

O'REILLY®



Early  
Release  
RAW &  
UNEDITED

# Hands-On Machine Learning with Scikit-Learn and PyTorch

Concepts, Tools, and Techniques  
to Build Intelligent Systems

Aurélien Géron

# **Hands-On Machine Learning with Scikit-Learn and PyTorch**

Concepts, Tools, and Techniques to Build Intelligent Systems

**Aurélien Geron**

**O'REILLY®**

# **Hands-On Machine Learning with Scikit-Learn and PyTorch**

by Aurélien Geron

Copyright © 2025 Aurélien Geron. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<https://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

Acquisitions Editor: Nicole Butterfield

Development Editor: Michele Cronin

Production Editor: Beth Kelly

Copyeditor: TO COME

Proofreader: TO COME

Indexer: TO COME

Interior Designer: David Futato

Cover Designer: TO COME

Illustrator: Kate Dullea

October 2025: First Edition

## **Revision History for the Early Release**

- 2025-05-15: First Release
- 2025-06-09: Second Release

See <https://oreilly.com/catalog/errata.csp?isbn=9798341607989> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Hands-On Machine Learning with Scikit-Learn and PyTorch*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

979-8-341-60793-4

[]

# Brief Table of Contents (Not Yet Final)

---

Part 1 The Fundamentals of Machine Learning

Chapter 1 The Machine Learning Landscape (available)

Chapter 2 End-to-End Machine Learning Projects (available)

Chapter 3 Classification (available)

Chapter 4 Training Models (available)

Chapter 5 Decision Trees (available)

Chapter 6 Ensemble Learning and Random Forests (available)

Chapter 7 Dimensionality Reduction (available)

Chapter 8 Unsupervised Learning Techniques (available)

Part 2 Neural Networks and Deep Learning

Chapter 9 Introduction to Artificial Neural Networks (available)

Chapter 10 Building Neural Networks with PyTorch (available)

Chapter 11 Training Deep Neural Networks (available)

*Chapter 12 Deep Computer Vision Using Convolutional Neural Networks (unavailable)*

*Chapter 13 Processing Sequences Using RNNs and CNNs (unavailable)*

*Chapter 14 Natural Language Processing with RNNs (unavailable)*

*Chapter 15 The Transformer Revolution (unavailable)*

*Chapter 16 Next-Generation Transformers and Beyond (unavailable)*

*Chapter 17 Autoencoders, GANs and Diffusion Models (unavailable)*

*Chapter 18 Reinforcement Learning (unavailable)*

*Appendix A Machine Learning Checklist (unavailable)*

*Appendix B Autodiff (unavailable)*

# Preface

---

## The Machine Learning Tsunami

In 2006, Geoffrey Hinton et al. published [a paper<sup>1</sup>](#) showing how to train a deep neural network capable of recognizing handwritten digits with state-of-the-art precision (>98%). They branded this technique “deep learning”. A deep neural network is a (very) simplified model of our cerebral cortex, composed of a stack of layers of artificial neurons. Training a deep neural net was widely considered impossible at the time,<sup>2</sup> and most researchers had abandoned the idea in the late 1990s. This paper revived the interest of the scientific community, and before long many new papers demonstrated that deep learning was not only possible, but capable of mind-blowing achievements that no other machine learning (ML) technique could hope to match (with the help of tremendous computing power and great amounts of data). This enthusiasm soon extended to many other areas of machine learning.

A decade later, machine learning had already conquered many industries, ranking web results, recommending videos to watch and products to buy, sorting items on production lines, sometimes even driving cars. Machine learning often made the headlines, for example when DeepMind’s AlphaFold machine learning system solved a long-standing protein-folding problem that had stumped researchers for decades. But most of the time, machine learning was just working discretely in the background. However, another decade later came the rise of AI assistants: from ChatGPT in 2022, Gemini, Claude, and Grok in 2023, and many others since then. AI has now truly taken off and it is rapidly transforming every single industry: what used to be Sci-Fi is now very real.<sup>3</sup>

## Machine Learning in Your Projects

So, naturally you are excited about machine learning and would love to join the party!

Perhaps you would like to give your homemade robot a brain of its own? Make it recognize faces? Or learn to walk around?

Or maybe your company has tons of data (user logs, financial data, production data, machine sensor data, hotline stats, HR reports, etc.), and more than likely you could

unearth some hidden gems if you just knew where to look. With machine learning, you could accomplish the following **and much more**:

- Segment customers and find the best marketing strategy for each group.
- Recommend products for each client based on what similar clients bought.
- Detect which transactions are likely to be fraudulent.
- Forecast next year's revenue.
- Predict peak workloads and suggest optimal staffing levels.
- Build a chatbot to assist your customers.

Whatever the reason, you have decided to learn machine learning and implement it in your projects. Great idea!

## Objective and Approach

This book assumes that you know close to nothing about machine learning. Its goal is to give you the concepts, tools, and intuition you need to implement programs capable of *learning from data*.

We will cover a large number of techniques, from the simplest and most commonly used (such as linear regression) to some of the deep learning techniques that regularly win competitions. For this, we will be using open-source and production-ready Python frameworks:

- **Scikit-Learn** is very easy to use, yet it implements many machine learning algorithms efficiently, so it makes for a great entry point to learning machine learning. It was created by David Cournapeau in 2007, then led by a team of researchers at the French Institute for Research in Computer Science and Automation (Inria), and recently Probabl.ai.
- **PyTorch** is a powerful and flexible library for deep learning. It makes it possible to train and run all sorts of neural networks efficiently, and it can distribute the computations across multiple GPUs (graphics processing units). PyTorch (PT) was developed by Facebook's AI Research lab (FAIR) and first released in 2016. It evolved from Torch, an older framework coded in Lua. In 2022, PyTorch was transitioned to the PyTorch Foundation, under the Linux Foundation, to promote community-driven development.

We will also use a few other open-source machine learning libraries along the way, including:

- **XGBoost** in Chapter 6, to implement a powerful technique called *gradient boosting*.
- **Hugging Face** libraries in Chapters [Link to Come] and [Link to Come] to download datasets and pretrained models, including Transformer models. Transformers are incredibly powerful and versatile, and they are the main building block of virtually all AI assistants today.
- **Gymnasium** in [Link to Come], for reinforcement learning (i.e., training autonomous agents).

The book favors a hands-on approach, growing an intuitive understanding of machine learning through concrete working examples and just a little bit of theory.

### TIP

While you can read this book without picking up your laptop, I highly recommend you experiment with the code examples.

## Code Examples

All the code examples in this book are open source and available online at <https://github.com/ageron/handson-ml2>, as Jupyter notebooks. These are interactive documents containing text, images, and executable code snippets (Python in our case). The easiest and quickest way to get started is to run these notebooks using Google Colab: this is a free service that allows you to run any Jupyter notebook directly online, without having to install anything on your machine. All you need is a web browser and a Google account.

### NOTE

In this book, I will assume that you are using Google Colab, but I have also tested the notebooks on other online platforms such as Kaggle and Binder, so you can use those if you prefer. Alternatively, you can install the required libraries and tools (or the Docker image for this book) and run the notebooks directly on your own machine. See the instructions at <https://homl.info/install-p>.

This book is here to help you get your job done. If you wish to use additional content beyond the code examples, and that use falls outside the scope of fair use guidelines, (such as selling or distributing content from O'Reilly books, or incorporating a significant amount of material from this book into your product's documentation), please reach out to O'Reilly for permission, at [permissions@oreilly.com](mailto:permissions@oreilly.com).

I appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*Hands-On Machine Learning with Scikit-Learn and PyTorch* by Aurélien Géron. Copyright 2025 Aurélien Géron, 978-1-098-12597-4.”

## Prerequisites

This book assumes that you have some Python programming experience. If you don't know Python yet, <https://learnpython.org> is a great place to start. The official tutorial on [Python.org](https://python.org) is also quite good.

This book also assumes that you are familiar with Python's main scientific libraries—in particular, [NumPy](#), [Pandas](#), and [Matplotlib](#). If you have never used these libraries, don't worry; they're easy to learn, and I've created a tutorial for each of them. You can access them online at <https://homl.info/tutorials-p>.

Moreover, if you want to fully understand how the machine learning algorithms work (not just how to use them), then you should have at least a basic understanding of a few math concepts, especially linear algebra. Specifically, you should know what vectors and matrices are, and how to perform some simple operations like adding vectors, or transposing and multiplying matrices. If you need a quick introduction to linear algebra (it's really not rocket science!), I provide a tutorial at <https://homl.info/tutorials-p>. You will also find a tutorial on differential calculus, which may be helpful to understand how neural networks are trained, but it's not entirely essential to grasp the important concepts. This book also uses other mathematical concepts occasionally, such as exponentials and logarithms, a bit of probability theory, and some basic concepts from statistics, but nothing too advanced. If you need help on any of these, please check out <https://khanacademy.org>, which offers many excellent and free math courses online.

## Roadmap

This book is organized in two parts. [Part I](#), “The Fundamentals of Machine Learning”, covers the following topics:

- What machine learning is, what problems it tries to solve, and the main categories and fundamental concepts of its systems
- The steps in a typical machine learning project
- Learning by fitting a model to data
- Minimizing a cost function (i.e., a measure of prediction errors)
- Handling, cleaning, and preparing data
- Selecting and engineering features (i.e., data fields)
- Selecting a model and tuning hyperparameters using cross-validation (e.g., training many model variants and choosing the one that performs best on data it didn't see during training)
- The challenges of machine learning, in particular underfitting and overfitting (the bias/variance trade-off)
- The most common learning algorithms: linear and polynomial regression, logistic regression,  $k$ -nearest neighbors, decision trees, random forests, and ensemble methods
- Reducing the dimensionality of the training data to fight the “curse of dimensionality”
- Other unsupervised learning techniques, including clustering, density estimation, and anomaly detection

Part II, “Neural Networks and Deep Learning”, Part 2 covers the following topics:

- What neural nets are and what they’re good for
- Building and training deep neural nets using PyTorch
- The most important neural net architectures: feedforward neural nets for tabular data; convolutional nets for computer vision; recurrent nets and long short-term memory (LSTM) nets for sequence processing; encoder–decoders, transformers, state space models (SSMs), and hybrid architectures for natural language processing, vision, and more; autoencoders, generative adversarial networks (GANs), and diffusion models for generative learning

- How to build an agent (e.g., a bot in a game) that can learn good strategies through trial and error, using reinforcement learning
- Loading and preprocessing large amounts of data efficiently

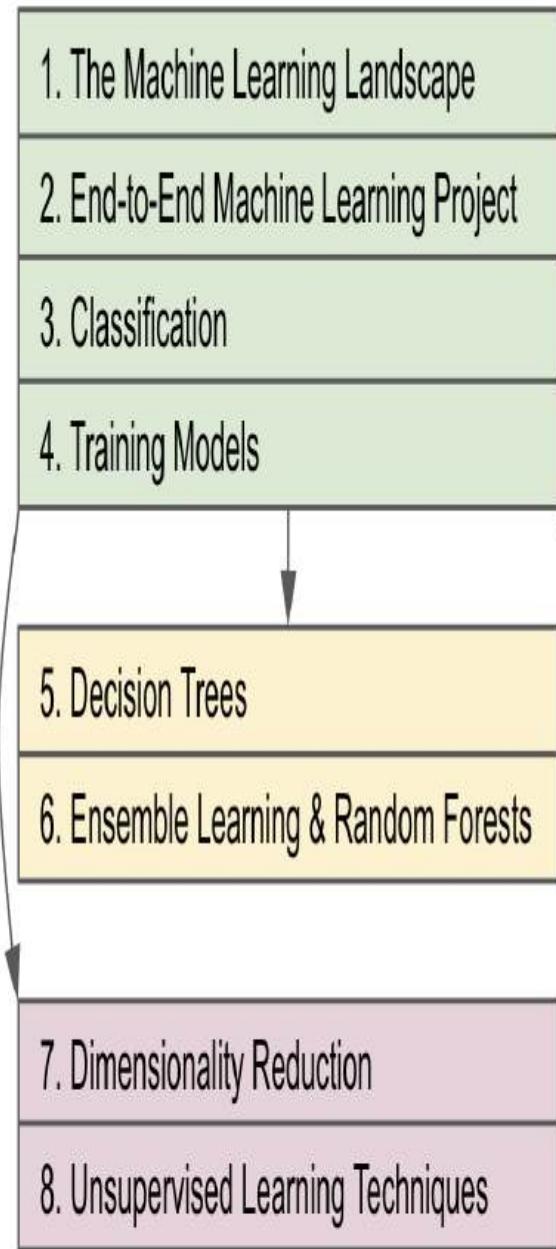
The first part is based mostly on Scikit-Learn, while the second part uses mostly PyTorch.

## CAUTION

Don't jump into deep waters too hastily: deep learning is no doubt one of the most exciting areas in machine learning, but you should master the fundamentals first. Moreover, many problems can be solved quite well using simpler techniques such as random forests and ensemble methods (discussed in [Part I](#)). Deep learning is best suited for complex problems such as image recognition, speech recognition, or natural language processing, and it often requires a lot of data, computing power, and patience (unless you can leverage a pretrained neural network, as you will see).

If you are particularly interested in one topic and want to reach it as quickly as possible, [Figure P-1](#) will show you which chapters you must read first, and which ones you can safely skip.

## Part 1 – The Fundamentals of ML



## Part 2 – Neural Networks and Deep Learning

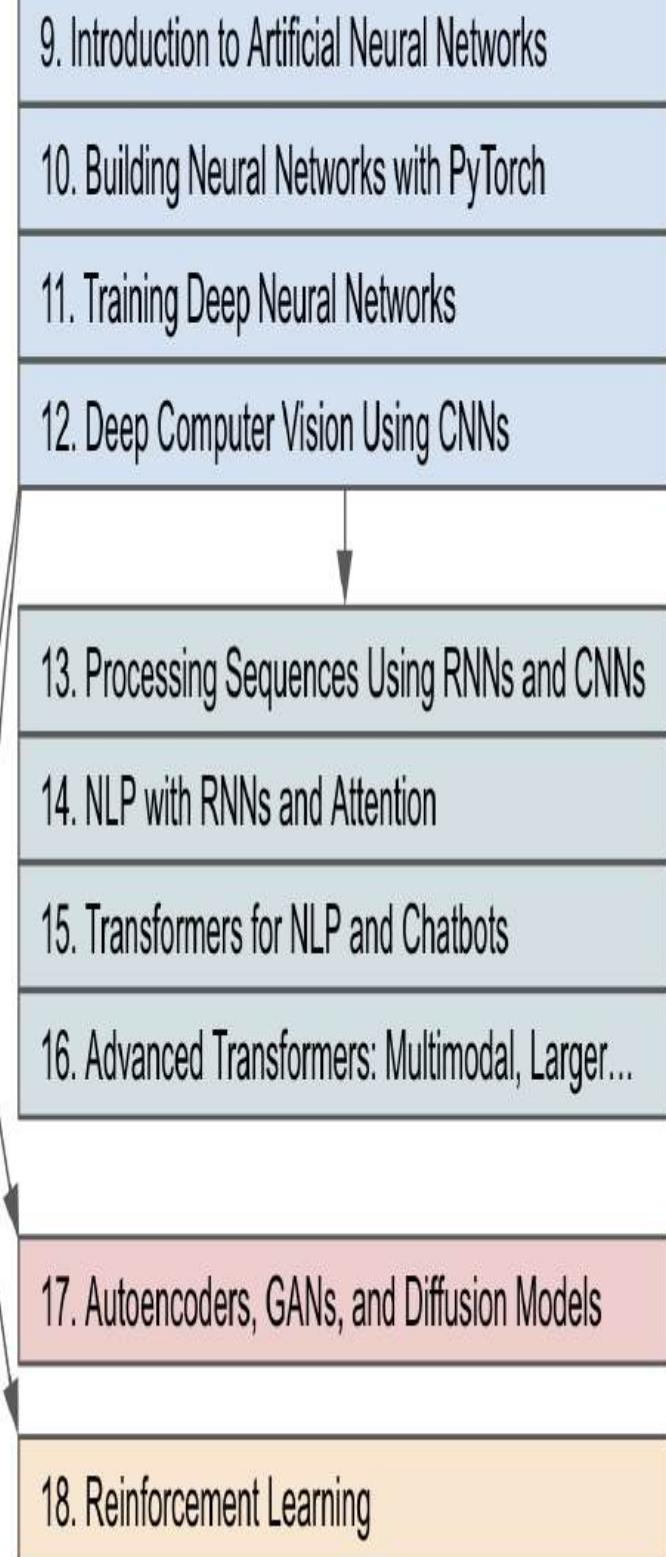


Figure P-1. Chapter Dependencies

# Changes Between the TensorFlow and PyTorch Versions

I wrote three TensorFlow editions of this book, published by O'Reilly in 2017, 2019, and 2022. TF was the leading deep learning library for many years, used internally by Google and therefore optimized for production at scale. But PyTorch has gradually taken the lead, owing to its simplicity, flexibility and openness: it now dominates research papers and open-source projects, which means that most new models are available in PyTorch first. As a result, the industry has also gradually shifted towards PyTorch. In recent years, Google has reduced its investments in TensorFlow, and focused more on JAX, another excellent deep learning library with a great mix of qualities for both research and production. However, its adoption is still low compared to PyTorch. This is why I've chosen to use PyTorch this time around!

If you have already read the latest TensorFlow version of this book, here are the main changes you will find in this book (see <https://homl.info/changes-p> for more details):

- All the code in the book was updated to recent library versions.
- All the code in **Part II** was migrated from TensorFlow & Keras to PyTorch. There were significant changes in all of these chapters.
- TensorFlow-specific content was removed, including former chapters 12 and 13, and former appendices C and D.
- A new chapter **Chapter 10** now introduces PyTorch.
- I also added a new chapter on transformers for natural language understanding and generation ([Link to Come]), in which you get to build a chatbot!
- And I added another new chapter on transformers for vision and multimodal models ([Link to Come]), in which I also cover several advanced techniques, such as mixture of experts (MoE), low-rank adapters (LoRA), state space models (SSMs), hybrid architectures, and more.
- To make room for all the new content, the chapters on support vector machines (SVMs) and deployment were moved online to the GitHub repository.

The three editions of the TensorFlow/Keras version of this book are nicknamed homl1, homl2, and homl3. This book, which is the first edition of the PyTorch version, is nicknamed homlp. Try saying that 3 times in a row.

## NOTE

Most of the changes compared to the latest TensorFlow edition are in the second part of the book. If you have read homl3, then don't expect big changes in the first part of the book: the fundamental concepts of machine learning haven't changed since 2022.

## Other Resources

Many excellent resources are available to learn about machine learning. For example, Andrew Ng's [ML course on Coursera](#) is amazing, although it requires a significant time investment.

There are also many interesting websites about machine learning, including Scikit-Learn's exceptional [User Guide](#). You may also enjoy [Dataquest](#), which provides very nice interactive tutorials, and countless ML blogs and YouTube channels.

There are many other introductory books about machine learning. In particular:

- Joel Grus's [\*Data Science from Scratch\*](#), 2nd edition (O'Reilly), presents the fundamentals of machine learning and implements some of the main algorithms in pure Python (from scratch, as the name suggests).
- Stephen Marsland's [\*Machine Learning: An Algorithmic Perspective\*](#), 2nd edition (Chapman & Hall), is a great introduction to machine learning, covering a wide range of topics in depth with code examples in Python (also from scratch, but using NumPy).
- Sebastian Raschka's [\*Machine Learning with PyTorch and Scikit-Learn\*](#), 1st edition (Packt Publishing), is also a great introduction to machine learning using Scikit-Learn and PyTorch.
- François Chollet's [\*Deep Learning with Python\*](#), 3rd edition (Manning), is a very practical book that covers a large range of topics in a clear and concise way, as you might expect from the author of the excellent Keras library. It favors code examples over mathematical theory.
- Andriy Burkov's [\*The Hundred-Page Machine Learning Book\*](#) (self-published) is very short but covers an impressive range of topics, introducing them in approachable terms without shying away from the math equations.

- Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin's *Learning from Data* (AMLBook) is a more theoretical approach to ML that provides deep insights, in particular on the bias/variance trade-off (see [Chapter 4](#)).
- Stuart Russell and Peter Norvig's *Artificial Intelligence: A Modern Approach*, 4th edition (Pearson), is a great (and huge) book covering an incredible amount of topics, including machine learning. It helps put ML into perspective.
- Jeremy Howard and Sylvain Gugger's *Deep Learning for Coders with fastai and PyTorch* (O'Reilly) provides a wonderfully clear and practical introduction to deep learning using the fastai and PyTorch libraries.
- Andrew Ng's *Machine Learning Yearning* is a free ebook that provides a thoughtful exploration of machine learning, focusing on the practical considerations of building and deploying models, including data quality and long-term maintenance.
- Lewis Tunstall, Leandro von Werra, and Thomas Wolf's *Natural Language Processing with Transformers: Building Language Applications with Hugging Face* (O'Reilly) is a great practical dive into Transformers using popular libraries by Hugging Face.
- Jay Alammar and Maarten Grootendorst's *Hands-On Large Language Models* is a beautifully illustrated book on LLMs, covering everything you need to know to understand, train, fine-tune, and use LLMs across a wide variety of tasks.

Finally, joining ML competition websites such as [Kaggle.com](#) will allow you to practice your skills on real-world problems, with help and insights from some of the best ML professionals out there.

## Conventions Used in This Book

The following typographical conventions are used in this book:

### *Italic*

Indicates new terms, URLs, email addresses, filenames, and file extensions.

### Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

#### **Constant width bold**

Shows commands or other text that should be typed literally by the user.

#### *Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.

#### *Punctuation*

To avoid any confusion, punctuation appears outside of quotes throughout the book. My apologies to the purists.

#### **TIP**

This element signifies a tip or suggestion.

#### **NOTE**

This element signifies a general note.

#### **WARNING**

This element indicates a warning or caution.

## **O'Reilly Online Learning**

#### **NOTE**

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit <https://oreilly.com>.

## How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

*[support@oreilly.com](mailto:support@oreilly.com)*

*<https://oreilly.com/about/contact.html>*

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at <https://homl.info/oreilly-p>.

For news and information about our books and courses, visit <https://oreilly.com>.

Find us on LinkedIn: <https://linkedin.com/company/oreilly-media>

Watch us on YouTube: <https://youtube.com/oreillymedia>

## Acknowledgments

Never in my wildest dreams did I imagine that the three TensorFlow editions of this book would reach such a large audience. I received so many messages from readers, many asking questions, some kindly pointing out errata, and most sending me

encouraging words. I cannot express how grateful I am to all these readers for their tremendous support. Thank you all so very much! Please do not hesitate to [file issues on GitHub](#) if you find errors in the code examples (or just to ask questions), or to submit [errata](#) if you find errors in the text. Some readers also shared how this book helped them get their first job, or how it helped them solve a concrete problem they were working on. I find such feedback incredibly motivating. If you find this book helpful, I would love it if you could share your story with me, either privately (e.g., via [LinkedIn](#)) or publicly (e.g., tweet me at [@aureliengeron](#) or write an [Amazon review](#)).

Huge thanks as well to all the wonderful people who offered their time and expertise to review this book, correcting errors and making countless suggestions.

This book wouldn't exist without O'Reilly's fantastic staff.

Last but not least, I am infinitely grateful to my beloved wife, Emmanuelle, and to our three wonderful children, Alexandre, Rémi, and Gabrielle, for encouraging me to work hard on this book. Their insatiable curiosity was priceless: explaining some of the most difficult concepts in this book to my wife and children helped me clarify my thoughts and directly improved many parts of it. Plus, they keep bringing me cookies and coffee, who could ask for more?

---

<sup>1</sup> Geoffrey E. Hinton et al., “A Fast Learning Algorithm for Deep Belief Nets”, *Neural Computation* 18 (2006): 1527–1554.

<sup>2</sup> Despite the fact that Yann LeCun’s deep convolutional neural networks had worked well for image recognition since the 1990s, although they were not as general-purpose.

<sup>3</sup> Geoffrey Hinton was awarded the 2018 Turing Award—along with Yann LeCun and Yoshua Bengio—as well as the 2024 Nobel prize in Physics along with John Hopfield, for their early work on neural networks back in the 1980s. DeepMind’s founder and CEO Demis Hassabis and director John Jumper were awarded the 2024 Nobel prize in Chemistry for their work on AlphaFold. They shared this Nobel prize with another protein researcher David Baker.

# **Part I. The Fundamentals of Machine Learning**

---

# Chapter 1. The Machine Learning Landscape

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 1st chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

Not so long ago, if you had picked up your phone and asked it the way home, it would have ignored you—and people would have questioned your sanity. But machine learning is no longer science fiction: billions of people use it every day. And the truth is it has actually been around for decades in some specialized applications, such as optical character recognition (OCR). The first ML application that really became mainstream, improving the lives of hundreds of millions of people, discretely took over the world back in the 1990s: the *spam filter*. It’s not exactly a self-aware robot, but it does technically qualify as machine learning: it has actually learned so well that you seldom need to flag an email as spam anymore. Then thanks to big data, hardware improvements, and a few algorithmic innovations, hundreds of ML applications followed and now quietly power hundreds of products and features that you use regularly: voice prompts, automatic translation, image search, product recommendations, and many more. And finally came ChatGPT, Gemini (formerly Bard), Claude, Perplexity, and many other chatbots: AI is no longer just powering services in the background, it **is** the service itself.

Where does machine learning start and where does it end? What exactly does it mean for a machine to *learn* something? If I download a copy of all Wikipedia articles, has my computer really learned something? Is it suddenly smarter? In this chapter I will start by clarifying what machine learning is and why you may want to use it.

Then, before we set out to explore the machine learning continent, we will take a look at the map and learn about the main regions and the most notable landmarks: supervised

versus unsupervised learning and their variants, online versus batch learning, instance-based versus model-based learning. Then we will look at the workflow of a typical ML project, discuss the main challenges you may face, and cover how to evaluate and fine-tune a machine learning system.

This chapter introduces a lot of fundamental concepts (and jargon) that every data scientist should know by heart. It will be a high-level overview (it's the only chapter without much code), all rather simple, but my goal is to ensure everything is crystal clear to you before we continue on to the rest of the book. So grab a coffee and let's get started!

### TIP

If you are already familiar with machine learning basics, you may want to skip directly to [Chapter 2](#). If you are not sure, try to answer all the questions listed at the end of the chapter before moving on.

## What Is Machine Learning?

Machine learning is the science (and art) of programming computers so they can *learn from data*.

Here is a slightly more general definition:

*[Machine learning is the] field of study that gives computers the ability to learn without being explicitly programmed.*

—Arthur Samuel, 1959

And a more engineering-oriented one:

*A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.*

—Tom Mitchell, 1997

Your spam filter is a machine learning program that, given examples of spam emails (flagged by users) and examples of regular emails (nonspam, also called “ham”), can learn to flag spam. The examples that the system uses to learn are called the *training set*. Each training example is called a *training instance* (or *sample*). The part of a machine learning system that learns and makes predictions is called a *model*. Neural networks and random forests are examples of models.

In this case, the task  $T$  is to flag spam for new emails, the experience  $E$  is the *training data*, and the performance measure  $P$  needs to be defined; for example, you can use the ratio of correctly classified emails. This particular performance measure is called *accuracy*, and it is often used in classification tasks (we will discuss several others in [Chapter 3](#)).

If you just download a copy of all Wikipedia articles, your computer has a lot more data, but it is not suddenly better at any task. This is not machine learning.

## Why Use Machine Learning?

Consider how you would write a spam filter using traditional programming techniques ([Figure 1-1](#)):

1. First you would examine what spam typically looks like. You might notice that some words or phrases (such as “4U”, “credit card”, “free”, and “amazing”) tend to come up a lot in the subject line. Perhaps you would also notice a few other patterns in the sender’s name, the email’s body, and other parts of the email.
2. You would write a detection algorithm for each of the patterns that you noticed, and your program would flag emails as spam if a number of these patterns were detected.
3. You would test your program and repeat steps 1 and 2 until it was good enough to launch.

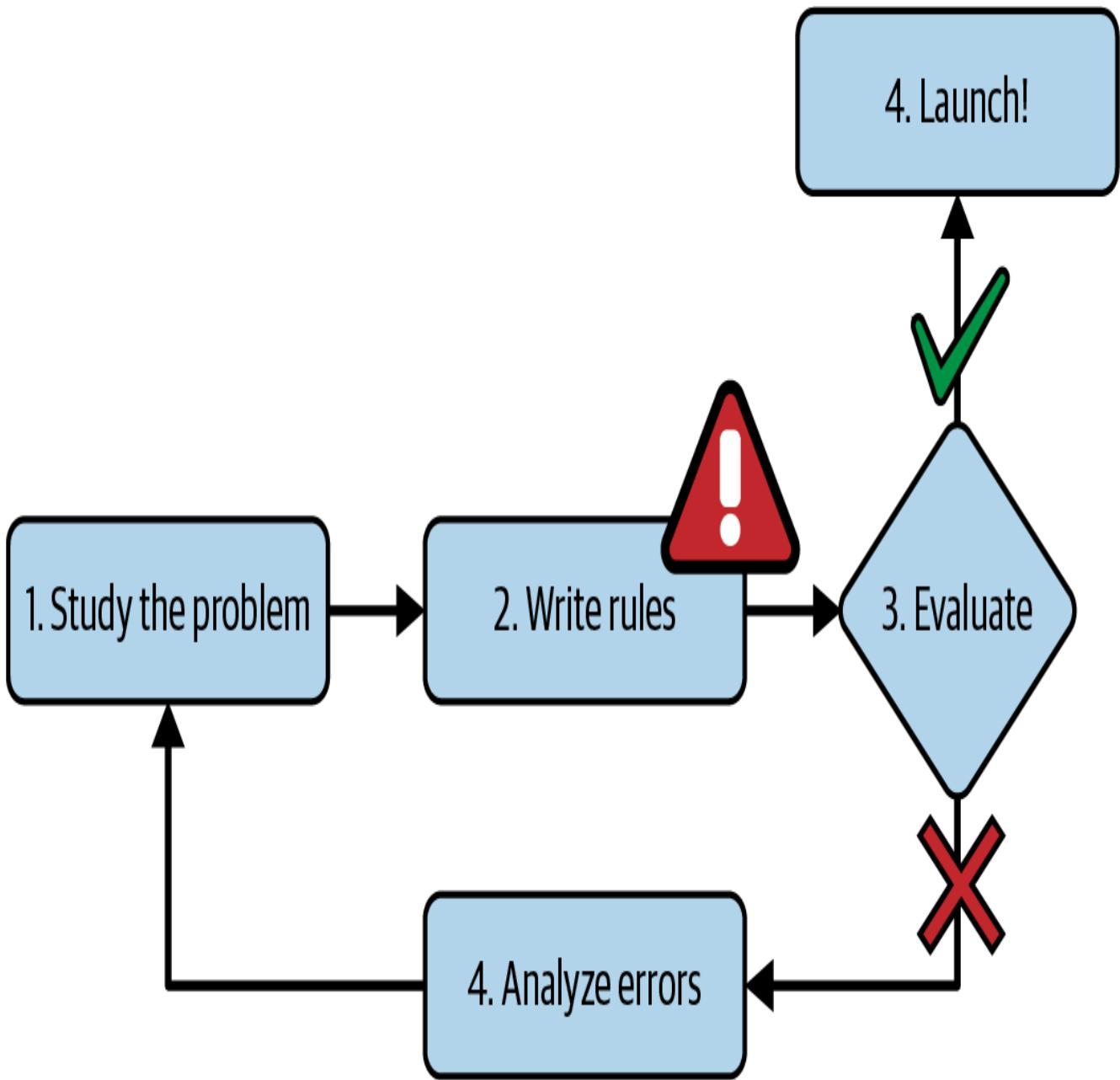


Figure 1-1. The traditional approach

Since the problem is difficult, your program will likely become a long list of complex rules—pretty hard to maintain.

In contrast, a spam filter based on machine learning techniques automatically learns which words and phrases are good predictors of spam by detecting unusually frequent patterns of words in the spam examples compared to the ham examples ([Figure 1-2](#)). The program is much shorter, easier to maintain, and most likely more accurate.

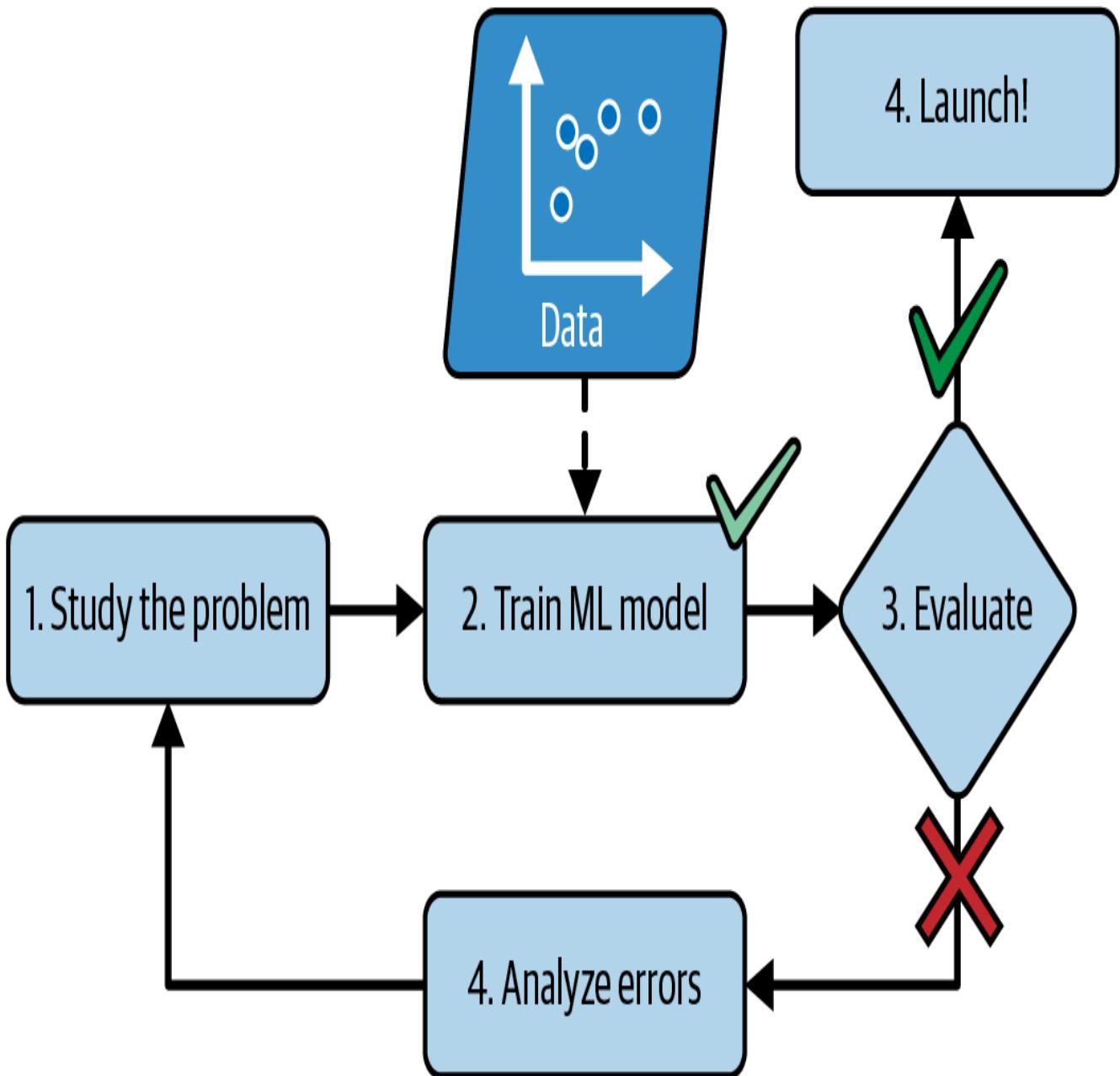
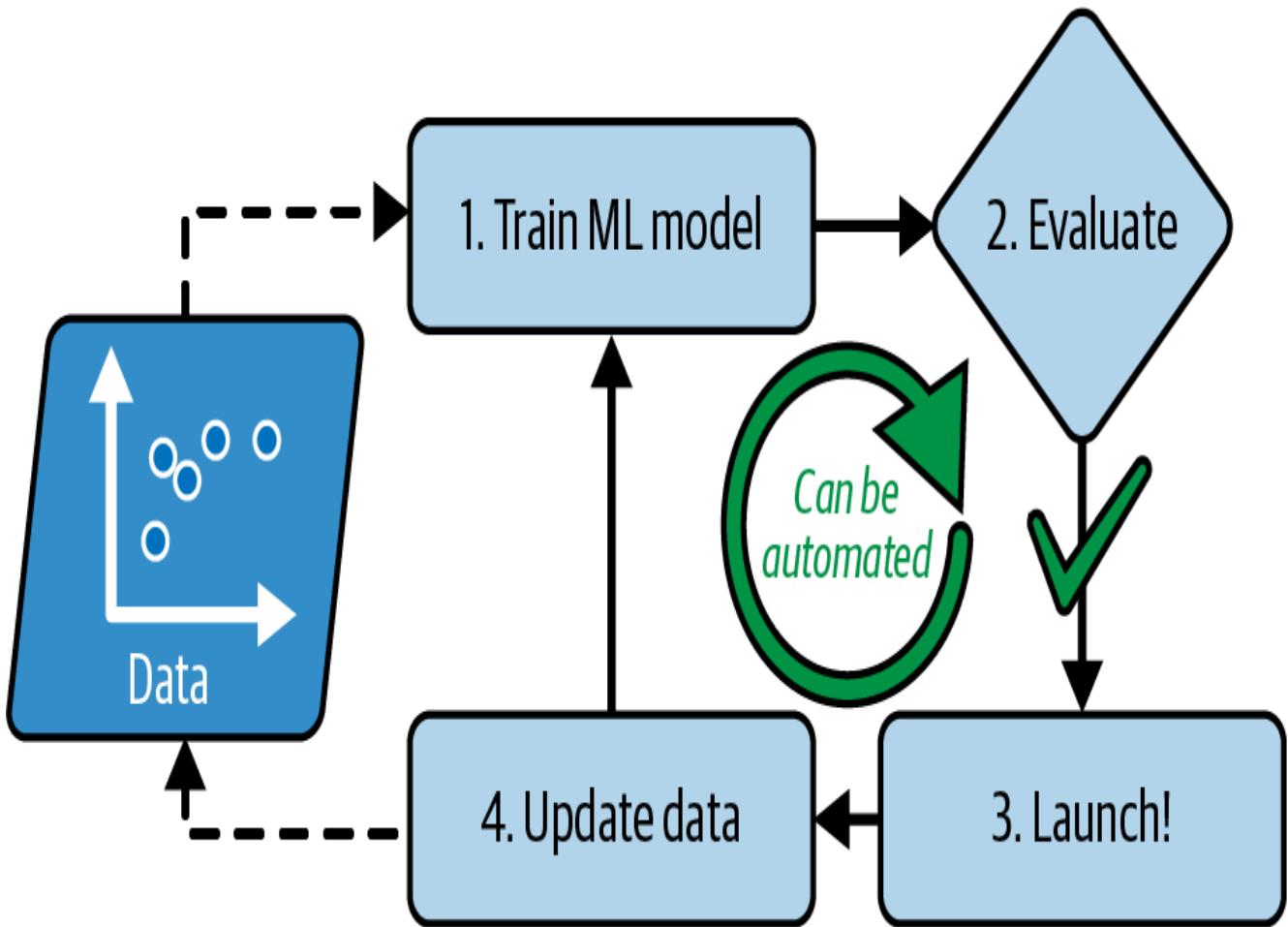


Figure 1-2. The machine learning approach

What if spammers notice that all their emails containing “4U” are blocked? They might start writing “For U” instead. A spam filter using traditional programming techniques would need to be updated to flag “For U” emails. If spammers keep working around your spam filter, you will need to keep writing new rules forever.

In contrast, a spam filter based on machine learning techniques automatically notices that “For U” has become unusually frequent in spam flagged by users, and it starts flagging them without your intervention (Figure 1-3).



*Figure 1-3. Automatically adapting to change*

Another area where machine learning shines is for problems that either are too complex for traditional approaches or have no known algorithm. For example, consider speech recognition. Say you want to start simple and write a program capable of distinguishing the words “one” and “two”. You might notice that the word “two” starts with a high-pitch sound (“T”), so you could hardcode an algorithm that measures high-pitch sound intensity and use that to distinguish ones and twos—but obviously this technique will not scale to thousands of words spoken by millions of very different people in noisy environments and in dozens of languages. The best solution (at least today) is to write an algorithm that learns by itself, given many example recordings for each word.

Finally, machine learning can help humans learn ([Figure 1-4](#)). ML models can be inspected to see what they have learned (although for some models this can be tricky). For instance, once a spam filter has been trained on enough spam, it can easily be inspected to reveal the list of words and combinations of words that it believes are the best predictors of spam. Sometimes this will reveal unsuspected correlations or new trends, and thereby lead to a better understanding of the problem. Digging into large amounts of data to discover hidden patterns is called *data mining*, and machine learning excels at it.

