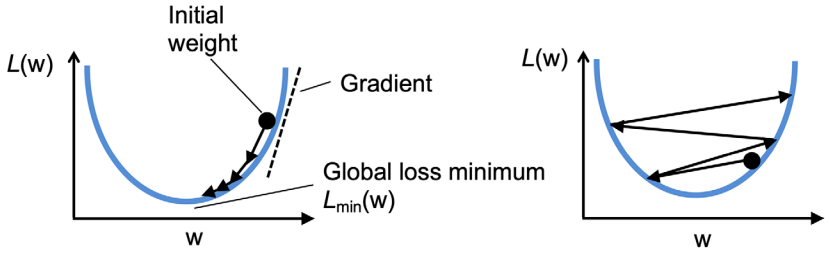


Figure 2.12: A comparison of a well-chosen learning rate and a learning rate that is too large

Improving gradient descent through feature scaling

Many machine learning algorithms that we will encounter throughout this book require some sort of feature scaling for optimal performance, which we will discuss in more detail in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, and *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*.

Gradient descent is one of the many algorithms that benefit from feature scaling. In this section, we will use a feature scaling method called **standardization**. This normalization procedure helps gradient descent learning to converge more quickly; however, it does not make the original dataset normally distributed. Standardization shifts the mean of each feature so that it is centered at zero and each feature has a standard deviation of 1 (unit variance). For instance, to standardize the *j*th feature, we can simply subtract the sample mean, μ_j, from every training example and divide it by its standard deviation, σ_j:

$$x'_j = \frac{x_j - \mu_j}{\sigma_j}$$

Here, x_j is a vector consisting of the *j*th feature values of all training examples, n, and this standardization technique is applied to each feature, j, in our dataset.

One of the reasons why standardization helps with gradient descent learning is that it is easier to find a learning rate that works well for all weights (and the bias). If the features are on vastly different scales, a learning rate that works well for updating one weight might be too large or too small to update the other weight equally well. Overall, using standardized features can stabilize the training such that the optimizer has to go through fewer steps to find a good or optimal solution (the global loss minimum). *Figure 2.13* illustrates possible gradient updates with unscaled features (left) and standardized features (right), where the concentric circles represent the loss surface as a function of two model weights in a two-dimensional classification problem:

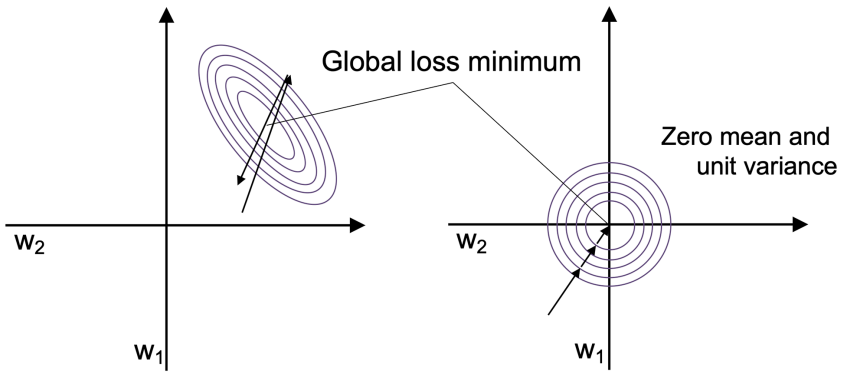


Figure 2.13: A comparison of unscaled and standardized features on gradient updates

Standardization can easily be achieved by using the built-in NumPy methods `mean` and `std`:

```
>>> X_std = np.copy(X)
>>> X_std[:,0] = (X[:,0] - X[:,0].mean()) / X[:,0].std()
>>> X_std[:,1] = (X[:,1] - X[:,1].mean()) / X[:,1].std()
```

After standardization, we will train Adaline again and see that it now converges after a small number of epochs using a learning rate of $\eta = 0.5$:

```
>>> ada_gd = AdalineGD(n_iter=20, eta=0.5)
>>> ada_gd.fit(X_std, y)
```

```
>>> plot_decision_regions(X_std, y, classifier=ada_gd)
>>> plt.title('Adaline - Gradient descent')
>>> plt.xlabel('Sepal length [standardized]')
>>> plt.ylabel('Petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_gd.losses_) + 1),
...          ada_gd.losses_, marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Mean squared error')
>>> plt.tight_layout()
>>> plt.show()
```

After executing this code, we should see a figure of the decision regions, as well as a plot of the declining loss, as shown in *Figure 2.14*:

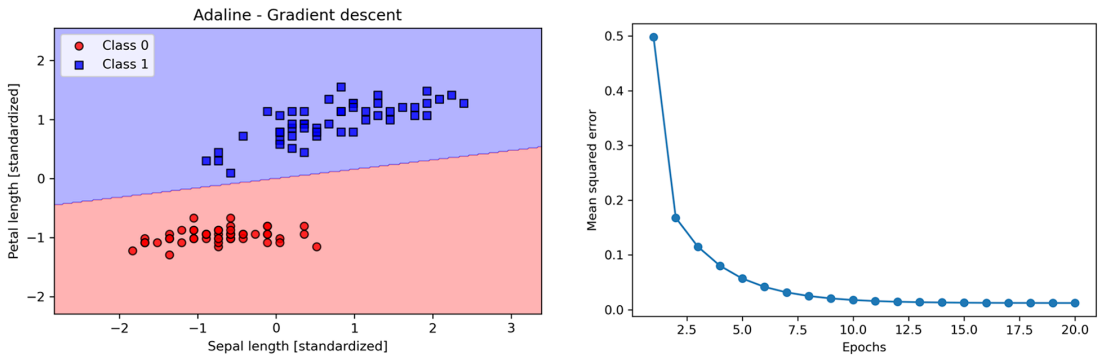


Figure 2.14: Plots of Adaline's decision regions and MSE by number of epochs

As we can see in the plots, Adaline has now converged after training on the standardized features. However, note that the MSE remains non-zero even though all flower examples were classified correctly.

Large-scale machine learning and stochastic gradient descent

In the previous section, we learned how to minimize a loss function by taking a step in the opposite direction of the loss gradient that is calculated from the whole training dataset; this is why this approach is sometimes also referred to as full batch gradient descent. Now imagine that we have a very large dataset with millions of data points, which is not uncommon in many machine learning applications. Running full batch gradient descent can be computationally quite costly in such scenarios, since we need to reevaluate the whole training dataset each time we take one step toward the global minimum.

A popular alternative to the batch gradient descent algorithm is **stochastic gradient descent** (**SGD**), which is sometimes also called iterative or online gradient descent. Instead of updating the weights based on the sum of the accumulated errors over all training examples, $\boldsymbol{x}^{(i)}$:

$$\Delta w_j = \frac{2\eta}{n}\sum_i \left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)x_j^{(i)}$$

we update the parameters incrementally for each training example, for instance:

$$\Delta w_j = \eta\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)x_j^{(i)}, \quad \Delta b = \eta\left(y^{(i)} - \sigma\left(z^{(i)}\right)\right)$$

Although SGD can be considered as an approximation of gradient descent, it typically reaches convergence much faster because of the more frequent weight updates. Since each gradient is calculated based on a single training example, the error surface is noisier than in gradient descent, which can also have the advantage that SGD can escape shallow local minima more readily if we are working with nonlinear loss functions, as we will see later in *Chapter 11*, *Implementing a Multilayer Artificial Neural Network from Scratch*. To obtain satisfying results via SGD, it is important to present training data in a random order; also, we want to shuffle the training dataset for every epoch to prevent cycles.

Adjusting the learning rate during training

In SGD implementations, the fixed learning rate, η, is often replaced by an adaptive learning rate that decreases over time, for example:

$$\frac{c_1}{[\text{number of iterations}] + c_2}$$

where c_1 and c_2 are constants. Note that SGD does not reach the global loss minimum but an area very close to it. And using an adaptive learning rate, we can achieve further annealing to the loss minimum.

Another advantage of SGD is that we can use it for **online learning**. In online learning, our model is trained on the fly as new training data arrives. This is especially useful if we are accumulating large amounts of data, for example, customer data in web applications. Using online learning, the system can immediately adapt to changes, and the training data can be discarded after updating the model if storage space is an issue.

Mini-batch gradient descent

A compromise between full batch gradient descent and SGD is so-called **mini-batch gradient descent**. Mini-batch gradient descent can be understood as applying full batch gradient descent to smaller subsets of the training data, for example, 32 training examples at a time. The advantage over full batch gradient descent is that convergence is reached faster via mini-batches because of the more frequent weight updates. Furthermore, mini-batch learning allows us to replace the `for` loop over the training examples in SGD with vectorized operations leveraging concepts from linear algebra (for example, implementing a weighted sum via a dot product), which can further improve the computational efficiency of our learning algorithm.

Since we already implemented the Adaline learning rule using gradient descent, we only need to make a few adjustments to modify the learning algorithm to update the weights via SGD. Inside the `fit` method, we will now update the weights after each training example. Furthermore, we will implement an additional `partial_fit` method, which does not reinitialize the weights, for online learning. In order to check whether our algorithm converged after training, we will calculate the loss as the average loss of the training examples in each epoch. Furthermore, we will add an option to shuffle the training data before each epoch to avoid repetitive cycles when we are optimizing the loss function; via the `random_state` parameter, we allow the specification of a random seed for reproducibility:

```
class AdalineSGD:
    """ADAptive LInear NEuron classifier.

    Parameters
    ------------
    eta : float
        Learning rate (between 0.0 and 1.0)
    n_iter : int
        Passes over the training dataset.
    shuffle : bool (default: True)
        Shuffles training data every epoch if True to prevent
        cycles.
    random_state : int
        Random number generator seed for random weight
        initialization.
```

```
    Attributes
    -----------
    w_ : 1d-array
        Weights after fitting.
    b_ : Scalar
        Bias unit after fitting.
    losses_ : list
        Mean squared error loss function value averaged over all
        training examples in each epoch.


    """
    def __init__(self, eta=0.01, n_iter=10,
                 shuffle=True, random_state=None):
        self.eta = eta
        self.n_iter = n_iter
        self.w_initialized = False
        self.shuffle = shuffle
        self.random_state = random_state

    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
            Training vectors, where n_examples is the number of
            examples and n_features is the number of features.
        y : array-like, shape = [n_examples]
            Target values.

        Returns
        -------
        self : object

        """
        self._initialize_weights(X.shape[1])
        self.losses_ = []
        for i in range(self.n_iter):
            if self.shuffle:
```

```
                X, y = self._shuffle(X, y)
            losses = []
            for xi, target in zip(X, y):
                losses.append(self._update_weights(xi, target))
            avg_loss = np.mean(losses)
            self.losses_.append(avg_loss)
        return self

    def partial_fit(self, X, y):
        """Fit training data without reinitializing the weights"""
        if not self.w_initialized:
            self._initialize_weights(X.shape[1])
        if y.ravel().shape[0] > 1:
            for xi, target in zip(X, y):
                self._update_weights(xi, target)
        else:
            self._update_weights(X, y)
        return self

    def _shuffle(self, X, y):
        """Shuffle training data"""
        r = self.rgen.permutation(len(y))
        return X[r], y[r]

    def _initialize_weights(self, m):
        """Initialize weights to small random numbers"""
        self.rgen = np.random.RandomState(self.random_state)
        self.w_ = self.rgen.normal(loc=0.0, scale=0.01,
                                   size=m)
        self.b_ = np.float_(0.)
        self.w_initialized = True

    def _update_weights(self, xi, target):
        """Apply Adaline learning rule to update the weights"""
        output = self.activation(self.net_input(xi))
        error = (target - output)
        self.w_ += self.eta * 2.0 * xi * (error)
        self.b_ += self.eta * 2.0 * error
        loss = error**2
        return loss
```

```
    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def activation(self, X):
        """Compute linear activation"""
        return X

    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(self.net_input(X))
                        >= 0.5, 1, 0)
```

The `_shuffle` method that we are now using in the `AdalineSGD` classifier works as follows: via the `permutation` function in `np.random`, we generate a random sequence of unique numbers in the range 0 to 100. Those numbers can then be used as indices to shuffle our feature matrix and class label vector.

We can then use the `fit` method to train the `AdalineSGD` classifier and use our `plot_decision_regions` to plot our training results:

```
>>> ada_sgd = AdalineSGD(n_iter=15, eta=0.01, random_state=1)
>>> ada_sgd.fit(X_std, y)
>>> plot_decision_regions(X_std, y, classifier=ada_sgd)
>>> plt.title('Adaline - Stochastic gradient descent')
>>> plt.xlabel('Sepal length [standardized]')
>>> plt.ylabel('Petal length [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
>>> plt.plot(range(1, len(ada_sgd.losses_) + 1), ada_sgd.losses_,
...          marker='o')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Average loss')
>>> plt.tight_layout()
>>> plt.show()
```

The two plots that we obtain from executing the preceding code example are shown in *Figure 2.15*:

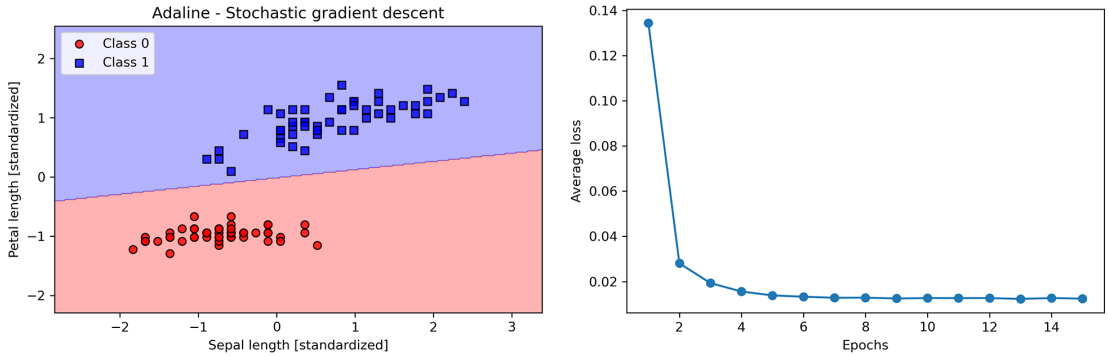


Figure 2.15: Decision regions and average loss plots after training an Adaline model using SGD

As you can see, the average loss goes down pretty quickly, and the final decision boundary after 15 epochs looks similar to the batch gradient descent Adaline. If we want to update our model, for example, in an online learning scenario with streaming data, we could simply call the `partial_fit` method on individual training examples—for instance, `ada_sgd.partial_fit(X_std[0, :], y[0])`.

Summary

In this chapter, we gained a good understanding of the basic concepts of linear classifiers for supervised learning. After we implemented a perceptron, we saw how we can train adaptive linear neurons efficiently via a vectorized implementation of gradient descent and online learning via SGD.

Now that we have seen how to implement simple classifiers in Python, we are ready to move on to the next chapter, where we will use the Python scikit-learn machine learning library to get access to more advanced and powerful machine learning classifiers, which are commonly used in academia as well as in industry.

The object-oriented approach that we used to implement the perceptron and Adaline algorithms will help with understanding the scikit-learn API, which is implemented based on the same core concepts that we used in this chapter: the `fit` and `predict` methods. Based on these core concepts, we will learn about logistic regression for modeling class probabilities and support vector machines for working with nonlinear decision boundaries. In addition, we will introduce a different class of supervised learning algorithms, tree-based algorithms, which are commonly combined into robust ensemble classifiers.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`

3

A Tour of Machine Learning Classifiers Using Scikit-Learn

In this chapter, we will take a tour of a selection of popular and powerful machine learning algorithms that are commonly used in academia as well as in industry. While learning about the differences between several supervised learning algorithms for classification, we will also develop an appreciation of their individual strengths and weaknesses. In addition, we will take our first steps with the scikit-learn library, which offers a user-friendly and consistent interface for using those algorithms efficiently and productively.

The topics that will be covered throughout this chapter are as follows:

- An introduction to robust and popular algorithms for classification, such as logistic regression, support vector machines, decision trees, and *k*-nearest neighbors
- Examples and explanations using the scikit-learn machine learning library, which provides a wide variety of machine learning algorithms via a user-friendly Python API
- Discussions about the strengths and weaknesses of classifiers with linear and nonlinear decision boundaries

Choosing a classification algorithm

Choosing an appropriate classification algorithm for a particular problem task requires practice and experience; each algorithm has its own quirks and is based on certain assumptions. To paraphrase the **no free lunch theorem** by David H. Wolpert, no single classifier works best across all possible scenarios (*The Lack of A Priori Distinctions Between Learning Algorithms*, *Wolpert, David H*, *Neural Computation 8.7* (1996): 1341-1390). In practice, it is always recommended that you compare the performance of at least a handful of different learning algorithms to select the best model for the particular problem; these may differ in the number of features or examples, the amount of noise in a dataset, and whether the classes are linearly separable.

Eventually, the performance of a classifier—computational performance as well as predictive power—depends heavily on the underlying data that is available for learning. The five main steps that are involved in training a supervised machine learning algorithm can be summarized as follows:

1. Selecting features and collecting labeled training examples
2. Choosing a performance metric
3. Choosing a learning algorithm and training a model
4. Evaluating the performance of the model
5. Changing the settings of the algorithm and tuning the model.

Since the approach of this book is to build machine learning knowledge step by step, we will mainly focus on the main concepts of the different algorithms in this chapter and revisit topics such as feature selection and preprocessing, performance metrics, and hyperparameter tuning for more detailed discussions later in the book.

First steps with scikit-learn – training a perceptron

In *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, you learned about two related learning algorithms for classification, the **perceptron** rule and **Adaline**, which we implemented in Python and NumPy by ourselves. Now we will take a look at the scikit-learn API, which, as mentioned, combines a user-friendly and consistent interface with a highly optimized implementation of several classification algorithms. The scikit-learn library offers not only a large variety of learning algorithms, but also many convenient functions to preprocess data and to fine-tune and evaluate our models. We will discuss this in more detail, together with the underlying concepts, in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, and *Chapter 5*, *Compressing Data via Dimensionality Reduction*.

To get started with the scikit-learn library, we will train a perceptron model similar to the one that we implemented in *Chapter 2*. For simplicity, we will use the already familiar **Iris dataset** throughout the following sections. Conveniently, the Iris dataset is already available via scikit-learn, since it is a simple yet popular dataset that is frequently used for testing and experimenting with algorithms. Similar to the previous chapter, we will only use two features from the Iris dataset for visualization purposes.

We will assign the petal length and petal width of the 150 flower examples to the feature matrix, `X`, and the corresponding class labels of the flower species to the vector array, `y`:

```
>>> from sklearn import datasets
>>> import numpy as np
>>> iris = datasets.load_iris()
>>> X = iris.data[:, [2, 3]]
>>> y = iris.target
```

```
>>> print('Class labels:', np.unique(y))
Class labels: [0 1 2]
```

The `np.unique(y)` function returned the three unique class labels stored in `iris.target`, and as we can see, the Iris flower class names, `Iris-setosa`, `Iris-versicolor`, and `Iris-virginica`, are already stored as integers (here: `0`, `1`, `2`). Although many scikit-learn functions and class methods also work with class labels in string format, using integer labels is a recommended approach to avoid technical glitches and improve computational performance due to a smaller memory footprint; furthermore, encoding class labels as integers is a common convention among most machine learning libraries.

To evaluate how well a trained model performs on unseen data, we will further split the dataset into separate training and test datasets. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will discuss the best practices around model evaluation in more detail. Using the `train_test_split` function from scikit-learn's `model_selection` module, we randomly split the `X` and `y` arrays into 30 percent test data (45 examples) and 70 percent training data (105 examples):

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(
...     X, y, test_size=0.3, random_state=1, stratify=y
... )
```

Note that the `train_test_split` function already shuffles the training datasets internally before splitting; otherwise, all examples from class `0` and class `1` would have ended up in the training datasets, and the test dataset would consist of 45 examples from class `2`. Via the `random_state` parameter, we provided a fixed random seed (`random_state=1`) for the internal pseudo-random number generator that is used for shuffling the datasets prior to splitting. Using such a fixed `random_state` ensures that our results are reproducible.

Lastly, we took advantage of the built-in support for stratification via `stratify=y`. In this context, stratification means that the `train_test_split` method returns training and test subsets that have the same proportions of class labels as the input dataset. We can use NumPy's `bincount` function, which counts the number of occurrences of each value in an array, to verify that this is indeed the case:

```
>>> print('Labels counts in y:', np.bincount(y))
Labels counts in y: [50 50 50]
>>> print('Labels counts in y_train:', np.bincount(y_train))
Labels counts in y_train: [35 35 35]
>>> print('Labels counts in y_test:', np.bincount(y_test))
Labels counts in y_test: [15 15 15]
```

Many machine learning and optimization algorithms also require feature scaling for optimal performance, as we saw in the **gradient descent** example in *Chapter 2*. Here, we will standardize the features using the `StandardScaler` class from scikit-learn's `preprocessing` module:

```
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> sc.fit(X_train)
>>> X_train_std = sc.transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

Using the preceding code, we loaded the `StandardScaler` class from the `preprocessing` module and initialized a new `StandardScaler` object that we assigned to the `sc` variable. Using the `fit` method, `StandardScaler` estimated the parameters, μ (sample mean) and σ (standard deviation), for each feature dimension from the training data. By calling the `transform` method, we then standardized the training data using those estimated parameters, μ and σ. Note that we used the same scaling parameters to standardize the test dataset so that both the values in the training and test dataset are comparable with one another.

Having standardized the training data, we can now train a perceptron model. Most algorithms in scikit-learn already support multiclass classification by default via the **one-versus-rest** (**OvR**) method, which allows us to feed the three flower classes to the perceptron all at once. The code is as follows:

```
>>> from sklearn.linear_model import Perceptron
>>> ppn = Perceptron(eta0=0.1, random_state=1)
>>> ppn.fit(X_train_std, y_train)
```

The scikit-learn interface will remind you of our perceptron implementation in *Chapter 2*. After loading the `Perceptron` class from the `linear_model` module, we initialized a new `Perceptron` object and trained the model via the `fit` method. Here, the model parameter, `eta0`, is equivalent to the learning rate, `eta`, that we used in our own perceptron implementation.

As you will remember from *Chapter 2*, finding an appropriate learning rate requires some experimentation. If the learning rate is too large, the algorithm will overshoot the global loss minimum. If the learning rate is too small, the algorithm will require more epochs until convergence, which can make the learning slow—especially for large datasets. Also, we used the `random_state` parameter to ensure the reproducibility of the initial shuffling of the training dataset after each epoch.

Having trained a model in scikit-learn, we can make predictions via the `predict` method, just like in our own perceptron implementation in *Chapter 2*. The code is as follows:

```
>>> y_pred = ppn.predict(X_test_std)
>>> print('Misclassified examples: %d' % (y_test != y_pred).sum())
Misclassified examples: 1
```

Executing the code, we can see that the perceptron misclassifies 1 out of the 45 flower examples. Thus, the misclassification error on the test dataset is approximately 0.022, or 2.2 percent $\left(\frac{1}{45} \approx 0.022\right)$.

Classification error versus accuracy

Instead of the misclassification error, many machine learning practitioners report the classification accuracy of a model, which is simply calculated as follows:

1–*error* = 0.978, or 97.8 percent

Whether we use the classification error or accuracy is merely a matter of preference.

Note that scikit-learn also implements a large variety of different performance metrics that are available via the `metrics` module. For example, we can calculate the classification accuracy of the perceptron on the test dataset as follows:

```
>>> from sklearn.metrics import accuracy_score
>>> print('Accuracy: %.3f' % accuracy_score(y_test, y_pred))
Accuracy: 0.978
```

Here, `y_test` is the true class labels and `y_pred` is the class labels that we predicted previously. Alternatively, each classifier in scikit-learn has a `score` method, which computes a classifier's prediction accuracy by combining the `predict` call with `accuracy_score`, as shown here:

```
>>> print('Accuracy: %.3f' % ppn.score(X_test_std, y_test))
Accuracy: 0.978
```

Overfitting

Note that we will evaluate the performance of our models based on the test dataset in this chapter. In *Chapter 6*, you will learn about useful techniques, including graphical analysis, such as learning curves, to detect and prevent overfitting. Overfitting, which we will return to later in this chapter, means that the model captures the patterns in the training data well but fails to generalize well to unseen data.

Finally, we can use our `plot_decision_regions` function from *Chapter 2* to plot the **decision regions** of our newly trained perceptron model and visualize how well it separates the different flower examples. However, let's add a small modification to highlight the data instances from the test dataset via small circles:

```
from matplotlib.colors import ListedColormap
import matplotlib.pyplot as plt
def plot_decision_regions(X, y, classifier, test_idx=None,
                          resolution=0.02):
    # setup marker generator and color map
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])
```

```
    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class examples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')
    # highlight test examples
    if test_idx:
        # plot all examples
        X_test, y_test = X[test_idx, :], y[test_idx]

        plt.scatter(X_test[:, 0], X_test[:, 1],
                    c='none', edgecolor='black', alpha=1.0,
                    linewidth=1, marker='o',
                    s=100, label='Test set')
```

With the slight modification that we made to the `plot_decision_regions` function, we can now specify the indices of the examples that we want to mark on the resulting plots. The code is as follows:

```
>>> X_combined_std = np.vstack((X_train_std, X_test_std))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X=X_combined_std,
...                       y=y_combined,
...                       classifier=ppn,
...                       test_idx=range(105, 150))
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in the resulting plot, the three flower classes can't be perfectly separated by a linear decision boundary:

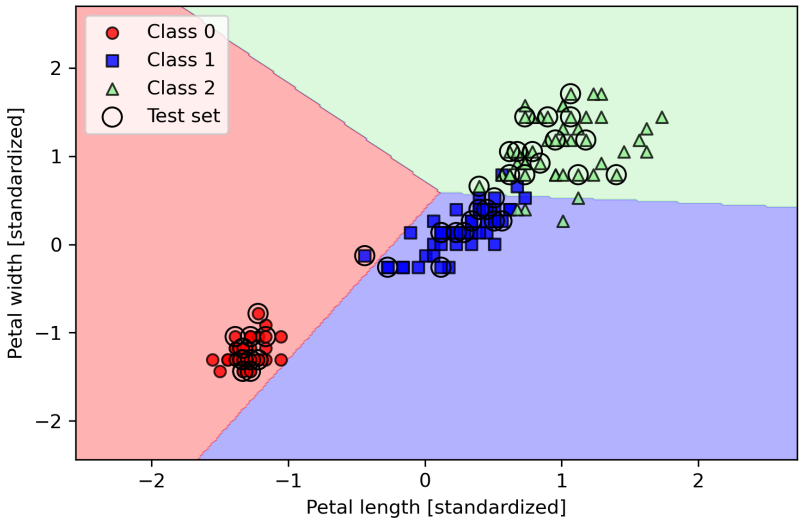


Figure 3.1: Decision boundaries of a multi-class perceptron model fitted to the Iris dataset

However, remember from our discussion in *Chapter 2* that the perceptron algorithm never converges on datasets that aren't perfectly linearly separable, which is why the use of the perceptron algorithm is typically not recommended in practice. In the following sections, we will look at more powerful linear classifiers that converge to a loss minimum even if the classes are not perfectly linearly separable.

Additional perceptron settings

The `Perceptron`, as well as other scikit-learn functions and classes, often has additional parameters that we omit for clarity. You can read more about those parameters using the `help` function in Python (for instance, `help(Perceptron)`) or by going through the excellent scikit-learn online documentation at `http://scikit-learn.org/stable/`.

Modeling class probabilities via logistic regression

Although the perceptron rule offers a nice and easy-going introduction to machine learning algorithms for classification, its biggest disadvantage is that it never converges if the classes are not perfectly linearly separable. The classification task in the previous section would be an example of such a scenario. The reason for this is that the weights are continuously being updated since there is always at least one misclassified training example present in each epoch. Of course, you can change the learning rate and increase the number of epochs, but be warned that the perceptron will never converge on this dataset.

To make better use of our time, we will now take a look at another simple, yet more powerful, algorithm for linear and binary classification problems: **logistic regression**. Note that, despite its name, logistic regression is a model for classification, not regression.

Logistic regression and conditional probabilities

Logistic regression is a classification model that is very easy to implement and performs very well on linearly separable classes. It is one of the most widely used algorithms for classification in industry. Similar to the perceptron and Adaline, the logistic regression model in this chapter is also a linear model for binary classification.

Logistic regression for multiple classes

Note that logistic regression can be readily generalized to multiclass settings, which is known as **multinomial logistic regression**, or **softmax regression**. More detailed coverage of multinomial logistic regression is outside the scope of this book, but the interested reader can find more information in my lecture notes at `https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L08_logistic__slides.pdf` or `https://www.youtube.com/watch?v=L0FU8NFpx4E`.

Another way to use logistic regression in multiclass settings is via the OvR technique, which we discussed previously.

To explain the main mechanics behind logistic regression as a probabilistic model for binary classification, let's first introduce the **odds**: the odds in favor of a particular event. The odds can be written as $\frac{p}{(1-p)}$, where p stands for the probability of the positive event. The term "positive event" does not necessarily mean "good," but refers to the event that we want to predict, for example, the probability that a patient has a certain disease given certain symptoms; we can think of the positive event as class label $y = 1$ and the symptoms as features $\boldsymbol{x}$. Hence, for brevity, we can define the probability p as $p := p(y = 1|\boldsymbol{x})$, the conditional probability that a particular example belongs to a certain class 1 given its features, $\boldsymbol{x}$.

We can then further define the **logit** function, which is simply the logarithm of the odds (log-odds):

$$\text{logit}(p) = \log \frac{p}{(1-p)}$$

Note that *log* refers to the natural logarithm, as it is the common convention in computer science. The *logit* function takes input values in the range 0 to 1 and transforms them into values over the entire real-number range.

Under the logistic model, we assume that there is a linear relationship between the weighted inputs (referred to as net inputs in *Chapter 2*) and the log-odds:

$$\text{logit}(p) = w_1 x_1 + \cdots + w_m x_m + b = \sum_{i=j} w_j x_j + b = \boldsymbol{w}^T \boldsymbol{x} + b$$

While the preceding describes an assumption we make about the linear relationship between the log-odds and the net inputs, what we are actually interested in is the probability p, the class-membership probability of an example given its features. While the logit function maps the probability to a real-number range, we can consider the inverse of this function to map the real-number range back to a [0, 1] range for the probability p.

This inverse of the logit function is typically called the **logistic sigmoid function**, which is sometimes simply abbreviated to **sigmoid function** due to its characteristic S-shape:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Here, z is the net input, the linear combination of weights, and the inputs (that is, the features associated with the training examples):

$$z = \boldsymbol{w}^T\boldsymbol{x} + b$$

Now, let's simply plot the sigmoid function for some values in the range –7 to 7 to see how it looks:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def sigmoid(z):
...     return 1.0 / (1.0 + np.exp(-z))
>>> z = np.arange(-7, 7, 0.1)
>>> sigma_z = sigmoid(z)
>>> plt.plot(z, sigma_z)
>>> plt.axvline(0.0, color='k')
>>> plt.ylim(-0.1, 1.1)
>>> plt.xlabel('z')
>>> plt.ylabel('$\sigma (z)$')
>>> # y axis ticks and gridline
>>> plt.yticks([0.0, 0.5, 1.0])
>>> ax = plt.gca()
>>> ax.yaxis.grid(True)
>>> plt.tight_layout()
>>> plt.show()
```

As a result of executing the previous code example, we should now see the S-shaped (sigmoidal) curve:

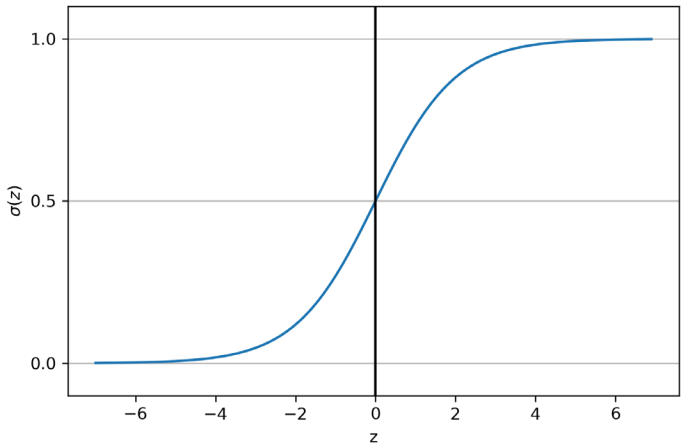


Figure 3.2: A plot of the logistic sigmoid function

We can see that $\sigma(z)$ approaches 1 if z goes toward infinity ($z \to \infty$) since e^{-z} becomes very small for large values of z. Similarly, $\sigma(z)$ goes toward 0 for $z \to -\infty$ as a result of an increasingly large denominator. Thus, we can conclude that this sigmoid function takes real-number values as input and transforms them into values in the range [0, 1] with an intercept at $\sigma(0) = 0.5$.

To build some understanding of the logistic regression model, we can relate it to *Chapter 2*. In Adaline, we used the identity function, $\sigma(z) = z$, as the activation function. In logistic regression, this activation function simply becomes the sigmoid function that we defined earlier.

The difference between Adaline and logistic regression is illustrated in the following figure, where the only difference is the activation function:

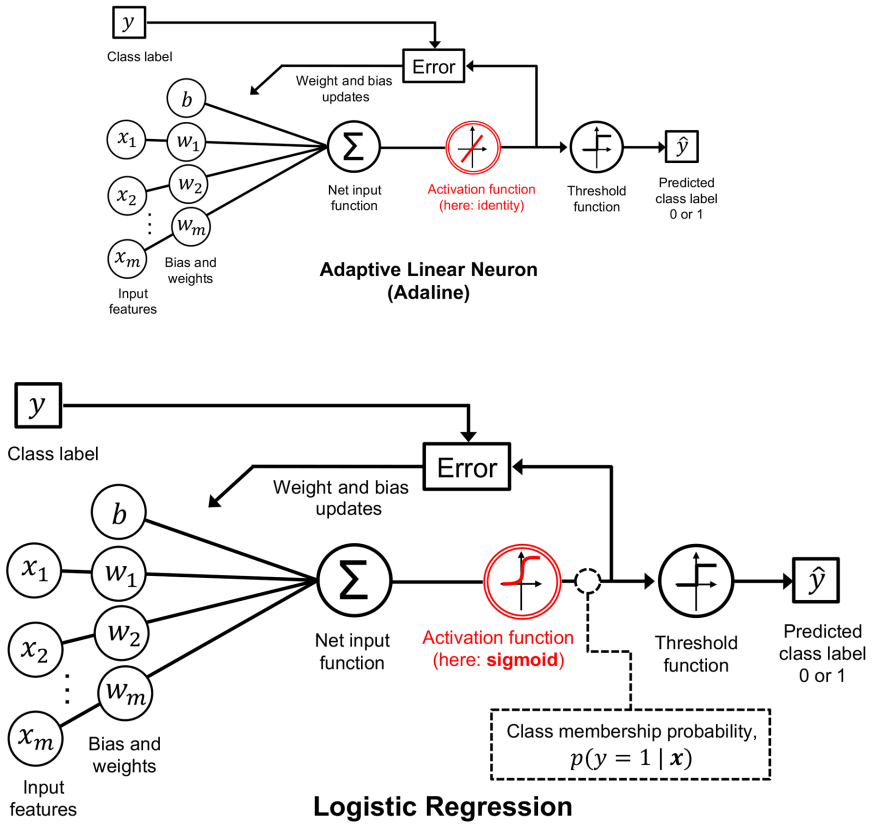


Figure 3.3: Logistic regression compared to Adaline

The output of the sigmoid function is then interpreted as the probability of a particular example belonging to class 1, $\sigma(z) = p(y = 1|\boldsymbol{x}; \boldsymbol{w}, b)$, given its features, x, and parameterized by the weights and bias, w and b. For example, if we compute $\sigma(z) = 0.8$ for a particular flower example, it means that the chance that this example is an `Iris-versicolor` flower is 80 percent. Therefore, the probability that this flower is an `Iris-setosa` flower can be calculated as $p(y = 0|\boldsymbol{x}; \boldsymbol{w}, b) = 1 - p(y = 1|\boldsymbol{x}; \boldsymbol{w}, b) = 0.2$, or 20 percent.

The predicted probability can then simply be converted into a binary outcome via a threshold function:

$$\hat{y} = \begin{cases} 1 & \text{if } \sigma(z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

If we look at the preceding plot of the sigmoid function, this is equivalent to the following:

$$\hat{y} = \begin{cases} 1 & \text{if } z \geq 0.0 \\ 0 & \text{otherwise} \end{cases}$$

In fact, there are many applications where we are not only interested in the predicted class labels, but where the estimation of the class-membership probability is particularly useful (the output of the sigmoid function prior to applying the threshold function). Logistic regression is used in weather forecasting, for example, not only to predict whether it will rain on a particular day, but also to report the chance of rain. Similarly, logistic regression can be used to predict the chance that a patient has a particular disease given certain symptoms, which is why logistic regression enjoys great popularity in the field of medicine.

Learning the model weights via the logistic loss function

You have learned how we can use the logistic regression model to predict probabilities and class labels; now, let's briefly talk about how we fit the parameters of the model, for instance, the weights and bias unit, w and b. In the previous chapter, we defined the mean squared error loss function as follows:

$$L(\boldsymbol{w}, b|\boldsymbol{x}) = \sum_i \frac{1}{2}\left(\sigma\left(z^{(i)}\right) - y^{(i)}\right)^2$$

We minimized this function in order to learn the parameters for our Adaline classification model. To explain how we can derive the loss function for logistic regression, let's first define the likelihood, $\mathcal{L}$, that we want to maximize when we build a logistic regression model, assuming that the individual examples in our dataset are independent of one another. The formula is as follows:

$$\mathcal{L}(\boldsymbol{w}, b|\boldsymbol{x}) = p(y|\boldsymbol{x}; \boldsymbol{w}, b) = \prod_{i=1}^{n} p\left(y^{(i)}|\boldsymbol{x}^{(i)}; \boldsymbol{w}, b\right) = \prod_{i=1}^{n} \left(\sigma\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \sigma\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

In practice, it is easier to maximize the (natural) log of this equation, which is called the **log-likelihood** function:

$$l(\boldsymbol{w}, b|\boldsymbol{x}) = \log \mathcal{L}(\boldsymbol{w}, b|\boldsymbol{x}) = \sum_{i=1} \left[y^{(i)} \log\left(\sigma\left(z^{(i)}\right)\right) + \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(z^{(i)}\right)\right)\right]$$

Firstly, applying the log function reduces the potential for numerical underflow, which can occur if the likelihoods are very small. Secondly, we can convert the product of factors into a summation of factors, which makes it easier to obtain the derivative of this function via the addition trick, as you may remember from calculus.

Deriving the likelihood function

We can obtain the expression for the likelihood of the model given the data, $\mathcal{L}(\mathbf{w}, b \mid \mathbf{x})$, as follows. Given that we have a binary classification problem with class labels 0 and 1, we can think of the label 1 as a Bernoulli variable—it can take on two values, 0 and 1, with the probability p of being 1: $Y \sim Bern(p)$. For a single data point, we can write this probability as $P\left(Y = 1 \mid X = \mathrm{x}^{(i)}\right) = \sigma\left(z^{(i)}\right)$ and $P\left(Y = 0 \mid X = \mathrm{x}^{(i)}\right) = 1 - \sigma\left(z^{(i)}\right)$.

Putting these two expressions together, and using the shorthand $P\left(Y = y^{(i)} \mid X = \mathrm{x}^{(i)}\right) = p\left(y^{(i)} \mid \mathrm{x}^{(i)}\right)$, we get the probability mass function of the Bernoulli variable:

$$p\left(y^{(i)} \mid \mathrm{x}^{(i)}\right) = \left(\sigma\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \sigma\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

We can write the likelihood of the training labels given the assumption that all training examples are independent, using the multiplication rule to compute the probability that all events occur, as follows:

$$\mathcal{L}(\mathbf{w}, b \mid \mathbf{x}) = \prod_{i=1}^{n} p\left(y^{(i)} \mid \mathbf{x}^{(i)}; \mathbf{w}, b\right)$$

Now, substituting the probability mass function of the Bernoulli variable, we arrive at the expression of the likelihood, which we attempt to maximize by changing the model parameters:

$$\mathcal{L}(\mathbf{w}, b \mid \mathbf{x}) = \prod_{i=1}^{n} \left(\sigma\left(z^{(i)}\right)\right)^{y^{(i)}} \left(1 - \sigma\left(z^{(i)}\right)\right)^{1-y^{(i)}}$$

Now, we could use an optimization algorithm such as gradient ascent to maximize this log-likelihood function. (Gradient ascent works exactly the same way as gradient descent explained in *Chapter 2*, except that gradient ascent maximizes a function instead of minimizing it.) Alternatively, let's rewrite the log-likelihood as a loss function, L, that can be minimized using gradient descent as in *Chapter 2*:

$$L(\boldsymbol{w}, b) = \sum_{i=1}^{n} \left[-y^{(i)} \log\left(\sigma\left(z^{(i)}\right)\right) - \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(z^{(i)}\right)\right)\right]$$

To get a better grasp of this loss function, let's take a look at the loss that we calculate for one single training example:

$$L(\sigma(z), y; \boldsymbol{w}, b) = -y \log(\sigma(z)) - (1 - y) \log(1 - \sigma(z))$$

Looking at the equation, we can see that the first term becomes zero if $y = 0$, and the second term becomes zero if $y = 1$:

$$L(\sigma(z), y; \boldsymbol{w}, b) = \begin{cases} -\log(\sigma(z)) & \text{if } y = 1 \\ -\log(1 - \sigma(z)) & \text{if } y = 0 \end{cases}$$

Let's write a short code snippet to create a plot that illustrates the loss of classifying a single training example for different values of $\sigma(z)$:

```
>>> def loss_1(z):
...     return - np.log(sigmoid(z))
>>> def loss_0(z):
...     return - np.log(1 - sigmoid(z))
>>> z = np.arange(-10, 10, 0.1)
>>> sigma_z = sigmoid(z)
>>> c1 = [loss_1(x) for x in z]
>>> plt.plot(sigma_z, c1, label='L(w, b) if y=1')
>>> c0 = [loss_0(x) for x in z]
>>> plt.plot(sigma_z, c0, linestyle='--', label='L(w, b) if y=0')
>>> plt.ylim(0.0, 5.1)
>>> plt.xlim([0, 1])
>>> plt.xlabel('$\sigma(z)$')
>>> plt.ylabel('L(w, b)')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

The resulting plot shows the sigmoid activation on the x axis in the range 0 to 1 (the inputs to the sigmoid function were z values in the range –10 to 10) and the associated logistic loss on the y axis:

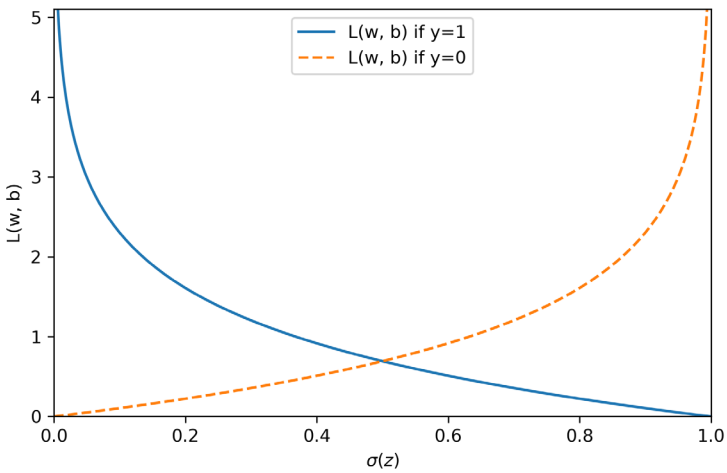


Figure 3.4: A plot of the loss function used in logistic regression

We can see that the loss approaches 0 (continuous line) if we correctly predict that an example belongs to class 1. Similarly, we can see on the y axis that the loss also approaches 0 if we correctly predict $y = 0$ (dashed line). However, if the prediction is wrong, the loss goes toward infinity. The main point is that we penalize wrong predictions with an increasingly larger loss.

Converting an Adaline implementation into an algorithm for logistic regression

If we were to implement logistic regression ourselves, we could simply substitute the loss function, L, in our Adaline implementation from *Chapter 2*, with the new loss function:

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left[-y^{(i)}\log\left(\sigma\left(z^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-\sigma\left(z^{(i)}\right)\right)\right]$$

We use this to compute the loss of classifying all training examples per epoch. Also, we need to swap the linear activation function with the sigmoid. If we make those changes to the Adaline code, we will end up with a working logistic regression implementation. The following is an implementation for full-batch gradient descent (but note that the same changes could be made to the stochastic gradient descent version as well):

```
class LogisticRegressionGD:
    """Gradient descent-based logistic regression classifier.

    Parameters
    ------------
    eta : float
      Learning rate (between 0.0 and 1.0)
    n_iter : int
      Passes over the training dataset.
    random_state : int
      Random number generator seed for random weight
      initialization.


    Attributes
    -----------
    w_ : 1d-array
      Weights after training.
    b_ : Scalar
      Bias unit after fitting.
    losses_ : list
      Mean squared error loss function values in each epoch.

    """
    def __init__(self, eta=0.01, n_iter=50, random_state=1):
        self.eta = eta
        self.n_iter = n_iter
        self.random_state = random_state
```

```
    def fit(self, X, y):
        """ Fit training data.

        Parameters
        ----------
        X : {array-like}, shape = [n_examples, n_features]
          Training vectors, where n_examples is the
          number of examples and n_features is the
          number of features.
        y : array-like, shape = [n_examples]
          Target values.