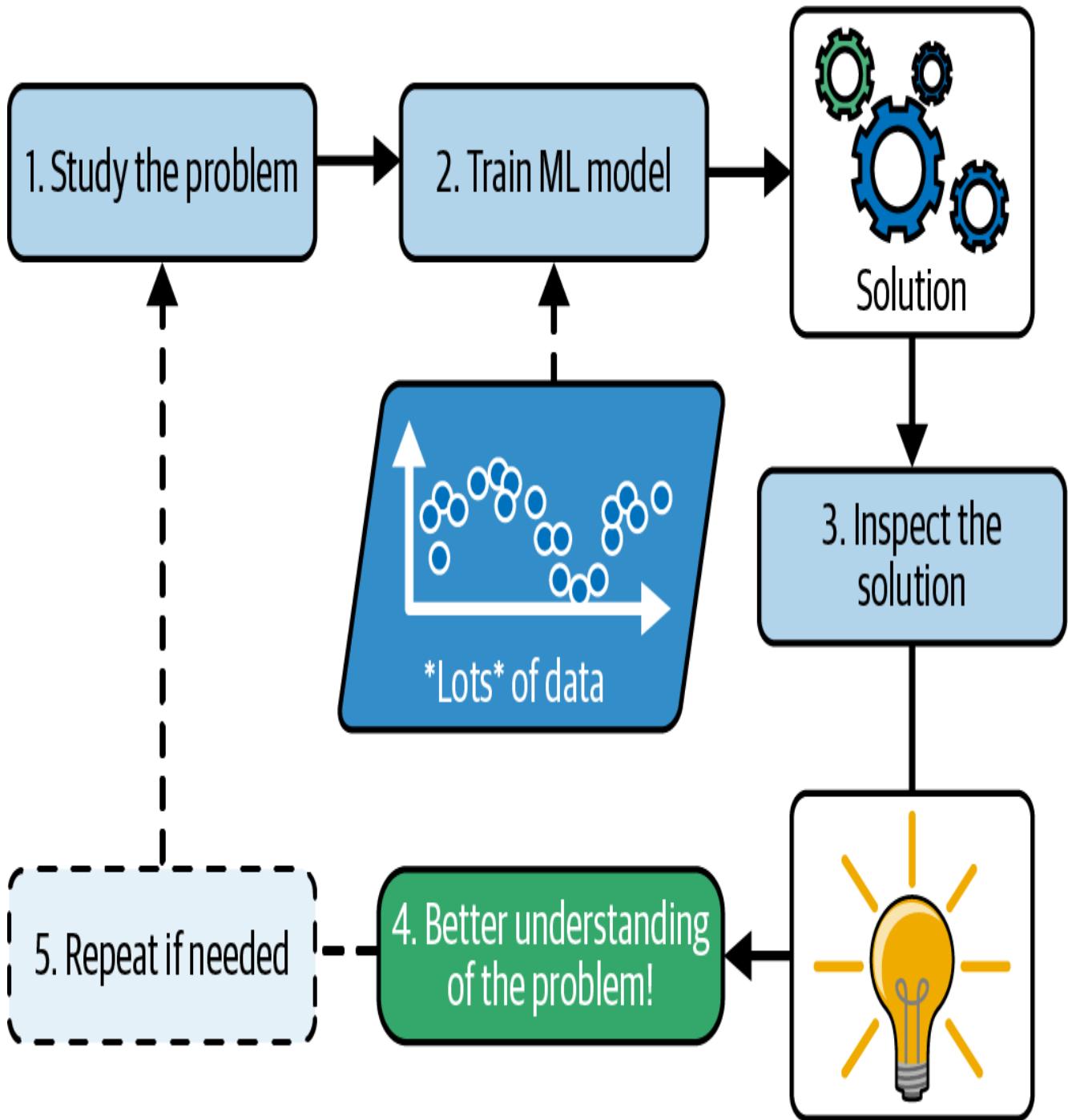


Figure 1-4. Machine learning can help humans learn

To summarize, machine learning is great for:

- Problems for which existing solutions require a lot of work and maintenance, such as long lists of rules (a machine learning model can often simplify code and perform better than the traditional approach)
- Complex problems for which using a traditional approach yields no good solution (the best machine learning techniques can perhaps find a solution)

- Fluctuating environments (a machine learning system can easily be retrained on new data, always keeping it up to date)
- Getting insights about complex problems and large amounts of data

## Examples of Applications

Let's look at some concrete examples of machine learning tasks, along with the techniques that can tackle them:

*Analyzing images of products on a production line to automatically classify them*

This is image classification, typically performed using convolutional neural networks (CNNs; see [Link to Come]) or vision transformers (see [Link to Come]).

*Detecting tumors in brain scans*

This is semantic image segmentation, where each pixel in the image is classified (as we want to determine the exact location and shape of tumors), typically using CNNs or vision transformers.

*Automatically classifying news articles*

This is natural language processing (NLP), and more specifically text classification, which can be tackled using recurrent neural networks (RNNs) and CNNs, but transformers work even better (see [Link to Come]).

*Automatically flagging offensive comments on discussion forums*

This is also text classification, using the same NLP tools.

*Summarizing long documents automatically*

This is a branch of NLP called text summarization, again using the same tools.

*Estimating a person's genetic risk for a given disease by analyzing a very long DNA sequence*

Such a task requires discovering spread out patterns across very long sequences, which is where state space models (SSMs) particularly

shine (see [Link to Come]).

#### *Creating a chatbot or a personal assistant*

This involves many NLP components, including natural language understanding (NLU) and question-answering modules.

#### *Forecasting your company's revenue next year, based on many performance metrics*

This is a regression task (i.e., predicting values) that may be tackled using any regression model, such as a linear regression or polynomial regression model (see [Chapter 4](#)), a regression support vector machine (see the online chapter on SVMs at <https://homl.info/svm-p>), a regression random forest (see [Chapter 6](#)), or an artificial neural network (see [Chapter 9](#)). If you want to take into account sequences of past performance metrics, you may want to use RNNs, CNNs, or transformers (see Chapters [Link to Come] to [Link to Come]).

#### *Making your app react to voice commands*

This is speech recognition, which requires processing audio samples: since they are long and complex sequences, they are typically processed using RNNs, CNNs, or transformers (see Chapters [Link to Come] to [Link to Come]).

#### *Detecting credit card fraud*

This is anomaly detection, which can be tackled using isolation forests, Gaussian mixture models (see [Chapter 8](#)), or autoencoders (see [Link to Come]).

#### *Segmenting clients based on their purchases so that you can design a different marketing strategy for each segment*

This is clustering, which can be achieved using  $k$ -means, DBSCAN, and more (see [Chapter 8](#)).

#### *Representing a complex, high-dimensional dataset in a clear and insightful diagram*

This is data visualization, often involving dimensionality reduction techniques (see [Chapter 7](#)).

*Recommending a product that a client may be interested in, based on past purchases*

This is a recommender system. One approach is to feed past purchases (and other information about the client) to an artificial neural network (see [Chapter 9](#)), and get it to output the most likely next purchase. This neural net would typically be trained on past sequences of purchases across all clients.

*Building an intelligent bot for a game*

This is often tackled using reinforcement learning (RL; see [Link to Come]), which is a branch of machine learning that trains agents (such as bots) to pick the actions that will maximize their rewards over time (e.g., a bot may get a reward every time the player loses some life points), within a given environment (such as the game). The famous AlphaGo program that beat the world champion at the game of Go was built using RL.

This list could go on and on, but hopefully it gives you a sense of the incredible breadth and complexity of the tasks that machine learning can tackle, and the types of techniques that you would use for each task.

## Types of Machine Learning Systems

There are so many different types of machine learning systems that it is useful to classify them in broad categories, based on the following criteria:

- How they are guided during training (supervised, unsupervised, semi-supervised, self-supervised, and others)
- Whether or not they can learn incrementally on the fly (online versus batch learning)
- Whether they work by simply comparing new data points to known data points, or instead by detecting patterns in the training data and building a predictive model, much like scientists do (instance-based versus model-based learning)

These criteria are not exclusive; you can combine them in any way you like. For example, a state-of-the-art spam filter may learn on the fly using a deep neural network

model trained using human-provided examples of spam and ham; this makes it an online, model-based, supervised learning system.

Let's look at each of these criteria a bit more closely.

## Training Supervision

ML systems can be classified according to the amount and type of supervision they get during training. There are many categories, but we'll discuss the main ones: supervised learning, unsupervised learning, self-supervised learning, semi-supervised learning, and reinforcement learning.

### Supervised learning

In *supervised learning*, the training set you feed to the algorithm includes the desired solutions, called *labels* ([Figure 1-5](#)).

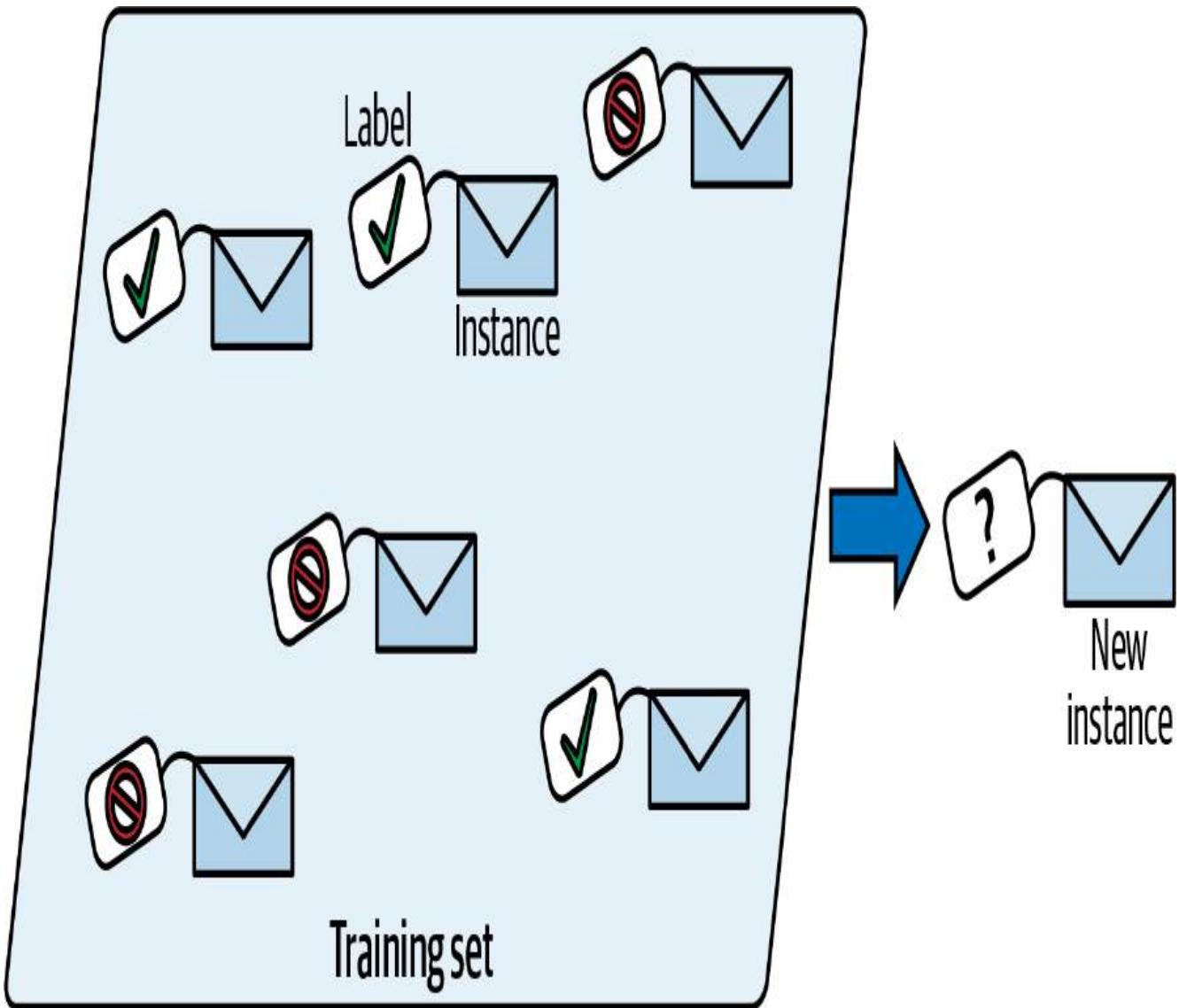


Figure 1-5. A labeled training set for spam classification (an example of supervised learning)

A typical supervised learning task is *classification*. The spam filter is a good example of this: it is trained with many example emails along with their *class* (spam or ham), and it must learn how to classify new emails.

Another typical task is to predict a *target* numeric value, such as the price of a car, given a set of *features* (mileage, age, brand, etc.). This sort of task is called *regression* (Figure 1-6).<sup>1</sup> To train the system, you need to give it many examples of cars, including both their features and their targets (i.e., their prices).

Note that some regression models can be used for classification as well, and vice versa. For example, *logistic regression* is commonly used for classification, as it can output a value that corresponds to the probability of belonging to a given class (e.g., 20% chance of being spam).

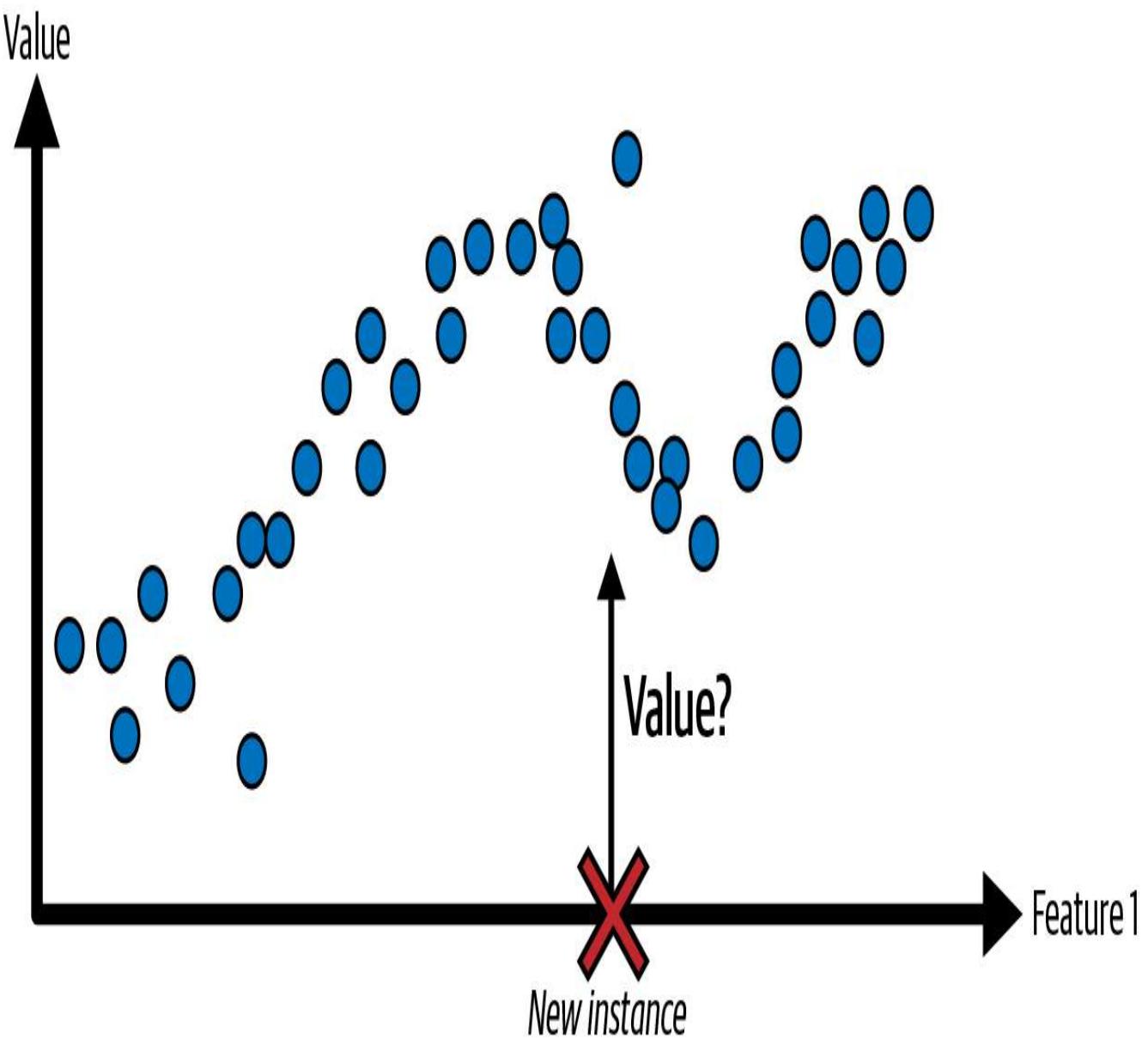


Figure 1-6. A regression problem: predict a value, given an input feature (there are usually multiple input features, and sometimes multiple output values)

#### NOTE

The words *target* and *label* are generally treated as synonyms in supervised learning, but *target* is more common in regression tasks and *label* is more common in classification tasks. Moreover, *features* are sometimes called *predictors* or *attributes*. These terms may refer to individual samples (e.g., “this car’s mileage feature is equal to 15,000”) or to all samples (e.g., “the mileage feature is strongly correlated with price”).

## Unsupervised learning

In *unsupervised learning*, as you might guess, the training data is unlabeled. The system tries to learn without a teacher.

For example, say you have a lot of data about your blog's visitors. You may want to run a *clustering* algorithm to try to detect groups of similar visitors (Figure 1-7). The features may include the user's age group, their region, their interests, the duration of their sessions, and so on. At no point do you tell the algorithm which group a visitor belongs to: it finds those connections without your help. For example, it might notice that 40% of your visitors are teenagers who love comic books and generally read your blog after school, while 20% are adults who enjoy sci-fi and who visit during the weekends. If you use a *hierarchical clustering* algorithm, it may also subdivide each group into smaller groups. This may help you target your posts for each group.

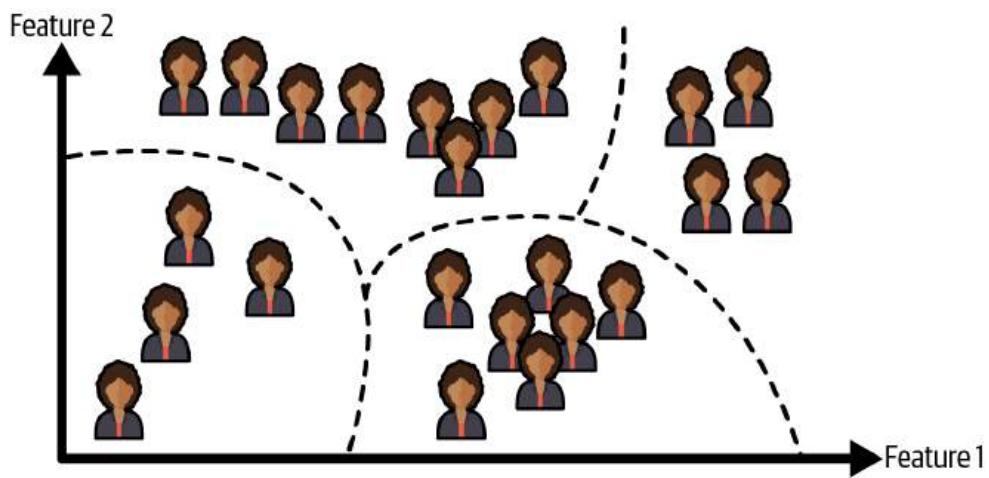


Figure 1-7. Clustering

*Visualization* algorithms are also good examples of unsupervised learning: you feed them a lot of complex and unlabeled data, and they output a 2D or 3D representation of your data that can easily be plotted (Figure 1-8). These algorithms try to preserve as much structure as they can (e.g., trying to keep separate clusters in the input space from overlapping in the visualization) so that you can understand how the data is organized and perhaps identify unsuspected patterns.

A related task is *dimensionality reduction*, in which the goal is to simplify the data without losing too much information. One way to do this is to merge several correlated features into one. For example, a car's mileage may be strongly correlated with its age, so the dimensionality reduction algorithm will merge them into one feature that represents the car's wear and tear. This is called *feature extraction*.

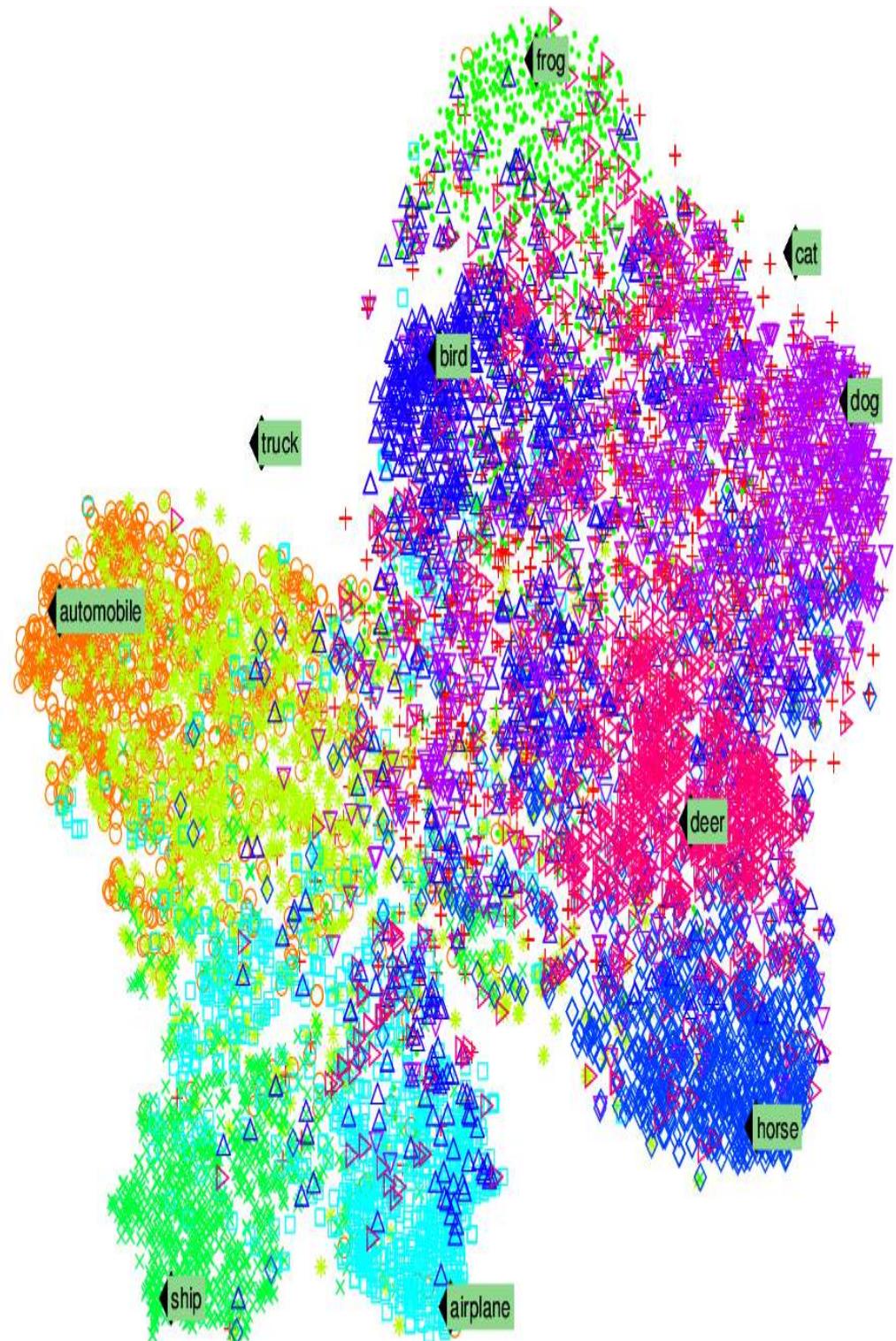
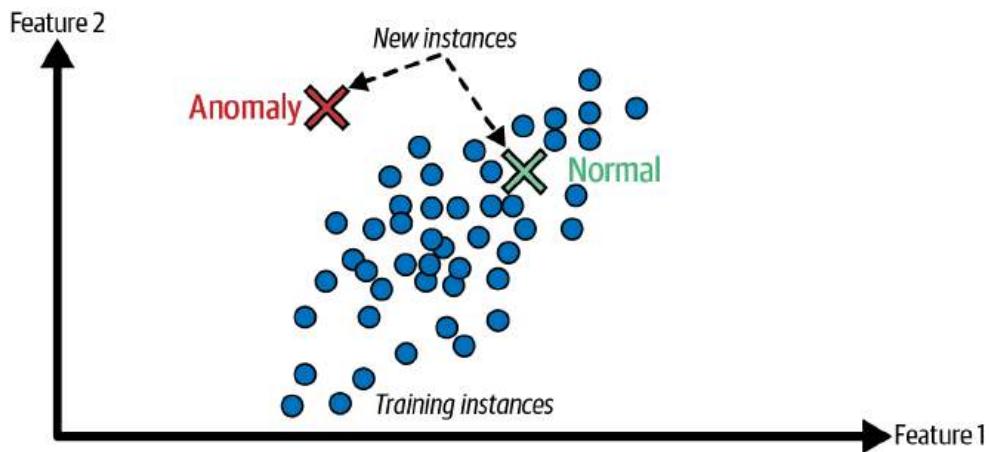


Figure 1-8. Example of a t-SNE visualization highlighting semantic clusters<sup>2</sup>

## TIP

It is often a good idea to try to reduce the number of dimensions in your training data using a dimensionality reduction algorithm before you feed it to another machine learning algorithm (such as a supervised learning algorithm). It will run much faster, the data will take up less disk and memory space, and in some cases it may also perform better.

Yet another important unsupervised task is *anomaly detection*—for example, detecting unusual credit card transactions to prevent fraud, catching manufacturing defects, or automatically removing outliers from a dataset before feeding it to another learning algorithm. The system is shown mostly normal instances during training, so it learns to recognize them; then, when it sees a new instance, it can tell whether it looks like a normal one or whether it is likely an anomaly (see [Figure 1-9](#)). The features may include distance from home, time of day, day of the week, amount withdrawn, merchant category, transaction frequency, etc. A very similar task is *novelty detection*: it aims to detect new instances that look different from all instances in the training set. This requires having a very “clean” training set, devoid of any instance that you would like the algorithm to detect. For example, if you have thousands of pictures of dogs, and 1% of these pictures represent Chihuahuas, then a novelty detection algorithm should not treat new pictures of Chihuahuas as novelties. On the other hand, anomaly detection algorithms may consider these dogs as so rare and so different from other dogs that they would likely classify them as anomalies (no offense to Chihuahuas).



*Figure 1-9. Anomaly detection*

Finally, another common unsupervised task is *association rule learning*, in which the goal is to dig into large amounts of data and discover interesting relations between attributes. For example, suppose you own a supermarket. Running an association rule on your sales logs may reveal that people who purchase barbecue sauce and potato chips also tend to buy steak. Thus, you may want to place these items close to one another.

## Semi-supervised learning

Since labeling data is usually time-consuming and costly, you will often have plenty of unlabeled instances, and few labeled instances. Some algorithms can deal with data that's partially labeled. This is called *semi-supervised learning* (Figure 1-10).

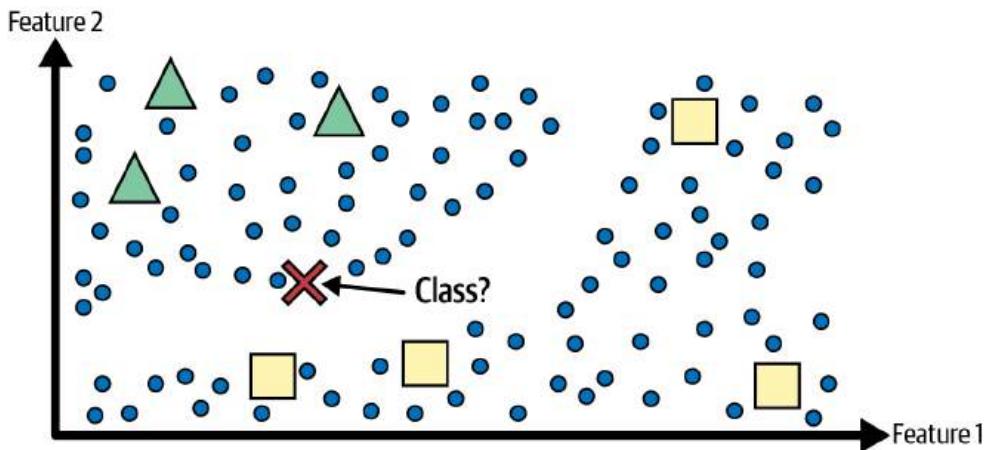


Figure 1-10. Semi-supervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

Some photo-hosting services, such as Google Photos, are good examples of this. Once you upload all your family photos to the service, it automatically recognizes that the same person A shows up in photos 1, 5, and 11, while another person B shows up in photos 2, 5, and 7. This is the unsupervised part of the algorithm (clustering). Now all the system needs is for you to tell it who these people are. Just add one label per person<sup>3</sup> and it is able to name everyone in every photo, which is useful for searching photos.

Most semi-supervised learning algorithms are combinations of unsupervised and supervised algorithms. For example, a clustering algorithm may be used to group similar instances together, and then every unlabeled instance can be labeled with the most common label in its cluster. Once the whole dataset is labeled, it is possible to use any supervised learning algorithm.

## Self-supervised learning

Another approach to machine learning involves actually generating a fully labeled dataset from a fully unlabeled one. Again, once the whole dataset is labeled, any supervised learning algorithm can be used. This approach is called *self-supervised learning*.

For example, if you have a large dataset of unlabeled images, you can randomly mask a small part of each image and then train a model to recover the original image (Figure 1-

11). During training, the masked images are used as the inputs to the model, and the original images are used as the labels.



*Figure 1-11. Self-supervised learning example: input (left) and target (right)*

The resulting model may be quite useful in itself—for example, to repair damaged images or to erase unwanted objects from pictures. But more often than not, a model trained using self-supervised learning is not the final goal. You’ll usually want to tweak and fine-tune the model for a slightly different task—one that you actually care about.

For example, suppose that what you really want is to have a pet classification model: given a picture of any pet, it will tell you what species it belongs to. If you have a large dataset of unlabeled photos of pets, you can start by training an image-repairing model using self-supervised learning. Once it’s performing well, it should be able to distinguish different pet species: when it repairs an image of a cat whose face is masked, it must know not to add a dog’s face. Assuming your model’s architecture allows it (and most neural network architectures do), it is then possible to tweak the model so that it predicts pet species instead of repairing images. The final step consists of fine-tuning the model on a labeled dataset: the model already knows what cats, dogs, and other pet species look like, so this step is only needed so the model can learn the mapping between the species it already knows and the labels we expect from it.

### NOTE

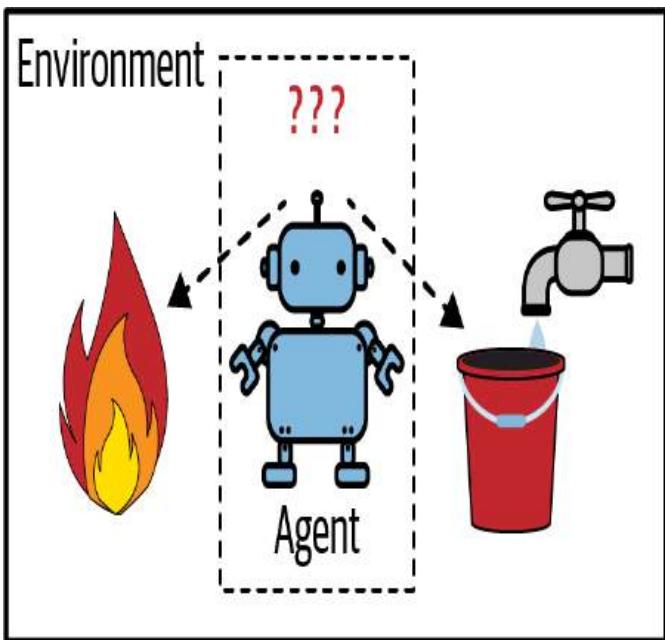
Transferring knowledge from one task to another is called *transfer learning*, and it’s one of the most important techniques in machine learning today, especially when using *deep neural networks* (i.e., neural networks composed of many layers of neurons). We will discuss this in detail in [Part II](#).

As we will see in [Link to Come], large language models (LLMs) are trained in a very similar way, by masking random words in a huge text corpus and training the model to predict the missing words. This large pretrained model can then be fine-tuned for various applications, from sentiment analysis to chatbots.

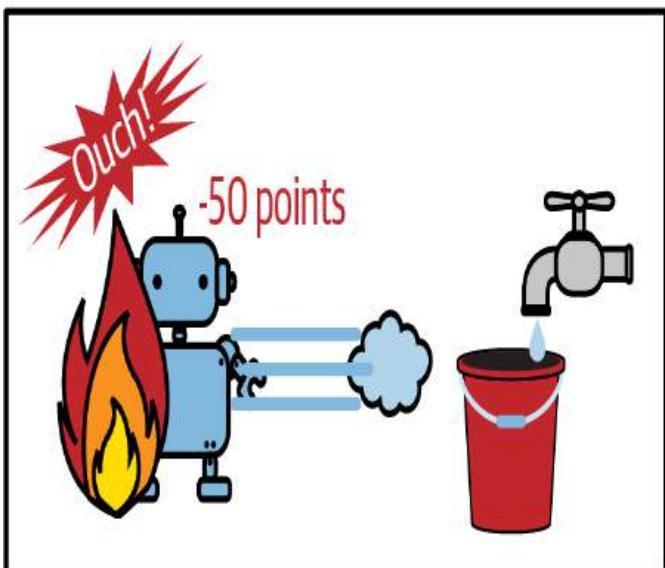
Some people consider self-supervised learning to be a part of unsupervised learning, since it deals with fully unlabeled datasets. But self-supervised learning uses (generated) labels during training, so in that regard it’s closer to supervised learning. And the term “unsupervised learning” is generally used when dealing with tasks like clustering, dimensionality reduction, or anomaly detection, whereas self-supervised learning focuses on the same tasks as supervised learning: mainly classification and regression. In short, it’s best to treat self-supervised learning as its own category.

## Reinforcement learning

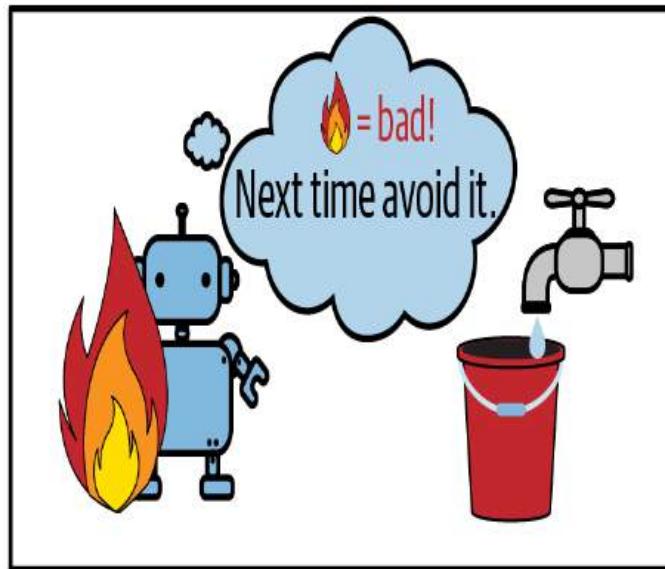
*Reinforcement learning* is a very different beast. The learning system, called an *agent* in this context, can observe the environment, select and perform actions, and get rewards in return (or *penalties* in the form of negative rewards, as shown in [Figure 1-12](#)). It must then learn by itself what is the best strategy, called a *policy*, to get the most reward over time. A policy defines what action the agent should choose when it is in a given situation.



- 1 Observe
- 2 Select action using policy



- 3 Action!
- 4 Get reward or penalty



- 5 Update policy (learning step)
- 6 Iterate until an optimal policy is found

*Figure 1-12. Reinforcement learning*

For example, many robots implement reinforcement learning algorithms to learn how to walk. DeepMind’s AlphaGo program is also a good example of reinforcement learning: it made the headlines in May 2017 when it beat Ke Jie, the number one ranked player in the world at the time, at the game of Go. It learned its winning policy by analyzing millions of games, and then playing many games against itself. Note that learning was turned off during the games against the champion; AlphaGo was just applying the policy it had learned. As you will see in the next section, this is called *offline learning*.

## Batch Versus Online Learning

Another criterion used to classify machine learning systems is whether or not the system can learn incrementally from a stream of incoming data. For example, Random Forests (see [Chapter 6](#)) can only be trained from scratch on the full dataset—this is called batch learning—while other models can be trained one batch of data at a time, for example using *gradient descent* (see [Chapter 4](#))—this is called online learning.

### Batch learning

In *batch learning*, the system must be trained using all the available data. This will generally take a lot of time and computing resources, so it is typically done offline. First the system is trained, and then it is launched into production and runs without learning anymore; it just applies what it has learned. This is called *offline learning*.

Unfortunately, a model’s performance tends to decay slowly over time, simply because the world continues to evolve while the model remains unchanged. This phenomenon is often called *data drift* (or *model rot*). The solution is to regularly retrain the model on up-to-date data. How often you need to do that depends on the use case: if the model classifies pictures of cats and dogs, its performance will decay very slowly, but if the model deals with fast-evolving systems, for example making predictions on the financial market, then it is likely to decay quite fast.

### WARNING

Even a model trained to classify pictures of cats and dogs may need to be retrained regularly, not because cats and dogs will mutate overnight, but because cameras keep changing, along with image formats, sharpness, brightness, and size ratios. Moreover, people may love different breeds next year, or they may decide to dress their pets with tiny hats—who knows?

If you want a batch learning system to know about new data (such as a new type of spam), you need to train a new version of the system from scratch on the full dataset (not just the new data, but also the old data), then replace the old model with the new one. Fortunately, the whole process of training, evaluating, and launching a machine learning system can be automated (as we saw in [Figure 1-3](#)), so even a batch learning system can adapt to change. Simply update the data and train a new version of the system from scratch as often as needed.

This solution is simple and often works fine, but training using the full set of data can take many hours, so you would typically train a new system only every 24 hours or even just weekly. If your system needs to adapt to rapidly changing data (e.g., to predict stock prices), then you need a more reactive solution.

Also, training on the full set of data requires a lot of computing resources (CPU, memory space, disk space, disk I/O, network I/O, etc.). If you have a lot of data and you automate your system to train from scratch every day, it will end up costing you a lot of money. If the amount of data is huge and your system must always be up-to-date, it may even be impossible to use batch learning.

Finally, if your system needs to be able to learn autonomously and it has limited resources (e.g., a smartphone application or a rover on Mars), then carrying around large amounts of training data and taking up a lot of resources to train for hours every day is a showstopper.

A better option in all these scenarios is to use algorithms that are capable of learning incrementally.

## Online learning

In *online learning*, you train the system incrementally by feeding it data instances sequentially, either individually or in small groups called *mini-batches*. Each learning step is fast and cheap, so the system can learn about new data on the fly, as it arrives (see [Figure 1-13](#)). The most common online algorithm by far is gradient descent, but there are a few others.

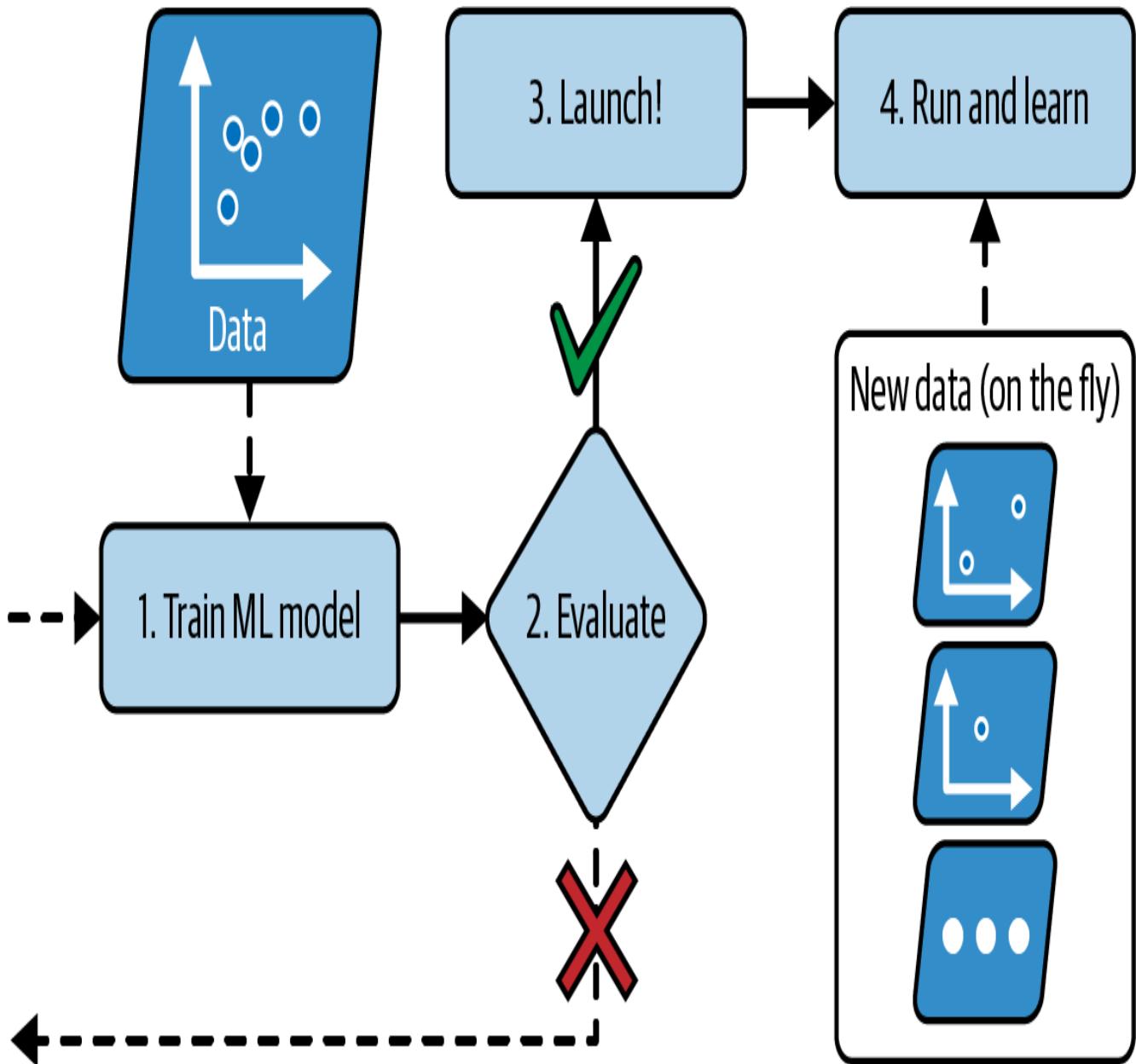


Figure 1-13. In online learning, a model is trained and launched into production, and then it keeps learning as new data comes in

Online learning is useful for systems that need to adapt to change extremely rapidly (e.g., to detect new patterns in the stock market). It is also a good option if you have limited computing resources; for example, if the model is trained on a mobile device.

Most importantly, online learning algorithms can be used to train models on huge datasets that cannot fit in one machine's memory (this is called *out-of-core* learning). The algorithm loads part of the data, runs a training step on that data, and repeats the process until it has run on all of the data (see [Figure 1-14](#)).