        Returns
        -------
        self : Instance of LogisticRegressionGD

        """
        rgen = np.random.RandomState(self.random_state)
        self.w_ = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1])
        self.b_ = np.float_(0.)
        self.losses_ = []

        for i in range(self.n_iter):
            net_input = self.net_input(X)
            output = self.activation(net_input)
            errors = (y - output)
            self.w_ += self.eta * 2.0 * X.T.dot(errors) / X.shape[0]
            self.b_ += self.eta * 2.0 * errors.mean()
            loss = (-y.dot(np.log(output))
                    - ((1 - y).dot(np.log(1 - output)))
                     / X.shape[0])
            self.losses_.append(loss)
        return self

    def net_input(self, X):
        """Calculate net input"""
        return np.dot(X, self.w_) + self.b_

    def activation(self, z):
        """Compute logistic sigmoid activation"""
        return 1. / (1. + np.exp(-np.clip(z, -250, 250)))
```

```
    def predict(self, X):
        """Return class label after unit step"""
        return np.where(self.activation(self.net_input(X)) >= 0.5, 1, 0)
```

When we fit a logistic regression model, we have to keep in mind that it only works for binary classification tasks.

So, let's consider only setosa and versicolor flowers (classes `0` and `1`) and check that our implementation of logistic regression works:

```
>>> X_train_01_subset = X_train_std[(y_train == 0) | (y_train == 1)]
>>> y_train_01_subset = y_train[(y_train == 0) | (y_train == 1)]
>>> lrgd = LogisticRegressionGD(eta=0.3,
...                             n_iter=1000,
...                             random_state=1)
>>> lrgd.fit(X_train_01_subset,
...          y_train_01_subset)
>>> plot_decision_regions(X=X_train_01_subset,
...                       y=y_train_01_subset,
...                       classifier=lrgd)
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

The resulting decision region plot looks as follows:

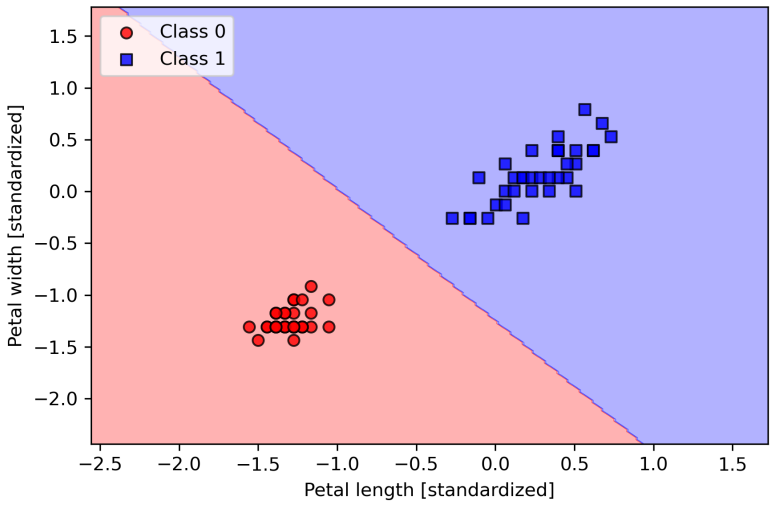


Figure 3.5: The decision region plot for the logistic regression model

The gradient descent learning algorithm for logistic regression

If you compared the `LogisticRegressionGD` in the previous code with the `AdalineGD` code from *Chapter 2*, you may have noticed that the weight and bias update rules remained unchanged (except for the scaling factor 2). Using calculus, we can show that the parameter updates via gradient descent are indeed similar for logistic regression and Adaline. However, please note that the following derivation of the gradient descent learning rule is intended for readers who are interested in the mathematical concepts behind the gradient descent learning rule for logistic regression. It is not essential for following the rest of this chapter.

Figure 3.6 summarizes how we can calculate the partial derivative of the log-likelihood function with respect to the *j*th weight:

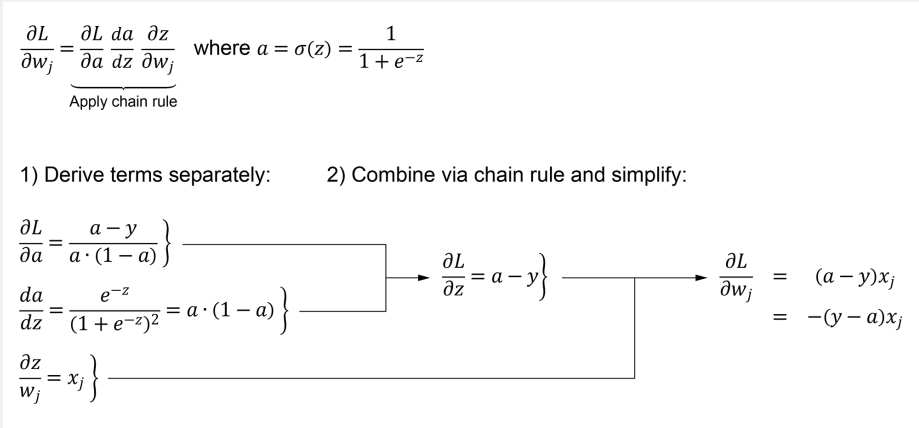


Figure 3.6: Calculating the partial derivative of the log-likelihood function

Note that we omitted averaging over the training examples for brevity.

Remember from *Chapter 2* that we take steps in the opposite direction of the gradient. Hence, we flip $\frac{\partial L}{w_j} = -(y-a)x_j$ and update the *j*th weight as follows, including the learning rate η:

$$w_j \coloneqq w_j + \eta(y-a)x_j$$

While the partial derivative of the loss function with respect to the bias unit is not shown, bias derivation follows the same overall concept using the chain rule, resulting in the following update rule:

$$b \coloneqq b + \eta(y-a)$$

Both the weight and bias unit updates are equal to the ones for Adaline in *Chapter 2*.

Training a logistic regression model with scikit-learn

We just went through useful coding and math exercises in the previous subsection, which helped to illustrate the conceptual differences between Adaline and logistic regression. Now, let's learn how to use scikit-learn's more optimized implementation of logistic regression, which also supports multiclass settings off the shelf. Note that in recent versions of scikit-learn, the technique used for multiclass classification, multinomial, or OvR, is chosen automatically. In the following code example, we will use the `sklearn.linear_model.LogisticRegression` class as well as the familiar `fit` method to train the model on all three classes in the standardized flower training dataset. Also, we set `multi_class='ovr'` for illustration purposes. As an exercise for the reader, you may want to compare the results with `multi_class='multinomial'`. Note that the `multinomial` setting is now the default choice in scikit-learn's `LogisticRegression` class and recommended in practice for mutually exclusive classes, such as those found in the Iris dataset. Here, "mutually exclusive" means that each training example can only belong to a single class (in contrast to multilabel classification, where a training example can be a member of multiple classes).

Now, let's have a look at the code example:

```
>>> from sklearn.linear_model import LogisticRegression
>>> lr = LogisticRegression(C=100.0, solver='lbfgs',
...                         multi_class='ovr')
>>> lr.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined,
...                       classifier=lr,
...                       test_idx=range(105, 150))
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After fitting the model on the training data, we plotted the decision regions, training examples, and test examples, as shown in *Figure 3.7*:

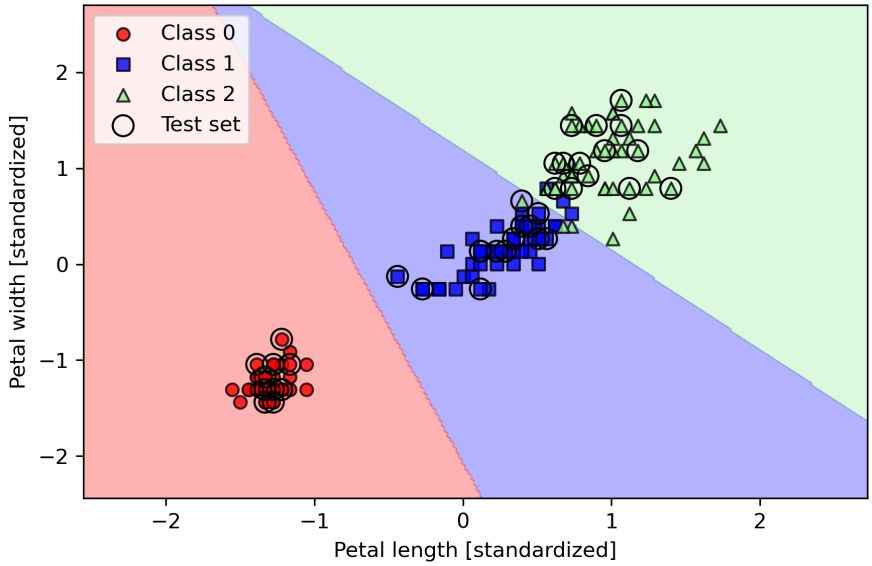


Figure 3.7: Decision regions for scikit-learn's multi-class logistic regression model

Algorithms for convex optimization

Note that there exist many different algorithms for solving optimization problems. For minimizing convex loss functions, such as the logistic regression loss, it is recommended to use more advanced approaches than regular **stochastic gradient descent** (**SGD**). In fact, scikit-learn implements a whole range of such optimization algorithms, which can be specified via the `solver` parameter, namely, `'newton-cg'`, `'lbfgs'`, `'liblinear'`, `'sag'`, and `'saga'`.

While the logistic regression loss is convex, most optimization algorithms should converge to the global loss minimum with ease. However, there are certain advantages of using one algorithm over the other. For example, in previous versions (for instance, v 0.21), scikit-learn used `'liblinear'` as a default, which cannot handle the multinomial loss and is limited to the OvR scheme for multiclass classification. However, in scikit-learn v 0.22, the default solver was changed to `'lbfgs'`, which stands for the limited-memory **Broyden–Fletcher–Goldfarb–Shanno** (**BFGS**) algorithm (`https://en.wikipedia.org/wiki/Limited-memory_BFGS`) and is more flexible in this regard.

Looking at the preceding code that we used to train the `LogisticRegression` model, you might now be wondering, "What is this mysterious parameter C?" We will discuss this parameter in the next subsection, where we will introduce the concepts of overfitting and regularization. However, before we move on to those topics, let's finish our discussion of class membership probabilities.

The probability that training examples belong to a certain class can be computed using the `predict_proba` method. For example, we can predict the probabilities of the first three examples in the test dataset as follows:

```
>>> lr.predict_proba(X_test_std[:3, :])
```

This code snippet returns the following array:

```
array([[3.81527885e-09, 1.44792866e-01, 8.55207131e-01],
       [8.34020679e-01, 1.65979321e-01, 3.25737138e-13],
       [8.48831425e-01, 1.51168575e-01, 2.62277619e-14]])
```

The first row corresponds to the class membership probabilities of the first flower, the second row corresponds to the class membership probabilities of the second flower, and so forth. Notice that the column-wise sum in each row is 1, as expected. (You can confirm this by executing `lr.predict_proba(X_test_std[:3, :]).sum(axis=1)`.)

The highest value in the first row is approximately 0.85, which means that the first example belongs to class 3 (`Iris-virginica`) with a predicted probability of 85 percent. So, as you may have already noticed, we can get the predicted class labels by identifying the largest column in each row, for example, using NumPy's `argmax` function:

```
>>> lr.predict_proba(X_test_std[:3, :]).argmax(axis=1)
```

The returned class indices are shown here (they correspond to `Iris-virginica`, `Iris-setosa`, and `Iris-setosa`):

```
array([2, 0, 0])
```

In the preceding code example, we computed the conditional probabilities and converted these into class labels manually by using NumPy's `argmax` function. In practice, the more convenient way of obtaining class labels when using scikit-learn is to call the `predict` method directly:

```
>>> lr.predict(X_test_std[:3, :])
array([2, 0, 0])
```

Lastly, a word of caution if you want to predict the class label of a single flower example: scikit-learn expects a two-dimensional array as data input; thus, we have to convert a single row slice into such a format first. One way to convert a single row entry into a two-dimensional data array is to use NumPy's `reshape` method to add a new dimension, as demonstrated here:

```
>>> lr.predict(X_test_std[0, :].reshape(1, -1))
array([2])
```

Tackling overfitting via regularization

Overfitting is a common problem in machine learning, where a model performs well on training data but does not generalize well to unseen data (test data). If a model suffers from overfitting, we also say that the model has a high variance, which can be caused by having too many parameters, leading to a model that is too complex given the underlying data. Similarly, our model can also suffer from **underfitting** (high bias), which means that our model is not complex enough to capture the pattern in the training data well and therefore also suffers from low performance on unseen data.

Although we have only encountered linear models for classification so far, the problems of overfitting and underfitting can be best illustrated by comparing a linear decision boundary to more complex, nonlinear decision boundaries, as shown in *Figure 3.8*:

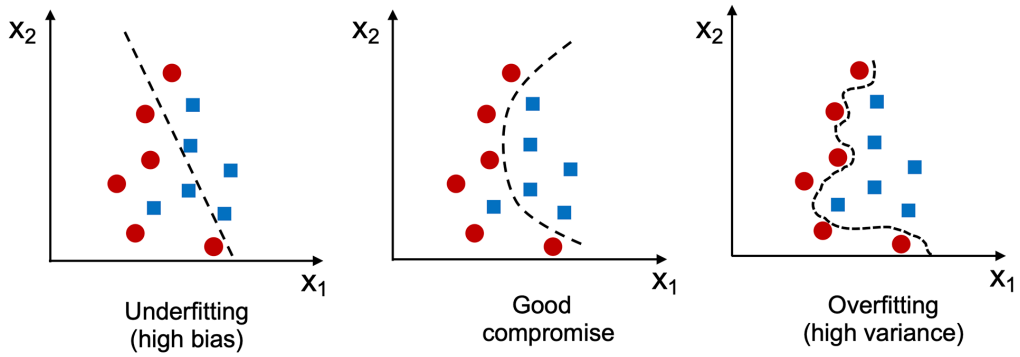


Figure 3.8: Examples of underfitted, well-fitted, and overfitted models

The bias-variance tradeoff

Often, researchers use the terms "bias" and "variance" or "bias-variance tradeoff" to describe the performance of a model—that is, you may stumble upon talks, books, or articles where people say that a model has a "high variance" or "high bias." So, what does that mean? In general, we might say that "high variance" is proportional to overfitting and "high bias" is proportional to underfitting.

In the context of machine learning models, variance measures the consistency (or variability) of the model prediction for classifying a particular example if we retrain the model multiple times, for example, on different subsets of the training dataset. We can say that the model is sensitive to the randomness in the training data. In contrast, bias measures how far off the predictions are from the correct values in general if we rebuild the model multiple times on different training datasets; bias is the measure of the systematic error that is not due to randomness.

If you are interested in the technical specification and derivation of the "bias" and "variance" terms, I've written about it in my lecture notes here: `https://sebastianraschka.com/pdf/lecture-notes/stat451fs20/08-model-eval-1-intro__notes.pdf`.

One way of finding a good bias-variance tradeoff is to tune the complexity of the model via regularization. Regularization is a very useful method for handling collinearity (high correlation among features), filtering out noise from data, and eventually preventing overfitting.

The concept behind regularization is to introduce additional information to penalize extreme parameter (weight) values. The most common form of regularization is so-called **L2 regularization** (sometimes also called L2 shrinkage or weight decay), which can be written as follows:

$$\frac{\lambda}{2n}\|\boldsymbol{w}\|^2 = \frac{\lambda}{2n}\sum_{j=1}^{m} w_j^2$$

Here, λ is the so-called **regularization parameter**. Note that the 2 in the denominator is merely a scaling factor, such that it cancels when computing the loss gradient. The sample size n is added to scale the regularization term similar to the loss.

Regularization and feature normalization

Regularization is another reason why feature scaling such as standardization is important. For regularization to work properly, we need to ensure that all our features are on comparable scales.

The loss function for logistic regression can be regularized by adding a simple regularization term, which will shrink the weights during model training:

$$L(\boldsymbol{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\left[-y^{(i)}\log\left(\sigma\left(z^{(i)}\right)\right) - \left(1-y^{(i)}\right)\log\left(1-\sigma\left(z^{(i)}\right)\right)\right] + \frac{\lambda}{2n}\|\boldsymbol{w}\|^2$$

The partial derivative of the unregularized loss is defined as:

$$\frac{\partial L(\boldsymbol{w}, b)}{\partial w_j} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(\sigma\left(\boldsymbol{w}^T\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}\right)$$

Adding the regularization term to the loss changes the partial derivative to the following form:

$$\frac{\partial L(\boldsymbol{w}, b)}{\partial w_j} = \left(\frac{1}{n}\sum_{i=1}^{n}\left(\sigma\left(\boldsymbol{w}^T\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)x_j^{(i)}\right) + \frac{\lambda}{n}w_j$$

Via the regularization parameter, λ, we can then control how closely we fit the training data, while keeping the weights small. By increasing the value of λ, we increase the regularization strength. Please note that the bias unit, which is essentially an intercept term or negative threshold, as we learned in *Chapter 2*, is usually not regularized.

The parameter, `C`, that is implemented for the `LogisticRegression` class in scikit-learn comes from a convention in support vector machines, which will be the topic of the next section. The term `C` is inversely proportional to the regularization parameter, λ. Consequently, decreasing the value of the inverse regularization parameter, `C`, means that we are increasing the regularization strength, which we can visualize by plotting the L2 regularization path for the two weight coefficients:

```
>>> weights, params = [], []
>>> for c in np.arange(-5, 5):
...     lr = LogisticRegression(C=10.**c,
...                             multi_class='ovr')
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10.**c)
>>> weights = np.array(weights)
>>> plt.plot(params, weights[:, 0],
...          label='Petal length')
>>> plt.plot(params, weights[:, 1], linestyle='--',
...          label='Petal width')
>>> plt.ylabel('Weight coefficient')
>>> plt.xlabel('C')
>>> plt.legend(loc='upper left')
>>> plt.xscale('log')
>>> plt.show()
```

By executing the preceding code, we fitted 10 logistic regression models with different values for the inverse-regularization parameter, `C`. For illustration purposes, we only collected the weight coefficients of class `1` (here, the second class in the dataset: `Iris-versicolor`) versus all classifiers—remember that we are using the OvR technique for multiclass classification.

As we can see in the resulting plot, the weight coefficients shrink if we decrease parameter `C`, that is, if we increase the regularization strength:

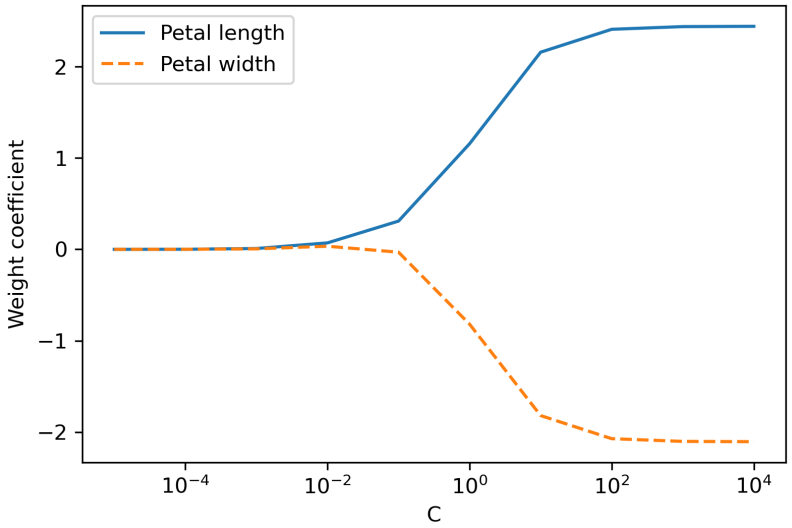


Figure 3.9: The impact of the inverse regularization strength parameter C on L2 regularized model results

Increasing the regularization strength can reduce overfitting, so we might ask why we don't strongly regularize all models by default. The reason is that we have to be careful when adjusting the regularization strength. For instance, if the regularization strength is too high and the weights coefficients approach zero, the model can perform very poorly due to underfitting, as illustrated in *Figure 3.8*.

An additional resource on logistic regression

Since in-depth coverage of the individual classification algorithms exceeds the scope of this book, *Logistic Regression: From Introductory to Advanced Concepts and Applications*, *Dr. Scott Menard*, *Sage Publications*, *2009*, is recommended to readers who want to learn more about logistic regression.

Maximum margin classification with support vector machines

Another powerful and widely used learning algorithm is the **support vector machine** (**SVM**), which can be considered an extension of the perceptron. Using the perceptron algorithm, we minimized misclassification errors. However, in SVMs, our optimization objective is to maximize the margin. The margin is defined as the distance between the separating hyperplane (decision boundary) and the training examples that are closest to this hyperplane, which are the so-called **support vectors**.

This is illustrated in *Figure 3.10*:

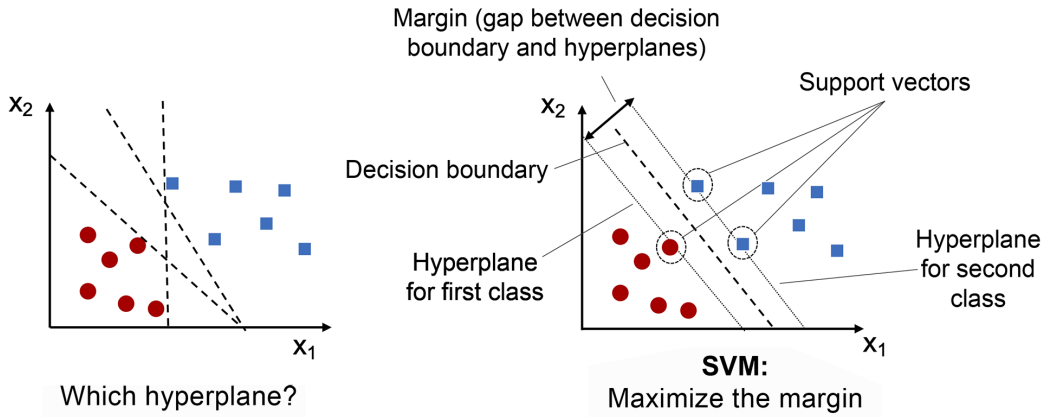


Figure 3.10: SVM maximizes the margin between the decision boundary and training data points

Maximum margin intuition

The rationale behind having decision boundaries with large margins is that they tend to have a lower generalization error, whereas models with small margins are more prone to overfitting.

Unfortunately, while the main intuition behind SVMs is relatively simple, the mathematics behind them is quite advanced and would require sound knowledge of constrained optimization.

Hence, the details behind maximum margin optimization in SVMs are beyond the scope of this book. However, we recommend the following resources if you are interested in learning more:

- Chris J.C. Burges' excellent explanation in *A Tutorial on Support Vector Machines for Pattern Recognition* (Data Mining and Knowledge Discovery, 2(2): 121-167, 1998)
- Vladimir Vapnik's book *The Nature of Statistical Learning Theory*, Springer Science+Business Media, Vladimir Vapnik, 2000
- Andrew Ng's very detailed lecture notes available at `https://see.stanford.edu/materials/aimlcs229/cs229-notes3.pdf`

Dealing with a nonlinearly separable case using slack variables

Although we don't want to dive much deeper into the more involved mathematical concepts behind the maximum-margin classification, let's briefly mention the so-called *slack variable*, which was introduced by Vladimir Vapnik in 1995 and led to the so-called **soft-margin classification**. The motivation for introducing the slack variable was that the linear constraints in the SVM optimization objective need to be relaxed for nonlinearly separable data to allow the convergence of the optimization in the presence of misclassifications, under appropriate loss penalization.

The use of the slack variable, in turn, introduces the variable, which is commonly referred to as *C* in SVM contexts. We can consider *C* as a hyperparameter for controlling the penalty for misclassification. Large values of *C* correspond to large error penalties, whereas we are less strict about misclassification errors if we choose smaller values for *C*. We can then use the *C* parameter to control the width of the margin and therefore tune the bias-variance tradeoff, as illustrated in *Figure 3.11*:

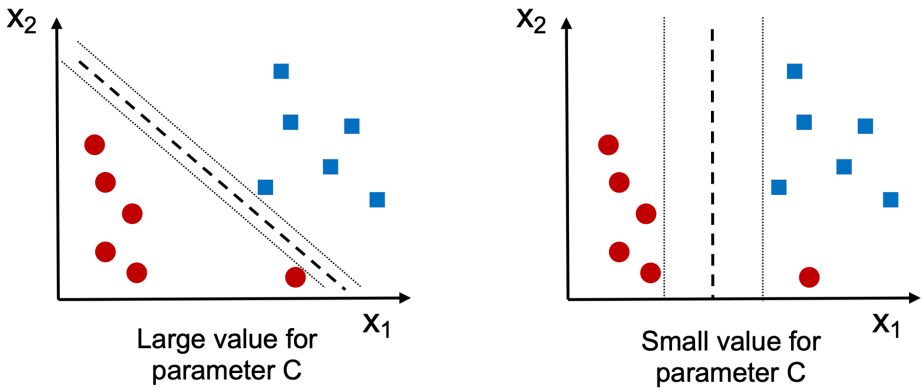


Figure 3.11: The impact of large and small values of the inverse regularization strength C on classification

This concept is related to regularization, which we discussed in the previous section in the context of regularized regression, where decreasing the value of `C` increases the bias (underfitting) and lowers the variance (overfitting) of the model.

Now that we have learned the basic concepts behind a linear SVM, let's train an SVM model to classify the different flowers in our Iris dataset:

```
>>> from sklearn.svm import SVC
>>> svm = SVC(kernel='linear', C=1.0, random_state=1)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined,
...                       classifier=svm,
...                       test_idx=range(105, 150))
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

The three decision regions of the SVM, visualized after training the classifier on the Iris dataset by executing the preceding code example, are shown in *Figure 3.12*:

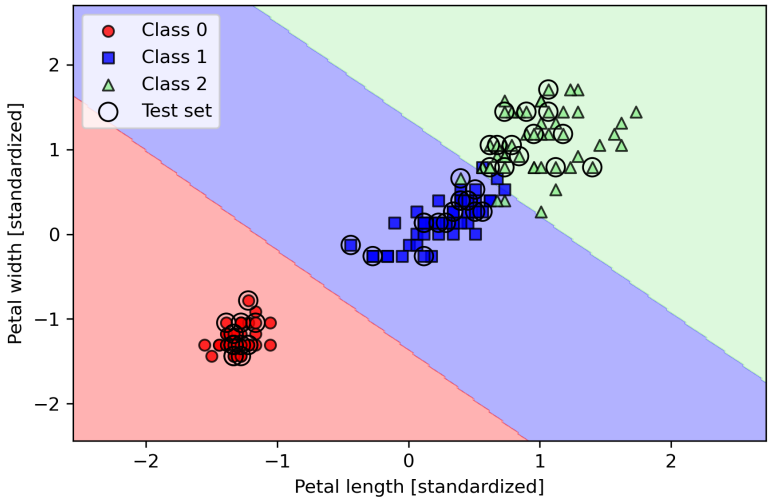


Figure 3.12: SVM's decision regions

Logistic regression versus SVMs

In practical classification tasks, linear logistic regression and linear SVMs often yield very similar results. Logistic regression tries to maximize the conditional likelihoods of the training data, which makes it more prone to outliers than SVMs, which mostly care about the points that are closest to the decision boundary (support vectors). On the other hand, logistic regression has the advantage of being a simpler model and can be implemented more easily, and is mathematically easier to explain. Furthermore, logistic regression models can be easily updated, which is attractive when working with streaming data.

Alternative implementations in scikit-learn

The scikit-learn library's `LogisticRegression` class, which we used in the previous sections, can make use of the LIBLINEAR library by setting `solver='liblinear'`. LIBLINEAR is a highly optimized C/C++ library developed at the National Taiwan University (`http://www.csie.ntu.edu.tw/~cjlin/liblinear/`).

Similarly, the `SVC` class that we used to train an SVM makes use of LIBSVM, which is an equivalent C/C++ library specialized for SVMs (`http://www.csie.ntu.edu.tw/~cjlin/libsvm/`).

The advantage of using LIBLINEAR and LIBSVM over, for example, native Python implementations is that they allow the extremely quick training of large amounts of linear classifiers. However, sometimes our datasets are too large to fit into computer memory. Thus, scikit-learn also offers alternative implementations via the `SGDClassifier` class, which also supports online learning via the `partial_fit` method. The concept behind the `SGDClassifier` class is similar to the stochastic gradient algorithm that we implemented in *Chapter 2* for Adaline.

We could initialize the SGD version of the perceptron (`loss='perceptron'`), logistic regression (`loss='log'`), and an SVM with default parameters (`loss='hinge'`), as follows:

```
>>> from sklearn.linear_model import SGDClassifier
>>> ppn = SGDClassifier(loss='perceptron')
>>> lr = SGDClassifier(loss='log')
>>> svm = SGDClassifier(loss='hinge')
```

Solving nonlinear problems using a kernel SVM

Another reason why SVMs enjoy high popularity among machine learning practitioners is that they can be easily **kernelized** to solve nonlinear classification problems. Before we discuss the main concept behind the so-called **kernel SVM**, the most common variant of SVMs, let's first create a synthetic dataset to see what such a nonlinear classification problem may look like.

Kernel methods for linearly inseparable data

Using the following code, we will create a simple dataset that has the form of an XOR gate using the `logical_xor` function from NumPy, where 100 examples will be assigned the class label `1`, and 100 examples will be assigned the class label `-1`:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> np.random.seed(1)
>>> X_xor = np.random.randn(200, 2)
>>> y_xor = np.logical_xor(X_xor[:, 0] > 0,
...                        X_xor[:, 1] > 0)
>>> y_xor = np.where(y_xor, 1, 0)
>>> plt.scatter(X_xor[y_xor == 1, 0],
...             X_xor[y_xor == 1, 1],
...             c='royalblue', marker='s',
...             label='Class 1')
```

```
>>> plt.scatter(X_xor[y_xor == 0, 0],
...             X_xor[y_xor == 0, 1],
...             c='tomato', marker='o',
...             label='Class 0')
>>> plt.xlim([-3, 3])
>>> plt.ylim([-3, 3])
>>> plt.xlabel('Feature 1')
>>> plt.ylabel('Feature 2')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code, we will have an XOR dataset with random noise, as shown in *Figure 3.13*:

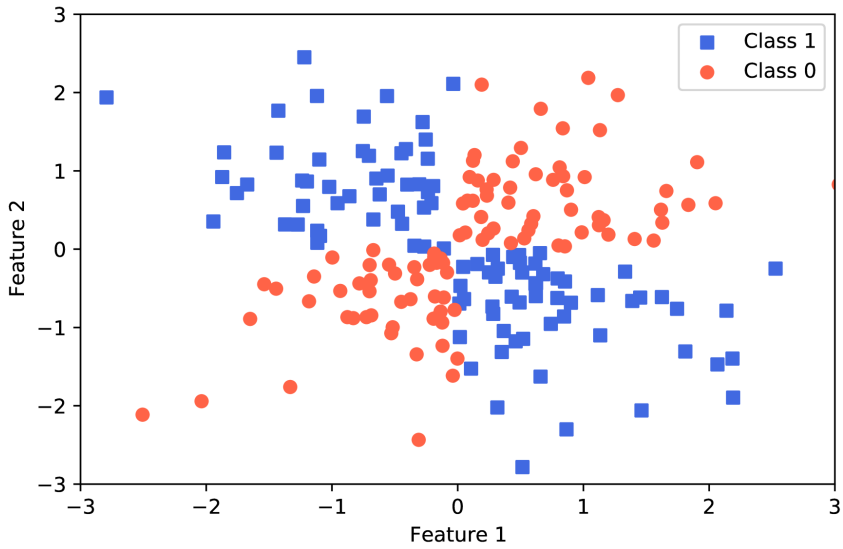


Figure 3.13: A plot of the XOR dataset

Obviously, we would not be able to separate the examples from the positive and negative class very well using a linear hyperplane as a decision boundary via the linear logistic regression or linear SVM model that we discussed in earlier sections.

The basic idea behind **kernel methods** for dealing with such linearly inseparable data is to create nonlinear combinations of the original features to project them onto a higher-dimensional space via a mapping function, ϕ, where the data becomes linearly separable. As shown in *Figure 3.14*, we can transform a two-dimensional dataset into a new three-dimensional feature space, where the classes become separable via the following projection:

$$\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1, x_2, x_1^2 + x_2^2)$$

This allows us to separate the two classes shown in the plot via a linear hyperplane that becomes a nonlinear decision boundary if we project it back onto the original feature space, as illustrated with the following concentric circle dataset:

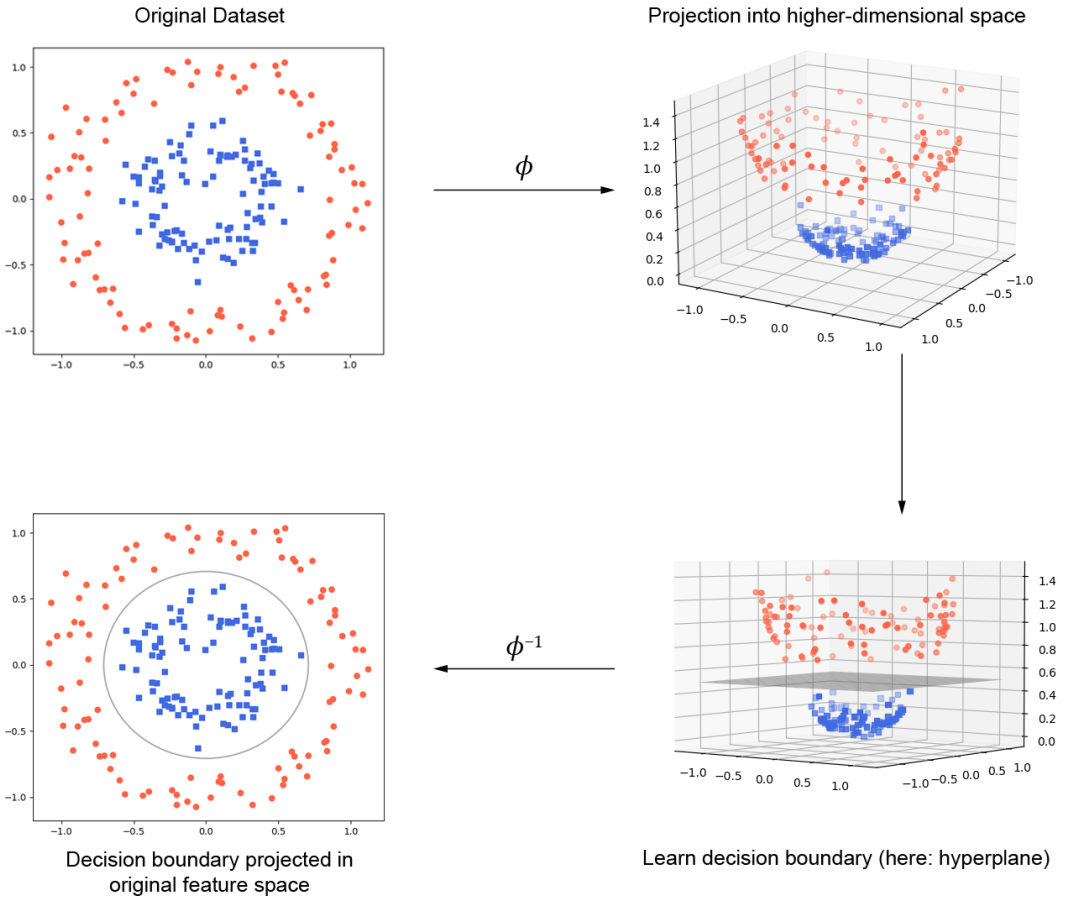


Figure 3.14: The process of classifying nonlinear data using kernel methods

Using the kernel trick to find separating hyperplanes in a high-dimensional space

To solve a nonlinear problem using an SVM, we would transform the training data into a higher-dimensional feature space via a mapping function, ϕ, and train a linear SVM model to classify the data in this new feature space. Then, we could use the same mapping function, ϕ, to transform new, unseen data to classify it using the linear SVM model.

However, one problem with this mapping approach is that the construction of the new features is computationally very expensive, especially if we are dealing with high-dimensional data. This is where the so-called **kernel trick** comes into play.

Although we did not go into much detail about how to solve the quadratic programming task to train an SVM, in practice, we just need to replace the dot product $\boldsymbol{x}^{(i)T}\boldsymbol{x}^{(j)}$ by $\phi\left(\boldsymbol{x}^{(i)}\right)^T \phi\left(\boldsymbol{x}^{(j)}\right)$. To save the expensive step of calculating this dot product between two points explicitly, we define a so-called **kernel function**:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \phi\left(\boldsymbol{x}^{(i)}\right)^T \phi\left(\boldsymbol{x}^{(j)}\right)$$

One of the most widely used kernels is the **radial basis function** (**RBF**) kernel, which can simply be called the **Gaussian kernel**:

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

This is often simplified to:

$$\kappa\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \exp\left(-\gamma\left\|\boldsymbol{x}^{(i)} - \boldsymbol{x}^{(j)}\right\|^2\right)$$

Here, $\gamma = \frac{1}{2\sigma^2}$ is a free parameter to be optimized.

Roughly speaking, the term "kernel" can be interpreted as a **similarity function** between a pair of examples. The minus sign inverts the distance measure into a similarity score, and, due to the exponential term, the resulting similarity score will fall into a range between 1 (for exactly similar examples) and 0 (for very dissimilar examples).

Now that we have covered the big picture behind the kernel trick, let's see if we can train a kernel SVM that is able to draw a nonlinear decision boundary that separates the XOR data well. Here, we simply use the `SVC` class from scikit-learn that we imported earlier and replace the `kernel='linear'` parameter with `kernel='rbf'`:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.10, C=10.0)
>>> svm.fit(X_xor, y_xor)
>>> plot_decision_regions(X_xor, y_xor, classifier=svm)
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in the resulting plot, the kernel SVM separates the XOR data relatively well:

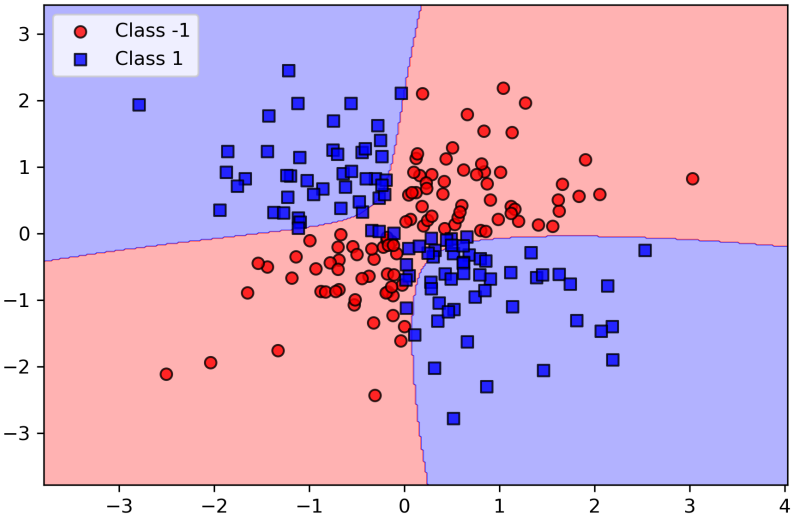


Figure 3.15: The decision boundary on the XOR data using a kernel method

The γ parameter, which we set to `gamma=0.1`, can be understood as a cut-off parameter for the Gaussian sphere. If we increase the value for γ, we increase the influence or reach of the training examples, which leads to a tighter and bumpier decision boundary. To get a better understanding of γ, let's apply an RBF kernel SVM to our Iris flower dataset:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=0.2, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105, 150))
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

Since we chose a relatively small value for γ, the resulting decision boundary of the RBF kernel SVM model will be relatively soft, as shown in *Figure 3.16*:

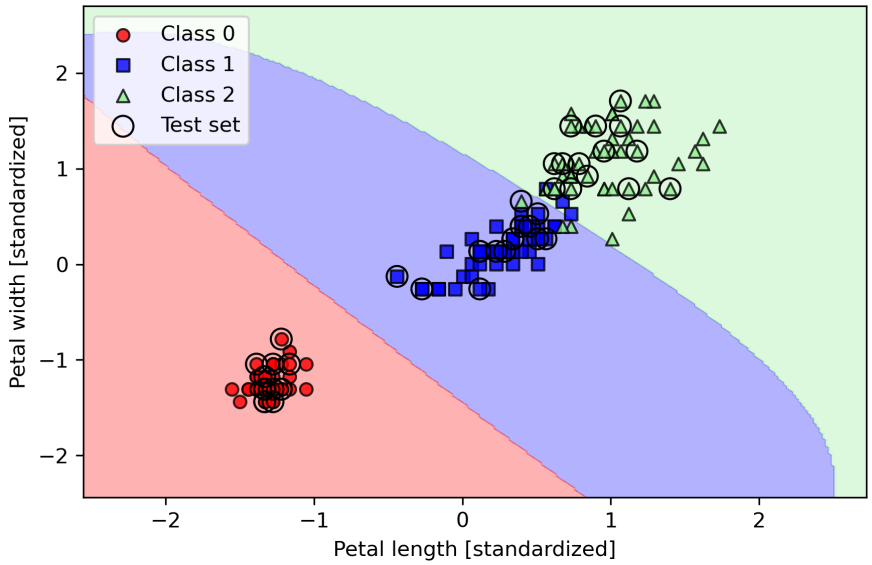


Figure 3.16: The decision boundaries on the Iris dataset using an RBF kernel SVM model with a small γ value

Now, let's increase the value of γ and observe the effect on the decision boundary:

```
>>> svm = SVC(kernel='rbf', random_state=1, gamma=100.0, C=1.0)
>>> svm.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std,
...                       y_combined, classifier=svm,
...                       test_idx=range(105,150))
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

In *Figure 3.17*, we can now see that the decision boundary around the classes `0` and `1` is much tighter using a relatively large value of γ:

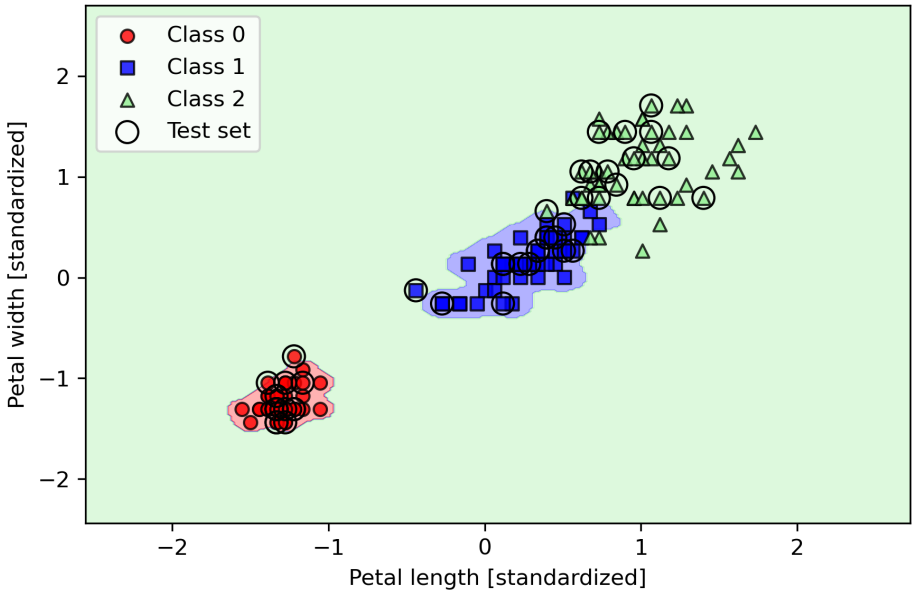


Figure 3.17: The decision boundaries on the Iris dataset using an RBF kernel SVM model with a large γ value

Although the model fits the training dataset very well, such a classifier will likely have a high generalization error on unseen data. This illustrates that the γ parameter also plays an important role in controlling overfitting or variance when the algorithm is too sensitive to fluctuations in the training dataset.

Decision tree learning

Decision tree classifiers are attractive models if we care about interpretability. As the name "decision tree" suggests, we can think of this model as breaking down our data by making a decision based on asking a series of questions.

Let's consider the following example in which we use a decision tree to decide upon an activity on a particular day:

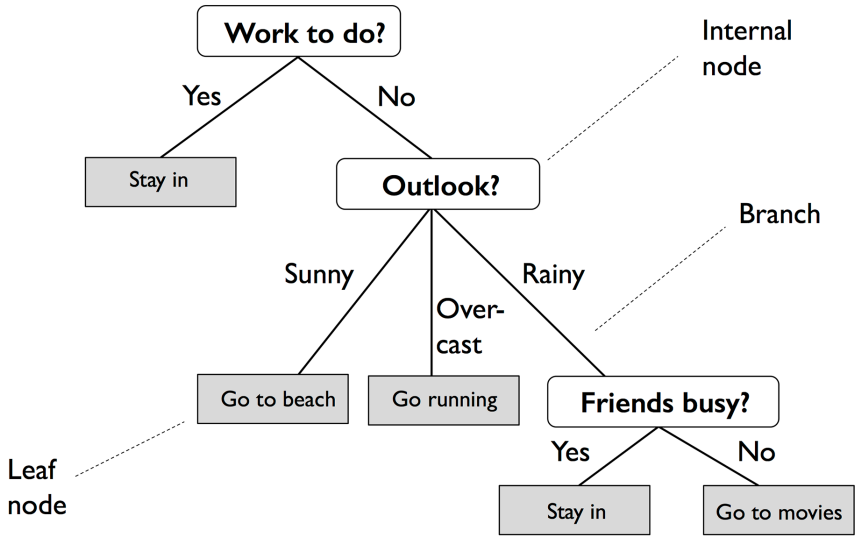


Figure 3.18: An example of a decision tree

Based on the features in our training dataset, the decision tree model learns a series of questions to infer the class labels of the examples. Although *Figure 3.18* illustrates the concept of a decision tree based on categorical variables, the same concept applies if our features are real numbers, like in the Iris dataset. For example, we could simply define a cut-off value along the **sepal width** feature axis and ask a binary question: "Is the sepal width ≥ 2.8 cm?"

Using the decision algorithm, we start at the tree root and split the data on the feature that results in the largest **information gain** (**IG**), which will be explained in more detail in the following section. In an iterative process, we can then repeat this splitting procedure at each child node until the leaves are pure. This means that the training examples at each node all belong to the same class. In practice, this can result in a very deep tree with many nodes, which can easily lead to overfitting. Thus, we typically want to **prune** the tree by setting a limit for the maximum depth of the tree.

Maximizing IG – getting the most bang for your buck

To split the nodes at the most informative features, we need to define an objective function to optimize via the tree learning algorithm. Here, our objective function is to maximize the IG at each split, which we define as follows:

$$IG(D_p, f) = I(D_p) - \sum_{j=1}^{m} \frac{N_j}{N_p} I(D_j)$$

Here, f is the feature to perform the split; D_p and D_j are the dataset of the parent and jth child node; I is our **impurity** measure; N_p is the total number of training examples at the parent node; and N_j is the number of examples in the jth child node. As we can see, the information gain is simply the difference between the impurity of the parent node and the sum of the child node impurities—the lower the impurities of the child nodes, the larger the information gain. However, for simplicity and to reduce the combinatorial search space, most libraries (including scikit-learn) implement binary decision trees. This means that each parent node is split into two child nodes, D_{left} and D_{right}:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right})$$

The three impurity measures or splitting criteria that are commonly used in binary decision trees are **Gini impurity** (I_G), **entropy** (I_H), and the **classification error** (I_E). Let's start with the definition of entropy for all **non-empty** classes ($p(i|t) \neq 0$):

$$I_H(t) = -\sum_{i=1}^{c} p(i|t) \log_2 p(i|t)$$

Here, $p(i|t)$ is the proportion of the examples that belong to class i for a particular node, t. The entropy is therefore 0 if all examples at a node belong to the same class, and the entropy is maximal if we have a uniform class distribution. For example, in a binary class setting, the entropy is 0 if $p(i=1|t) = 1$ or $p(i=0|t) = 0$. If the classes are distributed uniformly with $p(i=1|t) = 0.5$ and $p(i=0|t) = 0.5$, the entropy is 1. Therefore, we can say that the entropy criterion attempts to maximize the mutual information in the tree.

To provide a visual intuition, let us visualize the entropy values for different class distributions via the following code:

```
>>> def entropy(p):
...     return - p * np.log2(p) - (1 - p) * np.log2((1 - p))
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> plt.ylabel('Entropy')
>>> plt.xlabel('Class-membership probability p(i=1)')
>>> plt.plot(x, ent)
>>> plt.show()
```

Figure 3.19 below shows the plot produced by the preceding code:

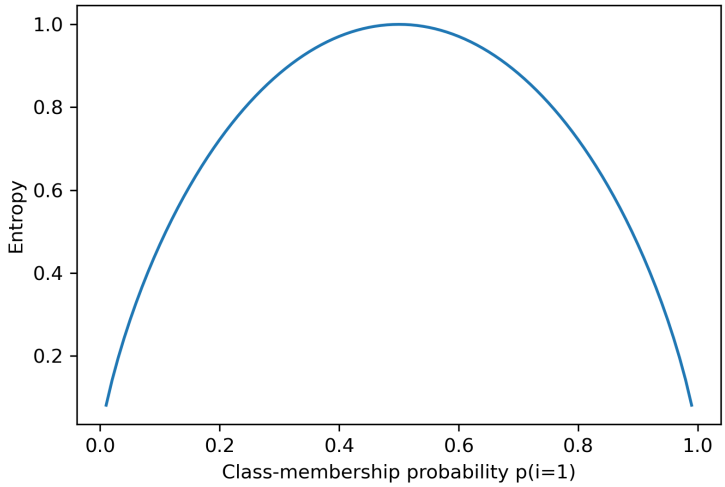


Figure 3.19: Entropy values for different class-membership probabilities

The Gini impurity can be understood as a criterion to minimize the probability of misclassification:

$$I_G(t) = \sum_{i=1}^{c} p(i|t)\,(1 - p(i|t)) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

Similar to entropy, the Gini impurity is maximal if the classes are perfectly mixed, for example, in a binary class setting ($c = 2$):

$$I_G(t) = 1 - \sum_{i=1}^{c} 0.5^2 = 0.5$$

However, in practice, both the Gini impurity and entropy typically yield very similar results, and it is often not worth spending much time on evaluating trees using different impurity criteria rather than experimenting with different pruning cut-offs. In fact, as you will see later in *Figure 3.21*, both the Gini impurity and entropy have a similar shape.

Another impurity measure is the classification error:

$$I_E(t) = 1 - \max\{p(i|t)\}$$

This is a useful criterion for pruning, but not recommended for growing a decision tree, since it is less sensitive to changes in the class probabilities of the nodes. We can illustrate this by looking at the two possible splitting scenarios shown in *Figure 3.20*:

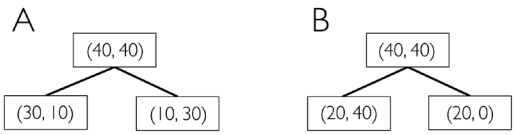


Figure 3.20: Decision tree data splits

We start with a dataset, D_p, at the parent node, which consists of 40 examples from class 1 and 40 examples from class 2 that we split into two datasets, D_{left} and D_{right}. The information gain using the classification error as a splitting criterion would be the same ($IG_E = 0.25$) in both scenarios, *A* and *B*:

$$I_E(D_p) = 1 - 0.5 = 0.5$$

$$A: \quad I_E(D_{left}) = 1 - \frac{3}{4} = 0.25$$

$$A: \quad I_E(D_{right}) = 1 - \frac{3}{4} = 0.25$$

$$A: \quad IG_E = 0.5 - \frac{4}{8}0.25 - \frac{4}{8}0.25 = 0.25$$

$$B: \quad I_E(D_{left}) = 1 - \frac{4}{6} = \frac{1}{3}$$

$$B: \quad I_E(D_{right}) = 1 - 1 = 0$$

$$B: \quad IG_E = 0.5 - \frac{6}{8} \times \frac{1}{3} - 0 = 0.25$$

However, the Gini impurity would favor the split in scenario *B* ($IG_G = 0.1\overline{6}$) over scenario *A* ($IG_G = 0.125$), which is indeed purer:

$$I_G(D_p) = 1 - (0.5^2 + 0.5^2) = 0.5$$

$$A: \quad I_G(D_{left}) = 1 - \left(\left(\frac{3}{4}\right)^2 + \left(\frac{1}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A: \quad I_G(D_{right}) = 1 - \left(\left(\frac{1}{4}\right)^2 + \left(\frac{3}{4}\right)^2\right) = \frac{3}{8} = 0.375$$

$$A: \quad IG_G = 0.5 - \frac{4}{8}0.375 - \frac{4}{8}0.375 = 0.125$$

$$B: \quad I_G(D_{left}) = 1 - \left(\left(\frac{2}{6}\right)^2 + \left(\frac{4}{6}\right)^2\right) = \frac{4}{9} = 0.\overline{4}$$

$$B: \quad I_G(D_{right}) = 1 - (1^2 + 0^2) = 0$$

$$B: \quad IG_G = 0.5 - \frac{6}{8}0.\overline{4} - 0 = 0.1\overline{6}$$

Similarly, the entropy criterion would also favor scenario B ($IG_H = 0.31$) over scenario A ($IG_H = 0.19$):

$$I_H(D_p) = -(0.5\log_2(0.5) + 0.5\log_2(0.5)) = 1$$

$$A: \quad I_H(D_{left}) = -\left(\frac{3}{4}\log_2\left(\frac{3}{4}\right) + \frac{1}{4}\log_2\left(\frac{1}{4}\right)\right) = 0.81$$

$$A: \quad I_H(D_{right}) = -\left(\frac{1}{4}\log_2\left(\frac{1}{4}\right) + \frac{3}{4}\log_2\left(\frac{3}{4}\right)\right) = 0.81$$

$$A: \quad IG_H = 1 - \frac{4}{8}0.81 - \frac{4}{8}0.81 = 0.19$$

$$B: \quad I_H(D_{left}) = -\left(\frac{2}{6}\log_2\left(\frac{2}{6}\right) + \frac{4}{6}\log_2\left(\frac{4}{6}\right)\right) = 0.92$$

$$B: \quad I_H(D_{right}) = 0$$

$$B: \quad IG_H = 1 - \frac{6}{8}0.92 - 0 = 0.31$$

For a more visual comparison of the three different impurity criteria that we discussed previously, let's plot the impurity indices for the probability range [0, 1] for class 1. Note that we will also add a scaled version of the entropy (entropy / 2) to observe that the Gini impurity is an intermediate measure between entropy and the classification error. The code is as follows:

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> def gini(p):
...     return p*(1 - p) + (1 - p)*(1 - (1-p))
>>> def entropy(p):
...     return - p*np.log2(p) - (1 - p)*np.log2((1 - p))
>>> def error(p):
...     return 1 - np.max([p, 1 - p])
>>> x = np.arange(0.0, 1.0, 0.01)
>>> ent = [entropy(p) if p != 0 else None for p in x]
>>> sc_ent = [e*0.5 if e else None for e in ent]
>>> err = [error(i) for i in x]
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
```

```
>>> for i, lab, ls, c, in zip([ent, sc_ent, gini(x), err],
...                           ['Entropy', 'Entropy (scaled)',
...                            'Gini impurity',
...                            'Misclassification error'],
...                           ['-', '-', '--', '-.'],
...                           ['black', 'lightgray',
...                            'red', 'green', 'cyan']):
...     line = ax.plot(x, i, label=lab,
...                    linestyle=ls, lw=2, color=c)
>>> ax.legend(loc='upper center', bbox_to_anchor=(0.5, 1.15),
...           ncol=5, fancybox=True, shadow=False)
>>> ax.axhline(y=0.5, linewidth=1, color='k', linestyle='--')
>>> ax.axhline(y=1.0, linewidth=1, color='k', linestyle='--')
>>> plt.ylim([0, 1.1])
>>> plt.xlabel('p(i=1)')
>>> plt.ylabel('impurity index')
>>> plt.show()
```

The plot produced by the preceding code example is as follows:

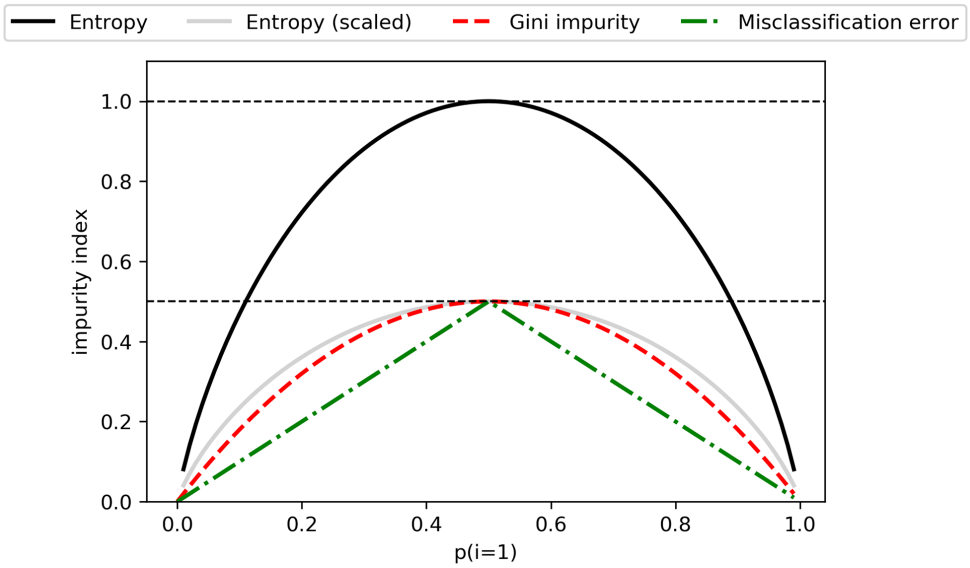


Figure 3.21: The different impurity indices for different class-membership probabilities between 0 and 1

Building a decision tree

Decision trees can build complex decision boundaries by dividing the feature space into rectangles. However, we have to be careful since the deeper the decision tree, the more complex the decision boundary becomes, which can easily result in overfitting. Using scikit-learn, we will now train a decision tree with a maximum depth of 4, using the Gini impurity as a criterion for impurity.

Although feature scaling may be desired for visualization purposes, note that feature scaling is not a requirement for decision tree algorithms. The code is as follows:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> tree_model = DecisionTreeClassifier(criterion='gini',
...                                     max_depth=4,
...                                     random_state=1)
>>> tree_model.fit(X_train, y_train)
>>> X_combined = np.vstack((X_train, X_test))
>>> y_combined = np.hstack((y_train, y_test))
>>> plot_decision_regions(X_combined,
...                       y_combined,
...                       classifier=tree_model,
...                       test_idx=range(105, 150))
>>> plt.xlabel('Petal length [cm]')
>>> plt.ylabel('Petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After executing the code example, we get the typical axis-parallel decision boundaries of the decision tree:

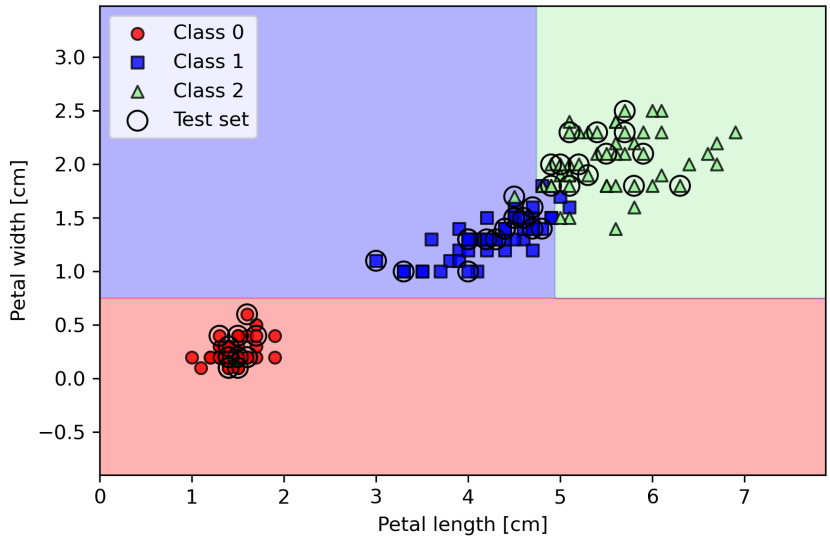


Figure 3.22: The decision boundaries of the Iris data using a decision tree

A nice feature in scikit-learn is that it allows us to readily visualize the decision tree model after training via the following code:

```
>>> from sklearn import tree
>>> feature_names = ['Sepal length', 'Sepal width',
...                  'Petal length', 'Petal width']
>>> tree.plot_tree(tree_model,
...                feature_names=feature_names,
...                filled=True)
>>> plt.show()
```

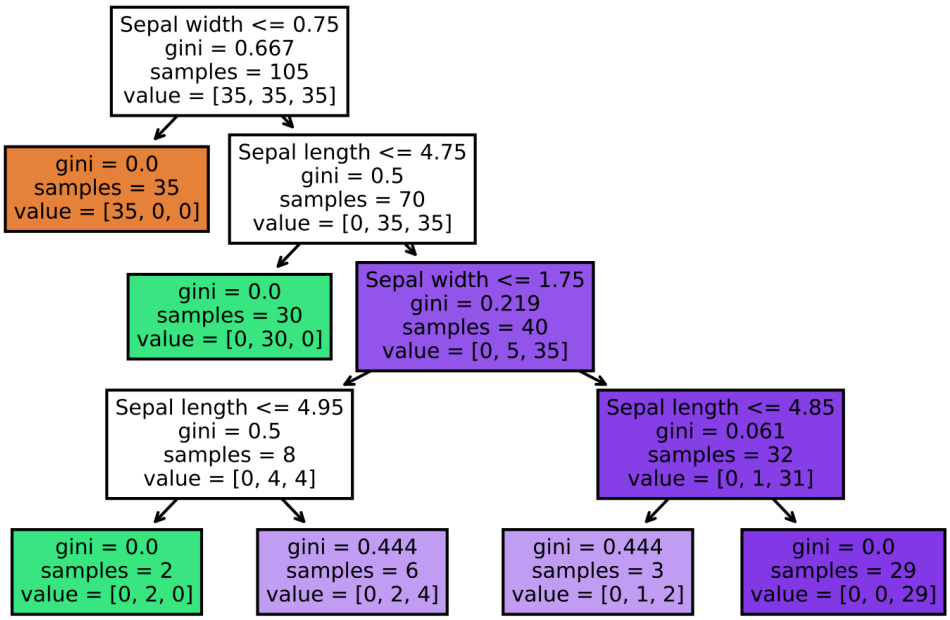


Figure 3.23: A decision tree model fit to the Iris dataset

Setting `filled=True` in the `plot_tree` function we called colors the nodes by the majority class label at that node. There are many additional options available, which you can find in the documentation at `https://scikit-learn.org/stable/modules/generated/sklearn.tree.plot_tree.html`.

Looking at the decision tree figure, we can now nicely trace back the splits that the decision tree determined from our training dataset. Regarding the feature splitting criterion at each node, note that the branches to the left correspond to "True" and branches to the right correspond to "False."

Looking at the root node, it starts with 105 examples at the top. The first split uses a sepal width cut-off ≤ 0.75 cm for splitting the root node into two child nodes with 35 examples (left child node) and 70 examples (right child node). After the first split, we can see that the left child node is already pure and only contains examples from the `Iris-setosa` class (Gini impurity = 0). The further splits on the right are then used to separate the examples from the `Iris-versicolor` and `Iris-virginica` class.

Looking at this tree, and the decision region plot of the tree, we can see that the decision tree does a very good job of separating the flower classes. Unfortunately, scikit-learn currently does not implement functionality to manually post-prune a decision tree. However, we could go back to our previous code example, change the `max_depth` of our decision tree to `3`, and compare it to our current model, but we leave this as an exercise for the interested reader.

Alternatively, scikit-learn provides an automatic cost complexity post-pruning procedure for decision trees. Interested readers can find more information about this more advanced topic in the following tutorial: `https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html`.

Combining multiple decision trees via random forests

Ensemble methods have gained huge popularity in applications of machine learning during the last decade due to their good classification performance and robustness toward overfitting. While we are going to cover different ensemble methods, including **bagging** and **boosting**, later in *Chapter 7*, *Combining Different Models for Ensemble Learning*, let's discuss the decision tree-based **random forest** algorithm, which is known for its good scalability and ease of use. A random forest can be considered as an **ensemble** of decision trees. The idea behind a random forest is to average multiple (deep) decision trees that individually suffer from high variance to build a more robust model that has a better generalization performance and is less susceptible to overfitting. The random forest algorithm can be summarized in four simple steps:

1. Draw a random **bootstrap** sample of size *n* (randomly choose *n* examples from the training dataset with replacement).
2. Grow a decision tree from the bootstrap sample. At each node:
    a. Randomly select *d* features without replacement.
    b. Split the node using the feature that provides the best split according to the objective function, for instance, maximizing the information gain.
3. Repeat *steps 1-2 k* times.
4. Aggregate the prediction by each tree to assign the class label by **majority vote.** Majority voting will be discussed in more detail in *Chapter 7*.

We should note one slight modification in *step 2* when we are training the individual decision trees: instead of evaluating all features to determine the best split at each node, we only consider a random subset of those.

Sampling with and without replacement

In case you are not familiar with the terms sampling "with" and "without" replacement, let's walk through a simple thought experiment. Let's assume that we are playing a lottery game where we randomly draw numbers from an urn. We start with an urn that holds five unique numbers, 0, 1, 2, 3, and 4, and we draw exactly one number on each turn. In the first round, the chance of drawing a particular number from the urn would be 1/5. Now, in sampling without replacement, we do not put the number back into the urn after each turn. Consequently, the probability of drawing a particular number from the set of remaining numbers in the next round depends on the previous round. For example, if we have a remaining set of numbers 0, 1, 2, and 4, the chance of drawing number 0 would become 1/4 in the next turn.

However, in random sampling with replacement, we always return the drawn number to the urn so that the probability of drawing a particular number at each turn does not change; we can draw the same number more than once. In other words, in sampling *with* replacement, the samples (numbers) are independent and have a covariance of zero. For example, the results from five rounds of drawing random numbers could look like this:

- Random sampling without replacement: 2, 1, 3, 4, 0
- Random sampling with replacement: 1, 3, 3, 4, 1

Although random forests don't offer the same level of interpretability as decision trees, a big advantage of random forests is that we don't have to worry so much about choosing good hyperparameter values. We typically don't need to prune the random forest since the ensemble model is quite robust to noise from averaging the predictions among the individual decision trees. The only parameter that we need to care about in practice is the number of trees, *k*, (*step 3*) that we choose for the random forest. Typically, the larger the number of trees, the better the performance of the random forest classifier at the expense of an increased computational cost.

Although it is less common in practice, other hyperparameters of the random forest classifier that can be optimized—using techniques that we will discuss in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*—are the size, n, of the bootstrap sample (*step 1*) and the number of features, d, that are randomly chosen for each split (*step 2a*), respectively. Via the sample size, n, of the bootstrap sample, we control the bias-variance tradeoff of the random forest.

Decreasing the size of the bootstrap sample increases the diversity among the individual trees since the probability that a particular training example is included in the bootstrap sample is lower. Thus, shrinking the size of the bootstrap samples may increase the *randomness* of the random forest, and it can help to reduce the effect of overfitting. However, smaller bootstrap samples typically result in a lower overall performance of the random forest and a small gap between training and test performance, but a low test performance overall. Conversely, increasing the size of the bootstrap sample may increase the degree of overfitting. Because the bootstrap samples, and consequently the individual decision trees, become more similar to one another, they learn to fit the original training dataset more closely.

In most implementations, including the `RandomForestClassifier` implementation in scikit-learn, the size of the bootstrap sample is chosen to be equal to the number of training examples in the original training dataset, which usually provides a good bias-variance tradeoff. For the number of features, d, at each split, we want to choose a value that is smaller than the total number of features in the training dataset. A reasonable default that is used in scikit-learn and other implementations is $d = \sqrt{m}$, where m is the number of features in the training dataset.

Conveniently, we don't have to construct the random forest classifier from individual decision trees by ourselves because there is already an implementation in scikit-learn that we can use:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> forest = RandomForestClassifier(n_estimators=25,
...                                 random_state=1,
...                                 n_jobs=2)
>>> forest.fit(X_train, y_train)
>>> plot_decision_regions(X_combined, y_combined,
...                       classifier=forest, test_idx=range(105,150))
>>> plt.xlabel('Petal length [cm]')
>>> plt.ylabel('Petal width [cm]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

After executing the preceding code, we should see the decision regions formed by the ensemble of trees in the random forest, as shown in *Figure 3.24*:

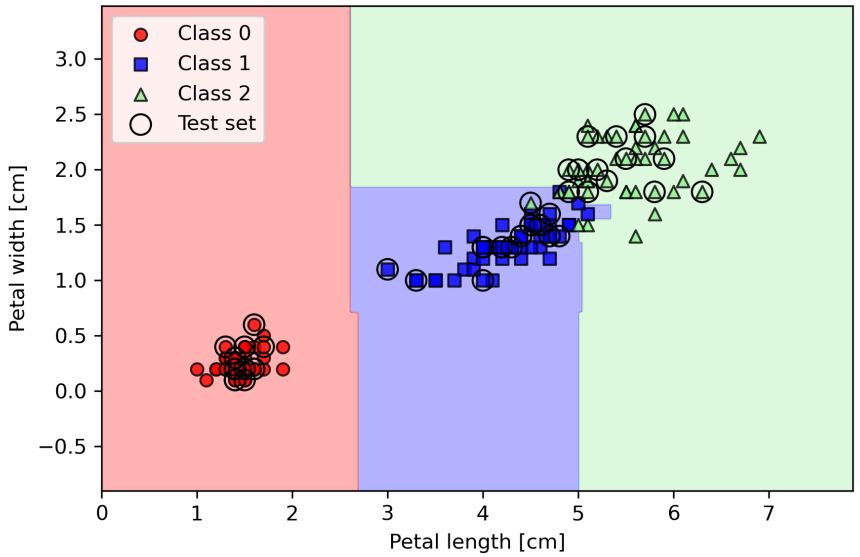


Figure 3.24: Decision boundaries on the Iris dataset using a random forest

Using the preceding code, we trained a random forest from 25 decision trees via the `n_estimators` parameter. By default, it uses the Gini impurity measure as a criterion to split the nodes. Although we are growing a very small random forest from a very small training dataset, we used the `n_jobs` parameter for demonstration purposes, which allows us to parallelize the model training using multiple cores of our computer (here, two cores). If you encounter errors with this code, your computer may not support multiprocessing. You can omit the `n_jobs` parameter or set it to `n_jobs=None`.

K-nearest neighbors — a lazy learning algorithm

The last supervised learning algorithm that we want to discuss in this chapter is the **k-nearest neighbor** (**KNN**) classifier, which is particularly interesting because it is fundamentally different from the learning algorithms that we have discussed so far.

KNN is a typical example of a **lazy learner**. It is called "lazy" not because of its apparent simplicity, but because it doesn't learn a discriminative function from the training data but memorizes the training dataset instead.

Parametric versus non-parametric models

Machine learning algorithms can be grouped into parametric and non-parametric models. Using parametric models, we estimate parameters from the training dataset to learn a function that can classify new data points without requiring the original training dataset anymore. Typical examples of parametric models are the perceptron, logistic regression, and the linear SVM. In contrast, non-parametric models can't be characterized by a fixed set of parameters, and the number of parameters changes with the amount of training data. Two examples of non-parametric models that we have seen so far are the decision tree classifier/random forest and the kernel (but not linear) SVM.

KNN belongs to a subcategory of non-parametric models described as instance-based learning. Models based on instance-based learning are characterized by memorizing the training dataset, and lazy learning is a special case of instance-based learning that is associated with no (zero) cost during the learning process.

The KNN algorithm itself is fairly straightforward and can be summarized by the following steps:

1. Choose the number of k and a distance metric
2. Find the k-nearest neighbors of the data record that we want to classify
3. Assign the class label by majority vote

Figure 3.25 illustrates how a new data point (?) is assigned the triangle class label based on majority voting among its five nearest neighbors:

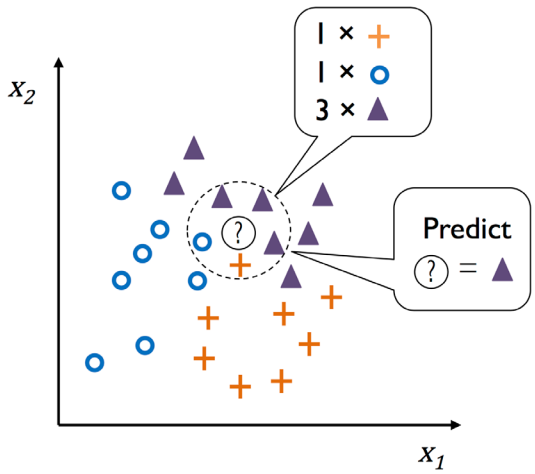


Figure 3.25: How k-nearest neighbors works

Based on the chosen distance metric, the KNN algorithm finds the k examples in the training dataset that are closest (most similar) to the point that we want to classify. The class label of the data point is then determined by a majority vote among its k nearest neighbors.

Advantages and disadvantages of memory-based approaches

The main advantage of such a memory-based approach is that the classifier immediately adapts as we collect new training data. However, the downside is that the computational complexity for classifying new examples grows linearly with the number of examples in the training dataset in the worst-case scenario—unless the dataset has very few dimensions (features) and the algorithm has been implemented using efficient data structures for querying the training data more effectively. Such data structures include k-d tree (`https://en.wikipedia.org/wiki/K-d_tree`) and ball tree (`https://en.wikipedia.org/wiki/Ball_tree`), which are both supported in scikit-learn. Furthermore, next to computational costs for querying data, large datasets can also be problematic in terms of limited storage capacities.

However, in many cases when we are working with relatively small to medium-sized datasets, memory-based methods can provide good predictive and computational performance and are thus a good choice for approaching many real-world problems. Recent examples of using nearest neighbor methods include predicting properties of pharmaceutical drug targets (*Machine Learning to Identify Flexibility Signatures of Class A GPCR Inhibition*, Biomolecules, 2020, Joe Bemister-Buffington, Alex J. Wolf, Sebastian Raschka, and Leslie A. Kuhn, `https://www.mdpi.com/2218-273X/10/3/454`) and state-of-the-art language models (*Efficient Nearest Neighbor Language Models*, 2021, Junxian He, Graham Neubig, and Taylor Berg-Kirkpatrick, `https://arxiv.org/abs/2109.04212`).

By executing the following code, we will now implement a KNN model in scikit-learn using a Euclidean distance metric:

```
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5, p=2,
...                            metric='minkowski')
>>> knn.fit(X_train_std, y_train)
>>> plot_decision_regions(X_combined_std, y_combined,
...                       classifier=knn, test_idx=range(105,150))
>>> plt.xlabel('Petal length [standardized]')
>>> plt.ylabel('Petal width [standardized]')
>>> plt.legend(loc='upper left')
>>> plt.tight_layout()
>>> plt.show()
```

By specifying five neighbors in the KNN model for this dataset, we obtain a relatively smooth decision boundary, as shown in *Figure 3.26*:

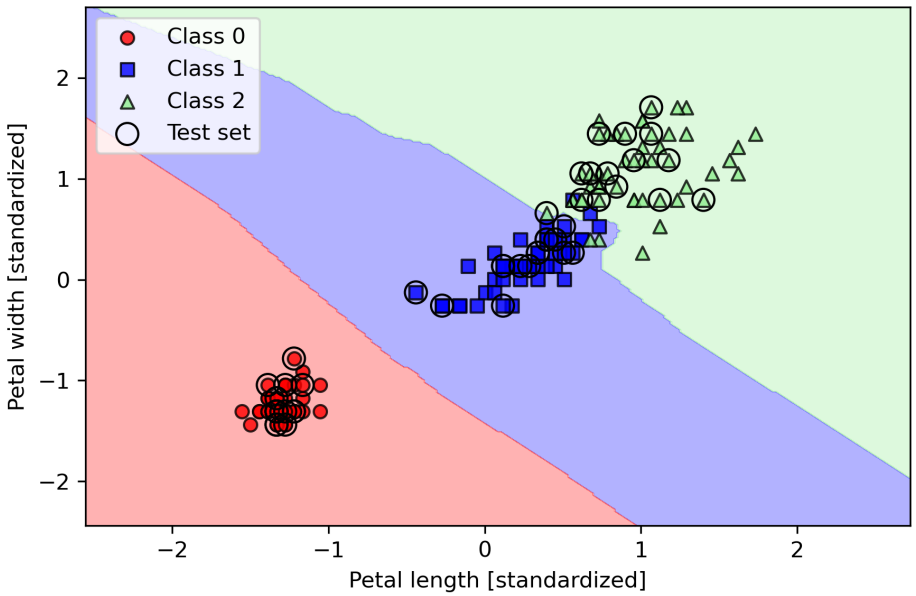


Figure 3.26: k-nearest neighbors' decision boundaries on the Iris dataset

Resolving ties

In the case of a tie, the scikit-learn implementation of the KNN algorithm will prefer the neighbors with a closer distance to the data record to be classified. If the neighbors have similar distances, the algorithm will choose the class label that comes first in the training dataset.

The *right* choice of k is crucial to finding a good balance between overfitting and underfitting. We also have to make sure that we choose a distance metric that is appropriate for the features in the dataset. Often, a simple Euclidean distance measure is used for real-value examples, for example, the flowers in our Iris dataset, which have features measured in centimeters. However, if we are using a Euclidean distance measure, it is also important to standardize the data so that each feature contributes equally to the distance. The `minkowski` distance that we used in the previous code is just a generalization of the Euclidean and Manhattan distance, which can be written as follows:

$$d\left(\boldsymbol{x}^{(i)}, \boldsymbol{x}^{(j)}\right) = \sqrt[p]{\sum_k \left|x_k^{(i)} - x_k^{(j)}\right|^p}$$

It becomes the Euclidean distance if we set the parameter `p=2` or the Manhattan distance at `p=1`. Many other distance metrics are available in scikit-learn and can be provided to the `metric` parameter. They are listed at `https://scikit-learn.org/stable/modules/generated/sklearn.metrics.DistanceMetric.html`.

Lastly, it is important to mention that KNN is very susceptible to overfitting due to the **curse of dimensionality**. The curse of dimensionality describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset. We can think of even the closest neighbors as being too far away in a high-dimensional space to give a good estimate.

We discussed the concept of regularization in the section about logistic regression as one way to avoid overfitting. However, in models where regularization is not applicable, such as decision trees and KNN, we can use feature selection and dimensionality reduction techniques to help us to avoid the curse of dimensionality. This will be discussed in more detail in the next two chapters.

Alternative machine learning implementations with GPU support

When working with large datasets, running k-nearest neighbors or fitting random forests with many estimators can require substantial computing resources and processing time. If you have a computer equipped with an NVIDIA GPU that is compatible with recent versions of NVIDIA's CUDA library, we recommend considering the RAPIDS ecosystem (`https://docs.rapids.ai/api`). For instance, RAPIDS' cuML (`https://docs.rapids.ai/api/cuml/stable/`) library implements many of scikit-learn's machine learning algorithms with GPU support to accelerate the processing speeds. You can find an introduction to cuML at `https://docs.rapids.ai/api/cuml/stable/estimator_intro.html`. If you are interested in learning more about the RAPIDS ecosystem, please also see the freely accessible journal article that we wrote in collaboration with the RAPIDS team: *Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence* (`https://www.mdpi.com/2078-2489/11/4/193`).

Summary

In this chapter, you learned about many different machine learning algorithms that are used to tackle linear and nonlinear problems. You have seen that decision trees are particularly attractive if we care about interpretability. Logistic regression is not only a useful model for online learning via SGD, but also allows us to predict the probability of a particular event.

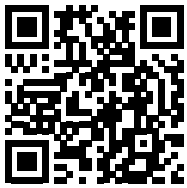
Although SVMs are powerful linear models that can be extended to nonlinear problems via the kernel trick, they have many parameters that have to be tuned in order to make good predictions. In contrast, ensemble methods, such as random forests, don't require much parameter tuning and don't overfit as easily as decision trees, which makes them attractive models for many practical problem domains. The KNN classifier offers an alternative approach to classification via lazy learning that allows us to make predictions without any model training, but with a more computationally expensive prediction step.

However, even more important than the choice of an appropriate learning algorithm is the available data in our training dataset. No algorithm will be able to make good predictions without informative and discriminatory features.

In the next chapter, we will discuss important topics regarding the preprocessing of data, feature selection, and dimensionality reduction, which means that we will need to build powerful machine learning models. Later, in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will see how we can evaluate and compare the performance of our models and learn useful tricks to fine-tune the different algorithms.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`



4

Building Good Training Datasets – Data Preprocessing

The quality of the data and the amount of useful information that it contains are key factors that determine how well a machine learning algorithm can learn. Therefore, it is absolutely critical to ensure that we examine and preprocess a dataset before we feed it to a machine learning algorithm. In this chapter, we will discuss the essential data preprocessing techniques that will help us to build good machine learning models.

The topics that we will cover in this chapter are as follows:

- Removing and imputing missing values from the dataset
- Getting categorical data into shape for machine learning algorithms
- Selecting relevant features for the model construction

Dealing with missing data

It is not uncommon in real-world applications for our training examples to be missing one or more values for various reasons. There could have been an error in the data collection process, certain measurements may not be applicable, or particular fields could have been simply left blank in a survey, for example. We typically see missing values as blank spaces in our data table or as placeholder strings such as `NaN`, which stands for "not a number," or `NULL` (a commonly used indicator of unknown values in relational databases). Unfortunately, most computational tools are unable to handle such missing values or will produce unpredictable results if we simply ignore them. Therefore, it is crucial that we take care of those missing values before we proceed with further analyses.

In this section, we will work through several practical techniques for dealing with missing values by removing entries from our dataset or imputing missing values from other training examples and features.

Identifying missing values in tabular data

Before we discuss several techniques for dealing with missing values, let's create a simple example `DataFrame` from a **comma-separated values** (**CSV**) file to get a better grasp of the problem:

```
>>> import pandas as pd
>>> from io import StringIO
>>> csv_data = \
... '''A,B,C,D
... 1.0,2.0,3.0,4.0
... 5.0,6.0,,8.0
... 10.0,11.0,12.0,'''
>>> # If you are using Python 2.7, you need
>>> # to convert the string to unicode:
>>> # csv_data = unicode(csv_data)
>>> df = pd.read_csv(StringIO(csv_data))
>>> df
      A     B     C    D
0   1.0   2.0   3.0  4.0
1   5.0   6.0   NaN  8.0
2  10.0  11.0  12.0  NaN
```

Using the preceding code, we read CSV-formatted data into a pandas `DataFrame` via the `read_csv` function and noticed that the two missing cells were replaced by `NaN`. The `StringIO` function in the preceding code example was simply used for the purposes of illustration. It allowed us to read the string assigned to `csv_data` into a pandas `DataFrame` as if it was a regular CSV file on our hard drive.

For a larger `DataFrame`, it can be tedious to look for missing values manually; in this case, we can use the `isnull` method to return a `DataFrame` with Boolean values that indicate whether a cell contains a numeric value (`False`) or if data is missing (`True`). Using the `sum` method, we can then return the number of missing values per column as follows:

```
>>> df.isnull().sum()
A    0
B    0
C    1
D    1
dtype: int64
```

This way, we can count the number of missing values per column; in the following subsections, we will take a look at different strategies for how to deal with this missing data.

Convenient data handling with pandas' DataFrame

Although scikit-learn was originally developed for working with NumPy arrays only, it can sometimes be more convenient to preprocess data using pandas' `DataFrame`. Nowadays, most scikit-learn functions support `DataFrame` objects as inputs, but since NumPy array handling is more mature in the scikit-learn API, it is recommended to use NumPy arrays when possible. Note that you can always access the underlying NumPy array of a `DataFrame` via the `values` attribute before you feed it into a scikit-learn estimator:

```
>>> df.values
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  nan,   8.],
       [ 10.,  11.,  12.,  nan]])
```

Eliminating training examples or features with missing values

One of the easiest ways to deal with missing data is simply to remove the corresponding features (columns) or training examples (rows) from the dataset entirely; rows with missing values can easily be dropped via the `dropna` method:

```
>>> df.dropna(axis=0)
     A    B    C    D
0   1.0  2.0  3.0  4.0
```

Similarly, we can drop columns that have at least one `NaN` in any row by setting the `axis` argument to `1`:

```
>>> df.dropna(axis=1)
     A     B
0   1.0   2.0
1   5.0   6.0
2  10.0  11.0
```

The `dropna` method supports several additional parameters that can come in handy:

```
>>> # only drop rows where all columns are NaN
>>> # (returns the whole array here since we don't
>>> # have a row with all values NaN)
>>> df.dropna(how='all')
     A     B     C     D
0   1.0   2.0   3.0   4.0
1   5.0   6.0   NaN   8.0
2  10.0  11.0  12.0   NaN
```

```
>>> # drop rows that have fewer than 4 real values
>>> df.dropna(thresh=4)
     A     B     C     D
0  1.0   2.0   3.0   4.0
>>> # only drop rows where NaN appear in specific columns (here: 'C')
>>> df.dropna(subset=['C'])
     A     B     C     D
0  1.0   2.0   3.0   4.0
2 10.0  11.0  12.0   NaN
```

Although the removal of missing data seems to be a convenient approach, it also comes with certain disadvantages; for example, we may end up removing too many samples, which will make a reliable analysis impossible. Or, if we remove too many feature columns, we will run the risk of losing valuable information that our classifier needs to discriminate between classes. In the next section, we will look at one of the most commonly used alternatives for dealing with missing values: interpolation techniques.

Imputing missing values

Often, the removal of training examples or dropping of entire feature columns is simply not feasible, because we might lose too much valuable data. In this case, we can use different interpolation techniques to estimate the missing values from the other training examples in our dataset. One of the most common interpolation techniques is **mean imputation**, where we simply replace the missing value with the mean value of the entire feature column. A convenient way to achieve this is by using the `SimpleImputer` class from scikit-learn, as shown in the following code:

```
>>> from sklearn.impute import SimpleImputer
>>> import numpy as np
>>> imr = SimpleImputer(missing_values=np.nan, strategy='mean')
>>> imr = imr.fit(df.values)
>>> imputed_data = imr.transform(df.values)
>>> imputed_data
array([[  1.,   2.,   3.,   4.],
       [  5.,   6.,  7.5,   8.],
       [ 10.,  11.,  12.,   6.]])
```

Here, we replaced each `NaN` value with the corresponding mean, which is separately calculated for each feature column. Other options for the `strategy` parameter are `median` or `most_frequent`, where the latter replaces the missing values with the most frequent values. This is useful for imputing categorical feature values, for example, a feature column that stores an encoding of color names, such as red, green, and blue. We will encounter examples of such data later in this chapter.

Alternatively, an even more convenient way to impute missing values is by using pandas' `fillna` method and providing an imputation method as an argument. For example, using pandas, we could achieve the same mean imputation directly in the `DataFrame` object via the following command:

```
>>> df.fillna(df.mean())
```

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.5	8.0
2	10.0	11.0	12.0	6.0

Figure 4.1: Replacing missing values in data with the mean

Additional imputation methods for missing data

For additional imputation techniques, including the `KNNImputer` based on a k-nearest neighbors approach to impute missing features by nearest neighbors, we recommend the scikit-learn imputation documentation at `https://scikit-learn.org/stable/modules/impute.html`.

Understanding the scikit-learn estimator API

In the previous section, we used the `SimpleImputer` class from scikit-learn to impute missing values in our dataset. The `SimpleImputer` class is part of the so-called **transformer** API in scikit-learn, which is used for implementing Python classes related to data transformation. (Please note that the scikit-learn transformer API is not to be confused with the transformer architecture that is used in natural language processing, which we will cover in more detail in *Chapter 16*, *Transformers – Improving Natural Language Processing with Attention Mechanisms*.) The two essential methods of those estimators are `fit` and `transform`. The `fit` method is used to learn the parameters from the training data, and the `transform` method uses those parameters to transform the data. Any data array that is to be transformed needs to have the same number of features as the data array that was used to fit the model.

Figure 4.2 illustrates how a scikit-learn transformer instance, fitted on the training data, is used to transform a training dataset as well as a new test dataset:

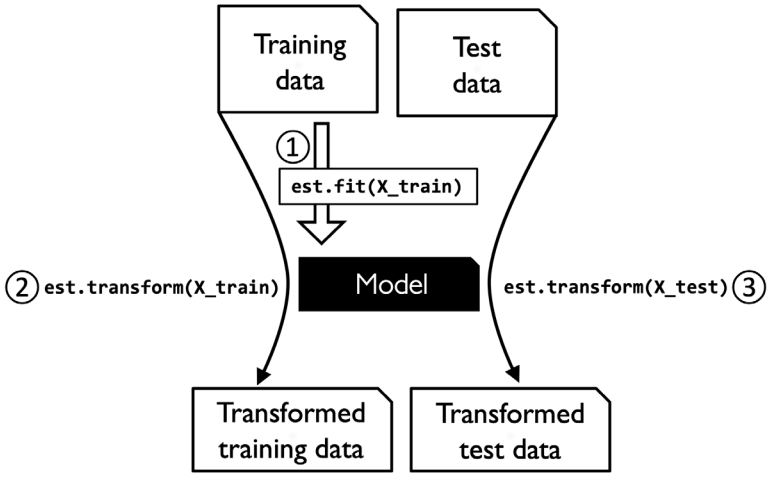


Figure 4.2: Using the scikit-learn API for data transformation

The classifiers that we used in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, belong to the so-called **estimators** in scikit-learn, with an API that is conceptually very similar to the scikit-learn transformer API. Estimators have a `predict` method but can also have a `transform` method, as you will see later in this chapter. As you may recall, we also used the `fit` method to learn the parameters of a model when we trained those estimators for classification. However, in supervised learning tasks, we additionally provide the class labels for fitting the model, which can then be used to make predictions about new, unlabeled data examples via the `predict` method, as illustrated in *Figure 4.3*:

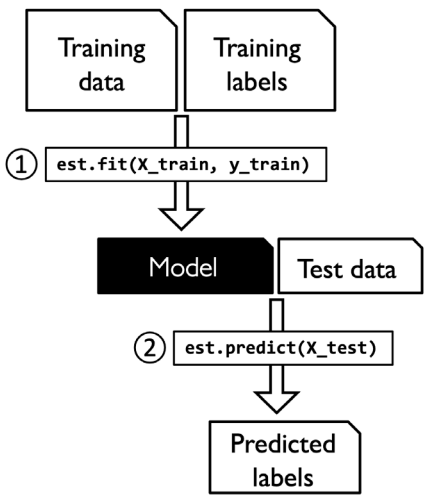


Figure 4.3: Using the scikit-learn API for predictive models such as classifiers

Handling categorical data

So far, we have only been working with numerical values. However, it is not uncommon for real-world datasets to contain one or more categorical feature columns. In this section, we will make use of simple yet effective examples to see how to deal with this type of data in numerical computing libraries.

When we are talking about categorical data, we have to further distinguish between **ordinal** and **nominal** features. Ordinal features can be understood as categorical values that can be sorted or ordered. For example, t-shirt size would be an ordinal feature, because we can define an order: $XL > L > M$. In contrast, nominal features don't imply any order; to continue with the previous example, we could think of t-shirt color as a nominal feature since it typically doesn't make sense to say that, for example, red is larger than blue.

Categorical data encoding with pandas

Before we explore different techniques for handling such categorical data, let's create a new `DataFrame` to illustrate the problem:

```
>>> import pandas as pd
>>> df = pd.DataFrame([
...            ['green', 'M', 10.1, 'class2'],
...            ['red', 'L', 13.5, 'class1'],
...            ['blue', 'XL', 15.3, 'class2']])
>>> df.columns = ['color', 'size', 'price', 'classlabel']
>>> df
    color  size  price  classlabel
0   green     M   10.1      class2
1     red     L   13.5      class1
2    blue    XL   15.3      class2
```

As we can see in the preceding output, the newly created `DataFrame` contains a nominal feature (`color`), an ordinal feature (`size`), and a numerical feature (`price`) column. The class labels (assuming that we created a dataset for a supervised learning task) are stored in the last column. The learning algorithms for classification that we discuss in this book do not use ordinal information in class labels.