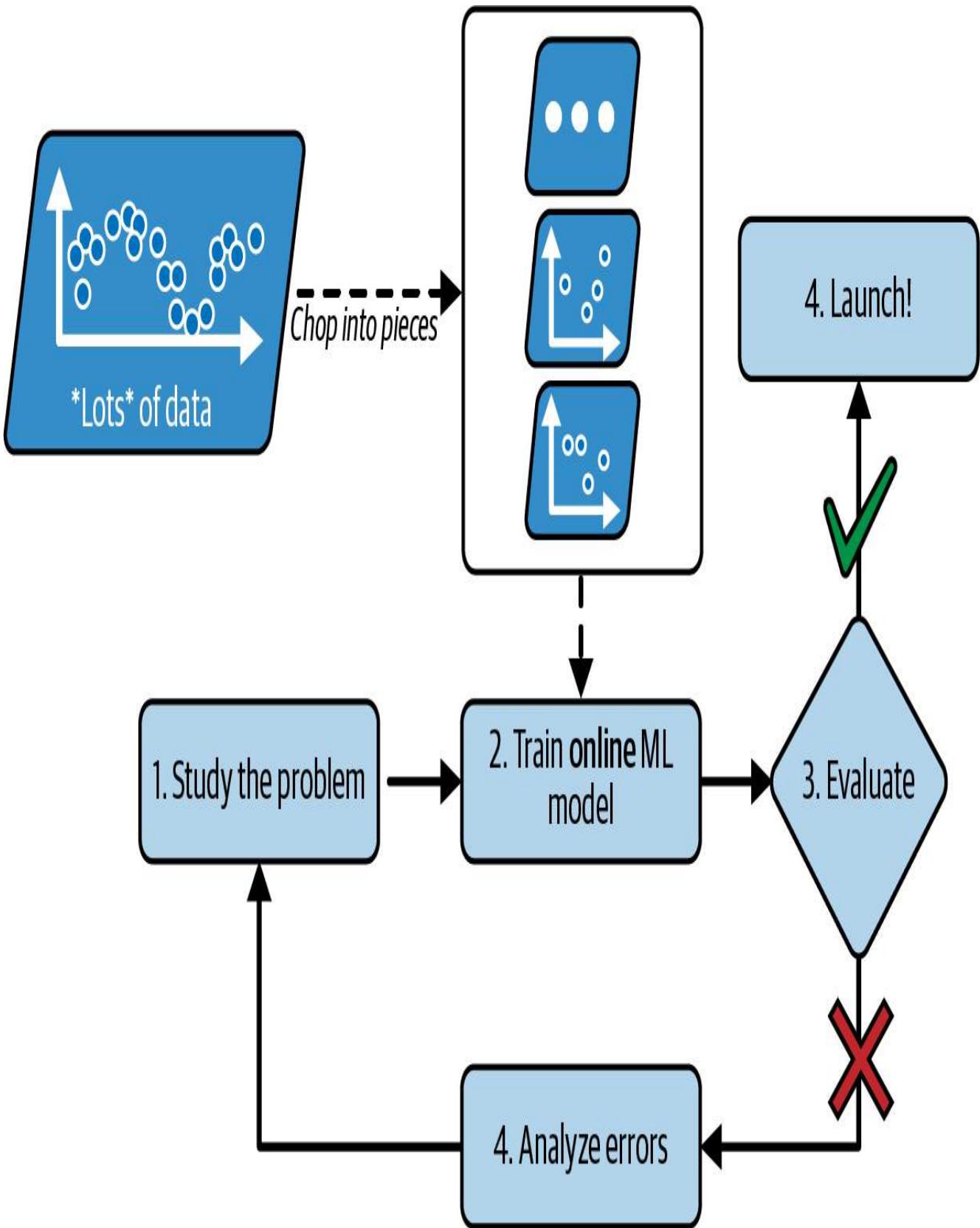


Figure 1-14. Using online learning to handle huge datasets

One important parameter of online learning systems is how fast they should adapt to changing data: this is called the *learning rate*. If you set a high learning rate, then your system will rapidly adapt to new data, but it will also tend to quickly forget the old

data: this is called *catastrophic forgetting* (or *catastrophic interference*). You don't want a spam filter to flag only the latest kinds of spam it was shown! Conversely, if you set a low learning rate, the system will have more inertia; that is, it will learn more slowly, but it will also be less sensitive to noise in the new data or to sequences of nonrepresentative data points (outliers).

### WARNING

Out-of-core learning is usually done offline (i.e., not on the live system), so *online learning* can be a confusing name. Think of it as *incremental learning*. Moreover, mini-batches are often just called "batches", so *batch learning* is also a confusing name. Think of it as learning from scratch on the full dataset.

A big challenge with online learning is that if bad data is fed to the system, the system's performance will decline, possibly quickly (depending on the data quality and learning rate). If it's a live system, your clients will notice. For example, bad data could come from a bug (e.g., a malfunctioning sensor on a robot), or it could come from someone trying to game the system (e.g., spamming a search engine to try to rank high in search results). To reduce this risk, you need to monitor your system closely and promptly switch learning off (and possibly revert to a previously working state) if you detect a drop in performance. You may also want to monitor the input data and react to abnormal data; for example, using an anomaly detection algorithm (see [Chapter 8](#)).

## Instance-Based Versus Model-Based Learning

One more way to categorize machine learning systems is by how they *generalize*. Most machine learning tasks are about making predictions. This means that given a number of training examples, the system needs to be able to make good predictions for (generalize to) examples it has never seen before. Having a good performance measure on the training data is good, but insufficient; the true goal is to perform well on new instances.

There are two main approaches to generalization: instance-based learning and model-based learning.

### Instance-based learning

Possibly the most trivial form of learning is simply to learn by heart. If you were to create a spam filter this way, it would just flag all emails that are identical to emails

that have already been flagged by users—not the worst solution, but certainly not the best.

Instead of just flagging emails that are identical to known spam emails, your spam filter could be programmed to also flag emails that are very similar to known spam emails. This requires a *measure of similarity* between two emails. A (very basic) similarity measure between two emails could be to count the number of words they have in common. The system would flag an email as spam if it has many words in common with a known spam email.

This is called *instance-based learning*: the system learns the examples by heart, then generalizes to new cases by using a similarity measure to compare them to the learned examples (or a subset of them). For example, in [Figure 1-15](#) the new instance would be classified as a triangle because the majority of the most similar instances belong to that class.

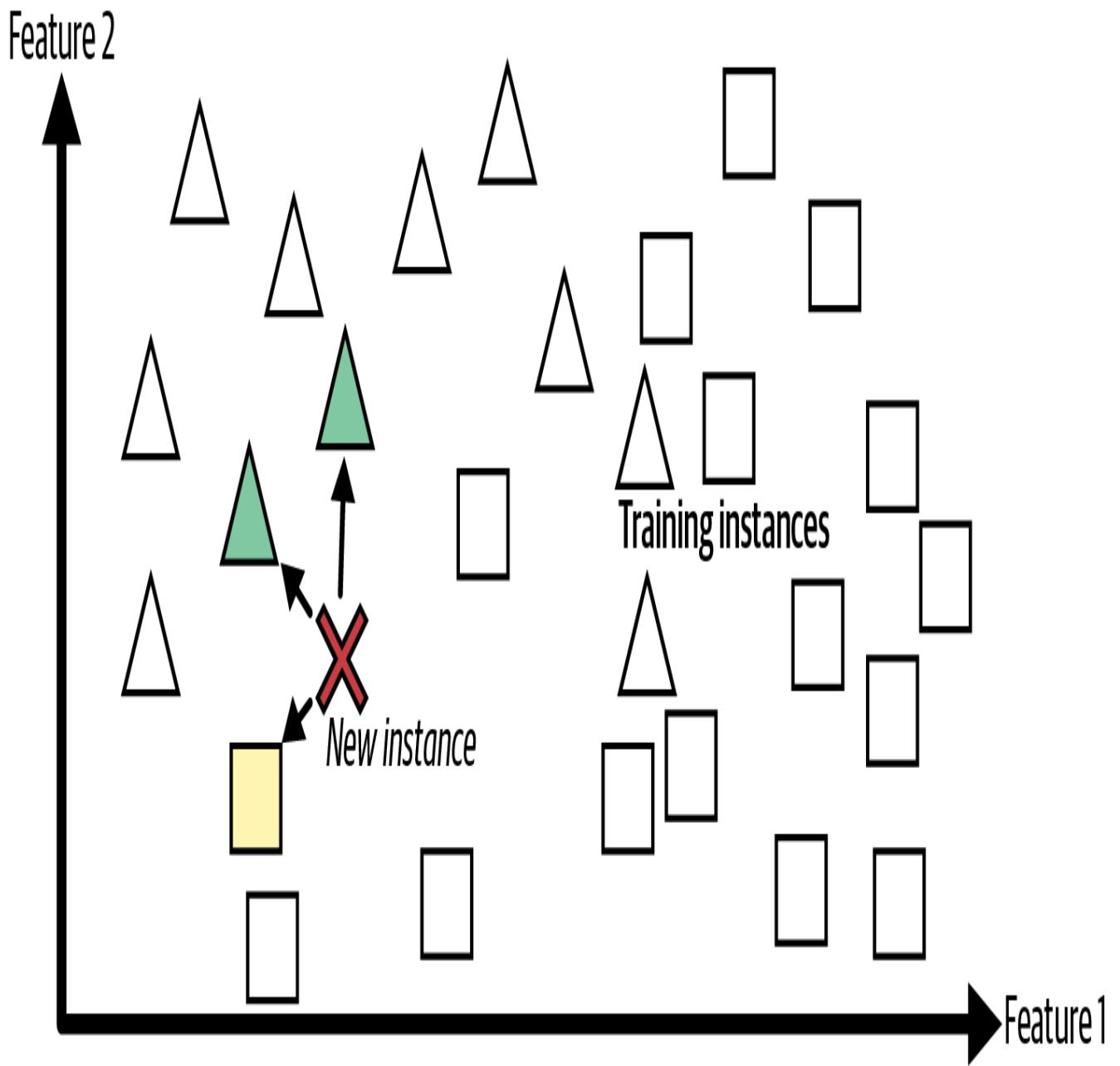


Figure 1-15. Instance-based learning: in this example we consider the class of the 3 nearest neighbors in the training set

Instance-based learning often shines with small datasets, especially if the data keeps changing, but it does not scale very well: it requires deploying a whole copy of the training set to production, making predictions requires searching for similar instances, which can be quite slow, and it doesn't work well with high-dimensional data such as images.

### Model-based learning and a typical machine learning workflow

Another way to generalize from a set of examples is to build a model of these examples and then use that model to make *predictions*. This is called *model-based learning*

(Figure 1-16).

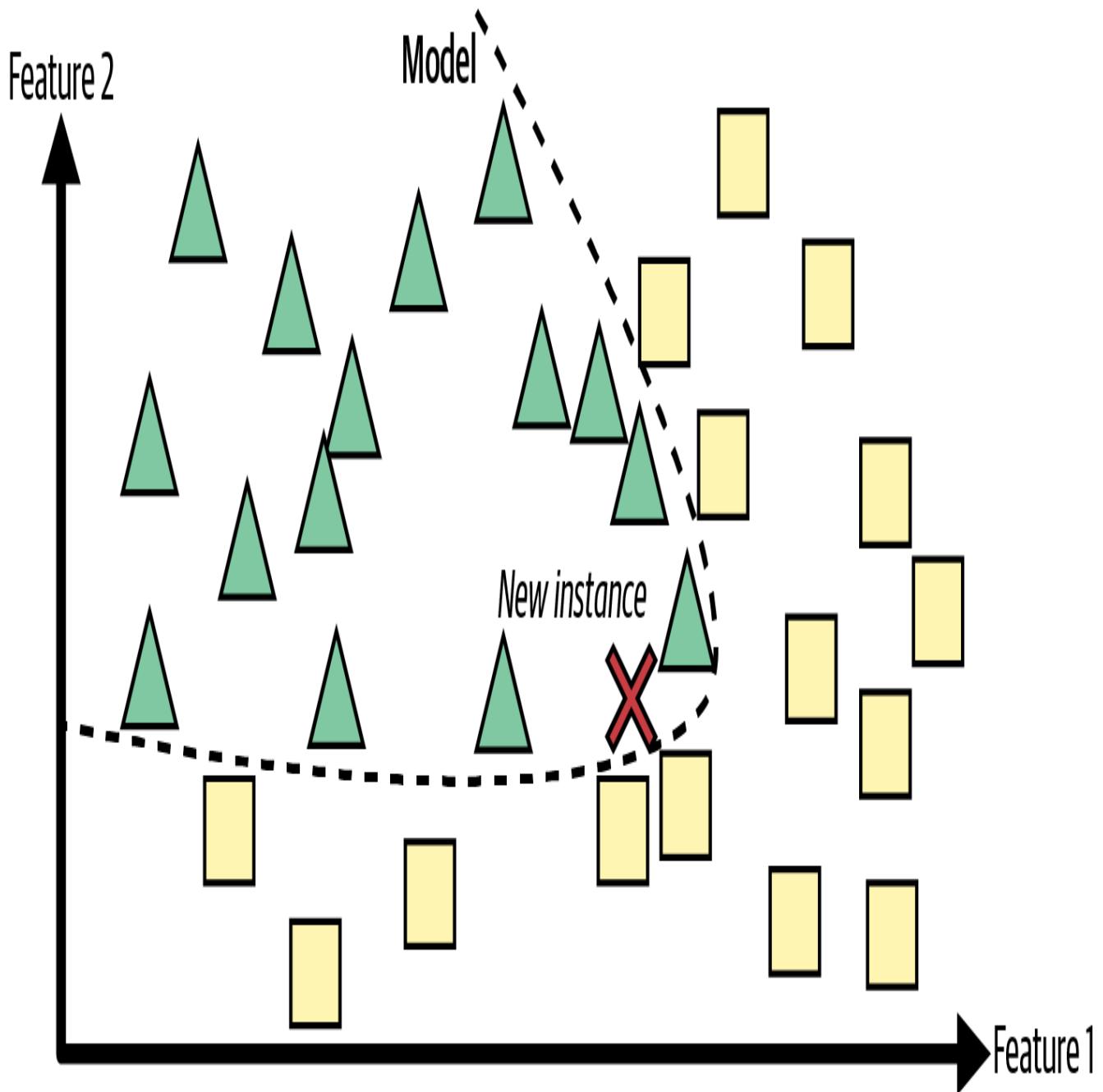


Figure 1-16. Model-based learning

For example, suppose you want to know if money makes people happy, so you download the Better Life Index data from the [OECD's website](#) and [World Bank](#) stats about gross domestic product (GDP) per capita. Then you join the tables and sort by GDP per capita. [Table 1-1](#) shows an excerpt of what you get.

*Table 1-1. Does money make people happier?*

<b>Country</b>	<b>GDP per capita (USD)</b>	<b>Life satisfaction</b>
Turkey	28,384	5.5
Hungary	31,008	5.6
France	42,026	6.5
United States	60,236	6.9
New Zealand	42,404	7.3
Australia	48,698	7.3
Denmark	55,938	7.6

Let's plot the data for these countries ([Figure 1-17](#)).

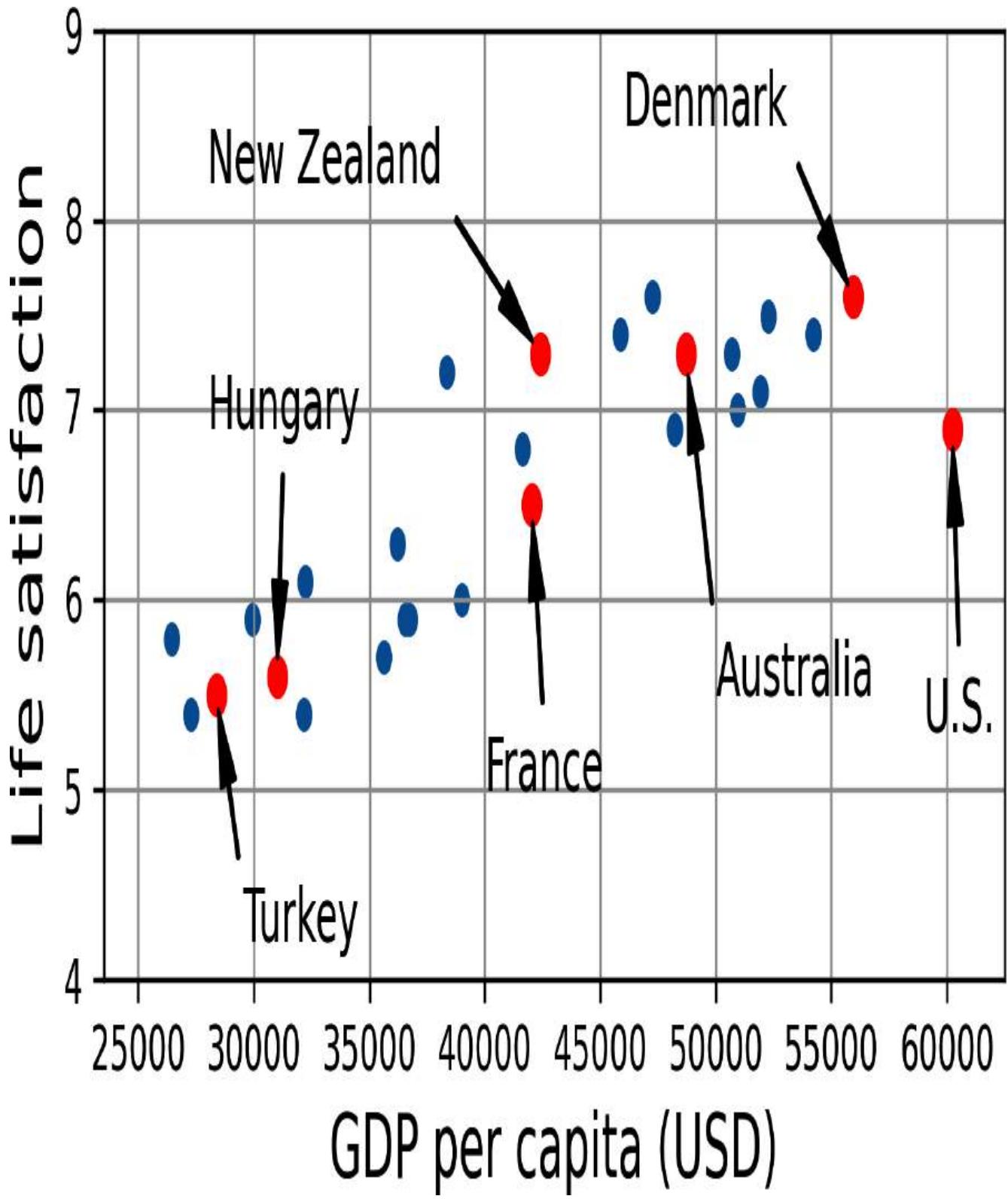


Figure 1-17. Do you see a trend here?

There does seem to be a trend here! Although the data is *noisy* (i.e., partly random), it looks like life satisfaction goes up more or less linearly as the country's GDP per capita increases. So you decide to model life satisfaction as a linear function of GDP per capita (you assume that any deviation from that line is just random noise). This step is

called *model selection*: you selected a *linear model* of life satisfaction with just one attribute, GDP per capita ([Equation 1-1](#)).

*Equation 1-1. A simple linear model*

$$\text{life\_satisfaction} = \theta_0 + \theta_1 \times \text{GDP\_per\_capita}$$

This model has two *model parameters*,  $\theta_0$  and  $\theta_1$ .<sup>4</sup> By tweaking these parameters, you can make your model represent any linear function, as shown in [Figure 1-18](#).

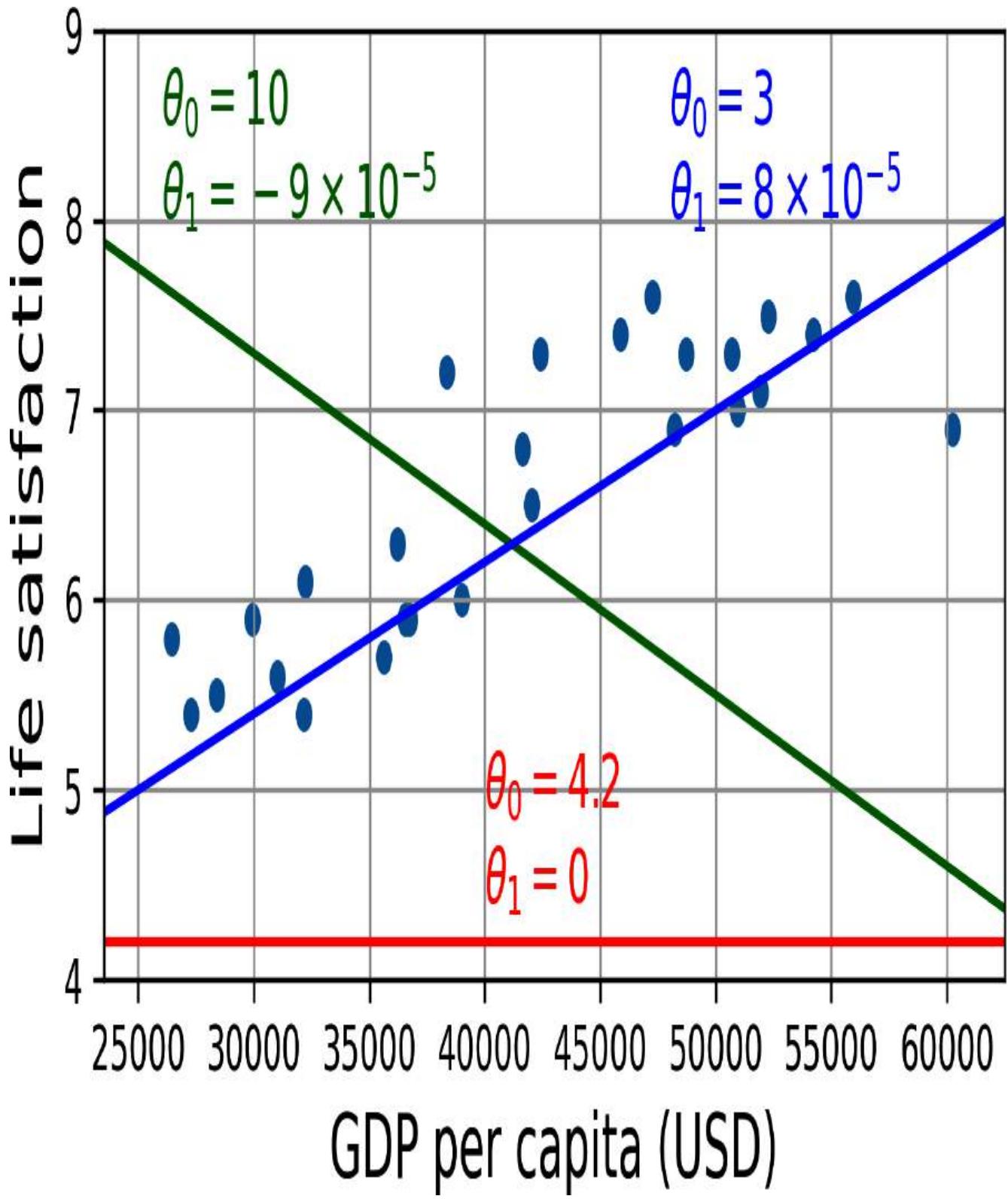


Figure 1-18. A few possible linear models

Before you can use your model, you need to define the parameter values  $\theta_0$  and  $\theta_1$ . How can you know which values will make your model perform best? To answer this question, you need to specify a performance measure. You can either define a *utility function* (or *fitness function*) that measures how *good* your model is, or you can define

a *cost function* (a.k.a., *loss function*) that measures how *bad* it is. For linear regression problems, people typically use a cost function that measures the distance between the linear model’s predictions and the training examples; the objective is to minimize this distance.

This is where the linear regression algorithm comes in: you feed it your training examples, and it finds the parameters that make the linear model fit best to your data. This is called *training* the model. In our case, the algorithm finds that the optimal parameter values are  $\theta_0 = 3.75$  and  $\theta_1 = 6.78 \times 10^{-5}$ .

### WARNING

Confusingly, the word “model” can refer to a *type of model* (e.g., linear regression), to a *fully specified model architecture* (e.g., linear regression with one input and one output), or to the *final trained model* ready to be used for predictions (e.g., linear regression with one input and one output, using  $\theta_0 = 3.75$  and  $\theta_1 = 6.78 \times 10^{-5}$ ). Model selection consists in choosing the type of model and fully specifying its architecture. Training a model means running an algorithm to find the model parameters that will make it best fit the training data, and hopefully make good predictions on new data.

Now the model fits the training data as closely as possible (for a linear model), as you can see in [Figure 1-19](#).

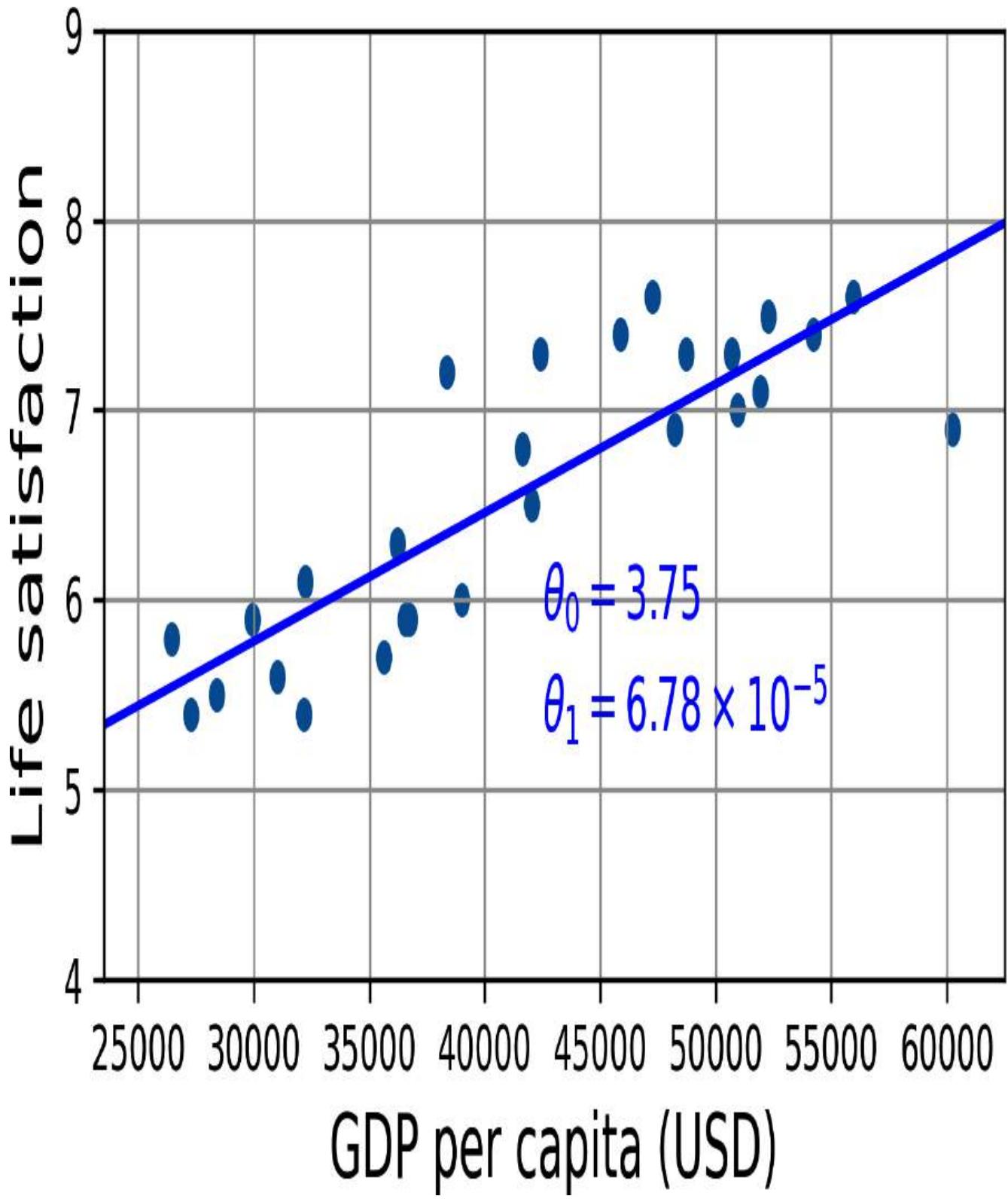


Figure 1-19. The linear model that fits the training data best

You are finally ready to run the model to make predictions. For example, say you want to know how happy Puerto Ricans are, and the OECD data does not have the answer. Fortunately, you can use your model to make a good prediction: you look up Puerto

Rico's GDP per capita, find \$33,442, and then apply your model and find that life satisfaction is likely to be somewhere around  $3.75 + 33,442 \times 6.78 \times 10^{-5} = 6.02$ .

To whet your appetite, **Example 1-1** shows the Python code that loads the data, separates the inputs X from the labels y, creates a scatterplot for visualization, and then trains a linear model and makes a prediction.<sup>5</sup>

*Example 1-1. Training and running a linear model using Scikit-Learn*

---

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression

# Download and prepare the data
data_root = "https://github.com/ageron/data/raw/main/"
lifesat = pd.read_csv(data_root + "lifesat/lifesat.csv")
X = lifesat[["GDP per capita (USD)"]].values
y = lifesat[["Life satisfaction"]].values

# Visualize the data
lifesat.plot(kind='scatter', grid=True,
             x="GDP per capita (USD)", y="Life satisfaction")
plt.axis([23_500, 62_500, 4, 9])
plt.show()

# Select a linear model
model = LinearRegression()

# Train the model
model.fit(X, y)

# Make a prediction for Puerto Rico
X_new = [[33_442.8]] # Puerto Rico' GDP per capita in 2020
print(model.predict(X_new)) # outputs [[6.01610329]]
```

## NOTE

If you had used an instance-based learning algorithm instead, you would have found that Poland has the closest GDP per capita to that of Puerto Rico (\$32,238), and since the OECD data tells us that Poles' life satisfaction is 6.1, you would have predicted a life satisfaction of 6.1 as well for Puerto Rico. If you zoom out a bit and look at the next two closest countries, you will find Portugal with a life satisfaction of 5.4, and Estonia with a life satisfaction of 5.7. Averaging these three values, you get 5.73, which is a bit below your model-based prediction. This simple algorithm is called *k-nearest neighbors* regression (in this example,  $k = 3$ ).

Replacing the linear regression model with *k*-nearest neighbors regression in the previous code is as easy as replacing these lines:

```
from sklearn.linear_model import LinearRegression  
model = LinearRegression()
```

with these two:

```
from sklearn.neighbors import KNeighborsRegressor  
model = KNeighborsRegressor(n_neighbors=3)
```

If all went well, your model will make good predictions. If not, you may need to use more attributes (employment rate, health, air pollution, etc.), get more or better-quality training data, or perhaps select a more powerful model (e.g., a polynomial regression model).

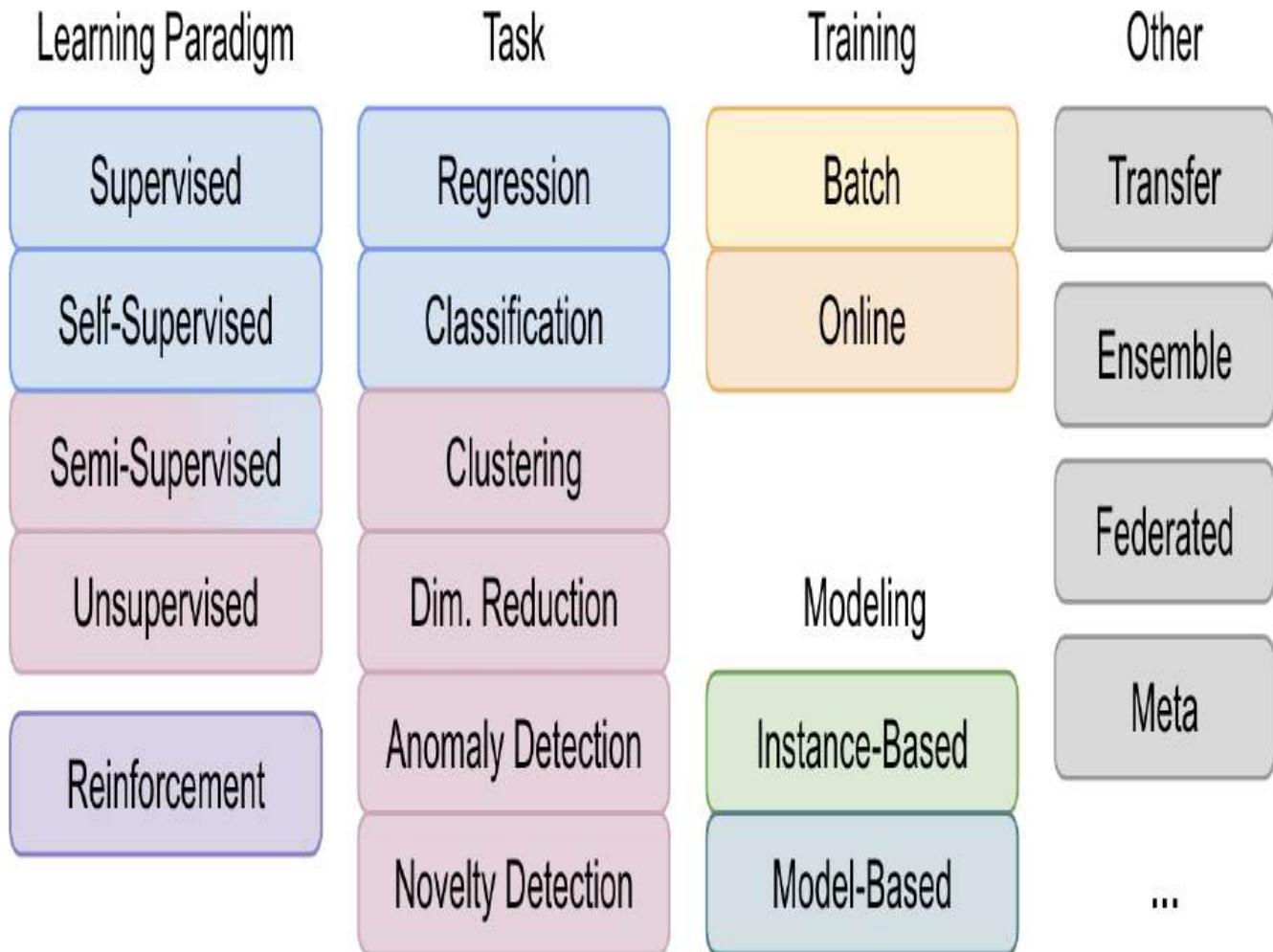
In summary:

- You studied the data.
- You selected a model.
- You trained it on the training data (i.e., the learning algorithm searched for the model parameter values that minimize a cost function).
- Finally, you applied the model to make predictions on new cases (this is called *inference*), hoping that this model will generalize well.

This is what a typical machine learning project looks like. In [Chapter 2](#) you will experience this firsthand by going through a project end to end.

We discussed quite a few categories of ML systems, but this field has more! For example, *ensemble learning* involves training multiple models and combining their individual predictions into improved predictions (see [Chapter 6](#)); *federated learning* is

a decentralized approach where models are trained across multiple devices (e.g., smartphones) and adapted to each user without exchanging raw data, thereby protecting the user's privacy; *meta-learning* is a learning-to-learn approach where models learn how to learn new tasks quickly with minimal data. And the list goes on! **Figure 1-20** summarizes the various classifications of ML systems we have discussed so far.



*Figure 1-20. Overview of ML categories*

We have covered a lot of ground so far: you now know what machine learning is really about, why it is useful, what some of the most common categories of ML systems are, and what a typical project workflow looks like. Now let's look at what can go wrong in learning and prevent you from making accurate predictions.

## Main Challenges of Machine Learning

In short, since your main task is to select a model and train it on some data, the two things that can go wrong are “bad model” and “bad data”. Let's start with examples of bad data.

## **Insufficient Quantity of Training Data**

For a toddler to learn what an apple is, all it takes is for you to point to an apple and say “apple” (possibly repeating this procedure a few times). Now the child is able to recognize apples in all sorts of colors and shapes. Genius.

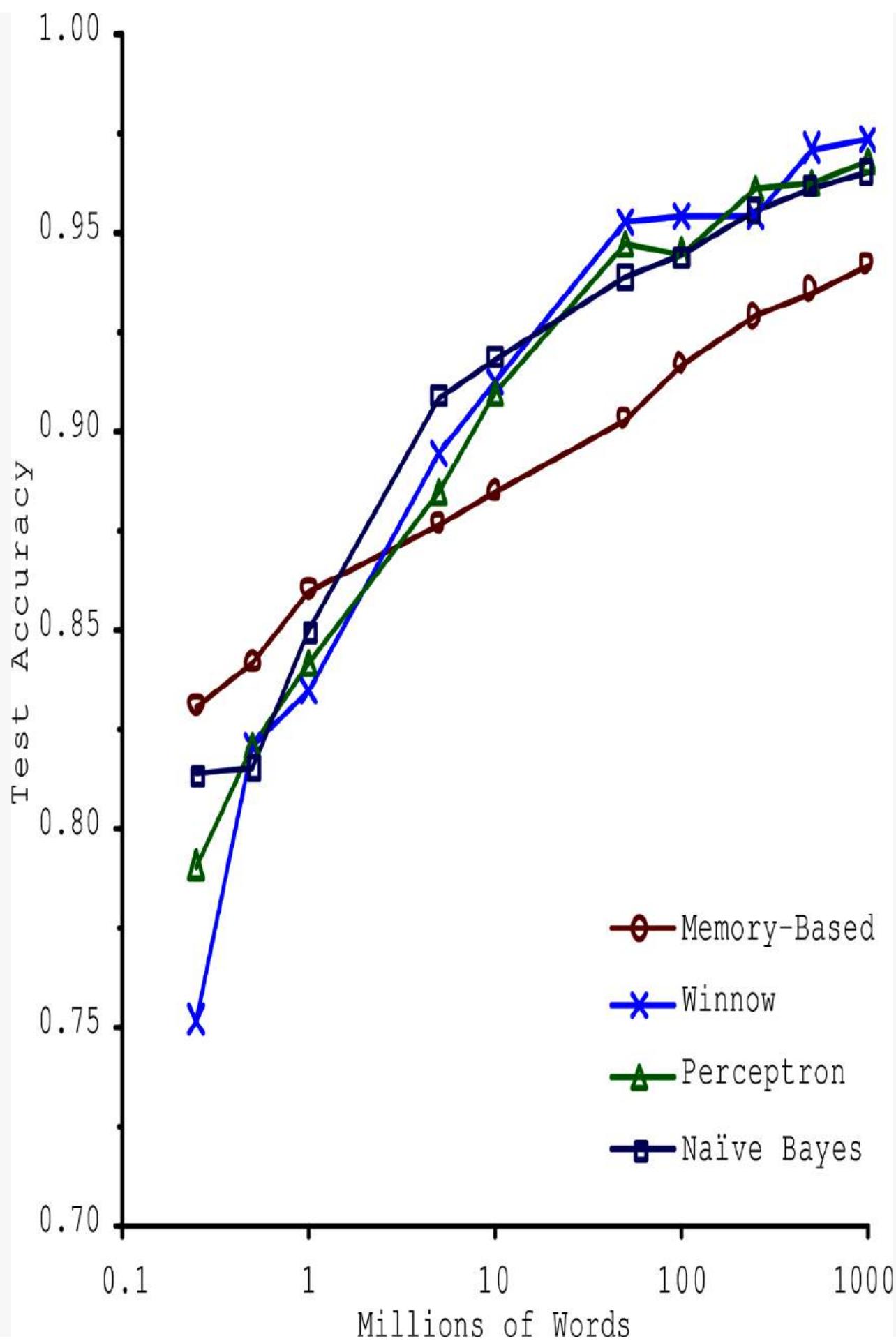
Machine learning is not quite there yet; it takes a lot of data for most machine learning algorithms to work properly. Even for very simple problems you typically need thousands of examples, and for complex problems such as image or speech recognition you may need millions of examples (unless you can reuse parts of an existing model, i.e., transfer learning).

## THE UNREASONABLE EFFECTIVENESS OF DATA

In a [famous paper](#) published in 2001, Microsoft researchers Michele Banko and Eric Brill showed that very different machine learning algorithms, including fairly simple ones, performed almost identically well on a complex problem of natural language disambiguation<sup>6</sup> once they were given enough data (as you can see in Figure 1-21).

As the authors put it, “these results suggest that we may want to reconsider the trade-off between spending time and money on algorithm development versus spending it on corpus development”.

The idea that data matters more than algorithms for complex problems was further popularized by Peter Norvig et al. in a paper titled “[The Unreasonable Effectiveness of Data](#)”, published in 2009.<sup>7</sup> It should be noted, however, that small and medium-sized datasets are still very common, and it is not always easy or cheap to get extra training data—so don’t abandon algorithms just yet.



## Nonrepresentative Training Data

In order to generalize well, it is crucial that your training data be representative of the new cases you want to generalize to. This is true whether you use instance-based learning or model-based learning.

For example, the set of countries you used earlier for training the linear model was not perfectly representative; it did not contain any country with a GDP per capita lower than \$23,500 or higher than \$62,500. [Figure 1-22](#) shows what the data looks like when you add such countries.

If you train a linear model on this data, you get the solid line, while the old model is represented by the dotted line. As you can see, not only does adding a few missing countries significantly alter the model, but it makes it clear that such a simple linear model is probably never going to work well. It seems that very rich countries are not happier than moderately rich countries (in fact, they seem slightly unhappier!), and conversely some poor countries seem happier than many rich countries.

By using a nonrepresentative training set, you trained a model that is unlikely to make accurate predictions, especially for very poor and very rich countries.

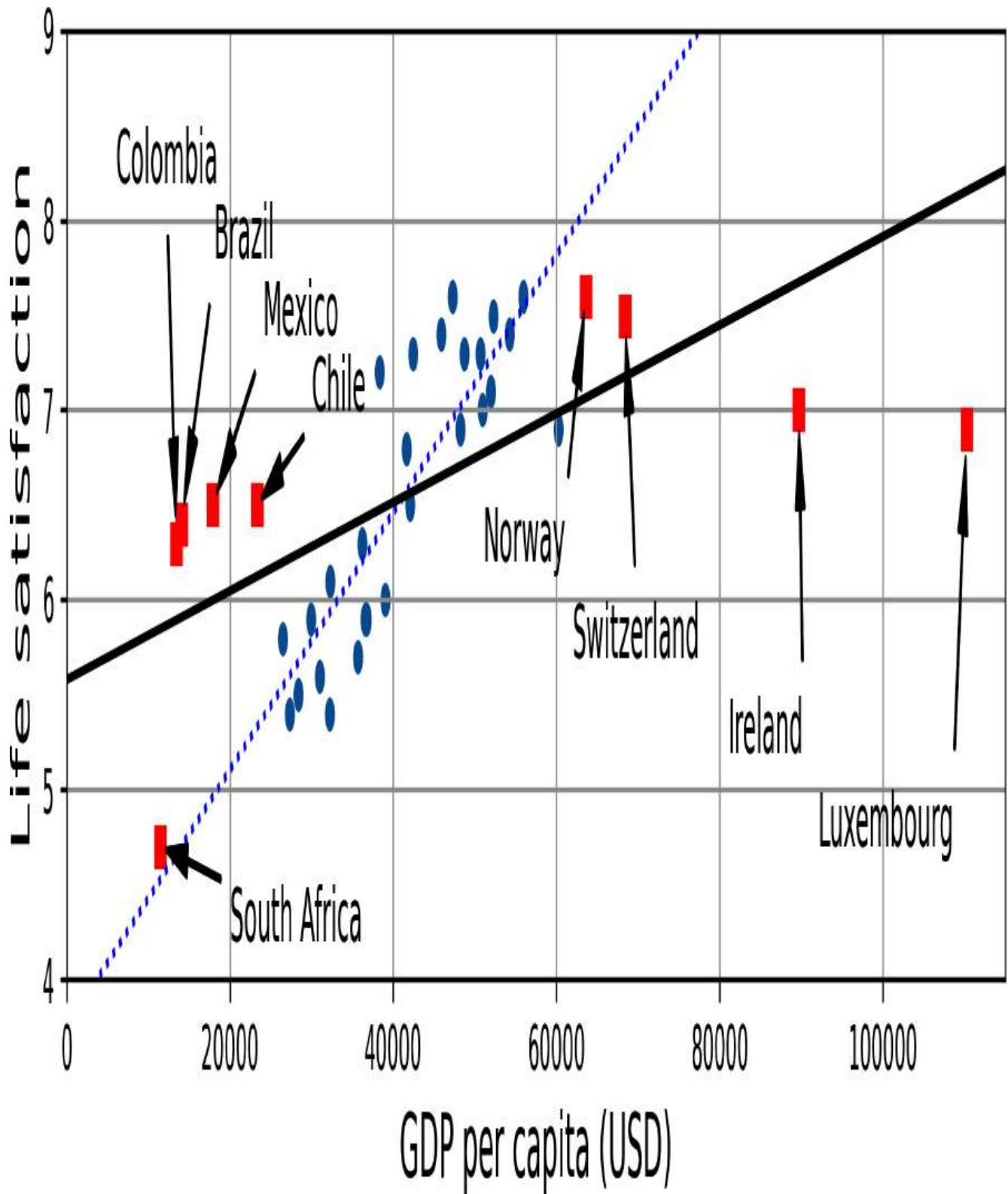


Figure 1-22. A more representative training sample

It is crucial to use a training set that is representative of the cases you want to generalize to. This is often harder than it sounds: if the sample is too small, you will have *sampling noise* (i.e., nonrepresentative data as a result of chance), but even very large

samples can be nonrepresentative if the sampling method is flawed. This is called *sampling bias*.

## EXAMPLES OF SAMPLING BIAS

Perhaps the most famous example of sampling bias happened during the US presidential election in 1936, which pitted Landon against Roosevelt: the *Literary Digest* conducted a very large poll, sending mail to about 10 million people. It got 2.4 million answers, and predicted with high confidence that Landon would get 57% of the votes. Instead, Roosevelt won with 62% of the votes. The flaw was in the *Literary Digest*'s sampling method:

- First, to obtain the addresses to send the polls to, the *Literary Digest* used telephone directories, lists of magazine subscribers, club membership lists, and the like. All of these lists tended to favor wealthier people, who were more likely to vote Republican (hence Landon).
- Second, less than 25% of the people who were polled answered. Again this introduced a sampling bias, by potentially ruling out people who didn't care much about politics, people who didn't like the *Literary Digest*, and other key groups. This is a special type of sampling bias called *nonresponse bias*.

Here is another example: say you want to build a system to recognize funk music videos. One way to build your training set is to search for “funk music” on YouTube and use the resulting videos. But this assumes that YouTube’s search engine returns a set of videos that are representative of all the funk music videos on YouTube. In reality, the search results are likely to be biased toward popular artists (and if you live in Brazil you will get a lot of “funk carioca” videos, which sound nothing like James Brown). On the other hand, how else can you get a large training set?

## Poor-Quality Data

Obviously, if your training data is full of errors, outliers, and noise (e.g., due to poor-quality measurements), it will make it harder for the system to detect the underlying patterns, so your system is less likely to perform well. It is often well worth the effort to spend time cleaning up your training data. The truth is, most data scientists spend a significant part of their time doing just that. The following are a couple examples of when you’d want to clean up training data:

- If some instances are clearly outliers, it may help to simply discard them or try to fix the errors manually.
- If some instances are missing a few features (e.g., 5% of your customers did not specify their age), you must decide whether you want to ignore this attribute altogether, ignore these instances, fill in the missing values (e.g., with the median age), or train one model with the feature and one model without it.

## Irrelevant Features

As the saying goes: garbage in, garbage out. Your system will only be capable of learning if the training data contains enough relevant features and not too many irrelevant ones. A critical part of the success of a machine learning project is coming up with a good set of features to train on. This process, called *feature engineering*, involves the following steps:

- *Feature selection* (selecting the most useful features to train on among existing features)
- *Feature extraction* (combining existing features to produce a more useful one—as we saw earlier, dimensionality reduction algorithms can help)
- Creating new features by gathering new data

Now that we have looked at many examples of bad data, let's look at a couple examples of bad algorithms.

## Overfitting the Training Data

Say you are visiting a foreign country and the taxi driver rips you off. You might be tempted to say that *all* taxi drivers in that country are thieves. Overgeneralizing is something that we humans do all too often, and unfortunately machines can fall into the same trap if we are not careful. In machine learning this is called *overfitting*: it means that the model performs well on the training data, but it does not generalize well.

[Figure 1-23](#) shows an example of a high-degree polynomial life satisfaction model that strongly overfits the training data. Even though it performs much better on the training data than the simple linear model, would you really trust its predictions?

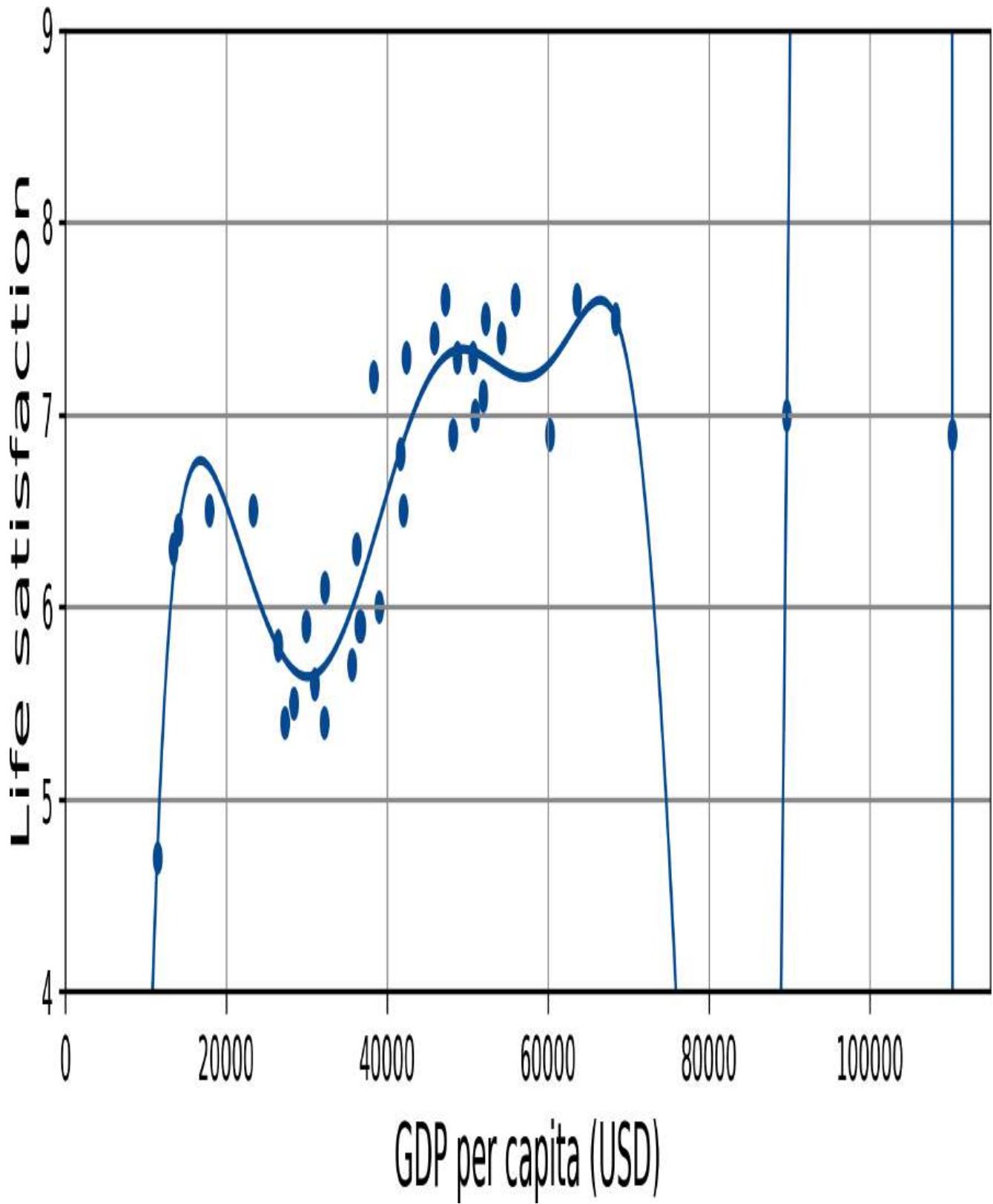


Figure 1-23. Overfitting the training data

Complex models such as deep neural networks can detect subtle patterns in the data, but if the training set is noisy, or if it is too small, which introduces sampling noise, then the model is likely to detect patterns in the noise itself (as in the taxi driver example). Obviously these patterns will not generalize to new instances. For example, say you

feed your life satisfaction model many more attributes, including uninformative ones such as the country's name. In that case, a complex model may detect patterns like the fact that all countries in the training data with a  $w$  in their name have a life satisfaction greater than 7: New Zealand (7.3), Norway (7.6), Sweden (7.3), and Switzerland (7.5). How confident are you that the  $w$ -satisfaction rule generalizes to Rwanda or Zimbabwe? Obviously this pattern occurred in the training data by pure chance, but the model has no way to tell whether a pattern is real or simply the result of noise in the data.

## WARNING

Overfitting happens when the model is too complex relative to the amount and noisiness of the training data, so it starts to learn random patterns in the training data. Here are possible solutions:

- Simplify the model by selecting one with fewer parameters (e.g., a linear model rather than a high-degree polynomial model), by reducing the number of attributes in the training data, or by constraining the model.
- Gather more training data.
- Reduce the noise in the training data (e.g., fix data errors and remove outliers).

Constraining a model to make it simpler and reduce the risk of overfitting is called *regularization*. For example, the linear model we defined earlier has two parameters,  $\theta_0$  and  $\theta_1$ . This gives the learning algorithm two *degrees of freedom* to adapt the model to the training data: it can tweak both the height ( $\theta_0$ ) and the slope ( $\theta_1$ ) of the line. If we forced  $\theta_1 = 0$ , the algorithm would have only one degree of freedom and would have a much harder time fitting the data properly: all it could do is move the line up or down to get as close as possible to the training instances, so it would end up around the mean. A very simple model indeed! If we allow the algorithm to modify  $\theta_1$  but we force it to keep it small, then the learning algorithm will effectively have somewhere in between one and two degrees of freedom. It will produce a model that's simpler than one with two degrees of freedom, but more complex than one with just one. You want to find the right balance between fitting the training data perfectly and keeping the model simple enough to ensure that it will generalize well.

Figure 1-24 shows three models. The dotted line represents the original model that was trained on the countries represented as circles (without the countries represented as squares), the solid line is our second model trained with all countries (circles and squares), and the dashed line is a model trained with the same data as the first model

but with a regularization constraint. You can see that regularization forced the model to have a smaller slope: this model does not fit the training data (circles) as well as the first model, but it actually generalizes better to new examples that it did not see during training (squares).

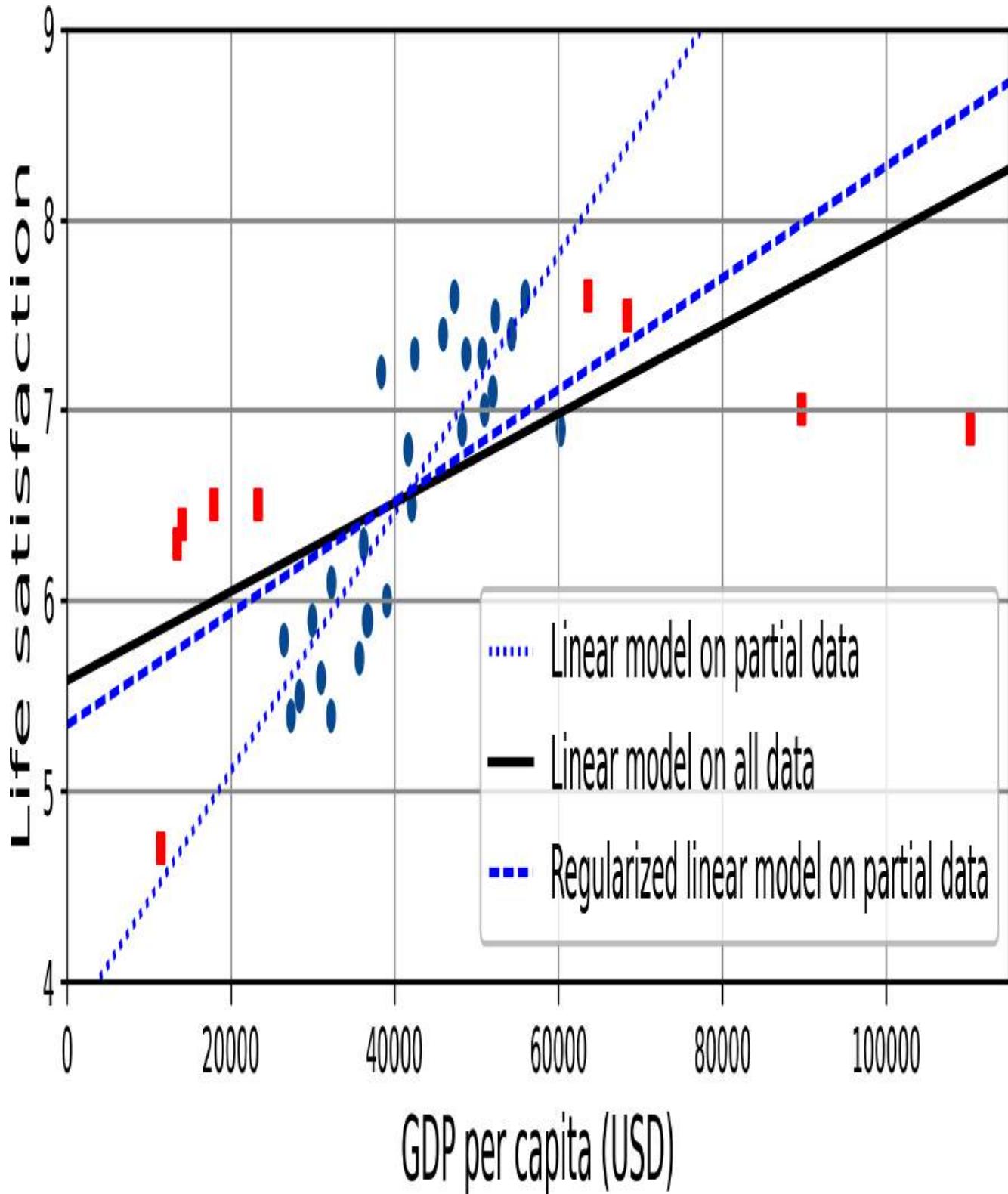


Figure 1-24. Regularization reduces the risk of overfitting

The amount of regularization to apply during learning can be controlled by a *hyperparameter*. A hyperparameter is a parameter of a learning algorithm (not of the model). As such, it is not affected by the learning algorithm itself; it must be set prior to training and remains constant during training. If you set the regularization hyperparameter to a very large value, you will get an almost flat model (a slope close to zero); the learning algorithm will almost certainly not overfit the training data, but it will be less likely to find a good solution. Tuning hyperparameters is an important part of building a machine learning system (you will see a detailed example in the next chapter).

## Underfitting the Training Data

As you might guess, *underfitting* is the opposite of overfitting: it occurs when your model is too simple to learn the underlying structure of the data. For example, a linear model of life satisfaction is prone to underfit; reality is just more complex than the model, so its predictions are bound to be inaccurate, even on the training examples.

Here are the main options for fixing this problem:

- Select a more powerful model, with more parameters.
- Feed better features to the learning algorithm (feature engineering).
- Reduce the constraints on the model (for example by reducing the regularization hyperparameter).

## Deployment Issues

Even if you have a large and clean dataset and you manage to train a beautiful model that neither underfits nor overfits the data, you may still run into issues during deployment: for example, the model may be too complex to maintain, or too large to fit in memory, or too slow, or it may not scale properly, it may have security vulnerabilities, it may become outdated if you don't update it often enough, etc.

In short, there's more to an ML project than just data and models. However, the skillset required to handle these operational problems are fairly different from those required for data modeling, which is why companies often have a dedicated MLOps team (ML operations) to handle this.

## Stepping Back

By now you know a lot about machine learning. However, we went through so many concepts that you may be feeling a little lost, so let's step back and look at the big picture:

- Machine learning is about making machines get better at some task by learning from data, instead of having to explicitly code rules.
- There are many different types of ML systems: supervised or not, batch or online, instance-based or model-based.
- In an ML project you gather data in a training set, and you feed the training set to a learning algorithm. If the algorithm is model-based, it tunes some parameters to fit the model to the training set (i.e., to make good predictions on the training set itself), and then hopefully it will be able to make good predictions on new cases as well. If the algorithm is instance-based, it just learns the examples by heart and generalizes to new instances by using a similarity measure to compare them to the learned instances.
- The system will not perform well if your training set is too small, or if the data is not representative, is noisy, or is polluted with irrelevant features (garbage in, garbage out). Your model needs to be neither too simple (in which case it will underfit) nor too complex (in which case it will overfit). Lastly, you must think carefully about deployment constraints.

There's just one last important topic to cover: once you have trained a model, you don't want to just "hope" it generalizes to new cases. You want to evaluate it and fine-tune it if necessary. Let's see how to do that.

## Testing and Validating

The only way to know how well a model will generalize to new cases is to actually try it out on new cases. One way to do that is to put your model in production and monitor how well it performs. This works well, but if your model is horribly bad, your users will complain—not the best idea.

A better option is to split your data into two sets: the *training set* and the *test set*. As these names imply, you train your model using the training set, and you test it using the test set. The error rate on new cases is called the *generalization error* (or *out-of-sample error*), and by evaluating your model on the test set, you get an estimate of this

error. This value tells you how well your model will perform on instances it has never seen before.

If the training error is low (i.e., your model makes few mistakes on the training set) but the generalization error is high, it means that your model is overfitting the training data.

### TIP

It is common to use 80% of the data for training and *hold out* 20% for testing. However, this depends on the size of the dataset: if it contains 10 million instances, then holding out 1% means your test set will contain 100,000 instances, probably more than enough to get a good estimate of the generalization error.

## Hyperparameter Tuning and Model Selection

Evaluating a model is simple enough: just use a test set. But suppose you are hesitating between two types of models (say, a linear model and a polynomial model): how can you decide between them? One option is to train both and compare how well they generalize using the test set.

Now suppose that the linear model generalizes better, but you want to apply some regularization to avoid overfitting. The question is, how do you choose the value of the regularization hyperparameter? One option is to train 100 different models using 100 different values for this hyperparameter. Suppose you find the best hyperparameter value that produces a model with the lowest generalization error—say, just 5% error. You launch this model into production, but unfortunately it does not perform as well as expected and produces 15% errors. What just happened?

The problem is that you measured the generalization error multiple times on the test set, and you adapted the model and hyperparameters to produce the best model *for that particular set*. This means the model is unlikely to perform as well on new data.

A common solution to this problem is called *holdout validation* (Figure 1-25): you simply hold out part of the training set to evaluate several candidate models and select the best one. The new held-out set is called the *validation set* (or the *development set*, or *dev set*). More specifically, you train multiple models with various hyperparameters on the reduced training set (i.e., the full training set minus the validation set), and you select the model that performs best on the validation set. After this holdout validation process, you train the best model on the full training set (including the validation set), and this gives you the final model. Lastly, you evaluate this final model on the test set to get an estimate of the generalization error.

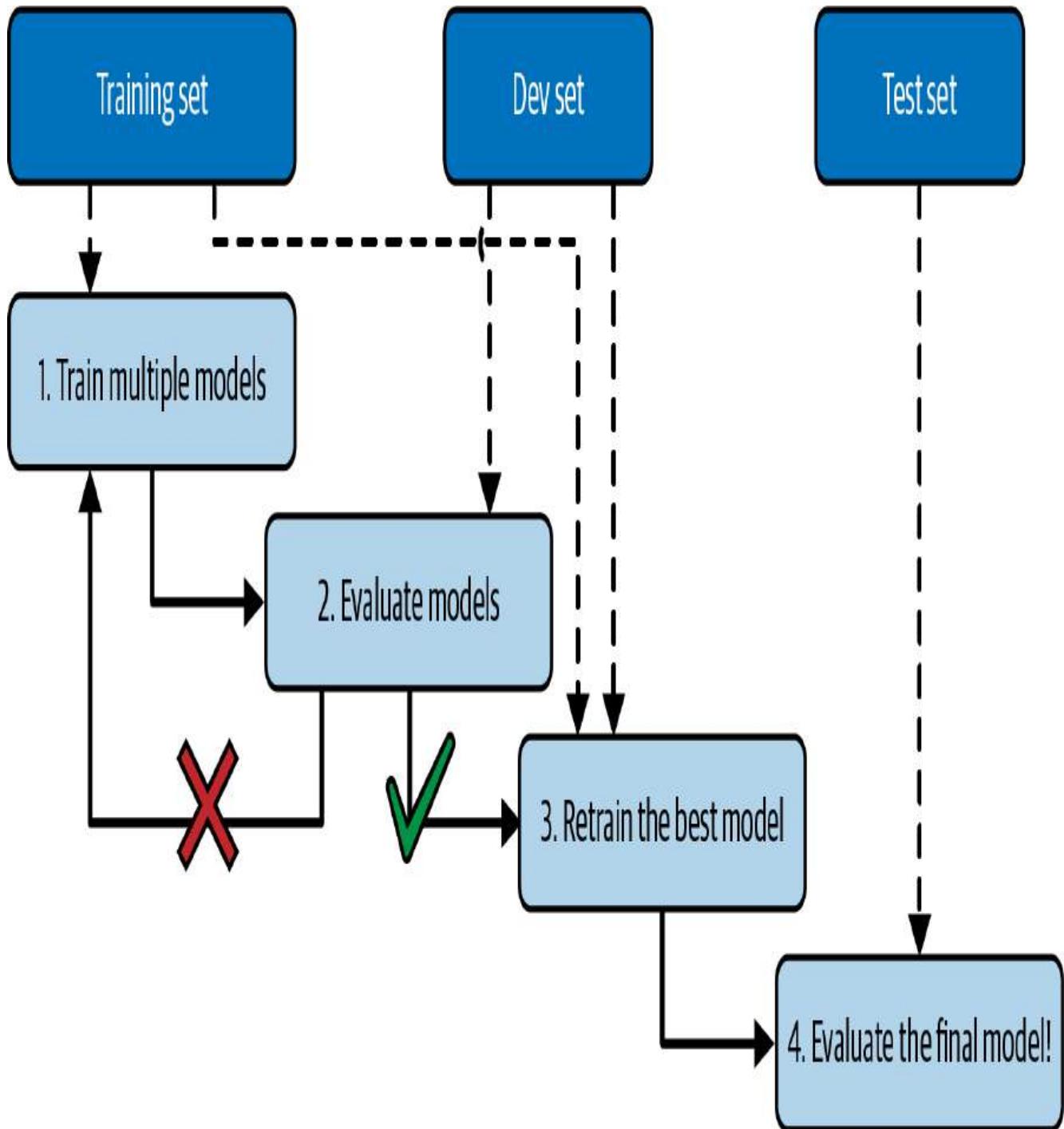


Figure 1-25. Model selection using holdout validation

This solution usually works quite well. However, if the validation set is too small, then the model evaluations will be imprecise: you may end up selecting a suboptimal model by mistake. Conversely, if the validation set is too large, then the remaining training set will be much smaller than the full training set. Why is this bad? Well, since the final model will be trained on the full training set, it is not ideal to compare candidate models trained on a much smaller training set. It would be like selecting the fastest sprinter to participate in a marathon. One way to solve this problem is to perform repeated *cross-validation*, using many small validation sets. Each model is evaluated

once per validation set after it is trained on the rest of the data. By averaging out all the evaluations of a model, you get a much more accurate measure of its performance. There is a drawback, however: the training time is multiplied by the number of validation sets.

## Data Mismatch

In some cases, it's easy to get a large amount of data for training, but this data probably won't be perfectly representative of the data that will be used in production. For example, suppose you want to create a mobile app to take pictures of flowers and automatically determine their species. You can easily download millions of pictures of flowers on the web, but they won't be perfectly representative of the pictures that will actually be taken using the app on a mobile device. Perhaps you only have 1,000 representative pictures (i.e., actually taken with the app).

In this case, the most important rule to remember is that both the validation set and the test set must be as representative as possible of the data you expect to use in production, so they should be composed exclusively of representative pictures: you can shuffle them and put half in the validation set and half in the test set (making sure that no duplicates or near-duplicates end up in both sets). After training your model on the web pictures, if you observe that the performance of the model on the validation set is disappointing, you will not know whether this is because your model has overfit the training set, or whether this is just due to the mismatch between the web pictures and the mobile app pictures.

One solution is to hold out some of the training pictures (from the web) in yet another set that Andrew Ng dubbed the *train-dev set* ([Figure 1-26](#)). After the model is trained (on the training set, *not* on the train-dev set), you can evaluate it on the train-dev set. If the model performs poorly, then it must have overfit the training set, so you should try to simplify or regularize the model, get more training data, and clean up the training data. But if it performs well on the train-dev set, then you can evaluate the model on the dev set. If it performs poorly, then the problem must be coming from the data mismatch. You can try to tackle this problem by preprocessing the web images to make them look more like the pictures that will be taken by the mobile app, and then retraining the model. Once you have a model that performs well on both the train-dev set and the dev set, you can evaluate it one last time on the test set to know how well it is likely to perform in production.

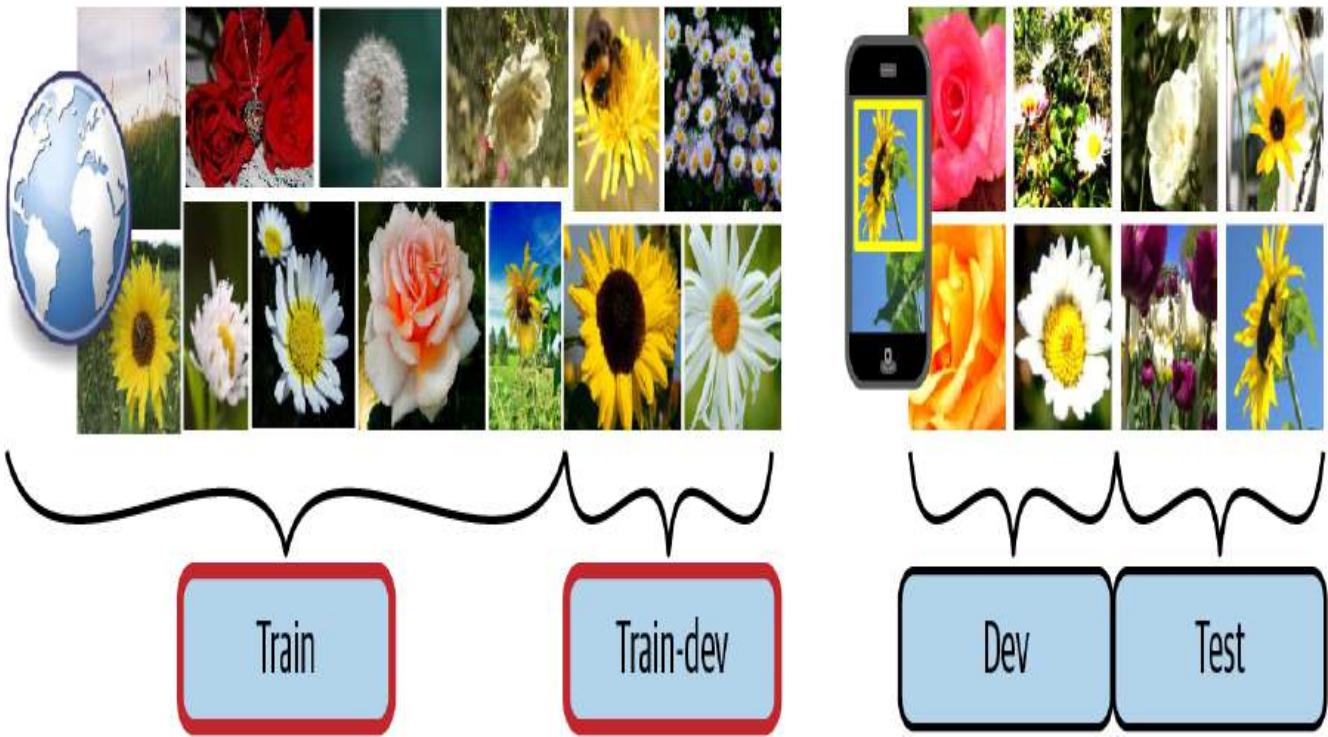


Figure 1-26. When real data is scarce (right), you may use similar abundant data (left) for training and hold out some of it in a train-dev set to evaluate overfitting; the real data is then used to evaluate data mismatch (dev set) and to evaluate the final model's performance (test set)

## NO FREE LUNCH THEOREM

A model is a simplified representation of the data. The simplifications are meant to discard the superfluous details that are unlikely to generalize to new instances. When you select a particular type of model, you are implicitly making *assumptions* about the data. For example, if you choose a linear model, you are implicitly assuming that the data is fundamentally linear and that the distance between the instances and the straight line is just noise, which can safely be ignored.

In a [famous 1996 paper](#),<sup>9</sup> David Wolpert demonstrated that if you make absolutely no assumption about the data, then there is no reason to prefer one model over any other. This is called the *No Free Lunch* (NFL) theorem. For some datasets the best model is a linear model, while for other datasets it is a neural network. There is no model that is *a priori* guaranteed to work better (hence the name of the theorem). The only way to know for sure which model is best is to evaluate them all. Since this is not possible, in practice you make some reasonable assumptions about the data and evaluate only a few reasonable models. For example, for simple tasks you may evaluate linear models with various levels of regularization, and for a complex problem you may evaluate various neural networks.

# Exercises

In this chapter we have covered some of the most important concepts in machine learning. In the next chapters we will dive deeper and write more code, but before we do, make sure you can answer the following questions:

1. How would you define machine learning?
2. Can you name four types of applications where it shines?
3. What is a labeled training set?
4. What are the two most common supervised tasks?
5. Can you name four common unsupervised tasks?
6. What type of algorithm would you use to allow a robot to walk in various unknown terrains?
7. What type of algorithm would you use to segment your customers into multiple groups?
8. Would you frame the problem of spam detection as a supervised learning problem or an unsupervised learning problem?
9. What is an online learning system?
10. What is out-of-core learning?
11. What type of algorithm relies on a similarity measure to make predictions?
12. What is the difference between a model parameter and a model hyperparameter?
13. What do model-based algorithms search for? What is the most common strategy they use to succeed? How do they make predictions?
14. Can you name four of the main challenges in machine learning?
15. If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name three possible solutions?
16. What is a test set, and why would you want to use it?
17. What is the purpose of a validation set?

18. What is the train-dev set, when do you need it, and how do you use it?

19. What can go wrong if you tune hyperparameters using the test set?

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.

---

- 1 Fun fact: this odd-sounding name is a statistics term introduced by Francis Galton while he was studying the fact that the children of tall people tend to be shorter than their parents. Since the children were shorter, he called this *regression to the mean*. This name was then applied to the methods he used to analyze correlations between variables.
- 2 Notice how animals are rather well separated from vehicles and how horses are close to deer but far from birds. Figure reproduced with permission from Richard Socher et al., “Zero-Shot Learning Through Cross-Modal Transfer”, *Proceedings of the 26th International Conference on Neural Information Processing Systems* 1 (2013): 935–943.
- 3 That’s when the system works perfectly. In practice it often creates a few clusters per person, and sometimes mixes up two people who look alike, so you may need to provide a few labels per person and manually clean up some clusters.
- 4 By convention, the Greek letter  $\theta$  (theta) is frequently used to represent model parameters.
- 5 It’s OK if you don’t understand all the code yet; I will present Scikit-Learn in the following chapters.
- 6 For example, knowing whether to write “to”, “two”, or “too”, depending on the context.
- 7 Peter Norvig et al., “The Unreasonable Effectiveness of Data”, *IEEE Intelligent Systems* 24, no. 2 (2009): 8–12.
- 8 Figure reproduced with permission from Michele Banko and Eric Brill, “Scaling to Very Very Large Corpora for Natural Language Disambiguation”, *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics* (2001): 26–33.
- 9 David Wolpert, “The Lack of A Priori Distinctions Between Learning Algorithms”, *Neural Computation* 8, no. 7 (1996): 1341–1390.

# Chapter 2. End-to-End Machine Learning Project

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 2nd chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

In this chapter you will work through an example project end to end, pretending to be a recently hired data scientist at a real estate company. This example is fictitious; the goal is to illustrate the main steps of a machine learning project, not to learn anything about the real estate business. Here are the main steps we will walk through:

1. Look at the big picture.
2. Get the data.
3. Explore and visualize the data to gain insights.
4. Prepare the data for machine learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

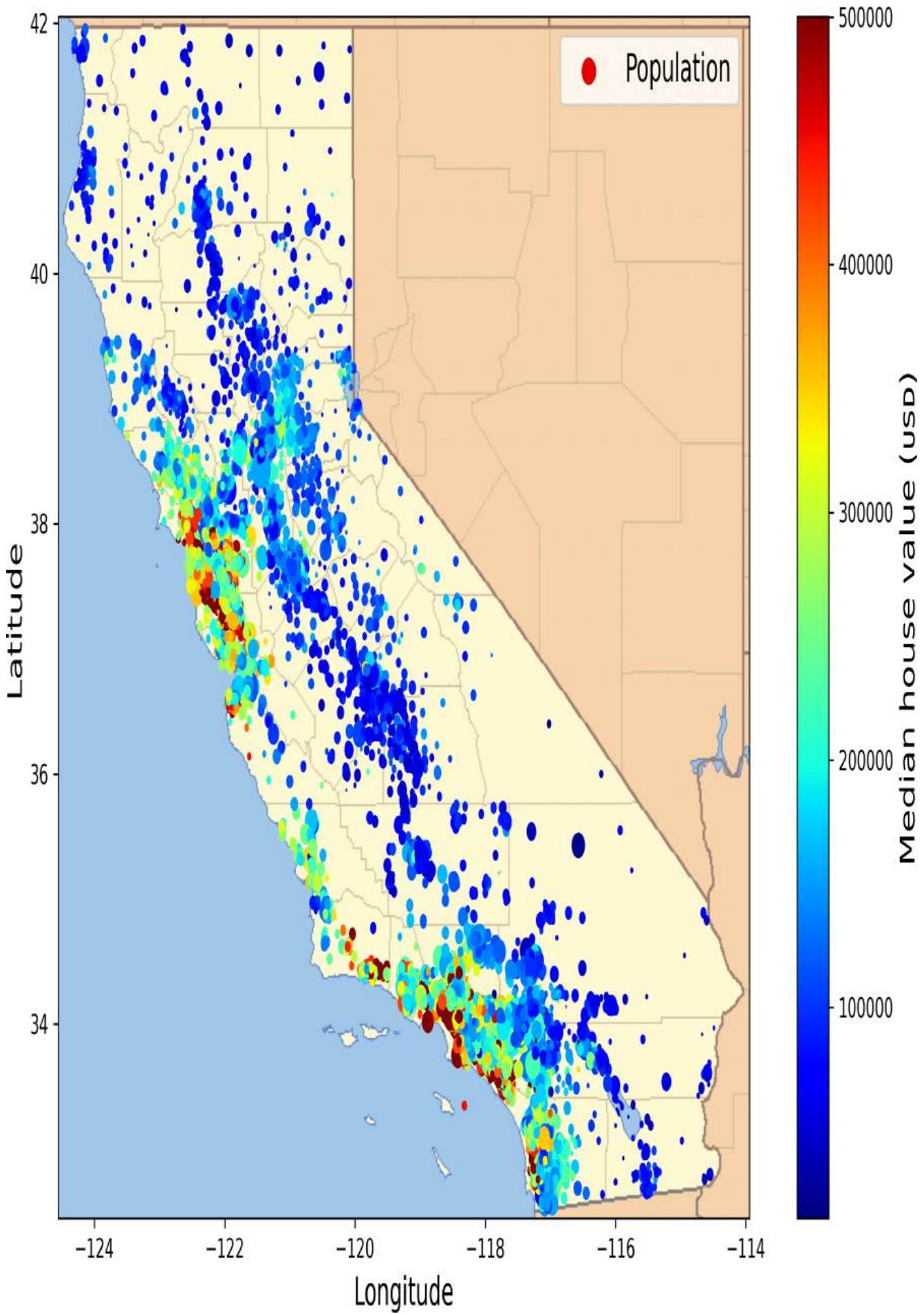
## Working with Real Data

When you are learning about machine learning, it is best to experiment with real-world data, not artificial datasets. Fortunately, there are thousands of open datasets to choose

from, ranging across all sorts of domains. Here are a few places you can look to get data:

- Popular open data repositories:
  - [Google Datasets Search](#)
  - [Hugging Face Datasets](#)
  - [OpenML.org](#)
  - [Kaggle.com](#)
  - [PapersWithCode.com](#)
  - [UC Irvine Machine Learning Repository](#)
  - [Stanford Large Network Dataset Collection](#)
  - [Amazon's AWS datasets](#)
  - [U.S. Government's Open Data](#)
  - [DataPortals.org](#)
  - [Wikipedia's list of machine learning datasets](#)

In this chapter we'll use the California Housing Prices dataset from the StatLib repository<sup>1</sup> (see [Figure 2-1](#)). This dataset is based on data from the 1990 California census. It is not exactly recent (a nice house in the Bay Area was still affordable at the time), but it has many qualities for learning, so we will pretend it is recent data. For teaching purposes I've added a categorical attribute and removed a few features.



*Figure 2-1. California housing prices*

## Look at the Big Picture

Welcome to the Machine Learning Housing Corporation! Your first task is to use California census data to build a model of housing prices in the state. This data includes metrics such as the population, median income, and median housing price for each block group in California. Block groups are the smallest geographical unit for which the US Census Bureau publishes sample data (a block group typically has a population of 600 to 3,000 people). I will call them “districts” for short.

Your model should learn from this data and be able to predict the median housing price in any district, given all the other metrics.

### TIP

Since you are a well-organized data scientist, the first thing you should do is pull out your machine learning project checklist. You can start with the one in [Link to Come]; it should work reasonably well for most machine learning projects, but make sure to adapt it to your needs. In this chapter we will go through many checklist items, but we will also skip a few, either because they are self-explanatory or because they will be discussed in later chapters.

## Frame the Problem

The first question to ask your boss is what exactly the business objective is. Building a model is probably not the end goal. How does the company expect to use and benefit from this model? Knowing the objective is important because it will determine how you frame the problem, which algorithms you will select, which performance measure you will use to evaluate your model, and how much effort you will spend tweaking it.

Your boss answers that your model’s output (a prediction of a district’s median housing price) will be essential to determine whether or not it is worth investing in a given area. More specifically, your model’s output will be fed to another machine learning system (see [Figure 2-2](#)), along with some other signals.<sup>2</sup> So it’s important to make our housing price model as accurate as we can.

The next question to ask your boss is what the current solution looks like (if any). The current situation will often give you a reference for performance, as well as insights on how to solve the problem. Your boss answers that the district housing prices are currently estimated manually by experts: a team gathers up-to-date information about a

district, and when they cannot get the median housing price, they estimate it using complex rules.

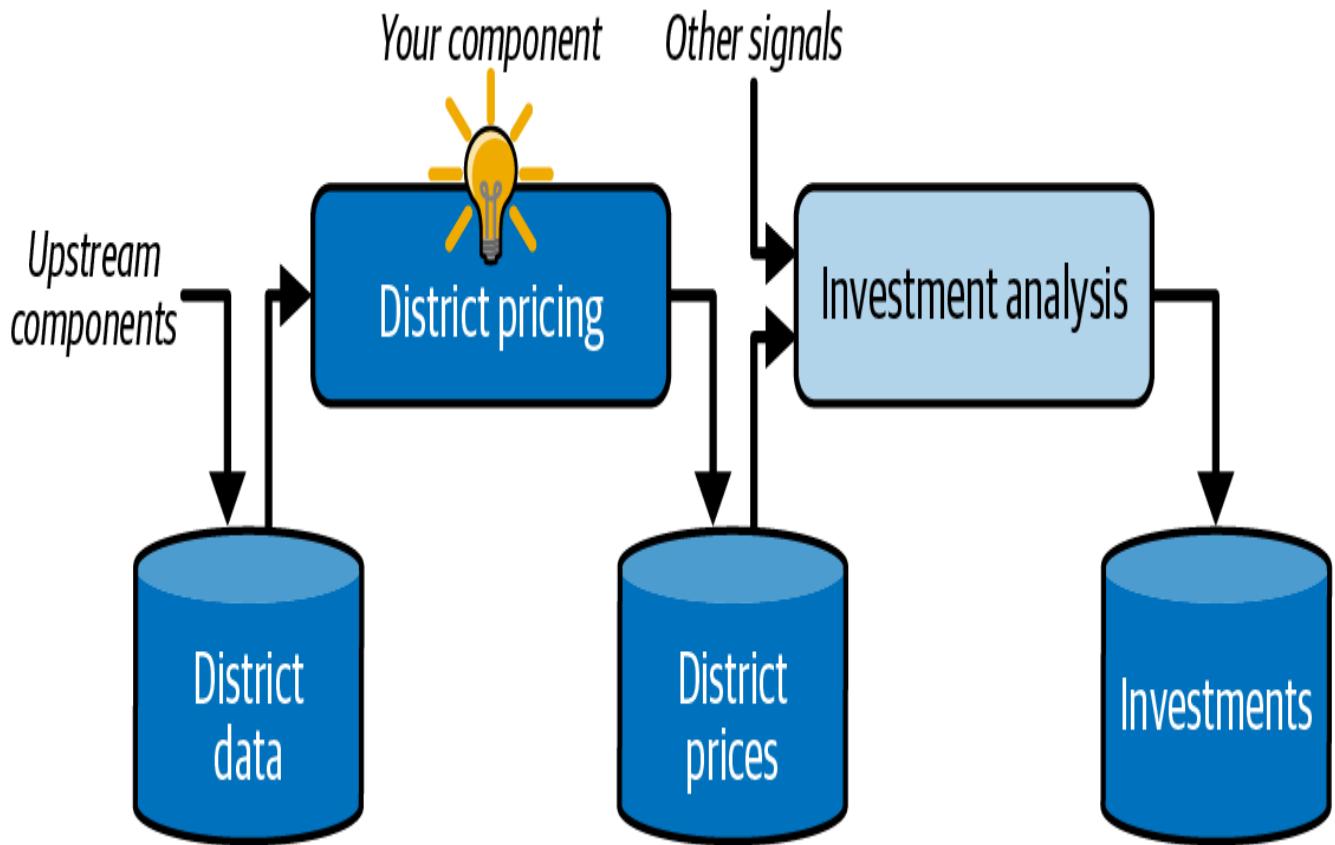


Figure 2-2. A machine learning pipeline for real estate investments

This is costly and time-consuming, and their estimates are not great; in cases where they manage to find out the actual median housing price, they often realize that their estimates were off by more than 30%. This is why the company thinks that it would be useful to train a model to predict a district's median housing price, given other data about that district. The census data looks like a great dataset to exploit for this purpose, since it includes the median housing prices of thousands of districts, as well as other data.

## PIPELINES

A sequence of data processing components is called a data *pipeline*. Pipelines are very common in machine learning systems, since there is a lot of data to manipulate and many data transformations to apply.

Components typically run asynchronously. Each component pulls in a large amount of data, processes it, and spits out the result in another data store. Then, some time later, the next component in the pipeline pulls in this data and spits out its own output. Each component is fairly self-contained: the interface between components is simply the data store. This makes the system simple to grasp (with the help of a data flow graph), and different teams can focus on different components. Moreover, if a component breaks down, the downstream components can often continue to run normally (at least for a while) by just using the last output from the broken component. This makes the architecture quite robust.

On the other hand, a broken component can go unnoticed for some time if proper monitoring is not implemented. The data gets stale and the overall system's performance drops.

With all this information, you are now ready to start designing your system. First, determine what kind of training supervision the model will need: is it a supervised, unsupervised, semi-supervised, self-supervised, or reinforcement learning task? And is it a classification task, a regression task, or something else? Should you use batch learning or online learning techniques? Before you read on, pause and try to answer these questions for yourself.

Have you found the answers? Let's see. This is clearly a typical supervised learning task, since the model can be trained with *labeled* examples (each instance comes with the expected output, i.e., the district's median housing price). It is a typical regression task, since the model will be asked to predict a value. More specifically, this is a *multiple regression* problem, since the system will use multiple features to make a prediction (the district's population, the median income, etc.). It is also a *univariate regression* problem, since we are only trying to predict a single value for each district. If we were trying to predict multiple values per district, it would be a *multivariate regression* problem. Finally, there is no continuous flow of data coming into the system, there is no particular need to adjust to changing data rapidly, and the data is small enough to fit in memory, so plain batch learning should do just fine.

## TIP

If the data were huge, you could either split your batch learning work across multiple servers (using the MapReduce technique) or use an online learning technique.

## Select a Performance Measure

Your next step is to select a performance measure. A typical performance measure for regression problems is the *root mean squared error* (RMSE). It gives an idea of how much error the system typically makes in its predictions, with a higher weight given to large errors. **Equation 2-1** shows the mathematical formula to compute the RMSE.

*Equation 2-1. Root mean squared error (RMSE)*

$$\text{RMSE}(\mathbf{X}, \mathbf{y}, h) = \sqrt{\frac{1}{m} \sum_{i=1}^m (h(\mathbf{x}^{(i)}) - y^{(i)})^2}$$

## NOTATIONS

This equation introduces several very common machine learning notations that I will use throughout this book:

- $m$  is the number of instances in the dataset you are measuring the RMSE on.
  - For example, if you are evaluating the RMSE on a validation set of 2,000 districts, then  $m = 2,000$ .
- $\mathbf{x}^{(i)}$  is a vector of all the feature values (excluding the label) of the  $i^{\text{th}}$  instance in the dataset, and  $y^{(i)}$  is its label (the desired output value for that instance).
  - For example, if the first district in the dataset is located at longitude  $-118.29^{\circ}$ , latitude  $33.91^{\circ}$ , and it has 1,416 inhabitants with a median income of \$38,372, and the median house value is \$156,400 (ignoring other features for now), then:

$$\mathbf{x}^{(1)} = \begin{pmatrix} -118.29 \\ 33.91 \\ 1,416 \\ 38,372 \end{pmatrix}$$

and:

$$y^{(1)} = 156,400$$

- $\mathbf{X}$  is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance, and the  $i^{\text{th}}$  row is equal to the transpose of  $\mathbf{x}^{(i)}$ , noted  $(\mathbf{x}^{(i)})^{\top}$ .<sup>3</sup>
  - For example, if the first district is as just described, then the matrix  $\mathbf{X}$  looks like this:

$$\mathbf{X} = \begin{pmatrix} (\mathbf{x}^{(1)})^\top \\ (\mathbf{x}^{(2)})^\top \\ \vdots \\ (\mathbf{x}^{(1999)})^\top \\ (\mathbf{x}^{(2000)})^\top \end{pmatrix} = \begin{pmatrix} -118.29 & 33.91 & 1,416 & 38,372 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

- $h$  is your system's prediction function, also called a *hypothesis*. When your system is given an instance's feature vector  $\mathbf{x}^{(i)}$ , it outputs a predicted value  $\hat{y}^{(i)} = h(\mathbf{x}^{(i)})$  for that instance ( $\hat{y}$  is pronounced "y-hat").
  - For example, if your system predicts that the median housing price in the first district is \$158,400, then  $\hat{y}^{(1)} = h(\mathbf{x}^{(1)}) = 158,400$ . The prediction error for this district is  $\hat{y}^{(1)} - y^{(1)} = 2,000$ .
- $\text{RMSE}(\mathbf{X}, \mathbf{y}, h)$  is the cost function measured on the set of examples using your hypothesis  $h$ .

We use lowercase italic font for scalar values (such as  $m$  or  $y^{(i)}$ ) and function names (such as  $h$ ), lowercase bold font for vectors (such as  $\mathbf{x}^{(i)}$ ), and uppercase bold font for matrices (such as  $\mathbf{X}$ ).

Although the RMSE is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function, especially when there are many outliers in the data, as the RMSE is quite sensitive to them. In that case, you may consider using the *mean absolute error* (MAE, also called the *average absolute deviation*), shown in [Equation 2-2](#):

*Equation 2-2. Mean absolute error (MAE)*

$$\text{MAE}(\mathbf{X}, \mathbf{y}, h) = \frac{1}{m} \sum_{i=1}^m |h(\mathbf{x}^{(i)}) - y^{(i)}|$$

Both the RMSE and the MAE are ways to measure the distance between two vectors: the vector of predictions and the vector of target values. Various distance measures, or *norms*, are possible:

- Computing the root of a sum of squares (RMSE) corresponds to the *Euclidean norm*: this is the notion of distance we are all familiar with. It is also called the  $\ell_2$  norm, noted  $\|\cdot\|_2$  (or just  $\|\cdot\|$ ).
- Computing the sum of absolutes (MAE) corresponds to the  $\ell_1$  norm, noted  $\|\cdot\|_1$ . This is sometimes called the *Manhattan norm* because it measures the distance between two points in a city if you can only travel along orthogonal city blocks.
- More generally, the  $\ell_k$  norm of a vector  $\mathbf{v}$  containing  $n$  elements is defined as  $\|\mathbf{v}\|_k = (\|v_1\|^k + \|v_2\|^k + \dots + \|v_n\|^k)^{1/k}$ .  $\ell_0$  gives the number of nonzero elements in the vector, and  $\ell_\infty$  gives the maximum absolute value in the vector.

The higher the norm index, the more it focuses on large values and neglects small ones. This is why the RMSE is more sensitive to outliers than the MAE. But when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.

## Check the Assumptions

Lastly, it is good practice to list and verify the assumptions that have been made so far (by you or others); this can help you catch serious issues early on. For example, the district prices that your system outputs are going to be fed into a downstream machine learning system, and you assume that these prices are going to be used as such. But what if the downstream system converts the prices into categories (e.g., “cheap”, “medium”, or “expensive”) and then uses those categories instead of the prices themselves? In this case, getting the price perfectly right is not important at all; your system just needs to get the category right. If that’s so, then the problem should have been framed as a classification task, not a regression task. You don’t want to find this out after working on a regression system for months.

Fortunately, after talking with the team in charge of the downstream system, you are confident that they do indeed need the actual prices, not just categories. Great! You’re all set, the lights are green, and you can start coding now!

## Get the Data

It’s time to get your hands dirty. Don’t hesitate to pick up your laptop and walk through the code examples. As I mentioned in the preface, all the code examples in this book are

open source and available **online** as Jupyter notebooks, which are interactive documents containing text, images, and executable code snippets (Python in our case). In this book I will assume you are running these notebooks on Google Colab, a free service that lets you run any Jupyter notebook directly online, without having to install anything on your machine. If you want to use another online platform (e.g., Kaggle) or if you want to install everything locally on your own machine, please see the instructions on the book’s GitHub page.

## Running the Code Examples Using Google Colab

First, open a web browser and visit <https://homl.info/colab-p>: this will lead you to Google Colab, and it will display the list of Jupyter notebooks for this book (see [Figure 2-3](#)). You will find one notebook per chapter, plus a few extra notebooks and tutorials for NumPy, Matplotlib, Pandas, linear algebra, and differential calculus. For example, if you click `02_end_to_end_machine_learning_project.ipynb`, the notebook from [Chapter 2](#) will open up in Google Colab (see [Figure 2-4](#)).

A Jupyter notebook is composed of a list of cells. Each cell contains either executable code or text. Try double-clicking the first text cell (which contains the sentence “Welcome to Machine Learning Housing Corp.!”). This will open the cell for editing. Notice that Jupyter notebooks use Markdown syntax for formatting (e.g., **\*\*bold\*\***, **\*italics\***, **# Title**, **[url](link text)**, and so on). Try modifying this text, then press Shift-Enter to see the result.

## Open notebook

Examples > Enter a GitHub URL or search by organization or user  
ageron   Include private repos

Recent > Repository:  Branch: 

Google Drive > ageron/handson-mlp  main 

GitHub > Path

Upload >  01\_the\_machine\_learning\_landscape.ipynb  

  02\_end\_to\_end\_machine\_learning\_project.ipynb  

 03\_classification.ipynb  

Figure 2-3. List of notebooks in Google Colab



Figure 2-4. Your notebook in Google Colab

Next, create a new code cell by selecting Insert → “Code cell” from the menu. Alternatively, you can click the + Code button in the toolbar, or hover your mouse over

the bottom of a cell until you see + Code and + Text appear, then click + Code. In the new code cell, type some Python code, such as `print("Hello World")`, then press Shift-Enter to run this code (or click the ▶ button on the left side of the cell).

If you’re not logged in to your Google account, you’ll be asked to log in now (if you don’t already have a Google account, you’ll need to create one). Once you are logged in, when you try to run the code you’ll see a security warning telling you that this notebook was not authored by Google. A malicious person could create a notebook that tries to trick you into entering your Google credentials so they can access your personal data, so before you run a notebook, always make sure you trust its author (or double-check what each code cell will do before running it). Assuming you trust me (or you plan to check every code cell), you can now click “Run anyway”.

Colab will then allocate a new *runtime* for you: this is a free virtual machine located on Google’s servers that contains a bunch of tools and Python libraries, including everything you’ll need for most chapters (in some chapters, you’ll need to run a command to install additional libraries). This will take a few seconds. Next, Colab will automatically connect to this runtime and use it to execute your new code cell.

Importantly, the code runs on the runtime, *not* on your machine. The code’s output will be displayed under the cell. Congrats, you’ve run some Python code on Colab!

### TIP

To insert a new code cell, you can also type Ctrl-M (or Cmd-M on macOS) followed by A (to insert above the active cell) or B (to insert below). There are many other keyboard shortcuts available: you can view and edit them by typing Ctrl-M (or Cmd-M) then H. If you choose to run the notebooks on Kaggle or on your own machine using JupyterLab or an IDE such as Visual Studio Code with the Jupyter extension, you will see some minor differences—runtimes are called *kernels*, the user interface and keyboard shortcuts are slightly different, etc.—but switching from one Jupyter environment to another is not too hard.

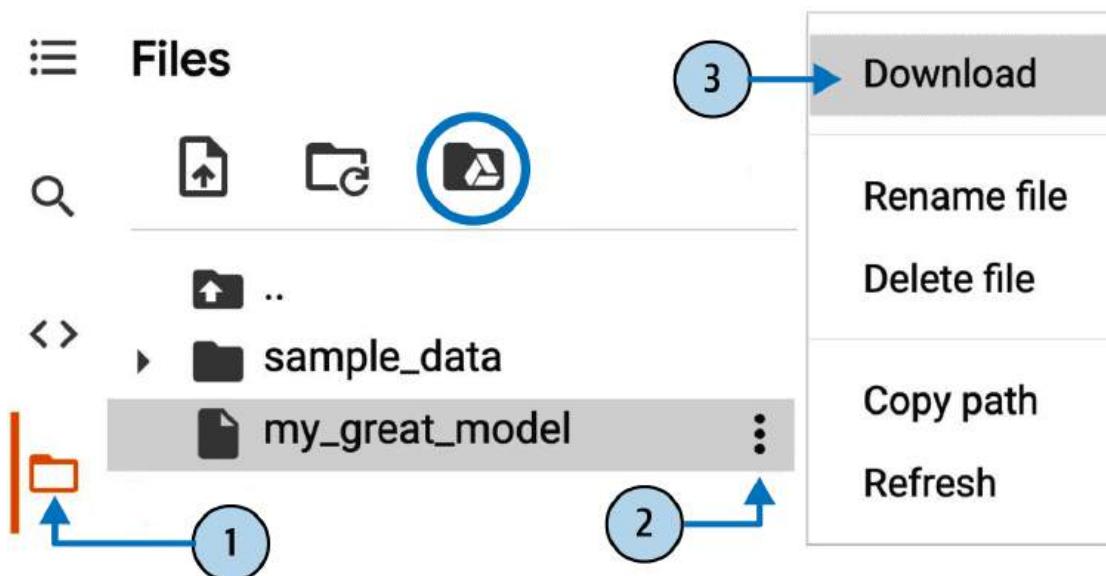
## Saving Your Code Changes and Your Data

You can make changes to a Colab notebook, and they will persist for as long as you keep your browser tab open. But once you close it, the changes will be lost. To avoid this, make sure you save a copy of the notebook to your Google Drive by selecting File → “Save a copy in Drive”. Alternatively, you can download the notebook to your computer by selecting File → Download → “Download .ipynb”. Then you can later visit <https://colab.research.google.com> and open the notebook again (either from Google Drive or by uploading it from your computer).

## WARNING

Google Colab is meant only for interactive use: you can play around in the notebooks and tweak the code as you like, but you cannot let the notebooks run unattended for a long period of time, or else the runtime will be shut down and all of its data will be lost.

If the notebook generates data that you care about, make sure you download this data before the runtime shuts down. To do this, click the Files icon (see step 1 in [Figure 2-5](#)), find the file you want to download, click the vertical dots next to it (step 2), and click Download (step 3). Alternatively, you can mount your Google Drive on the runtime, allowing the notebook to read and write files directly to Google Drive as if it were a local directory. For this, click the Files icon (step 1), then click the Google Drive icon (circled in [Figure 2-5](#)) and follow the on-screen instructions.



*Figure 2-5. Downloading a file from a Google Colab runtime (steps 1 to 3), or mounting your Google Drive (circled icon)*

By default, your Google Drive will be mounted at `/content/drive/MyDrive`. If you want to back up a data file, simply copy it to this directory by running `!cp /content/my_great_model /content/drive/MyDrive`. Any command starting with a bang (!) is treated as a shell command, not as Python code: `cp` is the Linux shell command to copy a file from one path to another. Note that Colab runtimes run on Linux (specifically, Ubuntu).

## The Power and Danger of Interactivity

Jupyter notebooks are interactive, and that's a great thing: you can run each cell one by one, stop at any point, insert a cell, play with the code, go back and run the same cell again, etc., and I highly encourage you to do so. If you just run the cells one by one without ever playing around with them, you won't learn as fast. However, this flexibility comes at a price: it's very easy to run cells in the wrong order, or to forget to run a cell. If this happens, the subsequent code cells are likely to fail. For example, the very first code cell in each notebook contains setup code (such as imports), so make sure you run it first, or else nothing will work.

### TIP

If you ever run into a weird error, try restarting the runtime (by selecting Runtime → “Restart runtime” from the menu) and then run all the cells again from the beginning of the notebook. This often solves the problem. If not, it’s likely that one of the changes you made broke the notebook: just revert to the original notebook and try again. If it still fails, please file an issue on GitHub.

## Book Code Versus Notebook Code

You may sometimes notice some little differences between the code in this book and the code in the notebooks. This may happen for several reasons:

- A library may have changed slightly by the time you read these lines, or perhaps despite my best efforts I made an error in the book. Sadly, I cannot magically fix the code in your copy of this book (unless you are reading an electronic copy and you can download the latest version), but I *can* fix the notebooks. So, if you run into an error after copying code from this book, please look for the fixed code in the notebooks: I will strive to keep them error-free and up-to-date with the latest library versions.
- The notebooks contain some extra code to beautify the figures (adding labels, setting font sizes, etc.) and to save them in high resolution for this book. You can safely ignore this extra code if you want.

I optimized the code for readability and simplicity: I made it as linear and flat as possible, defining very few functions or classes. The goal is to ensure that the code you are running is generally right in front of you, and not nested within several layers of abstractions that you have to search through. This also makes it easier for you to play with the code. For simplicity, there's limited error handling, and I placed some of the least common imports right where they are needed (instead of placing them at the top of the file, as is recommended by the PEP 8 Python style guide). That said, your production code will not be very different: just a bit more modular, and with additional tests and error handling.

OK! Once you're comfortable with Colab, you're ready to download the data.

## Download the Data

In typical environments your data would be available in a relational database or some other common data store, and spread across multiple tables/documents/files. To access it, you would first need to get your credentials and access authorizations<sup>4</sup> and familiarize yourself with the data schema. In this project, however, things are much simpler: you will just download a single compressed file, *housing.tgz*, which contains a comma-separated values (CSV) file called *housing.csv* with all the data.

Rather than manually downloading and decompressing the data, it's usually preferable to write a function that does it for you. This is useful in particular if the data changes regularly: you can write a small script that uses the function to fetch the latest data (or

you can set up a scheduled job to do that automatically at regular intervals). Automating the process of fetching the data is also useful if you need to install the dataset on multiple machines.

Here is the function to fetch and load the data:

```
from pathlib import Path
import pandas as pd
import tarfile
import urllib.request

def load_housing_data():
    tarball_path = Path("datasets/housing.tgz")
    if not tarball_path.is_file():
        Path("datasets").mkdir(parents=True, exist_ok=True)
        url = "https://github.com/ageron/data/raw/main/housing.tgz"
        urllib.request.urlretrieve(url, tarball_path)
        with tarfile.open(tarball_path) as housing_tarball:
            housing_tarball.extractall(path="datasets")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing_full = load_housing_data()
```

When `load_housing_data()` is called, it looks for the `datasets/housing.tgz` file. If it does not find it, it creates the `datasets` directory inside the current directory (which is `/content` by default, in Colab), downloads the `housing.tgz` file from the `ageron/data` GitHub repository, and extracts its content into the `datasets` directory; this creates the `datasets/housing` directory with the `housing.csv` file inside it. Lastly, the function loads this CSV file into a Pandas DataFrame object containing all the data, and returns it.

### NOTE

If you get an SSL CERTIFICATE\_VERIFY\_FAILED error on macOS, then you most likely need to install the `certifi` package, as explained at <https://homl.info/sslerror>.

### NOTE

If you are using Python 3.12 or 3.13, you should add `filter='data'` to the `extractall()` method's arguments: this limits what the extraction algorithm can do and improves security (see the documentation for more details).

## Take a Quick Look at the Data Structure

You start by looking at the top five rows of data using the DataFrame's `head()` method (see [Figure 2-6](#)).

```
housing.head()
```

	longitude	latitude	housing_median_age	median_income	ocean_proximity	median_house_value
0	-122.23	37.88	41.0	8.3252	NEAR BAY	452600.0
1	-122.22	37.86	21.0	8.3014	NEAR BAY	358500.0
2	-122.24	37.85	52.0	7.2574	NEAR BAY	352100.0
3	-122.25	37.85	52.0	5.6431	NEAR BAY	341300.0
4	-122.25	37.85	52.0	3.8462	NEAR BAY	342200.0

*Figure 2-6. Top five rows in the dataset*

Each row represents one district. There are 10 attributes (they are not all shown in the screenshot): `longitude`, `latitude`, `housing_median_age`, `total_rooms`,

`total_bedrooms`, `population`, `households`, `median_income`, `median_house_value`, and `ocean_proximity`.

The `info()` method is useful to get a quick description of the data, in particular the total number of rows, each attribute's type, and the number of non-null values:

```
>>> housing_full.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
 #   Column            Non-Null Count  Dtype  
 ---  -- 
 0   longitude         20640 non-null   float64
 1   latitude          20640 non-null   float64
 2   housing_median_age 20640 non-null   float64
 3   total_rooms        20640 non-null   float64
 4   total_bedrooms     20433 non-null   float64
 5   population         20640 non-null   float64
 6   households         20640 non-null   float64
 7   median_income      20640 non-null   float64
 8   median_house_value 20640 non-null   float64
 9   ocean_proximity    20640 non-null   object  
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

## NOTE

In this book, when a code example contains a mix of code and outputs, as is the case here, it is formatted like in the Python interpreter, for better readability: the code lines are prefixed with `>>>` (or `...` for indented blocks), and the outputs have no prefix.

There are 20,640 instances in the dataset, which means that it is fairly small by machine learning standards, but it's perfect to get started. You notice that the `total_bedrooms` attribute has only 20,433 non-null values, meaning that 207 districts are missing this feature. You will need to take care of this later.

All attributes are numerical, except for `ocean_proximity`. Its type is `object`, so it could hold any kind of Python object. But since you loaded this data from a CSV file, you know that it must be a text attribute. When you looked at the top five rows, you probably noticed that the values in the `ocean_proximity` column were repetitive, which means that it is probably a categorical attribute. You can find out what categories exist and how many districts belong to each category by using the `value_counts()` method:

```
>>> housing_full["ocean_proximity"].value_counts()
ocean_proximity
<1H OCEAN      9136
INLAND        6551
NEAR OCEAN     2658
NEAR BAY       2290
ISLAND          5
Name: count, dtype: int64
```

Let's look at the other fields. The `describe()` method shows a summary of the numerical attributes (Figure 2-7).

```
housing.describe()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	median_house_value
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000000	20640.000000
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553	206855.816909
std	2.003532	2.135952	12.585558	2181.615252	421.385070	115395.615874
min	-124.350000	32.540000	1.000000	2.000000	1.000000	14999.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.000000	119600.000000
50%	-118.490000	34.260000	29.000000	2127.000000	435.000000	179700.000000
75%	-118.010000	37.710000	37.000000	3148.000000	647.000000	264725.000000
max	-114.310000	41.950000	52.000000	39320.000000	6445.000000	500001.000000

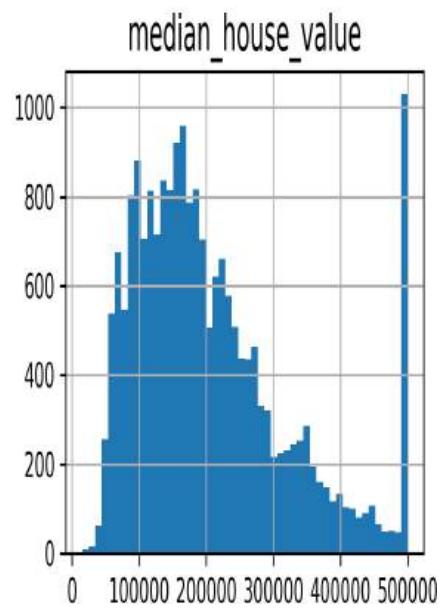
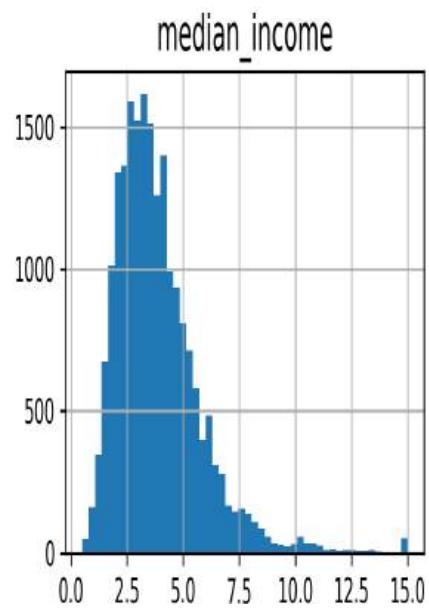
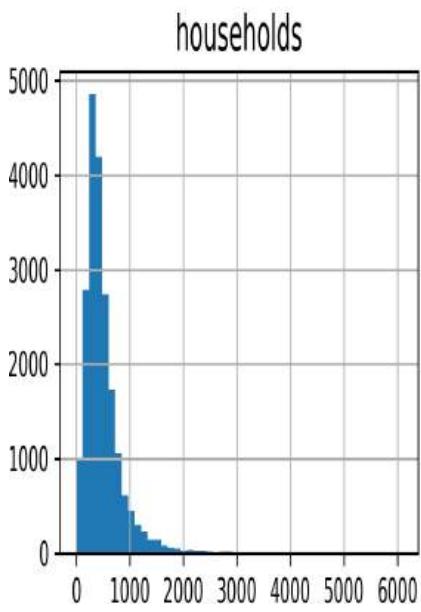
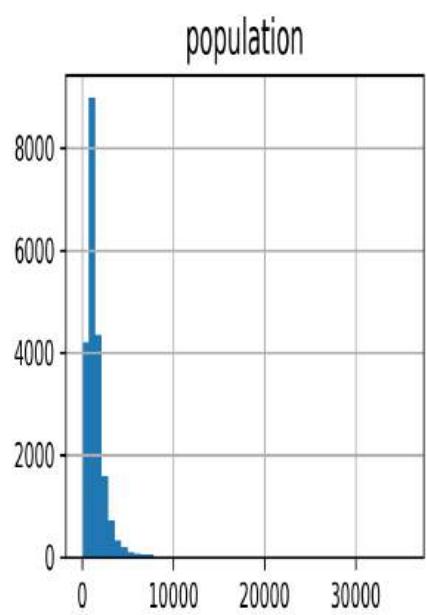
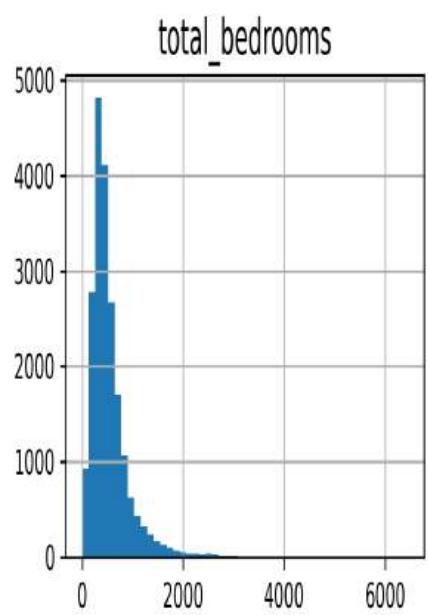
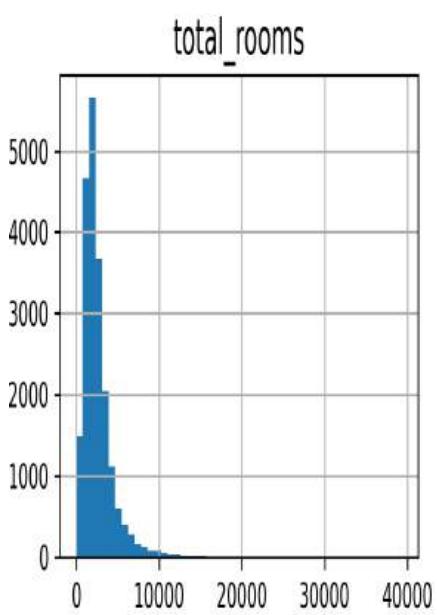
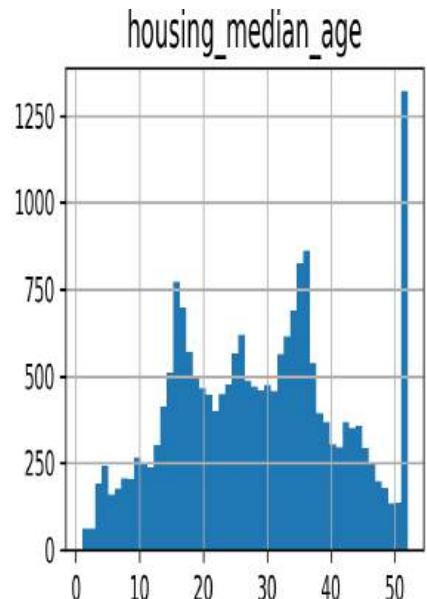
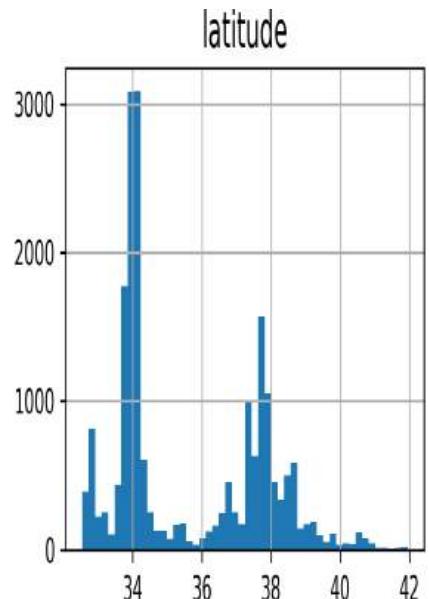
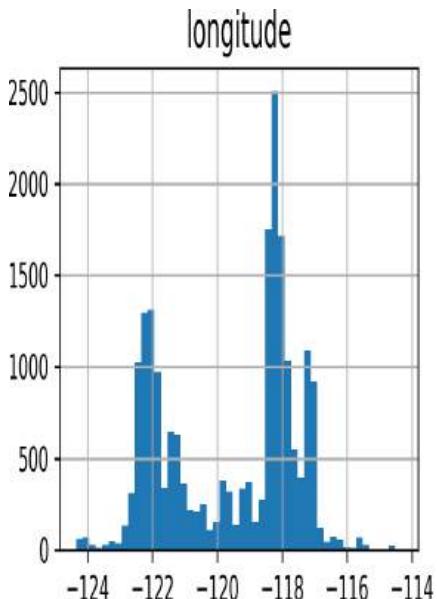
*Figure 2-7. Summary of each numerical attribute*

The `count`, `mean`, `min`, and `max` rows are self-explanatory. Note that the null values are ignored (so, for example, the `count` of `total_bedrooms` is 20,433, not 20,640). The `std` row shows the *standard deviation*, which measures how dispersed the values are.<sup>5</sup> The 25%, 50%, and 75% rows show the corresponding *percentiles*: a percentile indicates the value below which a given percentage of observations in a group of observations fall. For example, 25% of the districts have a `housing_median_age` lower than 18, while 50% are lower than 29 and 75% are lower than 37. These are often called the 25th percentile (or first *quartile*), the median, and the 75th percentile (or third quartile).

Another quick way to get a feel of the type of data you are dealing with is to plot a histogram for each numerical attribute. A histogram shows the number of instances (on the vertical axis) that have a given value range (on the horizontal axis). You can either plot this one attribute at a time, or you can call the `hist()` method on the whole dataset (as shown in the following code example), and it will plot a histogram for each numerical attribute (see [Figure 2-8](#)). The number of value ranges can be adjusted using the `bins` argument (try playing with it to see how it affects the histograms).

```
import matplotlib.pyplot as plt

housing_full.hist(bins=50, figsize=(12, 8))
plt.show()
```



*Figure 2-8. A histogram for each numerical attribute*

Looking at these histograms, you notice a few things:

- First, the median income attribute does not look like it is expressed in US dollars (USD). After checking with the team that collected the data, you are told that the data has been scaled and capped at 15 (actually, 15.0001) for higher median incomes, and at 0.5 (actually, 0.4999) for lower median incomes. The numbers represent roughly tens of thousands of dollars (e.g., 3 actually means about \$30,000). Working with preprocessed attributes is common in machine learning, and it is not necessarily a problem, but you should try to understand how the data was computed.
- The housing median age and the median house value were also capped. The latter may be a serious problem since it is your target attribute (your labels). Your machine learning algorithms may learn that prices never go beyond that limit. You need to check with your client team (the team that will use your system's output) to see if this is a problem or not. If they tell you that they need precise predictions even beyond \$500,000, then you have two options:
  - Collect proper labels for the districts whose labels were capped.
  - Remove those districts from the training set (and also from the test set, since your system should not be evaluated poorly if it predicts values beyond \$500,000).
- These attributes have very different scales. We will discuss this later in this chapter, when we explore feature scaling.
- Finally, many histograms are *skewed right*: they extend much farther to the right of the median than to the left. This may make it a bit harder for some machine learning algorithms to detect patterns. Later, you'll try transforming these attributes to have more symmetrical and bell-shaped distributions.

You should now have a better understanding of the kind of data you're dealing with.

### **WARNING**

Wait! Before you look at the data any further, you need to create a test set, put it aside, and never look at it.

## Create a Test Set

It may seem strange to voluntarily set aside part of the data at this stage. After all, you have only taken a quick glance at the data, and surely you should learn a whole lot more about it before you decide what algorithms to use, right? This is true, but your brain is an amazing pattern detection system, which also means that it is highly prone to overfitting: if you look at the test set, you may stumble upon some seemingly interesting pattern in the test data that leads you to select a particular kind of machine learning model. When you estimate the generalization error using the test set, your estimate will be too optimistic, and you will launch a system that will not perform as well as expected. This is called *data snooping* bias.

Creating a test set is theoretically simple; pick some instances randomly, typically 20% of the dataset (or less if your dataset is very large), and set them aside:

```
import numpy as np

def shuffle_and_split_data(data, test_ratio, rng):
    shuffled_indices = rng.permutation(len(data))
    test_set_size = int(len(data) * test_ratio)
    test_indices = shuffled_indices[:test_set_size]
    train_indices = shuffled_indices[test_set_size:]
    return data.iloc[train_indices], data.iloc[test_indices]
```

You can then use this function like this:

```
>>> rng = np.random.default_rng() # default random number generator
>>> train_set, test_set = shuffle_and_split_data(housing_full, 0.2, rng)
>>> len(train_set)
16512
>>> len(test_set)
4128
```

Well, this works, but it is not perfect: if you run the program again, it will generate a different test set! Over time, you (or your machine learning algorithms) will get to see the whole dataset, which is what you want to avoid.

One solution is to save the test set on the first run and then load it in subsequent runs. Another option is to set the random number generator's seed (e.g., by passing `seed=42` to the `default_rng()` function)<sup>6</sup> to ensure it always generates the same sequence of random numbers, every time you run the program.

However, both these solutions will break the next time you fetch an updated dataset. To have a stable train/test split even after updating the dataset, a common solution is to use

each instance's identifier to decide whether or not it should go in the test set (assuming instances have unique and immutable identifiers). For example, you could compute a hash of each instance's identifier and put that instance in the test set if the hash is lower than or equal to 20% of the maximum hash value. This ensures that the test set will remain consistent across multiple runs, even if you refresh the dataset. The new test set will contain 20% of the new instances, but it will not contain any instance that was previously in the training set.

Here is a possible implementation:

```
from zlib import crc32

def is_id_in_test_set(identifier, test_ratio):
    return crc32(np.int64(identifier)) < test_ratio * 2**32

def split_data_with_id_hash(data, test_ratio, id_column):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: is_id_in_test_set(id_, test_ratio))
    return data.loc[~in_test_set], data.loc[in_test_set]
```

Unfortunately, the housing dataset does not have an identifier column. The simplest solution is to use the row index as the ID:

```
housing_with_id = housing_full.reset_index() # adds an `index` column
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "index")
```

If you use the row index as a unique identifier, you need to make sure that new data gets appended to the end of the dataset and that no row ever gets deleted. If this is not possible, then you can try to use the most stable features to build a unique identifier. For example, a district's latitude and longitude are guaranteed to be stable for a few million years, so you could combine them into an ID like so:<sup>7</sup>

```
housing_with_id["id"] = (housing_full["longitude"] * 1000
                         + housing_full["latitude"])
train_set, test_set = split_data_with_id_hash(housing_with_id, 0.2, "id")
```

Scikit-Learn provides a few functions to split datasets into multiple subsets in various ways. The simplest function is `train_test_split()`, which does pretty much the same thing as the `shuffle_and_split_data()` function we defined earlier, with a couple of additional features. First, there is a `random_state` parameter that allows you to set the random generator seed. Second, you can pass it multiple datasets with an identical

number of rows, and it will split them on the same indices (this is very useful, for example, if you have a separate DataFrame for labels):

```
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing_full, test_size=0.2,
                                       random_state=42)
```

So far we have considered purely random sampling methods. This is generally fine if your dataset is large enough (especially relative to the number of attributes), but if it is not, you run the risk of introducing a significant sampling bias. When employees at a survey company decides to call 1,000 people to ask them a few questions, they don't just pick 1,000 people randomly in a phone book. They try to ensure that these 1,000 people are representative of the whole population, with regard to the questions they want to ask. For example, the US population is 51.1% females and 48.9% males, so a well-conducted survey in the US would try to maintain this ratio in the sample: 511 females and 489 males (at least if it seems possible that the answers may vary across genders). This is called *stratified sampling*: the population is divided into homogeneous subgroups called *strata*, and the right number of instances are sampled from each stratum to guarantee that the test set is representative of the overall population. If the people running the survey used purely random sampling, there would be about a 10.7% chance of sampling a skewed test set with less than 48.5% female or more than 53.5% female participants. Either way, the survey results would likely be quite biased.

Suppose you've chatted with some experts who told you that the median income is a very important attribute to predict median housing prices. You may want to ensure that the test set is representative of the various categories of incomes in the whole dataset. Since the median income is a continuous numerical attribute, you first need to create an income category attribute. Let's look at the median income histogram more closely (back in [Figure 2-8](#)): most median income values are clustered around 1.5 to 6 (i.e., \$15,000–\$60,000), but some median incomes go far beyond 6. It is important to have a sufficient number of instances in your dataset for each stratum, or else the estimate of a stratum's importance may be biased. This means that you should not have too many strata, and each stratum should be large enough. The following code uses the `pd.cut()` function to create an income category attribute with five categories (labeled from 1 to 5); category 1 ranges from 0 to 1.5 (i.e., less than \$15,000), category 2 from 1.5 to 3, and so on:

```

housing_full["income_cat"] = pd.cut(housing_full["median_income"],
                                    bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                                    labels=[1, 2, 3, 4, 5])

```

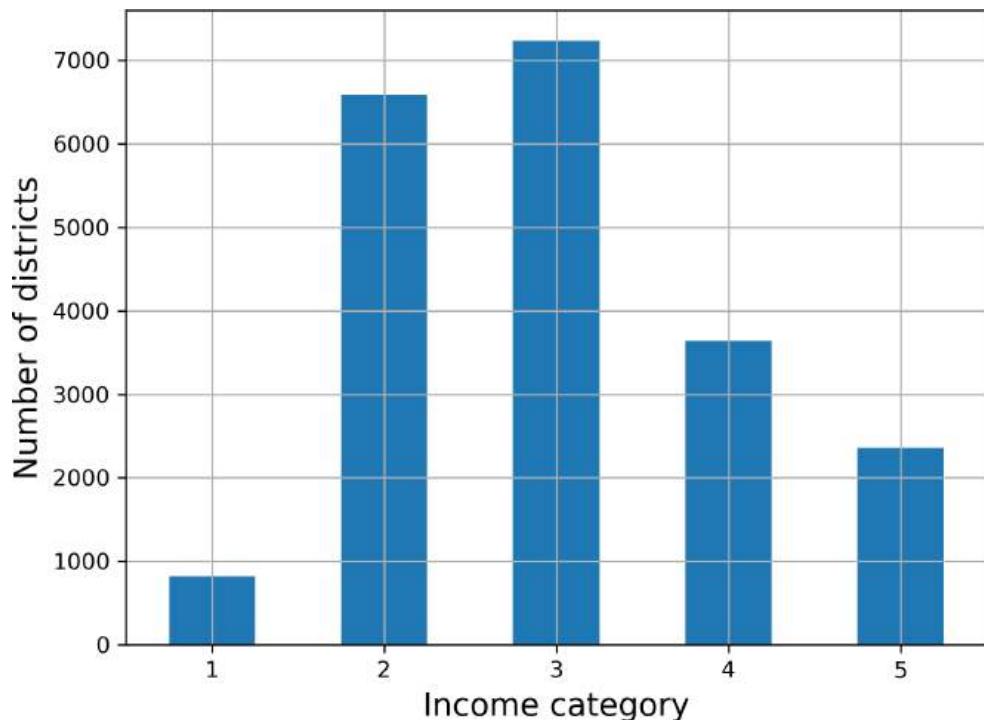
These income categories are represented in [Figure 2-9](#):

```

cat_counts = housing_full["income_cat"].value_counts().sort_index()
cat_counts.plot.bar(rot=0, grid=True)
plt.xlabel("Income category")
plt.ylabel("Number of districts")
plt.show()

```

Now you are ready to do stratified sampling based on the income category. Scikit-Learn provides a number of splitter classes in the `sklearn.model_selection` package that implement various strategies to split your dataset into a training set and a test set. Each splitter has a `split()` method that returns an iterator over different training/test splits of the same data.



*Figure 2-9. Histogram of income categories*

To be precise, the `split()` method yields the training and test *indices*, not the data itself. Having multiple splits can be useful if you want to better estimate the performance of your model, as you will see when we discuss cross-validation later in this chapter. For example, the following code generates 10 different stratified splits of the same dataset:

```
from sklearn.model_selection import StratifiedShuffleSplit

splitter = StratifiedShuffleSplit(n_splits=10, test_size=0.2, random_state=42)
strat_splits = []
for train_index, test_index in splitter.split(housing_full,
                                              housing_full["income_cat"]):
    strat_train_set_n = housing_full.iloc[train_index]
    strat_test_set_n = housing_full.iloc[test_index]
    strat_splits.append([strat_train_set_n, strat_test_set_n])
```

For now, you can just use the first split:

```
strat_train_set, strat_test_set = strat_splits[0]
```

Or, since stratified sampling is fairly common, there's a shorter way to get a single split using the `train_test_split()` function with the `stratify` argument:

```
strat_train_set, strat_test_set = train_test_split(
    housing_full, test_size=0.2, stratify=housing_full["income_cat"],
    random_state=42)
```

Let's see if this worked as expected. You can start by looking at the income category proportions in the test set:

```
>>> strat_test_set["income_cat"].value_counts() / len(strat_test_set)
income_cat
3    0.350533
2    0.318798
4    0.176357
5    0.114341
1    0.039971
Name: count, dtype: float64
```

With similar code you can measure the income category proportions in the full dataset. [Figure 2-10](#) compares the income category proportions in the overall dataset, in the test set generated with stratified sampling, and in a test set generated using purely random sampling. As you can see, the test set generated using stratified sampling has income category proportions almost identical to those in the full dataset, whereas the test set generated using purely random sampling is skewed.

Income Category	Overall %	Stratified %	Random %	Strat. Error %	Rand. Error %
1	3.98	4.00	4.24	0.36	6.45
2	31.88	31.88	30.74	-0.02	-3.59
3	35.06	35.05	34.52	-0.01	-1.53
4	17.63	17.64	18.41	0.03	4.42
5	11.44	11.43	12.09	-0.08	5.63

Figure 2-10. Sampling bias comparison of stratified versus purely random sampling

You won't use the `income_cat` column again, so you might as well drop it, reverting the data back to its original state:

```
for set_ in (strat_train_set, strat_test_set):
    set_.drop("income_cat", axis=1, inplace=True)
```

We spent quite a bit of time on test set generation for a good reason: this is an often neglected but critical part of a machine learning project. Moreover, many of these ideas will be useful later when we discuss cross-validation. Now it's time to move on to the next stage: exploring the data.

## Explore and Visualize the Data to Gain Insights

So far you have only taken a quick glance at the data to get a general understanding of the kind of data you are manipulating. Now the goal is to go into a little more depth.

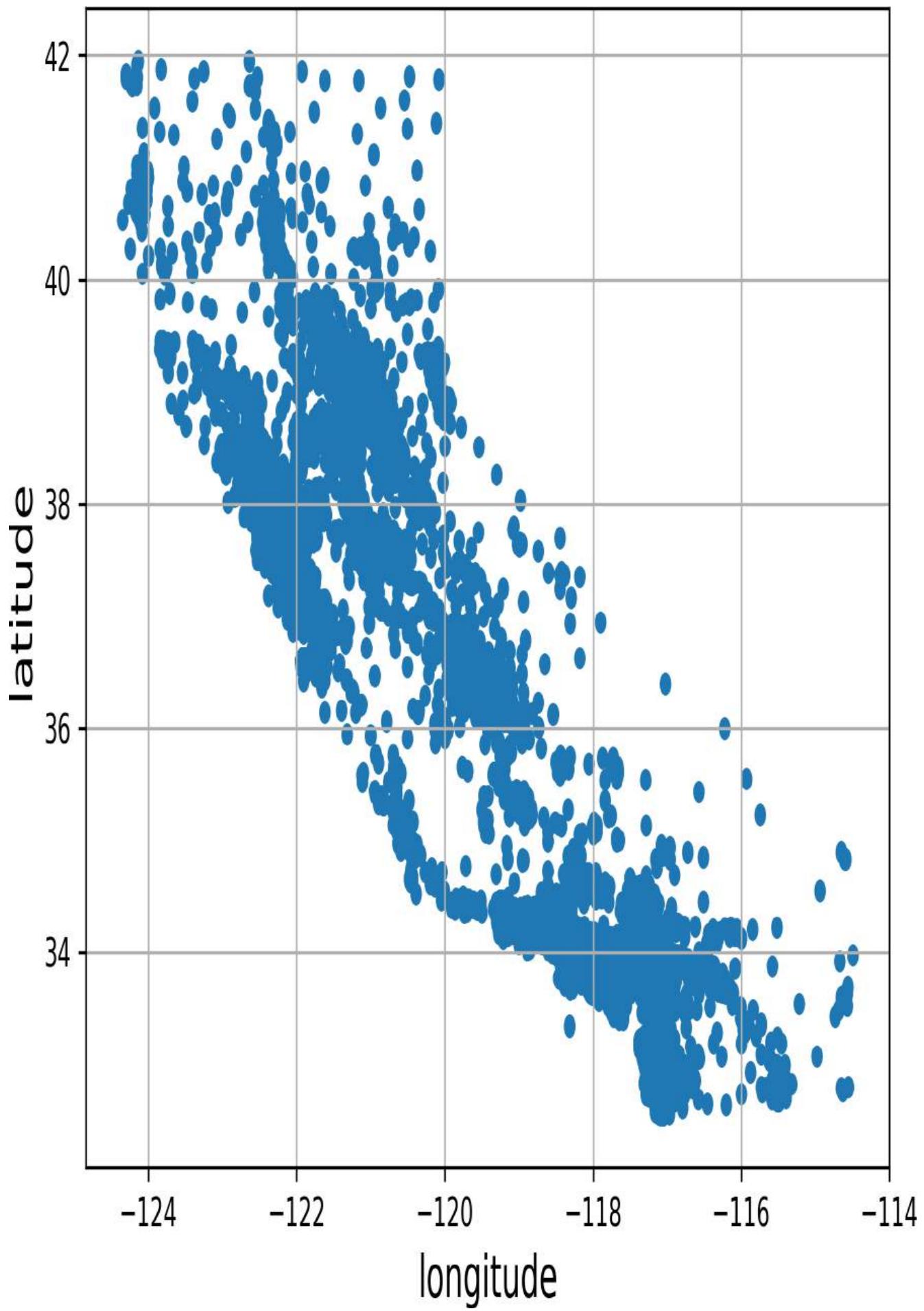
First, make sure you have put the test set aside and you are only exploring the training set. Also, if the training set is very large, you may want to sample an exploration set, to make manipulations easy and fast during the exploration phase. In this case, the training set is quite small, so you can just work directly on the full set. Since you're going to experiment with various transformations of the full training set, you should make a copy of the original so you can revert to it afterwards:

```
housing = strat_train_set.copy()
```

## Visualizing Geographical Data

Because the dataset includes geographical information (latitude and longitude), it is a good idea to create a scatterplot of all the districts to visualize the data (Figure 2-11):

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True)
plt.show()
```



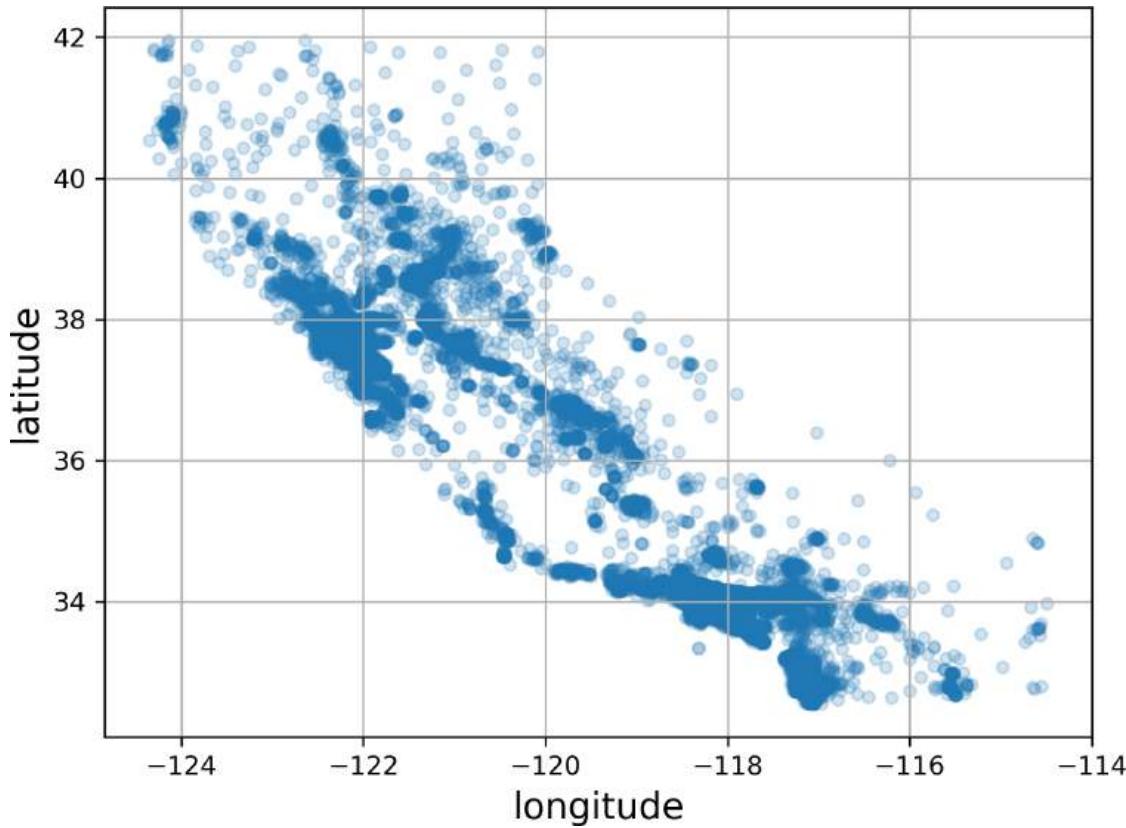
*Figure 2-11. A geographical scatterplot of the data*

This looks like California all right, but other than that it is hard to see any particular pattern. Setting the `alpha` option to `0.2` makes it much easier to visualize the places where there is a high density of data points ([Figure 2-12](#)):

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True, alpha=0.2)  
plt.show()
```

Now that's much better: you can clearly see the high-density areas, namely the Bay Area and around Los Angeles and San Diego, plus a long line of fairly high-density areas in the Central Valley (in particular, around Sacramento and Fresno).

Our brains are very good at spotting patterns in pictures, but you may need to play around with visualization parameters to make the patterns stand out.



*Figure 2-12. A better visualization that highlights high-density areas*

Next, you look at the housing prices ([Figure 2-13](#)). The radius of each circle represents the district's population (option `s`), and the color represents the price (option `c`). Here you use a predefined color map (option `cmap`) called `jet`, which ranges from blue (low values) to red (high prices):<sup>8</sup>

```
housing.plot(kind="scatter", x="longitude", y="latitude", grid=True,
             s=housing["population"] / 100, label="population",
             c="median_house_value", cmap="jet", colorbar=True,
             legend=True, sharex=False, figsize=(10, 7))
plt.show()
```

This image tells you that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density, as you probably knew already. A clustering algorithm should be useful for detecting the main cluster and for adding new features that measure the proximity to the cluster centers. The ocean proximity attribute may be useful as well, although in Northern California the housing prices in coastal districts are not too high, so it is not a simple rule.

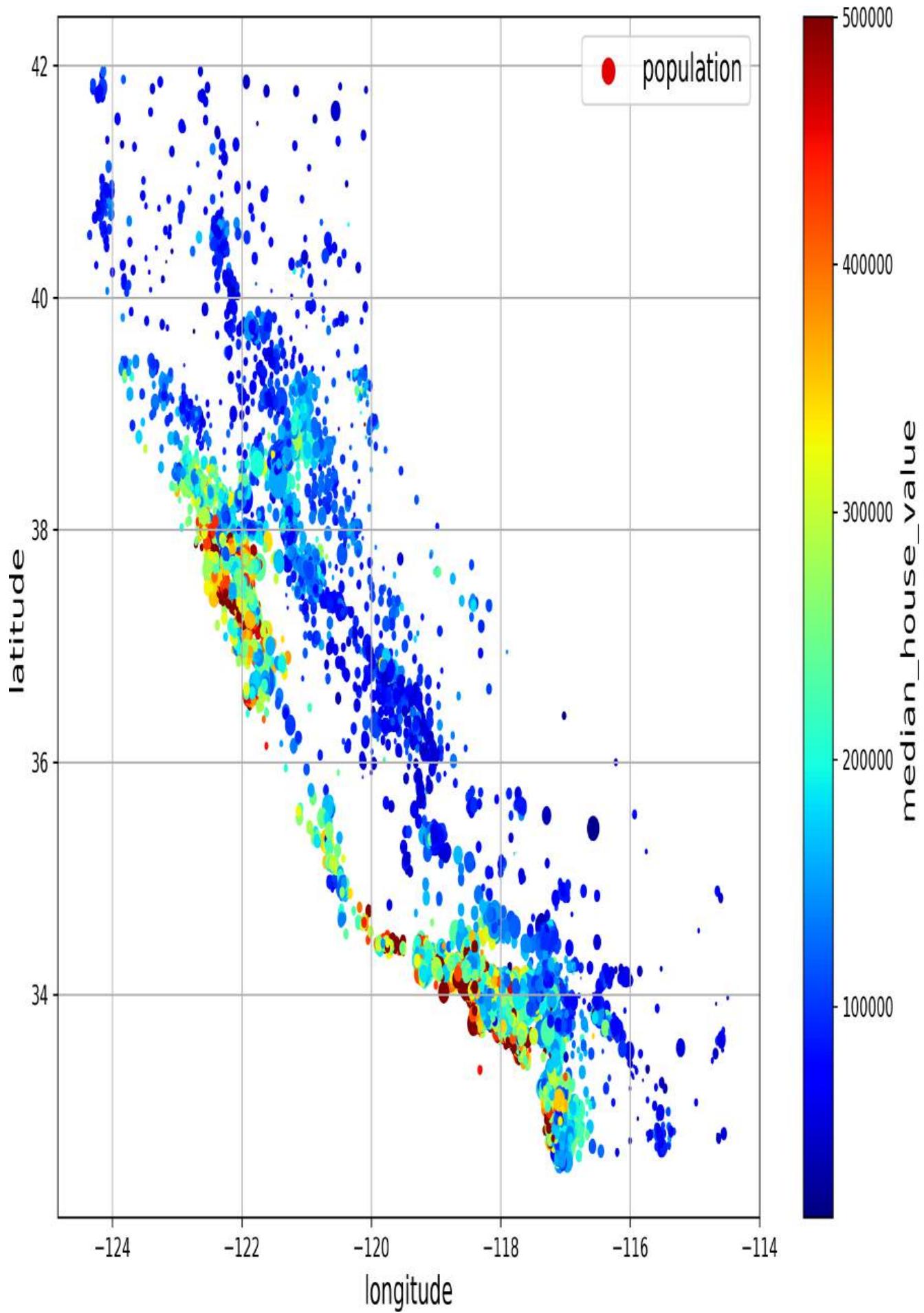


Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

## Look for Correlations

Since the dataset is not too large, you can easily compute the *standard correlation coefficient* (also called *Pearson's r*) between every pair of numerical attributes using the `corr()` method:

```
corr_matrix = housing.corr(numeric_only=True)
```

Now you can look at how much each attribute correlates with the median house value:

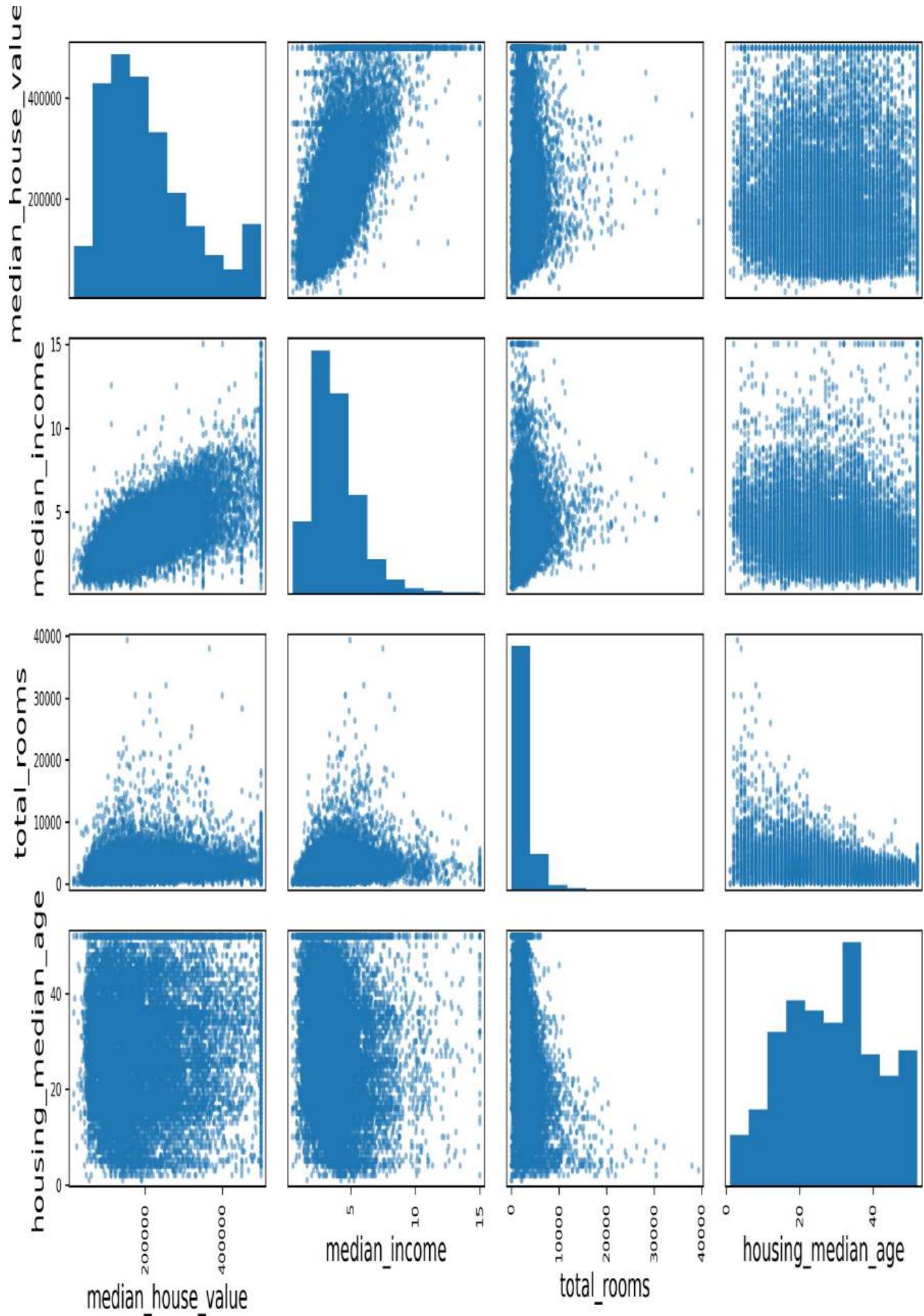
```
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.688380
total_rooms           0.137455
housing_median_age   0.102175
households            0.071426
total_bedrooms        0.054635
population            -0.020153
longitude             -0.050859
latitude              -0.139584
Name: median_house_value, dtype: float64
```

The correlation coefficient ranges from  $-1$  to  $1$ . When it is close to  $1$ , it means that there is a strong positive correlation; for example, the median house value tends to go up when the median income goes up. When the coefficient is close to  $-1$ , it means that there is a strong negative correlation; you can see a small negative correlation between the latitude and the median house value (i.e., prices have a slight tendency to go down when you go north). Finally, coefficients close to  $0$  mean that there is no linear correlation.

Another way to check for correlation between attributes is to use the Pandas `scatter_matrix()` function, which plots every numerical attribute against every other numerical attribute. Since there are now  $9$  numerical attributes, you would get  $9^2 = 81$  plots, which would not fit on a page—so you decide to focus on a few promising attributes that seem most correlated with the median housing value (Figure 2-14):

```
from pandas.plotting import scatter_matrix

attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(housing[attributes], figsize=(12, 8))
plt.show()
```

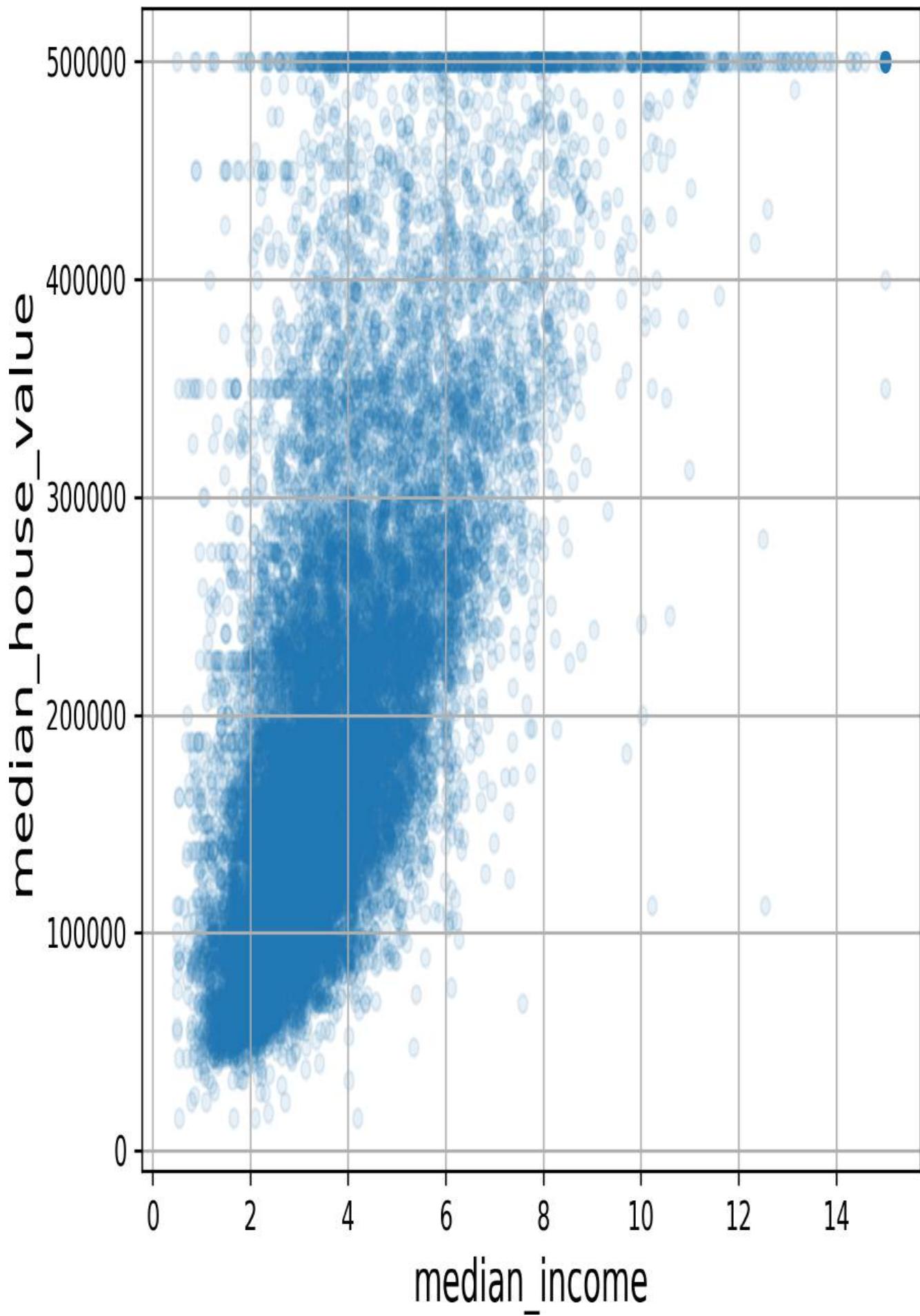


*Figure 2-14. This scatter matrix plots every numerical attribute against every other numerical attribute, plus a histogram of each numerical attribute's values on the main diagonal (top left to bottom right)*

The main diagonal would be full of straight lines if Pandas plotted each variable against itself, which would not be very useful. So instead, the Pandas displays a histogram of each attribute (other options are available; see the Pandas documentation for more details).

Looking at the correlation scatterplots, it seems like the most promising attribute to predict the median house value is the median income, so you zoom in on their scatterplot (Figure 2-15):

```
housing.plot(kind="scatter", x="median_income", y="median_house_value",
             alpha=0.1, grid=True)
plt.show()
```



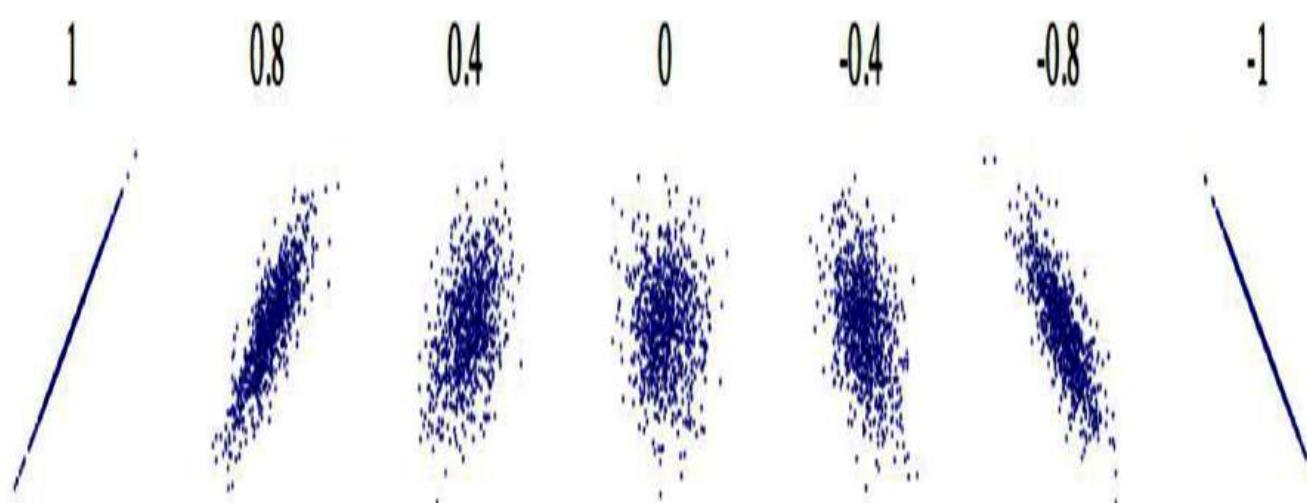
*Figure 2-15. Median income versus median house value*

This plot reveals a few things. First, the correlation is indeed quite strong; you can clearly see the upward trend, although the data is noisy. Second, the price cap you noticed earlier is clearly visible as a horizontal line at \$500,000. But the plot also reveals other less obvious straight lines: a horizontal line around \$450,000, another around \$350,000, perhaps one around \$280,000, and a few more below that. You may want to try removing the corresponding districts to prevent your algorithms from learning to reproduce these data quirks.

## **WARNING**

The correlation coefficient only measures linear correlations (“as  $x$  goes up,  $y$  generally goes up/down”). It may completely miss out on nonlinear relationships (e.g., “as  $x$  approaches 0,  $y$  generally goes up”).

[Figure 2-16](#) shows a variety of datasets along with their correlation coefficient. Note how all the plots of the bottom row have a correlation coefficient equal to 0, despite the fact that their axes are clearly *not* independent: these are examples of nonlinear relationships. Also, the second row shows examples where the correlation coefficient is equal to 1 or  $-1$ ; notice that this has nothing to do with the slope. For example, your height in inches has a correlation coefficient of 1 with your height in feet or in nanometers.



1 1 1 -1 -1 -1



0 0 0 0 0 0 0

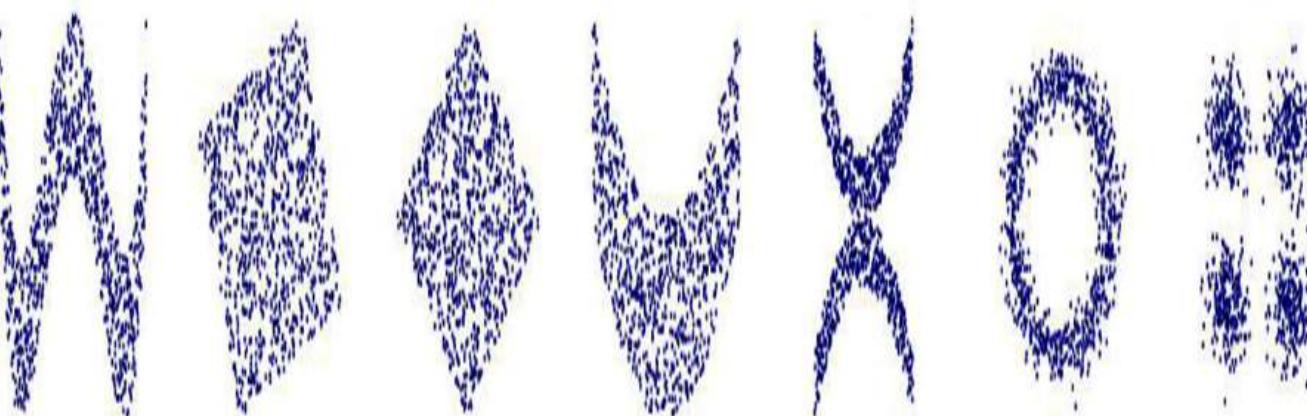


Figure 2-16. Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

## Experiment with Attribute Combinations

Hopefully the previous sections gave you an idea of a few ways you can explore the data and gain insights. You identified a few data quirks that you may want to clean up before feeding the data to a machine learning algorithm, and you found interesting correlations between attributes, in particular with the target attribute. You also noticed that some attributes have a skewed-right distribution, so you may want to transform them (e.g., by computing their logarithm or square root). Of course, your mileage will vary considerably with each project, but the general ideas are similar.

One last thing you may want to do before preparing the data for machine learning algorithms is to try out various attribute combinations. For example, the total number of rooms in a district is not very useful if you don't know how many households there are. What you really want is the number of rooms per household. Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms. And the population per household also seems like an interesting attribute combination to look at. You create these new attributes as follows:

```
housing["rooms_per_house"] = housing["total_rooms"] / housing["households"]
housing["bedrooms_ratio"] = housing["total_bedrooms"] / housing["total_rooms"]
housing["people_per_house"] = housing["population"] / housing["households"]
```

And then you look at the correlation matrix again:

```
>>> corr_matrix = housing.corr(numeric_only=True)
>>> corr_matrix["median_house_value"].sort_values(ascending=False)
median_house_value    1.000000
median_income         0.688380
rooms_per_house       0.143663
total_rooms           0.137455
housing_median_age    0.102175
households            0.071426
total_bedrooms        0.054635
population            -0.020153
people_per_house      -0.038224
longitude             -0.050859
latitude              -0.139584
bedrooms_ratio        -0.256397
Name: median_house_value, dtype: float64
```

Hey, not bad! The new `bedrooms_ratio` attribute is much more correlated with the median house value than the total number of rooms or bedrooms. It's a strong negative correlation, so it looks like houses with a lower bedroom/room ratio tend to be more expensive. The number of rooms per household is also more informative than the total number of rooms in a district—obviously the larger the houses, the more expensive they are.

### WARNING

When creating new combined features, make sure they are not too linearly correlated with existing features: *collinearity* can cause issues with some models, such as linear regression. In particular, avoid simple weighted sums of existing features.

This round of exploration does not have to be absolutely thorough; the point is to start off on the right foot and quickly gain insights that will help you get a first reasonably good prototype. But this is an iterative process: once you get a prototype up and running, you can analyze its output to gain more insights and come back to this exploration step.

## Prepare the Data for Machine Learning Algorithms

It's time to prepare the data for your machine learning algorithms. Instead of doing this manually, you should write functions for this purpose, for several good reasons:

- This will allow you to reproduce these transformations easily on any dataset (e.g., the next time you get a fresh dataset).
- You will gradually build a library of transformation functions that you can reuse in future projects.
- You can use these functions in your live system to transform the new data before feeding it to your algorithms.
- This will make it possible for you to easily try various transformations and see which combination of transformations works best.

But first, revert to a clean training set (by copying `strat_train_set` once again). You should also separate the predictors and the labels, since you don't necessarily want to

apply the same transformations to the predictors and the target values (note that `drop()` creates a copy of the data and does not affect `strat_train_set`):

```
housing = strat_train_set.drop("median_house_value", axis=1)
housing_labels = strat_train_set["median_house_value"].copy()
```

## Clean the Data

Most machine learning algorithms cannot work with missing features, so you'll need to take care of these. For example, you noticed earlier that the `total_bedrooms` attribute has some missing values. You have three options to fix this:

1. Get rid of the corresponding districts.
2. Get rid of the whole attribute.
3. Set the missing values to some value (zero, the mean, the median, etc.). This is called *imputation*.

You can accomplish these easily using the Pandas DataFrame's `dropna()`, `drop()`, and `fillna()` methods:

```
housing.dropna(subset=["total_bedrooms"], inplace=True) # option 1

housing.drop("total_bedrooms", axis=1, inplace=True) # option 2

median = housing["total_bedrooms"].median() # option 3
housing["total_bedrooms"] = housing["total_bedrooms"].fillna(median)
```

You decide to go for option 3 since it is the least destructive, but instead of the preceding code, you will use a handy Scikit-Learn class: `SimpleImputer`. The benefit is that it will store the median value of each feature: this will make it possible to impute missing values not only on the training set, but also on the validation set, the test set, and any new data fed to the model. To use it, first you need to create a `SimpleImputer` instance, specifying that you want to replace each attribute's missing values with the median of that attribute:

```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

Since the median can only be computed on numerical attributes, you then need to create a copy of the data with only the numerical attributes (this will exclude the text attribute `ocean_proximity`):

```
housing_num = housing.select_dtypes(include=[np.number])
```

Now you can fit the `imputer` instance to the training data using the `fit()` method:

```
imputer.fit(housing_num)
```

The `imputer` has simply computed the median of each attribute and stored the result in its `statistics_` instance variable. Only the `total_bedrooms` attribute had missing values, but you cannot be sure that there won't be any missing values in new data after the system goes live, so it is safer to apply the `imputer` to all the numerical attributes:

```
>>> imputer.statistics_
array([-118.51 , 34.26 , 29. , 2125. , 434. , 1167. , 408. , 3.5385])
>>> housing_num.median().values
array([-118.51 , 34.26 , 29. , 2125. , 434. , 1167. , 408. , 3.5385])
```

Now you can use this “trained” `imputer` to transform the training set by replacing missing values with the learned medians:

```
X = imputer.transform(housing_num)
```

Missing values can also be replaced with the mean value (`strategy="mean"`), or with the most frequent value (`strategy="most_frequent"`), or with a constant value (`strategy="constant"`, `fill_value=...`). The last two strategies support non-numerical data.

## TIP

There are also more powerful imputers available in the `sklearn.impute` package (both for numerical features only):

- `KNNImputer` replaces each missing value with the mean of the  $k$ -nearest neighbors' values for that feature. The distance is based on all the available features.
- `IterativeImputer` trains a regression model per feature to predict the missing values based on all the other available features. It then trains the model again on the updated data, and repeats the process several times, improving the models and the replacement values at each iteration.

## SCIKIT-LEARN DESIGN

Scikit-Learn's API is remarkably well designed. These are the [main design principles](#):<sup>9</sup>

### *Consistency*

All objects share a consistent and simple interface:

### *Estimators*

Any object that can estimate some parameters based on a dataset is called an *estimator* (e.g., a `SimpleImputer` is an estimator). The estimation itself is performed by the `fit()` method, and it takes a dataset as a parameter, or two for supervised learning algorithms—the second dataset contains the labels. Any other parameter needed to guide the estimation process is considered a hyperparameter (such as a `SimpleImputer`'s strategy), and it must be set as an instance variable (generally via a constructor parameter).

### *Transformers*

Some estimators (such as a `SimpleImputer`) can also transform a dataset; these are called *transformers*. Once again, the API is simple: the transformation is performed by the `transform()` method with the dataset to transform as a parameter. It returns the transformed dataset. This transformation generally relies on the learned parameters, as is the case for a `SimpleImputer`. All transformers also have a convenience method called `fit_transform()`, which is equivalent to calling `fit()` and then `transform()` (but sometimes `fit_transform()` is optimized and runs much faster).

### *Predictors*

Finally, some estimators, given a dataset, are capable of making predictions; they are called *predictors*. For example, the `LinearRegression` model in the previous chapter was a predictor: given a country's GDP per capita, it predicted life

satisfaction. A predictor has a `predict()` method that takes a dataset of new instances and returns a dataset of corresponding predictions. It also has a `score()` method that measures the quality of the predictions, given a test set (and the corresponding labels, in the case of supervised learning algorithms).<sup>10</sup>

### *Inspection*

All the estimator's hyperparameters are accessible directly via public instance variables (e.g., `imputer.strategy`), and all the estimator's learned parameters are accessible via public instance variables with an underscore suffix (e.g., `imputer.statistics_`).

### *Nonproliferation of classes*

Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.

### *Composition*

Existing building blocks are reused as much as possible. For example, it is easy to create a Pipeline estimator from an arbitrary sequence of transformers followed by a final estimator, as you will see.

### *Sensible defaults*

Scikit-Learn provides reasonable default values for most parameters, making it easy to quickly create a baseline working system.

Scikit-Learn transformers output NumPy arrays (or sometimes SciPy sparse matrices) even when they are fed Pandas DataFrames as input.<sup>11</sup> So, the output of `imputer.transform(housing_num)` is a NumPy array: `X` has neither column names nor index. Luckily, it's not too hard to wrap `X` in a DataFrame and recover the column names and index from `housing_num`:

```
housing_tr = pd.DataFrame(X, columns=housing_num.columns,  
                           index=housing_num.index)
```

## Handling Text and Categorical Attributes

So far we have only dealt with numerical attributes, but your data may also contain text attributes. In this dataset, there is just one: the `ocean_proximity` attribute. Let's look at its value for the first few instances:

```
>>> housing_cat = housing[["ocean_proximity"]]  
>>> housing_cat.head(8)  
ocean_proximity  
13096      NEAR BAY  
14973      <1H OCEAN  
3785        INLAND  
14689        INLAND  
20507      NEAR OCEAN  
1286        INLAND  
18078      <1H OCEAN  
4396      NEAR BAY
```

It's not arbitrary text: there are a limited number of possible values, each of which represents a category. So this attribute is a categorical attribute. Most machine learning algorithms prefer to work with numbers, so let's convert these categories from text to numbers. For this, we can use Scikit-Learn's `OrdinalEncoder` class:

```
from sklearn.preprocessing import OrdinalEncoder  
  
ordinal_encoder = OrdinalEncoder()  
housing_cat_encoded = ordinal_encoder.fit_transform(housing_cat)
```

Here's what the first few encoded values in `housing_cat_encoded` look like:

```
>>> housing_cat_encoded[:8]  
array([[3.],  
       [0.],  
       [1.],  
       [1.],  
       [4.],  
       [1.],  
       [0.],  
       [3.]])
```

You can get the list of categories using the `categories_` instance variable. It is a list containing a 1D array of categories for each categorical attribute (in this case, a list

containing a single array since there is just one categorical attribute):

```
>>> ordinal_encoder.categories_
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],
      dtype=object)]
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. This may be fine in some cases (e.g., for ordered categories such as “bad”, “average”, “good”, and “excellent”), but it is obviously not the case for the `ocean_proximity` column (for example, categories 0 and 4 are clearly more similar than categories 0 and 1). To fix this issue, a common solution is to create one binary attribute per category: one attribute equal to 1 when the category is “`<1H OCEAN`” (and 0 otherwise), another attribute equal to 1 when the category is “`INLAND`” (and 0 otherwise), and so on. This is called *one-hot encoding*, because only one attribute will be equal to 1 (hot), while the others will be 0 (cold). The new attributes are sometimes called *dummy* attributes. Scikit-Learn provides a `OneHotEncoder` class to convert categorical values into one-hot vectors:

```
from sklearn.preprocessing import OneHotEncoder

cat_encoder = OneHotEncoder()
housing_cat_1hot = cat_encoder.fit_transform(housing_cat)
```

By default, the output of a `OneHotEncoder` is a SciPy *sparse matrix*, instead of a NumPy array:

```
>>> housing_cat_1hot
<Compressed Sparse Row sparse matrix of dtype 'float64'
 with 16512 stored elements and shape (16512, 5)>
```

A sparse matrix is a very efficient representation for matrices that contain mostly zeros. Indeed, internally it only stores the nonzero values and their positions. When a categorical attribute has hundreds or thousands of categories, one-hot encoding it results in a very large matrix full of 0s except for a single 1 per row. In this case, a sparse matrix is exactly what you need: it will save plenty of memory and speed up computations. You can use a sparse matrix mostly like a normal 2D array,<sup>12</sup> but if you want to convert it to a (dense) NumPy array, just call the `toarray()` method:

```
>>> housing_cat_1hot.toarray()
array([[0., 0., 0., 1., 0.],
       [1., 0., 0., 0., 0.],
```

```
[0., 1., 0., 0., 0.],  
...,  
[0., 0., 0., 0., 1.],  
[1., 0., 0., 0., 0.],  
[0., 0., 0., 0., 1.]], shape=(16512, 5))
```

Alternatively, you can set `sparse_output=False` when creating the `OneHotEncoder`, in which case the `transform()` method will return a regular (dense) NumPy array directly:

```
cat_encoder = OneHotEncoder(sparse_output=False)  
housing_cat_1hot = cat_encoder.fit_transform(housing_cat) # now a dense array
```

As with the `OrdinalEncoder`, you can get the list of categories using the encoder's `categories_` instance variable:

```
>>> cat_encoder.categories_  
[array(['<1H OCEAN', 'INLAND', 'ISLAND', 'NEAR BAY', 'NEAR OCEAN'],  
      dtype=object)]
```

Pandas has a function called `get_dummies()`, which also converts each categorical feature into a one-hot representation, with one binary feature per category:

```
>>> df_test = pd.DataFrame({"ocean_proximity": ["INLAND", "NEAR BAY"]})  
>>> pd.get_dummies(df_test)  
ocean_proximity_INLAND  ocean_proximity_NEAR      BAY  
0                      True                    False  
1                      False                   True
```

It looks nice and simple, so why not use it instead of `OneHotEncoder`? Well, the advantage of `OneHotEncoder` is that it remembers which categories it was trained on. This is very important because once your model is in production, it should be fed exactly the same features as during training: no more, no less. Look what our trained `cat_encoder` outputs when we make it transform the same `df_test` (using `transform()`, not `fit_transform()`):

```
>>> cat_encoder.transform(df_test)  
array([[0., 1., 0., 0., 0.],  
      [0., 0., 0., 1., 0.]])
```

See the difference? `get_dummies()` saw only two categories, so it output two columns, whereas `OneHotEncoder` output one column per learned category, in the right order.

Moreover, if you feed `get_dummies()` a DataFrame containing an unknown category (e.g., "<2H OCEAN"), it will happily generate a column for it:

```
>>> df_test_unknown = pd.DataFrame({"ocean_proximity": ["<2H OCEAN", "ISLAND"]})
>>> pd.get_dummies(df_test_unknown)
   ocean_proximity_<2H OCEAN  ocean_proximity_ISLAND
0                      True                 False
1                     False                  True
```

But `OneHotEncoder` is smarter: it will detect the unknown category and raise an exception. If you prefer, you can set the `handle_unknown` hyperparameter to "`ignore`", in which case it will just represent the unknown category with zeros:

```
>>> cat_encoder.handle_unknown = "ignore"
>>> cat_encoder.transform(df_test_unknown)
array([[0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

## TIP

If a categorical attribute has a large number of possible categories (e.g., country code, profession, species), then one-hot encoding will result in a large number of input features. This may slow down training and degrade performance. If this happens, you may want to replace the categorical input with useful numerical features related to the categories: for example, you could replace the `ocean_proximity` feature with the distance to the ocean (similarly, a country code could be replaced with the country's population and GDP per capita). Alternatively, you can use one of the encoders provided by the `category_encoders` package on [GitHub](#). Or, when dealing with neural networks, you can replace each category with a learnable, low-dimensional vector called an *embedding* (see [Link to Come]). This is an example of *representation learning* (we will see more examples in [Link to Come]).