Mapping ordinal features

To make sure that the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers. Unfortunately, there is no convenient function that can automatically derive the correct order of the labels of our `size` feature, so we have to define the mapping manually. In the following simple example, let's assume that we know the numerical difference between features, for example, $XL = L + 1 = M + 2$:

```
>>> size_mapping = {'XL': 3,
...                 'L': 2,
...                 'M': 1}
```

```
>>> df['size'] = df['size'].map(size_mapping)
>>> df
    color  size  price  classlabel
0   green     1   10.1      class2
1     red     2   13.5      class1
2    blue     3   15.3      class2
```

If we want to transform the integer values back to the original string representation at a later stage, we can simply define a reverse-mapping dictionary, `inv_size_mapping = {v: k for k, v in size_mapping.items()}`, which can then be used via the pandas `map` method on the transformed feature column and is similar to the `size_mapping` dictionary that we used previously. We can use it as follows:

```
>>> inv_size_mapping = {v: k for k, v in size_mapping.items()}
>>> df['size'].map(inv_size_mapping)
0   M
1   L
2   XL
Name: size, dtype: object
```

Encoding class labels

Many machine learning libraries require that class labels are encoded as integer values. Although most estimators for classification in scikit-learn convert class labels to integers internally, it is considered good practice to provide class labels as integer arrays to avoid technical glitches. To encode the class labels, we can use an approach similar to the mapping of ordinal features discussed previously. We need to remember that class labels are *not* ordinal, and it doesn't matter which integer number we assign to a particular string label. Thus, we can simply enumerate the class labels, starting at `0`:

```
>>> import numpy as np
>>> class_mapping = {label: idx for idx, label in
...                  enumerate(np.unique(df['classlabel']))}
>>> class_mapping
{'class1': 0, 'class2': 1}
```

Next, we can use the mapping dictionary to transform the class labels into integers:

```
>>> df['classlabel'] = df['classlabel'].map(class_mapping)
>>> df
    color  size  price  classlabel
0   green     1   10.1           1
1     red     2   13.5           0
2    blue     3   15.3           1
```

We can reverse the key-value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation:

```
>>> inv_class_mapping = {v: k for k, v in class_mapping.items()}
>>> df['classlabel'] = df['classlabel'].map(inv_class_mapping)
>>> df
    color  size  price  classlabel
0   green     1   10.1      class2
1     red     2   13.5      class1
2    blue     3   15.3      class2
```

Alternatively, there is a convenient `LabelEncoder` class directly implemented in scikit-learn to achieve this:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> class_le = LabelEncoder()
>>> y = class_le.fit_transform(df['classlabel'].values)
>>> y
array([1, 0, 1])
```

Note that the `fit_transform` method is just a shortcut for calling `fit` and `transform` separately, and we can use the `inverse_transform` method to transform the integer class labels back into their original string representation:

```
>>> class_le.inverse_transform(y)
array(['class2', 'class1', 'class2'], dtype=object)
```

Performing one-hot encoding on nominal features

In the previous *Mapping ordinal features* section, we used a simple dictionary mapping approach to convert the ordinal `size` feature into integers. Since scikit-learn's estimators for classification treat class labels as categorical data that does not imply any order (nominal), we used the convenient `LabelEncoder` to encode the string labels into integers. We could use a similar approach to transform the nominal `color` column of our dataset, as follows:

```
>>> X = df[['color', 'size', 'price']].values
>>> color_le = LabelEncoder()
>>> X[:, 0] = color_le.fit_transform(X[:, 0])
>>> X
array([[1, 1, 10.1],
       [2, 2, 13.5],
       [0, 3, 15.3]], dtype=object)
```

After executing the preceding code, the first column of the NumPy array, `X`, now holds the new `color` values, which are encoded as follows:

- `blue = 0`
- `green = 1`
- `red = 2`

If we stop at this point and feed the array to our classifier, we will make one of the most common mistakes in dealing with categorical data. Can you spot the problem? Although the color values don't come in any particular order, common classification models, such as the ones covered in the previous chapters, will now assume that `green` is larger than `blue`, and `red` is larger than `green`. Although this assumption is incorrect, a classifier could still produce useful results. However, those results would not be optimal.

A common workaround for this problem is to use a technique called **one-hot encoding**. The idea behind this approach is to create a new dummy feature for each unique value in the nominal feature column. Here, we would convert the `color` feature into three new features: `blue`, `green`, and `red`. Binary values can then be used to indicate the particular `color` of an example; for example, a `blue` example can be encoded as `blue=1`, `green=0`, `red=0`. To perform this transformation, we can use the `OneHotEncoder` that is implemented in scikit-learn's `preprocessing` module:

```
>>> from sklearn.preprocessing import OneHotEncoder
>>> X = df[['color', 'size', 'price']].values
>>> color_ohe = OneHotEncoder()
>>> color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray()
    array([[0., 1., 0.],
           [0., 0., 1.],
           [1., 0., 0.]])
```

Note that we applied the `OneHotEncoder` to only a single column, `(X[:, 0].reshape(-1, 1))`, to avoid modifying the other two columns in the array as well. If we want to selectively transform columns in a multi-feature array, we can use the `ColumnTransformer`, which accepts a list of `(name, transformer, column(s))` tuples as follows:

```
>>> from sklearn.compose import ColumnTransformer
>>> X = df[['color', 'size', 'price']].values
>>> c_transf = ColumnTransformer([
...     ('onehot', OneHotEncoder(), [0]),
...     ('nothing', 'passthrough', [1, 2])
... ])
>>> c_transf.fit_transform(X).astype(float)
    array([[0.0, 1.0, 0.0, 1, 10.1],
           [0.0, 0.0, 1.0, 2, 13.5],
           [1.0, 0.0, 0.0, 3, 15.3]])
```

In the preceding code example, we specified that we want to modify only the first column and leave the other two columns untouched via the `'passthrough'` argument.

An even more convenient way to create those dummy features via one-hot encoding is to use the `get_dummies` method implemented in pandas. Applied to a `DataFrame`, the `get_dummies` method will only convert string columns and leave all other columns unchanged:

```
>>> pd.get_dummies(df[['price', 'color', 'size']])
    price  size  color_blue  color_green  color_red
0    10.1     1           0            1          0
1    13.5     2           0            0          1
2    15.3     3           1            0          0
```

When we are using one-hot encoding datasets, we have to keep in mind that this introduces multicollinearity, which can be an issue for certain methods (for instance, methods that require matrix inversion). If features are highly correlated, matrices are computationally difficult to invert, which can lead to numerically unstable estimates. To reduce the correlation among variables, we can simply remove one feature column from the one-hot encoded array. Note that we do not lose any important information by removing a feature column, though; for example, if we remove the column `color_blue`, the feature information is still preserved since if we observe `color_green=0` and `color_red=0`, it implies that the observation must be `blue`.

If we use the `get_dummies` function, we can drop the first column by passing a `True` argument to the `drop_first` parameter, as shown in the following code example:

```
>>> pd.get_dummies(df[['price', 'color', 'size']],
...                drop_first=True)
    price  size  color_green  color_red
0    10.1     1            1          0
1    13.5     2            0          1
2    15.3     3            0          0
```

In order to drop a redundant column via the `OneHotEncoder`, we need to set `drop='first'` and set `categories='auto'` as follows:

```
>>> color_ohe = OneHotEncoder(categories='auto', drop='first')
>>> c_transf = ColumnTransformer([
...             ('onehot', color_ohe, [0]),
...             ('nothing', 'passthrough', [1, 2])
... ])
>>> c_transf.fit_transform(X).astype(float)
array([[  1. ,  0. ,  1. ,  10.1],
       [  0. ,  1. ,  2. ,  13.5],
       [  0. ,  0. ,  3. ,  15.3]])
```

Additional encoding schemes for nominal data

While one-hot encoding is the most common way to encode unordered categorical variables, several alternative methods exist. Some of these techniques can be useful when working with categorical features that have high cardinality (a large number of unique category labels). Examples include:

- Binary encoding, which produces multiple binary features similar to one-hot encoding but requires fewer feature columns, i.e., $\log_2(K)$ instead of $K - 1$, where K is the number of unique categories. In binary encoding, numbers are first converted into binary representations, and then each binary number position will form a new feature column.
- Count or frequency encoding, which replaces the label of each category by the number of times or frequency it occurs in the training set.

These methods, as well as additional categorical encoding schemes, are available via the scikit-learn-compatible `category_encoders` library: `https://contrib.scikit-learn.org/category_encoders/`.

While these methods are not guaranteed to perform better than one-hot encoding in terms of model performance, we can consider the choice of a categorical encoding scheme as an additional "hyperparameter" for improving model performance.

Optional: encoding ordinal features

If we are unsure about the numerical differences between the categories of ordinal features, or the difference between two ordinal values is not defined, we can also encode them using a threshold encoding with 0/1 values. For example, we can split the feature `size` with values `M`, `L`, and `XL` into two new features, `x > M` and `x > L`. Let's consider the original `DataFrame`:

```
>>> df = pd.DataFrame([['green', 'M', 10.1,
...                     'class2'],
...                    ['red', 'L', 13.5,
...                     'class1'],
...                    ['blue', 'XL', 15.3,
...                     'class2']])
>>> df.columns = ['color', 'size', 'price',
...               'classlabel']
>>> df
```

We can use the `apply` method of pandas' `DataFrame` to write custom lambda expressions in order to encode these variables using the value-threshold approach:

```
>>> df['x > M'] = df['size'].apply(
...     lambda x: 1 if x in {'L', 'XL'} else 0)
>>> df['x > L'] = df['size'].apply(
...     lambda x: 1 if x == 'XL' else 0)
```

```
>>> del df['size']
>>> df
```

Partitioning a dataset into separate training and test datasets

We briefly introduced the concept of partitioning a dataset into separate datasets for training and testing in *Chapter 1*, *Giving Computers the Ability to Learn from Data*, and *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*. Remember that comparing predictions to true labels in the test set can be understood as the unbiased performance evaluation of our model before we let it loose in the real world. In this section, we will prepare a new dataset, the **Wine** dataset. After we have preprocessed the dataset, we will explore different techniques for feature selection to reduce the dimensionality of a dataset.

The Wine dataset is another open-source dataset that is available from the UCI machine learning repository (`https://archive.ics.uci.edu/ml/datasets/Wine`); it consists of 178 wine examples with 13 features describing their different chemical properties.

Obtaining the Wine dataset

You can find a copy of the Wine dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the dataset at `https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data` is temporarily unavailable on the UCI server. For instance, to load the Wine dataset from a local directory, you can replace this line:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None
)
```

with the following one:

```
df = pd.read_csv(
    'your/local/path/to/wine.data', header=None
)
```

Using the pandas library, we will directly read in the open-source Wine dataset from the UCI machine learning repository:

```
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/'
...                       'ml/machine-learning-databases/'
...                       'wine/wine.data', header=None)
```

```
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash', 'Magnesium',
...                    'Total phenols', 'Flavanoids',
...                    'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> print('Class labels', np.unique(df_wine['Class label']))
Class labels [1 2 3]
>>> df_wine.head()
```

The 13 different features in the Wine dataset, describing the chemical properties of the 178 wine examples, are listed in the following table:

	Class label	Alcohol	Malic acid	Ash	Alcalinity of ash	Magnesium	Total phenols	Flavanoids	Nonflavanoid phenols	Proanthocyanins	Color intensity	Hue	OD280/OD315 of diluted wines	Proline
0	1	14.23	1.71	2.43	15.6	127	2.80	3.06	0.28	2.29	5.64	1.04	3.92	1065
1	1	13.20	1.78	2.14	11.2	100	2.65	2.76	0.26	1.28	4.38	1.05	3.40	1050
2	1	13.16	2.36	2.67	18.6	101	2.80	3.24	0.30	2.81	5.68	1.03	3.17	1185
3	1	14.37	1.95	2.50	16.8	113	3.85	3.49	0.24	2.18	7.80	0.86	3.45	1480
4	1	13.24	2.59	2.87	21.0	118	2.80	2.69	0.39	1.82	4.32	1.04	2.93	735

Figure 4.4: A sample of the Wine dataset

The examples belong to one of three different classes, `1`, `2`, and `3`, which refer to the three different types of grape grown in the same region in Italy but derived from different wine cultivars, as described in the dataset summary (`https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.names`).

A convenient way to randomly partition this dataset into separate test and training datasets is to use the `train_test_split` function from scikit-learn's `model_selection` submodule:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                      test_size=0.3,
...                      random_state=0,
...                      stratify=y)
```

First, we assigned the NumPy array representation of the feature columns 1-13 to the variable `X` and we assigned the class labels from the first column to the variable `y`. Then, we used the `train_test_split` function to randomly split `X` and `y` into separate training and test datasets.

By setting `test_size=0.3`, we assigned 30 percent of the wine examples to `X_test` and `y_test`, and the remaining 70 percent of the examples were assigned to `X_train` and `y_train`, respectively. Providing the class label array `y` as an argument to `stratify` ensures that both training and test datasets have the same class proportions as the original dataset.

Choosing an appropriate ratio for partitioning a dataset into training and test datasets

If we are dividing a dataset into training and test datasets, we have to keep in mind that we are withholding valuable information that the learning algorithm could benefit from. Thus, we don't want to allocate too much information to the test set. However, the smaller the test set, the more inaccurate the estimation of the generalization error. Dividing a dataset into training and test datasets is all about balancing this tradeoff. In practice, the most commonly used splits are 60:40, 70:30, or 80:20, depending on the size of the initial dataset. However, for large datasets, 90:10 or 99:1 splits are also common and appropriate. For example, if the dataset contains more than 100,000 training examples, it might be fine to withhold only 10,000 examples for testing in order to get a good estimate of the generalization performance. More information and illustrations can be found in section one of my article *Model evaluation, model selection, and algorithm selection in machine learning*, which is freely available at `https://arxiv.org/pdf/1811.12808.pdf`. Also, we will revisit the topic of model evaluation and discuss it in more detail in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*.

Moreover, instead of discarding the allocated test data after model training and evaluation, it is a common practice to retrain a classifier on the entire dataset, as it can improve the predictive performance of the model. While this approach is generally recommended, it could lead to worse generalization performance if the dataset is small and the test dataset contains outliers, for example. Also, after refitting the model on the whole dataset, we don't have any independent data left to evaluate its performance.

Bringing features onto the same scale

Feature scaling is a crucial step in our preprocessing pipeline that can easily be forgotten. **Decision trees** and **random forests** are two of the very few machine learning algorithms where we don't need to worry about feature scaling. Those algorithms are scale-invariant. However, the majority of machine learning and optimization algorithms behave much better if features are on the same scale, as we saw in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*, when we implemented the **gradient descent optimization** algorithm.

The importance of feature scaling can be illustrated by a simple example. Let's assume that we have two features where one feature is measured on a scale from 1 to 10 and the second feature is measured on a scale from 1 to 100,000, respectively.

When we think of the squared error function in Adaline from *Chapter 2*, it makes sense to say that the algorithm will mostly be busy optimizing the weights according to the larger errors in the second feature. Another example is the **k-nearest neighbors** (**KNN**) algorithm with a Euclidean distance measure: the computed distances between examples will be dominated by the second feature axis.

Now, there are two common approaches to bringing different features onto the same scale: **normalization** and **standardization**. Those terms are often used quite loosely in different fields, and the meaning has to be derived from the context. Most often, normalization refers to the rescaling of the features to a range of [0, 1], which is a special case of **min-max scaling**. To normalize our data, we can simply apply the min-max scaling to each feature column, where the new value, $x_{norm}^{(i)}$, of an example, $x^{(i)}$, can be calculated as follows:

$$x_{norm}^{(i)} = \frac{x^{(i)} - x_{min}}{x_{max} - x_{min}}$$

Here, $x^{(i)}$ is a particular example, x_{min} is the smallest value in a feature column, and x_{max} is the largest value.

The min-max scaling procedure is implemented in scikit-learn and can be used as follows:

```
>>> from sklearn.preprocessing import MinMaxScaler
>>> mms = MinMaxScaler()
>>> X_train_norm = mms.fit_transform(X_train)
>>> X_test_norm = mms.transform(X_test)
```

Although normalization via min-max scaling is a commonly used technique that is useful when we need values in a bounded interval, standardization can be more practical for many machine learning algorithms, especially for optimization algorithms such as gradient descent. The reason is that many linear models, such as the logistic regression and SVM from *Chapter 3*, initialize the weights to 0 or small random values close to 0. Using standardization, we center the feature columns at mean 0 with standard deviation 1 so that the feature columns have the same parameters as a standard normal distribution (zero mean and unit variance), which makes it easier to learn the weights. However, we shall emphasize that standardization does not change the shape of the distribution, and it does not transform non-normally distributed data into normally distributed data. In addition to scaling data such that it has zero mean and unit variance, standardization maintains useful information about outliers and makes the algorithm less sensitive to them in contrast to min-max scaling, which scales the data to a limited range of values.

The procedure for standardization can be expressed by the following equation:

$$x_{std}^{(i)} = \frac{x^{(i)} - \mu_x}{\sigma_x}$$

Here, μ_x is the sample mean of a particular feature column, and σ_x is the corresponding standard deviation.

The following table illustrates the difference between the two commonly used feature scaling techniques, standardization and normalization, on a simple example dataset consisting of numbers 0 to 5:

Input	Standardized	Min-max normalized
0.0	-1.46385	0.0
1.0	-0.87831	0.2
2.0	-0.29277	0.4
3.0	0.29277	0.6
4.0	0.87831	0.8
5.0	1.46385	1.0

Table 4.1: A comparison between standardization and min-max normalization

You can perform the standardization and normalization shown in the table manually by executing the following code examples:

```
>>> ex = np.array([0, 1, 2, 3, 4, 5])
>>> print('standardized:', (ex - ex.mean()) / ex.std())
standardized: [-1.46385011  -0.87831007  -0.29277002  0.29277002
0.87831007  1.46385011]
>>> print('normalized:', (ex - ex.min()) / (ex.max() - ex.min()))
normalized: [ 0.  0.2  0.4  0.6  0.8  1. ]
```

Similar to the `MinMaxScaler` class, scikit-learn also implements a class for standardization:

```
>>> from sklearn.preprocessing import StandardScaler
>>> stdsc = StandardScaler()
>>> X_train_std = stdsc.fit_transform(X_train)
>>> X_test_std = stdsc.transform(X_test)
```

Again, it is also important to highlight that we fit the `StandardScaler` class only once—on the training data—and use those parameters to transform the test dataset or any new data point.

Other, more advanced methods for feature scaling are available from scikit-learn, such as `RobustScaler`. `RobustScaler` is especially helpful and recommended if we are working with small datasets that contain many outliers. Similarly, if the machine learning algorithm applied to this dataset is prone to **overfitting**, `RobustScaler` can be a good choice. Operating on each feature column independently, `RobustScaler` removes the median value and scales the dataset according to the 1st and 3rd quartile of the dataset (that is, the 25th and 75th quantile, respectively) such that more extreme values and outliers become less pronounced. The interested reader can find more information about `RobustScaler` in the official scikit-learn documentation at `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html`.

Selecting meaningful features

If we notice that a model performs much better on a training dataset than on the test dataset, this observation is a strong indicator of overfitting. As we discussed in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*, overfitting means the model fits the parameters too closely with regard to the particular observations in the training dataset but does not generalize well to new data; we say that the model has a **high variance**. The reason for the overfitting is that our model is too complex for the given training data. Common solutions to reduce the generalization error are as follows:

- Collect more training data
- Introduce a penalty for complexity via regularization
- Choose a simpler model with fewer parameters
- Reduce the dimensionality of the data

Collecting more training data is often not applicable. In *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*, we will learn about a useful technique to check whether more training data is helpful. In the following sections, we will look at common ways to reduce overfitting by regularization and dimensionality reduction via feature selection, which leads to simpler models by requiring fewer parameters to be fitted to the data. Then, in *Chapter 5*, *Compressing Data via Dimensionality Reduction*, we will take a look at additional feature extraction techniques.

L1 and L2 regularization as penalties against model complexity

You will recall from *Chapter 3* that **L2 regularization** is one approach to reduce the complexity of a model by penalizing large individual weights. We defined the squared L2 norm of our weight vector, $\boldsymbol{w}$, as follows:

$$L2: \quad \|\boldsymbol{w}\|_2^2 = \sum_{j=1}^{m} w_j^2$$

Another approach to reduce the model complexity is the related **L1 regularization**:

$$L1: \quad \|\boldsymbol{w}\|_1 = \sum_{j=1}^{m} |w_j|$$

Here, we simply replaced the square of the weights with the sum of the absolute values of the weights. In contrast to L2 regularization, L1 regularization usually yields sparse feature vectors, and most feature weights will be zero. Sparsity can be useful in practice if we have a high-dimensional dataset with many features that are irrelevant, especially in cases where we have more irrelevant dimensions than training examples. In this sense, L1 regularization can be understood as a technique for feature selection.

A geometric interpretation of L2 regularization

As mentioned in the previous section, L2 regularization adds a penalty term to the loss function that effectively results in less extreme weight values compared to a model trained with an unregularized loss function.

To better understand how L1 regularization encourages sparsity, let's take a step back and take a look at a geometric interpretation of regularization. Let's plot the contours of a convex loss function for two weight coefficients, w_1 and w_2.

Here, we will consider the **mean squared error** (**MSE**) loss function that we used for Adaline in *Chapter 2*, which computes the squared distances between the true and predicted class labels, y and $\hat{y}$, averaged over all N examples in the training set. Since the MSE is spherical, it is easier to draw than the loss function of logistic regression; however, the same concepts apply. Remember that our goal is to find the combination of weight coefficients that minimize the loss function for the training data, as shown in *Figure 4.5* (the point in the center of the ellipses):

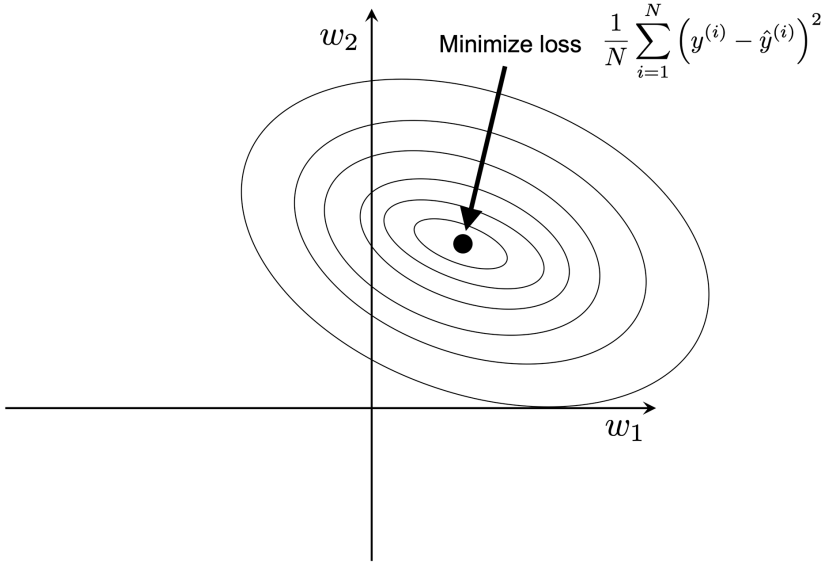


Figure 4.5: Minimizing the mean squared error loss function

We can think of regularization as adding a penalty term to the loss function to encourage smaller weights; in other words, we penalize large weights. Thus, by increasing the regularization strength via the regularization parameter, λ, we shrink the weights toward zero and decrease the dependence of our model on the training data. Let's illustrate this concept in the following figure for the L2 penalty term:

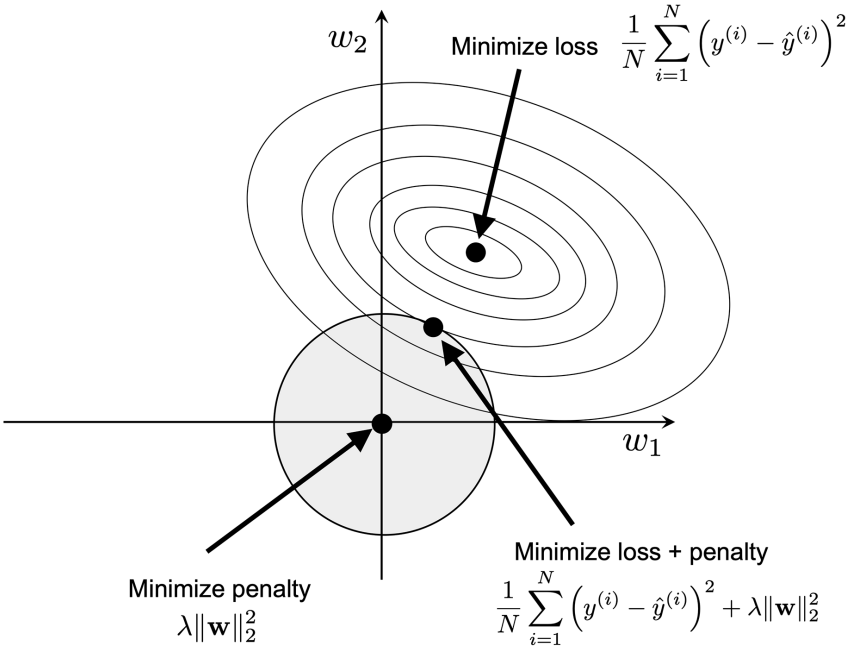


Figure 4.6: Applying L2 regularization to the loss function

The quadratic L2 regularization term is represented by the shaded ball. Here, our weight coefficients cannot exceed our regularization budget—the combination of the weight coefficients cannot fall outside the shaded area. On the other hand, we still want to minimize the loss function. Under the penalty constraint, our best effort is to choose the point where the L2 ball intersects with the contours of the unpenalized loss function. The larger the value of the regularization parameter, λ, gets, the faster the penalized loss grows, which leads to a narrower L2 ball. For example, if we increase the regularization parameter toward infinity, the weight coefficients will become effectively zero, denoted by the center of the L2 ball. To summarize the main message of the example, our goal is to minimize the sum of the unpenalized loss plus the penalty term, which can be understood as adding bias and preferring a simpler model to reduce the variance in the absence of sufficient training data to fit the model.

Sparse solutions with L1 regularization

Now, let's discuss L1 regularization and sparsity. The main concept behind L1 regularization is similar to what we discussed in the previous section. However, since the L1 penalty is the sum of the absolute weight coefficients (remember that the L2 term is quadratic), we can represent it as a diamond-shape budget, as shown in *Figure 4.7*:

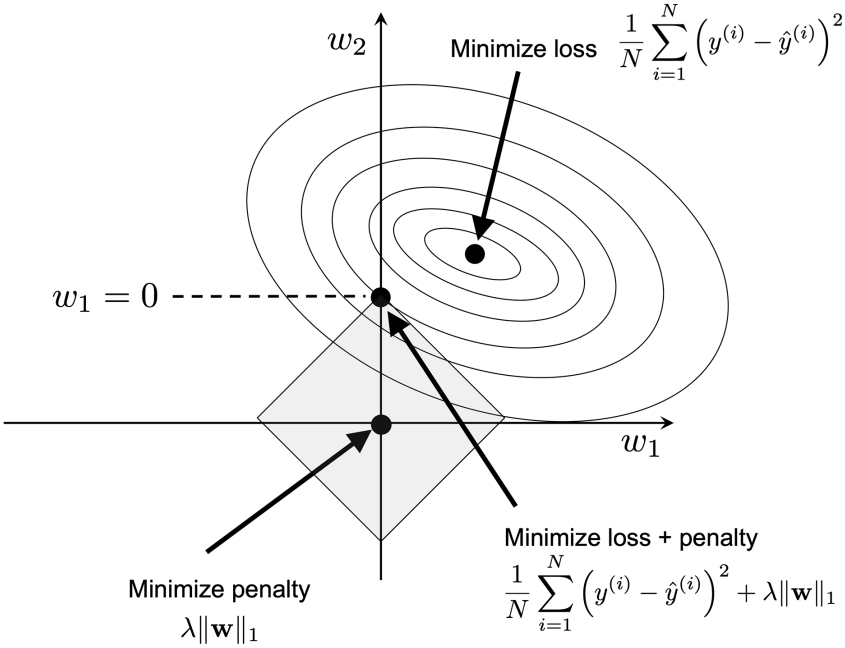


Figure 4.7: Applying L1 regularization to the loss function

In the preceding figure, we can see that the contour of the loss function touches the L1 diamond at $w_1 = 0$. Since the contours of an L1 regularized system are sharp, it is more likely that the optimum—that is, the intersection between the ellipses of the loss function and the boundary of the L1 diamond—is located on the axes, which encourages sparsity.

L1 regularization and sparsity

The mathematical details of why L1 regularization can lead to sparse solutions are beyond the scope of this book. If you are interested, an excellent explanation of L2 versus L1 regularization can be found in *Section 3.4*, *The Elements of Statistical Learning* by *Trevor Hastie, Robert Tibshirani*, and *Jerome Friedman, Springer Science+Business Media,* 2009.

For regularized models in scikit-learn that support L1 regularization, we can simply set the `penalty` parameter to `'l1'` to obtain a sparse solution:

```
>>> from sklearn.linear_model import LogisticRegression
>>> LogisticRegression(penalty='l1',
...                    solver='liblinear',
...                    multi_class='ovr')
```

Note that we also need to select a different optimization algorithm (for example, `solver='liblinear'`), since `'lbfgs'` currently does not support L1-regularized loss optimization. Applied to the standardized Wine data, the L1 regularized logistic regression would yield the following sparse solution:

```
>>> lr = LogisticRegression(penalty='l1',
...                         C=1.0,
...                         solver='liblinear',
...                         multi_class='ovr')
>>> # Note that C=1.0 is the default. You can increase
>>> # or decrease it to make the regularization effect
>>> # stronger or weaker, respectively.
>>> lr.fit(X_train_std, y_train)
>>> print('Training accuracy:', lr.score(X_train_std, y_train))
Training accuracy: 1.0
>>> print('Test accuracy:', lr.score(X_test_std, y_test))
Test accuracy: 1.0
```

Both training and test accuracies (both 100 percent) indicate that our model does a perfect job on both datasets. When we access the intercept terms via the `lr.intercept_` attribute, we can see that the array returns three values:

```
>>> lr.intercept_
    array([-1.26317363, -1.21537306, -2.37111954])
```

Since we fit the `LogisticRegression` object on a multiclass dataset via the **one-versus-rest** (**OvR**) approach, the first intercept belongs to the model that fits class 1 versus classes 2 and 3, the second value is the intercept of the model that fits class 2 versus classes 1 and 3, and the third value is the intercept of the model that fits class 3 versus classes 1 and 2:

```
>>> lr.coef_
array([[ 1.24647953,  0.18050894,  0.74540443, -1.16301108,
         0.        ,0.        ,  1.16243821,  0.        ,
         0.        ,  0.        , 0.        ,  0.55620267,
         2.50890638],
       [-1.53919461, -0.38562247, -0.99565934,  0.36390047,
        -0.05892612, 0.        ,  0.66710883,  0.        ,
         0.        , -1.9318798 , 1.23775092,  0.        ,
```

```
        -2.23280039],
       [ 0.13557571,  0.16848763,  0.35710712,  0.        ,
         0.        , 0.        , -2.43804744,  0.        ,
         0.        ,  1.56388787, -0.81881015, -0.49217022,
         0.        ]])
```

The weight array that we accessed via the `lr.coef_` attribute contains three rows of weight coefficients, one weight vector for each class. Each row consists of 13 weights, where each weight is multiplied by the respective feature in the 13-dimensional Wine dataset to calculate the net input:

$$z = w_1x_1 + \cdots + w_mx_m + b = \sum_{j=1}^{m} x_jw_j + b = \boldsymbol{w}^T\boldsymbol{x} + b$$

Accessing the bias unit and weight parameters of scikit-learn estimators

In scikit-learn, `intercept_` corresponds to the bias unit and `coef_` corresponds to the values w_j.

As a result of L1 regularization, which, as mentioned, serves as a method for feature selection, we just trained a model that is robust to the potentially irrelevant features in this dataset. Strictly speaking, though, the weight vectors from the previous example are not necessarily sparse because they contain more non-zero than zero entries. However, we could enforce sparsity (more zero entries) by further increasing the regularization strength—that is, choosing lower values for the `C` parameter.

In the last example on regularization in this chapter, we will vary the regularization strength and plot the regularization path—the weight coefficients of the different features for different regularization strengths:

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = plt.subplot(111)
>>> colors = ['blue', 'green', 'red', 'cyan',
...           'magenta', 'yellow', 'black',
...           'pink', 'lightgreen', 'lightblue',
...           'gray', 'indigo', 'orange']
>>> weights, params = [], []
>>> for c in np.arange(-4., 6.):
...     lr = LogisticRegression(penalty='l1', C=10.**c,
...                             solver='liblinear',
...                             multi_class='ovr', random_state=0)
...     lr.fit(X_train_std, y_train)
...     weights.append(lr.coef_[1])
...     params.append(10**c)
```

```
>>> weights = np.array(weights)
>>> for column, color in zip(range(weights.shape[1]), colors):
...     plt.plot(params, weights[:, column],
...              label=df_wine.columns[column + 1],
...              color=color)
>>> plt.axhline(0, color='black', linestyle='--', linewidth=3)
>>> plt.xlim([10**(-5), 10**5])
>>> plt.ylabel('Weight coefficient')
>>> plt.xlabel('C (inverse regularization strength)')
>>> plt.xscale('log')
>>> plt.legend(loc='upper left')
>>> ax.legend(loc='upper center',
...           bbox_to_anchor=(1.38, 1.03),
...           ncol=1, fancybox=True)
>>> plt.show()
```

The resulting plot provides us with further insights into the behavior of L1 regularization. As we can see, all feature weights will be zero if we penalize the model with a strong regularization parameter ($C < 0.01$); C is the inverse of the regularization parameter, λ:

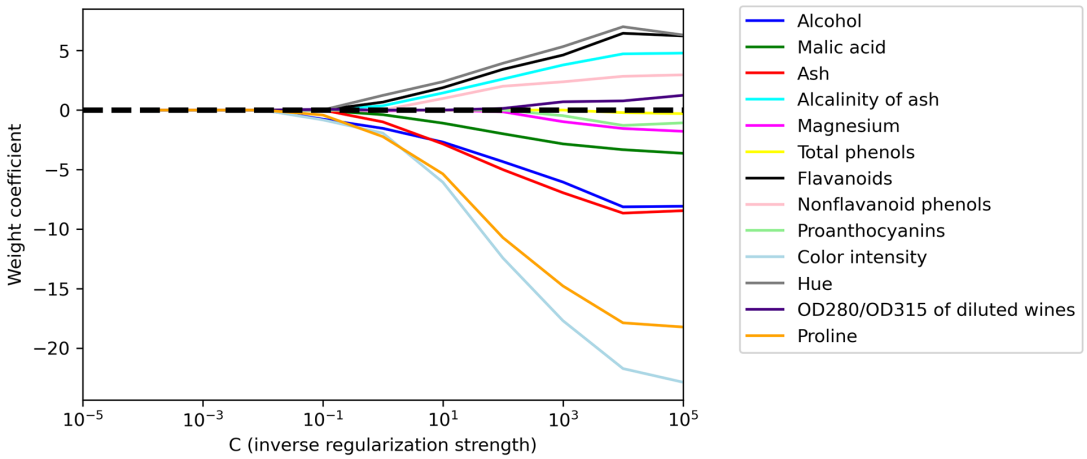


Figure 4.8: The impact of the value of the regularization strength hyperparameter C

Sequential feature selection algorithms

An alternative way to reduce the complexity of the model and avoid overfitting is **dimensionality reduction** via feature selection, which is especially useful for unregularized models. There are two main categories of dimensionality reduction techniques: **feature selection** and **feature extraction**. Via feature selection, we *select* a subset of the original features, whereas in feature extraction, we *derive* information from the feature set to construct a new feature subspace.

In this section, we will take a look at a classic family of feature selection algorithms. In the next chapter, *Chapter 5*, *Compressing Data via Dimensionality Reduction*, we will learn about different feature extraction techniques to compress a dataset onto a lower-dimensional feature subspace.

Sequential feature selection algorithms are a family of greedy search algorithms that are used to reduce an initial d-dimensional feature space to a k-dimensional feature subspace where $k<d$. The motivation behind feature selection algorithms is to automatically select a subset of features that are most relevant to the problem, to improve computational efficiency, or to reduce the generalization error of the model by removing irrelevant features or noise, which can be useful for algorithms that don't support regularization.

A classic sequential feature selection algorithm is **sequential backward selection** (**SBS**), which aims to reduce the dimensionality of the initial feature subspace with a minimum decay in the performance of the classifier to improve upon computational efficiency. In certain cases, SBS can even improve the predictive power of the model if a model suffers from overfitting.

Greedy search algorithms

Greedy algorithms make locally optimal choices at each stage of a combinatorial search problem and generally yield a suboptimal solution to the problem, in contrast to exhaustive search algorithms, which evaluate all possible combinations and are guaranteed to find the optimal solution. However, in practice, an exhaustive search is often computationally not feasible, whereas greedy algorithms allow for a less complex, computationally more efficient solution.

The idea behind the SBS algorithm is quite simple: SBS sequentially removes features from the full feature subset until the new feature subspace contains the desired number of features. To determine which feature is to be removed at each stage, we need to define the criterion function, J, that we want to minimize.

The criterion calculated by the criterion function can simply be the difference in the performance of the classifier before and after the removal of a particular feature. Then, the feature to be removed at each stage can simply be defined as the feature that maximizes this criterion; or in more simple terms, at each stage we eliminate the feature that causes the least performance loss after removal. Based on the preceding definition of SBS, we can outline the algorithm in four simple steps:

1. Initialize the algorithm with $k = d$, where d is the dimensionality of the full feature space, $\boldsymbol{X}_d$.
2. Determine the feature, $\boldsymbol{x}^-$, that maximizes the criterion: $\boldsymbol{x}^- = \text{argmax}\, J(\boldsymbol{X}_k - \boldsymbol{x})$, where $\boldsymbol{x} \in \boldsymbol{X}_k$.
3. Remove the feature, $\boldsymbol{x}^-$, from the feature set: $\boldsymbol{X}_{k-1} = \boldsymbol{X}_k - \boldsymbol{x}^-$; $k = k - 1$.
4. Terminate if k equals the number of desired features; otherwise, go to *step 2*.

A resource on sequential feature algorithms

You can find a detailed evaluation of several sequential feature algorithms in *Comparative Study of Techniques for Large-Scale Feature Selection* by *F. Ferri*, *P. Pudil*, *M. Hatef*, and *J. Kittler*, pages 403-413, 1994.

To practice our coding skills and ability to implement our own algorithms, let's go ahead and implement it in Python from scratch:

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

class SBS:
    def __init__(self, estimator, k_features,
                 scoring=accuracy_score,
                 test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state
    def fit(self, X, y):
        X_train, X_test, y_train, y_test = \
            train_test_split(X, y, test_size=self.test_size,
                             random_state=self.random_state)

        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                 X_test, y_test, self.indices_)
        self.scores_ = [score]
        while dim > self.k_features:
            scores = []
            subsets = []

            for p in combinations(self.indices_, r=dim - 1):
                score = self._calc_score(X_train, y_train,
```

```
                                        X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]

        return self

    def transform(self, X):
        return X[:, self.indices_]

    def _calc_score(self, X_train, y_train, X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

In the preceding implementation, we defined the `k_features` parameter to specify the desired number of features we want to return. By default, we use `accuracy_score` from scikit-learn to evaluate the performance of a model (an estimator for classification) on the feature subsets.

Inside the `while` loop of the `fit` method, the feature subsets created by the `itertools.combination` function are evaluated and reduced until the feature subset has the desired dimensionality. In each iteration, the accuracy score of the best subset is collected in a list, `self.scores_`, based on the internally created test dataset, `X_test`. We will use those scores later to evaluate the results. The column indices of the final feature subset are assigned to `self.indices_`, which we can use via the `transform` method to return a new data array with the selected feature columns. Note that, instead of calculating the criterion explicitly inside the `fit` method, we simply removed the feature that is not contained in the best performing feature subset.

Now, let's see our SBS implementation in action using the KNN classifier from scikit-learn:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.neighbors import KNeighborsClassifier
>>> knn = KNeighborsClassifier(n_neighbors=5)
>>> sbs = SBS(knn, k_features=1)
>>> sbs.fit(X_train_std, y_train)
```

Although our SBS implementation already splits the dataset into a test and training dataset inside the `fit` function, we still fed the training dataset, `X_train`, to the algorithm. The SBS `fit` method will then create new training subsets for testing (validation) and training, which is why this test set is also called the **validation dataset**. This approach is necessary to prevent our *original* test set from becoming part of the training data.

Remember that our SBS algorithm collects the scores of the best feature subset at each stage, so let's move on to the more exciting part of our implementation and plot the classification accuracy of the KNN classifier that was calculated on the validation dataset. The code is as follows:

```
>>> k_feat = [len(k) for k in sbs.subsets_]
>>> plt.plot(k_feat, sbs.scores_, marker='o')
>>> plt.ylim([0.7, 1.02])
>>> plt.ylabel('Accuracy')
>>> plt.xlabel('Number of features')
>>> plt.grid()
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in *Figure 4.9*, the accuracy of the KNN classifier improved on the validation dataset as we reduced the number of features, which is likely due to a decrease in the **curse of dimensionality** that we discussed in the context of the KNN algorithm in *Chapter 3*. Also, we can see in the following plot that the classifier achieved 100 percent accuracy for $k = \{3, 7, 8, 9, 10, 11, 12\}$:

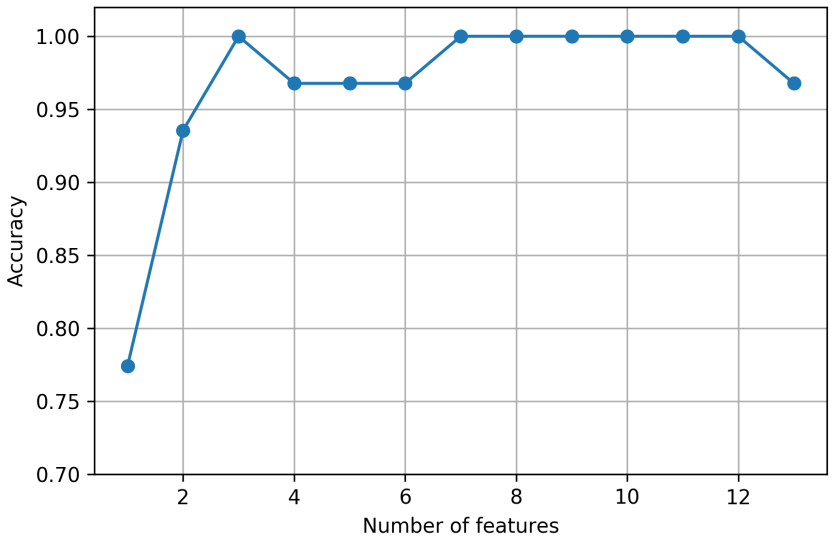


Figure 4.9: Impact of number of features on model accuracy

To satisfy our own curiosity, let's see what the smallest feature subset (k=3), which yielded such a good performance on the validation dataset, looks like:

```
>>> k3 = list(sbs.subsets_[10])
>>> print(df_wine.columns[1:][k3])
Index(['Alcohol', 'Malic acid', 'OD280/OD315 of diluted wines'],
dtype='object')
```

Using the preceding code, we obtained the column indices of the three-feature subset from the 11th position in the `sbs.subsets_` attribute and returned the corresponding feature names from the column index of the pandas Wine `DataFrame`.

Next, let's evaluate the performance of the KNN classifier on the original test dataset:

```
>>> knn.fit(X_train_std, y_train)
>>> print('Training accuracy:', knn.score(X_train_std, y_train))
Training accuracy: 0.967741935484
>>> print('Test accuracy:', knn.score(X_test_std, y_test))
Test accuracy: 0.962962962963
```

In the preceding code section, we used the complete feature set and obtained approximately 97 percent accuracy on the training dataset and approximately 96 percent accuracy on the test dataset, which indicates that our model already generalizes well to new data. Now, let's use the selected three-feature subset and see how well KNN performs:

```
>>> knn.fit(X_train_std[:, k3], y_train)
>>> print('Training accuracy:',
...       knn.score(X_train_std[:, k3], y_train))
Training accuracy: 0.951612903226
>>> print('Test accuracy:',
...       knn.score(X_test_std[:, k3], y_test))
Test accuracy: 0.925925925926
```

When using less than a quarter of the original features in the Wine dataset, the prediction accuracy on the test dataset declined slightly. This may indicate that those three features do not provide less discriminatory information than the original dataset. However, we also have to keep in mind that the Wine dataset is a small dataset and is very susceptible to randomness—that is, the way we split the dataset into training and test subsets, and how we split the training dataset further into a training and validation subset.

While we did not increase the performance of the KNN model by reducing the number of features, we shrank the size of the dataset, which can be useful in real-world applications that may involve expensive data collection steps. Also, by substantially reducing the number of features, we obtain simpler models, which are easier to interpret.

Feature selection algorithms in scikit-learn

You can find implementations of several different flavors of sequential feature selection related to the simple SBS that we implemented previously in the Python package `mlxtend` at `http://rasbt.github.io/mlxtend/user_guide/feature_selection/SequentialFeatureSelector/`. While our `mlxtend` implementation comes with many bells and whistles, we collaborated with the scikit-learn team to implement a simplified, user-friendly version, which has been part of the recent v0.24 release. The usage and behavior are very similar to the `SBS` code we implemented in this chapter. If you would like to learn more, please see the documentation at `https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SequentialFeatureSelector.html`.

There are many more feature selection algorithms available via scikit-learn. These include recursive backward elimination based on feature weights, tree-based methods to select features by importance, and univariate statistical tests. A comprehensive discussion of the different feature selection methods is beyond the scope of this book, but a good summary with illustrative examples can be found at `http://scikit-learn.org/stable/modules/feature_selection.html`.

Assessing feature importance with random forests

In previous sections, you learned how to use L1 regularization to zero out irrelevant features via logistic regression and how to use the SBS algorithm for feature selection and apply it to a KNN algorithm. Another useful approach for selecting relevant features from a dataset is using a **random forest**, an ensemble technique that was introduced in *Chapter 3*. Using a random forest, we can measure the feature importance as the averaged impurity decrease computed from all decision trees in the forest, without making any assumptions about whether our data is linearly separable or not. Conveniently, the random forest implementation in scikit-learn already collects the feature importance values for us so that we can access them via the `feature_importances_` attribute after fitting a `RandomForestClassifier`. By executing the following code, we will now train a forest of 500 trees on the Wine dataset and rank the 13 features by their respective importance measures—remember from our discussion in *Chapter 3* that we don't need to use standardized or normalized features in tree-based models:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> feat_labels = df_wine.columns[1:]
>>> forest = RandomForestClassifier(n_estimators=500,
...                                 random_state=1)
>>> forest.fit(X_train, y_train)
>>> importances = forest.feature_importances_
>>> indices = np.argsort(importances)[::-1]
>>> for f in range(X_train.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
```

```
...                             importances[indices[f]]))
>>> plt.title('Feature importance')
>>> plt.bar(range(X_train.shape[1]),
...         importances[indices],
...         align='center')
>>> plt.xticks(range(X_train.shape[1]),
...            feat_labels[indices], rotation=90)
>>> plt.xlim([-1, X_train.shape[1]])
>>> plt.tight_layout()
>>> plt.show()
 1) Proline                        0.185453
 2) Flavanoids                     0.174751
 3) Color intensity                0.143920
 4) OD280/OD315 of diluted wines   0.136162
 5) Alcohol                        0.118529
 6) Hue                            0.058739
 7) Total phenols                  0.050872
 8) Magnesium                      0.031357
 9) Malic acid                     0.025648
 10) Proanthocyanins               0.025570
 11) Alcalinity of ash             0.022366
 12) Nonflavanoid phenols          0.013354
 13) Ash                           0.013279
```

After executing the code, we created a plot that ranks the different features in the Wine dataset by their relative importance; note that the feature importance values are normalized so that they sum up to 1.0:

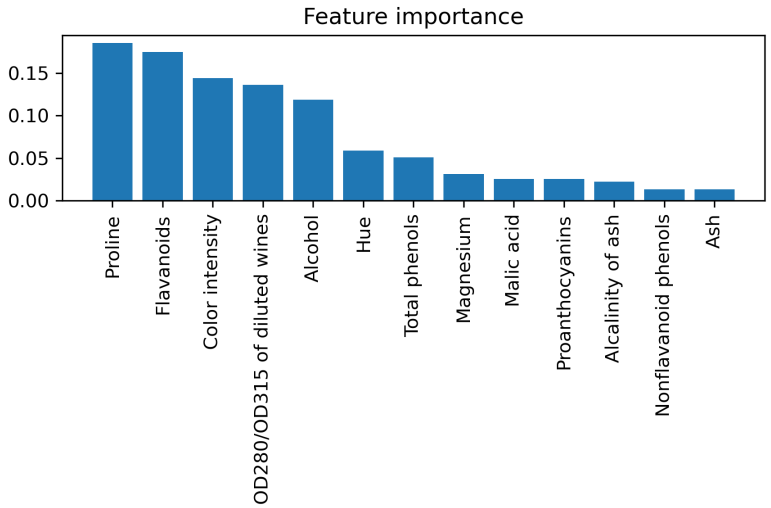


Figure 4.10: Random forest-based feature importance of the Wine dataset

We can conclude that the proline and flavonoid levels, the color intensity, the OD280/OD315 diffraction, and the alcohol concentration of wine are the most discriminative features in the dataset based on the average impurity decrease in the 500 decision trees. Interestingly, two of the top-ranked features in the plot are also in the three-feature subset selection from the SBS algorithm that we implemented in the previous section (alcohol concentration and OD280/OD315 of diluted wines).

However, as far as interpretability is concerned, the random forest technique comes with an important *gotcha* that is worth mentioning. If two or more features are highly correlated, one feature may be ranked very highly while the information on the other feature(s) may not be fully captured. On the other hand, we don't need to be concerned about this problem if we are merely interested in the predictive performance of a model rather than the interpretation of feature importance values.

To conclude this section about feature importance values and random forests, it is worth mentioning that scikit-learn also implements a `SelectFromModel` object that selects features based on a user-specified threshold after model fitting, which is useful if we want to use the `RandomForestClassifier` as a feature selector and intermediate step in a scikit-learn `Pipeline` object, which allows us to connect different preprocessing steps with an estimator, as you will see in *Chapter 6*, *Learning Best Practices for Model Evaluation and Hyperparameter Tuning*. For example, we could set the `threshold` to `0.1` to reduce the dataset to the five most important features using the following code:

```
>>> from sklearn.feature_selection import SelectFromModel
>>> sfm = SelectFromModel(forest, threshold=0.1, prefit=True)
>>> X_selected = sfm.transform(X_train)
>>> print('Number of features that meet this threshold',
...       'criterion:', X_selected.shape[1])
Number of features that meet this threshold criterion: 5
>>> for f in range(X_selected.shape[1]):
...     print("%2d) %-*s %f" % (f + 1, 30,
...                             feat_labels[indices[f]],
...                             importances[indices[f]]))
 1) Proline                        0.185453
 2) Flavanoids                     0.174751
 3) Color intensity                0.143920
 4) OD280/OD315 of diluted wines   0.136162
 5) Alcohol                        0.118529
```

Summary

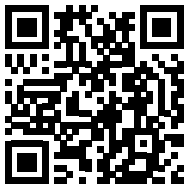
We started this chapter by looking at useful techniques to make sure that we handle missing data correctly. Before we feed data to a machine learning algorithm, we also have to make sure that we encode categorical variables correctly, and in this chapter, we saw how we can map ordinal and nominal feature values to integer representations.

Moreover, we briefly discussed L1 regularization, which can help us to avoid overfitting by reducing the complexity of a model. As an alternative approach to removing irrelevant features, we used a sequential feature selection algorithm to select meaningful features from a dataset.

In the next chapter, you will learn about yet another useful approach to dimensionality reduction: feature extraction. It allows us to compress features onto a lower-dimensional subspace, rather than removing features entirely as in feature selection.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`



5

Compressing Data via Dimensionality Reduction

In *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, you learned about the different approaches for reducing the dimensionality of a dataset using different feature selection techniques. An alternative approach to feature selection for dimensionality reduction is **feature extraction.** In this chapter, you will learn about two fundamental techniques that will help you to summarize the information content of a dataset by transforming it onto a new feature subspace of lower dimensionality than the original one. Data compression is an important topic in machine learning, and it helps us to store and analyze the increasing amounts of data that are produced and collected in the modern age of technology.

In this chapter, we will cover the following topics:

- Principal component analysis for unsupervised data compression
- Linear discriminant analysis as a supervised dimensionality reduction technique for maximizing class separability
- A brief overview of nonlinear dimensionality reduction techniques and t-distributed stochastic neighbor embedding for data visualization

Unsupervised dimensionality reduction via principal component analysis

Similar to feature selection, we can use different feature extraction techniques to reduce the number of features in a dataset. The difference between feature selection and feature extraction is that while we maintain the original features when we use feature selection algorithms, such as **sequential backward selection**, we use feature extraction to transform or project the data onto a new feature space.

In the context of dimensionality reduction, feature extraction can be understood as an approach to data compression with the goal of maintaining most of the relevant information. In practice, feature extraction is not only used to improve storage space or the computational efficiency of the learning algorithm but can also improve the predictive performance by reducing the **curse of dimensionality**—especially if we are working with non-regularized models.

The main steps in principal component analysis

In this section, we will discuss **principal component analysis** (**PCA**), an unsupervised linear transformation technique that is widely used across different fields, most prominently for feature extraction and dimensionality reduction. Other popular applications of PCA include exploratory data analysis and the denoising of signals in stock market trading, and the analysis of genome data and gene expression levels in the field of bioinformatics.

PCA helps us to identify patterns in data based on the correlation between features. In a nutshell, PCA aims to find the directions of maximum variance in high-dimensional data and projects the data onto a new subspace with equal or fewer dimensions than the original one. The orthogonal axes (principal components) of the new subspace can be interpreted as the directions of maximum variance given the constraint that the new feature axes are orthogonal to each other, as illustrated in *Figure 5.1*:

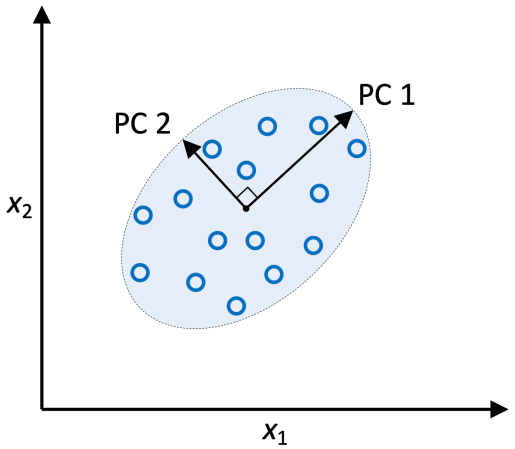


Figure 5.1: Using PCA to find the directions of maximum variance in a dataset

In *Figure 5.1*, x_1 and x_2 are the original feature axes, and **PC 1** and **PC 2** are the principal components.

If we use PCA for dimensionality reduction, we construct a $d \times k$-dimensional transformation matrix, $\boldsymbol{W}$, that allows us to map a vector of the features of the training example, $\boldsymbol{x}$, onto a new k-dimensional feature subspace that has fewer dimensions than the original d-dimensional feature space. For instance, the process is as follows. Suppose we have a feature vector, $\boldsymbol{x}$:

$$\boldsymbol{x} = [x_1, x_2, \dots, x_d], \quad \boldsymbol{x} \in \mathbb{R}^d$$

which is then transformed by a transformation matrix, $\boldsymbol{W} \in \mathbb{R}^{d \times k}$:

$$\boldsymbol{x}\boldsymbol{W} = \boldsymbol{z}$$

resulting in the output vector:

$$\boldsymbol{z} = [z_1, z_2, \dots, z_k], \quad \boldsymbol{z} \in \mathbb{R}^k$$

As a result of transforming the original d-dimensional data onto this new k-dimensional subspace (typically $k << d$), the first principal component will have the largest possible variance. All consequent principal components will have the largest variance given the constraint that these components are uncorrelated (orthogonal) to the other principal components—even if the input features are correlated, the resulting principal components will be mutually orthogonal (uncorrelated). Note that the PCA directions are highly sensitive to data scaling, and we need to standardize the features prior to PCA if the features were measured on different scales and we want to assign equal importance to all features.

Before looking at the PCA algorithm for dimensionality reduction in more detail, let's summarize the approach in a few simple steps:

1. Standardize the d-dimensional dataset.
2. Construct the covariance matrix.
3. Decompose the covariance matrix into its eigenvectors and eigenvalues.
4. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.
5. Select k eigenvectors, which correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$).
6. Construct a projection matrix, $\boldsymbol{W}$, from the "top" k eigenvectors.
7. Transform the d-dimensional input dataset, $\boldsymbol{X}$, using the projection matrix, $\boldsymbol{W}$, to obtain the new k-dimensional feature subspace.

In the following sections, we will perform a PCA step by step using Python as a learning exercise. Then, we will see how to perform a PCA more conveniently using scikit-learn.

Eigendecomposition: Decomposing a Matrix into Eigenvectors and Eigenvalues

Eigendecomposition, the factorization of a square matrix into so-called **eigenvalues** and **eigenvectors**, is at the core of the PCA procedure described in this section.

The covariance matrix is a special case of a square matrix: it's a symmetric matrix, which means that the matrix is equal to its transpose, $A = A^T$.

When we decompose such a symmetric matrix, the eigenvalues are real (rather than complex) numbers, and the eigenvectors are orthogonal (perpendicular) to each other. Furthermore, eigenvalues and eigenvectors come in pairs. If we decompose a covariance matrix into its eigenvectors and eigenvalues, the eigenvectors associated with the highest eigenvalue corresponds to the direction of maximum variance in the dataset. Here, this "direction" is a linear transformation of the dataset's feature columns.

While a more detailed discussion of eigenvalues and eigenvectors is beyond the scope of this book, a relatively thorough treatment with pointers to additional resources can be found on Wikipedia at `https://en.wikipedia.org/wiki/Eigenvalues_and_eigenvectors`.

Extracting the principal components step by step

In this subsection, we will tackle the first four steps of a PCA:

1. Standardizing the data
2. Constructing the covariance matrix
3. Obtaining the eigenvalues and eigenvectors of the covariance matrix
4. Sorting the eigenvalues by decreasing order to rank the eigenvectors

First, we will start by loading the Wine dataset that we worked with in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv(
...     'https://archive.ics.uci.edu/ml/'
...     'machine-learning-databases/wine/wine.data',
...     header=None
... )
```

Obtaining the Wine dataset

You can find a copy of the Wine dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the UCI server at `https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data` is temporarily unavailable. For instance, to load the Wine dataset from a local directory, you can replace the following lines:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases/wine/wine.data',
    header=None
)
```

with these ones:

```
df = pd.read_csv(
    'your/local/path/to/wine.data',
    header=None
)
```

Next, we will process the Wine data into separate training and test datasets—using 70 percent and 30 percent of the data, respectively—and standardize it to unit variance:

```
>>> from sklearn.model_selection import train_test_split
>>> X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y, test_size=0.3,
...                      stratify=y,
...                      random_state=0)
>>> # standardize the features
>>> from sklearn.preprocessing import StandardScaler
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> X_test_std = sc.transform(X_test)
```

After completing the mandatory preprocessing by executing the preceding code, let's advance to the second step: constructing the covariance matrix. The symmetric $d \times d$-dimensional covariance matrix, where d is the number of dimensions in the dataset, stores the pairwise covariances between the different features. For example, the covariance between two features, x_j and x_k, on the population level can be calculated via the following equation:

$$\sigma_{jk} = \frac{1}{n-1}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

Here, μ_j and μ_k are the sample means of features j and k, respectively. Note that the sample means are zero if we standardized the dataset. A positive covariance between two features indicates that the features increase or decrease together, whereas a negative covariance indicates that the features vary in opposite directions. For example, the covariance matrix of three features can then be written as follows (note that Σ is the Greek uppercase letter sigma, which is not to be confused with the summation symbol):

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{bmatrix}$$

The eigenvectors of the covariance matrix represent the principal components (the directions of maximum variance), whereas the corresponding eigenvalues will define their magnitude. In the case of the Wine dataset, we would obtain 13 eigenvectors and eigenvalues from the 13×13-dimensional covariance matrix.

Now, for our third step, let's obtain the eigenpairs of the covariance matrix. If you have taken a linear algebra class, you may have learned that an eigenvector, $\boldsymbol{v}$, satisfies the following condition:

$$\Sigma\boldsymbol{v} = \lambda\boldsymbol{v}$$

Here, λ is a scalar: the eigenvalue. Since the manual computation of eigenvectors and eigenvalues is a somewhat tedious and elaborate task, we will use the `linalg.eig` function from NumPy to obtain the eigenpairs of the Wine covariance matrix:

```
>>> import numpy as np
>>> cov_mat = np.cov(X_train_std.T)
>>> eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
>>> print('\nEigenvalues \n', eigen_vals)
Eigenvalues
[ 4.84274532  2.41602459  1.54845825  0.96120438  0.84166161
  0.6620634   0.51828472  0.34650377  0.3131368   0.10754642
  0.21357215  0.15362835  0.1808613 ]
```

Using the `numpy.cov` function, we computed the covariance matrix of the standardized training dataset. Using the `linalg.eig` function, we performed the eigendecomposition, which yielded a vector (`eigen_vals`) consisting of 13 eigenvalues and the corresponding eigenvectors stored as columns in a 13×13-dimensional matrix (`eigen_vecs`).

Eigendecomposition in NumPy

The `numpy.linalg.eig` function was designed to operate on both symmetric and non-symmetric square matrices. However, you may find that it returns complex eigenvalues in certain cases.

A related function, `numpy.linalg.eigh`, has been implemented to decompose Hermetian matrices, which is a numerically more stable approach to working with symmetric matrices such as the covariance matrix; `numpy.linalg.eigh` always returns real eigenvalues.

Total and explained variance

Since we want to reduce the dimensionality of our dataset by compressing it onto a new feature subspace, we only select the subset of the eigenvectors (principal components) that contains most of the information (variance). The eigenvalues define the magnitude of the eigenvectors, so we have to sort the eigenvalues by decreasing magnitude; we are interested in the top k eigenvectors based on the values of their corresponding eigenvalues. But before we collect those k most informative eigenvectors, let's plot the **variance explained ratios** of the eigenvalues. The variance explained ratio of an eigenvalue, λ_j, is simply the fraction of an eigenvalue, λ_j, and the total sum of the eigenvalues:

$$\text{Explained variance ratio} = \frac{\lambda_j}{\sum_{j=1}^{d}\lambda_j}$$

Using the NumPy `cumsum` function, we can then calculate the cumulative sum of explained variances, which we will then plot via Matplotlib's `step` function:

```
>>> tot = sum(eigen_vals)
>>> var_exp = [(i / tot) for i in
...            sorted(eigen_vals, reverse=True)]
>>> cum_var_exp = np.cumsum(var_exp)
>>> import matplotlib.pyplot as plt
>>> plt.bar(range(1,14), var_exp, align='center',
...         label='Individual explained variance')
>>> plt.step(range(1,14), cum_var_exp, where='mid',
...          label='Cumulative explained variance')
>>> plt.ylabel('Explained variance ratio')
>>> plt.xlabel('Principal component index')
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

The resulting plot indicates that the first principal component alone accounts for approximately 40 percent of the variance.

Also, we can see that the first two principal components combined explain almost 60 percent of the variance in the dataset:

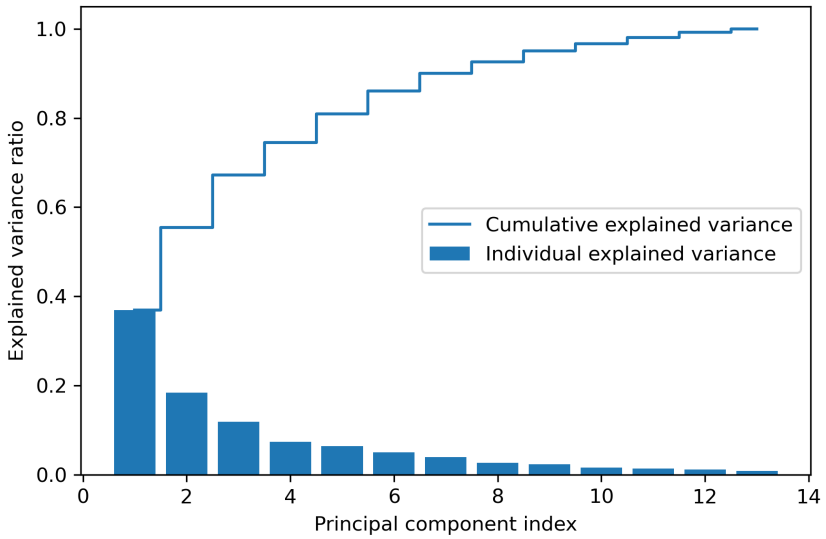


Figure 5.2: The proportion of the total variance captured by the principal components

Although the explained variance plot reminds us of the feature importance values that we computed in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, via random forests, we should remind ourselves that PCA is an unsupervised method, which means that information about the class labels is ignored. Whereas a random forest uses the class membership information to compute the node impurities, variance measures the spread of values along a feature axis.

Feature transformation

Now that we have successfully decomposed the covariance matrix into eigenpairs, let's proceed with the last three steps to transform the Wine dataset onto the new principal component axes. The remaining steps we are going to tackle in this section are the following:

1. Select k eigenvectors, which correspond to the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$).
2. Construct a projection matrix, $\boldsymbol{W}$, from the "top" k eigenvectors.
3. Transform the d-dimensional input dataset, $\boldsymbol{X}$, using the projection matrix, $\boldsymbol{W}$, to obtain the new k-dimensional feature subspace.

Or, in less technical terms, we will sort the eigenpairs by descending order of the eigenvalues, construct a projection matrix from the selected eigenvectors, and use the projection matrix to transform the data onto the lower-dimensional subspace.

We start by sorting the eigenpairs by decreasing order of the eigenvalues:

```
>>> # Make a list of (eigenvalue, eigenvector) tuples
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
...                for i in range(len(eigen_vals))]
>>> # Sort the (eigenvalue, eigenvector) tuples from high to low
>>> eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

Next, we collect the two eigenvectors that correspond to the two largest eigenvalues, to capture about 60 percent of the variance in this dataset. Note that two eigenvectors have been chosen for the purpose of illustration, since we are going to plot the data via a two-dimensional scatterplot later in this subsection. In practice, the number of principal components has to be determined by a tradeoff between computational efficiency and the performance of the classifier:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
...                eigen_pairs[1][1][:, np.newaxis]))
>>> print('Matrix W:\n', w)
Matrix W:
[[-0.13724218   0.50303478]
 [ 0.24724326   0.16487119]
 [-0.02545159   0.24456476]
 [ 0.20694508  -0.11352904]
 [-0.15436582   0.28974518]
```

```
 [-0.39376952   0.05080104]
 [-0.41735106  -0.02287338]
 [ 0.30572896   0.09048885]
 [-0.30668347   0.00835233]
 [ 0.07554066   0.54977581]
 [-0.32613263  -0.20716433]
 [-0.36861022  -0.24902536]
 [-0.29669651   0.38022942]]
```

By executing the preceding code, we have created a 13×2-dimensional projection matrix, W, from the top two eigenvectors.

Mirrored projections

Depending on which versions of NumPy and LAPACK you are using, you may obtain the matrix, W, with its signs flipped. Please note that this is not an issue; if v is an eigenvector of a matrix, Σ, we have:

$$\Sigma v = \lambda v$$

Here, v is the eigenvector, and $-v$ is also an eigenvector, which we can show as follows. Using basic algebra, we can multiply both sides of the equation by a scalar, α:

$$\alpha\Sigma v = \alpha\lambda v$$

Since matrix multiplication is associative for scalar multiplication, we can then rearrange this to the following:

$$\Sigma(\alpha v) = \lambda(\alpha v)$$

Now, we can see that αv is an eigenvector with the same eigenvalue, λ, for both $\alpha = 1$ and $\alpha = -1$. Hence, both v and $-v$ are eigenvectors.

Using the projection matrix, we can now transform an example, x (represented as a 13-dimensional row vector), onto the PCA subspace (the principal components one and two) obtaining x', now a two-dimensional example vector consisting of two new features:

$$x' = xW$$

```
>>> X_train_std[0].dot(w)
array([ 2.38299011,  0.45458499])
```

Similarly, we can transform the entire 124×13-dimensional training dataset onto the two principal components by calculating the matrix dot product:

$$X' = XW$$

```
>>> X_train_pca = X_train_std.dot(w)
```

Lastly, let's visualize the transformed Wine training dataset, now stored as an 124×2-dimensional matrix, in a two-dimensional scatterplot:

```
>>> colors = ['r', 'b', 'g']
>>> markers = ['o', 's', '^']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_pca[y_train==l, 0],
...                 X_train_pca[y_train==l, 1],
...                 c=c, label=f'Class {l}', marker=m)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in *Figure 5.3*, the data is more spread along the first principal component (*x* axis) than the second principal component (*y* axis), which is consistent with the explained variance ratio plot that we created in the previous subsection. However, we can tell that a linear classifier will likely be able to separate the classes well:

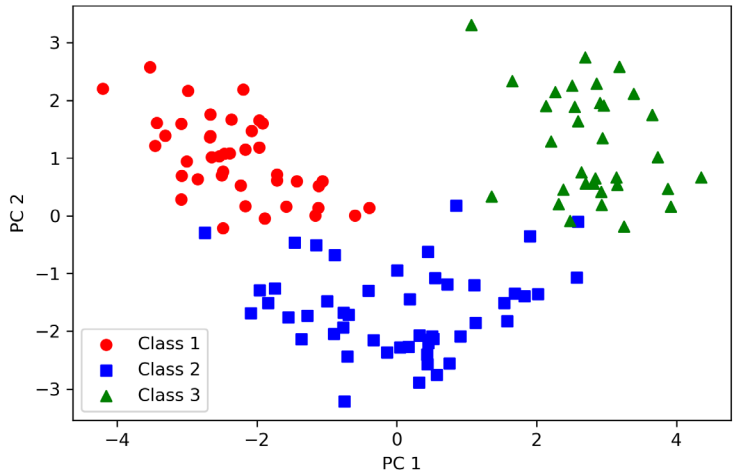


Figure 5.3: Data records from the Wine dataset projected onto a 2D feature space via PCA

Although we encoded the class label information for the purpose of illustration in the preceding scatterplot, we have to keep in mind that PCA is an unsupervised technique that doesn't use any class label information.

Principal component analysis in scikit-learn

Although the verbose approach in the previous subsection helped us to follow the inner workings of PCA, we will now discuss how to use the `PCA` class implemented in scikit-learn.

The `PCA` class is another one of scikit-learn's transformer classes, with which we first fit the model using the training data before we transform both the training data and the test dataset using the same model parameters. Now, let's use the `PCA` class from scikit-learn on the Wine training dataset, classify the transformed examples via logistic regression, and visualize the decision regions via the `plot_decision_regions` function that we defined in *Chapter 2*, *Training Simple Machine Learning Algorithms for Classification*:

```
from matplotlib.colors import ListedColormap
def plot_decision_regions(X, y, classifier, test_idx=None, resolution=0.02):

    # setup marker generator and color map
    markers = ('o', 's', '^', 'v', '<')
    colors = ('red', 'blue', 'lightgreen', 'gray', 'cyan')
    cmap = ListedColormap(colors[:len(np.unique(y))])

    # plot the decision surface
    x1_min, x1_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    x2_min, x2_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, resolution),
                           np.arange(x2_min, x2_max, resolution))
    lab = classifier.predict(np.array([xx1.ravel(), xx2.ravel()]).T)
    lab = lab.reshape(xx1.shape)
    plt.contourf(xx1, xx2, lab, alpha=0.3, cmap=cmap)
    plt.xlim(xx1.min(), xx1.max())
    plt.ylim(xx2.min(), xx2.max())

    # plot class examples
    for idx, cl in enumerate(np.unique(y)):
        plt.scatter(x=X[y == cl, 0],
                    y=X[y == cl, 1],
                    alpha=0.8,
                    c=colors[idx],
                    marker=markers[idx],
                    label=f'Class {cl}',
                    edgecolor='black')
```

For your convenience, you can place the preceding `plot_decision_regions` code into a separate code file in your current working directory, for example, `plot_decision_regions_script.py`, and import it into your current Python session:

```
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.decomposition import PCA
>>> # initializing the PCA transformer and
>>> # logistic regression estimator:
>>> pca = PCA(n_components=2)
>>> lr = LogisticRegression(multi_class='ovr',
...                         random_state=1,
...                         solver='lbfgs')
>>> # dimensionality reduction:
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> X_test_pca = pca.transform(X_test_std)
>>> # fitting the logistic regression model on the reduced dataset:
>>> lr.fit(X_train_pca, y_train)
>>> plot_decision_regions(X_train_pca, y_train, classifier=lr)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

By executing this code, we should now see the decision regions for the training data reduced to two principal component axes:

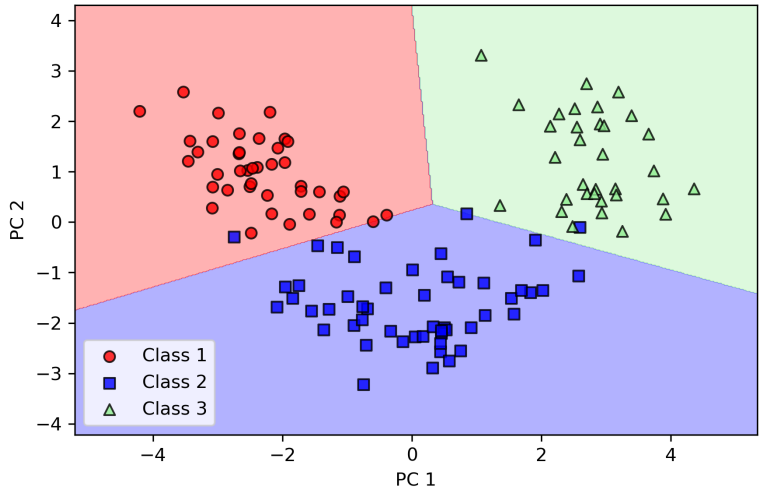


Figure 5.4: Training examples and logistic regression decision regions after using scikit-learn's PCA for dimensionality reduction

When we compare the PCA projections via scikit-learn with our own PCA implementation, we might see that the resulting plots are mirror images of each other. Note that this is not due to an error in either of those two implementations; the reason for this difference is that, depending on the eigensolver, eigenvectors can have either negative or positive signs.

Not that it matters, but we could simply revert the mirror image by multiplying the data by –1 if we wanted to; note that eigenvectors are typically scaled to unit length 1. For the sake of completeness, let's plot the decision regions of the logistic regression on the transformed test dataset to see if it can separate the classes well:

```
>>> plot_decision_regions(X_test_pca, y_test, classifier=lr)
>>> plt.xlabel('PC 1')
>>> plt.ylabel('PC 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

After we plot the decision regions for the test dataset by executing the preceding code, we can see that logistic regression performs quite well on this small two-dimensional feature subspace and only misclassifies a few examples in the test dataset:

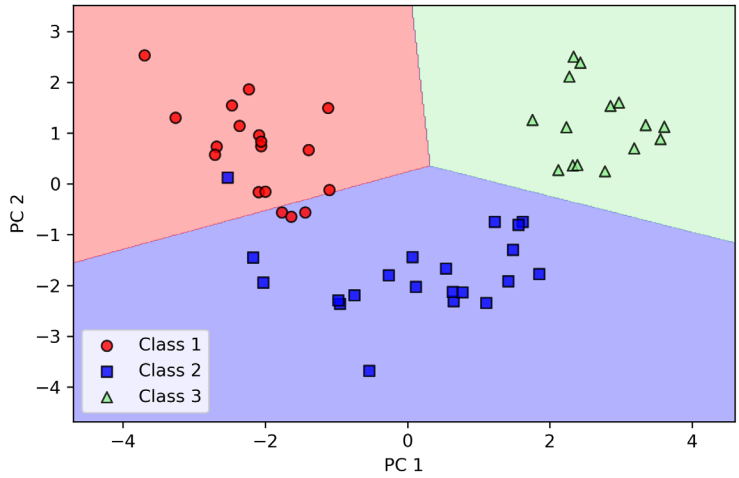


Figure 5.5: Test datapoints with logistic regression decision regions in the PCA-based feature space

If we are interested in the explained variance ratios of the different principal components, we can simply initialize the `PCA` class with the `n_components` parameter set to `None`, so all principal components are kept and the explained variance ratio can then be accessed via the `explained_variance_ratio_` attribute:

```
>>> pca = PCA(n_components=None)
>>> X_train_pca = pca.fit_transform(X_train_std)
>>> pca.explained_variance_ratio_
array([ 0.36951469, 0.18434927, 0.11815159, 0.07334252,
```

```
        0.06422108, 0.05051724, 0.03954654, 0.02643918,
        0.02389319, 0.01629614, 0.01380021, 0.01172226,
        0.00820609])
```

Note that we set `n_components=None` when we initialized the `PCA` class so that it will return all principal components in a sorted order, instead of performing a dimensionality reduction.

Assessing feature contributions

In this section, we will take a brief look at how we can assess the contributions of the original features to the principal components. As we learned, via PCA, we create principal components that represent linear combinations of the features. Sometimes, we are interested to know about how much each original feature contributes to a given principal component. These contributions are often called **loadings**.

The factor loadings can be computed by scaling the eigenvectors by the square root of the eigenvalues. The resulting values can then be interpreted as the correlation between the original features and the principal component. To illustrate this, let us plot the loadings for the first principal component.

First, we compute the 13×13-dimensional loadings matrix by multiplying the eigenvectors by the square root of the eigenvalues:

```
>>> loadings = eigen_vecs * np.sqrt(eigen_vals)
```

Then, we plot the loadings for the first principal component, `loadings[:, 0]`, which is the first column in this matrix:

```
>>> fig, ax = plt.subplots()
>>> ax.bar(range(13), loadings[:, 0], align='center')
>>> ax.set_ylabel('Loadings for PC 1')
>>> ax.set_xticks(range(13))
>>> ax.set_xticklabels(df_wine.columns[1:], rotation=90)
>>> plt.ylim([-1, 1])
>>> plt.tight_layout()
>>> plt.show()
```

In *Figure 5.6*, we can see that, for example, **Alcohol** has a negative correlation with the first principal component (approximately –0.3), whereas **Malic acid** has a positive correlation (approximately 0.54). Note that a value of 1 describes a perfect positive correlation whereas a value of –1 corresponds to a perfect negative correlation:

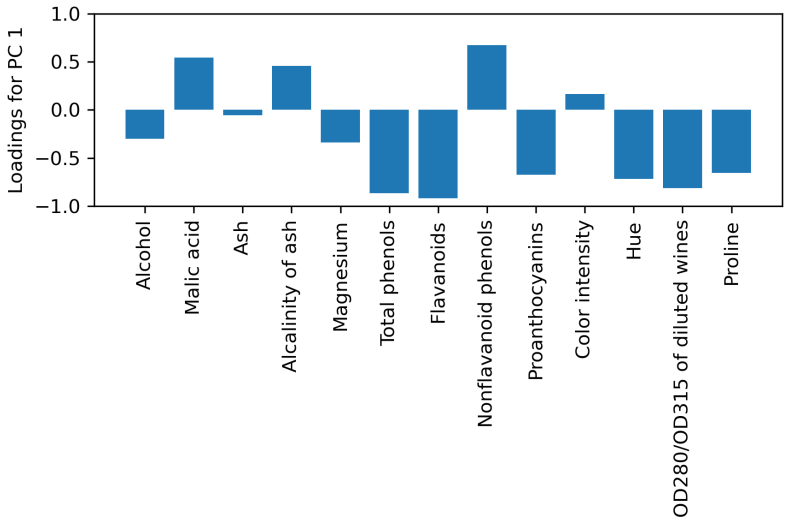


Figure 5.6: Feature correlations with the first principal component

In the preceding code example, we compute the factor loadings for our own PCA implementation. We can obtain the loadings from a fitted scikit-learn PCA object in a similar manner, where `pca.components_` represents the eigenvectors and `pca.explained_variance_` represents the eigenvalues:

```
>>> sklearn_loadings = pca.components_.T * np.sqrt(pca.explained_variance_)
```

To compare the scikit-learn PCA loadings with those we created previously, let us create a similar bar plot:

```
>>> fig, ax = plt.subplots()
>>> ax.bar(range(13), sklearn_loadings[:, 0], align='center')
>>> ax.set_ylabel('Loadings for PC 1')
>>> ax.set_xticks(range(13))
>>> ax.set_xticklabels(df_wine.columns[1:], rotation=90)
>>> plt.ylim([-1, 1])
>>> plt.tight_layout()
>>> plt.show()
```

As we can see, the bar plots look the same:

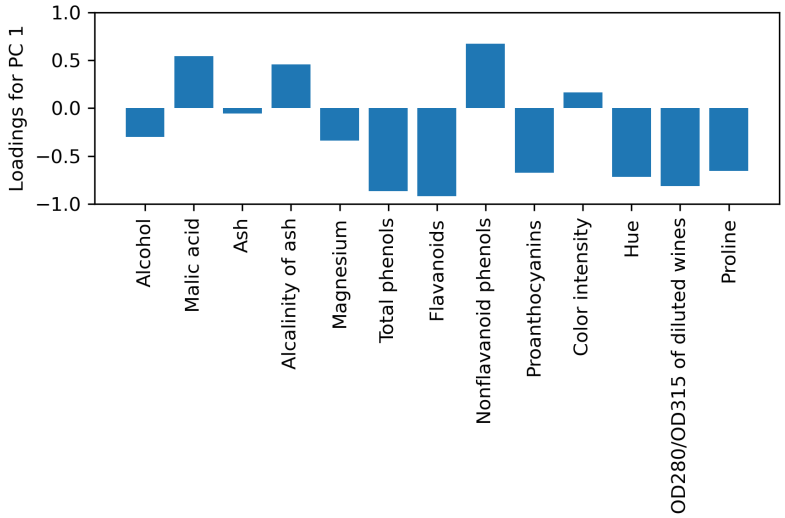


Figure 5.7: Feature correlations to the first principal component using scikit-learn

After exploring PCA as an unsupervised feature extraction technique, the next section will introduce **linear discriminant analysis** (**LDA**), which is a linear transformation technique that takes class label information into account.

Supervised data compression via linear discriminant analysis

LDA can be used as a technique for feature extraction to increase computational efficiency and reduce the degree of overfitting due to the curse of dimensionality in non-regularized models. The general concept behind LDA is very similar to PCA, but whereas PCA attempts to find the orthogonal component axes of maximum variance in a dataset, the goal in LDA is to find the feature subspace that optimizes class separability. In the following sections, we will discuss the similarities between LDA and PCA in more detail and walk through the LDA approach step by step.

Principal component analysis versus linear discriminant analysis

Both PCA and LDA are *linear transformation techniques* that can be used to reduce the number of dimensions in a dataset; the former is an unsupervised algorithm, whereas the latter is supervised. Thus, we might think that LDA is a superior feature extraction technique for classification tasks compared to PCA. However, A.M. Martinez reported that preprocessing via PCA tends to result in better classification results in an image recognition task in certain cases, for instance, if each class consists of only a small number of examples (*PCA Versus LDA* by *A. M. Martinez* and *A. C. Kak*, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(2): 228-233, 2001).

Fisher LDA

LDA is sometimes also called **Fisher's LDA**. Ronald A. Fisher initially formulated *Fisher's Linear Discriminant* for two-class classification problems in 1936 (*The Use of Multiple Measurements in Taxonomic Problems*, *R. A. Fisher*, *Annals of Eugenics*, 7(2): 179-188, 1936). In 1948, Fisher's linear discriminant was generalized for multiclass problems by C. Radhakrishna Rao under the assumption of equal class covariances and normally distributed classes, which we now call LDA (*The Utilization of Multiple Measurements in Problems of Biological Classification* by *C. R. Rao*, *Journal of the Royal Statistical Society*. Series B (Methodological), 10(2): 159-203, 1948).

Figure 5.8 summarizes the concept of LDA for a two-class problem. Examples from class 1 are shown as circles, and examples from class 2 are shown as crosses:

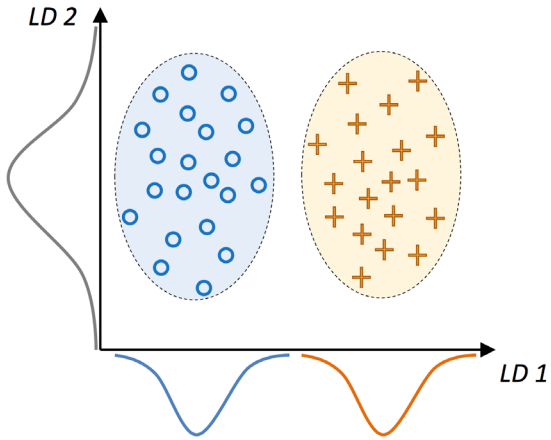


Figure 5.8: The concept of LDA for a two-class problem

A linear discriminant, as shown on the *x* axis (*LD 1*), would separate the two normal distributed classes well. Although the exemplary linear discriminant shown on the *y* axis (*LD 2*) captures a lot of the variance in the dataset, it would fail as a good linear discriminant since it does not capture any of the class-discriminatory information.

One assumption in LDA is that the data is normally distributed. Also, we assume that the classes have identical covariance matrices and that the training examples are statistically independent of each other. However, even if one, or more, of those assumptions is (slightly) violated, LDA for dimensionality reduction can still work reasonably well (*Pattern Classification 2nd Edition* by *R. O. Duda*, *P. E. Hart*, and *D. G. Stork*, *New York*, 2001).

The inner workings of linear discriminant analysis

Before we dive into the code implementation, let's briefly summarize the main steps that are required to perform LDA:

1. Standardize the d-dimensional dataset (d is the number of features).
2. For each class, compute the d-dimensional mean vector.
3. Construct the between-class scatter matrix, $\boldsymbol{S}_B$, and the within-class scatter matrix, $\boldsymbol{S}_W$.
4. Compute the eigenvectors and corresponding eigenvalues of the matrix, $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$.
5. Sort the eigenvalues by decreasing order to rank the corresponding eigenvectors.
6. Choose the k eigenvectors that correspond to the k largest eigenvalues to construct a $d \times k$-dimensional transformation matrix, $\boldsymbol{W}$; the eigenvectors are the columns of this matrix.
7. Project the examples onto the new feature subspace using the transformation matrix, $\boldsymbol{W}$.

As we can see, LDA is quite similar to PCA in the sense that we are decomposing matrices into eigenvalues and eigenvectors, which will form the new lower-dimensional feature space. However, as mentioned before, LDA takes class label information into account, which is represented in the form of the mean vectors computed in *step 2*. In the following sections, we will discuss these seven steps in more detail, accompanied by illustrative code implementations.

Computing the scatter matrices

Since we already standardized the features of the Wine dataset in the PCA section at the beginning of this chapter, we can skip the first step and proceed with the calculation of the mean vectors, which we will use to construct the within-class scatter matrix and between-class scatter matrix, respectively. Each mean vector, $\boldsymbol{m}_i$, stores the mean feature value, μ_m, with respect to the examples of class i:

$$\boldsymbol{m}_i = \frac{1}{n_i} \sum_{\boldsymbol{x} \in D_i} \boldsymbol{x}_m$$

This results in three mean vectors:

$$\boldsymbol{m}_i = \begin{bmatrix} \mu_{i,alcohol} \\ \mu_{i,malic\ acid} \\ \vdots \\ \mu_{i,proline} \end{bmatrix}^T \quad i \in \{1,2,3\}$$

These mean vectors can be computed by the following code, where we compute one mean vector for each of the three labels:

```
>>> np.set_printoptions(precision=4)
>>> mean_vecs = []
>>> for label in range(1,4):
...     mean_vecs.append(np.mean(
...                 X_train_std[y_train==label], axis=0))
...     print(f'MV {label}: {mean_vecs[label - 1]}\n')
```

```
MV 1: [ 0.9066  -0.3497  0.3201  -0.7189  0.5056  0.8807  0.9589  -0.5516
0.5416  0.2338  0.5897  0.6563  1.2075]
MV 2: [-0.8749  -0.2848  -0.3735  0.3157  -0.3848  -0.0433  0.0635  -0.0946
0.0703  -0.8286  0.3144  0.3608  -0.7253]
MV 3: [ 0.1992  0.866  0.1682  0.4148  -0.0451  -1.0286  -1.2876  0.8287
-0.7795  0.9649  -1.209  -1.3622  -0.4013]
```

Using the mean vectors, we can now compute the within-class scatter matrix, $\boldsymbol{S}_W$:

$$\boldsymbol{S}_W = \sum_{i=1}^{c} \boldsymbol{S}_i$$

This is calculated by summing up the individual scatter matrices, $\boldsymbol{S}_i$, of each individual class i:

$$\boldsymbol{S}_i = \sum_{\boldsymbol{x} \in D_i} (\boldsymbol{x} - \boldsymbol{m}_i)(\boldsymbol{x} - \boldsymbol{m}_i)^T$$

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label, mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.zeros((d, d))
...     for row in X_train_std[y_train == label]:
...         row, mv = row.reshape(d, 1), mv.reshape(d, 1)
...         class_scatter += (row - mv).dot((row - mv).T)
...     S_W += class_scatter
>>> print('Within-class scatter matrix: '
...       f'{S_W.shape[0]}x{S_W.shape[1]}')
Within-class scatter matrix: 13x13
```

The assumption that we are making when we are computing the scatter matrices is that the class labels in the training dataset are uniformly distributed. However, if we print the number of class labels, we see that this assumption is violated:

```
>>> print('Class label distribution:',
...       np.bincount(y_train)[1:])
Class label distribution: [41 50 33]
```

Thus, we want to scale the individual scatter matrices, $\boldsymbol{S}_i$, before we sum them up as the scatter matrix, $\boldsymbol{S}_W$. When we divide the scatter matrices by the number of class-examples, n_i, we can see that computing the scatter matrix is in fact the same as computing the covariance matrix, Σ_i—the covariance matrix is a normalized version of the scatter matrix:

$$\Sigma_i = \frac{1}{n_i} \boldsymbol{S}_i = \frac{1}{n_i} \sum_{\boldsymbol{x} \in D_i} (\boldsymbol{x} - \boldsymbol{m}_i)(\boldsymbol{x} - \boldsymbol{m}_i)^T$$

The code for computing the scaled within-class scatter matrix is as follows:

```
>>> d = 13 # number of features
>>> S_W = np.zeros((d, d))
>>> for label,mv in zip(range(1, 4), mean_vecs):
...     class_scatter = np.cov(X_train_std[y_train==label].T)
...     S_W += class_scatter
>>> print('Scaled within-class scatter matrix: '
...       f'{S_W.shape[0]}x{S_W.shape[1]}')
Scaled within-class scatter matrix: 13x13
```

After we compute the scaled within-class scatter matrix (or covariance matrix), we can move on to the next step and compute the between-class scatter matrix $\boldsymbol{S}_B$:

$$\boldsymbol{S}_B = \sum_{i=1}^{c} n_i(\boldsymbol{m}_i - \boldsymbol{m})(\boldsymbol{m}_i - \boldsymbol{m})^T$$

Here, $\boldsymbol{m}$ is the overall mean that is computed, including examples from all $\boldsymbol{c}$ classes:

```
>>> mean_overall = np.mean(X_train_std, axis=0)
>>> mean_overall = mean_overall.reshape(d, 1)

>>> d = 13 # number of features
>>> S_B = np.zeros((d, d))
>>> for i, mean_vec in enumerate(mean_vecs):
...     n = X_train_std[y_train == i + 1, :].shape[0]
...     mean_vec = mean_vec.reshape(d, 1) # make column vector
...     S_B += n * (mean_vec - mean_overall).dot(
...     (mean_vec - mean_overall).T)
>>> print('Between-class scatter matrix: '
...       f'{S_B.shape[0]}x{S_B.shape[1]}')
Between-class scatter matrix: 13x13
```

Selecting linear discriminants for the new feature subspace

The remaining steps of the LDA are similar to the steps of the PCA. However, instead of performing the eigendecomposition on the covariance matrix, we solve the generalized eigenvalue problem of the matrix, $\boldsymbol{S}_W^{-1}\boldsymbol{S}_B$:

```
>>> eigen_vals, eigen_vecs =\
...     np.linalg.eig(np.linalg.inv(S_W).dot(S_B))
```

After we compute the eigenpairs, we can sort the eigenvalues in descending order:

```
>>> eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:,i])
...                for i in range(len(eigen_vals))]
```

```
>>> eigen_pairs = sorted(eigen_pairs,
...                      key=lambda k: k[0], reverse=True)
>>> print('Eigenvalues in descending order:\n')
>>> for eigen_val in eigen_pairs:
...     print(eigen_val[0])
Eigenvalues in descending order:
349.617808906
172.76152219
3.78531345125e-14
2.11739844822e-14
1.51646188942e-14
1.51646188942e-14
1.35795671405e-14
1.35795671405e-14
7.58776037165e-15
5.90603998447e-15
5.90603998447e-15
2.25644197857e-15
0.0
```

In LDA, the number of linear discriminants is at most $c - 1$, where c is the number of class labels, since the in-between scatter matrix, S_B, is the sum of c matrices with rank one or less. We can indeed see that we only have two nonzero eigenvalues (the eigenvalues 3-13 are not exactly zero, but this is due to the floating-point arithmetic in NumPy.)