When you fit any Scikit-Learn estimator using a DataFrame, the estimator stores the column names in the `feature_names_in_` attribute. Scikit-Learn then ensures that any DataFrame fed to this estimator after that (e.g., to `transform()` or `predict()`) has the same column names. Transformers also provide a `get_feature_names_out()` method that you can use to build a DataFrame around the transformer's output:

```
>>> cat_encoder.feature_names_in_
array(['ocean_proximity'], dtype=object)
>>> cat_encoder.get_feature_names_out()
array(['ocean_proximity_<1H OCEAN', 'ocean_proximity_INLAND',
       'ocean_proximity_ISLAND', 'ocean_proximity_NEAR BAY',
       'ocean_proximity_NEAR OCEAN'], dtype=object)
>>> df_output = pd.DataFrame(cat_encoder.transform(df_test_unknown),
```

```
...           columns=cat_encoder.get_feature_names_out(),  
...           index=df_test_unknown.index)  
...
```

This feature helps avoid column mismatches, and it's also quite useful when debugging.

## Feature Scaling and Transformation

One of the most important transformations you need to apply to your data is *feature scaling*. With few exceptions, machine learning algorithms don't perform well when the input numerical attributes have very different scales. This is the case for the housing data: the total number of rooms ranges from about 6 to 39,320, while the median incomes only range from 0 to 15. Without any scaling, most models will be biased toward ignoring the median income and focusing more on the number of rooms.

There are two common ways to get all attributes to have the same scale: *min-max scaling* and *standardization*.

### WARNING

As with all estimators, it is important to fit the scalers to the training data only: never use `fit()` or `fit_transform()` for anything else than the training set. Once you have a trained scaler, you can then use it to `transform()` any other set, including the validation set, the test set, and new data. Note that while the training set values will always be scaled to the specified range, if new data contains outliers, these may end up scaled outside the range. If you want to avoid this, just set the `clip` hyperparameter to `True`.

Min-max scaling (many people call this *normalization*) is the simplest: for each attribute, the values are shifted and rescaled so that they end up ranging from 0 to 1. This is performed by subtracting the min value from all values, and dividing the results by the difference between the min and the max. Scikit-Learn provides a transformer called `MinMaxScaler` for this. It has a `feature_range` hyperparameter that lets you change the range if, for some reason, you don't want 0–1 (e.g., neural networks work best with zero-mean inputs, so a range of –1 to 1 is preferable). It's quite easy to use:

```
from sklearn.preprocessing import MinMaxScaler  
  
min_max_scaler = MinMaxScaler(feature_range=(-1, 1))  
housing_num_min_max_scaled = min_max_scaler.fit_transform(housing_num)
```

Standardization is different: first it subtracts the mean value (so standardized values have a zero mean), then it divides the result by the standard deviation (so standardized values have a standard deviation equal to 1). Unlike min-max scaling, standardization does not restrict values to a specific range. However, standardization is much less affected by outliers. For example, suppose a district has a median income equal to 100 (by mistake), instead of the usual 0–15. Min-max scaling to the 0–1 range would map this outlier down to 1 and it would crush all the other values down to 0–0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called `StandardScaler` for standardization:

```
from sklearn.preprocessing import StandardScaler  
  
std_scaler = StandardScaler()  
housing_num_std_scaled = std_scaler.fit_transform(housing_num)
```

### TIP

If you want to scale a sparse matrix without converting it to a dense matrix first, you can use a `StandardScaler` with its `with_mean` hyperparameter set to `False`: it will only divide the data by the standard deviation, without subtracting the mean (as this would break sparsity).

When a feature's distribution has a *heavy tail* (i.e., when values far from the mean are not exponentially rare), both min-max scaling and standardization will squash most values into a small range. Machine learning models generally don't like this at all, as you will see in [Chapter 4](#). So *before* you scale the feature, you should first transform it to shrink the heavy tail, and if possible to make the distribution roughly symmetrical. For example, a common way to do this for positive features with a heavy tail to the right is to replace the feature with its square root (or raise the feature to a power between 0 and 1). If the feature has a really long and heavy tail, such as a *power law distribution*, then replacing the feature with its logarithm may help. For example, the `population` feature roughly follows a power law: districts with 10,000 inhabitants are only 10 times less frequent than districts with 1,000 inhabitants, not exponentially less frequent. [Figure 2-17](#) shows how much better this feature looks when you compute its log: it's very close to a Gaussian distribution (i.e., bell-shaped).

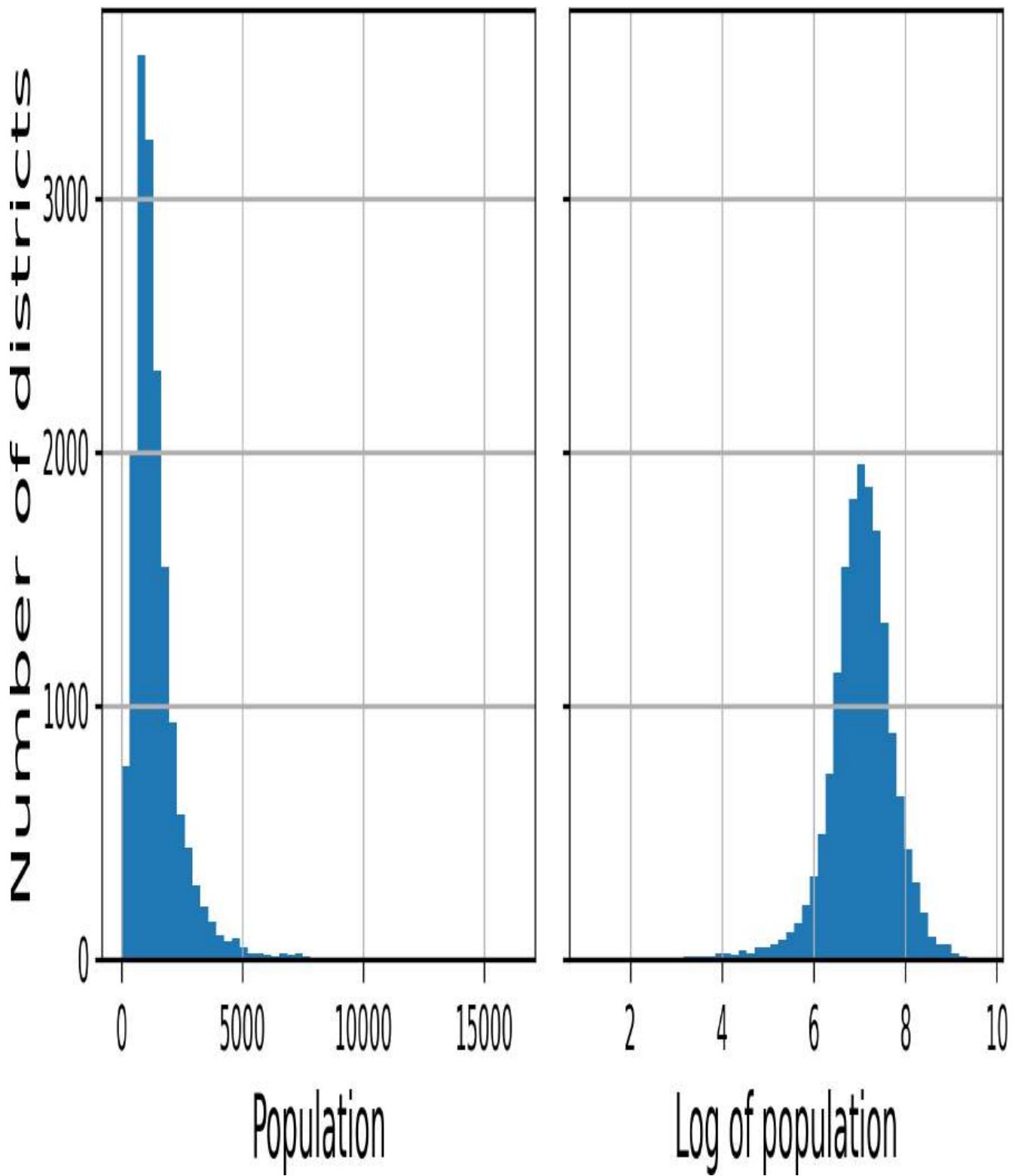


Figure 2-17. Transforming a feature to make it closer to a Gaussian distribution

Another approach to handle heavy-tailed features consists in *bucketizing* the feature. This means chopping its distribution into roughly equal-sized buckets, and replacing each feature value with the index of the bucket it belongs to, much like we did to create the `income_cat` feature (although we only used it for stratified sampling). For example, you could replace each value with its percentile. Bucketizing with equal-sized buckets

results in a feature with an almost uniform distribution, so there's no need for further scaling, or you can just divide by the number of buckets to force the values to the 0–1 range.

When a feature has a multimodal distribution (i.e., with two or more clear peaks, called *modes*), such as the `housing_median_age` feature, it can also be helpful to bucketize it, but this time treating the bucket IDs as categories, rather than as numerical values. This means that the bucket indices must be encoded, for example using a `OneHotEncoder` (so you usually don't want to use too many buckets). This approach will allow the regression model to more easily learn different rules for different ranges of this feature value. For example, perhaps houses built around 35 years ago have a peculiar style that fell out of fashion, and therefore they're cheaper than their age alone would suggest.

Another approach to transforming multimodal distributions is to add a feature for each of the modes (at least the main ones), representing the similarity between the housing median age and that particular mode. The similarity measure is typically computed using a *radial basis function* (RBF)—any function that depends only on the distance between the input value and a fixed point. The most commonly used RBF is the Gaussian RBF, whose output value decays exponentially as the input value moves away from the fixed point. For example, the Gaussian RBF similarity between the housing age  $x$  and 35 is given by the equation  $\exp(-\gamma(x - 35)^2)$ . The hyperparameter  $\gamma$  (gamma) determines how quickly the similarity measure decays as  $x$  moves away from 35. Using Scikit-Learn's `rbf_kernel()` function, you can create a new Gaussian RBF feature measuring the similarity between the housing median age and 35:

```
from sklearn.metrics.pairwise import rbf_kernel  
  
age_simil_35 = rbf_kernel(housing[["housing_median_age"]], [[35]], gamma=0.1)
```

Figure 2-18 shows this new feature as a function of the housing median age (solid line). It also shows what the feature would look like if you used a smaller `gamma` value. As the chart shows, the new age similarity feature peaks at 35, right around the spike in the housing median age distribution: if this particular age group is well correlated with lower prices, there's a good chance that this new feature will help.

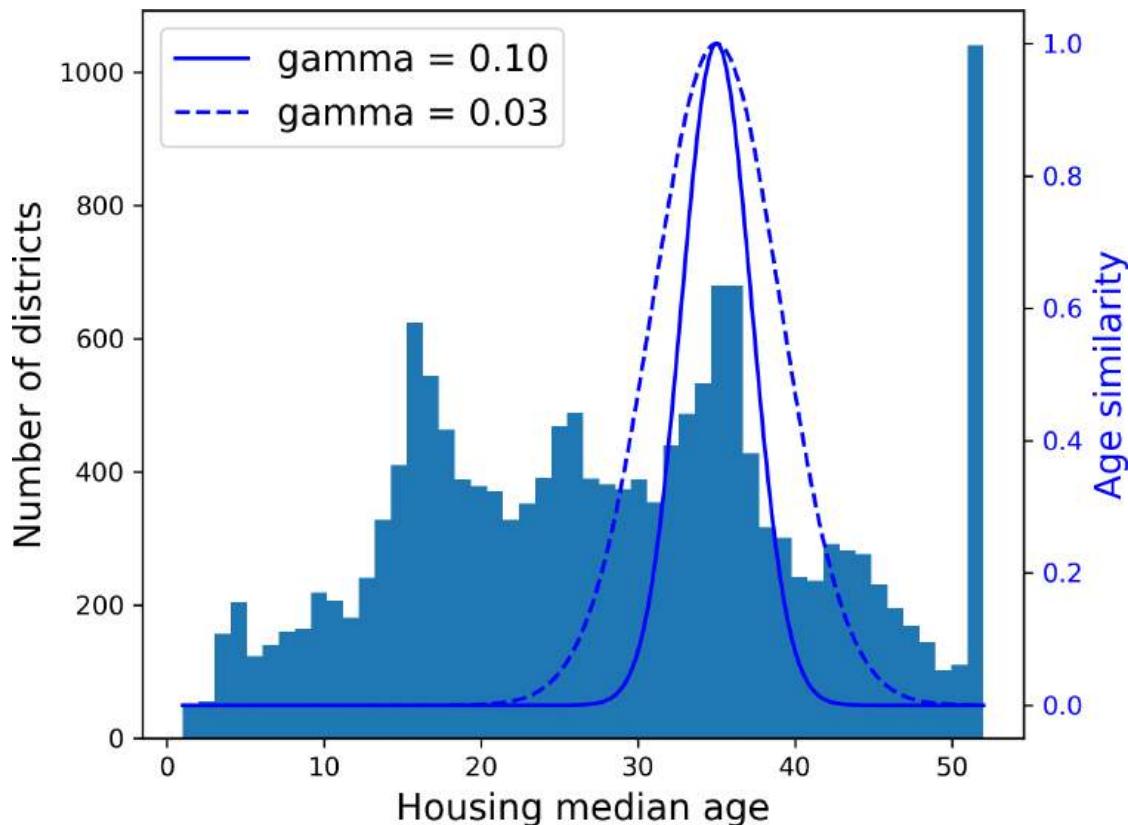


Figure 2-18. Gaussian RBF feature measuring the similarity between the housing median age and 35

So far we've only looked at the input features, but the target values may also need to be transformed. For example, if the target distribution has a heavy tail, you may choose to replace the target with its logarithm. But if you do, the regression model will now predict the *log* of the median house value, not the median house value itself. You will need to compute the exponential of the model's prediction if you want the predicted median house value.

Luckily, most of Scikit-Learn's transformers have an `inverse_transform()` method, making it easy to compute the inverse of their transformations. For example, the following code example shows how to scale the labels using a `StandardScaler` (just like we did for inputs), then train a simple linear regression model on the resulting scaled labels and use it to make predictions on some new data, which we transform back to the original scale using the trained scaler's `inverse_transform()` method. Note that we convert the labels from a Pandas Series to a DataFrame, since the `StandardScaler` expects 2D inputs. Also, in this example we just train the model on a single raw input feature (median income), for simplicity:

```
from sklearn.linear_model import LinearRegression

target_scaler = StandardScaler()
scaled_labels = target_scaler.fit_transform(housing_labels.to_frame())
```

```
model = LinearRegression()
model.fit(housing[["median_income"]], scaled_labels)
some_new_data = housing[["median_income"]].iloc[:5] # pretend this is new data

scaled_predictions = model.predict(some_new_data)
predictions = target_scaler.inverse_transform(scaled_predictions)
```

This works fine, but it's simpler and less error-prone to use a `TransformedTargetRegressor`, avoiding potential scaling mismatches. We just need to construct it, giving it the regression model and the label transformer, then fit it on the training set, using the original unscaled labels. It will automatically use the transformer to scale the labels and train the regression model on the resulting scaled labels, just like we did previously. Then, when we want to make a prediction, it will call the regression model's `predict()` method and use the scaler's `inverse_transform()` method to produce the prediction:

```
from sklearn.compose import TransformedTargetRegressor

model = TransformedTargetRegressor(LinearRegression(),
                                    transformer=StandardScaler())
model.fit(housing[["median_income"]], housing_labels)
predictions = model.predict(some_new_data)
```

## Custom Transformers

Although Scikit-Learn provides many useful transformers, you will occasionally need to write your own for tasks such as custom transformations, cleanup operations, or combining specific attributes.

For transformations that don't require any training, you can just write a function that takes a NumPy array as input and outputs the transformed array. For example, as discussed in the previous section, it's often a good idea to transform features with heavy-tailed distributions by replacing them with their logarithm (assuming the feature is positive and the tail is on the right). Let's create a log-transformer and apply it to the `population` feature:

```
from sklearn.preprocessing import FunctionTransformer

log_transformer = FunctionTransformer(np.log, inverse_func=np.exp)
log_pop = log_transformer.transform(housing[["population"]])
```

The `inverse_func` argument is optional. It lets you specify an inverse transform function, e.g., if you plan to use your transformer in a `TransformedTargetRegressor`.

Your transformation function can take hyperparameters as additional arguments. For example, here's how to create a transformer that computes the same Gaussian RBF similarity measure as earlier:

```
rbf_transformer = FunctionTransformer(rbf_kernel,
                                      kw_args=dict(Y=[[35.]], gamma=0.1))
age_simil_35 = rbf_transformer.transform(housing[["housing_median_age"]])
```

Note that there's no inverse function for the RBF kernel, since there are always two values at a given distance from a fixed point (except at distance 0). Also note that `rbf_kernel()` does not treat the features separately. If you pass it an array with two features, it will measure the 2D distance (Euclidean) to measure similarity. For example, here's how to add a feature that will measure the geographic similarity between each district and San Francisco:

```
sf_coords = 37.7749, -122.41
sf_transformer = FunctionTransformer(rbf_kernel,
                                      kw_args=dict(Y=[sf_coords], gamma=0.1))
sf_simil = sf_transformer.transform(housing[["latitude", "longitude"]])
```

Custom transformers are also useful to combine features. For example, here's a `FunctionTransformer` that computes the ratio between the input features 0 and 1:

```
>>> ratio_transformer = FunctionTransformer(lambda X: X[:, [0]] / X[:, [1]])
>>> ratio_transformer.transform(np.array([[1., 2.], [3., 4.]]))
array([[0.5 ],
       [0.75]])
```

`FunctionTransformer` is very handy, but what if you would like your transformer to be trainable, learning some parameters in the `fit()` method and using them later in the `transform()` method? For this, you need to write a custom class.

## NOTE

The rest of this section shows how to define custom transformer classes. In particular, it defines a custom transformer that we use to find the center of the 10 most expensive regions in California, and to measure the distance from each district to each of these centers, adding 10 corresponding RBF similarity features to the data. Since defining custom transformer classes is somewhat advanced, please feel free to skip to the next section and come back whenever needed.

Scikit-Learn relies on duck typing, so custom transformer classes do not have to inherit from any particular base class. All they need is three methods: `fit()` (which must return `self`), `transform()`, and `fit_transform()`. You can get `fit_transform()` for free by simply adding `TransformerMixin` as a base class: the default implementation will just call `fit()` and then `transform()`. If you add `BaseEstimator` as a base class (and avoid using `*args` and `**kwargs` in your constructor), you will also get two extra methods: `get_params()` and `set_params()`. These will be useful for automatic hyperparameter tuning.

For example, here's a custom transformer that acts much like the `StandardScaler`:

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.utils.validation import check_array, check_is_fitted

class StandardScalerClone(BaseEstimator, TransformerMixin):
    def __init__(self, with_mean=True): # no *args or **kwargs!
        self.with_mean = with_mean

    def fit(self, X, y=None): # y is required even though we don't use it
        X = check_array(X) # checks that X is an array with finite float values
        self.mean_ = X.mean(axis=0)
        self.scale_ = X.std(axis=0)
        self.n_features_in_ = X.shape[1] # every estimator stores this in fit()
        return self # always return self!

    def transform(self, X):
        check_is_fitted(self) # looks for learned attributes (with trailing _)
        X = check_array(X)
        assert self.n_features_in_ == X.shape[1]
        if self.with_mean:
            X = X - self.mean_
        return X / self.scale_
```

Here are a few things to note:

- The `sklearn.utils.validation` package contains several functions we can use to validate the inputs. For simplicity, we will skip such tests in the rest of

this book, but production code should have them.

- Scikit-Learn pipelines require the `fit()` method to have two arguments `X` and `y`, which is why we need the `y=None` argument even though we don't use `y`.
- All Scikit-Learn estimators set `n_features_in_` in the `fit()` method, and they ensure that the data passed to `transform()` or `predict()` has this number of features.
- The `fit()` method must return `self`.
- This implementation is not 100% complete: all estimators should set `feature_names_in_` in the `fit()` method when they are passed a DataFrame. Moreover, all transformers should provide a `get_feature_names_out()` method, as well as an `inverse_transform()` method when their transformation can be reversed. See the last exercise at the end of this chapter for more details.

A custom transformer can (and often does) use other estimators in its implementation. For example, the following code demonstrates a custom transformer that uses a KMeans clusterer in the `fit()` method to identify the main clusters in the training data, and then uses `rbf_kernel()` in the `transform()` method to measure how similar each sample is to each cluster center:

```
from sklearn.cluster import KMeans

class ClusterSimilarity(BaseEstimator, TransformerMixin):
    def __init__(self, n_clusters=10, gamma=1.0, random_state=None):
        self.n_clusters = n_clusters
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y=None, sample_weight=None):
        self.kmeans_ = KMeans(self.n_clusters, random_state=self.random_state)
        self.kmeans_.fit(X, sample_weight=sample_weight)
        return self # always return self!

    def transform(self, X):
        return rbf_kernel(X, self.kmeans_.cluster_centers_, gamma=self.gamma)

    def get_feature_names_out(self, names=None):
        return [f"Cluster {i} similarity" for i in range(self.n_clusters)]
```

## TIP

You can check whether your custom estimator respects Scikit-Learn's API by passing an instance to `check_estimator()` from the `sklearn.utils.estimator_checks` package. For the full API, check out <https://scikit-learn.org/stable/developers>.

As you will see in [Chapter 8](#), *k*-means is a clustering algorithm that locates clusters in the data. For example, we can use it to find the most populated regions in California. How many clusters *k*-means searches for is controlled by the `n_clusters` hyperparameter. The `fit()` method of `KMeans` supports an optional argument `sample_weight`, which lets the user specify the relative weights of the samples. For example, we can pass it the labels to ensure that *k*-means focuses more on the most expensive districts. After training, the cluster centers are available via the `cluster_centers_` attribute. *k*-means is a stochastic algorithm, meaning that it relies on randomness to locate the clusters, so if you want reproducible results, you must set the `random_state` parameter. As you can see, despite the complexity of the task, the code is fairly straightforward. Now let's use this custom transformer:

```
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
similarities = cluster_simil.fit_transform(housing[["latitude", "longitude"]],
                                         sample_weight=housing_labels)
```

This code creates a `ClusterSimilarity` transformer, setting the number of clusters to 10. Then it calls `fit_transform()` with the latitude and longitude of every district in the training set, weighting each district by its median house value. The transformer uses *k*-means to locate the clusters, then measures the Gaussian RBF similarity between each district and all 10 cluster centers. The result is a matrix with one row per district, and one column per cluster. Let's look at the first three rows, rounding to two decimal places:

```
>>> similarities[:3].round(2)
array([[0.  , 0.98, 0.  , 0.  , 0.  , 0.  , 0.13, 0.55, 0.  , 0.56],
       [0.64, 0.  , 0.11, 0.04, 0.  , 0.  , 0.  , 0.99, 0.  ],
       [0.  , 0.65, 0.  , 0.  , 0.01, 0.  , 0.49, 0.59, 0.  , 0.28]])
```

[Figure 2-19](#) shows the 10 cluster centers found by *k*-means. The districts are colored according to their geographic similarity to their closest cluster center. Notice that most clusters are located in highly populated and expensive areas.

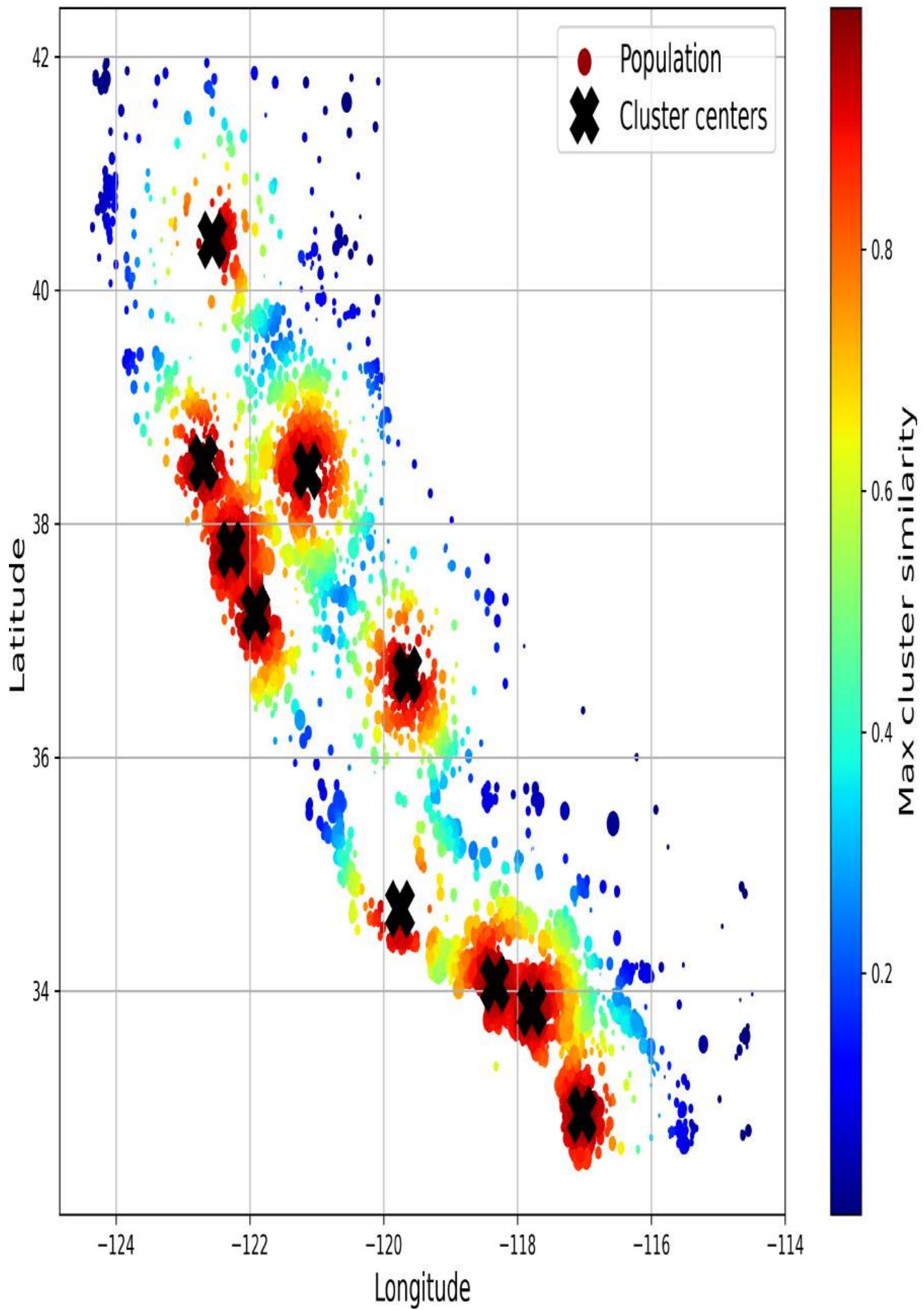


Figure 2-19. Gaussian RBF similarity to the nearest cluster center

## Transformation Pipelines

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the `Pipeline` class to help with such sequences of transformations. Here is a small pipeline for numerical attributes, which will first impute then scale the input features:

```
from sklearn.pipeline import Pipeline

num_pipeline = Pipeline([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])

```

The `Pipeline` constructor takes a list of name/estimator pairs (2-tuples) defining a sequence of steps. The names can be anything you like, as long as they are unique and don't contain double underscores (`__`). They will be useful later, when we discuss hyperparameter tuning. The estimators must all be transformers (i.e., they must have a `fit_transform()` method), except for the last one, which can be anything: a transformer, a predictor, or any other type of estimator.

### TIP

In a Jupyter notebook, if you `import sklearn` and run `sklearn.set_config(display="diagram")`, all Scikit-Learn estimators will be rendered as interactive diagrams. This is particularly useful for visualizing pipelines. To visualize `num_pipeline`, run a cell with `num_pipeline` as the last line. Clicking an estimator will show more details.

If you don't want to have to name the transformers, you can use the convenient `make_pipeline()` function instead; it takes transformers as positional arguments and creates a `Pipeline` using the names of the transformers' classes, in lowercase and without underscores (e.g., `"simpleimputer"`):

```
from sklearn.pipeline import make_pipeline

num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())
```

If multiple transformers have the same name, an index is appended to their names (e.g., "foo-1", "foo-2", etc.).

When you call the pipeline's `fit()` method, it calls `fit_transform()` sequentially on all the transformers, passing the output of each call as the parameter to the next call until it reaches the final estimator, for which it just calls the `fit()` method.

The pipeline exposes the same methods as the final estimator. In this example the last estimator is a `StandardScaler`, which is a transformer, so the pipeline also acts like a transformer. If you call the pipeline's `transform()` method, it will sequentially apply all the transformations to the data. If the last estimator were a predictor instead of a transformer, then the pipeline would have a `predict()` method rather than a `transform()` method. Calling it would sequentially apply all the transformations to the data and pass the result to the predictor's `predict()` method.

Let's call the pipeline's `fit_transform()` method and look at the output's first two rows, rounded to two decimal places:

```
>>> housing_num_prepared = num_pipeline.fit_transform(housing_num)
>>> housing_num_prepared[:2].round(2)
array([[-1.42,  1.01,  1.86,  0.31,  1.37,  0.14,  1.39, -0.94],
       [ 0.6 , -0.7 ,  0.91, -0.31, -0.44, -0.69, -0.37,  1.17]])
```

As you saw earlier, if you want to recover a nice DataFrame, you can use the pipeline's `get_feature_names_out()` method:

```
df_housing_num_prepared = pd.DataFrame(
    housing_num_prepared, columns=num_pipeline.get_feature_names_out(),
    index=housing_num.index)
```

Pipelines support indexing; for example, `pipeline[1]` returns the second estimator in the pipeline, and `pipeline[:-1]` returns a `Pipeline` object containing all but the last estimator. You can also access the estimators via the `steps` attribute, which is a list of name/estimator pairs, or via the `named_steps` dictionary attribute, which maps the names to the estimators. For example, `num_pipeline["simpleimputer"]` returns the estimator named "`simpleimputer`".

So far, we have handled the categorical columns and the numerical columns separately. It would be more convenient to have a single transformer capable of handling all columns, applying the appropriate transformations to each column. For this, you can use a `ColumnTransformer`. For example, the following `ColumnTransformer` will apply

`num_pipeline` (the one we just defined) to the numerical attributes and `cat_pipeline` to the categorical attribute:

```
from sklearn.compose import ColumnTransformer

num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
cat_attribs = ["ocean_proximity"]

cat_pipeline = make_pipeline(
    SimpleImputer(strategy="most_frequent"),
    OneHotEncoder(handle_unknown="ignore"))

preprocessing = ColumnTransformer([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs),
])

```

First we import the `ColumnTransformer` class, then we define the list of numerical and categorical column names and construct a simple pipeline for categorical attributes. Lastly, we construct a `ColumnTransformer`. Its constructor requires a list of triplets (3-tuples), each containing a name (which must be unique and not contain double underscores), a transformer, and a list of names (or indices) of columns that the transformer should be applied to.

### TIP

Instead of using a transformer, you can specify the string "drop" if you want the columns to be dropped, or you can specify "passthrough" if you want the columns to be left untouched. By default, the remaining columns (i.e., the ones that were not listed) will be dropped, but you can set the `remainder` hyperparameter to any transformer (or to "passthrough") if you want these columns to be handled differently.

Since listing all the column names is not very convenient, Scikit-Learn provides a `make_column_selector` class that you can use to automatically select all the features of a given type, such as numerical or categorical. You can pass a selector to the `ColumnTransformer` instead of column names or indices. Moreover, if you don't care about naming the transformers, you can use `make_column_transformer()`, which chooses the names for you, just like `make_pipeline()` does. For example, the following code creates the same `ColumnTransformer` as earlier, except the

transformers are automatically named "pipeline-1" and "pipeline-2" instead of "num" and "cat":

```
from sklearn.compose import make_column_selector, make_column_transformer

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)),
)
```

Now we're ready to apply this `ColumnTransformer` to the housing data:

```
housing_prepared = preprocessing.fit_transform(housing)
```

Great! We have a preprocessing pipeline that takes the entire training dataset and applies each transformer to the appropriate columns, then concatenates the transformed columns horizontally (transformers must never change the number of rows). Once again this returns a NumPy array, but you can get the column names using `preprocessing.get_feature_names_out()` and wrap the data in a nice DataFrame as we did before.

### NOTE

The `OneHotEncoder` returns a sparse matrix and the `num_pipeline` returns a dense matrix. When there is such a mix of sparse and dense matrices, the `ColumnTransformer` estimates the density of the final matrix (i.e., the ratio of nonzero cells), and it returns a sparse matrix if the density is lower than a given threshold (by default, `sparse_threshold=0.3`). In this example, it returns a dense matrix.

Your project is going really well and you're almost ready to train some models! You now want to create a single pipeline that will perform all the transformations you've experimented with up to now. Let's recap what the pipeline will do and why:

- Missing values in numerical features will be imputed by replacing them with the median, as most ML algorithms don't expect missing values. In categorical features, missing values will be replaced by the most frequent category.
- The categorical feature will be one-hot encoded, as most ML algorithms only accept numerical inputs.
- A few ratio features will be computed and added: `bedrooms_ratio`, `rooms_per_house`, and `people_per_house`. Hopefully these will better

correlate with the median house value, and thereby help the ML models.

- A few cluster similarity features will also be added. These will likely be more useful to the model than latitude and longitude.
- Features with a long tail will be replaced by their logarithm, as most models prefer features with roughly uniform or Gaussian distributions.
- All numerical features will be standardized, as most ML algorithms prefer when all features have roughly the same scale.

The code that builds the pipeline to do all of this should look familiar to you by now:

```
def column_ratio(X):
    return X[:, [0]] / X[:, [1]]

def ratio_name(function_transformer, feature_names_in):
    return ["ratio"] # feature names out

def ratio_pipeline():
    return make_pipeline(
        SimpleImputer(strategy="median"),
        FunctionTransformer(column_ratio, feature_names_out=ratio_name),
        StandardScaler())

log_pipeline = make_pipeline(
    SimpleImputer(strategy="median"),
    FunctionTransformer(np.log, feature_names_out="one-to-one"),
    StandardScaler())
cluster_simil = ClusterSimilarity(n_clusters=10, gamma=1., random_state=42)
default_num_pipeline = make_pipeline(SimpleImputer(strategy="median"),
                                     StandardScaler())
preprocessing = ColumnTransformer([
    ("bedrooms", ratio_pipeline(), ["total_bedrooms", "total_rooms"]),
    ("rooms_per_house", ratio_pipeline(), ["total_rooms", "households"]),
    ("people_per_house", ratio_pipeline(), ["population", "households"]),
    ("log", log_pipeline, ["total_bedrooms", "total_rooms", "population",
                           "households", "median_income"]),
    ("geo", cluster_simil, ["latitude", "longitude"]),
    ("cat", cat_pipeline, make_column_selector(dtype_include=object)),
],
    remainder=default_num_pipeline) # one column remaining: housing_median_age
```

If you run this `ColumnTransformer`, it performs all the transformations and outputs a NumPy array with 24 features:

```
>>> housing_prepared = preprocessing.fit_transform(housing)
>>> housing_prepared.shape
```

```
(16512, 24)
>>> preprocessing.get_feature_names_out()
array(['bedrooms_ratio', 'rooms_per_house_ratio',
       'people_per_house_ratio', 'log_total_bedrooms',
       'log_total_rooms', 'log_population', 'log_households',
       'log_median_income', 'geo_Cluster 0 similarity', [...],
       'geo_Cluster 9 similarity', 'cat_ocean_proximity_<1H OCEAN',
       'cat_ocean_proximity_INLAND', 'cat_ocean_proximity_ISLAND',
       'cat_ocean_proximity_NEAR BAY', 'cat_ocean_proximity_NEAR OCEAN',
       'remainder_housing_median_age'], dtype=object)
```

## Select and Train a Model

At last! You framed the problem, you got the data and explored it, you sampled a training set and a test set, and you wrote a preprocessing pipeline to automatically clean up and prepare your data for machine learning algorithms. You are now ready to select and train a machine learning model.

## Train and Evaluate on the Training Set

The good news is that thanks to all these previous steps, things are now going to be easy! You decide to train a very basic linear regression model to get started:

```
from sklearn.linear_model import LinearRegression

lin_reg = make_pipeline(preprocessing, LinearRegression())
lin_reg.fit(housing, housing_labels)
```

Done! You now have a working linear regression model. You try it out on the training set, looking at the first five predictions and comparing them to the labels:

```
>>> housing_predictions = lin_reg.predict(housing)
>>> housing_predictions[:5].round(-2) # -2 = rounded to the nearest hundred
array([246000., 372700., 135700., 91400., 330900.])
>>> housing_labels.iloc[:5].values
array([458300., 483800., 101700., 96100., 361800.])
```

Well, it works, but not always: the first prediction is way off (by over \$200,000!), while the other predictions are better: two are off by about 25%, and two are off by less than 10%. Remember that you chose to use the RMSE as your performance measure, so you want to measure this regression model's RMSE on the whole training set using Scikit-Learn's `root_mean_squared_error()` function:

```
>>> from sklearn.metrics import root_mean_squared_error
>>> lin_rmse = root_mean_squared_error(housing_labels, housing_predictions)
>>> lin_rmse
68972.88910758478
```

## TIP

We're not using the `score()` method here because it returns the  $R^2$  coefficient of determination instead of the RMSE. This coefficient represents the ratio of the variance in the data that the model can explain: the closer to 1 (which is the max value), the better. If the model simply predicts the mean all the time, it does not explain any part of the variance, so the model's  $R^2$  score is 0. And if the model does even worse than that, then its  $R^2$  score can be negative, and indeed arbitrarily low.

This is better than nothing, but clearly not a great score: the `median_housing_values` of most districts range between \$120,000 and \$265,000, so a typical prediction error of \$68,973 is really not very satisfying. This is an example of a model underfitting the training data. When this happens it can mean that the features do not provide enough information to make good predictions, or that the model is not powerful enough. As we saw in the previous chapter, the main ways to fix underfitting are to select a more powerful model, to feed the training algorithm with better features, or to reduce the constraints on the model. This model is not regularized, which rules out the last option. You could try to add more features, but first you want to try a more complex model to see how it does.

You decide to try a `DecisionTreeRegressor`, as this is a fairly powerful model capable of finding complex nonlinear relationships in the data (decision trees are presented in more detail in [Chapter 5](#)):

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = make_pipeline(preprocessing, DecisionTreeRegressor(random_state=42))
tree_reg.fit(housing, housing_labels)
```

Now that the model is trained, you evaluate it on the training set:

```
>>> housing_predictions = tree_reg.predict(housing)
>>> tree_rmse = root_mean_squared_error(housing_labels, housing_predictions)
>>> tree_rmse
0.0
```

Wait, what!? No error at all? Could this model really be absolutely perfect? Of course, it is much more likely that the model has badly overfit the data. How can you be sure? As you saw earlier, you don't want to touch the test set until you are ready to launch a model you are confident about, so you need to use part of the training set for training and part of it for model validation.

## Better Evaluation Using Cross-Validation

One way to evaluate the decision tree model would be to use the `train_test_split()` function to split the training set into a smaller training set and a validation set, then train your models against the smaller training set and evaluate them against the validation set. It's a bit of effort, but nothing too difficult, and it would work fairly well.

A great alternative is to use Scikit-Learn's *k-fold cross-validation* feature. You split the training set into  $k$  nonoverlapping subsets called *folds*, then you train and evaluate your model  $k$  times, picking a different fold for evaluation every time (i.e., the validation fold) and using the other  $k - 1$  folds for training. This process produces  $k$  evaluation scores (see [Figure 2-20](#)).