Collinearity

Note that in the rare case of perfect collinearity (all aligned example points fall on a straight line), the covariance matrix would have rank one, which would result in only one eigenvector with a nonzero eigenvalue.

To measure how much of the class-discriminatory information is captured by the linear discriminants (eigenvectors), let's plot the linear discriminants by decreasing eigenvalues, similar to the explained variance plot that we created in the PCA section. For simplicity, we will call the content of class-discriminatory information **discriminability**:

```
>>> tot = sum(eigen_vals.real)
>>> discr = [(i / tot) for i in sorted(eigen_vals.real,
...                                    reverse=True)]
>>> cum_discr = np.cumsum(discr)
>>> plt.bar(range(1, 14), discr, align='center',
...         label='Individual discriminability')
```

```
>>> plt.step(range(1, 14), cum_discr, where='mid',
...          label='Cumulative discriminability')
>>> plt.ylabel('"Discriminability" ratio')
>>> plt.xlabel('Linear Discriminants')
>>> plt.ylim([-0.1, 1.1])
>>> plt.legend(loc='best')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in *Figure 5.9*, the first two linear discriminants alone capture 100 percent of the useful information in the Wine training dataset:

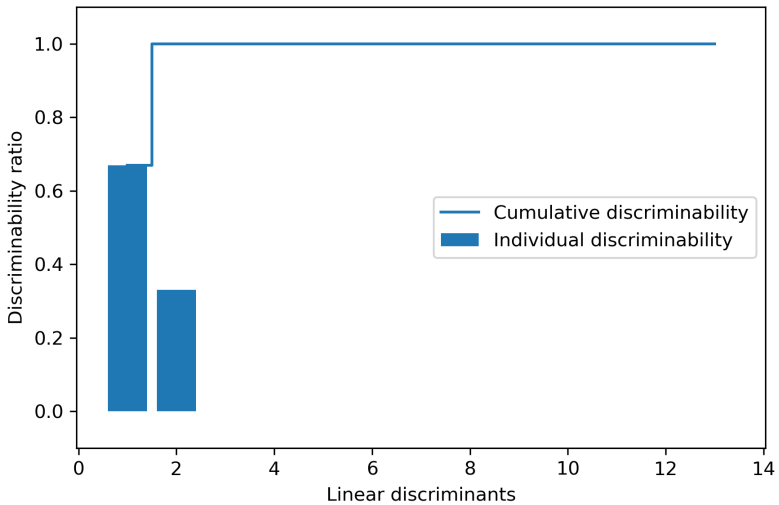


Figure 5.9: The top two discriminants capture 100 percent of the useful information

Let's now stack the two most discriminative eigenvector columns to create the transformation matrix, W:

```
>>> w = np.hstack((eigen_pairs[0][1][:, np.newaxis].real,
...                eigen_pairs[1][1][:, np.newaxis].real))
>>> print('Matrix W:\n', w)
Matrix W:
 [[-0.1481  -0.4092]
  [ 0.0908  -0.1577]
  [-0.0168  -0.3537]
  [ 0.1484   0.3223]
  [-0.0163  -0.0817]
  [ 0.1913   0.0842]
  [-0.7338   0.2823]
  [-0.075   -0.0102]]
```

```
  [ 0.0018   0.0907]
  [ 0.294   -0.2152]
  [-0.0328   0.2747]
  [-0.3547  -0.0124]
  [-0.3915  -0.5958]]
```

Projecting examples onto the new feature space

Using the transformation matrix W that we created in the previous subsection, we can now transform the training dataset by multiplying the matrices:

$$X' = XW$$

```
>>> X_train_lda = X_train_std.dot(w)
>>> colors = ['r', 'b', 'g']
>>> markers = ['o', 's', '^']
>>> for l, c, m in zip(np.unique(y_train), colors, markers):
...     plt.scatter(X_train_lda[y_train==l, 0],
...                 X_train_lda[y_train==l, 1] * (-1),
...                 c=c, label= f'Class {l}', marker=m)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower right')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in *Figure 5.10*, the three Wine classes are now perfectly linearly separable in the new feature subspace:

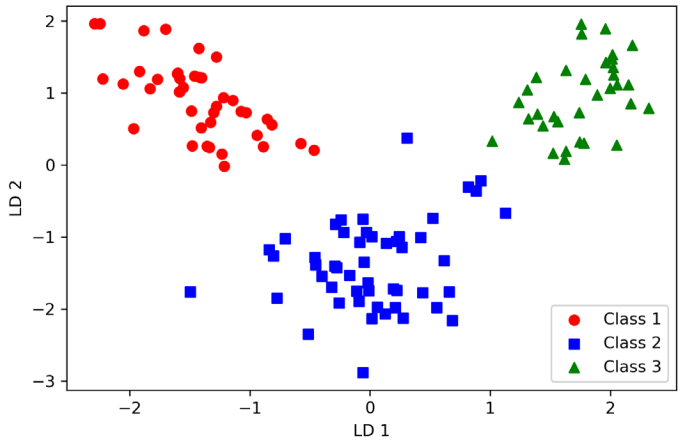


Figure 5.10: Wine classes perfectly separable after projecting the data onto the first two discriminants

LDA via scikit-learn

That step-by-step implementation was a good exercise to understand the inner workings of LDA and understand the differences between LDA and PCA. Now, let's look at the `LDA` class implemented in scikit-learn:

```
>>> # the following import statement is one line
>>> from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
>>> lda = LDA(n_components=2)
>>> X_train_lda = lda.fit_transform(X_train_std, y_train)
```

Next, let's see how the logistic regression classifier handles the lower-dimensional training dataset after the LDA transformation:

```
>>> lr = LogisticRegression(multi_class='ovr', random_state=1,
...                         solver='lbfgs')
>>> lr = lr.fit(X_train_lda, y_train)
>>> plot_decision_regions(X_train_lda, y_train, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

Looking at *Figure 5.11*, we can see that the logistic regression model misclassifies one of the examples from class 2:

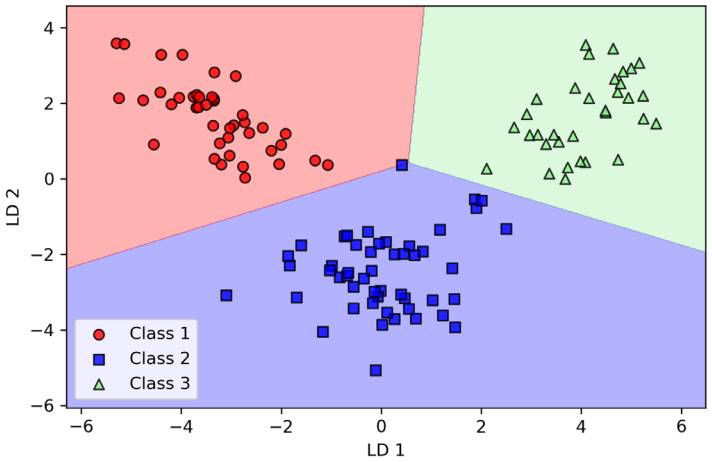


Figure 5.11: The logistic regression model misclassifies one of the classes

By lowering the regularization strength, we could probably shift the decision boundaries so that the logistic regression model classifies all examples in the training dataset correctly. However, and more importantly, let's take a look at the results on the test dataset:

```
>>> X_test_lda = lda.transform(X_test_std)
>>> plot_decision_regions(X_test_lda, y_test, classifier=lr)
>>> plt.xlabel('LD 1')
>>> plt.ylabel('LD 2')
>>> plt.legend(loc='lower left')
>>> plt.tight_layout()
>>> plt.show()
```

As we can see in *Figure 5.12*, the logistic regression classifier is able to get a perfect accuracy score for classifying the examples in the test dataset by only using a two-dimensional feature subspace, instead of the original 13 Wine features:

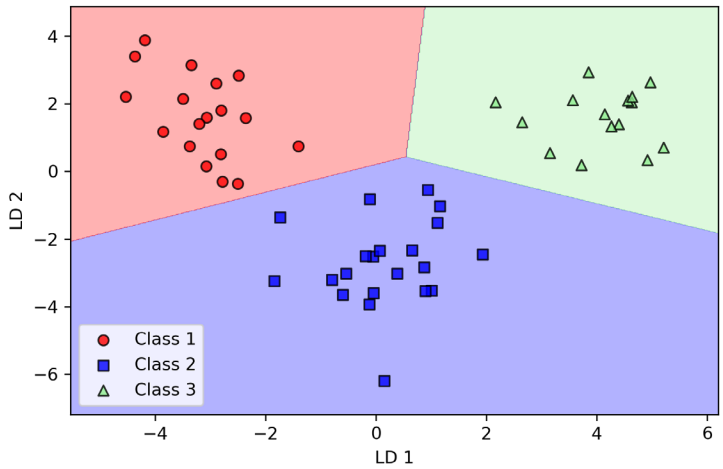


Figure 5.12: The logistic regression model works perfectly on the test data

Nonlinear dimensionality reduction and visualization

In the previous section, we covered linear transformation techniques, such as PCA and LDA, for feature extraction. In this section, we will discuss why considering nonlinear dimensionality reduction techniques might be worthwhile.

One nonlinear dimensionality reduction technique that is particularly worth highlighting is **t-distributed stochastic neighbor embedding** (**t-SNE**) since it is frequently used in literature to visualize high-dimensional datasets in two or three dimensions. We will see how we can apply t-SNE to plot images of handwritten images in a 2-dimensional feature space.

Why consider nonlinear dimensionality reduction?

Many machine learning algorithms make assumptions about the linear separability of the input data.

You have learned that the perceptron even requires perfectly linearly separable training data to converge. Other algorithms that we have covered so far assume that the lack of perfect linear separability is due to noise: Adaline, logistic regression, and the (standard) SVM to just name a few.

However, if we are dealing with nonlinear problems, which we may encounter rather frequently in real-world applications, linear transformation techniques for dimensionality reduction, such as PCA and LDA, may not be the best choice:

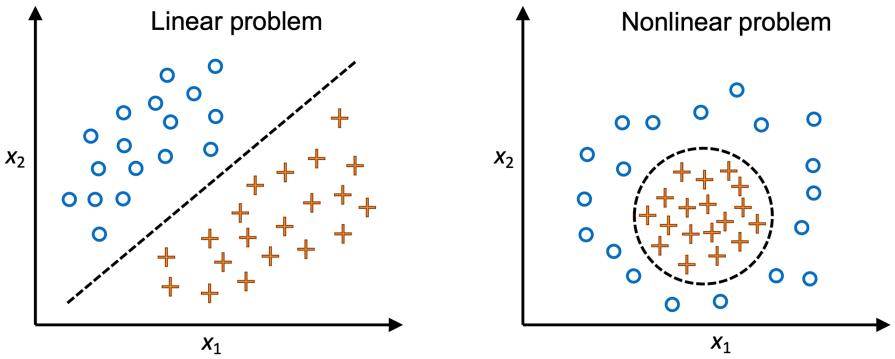


Figure 5.13: The difference between linear and nonlinear problems

The scikit-learn library implements a selection of advanced techniques for nonlinear dimensionality reduction that are beyond the scope of this book. The interested reader can find a nice overview of the current implementations in scikit-learn, complemented by illustrative examples, at `http://scikit-learn.org/stable/modules/manifold.html`.

The development and application of nonlinear dimensionality reduction techniques is also often referred to as manifold learning, where a manifold refers to a lower dimensional topological space embedded in a high-dimensional space. Algorithms for manifold learning have to capture the complicated structure of the data in order to project it onto a lower-dimensional space where the relationship between data points is preserved.

A classic example of manifold learning is the 3-dimensional Swiss roll illustrated in *Figure 5.14*:

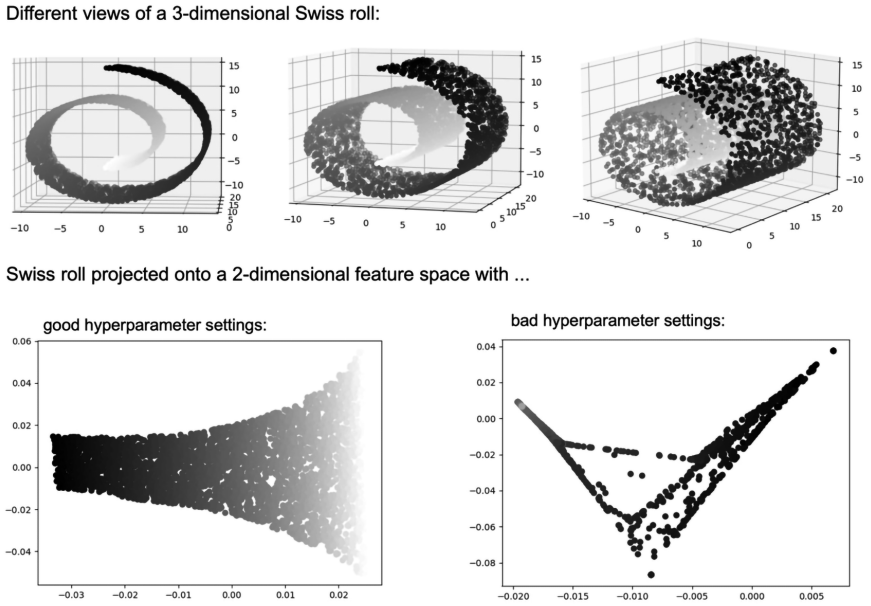


Figure 5.14: Three-dimensional Swiss roll projected into a lower, two-dimensional space

While nonlinear dimensionality reduction and manifold learning algorithms are very powerful, we should note that these techniques are notoriously hard to use, and with non-ideal hyperparameter choices, they may cause more harm than good. The reason behind this difficulty is that we are often working with high-dimensional datasets that we cannot readily visualize and where the structure is not obvious (unlike the Swiss roll example in *Figure 5.14*). Moreover, unless we project the dataset into two or three dimensions (which is often not sufficient for capturing more complicated relationships), it is hard or even impossible to assess the quality of the results. Hence, many people still rely on simpler techniques such as PCA and LDA for dimensionality reduction.

Visualizing data via t-distributed stochastic neighbor embedding

After introducing nonlinear dimensionality reduction and discussing some of its challenges, let's take a look at a hands-on example involving t-SNE, which is often used for visualizing complex datasets in two or three dimensions.

In a nutshell, t-SNE is modeling data points based on their pair-wise distances in the high-dimensional (original) feature space. Then, it finds a probability distribution of pair-wise distances in the new, lower-dimensional space that is close to the probability distribution of pair-wise distances in the original space. Or, in other words, t-SNE learns to embed data points into a lower-dimensional space such that the pairwise distances in the original space are preserved. You can find more details about this method in the original research paper *Visualizing data using t-SNE* by *Maaten and Hinton, Journal of Machine Learning Research*, 2018 (`https://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf`). However, as the research paper title suggests, t-SNE is a technique intended for visualization purposes as it requires the whole dataset for the projection. Since it projects the points directly (unlike PCA, it does not involve a projection matrix), we cannot apply t-SNE to new data points.

The following code shows a quick demonstration of how t-SNE can be applied to a 64-dimensional dataset. First, we load the Digits dataset from scikit-learn, which consists of low-resolution handwritten digits (the numbers 0-9):

```
>>> from sklearn.datasets import load_digits
>>> digits = load_digits()
```

The digits are 8×8 grayscale images. The following code plots the first four images in the dataset, which consists of 1,797 images in total:

```
>>> fig, ax = plt.subplots(1, 4)
>>> for i in range(4):
>>>     ax[i].imshow(digits.images[i], cmap='Greys')
>>> plt.show()
```

As we can see in *Figure 5.15*, the images are relatively low resolution, 8×8 pixels (that is, 64 pixels per image):

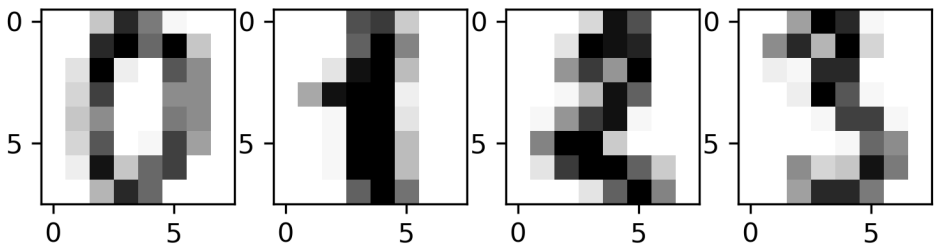


Figure 5.15: Low resolution images of handwritten digits

Note that the `digits.data` attribute lets us access a tabular version of this dataset where the examples are represented by the rows, and the columns correspond to the pixels:

```
>>> digits.data.shape
(1797, 64)
```

Next, let us assign the features (pixels) to a new variable `X_digits` and the labels to another new variable `y_digits`:

```
>>> y_digits = digits.target
>>> X_digits = digits.data
```

Then, we import the t-SNE class from scikit-learn and fit a new `tsne` object. Using `fit_transform`, we perform the t-SNE fitting and data transformation in one step:

```
>>> from sklearn.manifold import TSNE
>>> tsne = TSNE(n_components=2, init='pca',
...             random_state=123)
>>> X_digits_tsne = tsne.fit_transform(X_digits)
```

Using this code, we projected the 64-dimensional dataset onto a 2-dimensional space. We specified `init='pca'`, which initializes the t-SNE embedding using PCA as it is recommended in the research article *Initialization is critical for preserving global data structure in both t-SNE and UMAP* by *Kobak* and *Linderman*, *Nature Biotechnology Volume 39*, pages 156–157, 2021 (`https://www.nature.com/articles/s41587-020-00809-z`).

Note that t-SNE includes additional hyperparameters such as the perplexity and learning rate (often called **epsilon**), which we omitted in the example (we used the scikit-learn default values). In practice, we recommend you explore these parameters as well. More information about these parameters and their effects on the results can be found in the excellent article *How to Use t-SNE Effectively* by *Wattenberg*, *Viegas*, and *Johnson*, *Distill*, 2016 (`https://distill.pub/2016/misread-tsne/`).

Finally, let us visualize the 2D t-SNE embeddings using the following code:

```
>>> import matplotlib.patheffects as PathEffects
>>> def plot_projection(x, colors):

...     f = plt.figure(figsize=(8, 8))
...     ax = plt.subplot(aspect='equal')
...     for i in range(10):
...         plt.scatter(x[colors == i, 0],
...                     x[colors == i, 1])

...     for i in range(10):
...         xtext, ytext = np.median(x[colors == i, :], axis=0)
...         txt = ax.text(xtext, ytext, str(i), fontsize=24)
...         txt.set_path_effects([
...             PathEffects.Stroke(linewidth=5, foreground="w"),
...             PathEffects.Normal()])
```

```
>>> plot_projection(X_digits_tsne, y_digits)
>>> plt.show()
```

Like PCA, t-SNE is an unsupervised method, and in the preceding code, we use the class labels `y_digits` (0-9) only for visualization purposes via the functions color argument. Matplotlib's `PathEffects` are used for visual purposes, such that the class label is displayed in the center (via `np.median`) of data points belonging to each respective digit. The resulting plot is as follows:

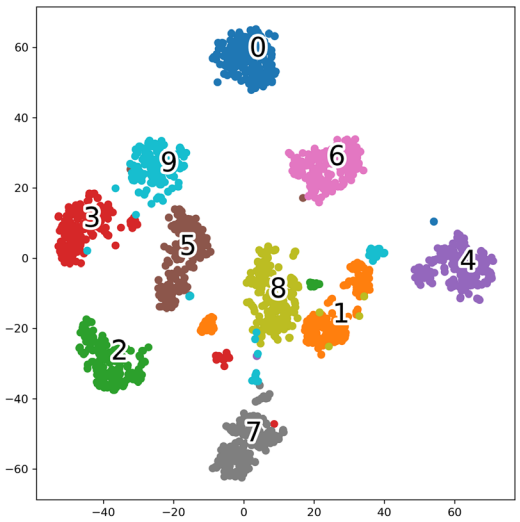


Figure 5.16: A visualization of how t-SNE embeds the handwritten digits in a 2D feature space

As we can see, t-SNE is able to separate the different digits (classes) nicely, although not perfectly. It might be possible to achieve better separation by tuning the hyperparameters. However, a certain degree of class mixing might be unavoidable due to illegible handwriting. For instance, by inspecting individual images, we might find that certain instances of the number 3 indeed look like the number 9, and so forth.

Uniform manifold approximation and projection

Another popular visualization technique is **uniform manifold approximation and projection** (**UMAP**). While UMAP can produce similarly good results as t-SNE (for example, see the Kobak and Linderman paper referenced previously), it is typically faster, and it can also be used to project new data, which makes it more attractive as a dimensionality reduction technique in a machine learning context, similar to PCA. Interested readers can find more information about UMAP in the original paper: *UMAP: Uniform manifold approximation and projection for dimension reduction* by *McInnes, Healy*, and *Melville*, 2018 (`https://arxiv.org/abs/1802.03426`). A scikit-learn compatible implementation of UMAP can be found at `https://umap-learn.readthedocs.io`.

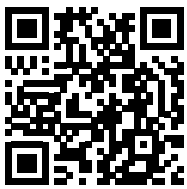
Summary

In this chapter, you learned about two fundamental dimensionality reduction techniques for feature extraction: PCA and LDA. Using PCA, we projected data onto a lower-dimensional subspace to maximize the variance along the orthogonal feature axes, while ignoring the class labels. LDA, in contrast to PCA, is a technique for supervised dimensionality reduction, which means that it considers class information in the training dataset to attempt to maximize the class separability in a linear feature space. Lastly, you also learned about t-SNE, which is a nonlinear feature extraction technique that can be used for visualizing data in two or three dimensions.

Equipped with PCA and LDA as fundamental data preprocessing techniques, you are now well prepared to learn about the best practices for efficiently incorporating different preprocessing techniques and evaluating the performance of different models in the next chapter.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`



6

Learning Best Practices for Model Evaluation and Hyperparameter Tuning

In the previous chapters, we learned about the essential machine learning algorithms for classification and how to get our data into shape before we feed it into those algorithms. Now, it's time to learn about the best practices of building good machine learning models by fine-tuning the algorithms and evaluating the performance of the models. In this chapter, we will learn how to do the following:

- Assess the performance of machine learning models
- Diagnose the common problems of machine learning algorithms
- Fine-tune machine learning models
- Evaluate predictive models using different performance metrics

Streamlining workflows with pipelines

When we applied different preprocessing techniques in the previous chapters, such as standardization for feature scaling in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*, or principal component analysis for data compression in *Chapter 5*, *Compressing Data via Dimensionality Reduction*, you learned that we have to reuse the parameters that were obtained during the fitting of the training data to scale and compress any new data, such as the examples in the separate test dataset. In this section, you will learn about an extremely handy tool, the `Pipeline` class in scikit-learn. It allows us to fit a model including an arbitrary number of transformation steps and apply it to make predictions about new data.

Loading the Breast Cancer Wisconsin dataset

In this chapter, we will be working with the Breast Cancer Wisconsin dataset, which contains 569 examples of malignant and benign tumor cells. The first two columns in the dataset store the unique ID numbers of the examples and the corresponding diagnoses (`M` = malignant, `B` = benign), respectively. Columns 3-32 contain 30 real-valued features that have been computed from digitized images of the cell nuclei, which can be used to build a model to predict whether a tumor is benign or malignant. The Breast Cancer Wisconsin dataset has been deposited in the UCI Machine Learning Repository, and more detailed information about this dataset can be found at `https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)`.

Obtaining the Breast Cancer Wisconsin dataset

You can find a copy of the dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the UCI server at `https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/wdbc.data` is temporarily unavailable. For instance, to load the dataset from a local directory, you can replace the following lines:

```
df = pd.read_csv(
    'https://archive.ics.uci.edu/ml/'
    'machine-learning-databases'
    '/breast-cancer-wisconsin/wdbc.data',
    header=None
)
```

with these:

```
df = pd.read_csv(
    'your/local/path/to/wdbc.data',
    header=None
)
```

In this section, we will read in the dataset and split it into training and test datasets in three simple steps:

1. We will start by reading in the dataset directly from the UCI website using pandas:

```
>>> import pandas as pd
>>> df = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                  'machine-learning-databases'
...                  '/breast-cancer-wisconsin/wdbc.data',
...                  header=None)
```

2. Next, we will assign the 30 features to a NumPy array, `X`. Using a `LabelEncoder` object, we will transform the class labels from their original string representation (`'M'` and `'B'`) into integers:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> X = df.loc[:, 2:].values
>>> y = df.loc[:, 1].values
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> le.classes_
array(['B', 'M'], dtype=object)
```

3. After encoding the class labels (diagnosis) in an array, `y`, the malignant tumors are now represented as class `1`, and the benign tumors are represented as class `0`, respectively. We can double-check this mapping by calling the `transform` method of the fitted `LabelEncoder` on two dummy class labels:

```
>>> le.transform(['M', 'B'])
array([1, 0])
```

4. Before we construct our first model pipeline in the following subsection, let's divide the dataset into a separate training dataset (80 percent of the data) and a separate test dataset (20 percent of the data):

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = \
...     train_test_split(X, y,
...                      test_size=0.20,
...                      stratify=y,
...                      random_state=1)
```

Combining transformers and estimators in a pipeline

In the previous chapter, you learned that many learning algorithms require input features on the same scale for optimal performance. Since the features in the Breast Cancer Wisconsin dataset are measured on various different scales, we will standardize the columns in the Breast Cancer Wisconsin dataset before we feed them to a linear classifier, such as logistic regression. Furthermore, let's assume that we want to compress our data from the initial 30 dimensions into a lower two-dimensional subspace via **principal component analysis** (**PCA**), a feature extraction technique for dimensionality reduction that was introduced in *Chapter 5*.

Instead of going through the model fitting and data transformation steps for the training and test datasets separately, we can chain the `StandardScaler`, `PCA`, and `LogisticRegression` objects in a pipeline:

```
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.decomposition import PCA
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.pipeline import make_pipeline
>>> pipe_lr = make_pipeline(StandardScaler(),
...                         PCA(n_components=2),
...                         LogisticRegression())
>>> pipe_lr.fit(X_train, y_train)
>>> y_pred = pipe_lr.predict(X_test)
>>> test_acc = pipe_lr.score(X_test, y_test)
>>> print(f'Test accuracy: {test_acc:.3f}')
Test accuracy: 0.956
```

The `make_pipeline` function takes an arbitrary number of scikit-learn transformers (objects that support the `fit` and `transform` methods as input), followed by a scikit-learn estimator that implements the `fit` and `predict` methods. In our preceding code example, we provided two scikit-learn transformers, `StandardScaler` and `PCA`, and a `LogisticRegression` estimator as inputs to the `make_pipeline` function, which constructs a scikit-learn `Pipeline` object from these objects.

We can think of a scikit-learn `Pipeline` as a meta-estimator or wrapper around those individual transformers and estimators. If we call the `fit` method of `Pipeline`, the data will be passed down a series of transformers via `fit` and `transform` calls on these intermediate steps until it arrives at the estimator object (the final element in a pipeline). The estimator will then be fitted to the transformed training data.

When we executed the `fit` method on the `pipe_lr` pipeline in the preceding code example, `StandardScaler` first performed `fit` and `transform` calls on the training data. Second, the transformed training data was passed on to the next object in the pipeline, `PCA`. Similar to the previous step, `PCA` also executed `fit` and `transform` on the scaled input data and passed it to the final element of the pipeline, the estimator.

Finally, the `LogisticRegression` estimator was fit to the training data after it underwent transformations via `StandardScaler` and `PCA`. Again, we should note that there is no limit to the number of intermediate steps in a pipeline; however, if we want to use the pipeline for prediction tasks, the last pipeline element has to be an estimator.

Similar to calling `fit` on a pipeline, pipelines also implement a `predict` method if the last step in the pipeline is an estimator. If we feed a dataset to the `predict` call of a `Pipeline` object instance, the data will pass through the intermediate steps via `transform` calls. In the final step, the estimator object will then return a prediction on the transformed data.

The pipelines of the scikit-learn library are immensely useful wrapper tools that we will use frequently throughout the rest of this book. To make sure that you've got a good grasp of how the `Pipeline` object works, please take a close look at *Figure 6.1*, which summarizes our discussion from the previous paragraphs:

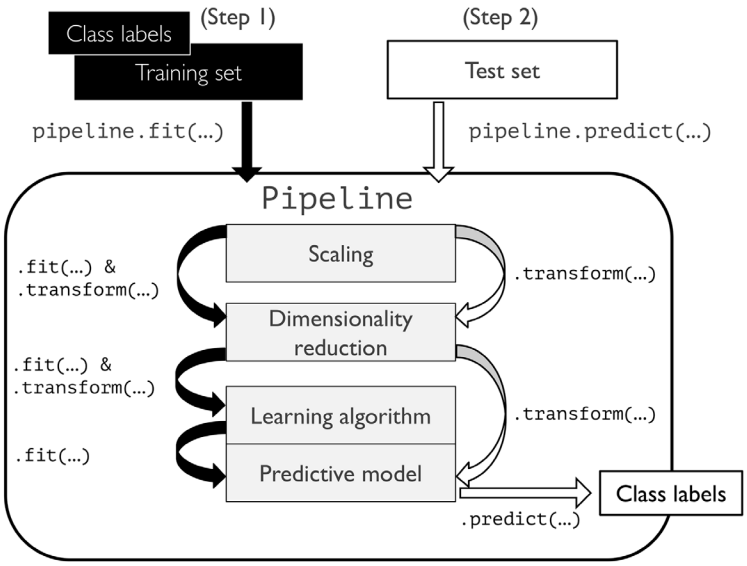


Figure 6.1: The inner workings of the Pipeline object

Using k-fold cross-validation to assess model performance

In this section, you will learn about the common cross-validation techniques **holdout cross-validation** and **k-fold cross-validation**, which can help us to obtain reliable estimates of the model's generalization performance, that is, how well the model performs on unseen data.

The holdout method

A classic and popular approach for estimating the generalization performance of machine learning models is the holdout method. Using the holdout method, we split our initial dataset into separate training and test datasets—the former is used for model training, and the latter is used to estimate its generalization performance. However, in typical machine learning applications, we are also interested in tuning and comparing different parameter settings to further improve the performance for making predictions on unseen data. This process is called **model selection**, with the name referring to a given classification problem for which we want to select the *optimal* values of *tuning parameters* (also called **hyperparameters**). However, if we reuse the same test dataset over and over again during model selection, it will become part of our training data and thus the model will be more likely to overfit. Despite this issue, many people still use the test dataset for model selection, which is not a good machine learning practice.

A better way of using the holdout method for model selection is to separate the data into three parts: a training dataset, a validation dataset, and a test dataset. The training dataset is used to fit the different models, and the performance on the validation dataset is then used for model selection. The advantage of having a test dataset that the model hasn't seen before during the training and model selection steps is that we can obtain a less biased estimate of its ability to generalize to new data. *Figure 6.2* illustrates the concept of holdout cross-validation, where we use a validation dataset to repeatedly evaluate the performance of the model after training using different hyperparameter values. Once we are satisfied with the tuning of hyperparameter values, we estimate the model's generalization performance on the test dataset:

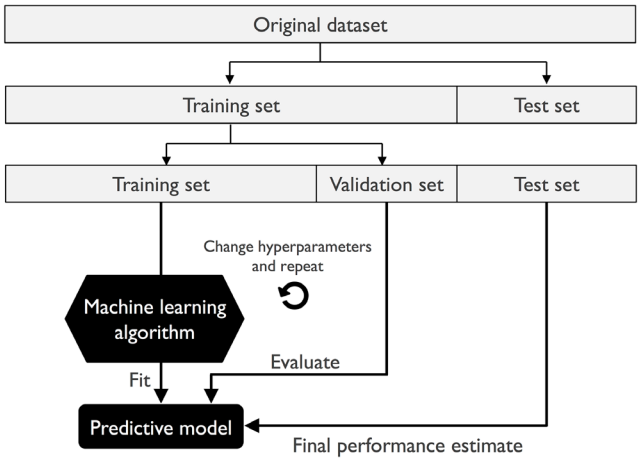


Figure 6.2: How to use training, validation, and test datasets

A disadvantage of the holdout method is that the performance estimate may be very sensitive to how we partition the training dataset into the training and validation subsets; the estimate will vary for different examples of the data. In the next subsection, we will take a look at a more robust technique for performance estimation, k-fold cross-validation, where we repeat the holdout method k times on k subsets of the training data.

K-fold cross-validation

In k-fold cross-validation, we randomly split the training dataset into k folds without replacement. Here, $k - 1$ folds, the so-called *training folds*, are used for the model training, and one fold, the so-called *test fold*, is used for performance evaluation. This procedure is repeated k times so that we obtain k models and performance estimates.

Sampling with and without replacement

We looked at an example to illustrate sampling with and without replacement in *Chapter 3*. If you haven't read that chapter, or want a refresher, refer to the information box titled *Sampling with and without replacement* in the *Combining multiple decision trees via random forests* section.

We then calculate the average performance of the models based on the different, independent test folds to obtain a performance estimate that is less sensitive to the sub-partitioning of the training data compared to the holdout method. Typically, we use k-fold cross-validation for model tuning, that is, finding the optimal hyperparameter values that yield a satisfying generalization performance, which is estimated from evaluating the model performance on the test folds.

Once we have found satisfactory hyperparameter values, we can retrain the model on the complete training dataset and obtain a final performance estimate using the independent test dataset. The rationale behind fitting a model to the whole training dataset after k-fold cross-validation is that first, we are typically interested in a single, final model (versus k individual models), and second, providing more training examples to a learning algorithm usually results in a more accurate and robust model.

Since k-fold cross-validation is a resampling technique without replacement, the advantage of this approach is that in each iteration, each example will be used exactly once, and the training and test folds are disjoint. Furthermore, all test folds are disjoint; that is, there is no overlap between the test folds. *Figure 6.3* summarizes the concept behind k-fold cross-validation with $k = 10$. The training dataset is divided into 10 folds, and during the 10 iterations, 9 folds are used for training, and 1 fold will be used as the test dataset for model evaluation.

Also, the estimated performances, E_i (for example, classification accuracy or error), for each fold are then used to calculate the estimated average performance, E, of the model:

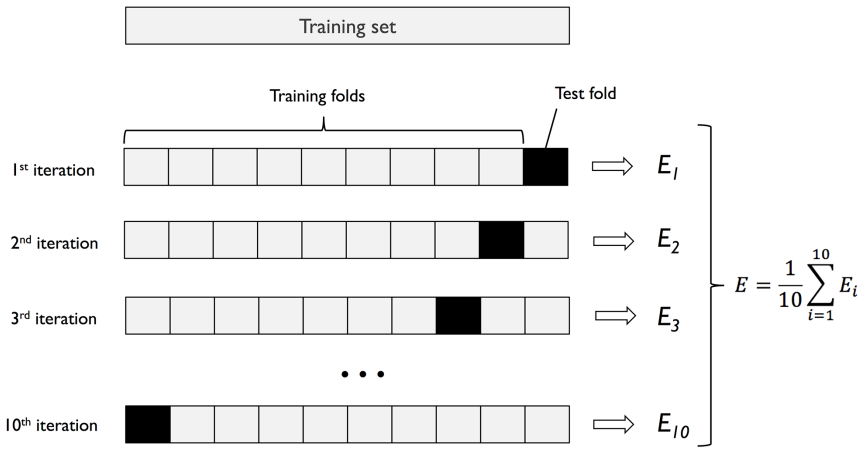


Figure 6.3: How k-fold cross-validation works

In summary, k-fold cross-validation makes better use of the dataset than the holdout method with a validation set, since in k-fold cross-validation all data points are being used for evaluation.

A good standard value for k in k-fold cross-validation is 10, as empirical evidence shows. For instance, experiments by Ron Kohavi on various real-world datasets suggest that 10-fold cross-validation offers the best tradeoff between bias and variance (*A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection* by *Kohavi, Ron*, *International Joint Conference on Artificial Intelligence (IJCAI)*, 14 (12): 1137-43, 1995, `https://www.ijcai.org/Proceedings/95-2/Papers/016.pdf`).

However, if we are working with relatively small training sets, it can be useful to increase the number of folds. If we increase the value of k, more training data will be used in each iteration, which results in a lower pessimistic bias toward estimating the generalization performance by averaging the individual model estimates. However, large values of k will also increase the runtime of the cross-validation algorithm and yield estimates with higher variance, since the training folds will be more similar to each other. On the other hand, if we are working with large datasets, we can choose a smaller value for k, for example, $k = 5$, and still obtain an accurate estimate of the average performance of the model while reducing the computational cost of refitting and evaluating the model on the different folds.

Leave-one-out cross-validation

A special case of k-fold cross-validation is the **leave-one-out cross-validation** (**LOOCV**) method. In LOOCV, we set the number of folds equal to the number of training examples ($k = n$) so that only one training example is used for testing during each iteration, which is a recommended approach for working with very small datasets.

A slight improvement over the standard k-fold cross-validation approach is stratified k-fold cross-validation, which can yield better bias and variance estimates, especially in cases of unequal class proportions, which has also been shown in the same study by Ron Kohavi referenced previously in this section. In stratified cross-validation, the class label proportions are preserved in each fold to ensure that each fold is representative of the class proportions in the training dataset, which we will illustrate by using the `StratifiedKFold` iterator in scikit-learn:

```
>>> import numpy as np
>>> from sklearn.model_selection import StratifiedKFold
>>> kfold = StratifiedKFold(n_splits=10).split(X_train, y_train)
>>> scores = []
>>> for k, (train, test) in enumerate(kfold):
...     pipe_lr.fit(X_train[train], y_train[train])
...     score = pipe_lr.score(X_train[test], y_train[test])
...     scores.append(score)
...     print(f'Fold: {k+1:02d}, '
...           f'Class distr.: {np.bincount(y_train[train])}, '
...           f'Acc.: {score:.3f}')
Fold: 01, Class distr.: [256 153], Acc.: 0.935
Fold: 02, Class distr.: [256 153], Acc.: 0.935
Fold: 03, Class distr.: [256 153], Acc.: 0.957
Fold: 04, Class distr.: [256 153], Acc.: 0.957
```

```
Fold: 05, Class distr.: [256 153], Acc.: 0.935
Fold: 06, Class distr.: [257 153], Acc.: 0.956
Fold: 07, Class distr.: [257 153], Acc.: 0.978
Fold: 08, Class distr.: [257 153], Acc.: 0.933
Fold: 09, Class distr.: [257 153], Acc.: 0.956
Fold: 10, Class distr.: [257 153], Acc.: 0.956
>>> mean_acc = np.mean(scores)
>>> std_acc = np.std(scores)
>>> print(f'\nCV accuracy: {mean_acc:.3f} +/- {std_acc:.3f}')
CV accuracy: 0.950 +/- 0.014
```

First, we initialized the `StratifiedKFold` iterator from the `sklearn.model_selection` module with the `y_train` class labels in the training dataset, and we specified the number of folds via the `n_splits` parameter. When we used the `kfold` iterator to loop through the `k` folds, we used the returned indices in `train` to fit the logistic regression pipeline that we set up at the beginning of this chapter. Using the `pipe_lr` pipeline, we ensured that the examples were scaled properly (for instance, standardized) in each iteration. We then used the `test` indices to calculate the accuracy score of the model, which we collected in the `scores` list to calculate the average accuracy and the standard deviation of the estimate.

Although the previous code example was useful to illustrate how k-fold cross-validation works, scikit-learn also implements a k-fold cross-validation scorer, which allows us to evaluate our model using stratified k-fold cross-validation less verbosely:

```
>>> from sklearn.model_selection import cross_val_score
>>> scores = cross_val_score(estimator=pipe_lr,
...                          X=X_train,
...                          y=y_train,
...                          cv=10,
...                          n_jobs=1)
>>> print(f'CV accuracy scores: {scores}')
CV accuracy scores: [ 0.93478261  0.93478261  0.95652174
                      0.95652174  0.93478261  0.95555556
                      0.97777778  0.93333333  0.95555556
                      0.95555556]
>>> print(f'CV accuracy: {np.mean(scores):.3f} '
...       f'+/- {np.std(scores):.3f}')
CV accuracy: 0.950 +/- 0.014
```

An extremely useful feature of the `cross_val_score` approach is that we can distribute the evaluation of the different folds across multiple **central processing units** (**CPUs**) on our machine. If we set the `n_jobs` parameter to `1`, only one CPU will be used to evaluate the performances, just like in our `StratifiedKFold` example previously. However, by setting `n_jobs=2`, we could distribute the 10 rounds of cross-validation to two CPUs (if available on our machine), and by setting `n_jobs=-1`, we can use all available CPUs on our machine to do the computation in parallel.

Estimating generalization performance

Please note that a detailed discussion of how the variance of the generalization performance is estimated in cross-validation is beyond the scope of this book, but you can refer to a comprehensive article about model evaluation and cross-validation (*Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning* by *S. Raschka*), which we share at `https://arxiv.org/abs/1811.12808`. This article also discusses alternative cross-validation techniques, such as the .632 and .632+ bootstrap cross-validation methods.

In addition, you can find a detailed discussion in an excellent article by M. Markatou and others (*Analysis of Variance of Cross-validation Estimators of the Generalization Error* by *M. Markatou, H. Tian, S. Biswas,* and *G. M. Hripcsak, Journal of Machine Learning Research*, 6: 1127-1168, 2005), which is available at `https://www.jmlr.org/papers/v6/markatou05a.html`.

Debugging algorithms with learning and validation curves

In this section, we will take a look at two very simple yet powerful diagnostic tools that can help us to improve the performance of a learning algorithm: **learning curves** and **validation curves**. In the next subsections, we will discuss how we can use learning curves to diagnose whether a learning algorithm has a problem with overfitting (high variance) or underfitting (high bias). Furthermore, we will take a look at validation curves, which can help us to address the common issues of learning algorithms.

Diagnosing bias and variance problems with learning curves

If a model is too complex for a given training dataset—for example, think of a very deep decision tree—the model tends to overfit the training data and does not generalize well to unseen data. Often, it can help to collect more training examples to reduce the degree of overfitting.

However, in practice, it can often be very expensive or simply not feasible to collect more data. By plotting the model training and validation accuracies as functions of the training dataset size, we can easily detect whether the model suffers from high variance or high bias, and whether the collection of more data could help to address this problem.

But before we discuss how to plot learning curves in scikit-learn, let's discuss those two common model issues by walking through the following illustration:

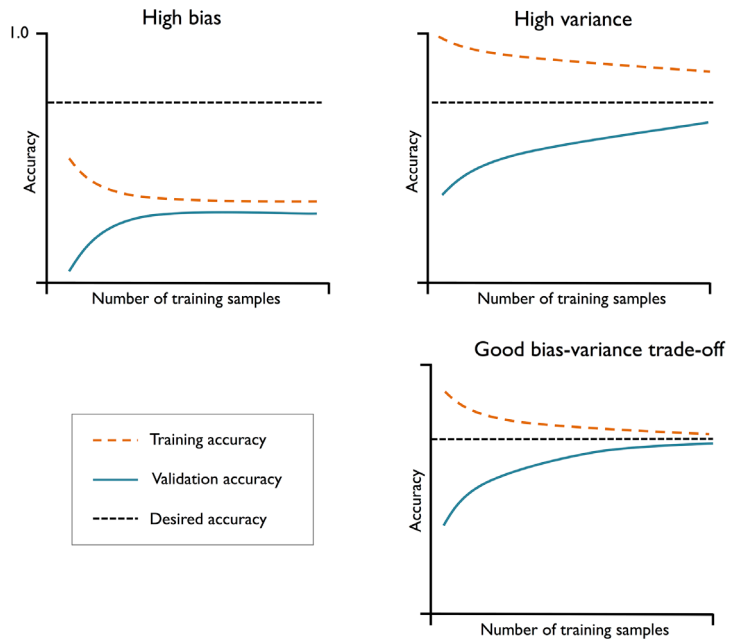


Figure 6.4: Common model issues

The graph in the upper left shows a model with a high bias. This model has both low training and cross-validation accuracy, which indicates that it underfits the training data. Common ways to address this issue are to increase the number of model parameters, for example, by collecting or constructing additional features, or by decreasing the degree of regularization, for example, in **support vector machine** (**SVM**) or logistic regression classifiers.

The graph in the upper-right shows a model that suffers from high variance, which is indicated by the large gap between the training and cross-validation accuracy. To address this problem of overfitting, we can collect more training data, reduce the complexity of the model, or increase the regularization parameter, for example.

For unregularized models, it can also help to decrease the number of features via feature selection (*Chapter 4*) or feature extraction (*Chapter 5*) to decrease the degree of overfitting. While collecting more training data usually tends to decrease the chance of overfitting, it may not always help, for example, if the training data is extremely noisy or the model is already very close to optimal.

In the next subsection, we will see how to address those model issues using validation curves, but let's first see how we can use the learning curve function from scikit-learn to evaluate the model:

```
>>> import matplotlib.pyplot as plt
>>> from sklearn.model_selection import learning_curve
>>> pipe_lr = make_pipeline(StandardScaler(),
...                         LogisticRegression(penalty='l2',
...                                            max_iter=10000))
>>> train_sizes, train_scores, test_scores =\
...                 learning_curve(estimator=pipe_lr,
...                                X=X_train,
...                                y=y_train,
...                                train_sizes=np.linspace(
...                                            0.1, 1.0, 10),
...                                cv=10,
...                                n_jobs=1)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(train_sizes, train_mean,
...          color='blue', marker='o',
...          markersize=5, label='Training accuracy')
>>> plt.fill_between(train_sizes,
...                  train_mean + train_std,
...                  train_mean - train_std,
...                  alpha=0.15, color='blue')
>>> plt.plot(train_sizes, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='Validation accuracy')
>>> plt.fill_between(train_sizes,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xlabel('Number of training examples')
>>> plt.ylabel('Accuracy')
>>> plt.legend(loc='lower right')
>>> plt.ylim([0.8, 1.03])
>>> plt.show()
```

Note that we passed `max_iter=10000` as an additional argument when instantiating the `LogisticRegression` object (which uses 1,000 iterations as a default) to avoid convergence issues for the smaller dataset sizes or extreme regularization parameter values (covered in the next section). After we have successfully executed the preceding code, we will obtain the following learning curve plot:

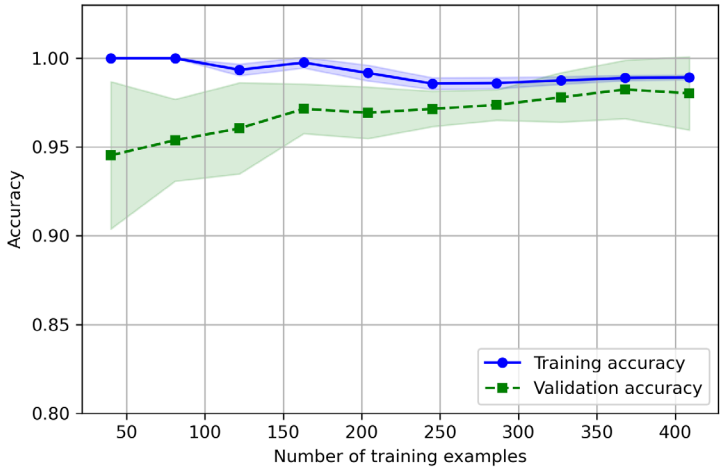


Figure 6.5: A learning curve showing training and validation dataset accuracy by the number of training examples

Via the `train_sizes` parameter in the `learning_curve` function, we can control the absolute or relative number of training examples that are used to generate the learning curves. Here, we set `train_sizes=np.linspace(0.1, 1.0, 10)` to use 10 evenly spaced, relative intervals for the training dataset sizes. By default, the `learning_curve` function uses stratified k-fold cross-validation to calculate the cross-validation accuracy of a classifier, and we set $k = 10$ via the `cv` parameter for 10-fold stratified cross-validation.

Then, we simply calculated the average accuracies from the returned cross-validated training and test scores for the different sizes of the training dataset, which we plotted using Matplotlib's `plot` function. Furthermore, we added the standard deviation of the average accuracy to the plot using the `fill_between` function to indicate the variance of the estimate.

As we can see in the preceding learning curve plot, our model performs quite well on both the training and validation datasets if it has seen more than 250 examples during training. We can also see that the training accuracy increases for training datasets with fewer than 250 examples, and the gap between validation and training accuracy widens—an indicator of an increasing degree of overfitting.

Addressing over- and underfitting with validation curves

Validation curves are a useful tool for improving the performance of a model by addressing issues such as overfitting or underfitting. Validation curves are related to learning curves, but instead of plotting the training and test accuracies as functions of the sample size, we vary the values of the model parameters, for example, the inverse regularization parameter, `C`, in logistic regression.

Let's go ahead and see how we create validation curves via scikit-learn:

```
>>> from sklearn.model_selection import validation_curve
>>> param_range = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0]
>>> train_scores, test_scores = validation_curve(
...                             estimator=pipe_lr,
...                             X=X_train,
...                             y=y_train,
...                             param_name='logisticregression__C',
...                             param_range=param_range,
...                             cv=10)
>>> train_mean = np.mean(train_scores, axis=1)
>>> train_std = np.std(train_scores, axis=1)
>>> test_mean = np.mean(test_scores, axis=1)
>>> test_std = np.std(test_scores, axis=1)
>>> plt.plot(param_range, train_mean,
...          color='blue', marker='o',
...          markersize=5, label='Training accuracy')
>>> plt.fill_between(param_range, train_mean + train_std,
...                  train_mean - train_std, alpha=0.15,
...                  color='blue')
>>> plt.plot(param_range, test_mean,
...          color='green', linestyle='--',
...          marker='s', markersize=5,
...          label='Validation accuracy')
>>> plt.fill_between(param_range,
...                  test_mean + test_std,
...                  test_mean - test_std,
...                  alpha=0.15, color='green')
>>> plt.grid()
>>> plt.xscale('log')
>>> plt.legend(loc='lower right')
>>> plt.xlabel('Parameter C')
>>> plt.ylabel('Accuracy')
>>> plt.ylim([0.8, 1.0])
>>> plt.show()
```

Using the preceding code, we obtained the validation curve plot for the parameter `C`:

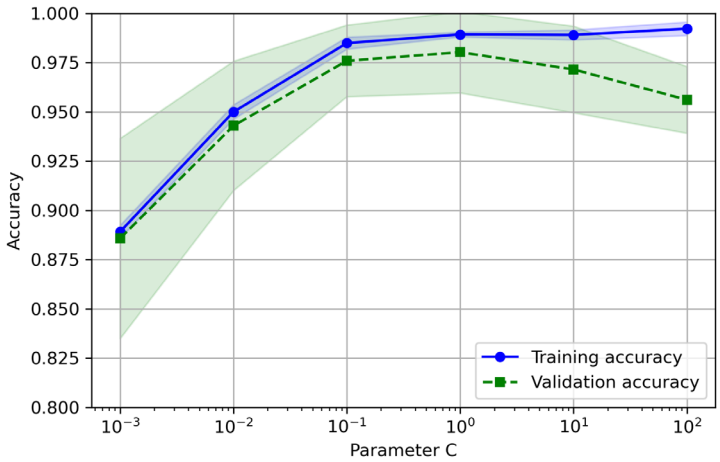


Figure 6.6: A validation curve plot for the SVM hyperparameter C

Similar to the `learning_curve` function, the `validation_curve` function uses stratified k-fold cross-validation by default to estimate the performance of the classifier. Inside the `validation_curve` function, we specified the parameter that we wanted to evaluate. In this case, it is `C`, the inverse regularization parameter of the `LogisticRegression` classifier, which we wrote as `'logisticregression__C'` to access the `LogisticRegression` object inside the scikit-learn pipeline for a specified value range that we set via the `param_range` parameter. Similar to the learning curve example in the previous section, we plotted the average training and cross-validation accuracies and the corresponding standard deviations.

Although the differences in the accuracy for varying values of `C` are subtle, we can see that the model slightly underfits the data when we increase the regularization strength (small values of `C`). However, for large values of `C`, it means lowering the strength of regularization, so the model tends to slightly overfit the data. In this case, the sweet spot appears to be between 0.1 and 1.0 of the `C` value.

Fine-tuning machine learning models via grid search

In machine learning, we have two types of parameters: those that are learned from the training data, for example, the weights in logistic regression, and the parameters of a learning algorithm that are optimized separately. The latter are the tuning parameters (or hyperparameters) of a model, for example, the regularization parameter in logistic regression or the maximum depth parameter of a decision tree.

In the previous section, we used validation curves to improve the performance of a model by tuning one of its hyperparameters. In this section, we will take a look at a popular hyperparameter optimization technique called **grid search**, which can further help to improve the performance of a model by finding the *optimal* combination of hyperparameter values.

Tuning hyperparameters via grid search

The grid search approach is quite simple: it's a brute-force exhaustive search paradigm where we specify a list of values for different hyperparameters, and the computer evaluates the model performance for each combination to obtain the optimal combination of values from this set:

```
>>> from sklearn.model_selection import GridSearchCV
>>> from sklearn.svm import SVC
>>> pipe_svc = make_pipeline(StandardScaler(),
...                          SVC(random_state=1))
>>> param_range = [0.0001, 0.001, 0.01, 0.1,
...                1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{'svc__C': param_range,
...                'svc__kernel': ['linear']},
...               {'svc__C': param_range,
...                'svc__gamma': param_range,
...                'svc__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=10,
...                   refit=True,
...                   n_jobs=-1)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.9846153846153847
>>> print(gs.best_params_)
{'svc__C': 100.0, 'svc__gamma': 0.001, 'svc__kernel': 'rbf'}
```

Using the preceding code, we initialized a `GridSearchCV` object from the `sklearn.model_selection` module to train and tune an SVM pipeline. We set the `param_grid` parameter of `GridSearchCV` to a list of dictionaries to specify the parameters that we'd want to tune. For the linear SVM, we only evaluated the inverse regularization parameter, `C`; for the **radial basis function** (**RBF**) kernel SVM, we tuned both the `svc__C` and `svc__gamma` parameters. Note that the `svc__gamma` parameter is specific to kernel SVMs.

`GridSearchCV` uses k-fold cross-validation for comparing models trained with different hyperparameter settings. Via the `cv=10` setting, it will carry out 10-fold cross-validation and compute the average accuracy (via `scoring='accuracy'`) across these 10-folds to assess the model performance. We set `n_jobs=-1` so that `GridSearchCV` can use all our processing cores to speed up the grid search by fitting models to the different folds in parallel, but if your machine has problems with this setting, you may change this setting to `n_jobs=None` for single processing.

After we used the training data to perform the grid search, we obtained the score of the best-performing model via the `best_score_` attribute and looked at its parameters, which can be accessed via the `best_params_` attribute. In this particular case, the RBF kernel SVM model with `svc__C = 100.0` yielded the best k-fold cross-validation accuracy: 98.5 percent.

Finally, we use the independent test dataset to estimate the performance of the best-selected model, which is available via the `best_estimator_` attribute of the `GridSearchCV` object:

```
>>> clf = gs.best_estimator_
>>> clf.fit(X_train, y_train)
>>> print(f'Test accuracy: {clf.score(X_test, y_test):.3f}')
Test accuracy: 0.974
```

Please note that fitting a model with the best settings (`gs.best_estimator_`) on the training set manually via `clf.fit(X_train, y_train)` after completing the grid search is not necessary. The `GridSearchCV` class has a `refit` parameter, which will refit the `gs.best_estimator_` to the whole training set automatically if we set `refit=True` (default).

Exploring hyperparameter configurations more widely with randomized search

Since grid search is an exhaustive search, it is guaranteed to find the optimal hyperparameter configuration if it is contained in the user-specified parameter grid. However, specifying large hyperparameter grids makes grid search very expensive in practice. An alternative approach for sampling different parameter combinations is randomized search. In randomized search, we draw hyperparameter configurations randomly from distributions (or discrete sets). In contrast to grid search, randomized search does not do an exhaustive search over the hyperparameter space. Still, it allows us to explore a wider range of hyperparameter value settings in a more cost- and time-effective manner. This concept is illustrated in *Figure 6.7*, which shows a fixed grid of nine hyperparameter settings being searched via grid search and randomized search:

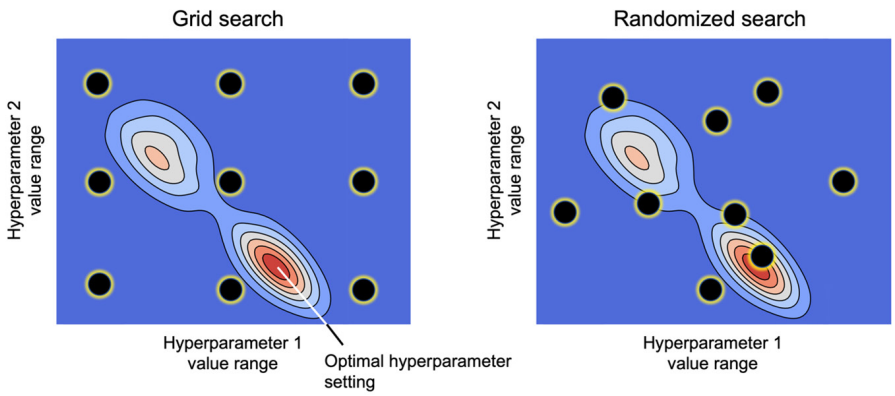


Figure 6.7: A comparison of grid search and randomized search for sampling nine different hyperparameter configurations each

The main takeaway is that while grid search only explores discrete, user-specified choices, it may miss good hyperparameter configurations if the search space is too scarce. Interested readers can find additional details about randomized search, along with empirical studies, in the following article: *Random Search for Hyper-Parameter Optimization* by *J. Bergstra*, *Y. Bengio*, *Journal of Machine Learning Research*, pp. 281-305, 2012, `https://www.jmlr.org/papers/volume13/bergstra12a/bergstra12a`.

Let's look at how we can use randomized search for tuning an SVM. Scikit-learn implements a `RandomizedSearchCV` class, which is analogous to the `GridSearchCV` we used in the previous subsection. The main difference is that we can specify distributions as part of our parameter grid and specify the total number of hyperparameter configurations to be evaluated. For example, let's consider the hyperparameter range we used for several hyperparameters when tuning the SVM in the grid search example in the previous section:

```
>>> import scipy.stats
>>> param_range = [0.0001, 0.001, 0.01, 0.1,
...                1.0, 10.0, 100.0, 1000.0]
```

Note that while `RandomizedSearchCV` can accept similar discrete lists of values as inputs for the parameter grid, which is useful when considering categorical hyperparameters, its main power lies in the fact that we can replace these lists with distributions to sample from. Thus, for example, we may substitute the preceding list with the following distribution from SciPy:

```
>>> param_range = scipy.stats.loguniform(0.0001, 1000.0)
```

For instance, using a loguniform distribution instead of a regular uniform distribution will ensure that in a sufficiently large number of trials, the same number of samples will be drawn from the [0.0001, 0.001] range as, for example, the [10.0, 100.0] range. To check its behavior, we can draw 10 random samples from this distribution via the `rvs(10)` method, as shown here:

```
>>> np.random.seed(1)
>>> param_range.rvs(10)
array([8.30145146e-02, 1.10222804e+01, 1.00184520e-04, 1.30715777e-02,
       1.06485687e-03, 4.42965766e-04, 2.01289666e-03, 2.62376594e-02,
       5.98924832e-02, 5.91176467e-01])
```

Specifying distributions

`RandomizedSearchCV` supports arbitrary distributions as long as we can sample from them by calling the `rvs()` method. A list of all distributions currently available via `scipy.stats` can be found here: `https://docs.scipy.org/doc/scipy/reference/stats.html#probability-distributions`.

Let's now see the `RandomizedSearchCV` in action and tune an SVM as we did with `GridSearchCV` in the previous section:

```
>>> from sklearn.model_selection import RandomizedSearchCV
>>> pipe_svc = make_pipeline(StandardScaler(),
...                          SVC(random_state=1))
```

```
>>> param_grid = [{'svc__C': param_range,
...                'svc__kernel': ['linear']},
...               {'svc__C': param_range,
...                'svc__gamma': param_range,
...                'svc__kernel': ['rbf']}]
>>> rs = RandomizedSearchCV(estimator=pipe_svc,
...                         param_distributions=param_grid,
...                         scoring='accuracy',
...                         refit=True,
...                         n_iter=20,
...                         cv=10,
...                         random_state=1,
...                         n_jobs=-1)

>>> rs = rs.fit(X_train, y_train)
>>> print(rs.best_score_)
0.9670531400966184

>>> print(rs.best_params_)
{'svc__C': 0.05971247755848464, 'svc__kernel': 'linear'}
```

Based on this code example, we can see that the usage is very similar to `GridSearchCV`, except that we could use distributions for specifying parameter ranges and specified the number of iterations—20 iterations—by setting `n_iter=20`.

More resource-efficient hyperparameter search with successive halving

Taking the idea of randomized search one step further, scikit-learn implements a successive halving variant, `HalvingRandomSearchCV`, that makes finding suitable hyperparameter configurations more efficient. Successive halving, given a large set of candidate configurations, successively throws out unpromising hyperparameter configurations until only one configuration remains. We can summarize the procedure via the following steps:

1. Draw a large set of candidate configurations via random sampling
2. Train the models with limited resources, for example, a small subset of the training data (as opposed to using the entire training set)
3. Discard the bottom 50 percent based on predictive performance
4. Go back to *step 2* with an increased amount of available resources

The steps are repeated until only one hyperparameter configuration remains. Note that there is also a successive halving implementation for the grid search variant called `HalvingGridSearchCV`, where all specified hyperparameter configurations are used in *step 1* instead of random samples.

In scikit-learn 1.0, `HalvingRandomSearchCV` is still experimental, which is why we have to enable it first:

```
>>> from sklearn.experimental import enable_halving_search_cv
```

(The above code may not work or be supported in future releases.)

After enabling the experimental support, we can use randomized search with successive halving as shown in the following:

```
>>> from sklearn.model_selection import HalvingRandomSearchCV

>>> hs = HalvingRandomSearchCV(pipe_svc,
...                            param_distributions=param_grid,
...                            n_candidates='exhaust',
...                            resource='n_samples',
...                            factor=1.5,
...                            random_state=1,
...                            n_jobs=-1)
```

The `resource='n_samples'` (default) setting specifies that we consider the training set size as the resource we vary between the rounds. Via the `factor` parameter, we can determine how many candidates are eliminated in each round. For example, setting `factor=2` eliminates half of the candidates, and setting `factor=1.5` means that only 100%/1.5 ≈ 66% of the candidates make it into the next round. Instead of choosing a fixed number of iterations as in `RandomizedSearchCV`, we set `n_candidates='exhaust'` (default), which will sample the number of hyperparameter configurations such that the maximum number of resources (here: training examples) are used in the last round.

We can then carry out the search similar to `RandomizedSearchCV`:

```
>>> hs = hs.fit(X_train, y_train)
>>> print(hs.best_score_)
0.9617647058823529

>>> print(hs.best_params_)
{'svc__C': 4.934834261073341, 'svc__kernel': 'linear'}
```

```
>>> clf = hs.best_estimator_
>>> print(f'Test accuracy: {hs.score(X_test, y_test):.3f}')
Test accuracy: 0.982
```

If we compare the results from `GridSearchCV` and `RandomizedSearchCV` from the previous two subsections with the model from `HalvingRandomSearchCV`, we can see that the latter yields a model that performs slightly better on the test set (98.2 percent accuracy as opposed to 97.4 percent).

Hyperparameter tuning with hyperopt

Another popular library for hyperparameter optimization is hyperopt (`https://github.com/hyperopt/hyperopt`), which implements several different methods for hyperparameter optimization, including randomized search and the **Tree-structured Parzen Estimators** (**TPE**) method. TPE is a Bayesian optimization method based on a probabilistic model that is continuously updated based on past hyperparameter evaluations and the associated performance scores instead of regarding these evaluations as independent events. You can find out more about TPE in *Algorithms for Hyper-Parameter Optimization*. *Bergstra J, Bardenet R, Bengio Y, Kegl B. NeurIPS 2011*. pp. 2546–2554, `https://dl.acm.org/doi/10.5555/2986459.2986743`.

While hyperopt provides a general-purpose interface for hyperparameter optimization, there is also a scikit-learn-specific package called hyperopt-sklearn for additional convenience: `https://github.com/hyperopt/hyperopt-sklearn`.

Algorithm selection with nested cross-validation

Using k-fold cross-validation in combination with grid search or randomized search is a useful approach for fine-tuning the performance of a machine learning model by varying its hyperparameter values, as we saw in the previous subsections. If we want to select among different machine learning algorithms, though, another recommended approach is **nested cross-validation**. In a nice study on the bias in error estimation, Sudhir Varma and Richard Simon concluded that the true error of the estimate is almost unbiased relative to the test dataset when nested cross-validation is used (*Bias in Error Estimation When Using Cross-Validation for Model Selection* by *S. Varma* and *R. Simon*, *BMC Bioinformatics*, 7(1): 91, 2006, `https://bmcbioinformatics.biomedcentral.com/articles/10.1186/1471-2105-7-91`).

In nested cross-validation, we have an outer k-fold cross-validation loop to split the data into training and test folds, and an inner loop is used to select the model using k-fold cross-validation on the training fold. After model selection, the test fold is then used to evaluate the model performance. *Figure 6.8* explains the concept of nested cross-validation with only five outer and two inner folds, which can be useful for large datasets where computational performance is important; this particular type of nested cross-validation is also known as **5×2 cross-validation**:

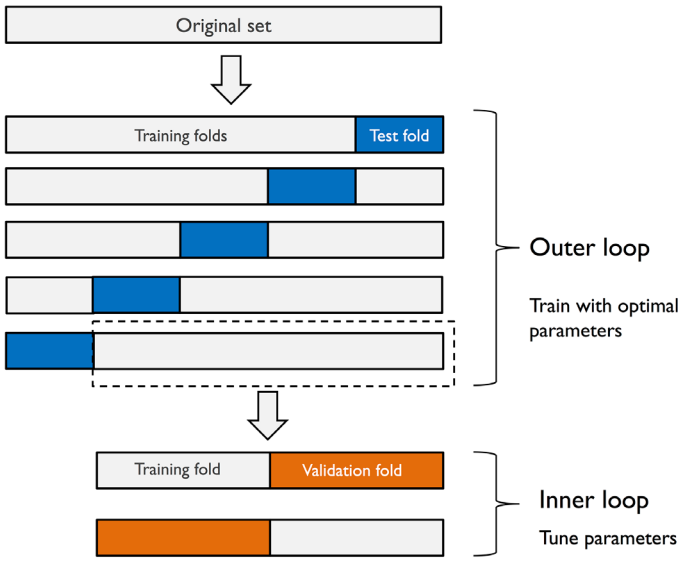


Figure 6.8: The concept of nested cross-validation

In scikit-learn, we can perform nested cross-validation with grid search as follows:

```
>>> param_range = [0.0001, 0.001, 0.01, 0.1,
...                1.0, 10.0, 100.0, 1000.0]
>>> param_grid = [{'svc__C': param_range,
...                'svc__kernel': ['linear']},
...               {'svc__C': param_range,
...                'svc__gamma': param_range,
...                'svc__kernel': ['rbf']}]
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring='accuracy',
...                   cv=2)
```

```
>>> scores = cross_val_score(gs, X_train, y_train,
...                          scoring='accuracy', cv=5)
>>> print(f'CV accuracy: {np.mean(scores):.3f} '
...       f'+/- {np.std(scores):.3f}')
CV accuracy: 0.974 +/- 0.015
```

The returned average cross-validation accuracy gives us a good estimate of what to expect if we tune the hyperparameters of a model and use it on unseen data.

For example, we can use the nested cross-validation approach to compare an SVM model to a simple decision tree classifier; for simplicity, we will only tune its depth parameter:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> gs = GridSearchCV(
...     estimator=DecisionTreeClassifier(random_state=0),
...     param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}],
...     scoring='accuracy',
...     cv=2
... )
>>> scores = cross_val_score(gs, X_train, y_train,
...                          scoring='accuracy', cv=5)
>>> print(f'CV accuracy: {np.mean(scores):.3f} '
...       f'+/- {np.std(scores):.3f}')
CV accuracy: 0.934 +/- 0.016
```

As we can see, the nested cross-validation performance of the SVM model (97.4 percent) is notably better than the performance of the decision tree (93.4 percent), and thus, we'd expect that it might be the better choice to classify new data that comes from the same population as this particular dataset.

Looking at different performance evaluation metrics

In the previous sections and chapters, we evaluated different machine learning models using prediction accuracy, which is a useful metric with which to quantify the performance of a model in general. However, there are several other performance metrics that can be used to measure a model's relevance, such as precision, recall, the **F1 score**, and **Matthews correlation coefficient** (**MCC**).

Reading a confusion matrix

Before we get into the details of different scoring metrics, let's take a look at a **confusion matrix**, a matrix that lays out the performance of a learning algorithm.

A confusion matrix is simply a square matrix that reports the counts of the **true positive** (**TP**), **true negative** (**TN**), **false positive** (**FP**), and **false negative** (**FN**) predictions of a classifier, as shown in *Figure 6.9*:

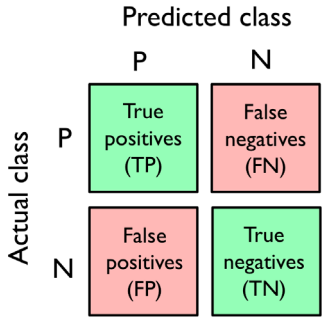


Figure 6.9: The confusion matrix

Although these metrics can be easily computed manually by comparing the actual and predicted class labels, scikit-learn provides a convenient `confusion_matrix` function that we can use, as follows:

```
>>> from sklearn.metrics import confusion_matrix
>>> pipe_svc.fit(X_train, y_train)
>>> y_pred = pipe_svc.predict(X_test)
>>> confmat = confusion_matrix(y_true=y_test, y_pred=y_pred)
>>> print(confmat)
[[71  1]
 [ 2 40]]
```

The array that was returned after executing the code provides us with information about the different types of error the classifier made on the test dataset. We can map this information onto the confusion matrix illustration in *Figure 6.9* using Matplotlib's `matshow` function:

```
>>> fig, ax = plt.subplots(figsize=(2.5, 2.5))
>>> ax.matshow(confmat, cmap=plt.cm.Blues, alpha=0.3)
>>> for i in range(confmat.shape[0]):
...     for j in range(confmat.shape[1]):
...         ax.text(x=j, y=i, s=confmat[i, j],
...                 va='center', ha='center')
>>> ax.xaxis.set_ticks_position('bottom')
>>> plt.xlabel('Predicted label')
>>> plt.ylabel('True label')
>>> plt.show()
```

Now, the following confusion matrix plot, with the added labels, should make the results a little bit easier to interpret:

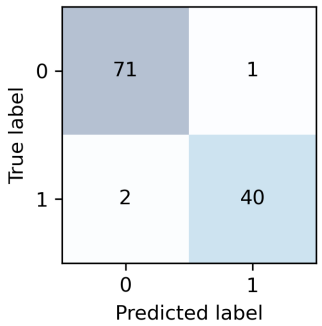


Figure 6.10: A confusion matrix for our data

Assuming that class `1` (malignant) is the positive class in this example, our model correctly classified 71 of the examples that belong to class `0` (TN) and 40 examples that belong to class `1` (TP), respectively. However, our model also incorrectly misclassified two examples from class `1` as class `0` (FN), and it predicted that one example is malignant although it is a benign tumor (FP). In the next subsection, we will learn how we can use this information to calculate various error metrics.

Optimizing the precision and recall of a classification model

Both the prediction **error** (**ERR**) and **accuracy** (**ACC**) provide general information about how many examples are misclassified. The error can be understood as the sum of all false predictions divided by the number of total predictions, and the accuracy is calculated as the sum of correct predictions divided by the total number of predictions, respectively:

$$ERR = \frac{FP + FN}{FP + FN + TP + TN}$$

The prediction accuracy can then be calculated directly from the error:

$$ACC = \frac{TP + TN}{FP + FN + TP + TN} = 1 - ERR$$

The **true positive rate** (**TPR**) and **false positive rate** (**FPR**) are performance metrics that are especially useful for imbalanced class problems:

$$FPR = \frac{FP}{N} = \frac{FP}{FP + TN}$$

$$TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

In tumor diagnosis, for example, we are more concerned about the detection of malignant tumors in order to help a patient with the appropriate treatment. However, it is also important to decrease the number of benign tumors incorrectly classified as malignant (FP) to not unnecessarily concern patients. In contrast to the FPR, the TPR provides useful information about the fraction of positive (or relevant) examples that were correctly identified out of the total pool of positives (P).

The performance metrics **precision** (**PRE**) and **recall** (**REC**) are related to those TP and TN rates, and in fact, REC is synonymous with TPR:

$$REC = TPR = \frac{TP}{P} = \frac{TP}{FN + TP}$$

In other words, recall quantifies how many of the relevant records (the positives) are captured as such (the true positives). Precision quantifies how many of the records predicted as relevant (the sum of true and false positives) are actually relevant (true positives):

$$PRE = \frac{TP}{TP + FP}$$

Revisiting the malignant tumor detection example, optimizing for recall helps with minimizing the chance of not detecting a malignant tumor. However, this comes at the cost of predicting malignant tumors in patients although the patients are healthy (a high number of FPs). If we optimize for precision, on the other hand, we emphasize correctness if we predict that a patient has a malignant tumor. However, this comes at the cost of missing malignant tumors more frequently (a high number of FNs).

To balance the up- and downsides of optimizing PRE and REC, the harmonic mean of PRE and REC is used, the so-called F1 score:

$$F1 = 2\frac{PRE \times REC}{PRE + REC}$$

Further reading on precision and recall

If you are interested in a more thorough discussion of the different performance metrics, such as precision and recall, read David M. W. Powers' technical report *Evaluation: From Precision, Recall and F-Factor to ROC, Informedness, Markedness & Correlation*, which is freely available at `https://arxiv.org/abs/2010.16061`.

Lastly, a measure that summarizes a confusion matrix is the MCC, which is especially popular in biological research contexts. The MCC is calculated as follows:

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

In contrast to PRE, REC, and the F1 score, the MCC ranges between –1 and 1, and it takes all elements of a confusion matrix into account—for instance, the F1 score does not involve the TN. While the MCC values are harder to interpret than the F1 score, it is regarded as a superior metric, as described in the following article: *The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation* by *D. Chicco* and *G. Jurman*, *BMC Genomics*. pp. 281-305, 2012, `https://bmcgenomics.biomedcentral.com/articles/10.1186/s12864-019-6413-7`.

Those scoring metrics are all implemented in scikit-learn and can be imported from the `sklearn.metrics` module as shown in the following snippet:

```
>>> from sklearn.metrics import precision_score
>>> from sklearn.metrics import recall_score, f1_score
>>> from sklearn.metrics import matthews_corrcoef

>>> pre_val = precision_score(y_true=y_test, y_pred=y_pred)
>>> print(f'Precision: {pre_val:.3f}')
Precision: 0.976
>>> rec_val = recall_score(y_true=y_test, y_pred=y_pred)
>>> print(f'Recall: {rec_val:.3f}')
Recall: 0.952
>>> f1_val = f1_score(y_true=y_test, y_pred=y_pred)
>>> print(f'F1: {f1_val:.3f}')
F1: 0.964
>>> mcc_val = matthews_corrcoef(y_true=y_test, y_pred=y_pred)
>>> print(f'MCC: {mcc_val:.3f}')
MCC: 0.943
```

Furthermore, we can use a different scoring metric than accuracy in the `GridSearchCV` via the scoring parameter. A complete list of the different values that are accepted by the scoring parameter can be found at `http://scikit-learn.org/stable/modules/model_evaluation.html`.

Remember that the positive class in scikit-learn is the class that is labeled as class `1`. If we want to specify a different *positive label*, we can construct our own scorer via the `make_scorer` function, which we can then directly provide as an argument to the `scoring` parameter in `GridSearchCV` (in this example, using the `f1_score` as a metric):

```
>>> from sklearn.metrics import make_scorer
>>> c_gamma_range = [0.01, 0.1, 1.0, 10.0]
>>> param_grid = [{'svc__C': c_gamma_range,
...                'svc__kernel': ['linear']},
...               {'svc__C': c_gamma_range,
...                'svc__gamma': c_gamma_range,
...                'svc__kernel': ['rbf']}]
```

```
>>> scorer = make_scorer(f1_score, pos_label=0)
>>> gs = GridSearchCV(estimator=pipe_svc,
...                   param_grid=param_grid,
...                   scoring=scorer,
...                   cv=10)
>>> gs = gs.fit(X_train, y_train)
>>> print(gs.best_score_)
0.986202145696
>>> print(gs.best_params_)
{'svc__C': 10.0, 'svc__gamma': 0.01, 'svc__kernel': 'rbf'}
```

Plotting a receiver operating characteristic

Receiver operating characteristic (**ROC**) graphs are useful tools to select models for classification based on their performance with respect to the FPR and TPR, which are computed by shifting the decision threshold of the classifier. The diagonal of a ROC graph can be interpreted as *random guessing*, and classification models that fall below the diagonal are considered as worse than random guessing. A perfect classifier would fall into the top-left corner of the graph with a TPR of 1 and an FPR of 0. Based on the ROC curve, we can then compute the so-called **ROC area under the curve** (**ROC AUC**) to characterize the performance of a classification model.

Similar to ROC curves, we can compute **precision-recall curves** for different probability thresholds of a classifier. A function for plotting those precision-recall curves is also implemented in scikit-learn and is documented at `http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_curve.html`.

Executing the following code example, we will plot a ROC curve of a classifier that only uses two features from the Breast Cancer Wisconsin dataset to predict whether a tumor is benign or malignant. Although we are going to use the same logistic regression pipeline that we defined previously, we are only using two features this time. This is to make the classification task more challenging for the classifier, by withholding useful information contained in the other features, so that the resulting ROC curve becomes visually more interesting. For similar reasons, we are also reducing the number of folds in the `StratifiedKFold` validator to three. The code is as follows:

```
>>> from sklearn.metrics import roc_curve, auc
>>> from numpy import interp
>>> pipe_lr = make_pipeline(
...     StandardScaler(),
...     PCA(n_components=2),
...     LogisticRegression(penalty='l2', random_state=1,
...                        solver='lbfgs', C=100.0)
... )
>>> X_train2 = X_train[:, [4, 14]]
```

```
>>> cv = list(StratifiedKFold(n_splits=3).split(X_train, y_train))
>>> fig = plt.figure(figsize=(7, 5))
>>> mean_tpr = 0.0
>>> mean_fpr = np.linspace(0, 1, 100)
>>> all_tpr = []
>>> for i, (train, test) in enumerate(cv):
...     probas = pipe_lr.fit(
...         X_train2[train],
...         y_train[train]
...     ).predict_proba(X_train2[test])
...     fpr, tpr, thresholds = roc_curve(y_train[test],
...                                      probas[:, 1],
...                                      pos_label=1)
...     mean_tpr += interp(mean_fpr, fpr, tpr)
...     mean_tpr[0] = 0.0
...     roc_auc = auc(fpr, tpr)
...     plt.plot(fpr,
...              tpr,
...              label=f'ROC fold {i+1} (area = {roc_auc:.2f})')
>>> plt.plot([0, 1],
...          [0, 1],
...          linestyle='--',
...          color=(0.6, 0.6, 0.6),
...          label='Random guessing (area=0.5)')
>>> mean_tpr /= len(cv)
>>> mean_tpr[-1] = 1.0
>>> mean_auc = auc(mean_fpr, mean_tpr)
>>> plt.plot(mean_fpr, mean_tpr, 'k--',
...          label=f'Mean ROC (area = {mean_auc:.2f})', lw=2)
>>> plt.plot([0, 0, 1],
...          [0, 1, 1],
...          linestyle=':',
...          color='black',
...          label='Perfect performance (area=1.0)')
>>> plt.xlim([-0.05, 1.05])
>>> plt.ylim([-0.05, 1.05])
>>> plt.xlabel('False positive rate')
>>> plt.ylabel('True positive rate')
>>> plt.legend(loc='lower right')
>>> plt.show()
```

In the preceding code example, we used the already familiar `StratifiedKFold` class from scikit-learn and calculated the ROC performance of the `LogisticRegression` classifier in our `pipe_lr` pipeline using the `roc_curve` function from the `sklearn.metrics` module separately for each iteration. Furthermore, we interpolated the average ROC curve from the three folds via the `interp` function that we imported from NumPy and calculated the area under the curve via the `auc` function. The resulting ROC curve indicates that there is a certain degree of variance between the different folds, and the average ROC AUC (0.76) falls between a perfect score (1.0) and random guessing (0.5):

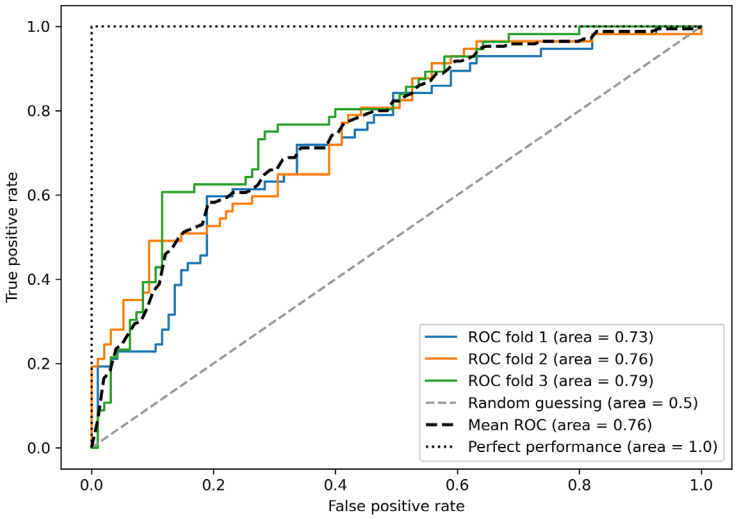


Figure 6.11: The ROC plot

Note that if we are just interested in the ROC AUC score, we could also directly import the `roc_auc_score` function from the `sklearn.metrics` submodule, which can be used similarly to the other scoring functions (for example, `precision_score`) that were introduced in the previous sections.

Reporting the performance of a classifier as the ROC AUC can yield further insights into a classifier's performance with respect to imbalanced samples. However, while the accuracy score can be interpreted as a single cutoff point on a ROC curve, A. P. Bradley showed that the ROC AUC and accuracy metrics mostly agree with each other: *The Use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms* by *A. P. Bradley*, *Pattern Recognition*, 30(7): 1145-1159, 1997, `https://reader.elsevier.com/reader/sd/pii/S0031320396001422`.

Scoring metrics for multiclass classification

The scoring metrics that we've discussed so far are specific to binary classification systems. However, scikit-learn also implements macro and micro averaging methods to extend those scoring metrics to multiclass problems via **one-vs.-all** (**OvA**) classification. The micro-average is calculated from the individual TPs, TNs, FPs, and FNs of the system. For example, the micro-average of the precision score in a k-class system can be calculated as follows:

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

The macro-average is simply calculated as the average scores of the different systems:

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Micro-averaging is useful if we want to weight each instance or prediction equally, whereas macro-averaging weights all classes equally to evaluate the overall performance of a classifier with regard to the most frequent class labels.

If we are using binary performance metrics to evaluate multiclass classification models in scikit-learn, a normalized or weighted variant of the macro-average is used by default. The weighted macro-average is calculated by weighting the score of each class label by the number of true instances when calculating the average. The weighted macro-average is useful if we are dealing with class imbalances, that is, different numbers of instances for each label.

While the weighted macro-average is the default for multiclass problems in scikit-learn, we can specify the averaging method via the `average` parameter inside the different scoring functions that we import from the `sklearn.metrics` module, for example, the `precision_score` or `make_scorer` functions:

```
>>> pre_scorer = make_scorer(score_func=precision_score,
...                          pos_label=1,
...                          greater_is_better=True,
...                          average='micro')
```

Dealing with class imbalance

We've mentioned class imbalances several times throughout this chapter, and yet we haven't actually discussed how to deal with such scenarios appropriately if they occur. Class imbalance is a quite common problem when working with real-world data—examples from one class or multiple classes are over-represented in a dataset. We can think of several domains where this may occur, such as spam filtering, fraud detection, or screening for diseases.

Imagine that the Breast Cancer Wisconsin dataset that we've been working with in this chapter consisted of 90 percent healthy patients. In this case, we could achieve 90 percent accuracy on the test dataset by just predicting the majority class (benign tumor) for all examples, without the help of a supervised machine learning algorithm. Thus, training a model on such a dataset that achieves approximately 90 percent test accuracy would mean our model hasn't learned anything useful from the features provided in this dataset.

In this section, we will briefly go over some of the techniques that could help with imbalanced datasets. But before we discuss different methods to approach this problem, let's create an imbalanced dataset from our dataset, which originally consisted of 357 benign tumors (class `0`) and 212 malignant tumors (class `1`):

```
>>> X_imb = np.vstack((X[y == 0], X[y == 1][:40]))
>>> y_imb = np.hstack((y[y == 0], y[y == 1][:40]))
```

In this code snippet, we took all 357 benign tumor examples and stacked them with the first 40 malignant examples to create a stark class imbalance. If we were to compute the accuracy of a model that always predicts the majority class (benign, class `0`), we would achieve a prediction accuracy of approximately 90 percent:

```
>>> y_pred = np.zeros(y_imb.shape[0])
>>> np.mean(y_pred == y_imb) * 100
89.92443324937027
```

Thus, when we fit classifiers on such datasets, it would make sense to focus on other metrics than accuracy when comparing different models, such as precision, recall, the ROC curve—whatever we care most about in our application. For instance, our priority might be to identify the majority of patients with malignant cancer to recommend an additional screening, so recall should be our metric of choice. In spam filtering, where we don't want to label emails as spam if the system is not very certain, precision might be a more appropriate metric.

Aside from evaluating machine learning models, class imbalance influences a learning algorithm during model fitting itself. Since machine learning algorithms typically optimize a reward or loss function that is computed as a sum over the training examples that it sees during fitting, the decision rule is likely going to be biased toward the majority class.

In other words, the algorithm implicitly learns a model that optimizes the predictions based on the most abundant class in the dataset to minimize the loss or maximize the reward during training.

One way to deal with imbalanced class proportions during model fitting is to assign a larger penalty to wrong predictions on the minority class. Via scikit-learn, adjusting such a penalty is as convenient as setting the `class_weight` parameter to `class_weight='balanced'`, which is implemented for most classifiers.

Other popular strategies for dealing with class imbalance include upsampling the minority class, downsampling the majority class, and the generation of synthetic training examples. Unfortunately, there's no universally best solution or technique that works best across different problem domains. Thus, in practice, it is recommended to try out different strategies on a given problem, evaluate the results, and choose the technique that seems most appropriate.

The scikit-learn library implements a simple `resample` function that can help with the upsampling of the minority class by drawing new samples from the dataset with replacement. The following code will take the minority class from our imbalanced Breast Cancer Wisconsin dataset (here, class `1`) and repeatedly draw new samples from it until it contains the same number of examples as class label `0`:

```
>>> from sklearn.utils import resample
>>> print('Number of class 1 examples before:',
...       X_imb[y_imb == 1].shape[0])
Number of class 1 examples before: 40
>>> X_upsampled, y_upsampled = resample(
...         X_imb[y_imb == 1],
```

```
...         y_imb[y_imb == 1],
...         replace=True,
...         n_samples=X_imb[y_imb == 0].shape[0],
...         random_state=123)
>>> print('Number of class 1 examples after:',
...       X_upsampled.shape[0])
Number of class 1 examples after: 357
```

After resampling, we can then stack the original class `0` samples with the upsampled class `1` subset to obtain a balanced dataset as follows:

```
>>> X_bal = np.vstack((X[y == 0], X_upsampled))
>>> y_bal = np.hstack((y[y == 0], y_upsampled))
```

Consequently, a majority vote prediction rule would only achieve 50 percent accuracy:

```
>>> y_pred = np.zeros(y_bal.shape[0])
>>> np.mean(y_pred == y_bal) * 100
50
```

Similarly, we could downsample the majority class by removing training examples from the dataset. To perform downsampling using the `resample` function, we could simply swap the class `1` label with class `0` in the previous code example and vice versa.

Generating new training data to address class imbalance

Another technique for dealing with class imbalance is the generation of synthetic training examples, which is beyond the scope of this book. Probably the most widely used algorithm for synthetic training data generation is **Synthetic Minority Over-sampling Technique** (**SMOTE**), and you can learn more about this technique in the original research article by *Nitesh Chawla* and others: *SMOTE: Synthetic Minority Over-sampling Technique*, *Journal of Artificial Intelligence Research*, 16: 321-357, 2002, which is available at `https://www.jair.org/index.php/jair/article/view/10302`. It is also highly recommended to check out imbalanced-learn, a Python library that is entirely focused on imbalanced datasets, including an implementation of SMOTE. You can learn more about imbalanced-learn at `https://github.com/scikit-learn-contrib/imbalanced-learn`.

Summary

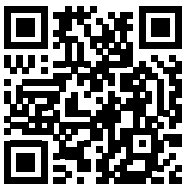
At the beginning of this chapter, we discussed how to chain different transformation techniques and classifiers in convenient model pipelines that help us to train and evaluate machine learning models more efficiently. We then used those pipelines to perform k-fold cross-validation, one of the essential techniques for model selection and evaluation. Using k-fold cross-validation, we plotted learning and validation curves to diagnose common problems of learning algorithms, such as overfitting and underfitting.

Using grid search, randomized search, and successive halving, we further fine-tuned our model. We then used confusion matrices and various performance metrics to evaluate and optimize a model's performance for specific problem tasks. Finally, we concluded this chapter by discussing different methods for dealing with imbalanced data, which is a common problem in many real-world applications. Now, you should be well equipped with the essential techniques to build supervised machine learning models for classification successfully.

In the next chapter, we will look at ensemble methods: methods that allow us to combine multiple models and classification algorithms to boost the predictive performance of a machine learning system even further.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`



7

Combining Different Models for Ensemble Learning

In the previous chapter, we focused on the best practices for tuning and evaluating different models for classification. In this chapter, we will build upon those techniques and explore different methods for constructing a set of classifiers that can often have a better predictive performance than any of its individual members. We will learn how to do the following:

- Make predictions based on majority voting
- Use bagging to reduce overfitting by drawing random combinations of the training dataset with repetition
- Apply boosting to build powerful models from weak learners that learn from their mistakes

Learning with ensembles

The goal of **ensemble methods** is to combine different classifiers into a meta-classifier that has better generalization performance than each individual classifier alone. For example, assuming that we collected predictions from 10 experts, ensemble methods would allow us to strategically combine those predictions by the 10 experts to come up with a prediction that was more accurate and robust than the predictions by each individual expert. As you will see later in this chapter, there are several different approaches for creating an ensemble of classifiers. This section will introduce a basic explanation of how ensembles work and why they are typically recognized for yielding a good generalization performance.

In this chapter, we will focus on the most popular ensemble methods that use the **majority voting** principle. Majority voting simply means that we select the class label that has been predicted by the majority of classifiers, that is, received more than 50 percent of the votes. Strictly speaking, the term "majority vote" refers to binary class settings only. However, it is easy to generalize the majority voting principle to multiclass settings, which is known as **plurality voting**. (In the UK, people distinguish between majority and plurality voting via the terms "absolute" and "relative" majority, respectively.)

Here, we select the class label that received the most votes (the mode). *Figure 7.1* illustrates the concept of majority and plurality voting for an ensemble of 10 classifiers, where each unique symbol (triangle, square, and circle) represents a unique class label:

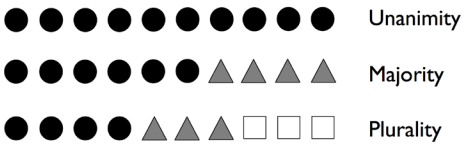


Figure 7.1: The different voting concepts

Using the training dataset, we start by training m different classifiers (C_1, ..., C_m). Depending on the technique, the ensemble can be built from different classification algorithms, for example, decision trees, support vector machines, logistic regression classifiers, and so on. Alternatively, we can also use the same base classification algorithm, fitting different subsets of the training dataset. One prominent example of this approach is the random forest algorithm combining different decision tree classifiers, which we covered in *Chapter 3*, *A Tour of Machine Learning Classifiers Using Scikit-Learn*. *Figure 7.2* illustrates the concept of a general ensemble approach using majority voting:

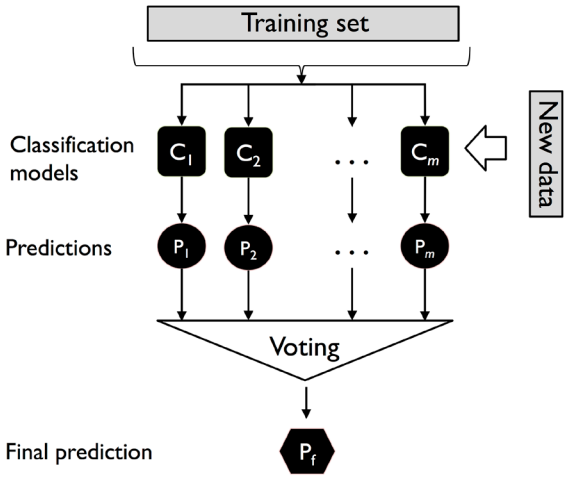


Figure 7.2: A general ensemble approach

To predict a class label via simple majority or plurality voting, we can combine the predicted class labels of each individual classifier, C_j, and select the class label, $\hat{y}$, that received the most votes:

$$\hat{y} = \text{mode}\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

(In statistics, the mode is the most frequent event or result in a set. For example, mode{1, 2, 1, 1, 2, 4, 5, 4} = 1.)

For example, in a binary classification task where class1 = –1 and class2 = +1, we can write the majority vote prediction as follows:

$$C(\boldsymbol{x}) = \text{sign}\left[\sum_{j}^{m} C_j(\boldsymbol{x})\right] = \begin{cases} 1 & \text{if } \sum_j C_j(\boldsymbol{x}) \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

To illustrate why ensemble methods can work better than individual classifiers alone, let's apply some concepts of combinatorics. For the following example, we will make the assumption that all n-base classifiers for a binary classification task have an equal error rate, ε. Furthermore, we will assume that the classifiers are independent and the error rates are not correlated. Under those assumptions, we can simply express the error probability of an ensemble of base classifiers as a probability mass function of a binomial distribution:

$$P(y \geq k) = \sum_{k}^{n} \left\langle {n \atop k} \right\rangle \varepsilon^k (1-\varepsilon)^{n-k} = \varepsilon_{\text{ensemble}}$$

Here, $\left\langle {n \atop k} \right\rangle$ is the binomial coefficient *n choose k*. In other words, we compute the probability that the prediction of the ensemble is wrong. Now, let's take a look at a more concrete example of 11 base classifiers ($n = 11$), where each classifier has an error rate of 0.25 ($\varepsilon = 0.25$):

$$P(y \geq k) = \sum_{k=6}^{11} \left\langle {11 \atop k} \right\rangle 0.25^k (1-0.25)^{11-k} = 0.034$$

The binomial coefficient

The binomial coefficient refers to the number of ways we can choose subsets of k unordered elements from a set of size n; thus, it is often called "n choose k." Since the order does not matter here, the binomial coefficient is also sometimes referred to as *combination* or *combinatorial number*, and in its unabbreviated form, it is written as follows:

$$\frac{n!}{(n-k)!\,k!}$$

Here, the symbol (!) stands for factorial—for example, 3! = 3×2×1 = 6.

As you can see, the error rate of the ensemble (0.034) is much lower than the error rate of each individual classifier (0.25) if all the assumptions are met. Note that, in this simplified illustration, a 50-50 split by an even number of classifiers, n, is treated as an error, whereas this is only true half of the time. To compare such an idealistic ensemble classifier to a base classifier over a range of different base error rates, let's implement the probability mass function in Python:

```
>>> from scipy.special import comb
>>> import math
>>> def ensemble_error(n_classifier, error):
```

```
...     k_start = int(math.ceil(n_classifier / 2.))
...     probs = [comb(n_classifier, k) *
...              error**k *
...              (1-error)**(n_classifier - k)
...              for k in range(k_start, n_classifier + 1)]
...     return sum(probs)
>>> ensemble_error(n_classifier=11, error=0.25)
0.03432750701904297
```

After we have implemented the `ensemble_error` function, we can compute the ensemble error rates for a range of different base errors from 0.0 to 1.0 to visualize the relationship between ensemble and base errors in a line graph:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> error_range = np.arange(0.0, 1.01, 0.01)
>>> ens_errors = [ensemble_error(n_classifier=11, error=error)
...               for error in error_range]
>>> plt.plot(error_range, ens_errors,
...          label='Ensemble error',
...          linewidth=2)
>>> plt.plot(error_range, error_range,
...          linestyle='--', label='Base error',
...          linewidth=2)
>>> plt.xlabel('Base error')
>>> plt.ylabel('Base/Ensemble error')
>>> plt.legend(loc='upper left')
>>> plt.grid(alpha=0.5)
>>> plt.show()
```

As you can see in the resulting plot, the error probability of an ensemble is always better than the error of an individual base classifier, as long as the base classifiers perform better than random guessing ($\varepsilon < 0.5$).

Note that the *y* axis depicts the base error (dotted line) as well as the ensemble error (continuous line):

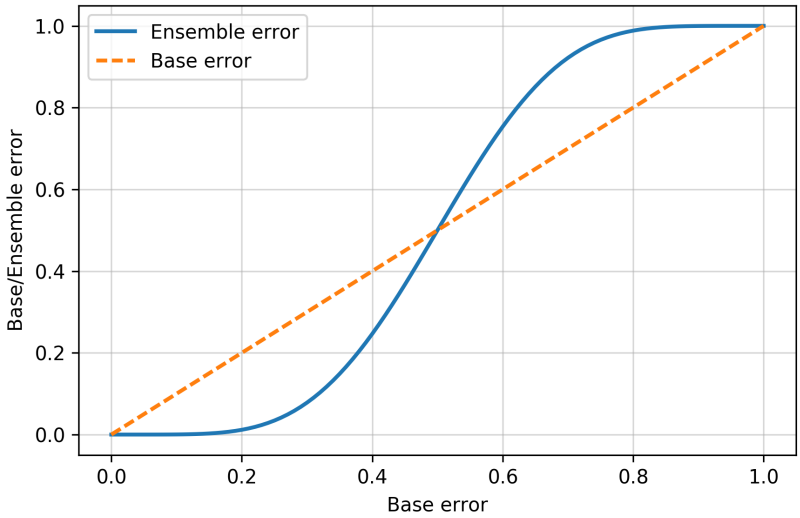


Figure 7.3: A plot of the ensemble error versus the base error

Combining classifiers via majority vote

After the short introduction to ensemble learning in the previous section, let's start with a warm-up exercise and implement a simple ensemble classifier for majority voting in Python.

Plurality voting

Although the majority voting algorithm that we will discuss in this section also generalizes to multiclass settings via plurality voting, the term "majority voting" will be used for simplicity, as is often the case in the literature.

Implementing a simple majority vote classifier

The algorithm that we are going to implement in this section will allow us to combine different classification algorithms associated with individual weights for confidence. Our goal is to build a stronger meta-classifier that balances out the individual classifiers' weaknesses on a particular dataset. In more precise mathematical terms, we can write the weighted majority vote as follows:

$$\hat{y} = \arg\max_{i} \sum_{j=1}^{m} w_j \chi_A\big(C_j(\boldsymbol{x}) = i\big)$$

Here, w_j is a weight associated with a base classifier, C_j; $\hat{y}$ is the predicted class label of the ensemble; A is the set of unique class labels; χ_A (Greek chi) is the characteristic function or indicator function, which returns 1 if the predicted class of the jth classifier matches i ($C_j(\boldsymbol{x}) = i$). For equal weights, we can simplify this equation and write it as follows:

$$\hat{y} = \text{mode}\{C_1(\boldsymbol{x}), C_2(\boldsymbol{x}), \ldots, C_m(\boldsymbol{x})\}$$

To better understand the concept of *weighting*, we will now take a look at a more concrete example. Let's assume that we have an ensemble of three base classifiers, $C_j (j \in \{1, 2, 3\})$, and we want to predict the class label, $C_j(\boldsymbol{x}) \in \{0, 1\}$, of a given example, $\boldsymbol{x}$. Two out of three base classifiers predict the class label 0, and one, C_3, predicts that the example belongs to class 1. If we weight the predictions of each base classifier equally, the majority vote predicts that the example belongs to class 0:

$$C_1(\boldsymbol{x}) \rightarrow 0, \quad C_2(\boldsymbol{x}) \rightarrow 0, \quad C_3(\boldsymbol{x}) \rightarrow 1$$

$$\hat{y} = \text{mode}\{0, 0, 1\} = 0$$

Now, let's assign a weight of 0.6 to C_3, and let's weight C_1 and C_2 by a coefficient of 0.2:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j \chi_A\big(C_j(\boldsymbol{x}) = i\big)$$

$$= \arg\max_i [0.2 \times i_0 + 0.2 \times i_0, 0.6 \times i_1] = 1$$

More simply, since 3×0.2 = 0.6, we can say that the prediction made by C_3 has three times more weight than the predictions by C_1 or C_2, which we can write as follows:

$$\hat{y} = \text{mode}\{0, 0, 1, 1, 1\} = 1$$

To translate the concept of the weighted majority vote into Python code, we can use NumPy's convenient `argmax` and `bincount` functions, where `bincount` counts the number of occurrences of each class label. The `argmax` function then returns the index position of the highest count, corresponding to the majority class label (this assumes that class labels start at 0):

```
>>> import numpy as np
>>> np.argmax(np.bincount([0, 0, 1],
...           weights=[0.2, 0.2, 0.6]))
1
```

As you will remember from the discussion on logistic regression in *Chapter 3*, certain classifiers in scikit-learn can also return the probability of a predicted class label via the `predict_proba` method. Using the predicted class probabilities instead of the class labels for majority voting can be useful if the classifiers in our ensemble are well calibrated. The modified version of the majority vote for predicting class labels from probabilities can be written as follows:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}$$

Here, p_{ij} is the predicted probability of the jth classifier for class label i.

To continue with our previous example, let's assume that we have a binary classification problem with class labels $i \in \{0, 1\}$ and an ensemble of three classifiers, $C_j (j \in \{1, 2, 3\})$. Let's assume that the classifiers C_j return the following class membership probabilities for a particular example, $\boldsymbol{x}$:

$$C_1(\boldsymbol{x}) \rightarrow [0.9, 0.1], \quad C_2(\boldsymbol{x}) \rightarrow [0.8, 0.2], \quad C_3(\boldsymbol{x}) \rightarrow [0.4, 0.6]$$

Using the same weights as previously (0.2, 0.2, and 0.6), we can then calculate the individual class probabilities as follows:

$$p(i_0|\boldsymbol{x}) = 0.2 \times 0.9 + 0.2 \times 0.8 + 0.6 \times 0.4 = 0.58$$

$$p(i_1|\boldsymbol{x}) = 0.2 \times 0.1 + 0.2 \times 0.2 + 0.6 \times 0.6 = 0.42$$

$$\hat{y} = \arg\max_i [p(i_0|\boldsymbol{x}), p(i_1|\boldsymbol{x})] = 0$$

To implement the weighted majority vote based on class probabilities, we can again make use of NumPy, using `np.average` and `np.argmax`:

```
>>> ex = np.array([[0.9, 0.1],
...                [0.8, 0.2],
...                [0.4, 0.6]])
>>> p = np.average(ex, axis=0, weights=[0.2, 0.2, 0.6])
>>> p
array([0.58, 0.42])
>>> np.argmax(p)
0
```

Putting everything together, let's now implement `MajorityVoteClassifier` in Python:

```
from sklearn.base import BaseEstimator
from sklearn.base import ClassifierMixin
from sklearn.preprocessing import LabelEncoder
from sklearn.base import clone
from sklearn.pipeline import _name_estimators
import numpy as np
import operator
class MajorityVoteClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, classifiers, vote='classlabel', weights=None):

        self.classifiers = classifiers
        self.named_classifiers = {
            key: value for key,
            value in _name_estimators(classifiers)
        }
        self.vote = vote
```

```
        self.weights = weights

    def fit(self, X, y):
        if self.vote not in ('probability', 'classlabel'):
            raise ValueError(f"vote must be 'probability' "
                             f"or 'classlabel'"
                             f"; got (vote={self.vote})")
        if self.weights and
        len(self.weights) != len(self.classifiers):
            raise ValueError(f'Number of classifiers and'
                             f' weights must be equal'
                             f'; got {len(self.weights)} weights,'
                             f' {len(self.classifiers)} classifiers')
        # Use LabelEncoder to ensure class labels start
        # with 0, which is important for np.argmax
        # call in self.predict
        self.lablenc_ = LabelEncoder()
        self.lablenc_.fit(y)
        self.classes_ = self.lablenc_.classes_
        self.classifiers_ = []
        for clf in self.classifiers:
            fitted_clf = clone(clf).fit(X,
                               self.lablenc_.transform(y))
            self.classifiers_.append(fitted_clf)
        return self
```

We've added a lot of comments to the code to explain the individual parts. However, before we implement the remaining methods, let's take a quick break and discuss some of the code that may look confusing at first. We used the `BaseEstimator` and `ClassifierMixin` parent classes to get some base functionality *for free*, including the `get_params` and `set_params` methods to set and return the classifier's parameters, as well as the `score` method to calculate the prediction accuracy.

Next, we will add the `predict` method to predict the class label via a majority vote based on the class labels if we initialize a new `MajorityVoteClassifier` object with `vote='classlabel'`. Alternatively, we will be able to initialize the ensemble classifier with `vote='probability'` to predict the class label based on the class membership probabilities. Furthermore, we will also add a `predict_proba` method to return the averaged probabilities, which is useful when computing the **receiver operating characteristic area under the curve** (**ROC AUC**):

```
    def predict(self, X):
        if self.vote == 'probability':
            maj_vote = np.argmax(self.predict_proba(X), axis=1)
        else: # 'classlabel' vote
```

```
            # Collect results from clf.predict calls
            predictions = np.asarray([
                clf.predict(X) for clf in self.classifiers_
            ]).T

            maj_vote = np.apply_along_axis(
                lambda x: np.argmax(
                    np.bincount(x, weights=self.weights)
                ),
                axis=1, arr=predictions
            )
        maj_vote = self.lablenc_.inverse_transform(maj_vote)
        return maj_vote

    def predict_proba(self, X):
        probas = np.asarray([clf.predict_proba(X)
                             for clf in self.classifiers_])
        avg_proba = np.average(probas, axis=0,
                               weights=self.weights)
        return avg_proba

    def get_params(self, deep=True):
        if not deep:
            return super().get_params(deep=False)
        else:
            out = self.named_classifiers.copy()
            for name, step in self.named_classifiers.items():
                for key, value in step.get_params(
                        deep=True).items():
                    out[f'{name}__{key}'] = value
            return out
```

Also, note that we defined our own modified version of the `get_params` method to use the `_name_estimators` function to access the parameters of individual classifiers in the ensemble; this may look a little bit complicated at first, but it will make perfect sense when we use grid search for hyperparameter tuning in later sections.

VotingClassifier in scikit-learn

Although the `MajorityVoteClassifier` implementation is very useful for demonstration purposes, we implemented a more sophisticated version of this majority vote classifier in scikit-learn based on the implementation in the first edition of this book. The ensemble classifier is available as `sklearn.ensemble.VotingClassifier` in scikit-learn version 0.17 and newer. You can find out more about `VotingClassifier` at `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.VotingClassifier.html`

Using the majority voting principle to make predictions

Now it is time to put the `MajorityVoteClassifier` that we implemented in the previous section into action. But first, let's prepare a dataset that we can test it on. Since we are already familiar with techniques to load datasets from CSV files, we will take a shortcut and load the Iris dataset from scikit-learn's `datasets` module. Furthermore, we will only select two features, *sepal width* and *petal length*, to make the classification task more challenging for illustration purposes. Although our `MajorityVoteClassifier` generalizes to multiclass problems, we will only classify flower examples from the `Iris-versicolor` and `Iris-virginica` classes, with which we will compute the ROC AUC later. The code is as follows:

```
>>> from sklearn import datasets
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import StandardScaler
>>> from sklearn.preprocessing import LabelEncoder
>>> iris = datasets.load_iris()
>>> X, y = iris.data[50:, [1, 2]], iris.target[50:]
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
```

Class membership probabilities from decision trees

Note that scikit-learn uses the `predict_proba` method (if applicable) to compute the ROC AUC score. In *Chapter 3*, we saw how the class probabilities are computed in logistic regression models. In decision trees, the probabilities are calculated from a frequency vector that is created for each node at training time. The vector collects the frequency values of each class label computed from the class label distribution at that node. Then, the frequencies are normalized so that they sum up to 1. Similarly, the class labels of the k-nearest neighbors are aggregated to return the normalized class label frequencies in the k-nearest neighbors algorithm. Although the normalized probabilities returned by both the decision tree and k-nearest neighbors classifier may look similar to the probabilities obtained from a logistic regression model, we have to be aware that they are actually not derived from probability mass functions.

Next, we will split the Iris examples into 50 percent training and 50 percent test data:

```
>>> X_train, X_test, y_train, y_test =\
...     train_test_split(X, y,
...                      test_size=0.5,
...                      random_state=1,
...                      stratify=y)
```

Using the training dataset, we now will train three different classifiers:

- Logistic regression classifier
- Decision tree classifier
- k-nearest neighbors classifier

We will then evaluate the model performance of each classifier via 10-fold cross-validation on the training dataset before we combine them into an ensemble classifier:

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.tree import DecisionTreeClassifier
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.pipeline import Pipeline
>>> import numpy as np
>>> clf1 = LogisticRegression(penalty='l2',
...                           C=0.001,
...                           solver='lbfgs',
...                           random_state=1)
>>> clf2 = DecisionTreeClassifier(max_depth=1,
...                               criterion='entropy',
...                               random_state=0)
>>> clf3 = KNeighborsClassifier(n_neighbors=1,
...                             p=2,
...                             metric='minkowski')
>>> pipe1 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf1]])
>>> pipe3 = Pipeline([['sc', StandardScaler()],
...                   ['clf', clf3]])
>>> clf_labels = ['Logistic regression', 'Decision tree', 'KNN']
>>> print('10-fold cross validation:\n')
>>> for clf, label in zip([pipe1, clf2, pipe3], clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
```

```
...                             cv=10,
...                             scoring='roc_auc')
...     print(f'ROC AUC: {scores.mean():.2f} '
...           f'(+/- {scores.std():.2f}) [{label}]')
```

The output that we receive, as shown in the following snippet, shows that the predictive performances of the individual classifiers are almost equal:

```
10-fold cross validation:
ROC AUC: 0.92 (+/- 0.15) [Logistic regression]
ROC AUC: 0.87 (+/- 0.18) [Decision tree]
ROC AUC: 0.85 (+/- 0.13) [KNN]
```

You may be wondering why we trained the logistic regression and k-nearest neighbors classifier as part of a pipeline. The reason behind it is that, as discussed in *Chapter 3*, both the logistic regression and k-nearest neighbors algorithms (using the Euclidean distance metric) are not scale-invariant, in contrast to decision trees. Although the Iris features are all measured on the same scale (cm), it is a good habit to work with standardized features.

Now, let's move on to the more exciting part and combine the individual classifiers for majority rule voting in our `MajorityVoteClassifier`:

```
>>> mv_clf = MajorityVoteClassifier(
...     classifiers=[pipe1, clf2, pipe3]
... )
>>> clf_labels += ['Majority voting']
>>> all_clf = [pipe1, clf2, pipe3, mv_clf]
>>> for clf, label in zip(all_clf, clf_labels):
...     scores = cross_val_score(estimator=clf,
...                              X=X_train,
...                              y=y_train,
...                              cv=10,
...                              scoring='roc_auc')
...     print(f'ROC AUC: {scores.mean():.2f} '
...           f'(+/- {scores.std():.2f}) [{label}]')
ROC AUC: 0.92 (+/- 0.15) [Logistic regression]
ROC AUC: 0.87 (+/- 0.18) [Decision tree]
ROC AUC: 0.85 (+/- 0.13) [KNN]
ROC AUC: 0.98 (+/- 0.05) [Majority voting]
```

As you can see, the performance of `MajorityVotingClassifier` has improved over the individual classifiers in the 10-fold cross-validation evaluation.

Evaluating and tuning the ensemble classifier

In this section, we are going to compute the ROC curves from the test dataset to check that `MajorityVoteClassifier` generalizes well with unseen data. We must remember that the test dataset is not to be used for model selection; its purpose is merely to report an unbiased estimate of the generalization performance of a classifier system:

```
>>> from sklearn.metrics import roc_curve
>>> from sklearn.metrics import auc
>>> colors = ['black', 'orange', 'blue', 'green']
>>> linestyles = [':', '--', '-.', '-']
>>> for clf, label, clr, ls \
...     in zip(all_clf, clf_labels, colors, linestyles):
...     # assuming the label of the positive class is 1
...     y_pred = clf.fit(X_train,
...                      y_train).predict_proba(X_test)[:, 1]
...     fpr, tpr, thresholds = roc_curve(y_true=y_test,
...                                      y_score=y_pred)
...     roc_auc = auc(x=fpr, y=tpr)
...     plt.plot(fpr, tpr,
...              color=clr,
...              linestyle=ls,
...              label=f'{label} (auc = {roc_auc:.2f})')
>>> plt.legend(loc='lower right')
>>> plt.plot([0, 1], [0, 1],
...          linestyle='--',
...          color='gray',
...          linewidth=2)
>>> plt.xlim([-0.1, 1.1])
>>> plt.ylim([-0.1, 1.1])
>>> plt.grid(alpha=0.5)
>>> plt.xlabel('False positive rate (FPR)')
>>> plt.ylabel('True positive rate (TPR)')
>>> plt.show()
```

As you can see in the resulting ROC, the ensemble classifier also performs well on the test dataset (ROC AUC = 0.95). However, you can see that the logistic regression classifier performs similarly well on the same dataset, which is probably due to the high variance (in this case, the sensitivity of how we split the dataset) given the small size of the dataset:

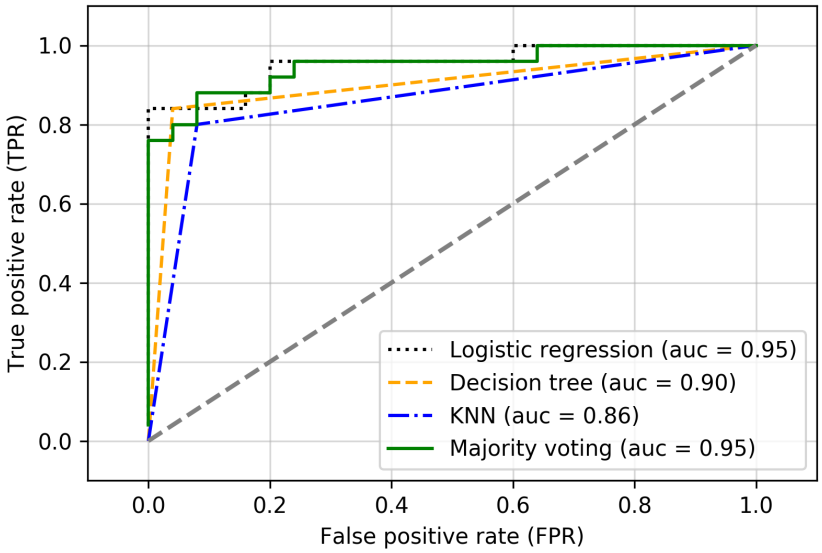


Figure 7.4: The ROC curve for the different classifiers

Since we only selected two features for the classification examples, it would be interesting to see what the decision region of the ensemble classifier actually looks like.

Although it is not necessary to standardize the training features prior to model fitting, because our logistic regression and k-nearest neighbors pipelines will automatically take care of it, we will standardize the training dataset so that the decision regions of the decision tree will be on the same scale for visual purposes. The code is as follows:

```
>>> sc = StandardScaler()
>>> X_train_std = sc.fit_transform(X_train)
>>> from itertools import product
>>> x_min = X_train_std[:, 0].min() - 1
>>> x_max = X_train_std[:, 0].max() + 1
>>> y_min = X_train_std[:, 1].min() - 1
>>>
>>> y_max = X_train_std[:, 1].max() + 1
```

```
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=2, ncols=2,
...                         sharex='col',
...                         sharey='row',
...                         figsize=(7, 5))
>>> for idx, clf, tt in zip(product([0, 1], [0, 1]),
...                         all_clf, clf_labels):
...     clf.fit(X_train_std, y_train)
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx[0], idx[1]].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==0, 0],
...                                   X_train_std[y_train==0, 1],
...                                   c='blue',
...                                   marker='^',
...                                   s=50)
...     axarr[idx[0], idx[1]].scatter(X_train_std[y_train==1, 0],
...                                   X_train_std[y_train==1, 1],
...                                   c='green',
...                                   marker='o',
...                                   s=50)
...     axarr[idx[0], idx[1]].set_title(tt)
>>> plt.text(-3.5, -5.,
...          s='Sepal width [standardized]',
...          ha='center', va='center', fontsize=12)
>>> plt.text(-12.5, 4.5,
...          s='Petal length [standardized]',
...          ha='center', va='center',
...          fontsize=12, rotation=90)
>>> plt.show()
```

Interestingly, but also as expected, the decision regions of the ensemble classifier seem to be a hybrid of the decision regions from the individual classifiers. At first glance, the majority vote decision boundary looks a lot like the decision of the decision tree stump, which is orthogonal to the *y* axis for *sepal width* ≥ 1.

However, you can also notice the nonlinearity from the k-nearest neighbor classifier mixed in:

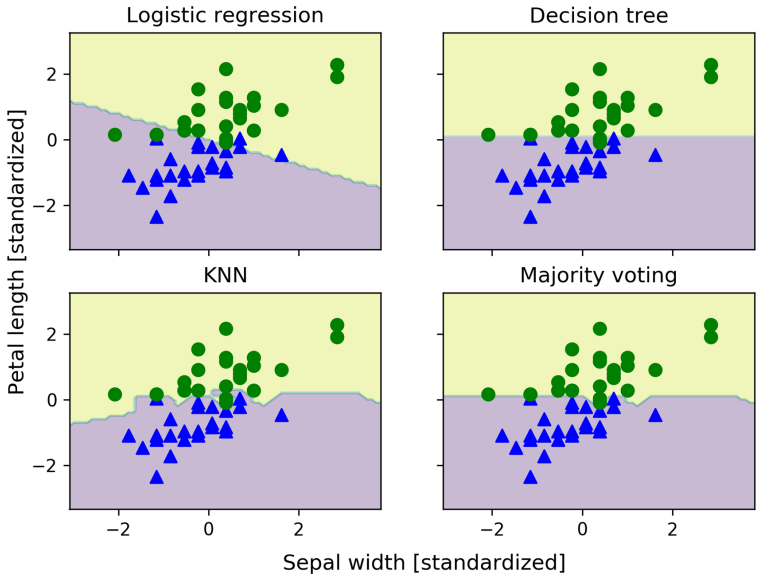


Figure 7.5: The decision boundaries for the different classifiers

Before we tune the individual classifier's parameters for ensemble classification, let's call the `get_params` method to get a basic idea of how we can access the individual parameters inside a `GridSearchCV` object:

```
>>> mv_clf.get_params()
{'decisiontreeclassifier':
 DecisionTreeClassifier(class_weight=None, criterion='entropy',
                        max_depth=1, max_features=None,
                        max_leaf_nodes=None, min_samples_leaf=1,
                        min_samples_split=2,
                        min_weight_fraction_leaf=0.0,
                        random_state=0, splitter='best'),
 'decisiontreeclassifier__class_weight': None,
 'decisiontreeclassifier__criterion': 'entropy',
 [...]
 'decisiontreeclassifier__random_state': 0,
 'decisiontreeclassifier__splitter': 'best',
 'pipeline-1':
 Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                       with_std=True)),
                 ('clf', LogisticRegression(C=0.001,
```

```
                                          class_weight=None,
                                          dual=False,
                                          fit_intercept=True,
                                          intercept_scaling=1,
                                          max_iter=100,
                                          multi_class='ovr',
                                          penalty='l2',
                                          random_state=0,
                                          solver='liblinear',
                                          tol=0.0001,
                                          verbose=0))]),
'pipeline-1__clf':
LogisticRegression(C=0.001, class_weight=None, dual=False,
                   fit_intercept=True, intercept_scaling=1,
                   max_iter=100, multi_class='ovr',
                   penalty='l2', random_state=0,
                   solver='liblinear', tol=0.0001, verbose=0),
'pipeline-1__clf__C': 0.001,
'pipeline-1__clf__class_weight': None,
'pipeline-1__clf__dual': False,
[...]
'pipeline-1__sc__with_std': True,
'pipeline-2':
Pipeline(steps=[('sc', StandardScaler(copy=True, with_mean=True,
                                      with_std=True)),
                ('clf', KNeighborsClassifier(algorithm='auto',
                                             leaf_size=30,
                                             metric='minkowski',
                                             metric_params=None,
                                             n_neighbors=1,
                                             p=2,
                                             weights='uniform'))]),
'pipeline-2__clf':
KNeighborsClassifier(algorithm='auto', leaf_size=30,
                     metric='minkowski', metric_params=None,
                     n_neighbors=1, p=2, weights='uniform'),
'pipeline-2__clf__algorithm': 'auto',
[...]
'pipeline-2__sc__with_std': True}
```

Based on the values returned by the `get_params` method, we now know how to access the individual classifier's attributes. Let's now tune the inverse regularization parameter, `C`, of the logistic regression classifier and the decision tree depth via a grid search for demonstration purposes:

```
>>> from sklearn.model_selection import GridSearchCV
>>> params = {'decisiontreeclassifier__max_depth': [1, 2],
...           'pipeline-1__clf__C': [0.001, 0.1, 100.0]}
>>> grid = GridSearchCV(estimator=mv_clf,
...                     param_grid=params,
...                     cv=10,
...                     scoring='roc_auc')
>>> grid.fit(X_train, y_train)
```

After the grid search has completed, we can print the different hyperparameter value combinations and the average ROC AUC scores computed via 10-fold cross-validation as follows:

```
>>> for r, _ in enumerate(grid.cv_results_['mean_test_score']):
...     mean_score = grid.cv_results_['mean_test_score'][r]
...     std_dev = grid.cv_results_['std_test_score'][r]
...     params = grid.cv_results_['params'][r]
...     print(f'{mean_score:.3f} +/- {std_dev:.2f} {params}')
0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 1,
                'pipeline-1__clf__C': 0.001}
0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 1,
                'pipeline-1__clf__C': 0.1}
0.967 +/- 0.10 {'decisiontreeclassifier__max_depth': 1,
                'pipeline-1__clf__C': 100.0}
0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 2,
                'pipeline-1__clf__C': 0.001}
0.983 +/- 0.05 {'decisiontreeclassifier__max_depth': 2,
                'pipeline-1__clf__C': 0.1}
0.967 +/- 0.10 {'decisiontreeclassifier__max_depth': 2,
                'pipeline-1__clf__C': 100.0}
>>> print(f'Best parameters: {grid.best_params_}')
Best parameters: {'decisiontreeclassifier__max_depth': 1,
                  'pipeline-1__clf__C': 0.001}
>>> print(f'ROC AUC : {grid.best_score_:.2f}')
ROC AUC: 0.98
```

As you can see, we get the best cross-validation results when we choose a lower regularization strength (`C=0.001`), whereas the tree depth does not seem to affect the performance at all, suggesting that a decision stump is sufficient to separate the data. To remind ourselves that it is a bad practice to use the test dataset more than once for model evaluation, we are not going to estimate the generalization performance of the tuned hyperparameters in this section. We will move on swiftly to an alternative approach for ensemble learning: **bagging**.

Building ensembles using stacking

The majority vote approach we implemented in this section is not to be confused with stacking. The stacking algorithm can be understood as a two-level ensemble, where the first level consists of individual classifiers that feed their predictions to the second level, where another classifier (typically logistic regression) is fit to the level-one classifier predictions to make the final predictions. For more information on stacking, see the following resources:

- The stacking algorithm has been described in more detail by David H. Wolpert in *Stacked generalization*, *Neural Networks*, 5(2):241–259, 1992 (`https://www.sciencedirect.com/science/article/pii/S0893608005800231`).
- Interested readers can find our video tutorial about stacking on YouTube at `https://www.youtube.com/watch?v=8T2emza6g80`.
- A scikit-learn compatible version of a stacking classifier is available from mlxtend: `http://rasbt.github.io/mlxtend/user_guide/classifier/StackingCVClassifier/`.
- Also, a `StackingClassifier` has recently been added to scikit-learn (available in version 0.22 and newer); for more information, please see the documentation at `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.StackingClassifier.html`.

Bagging – building an ensemble of classifiers from bootstrap samples

Bagging is an ensemble learning technique that is closely related to the `MajorityVoteClassifier` that we implemented in the previous section. However, instead of using the same training dataset to fit the individual classifiers in the ensemble, we draw bootstrap samples (random samples with replacement) from the initial training dataset, which is why bagging is also known as *bootstrap aggregating*.

The concept of bagging is summarized in *Figure 7.6*:

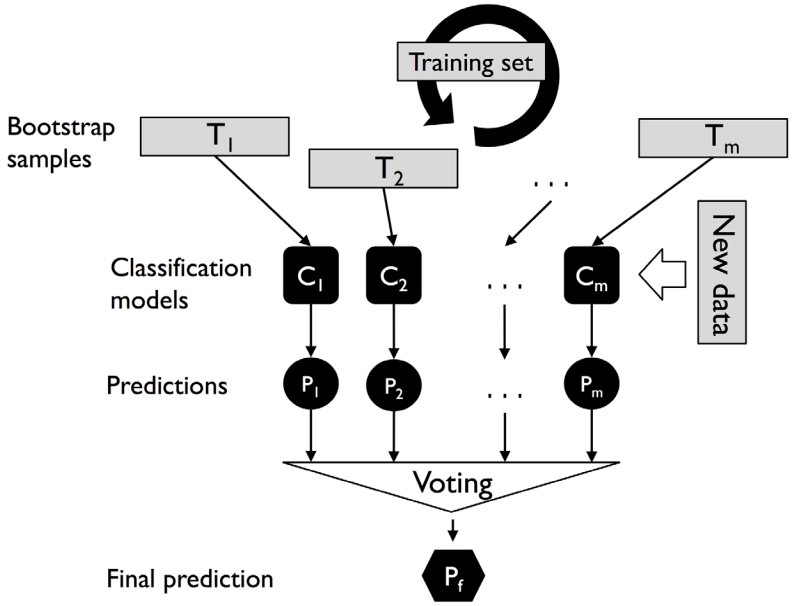


Figure 7.6: The concept of bagging

In the following subsections, we will work through a simple example of bagging by hand and use scikit-learn for classifying wine examples.

Bagging in a nutshell

To provide a more concrete example of how the bootstrap aggregating of a bagging classifier works, let's consider the example shown in *Figure 7.7*. Here, we have seven different training instances (denoted as indices 1-7) that are sampled randomly with replacement in each round of bagging. Each bootstrap sample is then used to fit a classifier, C_j, which is most typically an unpruned decision tree:

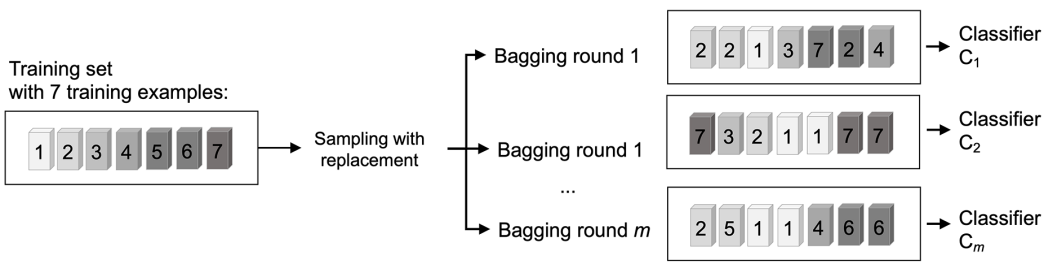


Figure 7.7: An example of bagging

As you can see from *Figure 7.7*, each classifier receives a random subset of examples from the training dataset. We denote these random samples obtained via bagging as *Bagging round 1*, *Bagging round 2*, and so on. Each subset contains a certain portion of duplicates and some of the original examples don't appear in a resampled dataset at all due to sampling with replacement. Once the individual classifiers are fit to the bootstrap samples, the predictions are combined using majority voting.

Note that bagging is also related to the random forest classifier that we introduced in *Chapter 3*. In fact, random forests are a special case of bagging where we also use random feature subsets when fitting the individual decision trees.

Model ensembles using bagging

Bagging was first proposed by Leo Breiman in a technical report in 1994; he also showed that bagging can improve the accuracy of unstable models and decrease the degree of overfitting. We highly recommend that you read about his research in *Bagging predictors* by *L. Breiman*, *Machine Learning*, 24(2):123–140, 1996, which is freely available online, to learn more details about bagging.

Applying bagging to classify examples in the Wine dataset

To see bagging in action, let's create a more complex classification problem using the Wine dataset that was introduced in *Chapter 4*, *Building Good Training Datasets – Data Preprocessing*. Here, we will only consider the Wine classes 2 and 3, and we will select two features – `Alcohol` and `OD280/OD315 of diluted wines`:

```
>>> import pandas as pd
>>> df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/'
...                       'machine-learning-databases/'
...                       'wine/wine.data',
...                       header=None)
>>> df_wine.columns = ['Class label', 'Alcohol',
...                    'Malic acid', 'Ash',
...                    'Alcalinity of ash',
...                    'Magnesium', 'Total phenols',
...                    'Flavanoids', 'Nonflavanoid phenols',
...                    'Proanthocyanins',
...                    'Color intensity', 'Hue',
...                    'OD280/OD315 of diluted wines',
...                    'Proline']
>>> # drop 1 class
>>> df_wine = df_wine[df_wine['Class label'] != 1]
>>> y = df_wine['Class label'].values
>>> X = df_wine[['Alcohol',
...              'OD280/OD315 of diluted wines']].values
```

Next, we will encode the class labels into binary format and split the dataset into 80 percent training and 20 percent test datasets:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> from sklearn.model_selection import train_test_split
>>> le = LabelEncoder()
>>> y = le.fit_transform(y)
>>> X_train, X_test, y_train, y_test =\
...            train_test_split(X, y,
...                             test_size=0.2,
...                             random_state=1,
...                             stratify=y)
```

Obtaining the Wine dataset

You can find a copy of the Wine dataset (and all other datasets used in this book) in the code bundle of this book, which you can use if you are working offline or the UCI server at `https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data` is temporarily unavailable. For instance, to load the Wine dataset from a local directory, take the following lines:

```
df = pd.read_csv('https://archive.ics.uci.edu/ml/'
                 'machine-learning-databases'
                 '/wine/wine.data',
                 header=None)
```

and replace them with these:

```
df = pd.read_csv('your/local/path/to/wine.data',
                 header=None)
```

A `BaggingClassifier` algorithm is already implemented in scikit-learn, which we can import from the `ensemble` submodule. Here, we will use an unpruned decision tree as the base classifier and create an ensemble of 500 decision trees fit on different bootstrap samples of the training dataset:

```
>>> from sklearn.ensemble import BaggingClassifier
>>> tree = DecisionTreeClassifier(criterion='entropy',
...                               random_state=1,
...                               max_depth=None)
>>> bag = BaggingClassifier(base_estimator=tree,
...                         n_estimators=500,
...                         max_samples=1.0,
...                         max_features=1.0,
...                         bootstrap=True,
```

```
...                         bootstrap_features=False,
...                         n_jobs=1,
...                         random_state=1)
```

Next, we will calculate the accuracy score of the prediction on the training and test datasets to compare the performance of the bagging classifier to the performance of a single unpruned decision tree:

```
>>> from sklearn.metrics import accuracy_score
>>> tree = tree.fit(X_train, y_train)
>>> y_train_pred = tree.predict(X_train)
>>> y_test_pred = tree.predict(X_test)
>>> tree_train = accuracy_score(y_train, y_train_pred)
>>> tree_test = accuracy_score(y_test, y_test_pred)
>>> print(f'Decision tree train/test accuracies '
...       f'{tree_train:.3f}/{tree_test:.3f}')
Decision tree train/test accuracies 1.000/0.833
```

Based on the accuracy values that we printed here, the unpruned decision tree predicts all the class labels of the training examples correctly; however, the substantially lower test accuracy indicates high variance (overfitting) of the model:

```
>>> bag = bag.fit(X_train, y_train)
>>> y_train_pred = bag.predict(X_train)
>>> y_test_pred = bag.predict(X_test)
>>> bag_train = accuracy_score(y_train, y_train_pred)
>>> bag_test = accuracy_score(y_test, y_test_pred)
>>> print(f'Bagging train/test accuracies '
...       f'{bag_train:.3f}/{bag_test:.3f}')
Bagging train/test accuracies 1.000/0.917
```

Although the training accuracies of the decision tree and bagging classifier are similar on the training dataset (both 100 percent), we can see that the bagging classifier has a slightly better generalization performance, as estimated on the test dataset. Next, let's compare the decision regions between the decision tree and the bagging classifier:

```
>>> x_min = X_train[:, 0].min() - 1
>>> x_max = X_train[:, 0].max() + 1
>>> y_min = X_train[:, 1].min() - 1
>>> y_max = X_train[:, 1].max() + 1
>>> xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
...                      np.arange(y_min, y_max, 0.1))
>>> f, axarr = plt.subplots(nrows=1, ncols=2,
...                         sharex='col',
...                         sharey='row',
```

```
...                          figsize=(8, 3))
>>> for idx, clf, tt in zip([0, 1],
...                         [tree, bag],
...                         ['Decision tree', 'Bagging']):
...     clf.fit(X_train, y_train)
...
...     Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
...     Z = Z.reshape(xx.shape)
...     axarr[idx].contourf(xx, yy, Z, alpha=0.3)
...     axarr[idx].scatter(X_train[y_train==0, 0],
...                        X_train[y_train==0, 1],
...                        c='blue', marker='^')
...     axarr[idx].scatter(X_train[y_train==1, 0],
...                        X_train[y_train==1, 1],
...                        c='green', marker='o')
...     axarr[idx].set_title(tt)
>>> axarr[0].set_ylabel('OD280/OD315 of diluted wines', fontsize=12)
>>> plt.tight_layout()
>>> plt.text(0, -0.2,
...          s='Alcohol',
...          ha='center',
...          va='center',
...          fontsize=12,
...          transform=axarr[1].transAxes)
>>> plt.show()
```

As we can see in the resulting plot, the piece-wise linear decision boundary of the three-node deep decision tree looks smoother in the bagging ensemble:

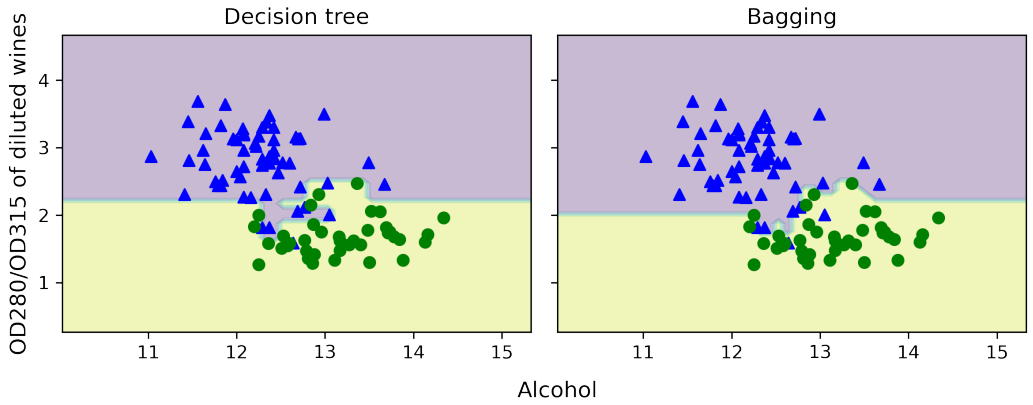


Figure 7.8: The piece-wise linear decision boundary of a decision tree versus bagging