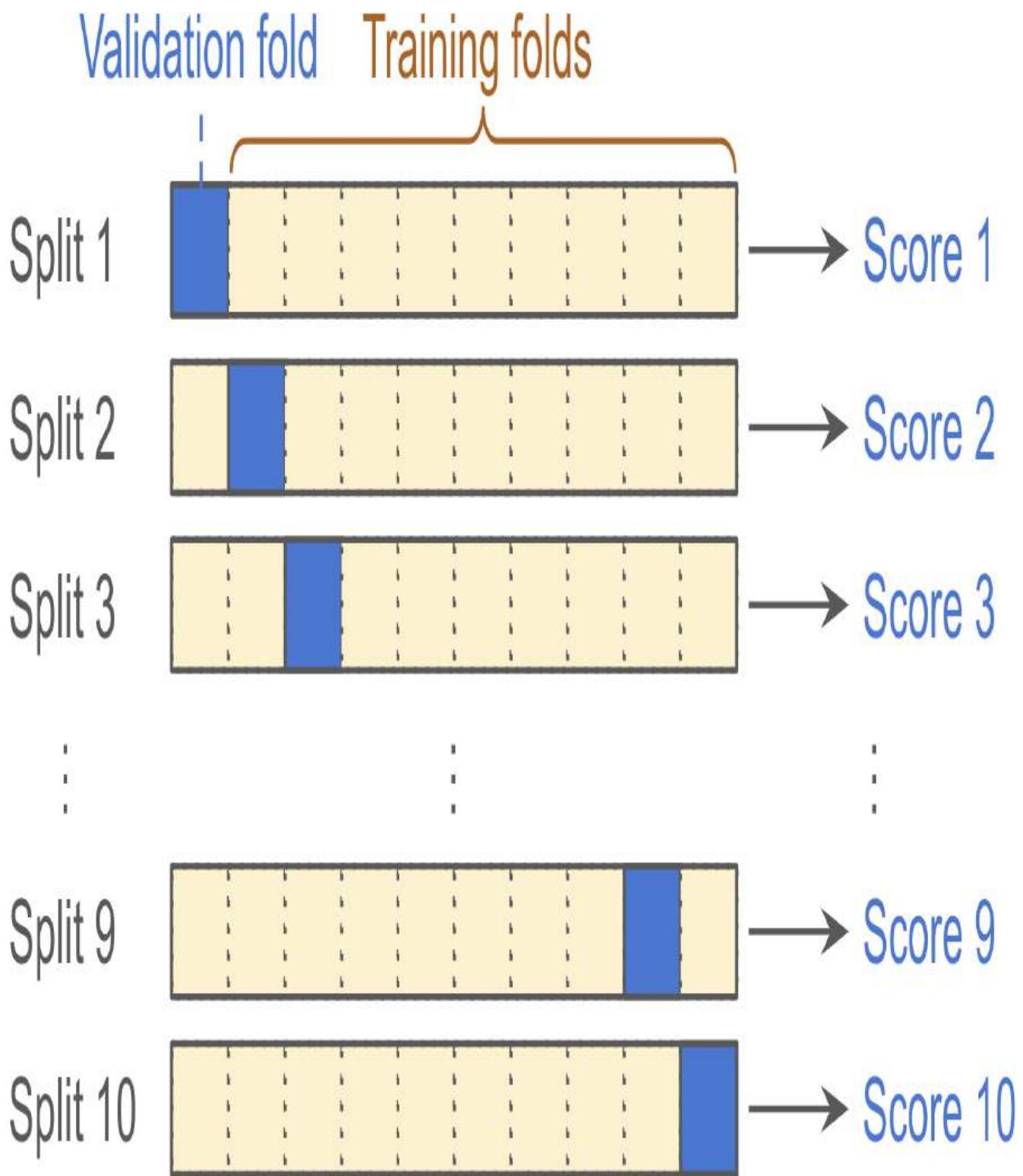


Figure 2-20.  $k$ -fold cross-validation, with  $k = 10$

Scikit-Learn provides a convenient `cross_val_score()` function that does just that, and it returns an array containing the  $k$  evaluation scores. For example, let's use it to evaluate our tree regressor, using  $k = 10$ :

```
from sklearn.model_selection import cross_val_score
```

```
tree_rmses = -cross_val_score(tree_reg, housing, housing_labels,  
                           scoring="neg_root_mean_squared_error", cv=10)
```

## WARNING

Scikit-Learn's cross-validation features expect a utility function (greater is better) rather than a cost function (lower is better), so the scoring function is actually the opposite of the RMSE. It's a negative value, so you need to switch the sign of the output to get the RMSE scores.

Let's look at the results:

```
>>> pd.Series(tree_rmses).describe()  
count      10.000000  
mean     66573.734600  
std      1103.402323  
min     64607.896046  
25%     66204.731788  
50%     66388.272499  
75%     66826.257468  
max     68532.210664  
dtype: float64
```

Now the decision tree doesn't look as good as it did earlier. In fact, it seems to perform almost as poorly as the linear regression model! Notice that cross-validation allows you to get not only an estimate of the performance of your model, but also a measure of how precise this estimate is (i.e., its standard deviation). The decision tree has an RMSE of about 66,573, with a standard deviation of about 1,103. You would not have this information if you just used one validation set. But cross-validation comes at the cost of training the model several times, so it is not always feasible.

If you compute the same metric for the linear regression model, you will find that the mean RMSE is 70,003 and the standard deviation is 4,182. So the decision tree model seems to perform very slightly better than the linear model, but the difference is minimal due to severe overfitting. We know there's an overfitting problem because the training error is low (actually zero) while the validation error is high.

Let's try one last model now: the `RandomForestRegressor`. As you will see in [Chapter 6](#), random forests work by training many decision trees on random subsets of the features, then averaging out their predictions. Such models composed of many other models are called *ensembles*: if the underlying models are very diverse, then their errors will not be very correlated, and therefore averaging out the predictions will

smooth out the errors, reduce overfitting, and improve the overall performance. The code is much the same as earlier:

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = make_pipeline(preprocessing,
                           RandomForestRegressor(random_state=42))
forest_rmses = -cross_val_score(forest_reg, housing, housing_labels,
                                 scoring="neg_root_mean_squared_error", cv=10)
```

Let's look at the scores:

```
>>> pd.Series(forest_rmses).describe()
count    10.000000
mean    47038.092799
std     1021.491757
min     45495.976649
25%    46510.418013
50%    47118.719249
75%    47480.519175
max    49140.832210
dtype: float64
```

Wow, this is much better: random forests really look very promising for this task! However, if you train a `RandomForestRegressor` and measure the RMSE on the training set, you will find roughly 17,551: that's much lower, meaning that there's still quite a lot of overfitting going on. Possible solutions are to simplify the model, constrain it (i.e., regularize it), or get a lot more training data. Before you dive much deeper into random forests, however, you should try out many other models from various categories of machine learning algorithms (e.g., several support vector machines with different kernels, and possibly a neural network), without spending too much time tweaking the hyperparameters. The goal is to shortlist a few (two to five) promising models.

## Fine-Tune Your Model

Let's assume that you now have a shortlist of promising models. You now need to fine-tune them. Let's look at a few ways you can do that.

### Grid Search

One option would be to fiddle with the hyperparameters manually, until you find a great combination of hyperparameter values. This would be very tedious work, and you may not have time to explore many combinations.

Instead, you can use Scikit-Learn's `GridSearchCV` class to search for you. All you need to do is tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values. For example, the following code searches for the best combination of hyperparameter values for the `RandomForestRegressor`:

```
from sklearn.model_selection import GridSearchCV

full_pipeline = Pipeline([
    ("preprocessing", preprocessing),
    ("random_forest", RandomForestRegressor(random_state=42)),
])
param_grid = [
    {'preprocessing_geo_n_clusters': [5, 8, 10],
     'random_forest_max_features': [4, 6, 8]},
    {'preprocessing_geo_n_clusters': [10, 15],
     'random_forest_max_features': [6, 8, 10]},
]
grid_search = GridSearchCV(full_pipeline, param_grid, cv=3,
                           scoring='neg_root_mean_squared_error')
grid_search.fit(housing, housing_labels)
```

Notice that you can refer to any hyperparameter of any estimator in a pipeline, even if this estimator is nested deep inside several pipelines and column transformers. For example, when Scikit-Learn sees "`preprocessing_geo_n_clusters`", it splits this string at the double underscores, then it looks for an estimator named "`preprocessing`" in the pipeline and finds the `preprocessing ColumnTransformer`. Next, it looks for a transformer named "`geo`" inside this `ColumnTransformer` and finds the `ClusterSimilarity` transformer we used on the latitude and longitude attributes. Then it finds this transformer's `n_clusters` hyperparameter. Similarly, `random_forest_max_features` refers to the `max_features` hyperparameter of the estimator named "`random_forest`", which is of course the `RandomForestRegressor` model (the `max_features` hyperparameter will be explained in [Chapter 6](#)).

## TIP

Wrapping preprocessing steps in a Scikit-Learn pipeline allows you to tune the preprocessing hyperparameters along with the model hyperparameters. This is a good thing since they often interact. For example, perhaps increasing `n_clusters` requires increasing `max_features` as well. If fitting the pipeline transformers is computationally expensive, you can set the pipeline's `memory` parameter to the path of a caching directory: when you first fit the pipeline, Scikit-Learn will save the fitted transformers to this directory. If you then fit the pipeline again with the same hyperparameters, Scikit-Learn will just load the cached transformers.

There are two dictionaries in this `param_grid`, so `GridSearchCV` will first evaluate all  $3 \times 3 = 9$  combinations of `n_clusters` and `max_features` hyperparameter values specified in the first `dict`, then it will try all  $2 \times 3 = 6$  combinations of hyperparameter values in the second `dict`. So in total the grid search will explore  $9 + 6 = 15$  combinations of hyperparameter values, and it will train the pipeline 3 times per combination, since we are using 3-fold cross validation. This means there will be a grand total of  $15 \times 3 = 45$  rounds of training! It may take a while, but when it is done you can get the best combination of parameters like this:

```
>>> grid_search.best_params_
{'preprocessing_geo_n_clusters': 15, 'random_forest_max_features': 6}
```

In this example, the best model is obtained by setting `n_clusters` to 15 and setting `max_features` to 6.

## TIP

Since 15 is the maximum value that was evaluated for `n_clusters`, you should probably try searching again with higher values; the score may continue to improve.

You can access the best estimator using `grid_search.best_estimator_`. If `GridSearchCV` is initialized with `refit=True` (which is the default), then once it finds the best estimator using cross-validation, it retrains it on the whole training set. This is usually a good idea, since feeding it more data will likely improve its performance.

The evaluation scores are available using `grid_search.cv_results_`. This is a dictionary, but if you wrap it in a DataFrame you get a nice list of all the test scores for each combination of hyperparameters and for each cross-validation split, as well as the mean test score across all splits:

```

>>> cv_res = pd.DataFrame(grid_search.cv_results_)
>>> cv_res.sort_values(by="mean_test_score", ascending=False, inplace=True)
>>> [...] # change column names to fit on this page, and show rmse = -score
>>> cv_res.head() # note: the 1st column is the row ID
   n_clusters  max_features  split0  split1  split2  mean_test_rmse
12          15              6    42725    43710    44335        43590
13          15              8    43487    43813    44900        44066
 7          10              6    43710    44159    44967        44278
 9          10              6    43710    44159    44967        44278
 6          10              4    43803    44072    44961        44279

```

The mean test RMSE score for the best model is 43,590, which is better than the score you got earlier using the default hyperparameter values (which was 47,038). Congratulations, you have successfully fine-tuned your best model!

## Randomized Search

The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but `RandomizedSearchCV` is often preferable, especially when the hyperparameter search space is large. This class can be used in much the same way as the `GridSearchCV` class, but instead of trying out all possible combinations it evaluates a fixed number of combinations, selecting a random value for each hyperparameter at every iteration. This may sound surprising, but this approach has several benefits:

- If some of your hyperparameters are continuous (or discrete but with many possible values), and you let randomized search run for, say, 1,000 iterations, then it will explore 1,000 different values for each of these hyperparameters, whereas grid search would only explore the few values you listed for each one.
- Suppose a hyperparameter does not actually make much difference, but you don't know it yet. If it has 10 possible values and you add it to your grid search, then training will take 10 times longer. But if you add it to a random search, it will not make any difference.
- If there are 6 hyperparameters to explore, each with 10 possible values, then grid search offers no other choice than training the model a million times, whereas random search can always run for any number of iterations you choose.

For each hyperparameter, you must provide either a list of possible values, or a probability distribution:

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

param_dists = {'preprocessing_geo_n_clusters': randint(low=3, high=50),
               'random_forest_max_features': randint(low=2, high=20)}

rnd_search = RandomizedSearchCV(
    full_pipeline, param_distributions=param_dists, n_iter=10, cv=3,
    scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(housing, housing_labels)
```

Scikit-Learn also has `HalvingRandomSearchCV` and `HalvingGridSearchCV` hyperparameter search classes. Their goal is to use the computational resources more efficiently, either to train faster or to explore a larger hyperparameter space. Here's how they work: in the first round, many hyperparameter combinations (called "candidates") are generated using either the grid approach or the random approach. These candidates are then used to train models that are evaluated using cross-validation, as usual. However, training uses limited resources, which speeds up this first round considerably. By default, "limited resources" means that the models are trained on a small part of the training set. However, other limitations are possible, such as reducing the number of training iterations if the model has a hyperparameter to set it. Once every candidate has been evaluated, only the best ones go on to the second round, where they are allowed more resources to compete. After several rounds, the final candidates are evaluated using full resources. This may save you some time tuning hyperparameters.

## Ensemble Methods

Another way to fine-tune your system is to try to combine the models that perform best. The group (or "ensemble") will often perform better than the best individual model—just like random forests perform better than the individual decision trees they rely on—especially if the individual models make very different types of errors. For example, you could train and fine-tune a  $k$ -nearest neighbors model, then create an ensemble model that just predicts the mean of the random forest prediction and that model's prediction. We will cover this topic in more detail in [Chapter 6](#).

## Analyzing the Best Models and Their Errors

You will often gain good insights on the problem by inspecting the best models. For example, the `RandomForestRegressor` can indicate the relative importance of each attribute for making accurate predictions:

```
>>> final_model = rnd_search.best_estimator_ # includes preprocessing
>>> feature_importances = final_model["random_forest"].feature_importances_
>>> feature_importances.round(2)
array([0.07, 0.05, 0.05, 0.01, 0.01, 0.01, 0.01, 0.19, [...], 0. , 0.01])
```

Let's sort these importance scores in descending order and display them next to their corresponding attribute names:

```
>>> sorted(zip(feature_importances,
...             final_model["preprocessing"].get_feature_names_out()),
...         reverse=True)
...
[(np.float64(0.18673440937412847), 'log_median_income'),
 (np.float64(0.0732445627048804), 'cat_ocean_proximity_INLAND'),
 (np.float64(0.06573054277260683), 'bedrooms_ratio'),
 (np.float64(0.05353772260040246), 'rooms_per_house_ratio'),
 (np.float64(0.04599485538141581), 'people_per_house_ratio'),
 (np.float64(0.04179425251060039), 'geo_Cluster 30 similarity'),
 (np.float64(0.026183337424678557), 'geo_Cluster 25 similarity'),
 (np.float64(0.02360324145452081), 'geo_Cluster 36 similarity'),
 [...]
 (np.float64(0.0004295477685850398), 'cat_ocean_proximity_NEAR BAY'),
 (np.float64(3.019022110267028e-05), 'cat_ocean_proximity_ISLAND')]
```

With this information, you may want to try dropping some of the less useful features (e.g., apparently only one `ocean_proximity` category is really useful, so you could try dropping the others).

### TIP

The `sklearn.feature_selection.SelectFromModel` transformer can automatically drop the least useful features for you: when you fit it, it trains a model (typically a random forest), looks at its `feature_importances_` attribute, and selects the most useful features. Then when you call `transform()`, it drops the other features.

You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem: adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.

Now is also a good time to check *model fairness*: it should not only work well on average, but also on various categories of districts, whether they're rural or urban, rich or poor, northern or southern, minority or not, etc. This requires a detailed *bias analysis*: creating subsets of your validation set for each category, and analyzing your model's performance on them. That's a lot of work, but it's important: if your model performs poorly on a whole category of districts, then it should probably not be deployed until the issue is resolved, or at least it should not be used to make predictions for that category, as it may do more harm than good.

## Evaluate Your System on the Test Set

After tweaking your models for a while, you eventually have a system that performs sufficiently well. You are ready to evaluate the final model on the test set. There is nothing special about this process; just get the predictors and the labels from your test set and run your `final_model` to transform the data and make predictions, then evaluate these predictions:

```
X_test = strat_test_set.drop("median_house_value", axis=1)
y_test = strat_test_set["median_house_value"].copy()

final_predictions = final_model.predict(X_test)

final_rmse = root_mean_squared_error(y_test, final_predictions)
print(final_rmse) # prints 41448.084299370465
```

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1% better than the model currently in production? You might want to have an idea of how precise this estimate is. For this, you can compute a 95% *confidence interval* for the generalization error using `scipy.stats.bootstrap()`. You get a fairly large interval from 39,515 to 43,706, and your previous point estimate of 41,448 is roughly in the middle of it:

```
from scipy import stats

def rmse(squared_errors):
    return np.sqrt(np.mean(squared_errors))

confidence = 0.95
squared_errors = (final_predictions - y_test) ** 2
boot_result = stats.bootstrap([squared_errors], rmse,
                             confidence_level=confidence, random_state=42)
rmse_lower, rmse_upper = boot_result.confidence_interval
```

If you do a lot of hyperparameter tuning, the performance will usually be slightly worse than what you measured using cross-validation. That's because your system ends up fine-tuned to perform well on the validation data and will likely not perform as well on unknown datasets. That's not the case in this example since the test RMSE is lower than the validation RMSE, but when it happens you must resist the temptation to tweak the hyperparameters to make the numbers look good on the test set; the improvements would be unlikely to generalize to new data.

Now comes the project prelaunch phase. Presenting your solution effectively is what sets great data scientists apart from good ones. You should create concise reports (Markdown, PDFs, slides), visualize key insights (e.g., using Matplotlib or other tools such as SeaBorn or Tableau), and tailor your message to the audience: technical for peers, high-level for stakeholders. Provide impactful and easy-to-remember statements (e.g., “the median income is the number one predictor of housing prices”). Highlight what you have learned, what worked and what did not, what assumptions were made, and what your system’s limitations are.

Your results should be reproducible (as much as possible): make the code accessible to your team (e.g., via GitHub), add a structured *README* file to guide a technical person through the installation steps. Provide clear notebooks (e.g., Jupyter) with code, explanations, and results, writing clean, well-commented code. Define a *requirements.txt* or *environment.yml* file containing all the required libraries along with their precise versions (or create a Docker image). Set seeds for random generators, and remove any other source of variability.

In this California housing example, the final performance of the system is not much better than the experts’ price estimates, which were often off by 30%, but it may still be a good idea to launch it, especially if this frees up some time for the experts so they can work on more interesting and productive tasks.

## Launch, Monitor, and Maintain Your System

Perfect, you got approval to launch! You now need to get your solution ready for production (e.g., polish the code, write documentation and tests, and so on). Then you can deploy your model to your production environment. The most basic way to do this is just to save the best model you trained, transfer the file to your production environment, and load it. To save the model, you can use the `joblib` library like this:

```
import joblib

joblib.dump(final_model, "my_california_housing_model.pkl")
```

## TIP

It's often a good idea to save every model you experiment with so that you can come back easily to any model you want. You may also save the cross-validation scores and perhaps the actual predictions on the validation set. This will allow you to easily compare scores across model types, and compare the types of errors they make.

Once your model is transferred to production, you can load it and use it. For this you must first import any custom classes and functions the model relies on (which means transferring the code to production), then load the model using `joblib` and use it to make predictions:

```
import joblib
[...] # import KMeans, BaseEstimator, TransformerMixin, rbf_kernel, etc.

def column_ratio(X): [...]
def ratio_name(function_transformer, feature_names_in): [...]
class ClusterSimilarity(BaseEstimator, TransformerMixin): [...]

final_model_reloaded = joblib.load("my_california_housing_model.pkl")

new_data = [...] # some new districts to make predictions for
predictions = final_model_reloaded.predict(new_data)
```

For example, perhaps the model will be used within a website: the user will type in some data about a new district and click the Estimate Price button. This will send a query containing the data to the web server, which will forward it to your web application, and finally your code will simply call the model's `predict()` method (you want to load the model upon server startup, rather than every time the model is used). Alternatively, you can wrap the model within a dedicated web service that your web application can query through a REST API<sup>13</sup> (see Figure 2-21). This makes it easier to upgrade your model to new versions without interrupting the main application. It also simplifies scaling, since you can start as many web services as needed and load-balance the requests coming from your web application across these web services. Moreover, it allows your web application to use any programming language, not just Python.

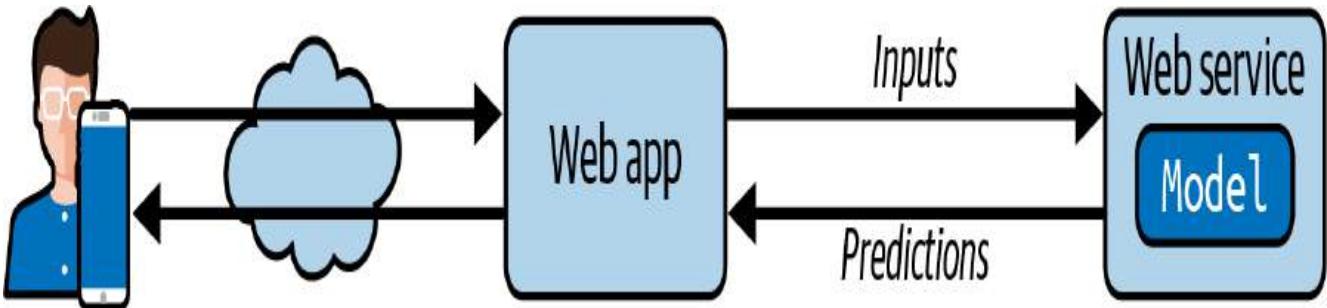


Figure 2-21. A model deployed as a web service and used by a web application

Another popular strategy is to deploy your model to the cloud, for example on Google’s Vertex AI (formerly known as Google Cloud AI Platform and Google Cloud ML Engine): just save your model using `joblib` and upload it to Google Cloud Storage (GCS), then head over to Vertex AI and create a new model version, pointing it to the GCS file. That’s it! This gives you a simple web service that takes care of load balancing and scaling for you. It takes JSON requests containing the input data (e.g., of a district) and returns JSON responses containing the predictions. You can then use this web service in your website (or whatever production environment you are using).

But deployment is not the end of the story. You also need to write monitoring code to check your system’s live performance at regular intervals and trigger alerts when it drops. It may drop very quickly, for example if a component breaks in your infrastructure, but be aware that it could also decay very slowly, which can easily go unnoticed for a long time. This is quite common because of data drift: if the model was trained with last year’s data, it may not be adapted to today’s data.

So, you need to monitor your model’s live performance. But how do you do that? Well, it depends. In some cases, the model’s performance can be inferred from downstream metrics. For example, if your model is part of a recommender system and it suggests products that the users may be interested in, then it’s easy to monitor the number of recommended products sold each day. If this number drops (compared to non-recommended products), then the prime suspect is the model. This may be because the data pipeline is broken, or perhaps the model needs to be retrained on fresh data (as we will discuss shortly).

However, you may also need human analysis to assess the model’s performance. For example, suppose you trained an image classification model (we’ll look at these in [Chapter 3](#)) to detect various product defects on a production line. How can you get an alert if the model’s performance drops, before thousands of defective products get shipped to your clients? One solution is to send to human raters a sample of all the pictures that the model classified (especially pictures that the model wasn’t so sure about). Depending on the task, the raters may need to be experts, or they could be

nonspecialists, such as workers on a crowdsourcing platform (e.g., Amazon Mechanical Turk). In some applications they could even be the users themselves, responding, for example, via surveys or repurposed captchas.<sup>14</sup>

Either way, you need to put in place a monitoring system (with or without human raters to evaluate the live model), as well as all the relevant processes to define what to do in case of failures and how to prepare for them. Unfortunately, this can be a lot of work. In fact, it is often much more work than building and training a model.

If the data keeps evolving, you will need to update your datasets and retrain your model regularly. You should probably automate the whole process as much as possible. Here are a few things you can automate:

- Collect fresh data regularly and label it (e.g., using human raters).
- Write a script to train the model and fine-tune the hyperparameters automatically. This script could run automatically, for example every day or every week, depending on your needs.
- Write another script that will evaluate both the new model and the previous model on the updated test set, and deploy the model to production if the performance has not decreased (if it did, make sure you investigate why). The script should probably test the performance of your model on various subsets of the test set, such as poor or rich districts, rural or urban districts, etc.

You should also make sure you evaluate the model's input data quality. Sometimes performance will degrade slightly because of a poor-quality signal (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale), but it may take a while before your system's performance degrades enough to trigger an alert. If you monitor your model's inputs, you may catch this earlier. For example, you could trigger an alert if more and more inputs are missing a feature, or the mean or standard deviation drifts too far from the training set, or a categorical feature starts containing new categories.

Finally, make sure you keep backups of every model you create and have the process and tools in place to roll back to a previous model quickly, in case the new model starts failing badly for some reason. Having backups also makes it possible to easily compare new models with previous ones. Similarly, you should keep backups of every version of your datasets so that you can roll back to a previous dataset if the new one ever gets corrupted (e.g., if the fresh data that gets added to it turns out to be full of outliers).

Having backups of your datasets also allows you to evaluate any model against any previous dataset.

As you can see, machine learning involves quite a lot of infrastructure. This is a very broad topic called *ML Operations* (MLOps), which deserves its own book. So don't be surprised if your first ML project takes a lot of effort and time to build and deploy to production. Fortunately, once all the infrastructure is in place, going from idea to production will be much faster.

## Try It Out!

Hopefully this chapter gave you a good idea of what a machine learning project looks like as well as showing you some of the tools you can use to train a great system. As you can see, much of the work is in the data preparation step: building monitoring tools, setting up human evaluation pipelines, and automating regular model training. The machine learning algorithms are important, of course, but it is probably preferable to be comfortable with the overall process and know three or four algorithms well rather than to spend all your time exploring advanced algorithms.

So, if you have not already done so, now is a good time to pick up a laptop, select a dataset that you are interested in, and try to go through the whole process from A to Z. A good place to start is on a competition website such as [Kaggle](#): you will have a dataset to play with, a clear goal, and people to share the experience with. Have fun!

## Exercises

The following exercises are based on this chapter's housing dataset:

1. Try a support vector machine regressor (`sklearn.svm.SVR`) with various hyperparameters, such as `kernel="linear"` (with various values for the `C` hyperparameter) or `kernel="rbf"` (with various values for the `C` and `gamma` hyperparameters). Note that support vector machines don't scale well to large datasets, so you should probably train your model on just the first 5,000 instances of the training set and use only 3-fold cross-validation, or else it will take hours. Don't worry about what the hyperparameters mean for now; these are explained in the online chapter on SVMs at <https://homl.info/svm-p>. How does the best SVR predictor perform?
2. Try replacing the `GridSearchCV` with a `RandomizedSearchCV`.

3. Try adding a `SelectFromModel` transformer in the preparation pipeline to select only the most important attributes.
4. Try creating a custom transformer that trains a  $k$ -nearest neighbors regressor (`sklearn.neighbors.KNeighborsRegressor`) in its `fit()` method, and outputs the model's predictions in its `transform()` method. Then add this feature to the preprocessing pipeline, using latitude and longitude as the inputs to this transformer. This will add a feature in the model that corresponds to the housing median price of the nearest districts.
5. Automatically explore some preparation options using `RandomizedSearchCV`.
6. Try to implement the `StandardScalerClone` class again from scratch, then add support for the `inverse_transform()` method: executing `scaler.inverse_transform(scaler.fit_transform(X))` should return an array very close to `X`. Then add support for feature names: set `feature_names_in_` in the `fit()` method if the input is a DataFrame. This attribute should be a NumPy array of column names. Lastly, implement the `get_feature_names_out()` method: it should have one optional `input_features=None` argument. If passed, the method should check that its length matches `n_features_in_`, and it should match `feature_names_in_` if it is defined; then `input_features` should be returned. If `input_features` is `None`, then the method should either return `feature_names_in_` if it is defined or `np.array(["x0", "x1", ...])` with length `n_features_in_` otherwise.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.

---

<sup>1</sup> The original dataset appeared in R. Kelley Pace and Ronald Barry, “Sparse Spatial Autoregressions”, *Statistics & Probability Letters* 33, no. 3 (1997): 291–297.

<sup>2</sup> A piece of information fed to a machine learning system is often called a *signal*, in reference to Claude Shannon’s information theory, which he developed at Bell Labs to improve telecommunications. His theory: you want a high signal-to-noise ratio.

<sup>3</sup> Recall that the transpose operator flips a column vector into a row vector (and vice versa).

<sup>4</sup> You might also need to check legal constraints, such as private fields that should never be copied to unsafe data stores.

- 5 The standard deviation is generally denoted  $\sigma$  (the Greek letter sigma), and it is the square root of the *variance*, which is the average of the squared deviation from the mean. When a feature has a bell-shaped *normal distribution* (also called a *Gaussian distribution*), which is very common, the “68-95-99.7” rule applies: about 68% of the values fall within  $1\sigma$  of the mean, 95% within  $2\sigma$ , and 99.7% within  $3\sigma$ .
- 6 You will often see people set the random seed to 42. This number has no special property, other than being the Answer to the Ultimate Question of Life, the Universe, and Everything.
- 7 The location information is actually quite coarse, and as a result many districts will have the exact same ID, so they will end up in the same set (test or train). This introduces some unfortunate sampling bias.
- 8 If you are reading this in grayscale, grab a red pen and scribble over most of the coastline from the Bay Area down to San Diego (as you might expect). You can add a patch of yellow around Sacramento as well.
- 9 For more details on the design principles, see Lars Buitinck et al., “API Design for Machine Learning Software: Experiences from the Scikit-Learn Project”, arXiv preprint arXiv:1309.0238 (2013).
- 10 Some predictors also provide methods to measure the confidence of their predictions.
- 11 If you run `sklearn.set_config(transform_output="pandas")`, all transformers will output Pandas DataFrames when they receive a DataFrame as input: Pandas in, Pandas out.
- 12 See SciPy’s documentation for more details.
- 13 In a nutshell, a REST (or RESTful) API is an HTTP-based API that follows some conventions, such as using standard HTTP verbs to read, update, create, or delete resources (GET, POST, PUT, and DELETE) and using JSON for the inputs and outputs.
- 14 A captcha is a test to ensure a user is not a robot. These tests have often been used as a cheap way to label training data.

# Chapter 3. Classification

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 3rd chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

In [Chapter 1](#) I mentioned that the most common supervised learning tasks are regression (predicting values) and classification (predicting classes). In [Chapter 2](#) we explored a regression task, predicting housing values, using various algorithms such as linear regression, decision trees, and random forests (which will be explained in further detail in later chapters). Now we will turn our attention to classification systems.

## MNIST

In this chapter we will be using the MNIST dataset, which is a set of 70,000 small images of digits handwritten by high school students and employees of the US Census Bureau. Each image is labeled with the digit it represents. This set has been studied so much that it is often called the “hello world” of machine learning: whenever people come up with a new classification algorithm they are curious to see how it will perform on MNIST, and anyone who learns machine learning tackles this dataset sooner or later. Scikit-Learn provides many helper functions to download popular datasets. MNIST is one of them. The following code fetches the MNIST dataset from OpenML.org:<sup>1</sup>

```
from sklearn.datasets import fetch_openml  
  
mnist = fetch_openml('mnist_784', as_frame=False)
```

The `sklearn.datasets` package contains mostly three types of functions: `fetch_*` functions such as `fetch_openml()` to download real-life datasets, `load_*` functions to load small toy datasets bundled with Scikit-Learn (so they don’t need to be downloaded

over the internet), and `make_*` functions to generate fake datasets, useful for tests. Generated datasets are usually returned as an (`X`, `y`) tuple containing the input data and the targets, both as NumPy arrays. Other datasets are returned as `sklearn.utils.Bunch` objects, which are dictionaries whose entries can also be accessed as attributes. They generally contain the following entries:

"DESCR"

A description of the dataset

"data"

The input data, usually as a 2D NumPy array

"target"

The labels, usually as a 1D NumPy array

The `fetch_openml()` function is a bit unusual since by default it returns the inputs as a Pandas DataFrame and the labels as a Pandas Series (unless the dataset is sparse). But the MNIST dataset contains images, and DataFrames aren't ideal for that, so it's preferable to set `as_frame=False` to get the data as NumPy arrays instead. Let's look at these arrays:

```
>>> X, y = mnist.data, mnist.target
>>> X
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]])
>>> X.shape
(70000, 784)
>>> y
array(['5', '0', '4', ..., '4', '5', '6'], dtype=object)
>>> y.shape
(70000,)
```

There are 70,000 images, and each image has 784 features. This is because each image is  $28 \times 28$  pixels, and each feature simply represents one pixel's intensity, from 0 (white) to 255 (black). Let's take a peek at one digit from the dataset (Figure 3-1). All we need to do is grab an instance's feature vector, reshape it to a  $28 \times 28$  array, and

display it using Matplotlib's `imshow()` function. We use `cmap="binary"` to get a grayscale color map where 0 is white and 255 is black:

```
import matplotlib.pyplot as plt

def plot_digit(image_data):
    image = image_data.reshape(28, 28)
    plt.imshow(image, cmap="binary")
    plt.axis("off")

some_digit = X[0]
plot_digit(some_digit)
plt.show()
```

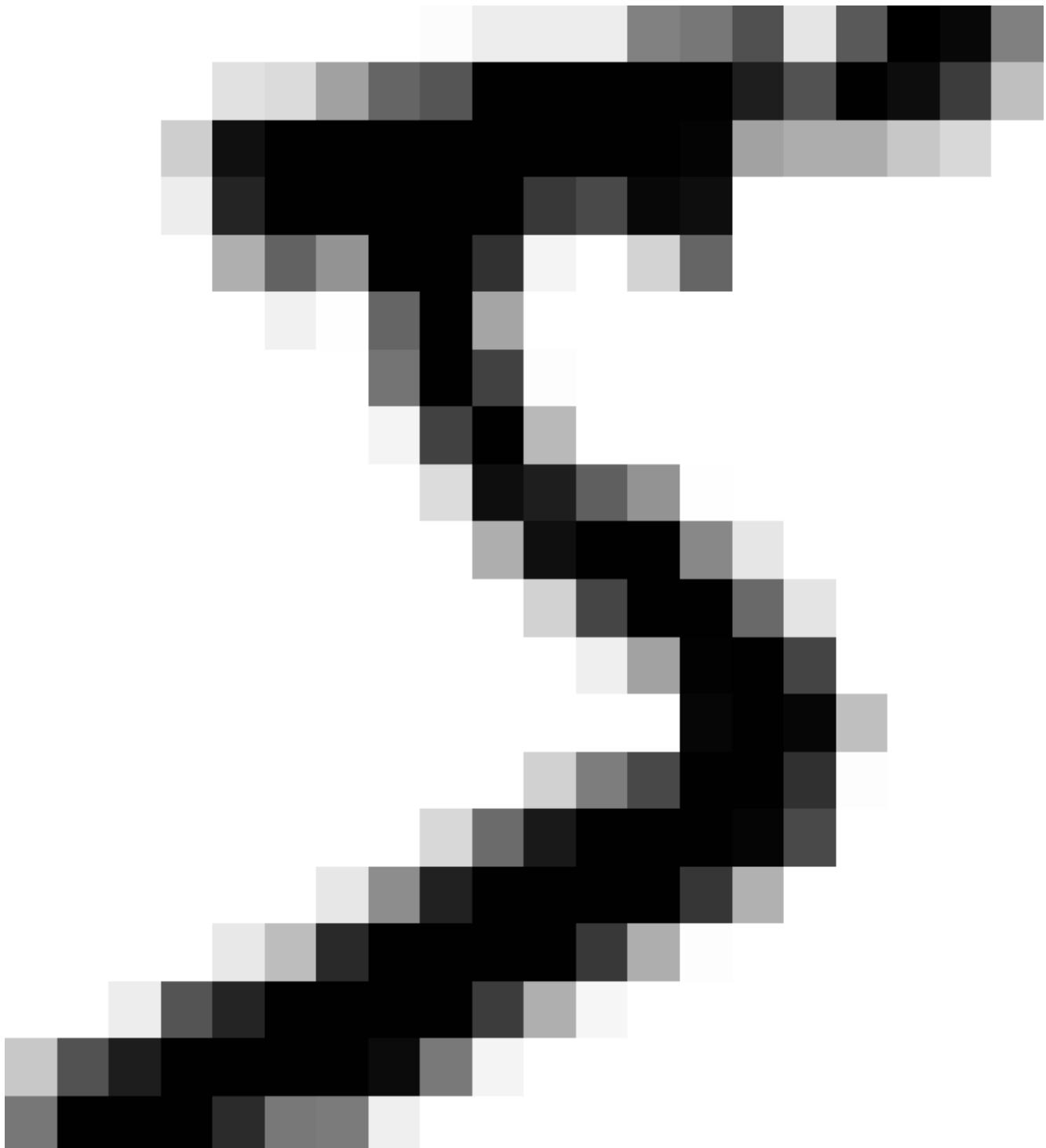


Figure 3-1. Example of an MNIST image

This looks like a 5, and indeed that's what the label tells us:

```
>>> y[0]  
'5'
```

To give you a feel for the complexity of the classification task, Figure 3-2 shows a few more images from the MNIST dataset. There's quite a large variety of digit shapes. That

said, the images are clean, well centered, not too rotated, and the digits all have roughly the same size: this dataset will not require much preprocessing (real world datasets aren't usually that friendly).

But wait! You should always create a test set and set it aside before inspecting the data closely. The MNIST dataset returned by `fetch_openml()` is actually already split into a training set (the first 60,000 images) and a test set (the last 10,000 images):<sup>2</sup>

```
X_train, X_test, y_train, y_test = X[:60000], X[60000:], y[:60000], y[60000:]
```

The training set is already shuffled for us, which is good because this guarantees that all cross-validation folds will be similar (we don't want one fold to be missing some digits). Moreover, some learning algorithms are sensitive to the order of the training instances, and they perform poorly if they get many similar instances in a row. Shuffling the dataset ensures that this won't happen.<sup>3</sup>

5041921314  
3536172869  
4091124327  
3869056076  
1819398593  
3074980941  
4460456100  
1716302117  
8026783904  
6746807831

*Figure 3-2. Digits from the MNIST dataset*

## Training a Binary Classifier

Let's simplify the problem for now and only try to identify one digit—for example, the number 5. This “5-detector” will be an example of a *binary classifier*, capable of distinguishing between just two classes, 5 and non-5. First we'll create the target vectors for this classification task:

```
y_train_5 = (y_train == '5') # True for all 5s, False for all other digits
y_test_5 = (y_test == '5')
```

Now let's pick a classifier and train it. A good place to start is with a *stochastic gradient descent* (SGD, or stochastic GD) classifier, using Scikit-Learn's `SGDClassifier` class. This classifier is capable of handling very large datasets efficiently. This is in part because SGD deals with training instances independently, one at a time, which also makes SGD well suited for online learning, as you will see later. Let's create an `SGDClassifier` and train it on the whole training set:

```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X_train, y_train_5)
```

Now we can use it to detect images of the number 5:

```
>>> sgd_clf.predict([some_digit])
array([ True])
```

The classifier guesses that this image represents a 5 (`True`). Looks like it guessed right in this particular case! Now, let's evaluate this model's performance.

## Performance Measures

Evaluating a classifier is often significantly trickier than evaluating a regressor, so we will spend a large part of this chapter on this topic. There are many performance measures available, so grab another coffee and get ready to learn a bunch of new concepts and acronyms!

## Measuring Accuracy Using Cross-Validation

A good way to evaluate a model is to use cross-validation, just as you did in [Chapter 2](#). Let's use the `cross_val_score()` function to evaluate our `SGDClassifier` model, using  $k$ -fold cross-validation with three folds. Remember that  $k$ -fold cross-validation means splitting the training set into  $k$  folds (in this case, three), then training the model  $k$  times, holding out a different fold each time for evaluation (see [Chapter 2](#)):

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train_5, y_train_5, cv=3, scoring="accuracy")
array([0.95035, 0.96035, 0.9604])
```

Wow! Above 95% accuracy (ratio of correct predictions) on all cross-validation folds? This looks amazing, doesn't it? Well, before you get too excited, let's look at a dummy classifier that just classifies every single image in the most frequent class, which in this case is the negative class (i.e., *non 5*):

```
from sklearn.dummy import DummyClassifier

dummy_clf = DummyClassifier()
dummy_clf.fit(X_train, y_train_5)
print(any(dummy_clf.predict(X_train))) # prints False: no 5s detected
```

Can you guess this model's accuracy? Let's find out:

```
>>> cross_val_score(dummy_clf, X_train, y_train_5, cv=3, scoring="accuracy")
array([0.90965, 0.90965, 0.90965])
```

That's right, it has over 90% accuracy! This is simply because only about 10% of the images are 5s, so if you always guess that an image is *not* a 5, you will be right about 90% of the time. Beats Nostradamus.

This demonstrates why accuracy is generally not the preferred performance measure for classifiers, especially when you are dealing with *skewed datasets* (i.e., when some classes are much more frequent than others). A much better way to evaluate the performance of a classifier is to look at the *confusion matrix* (CM).

## IMPLEMENTING CROSS-VALIDATION

Occasionally you will need more control over the cross-validation process than what Scikit-Learn provides off the shelf. In these cases, you can implement cross-validation yourself. The following code does roughly the same thing as Scikit-Learn's `cross_val_score()` function, and it prints the same result:

```
from sklearn.model_selection import StratifiedKFold
from sklearn.base import clone

skfolds = StratifiedKFold(n_splits=3) # add shuffle=True if the dataset is
# not already shuffled
for train_index, test_index in skfolds.split(X_train, y_train_5):
    clone_clf = clone(sgd_clf)
    X_train_folds = X_train[train_index]
    y_train_folds = y_train_5[train_index]
    X_test_fold = X_train[test_index]
    y_test_fold = y_train_5[test_index]

    clone_clf.fit(X_train_folds, y_train_folds)
    y_pred = clone_clf.predict(X_test_fold)
    n_correct = sum(y_pred == y_test_fold)
    print(n_correct / len(y_pred)) # prints 0.95035, 0.96035, and 0.9604
```

The `StratifiedKFold` class performs stratified sampling (as explained in [Chapter 2](#)) to produce folds that contain a representative ratio of each class. At each iteration the code creates a clone of the classifier, trains that clone on the training folds, and makes predictions on the test fold. Then it counts the number of correct predictions and outputs the ratio of correct predictions.

## Confusion Matrices

The general idea of a confusion matrix is to count the number of times instances of class A are classified as class B, for all A/B pairs. For example, to know the number of times the classifier confused images of 8s with 0s, you would look at row #8, column #0 of the confusion matrix.

To compute the confusion matrix, you first need to have a set of predictions so that they can be compared to the actual targets. You could make predictions on the test set, but it's best to keep that untouched for now (remember that you want to use the test set only at the very end of your project, once you have a classifier that you are ready to launch). Instead, you can use the `cross_val_predict()` function:

```

from sklearn.model_selection import cross_val_predict

y_train_pred = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3)

```

Just like the `cross_val_score()` function, `cross_val_predict()` performs  $k$ -fold cross-validation, but instead of returning the evaluation scores, it returns the predictions made on each test fold. This means that you get a clean prediction for each instance in the training set (by “clean” I mean “out-of-sample”: the model makes predictions on data that it never saw during training).

Now you are ready to get the confusion matrix using the `confusion_matrix()` function. Just pass it the target classes (`y_train_5`) and the predicted classes (`y_train_pred`):

```

>>> from sklearn.metrics import confusion_matrix
>>> cm = confusion_matrix(y_train_5, y_train_pred)
>>> cm
array([[53892,    687],
       [ 1891,  3530]])

```

Each row in a confusion matrix represents an *actual class*, while each column represents a *predicted class*. The first row of this matrix considers non-5 images (the *negative class*): 53,892 of them were correctly classified as non-5s (they are called *true negatives*), while the remaining 687 were wrongly classified as 5s (*false positives*, also called *type I errors*). The second row considers the images of 5s (the *positive class*): 1,891 were wrongly classified as non-5s (*false negatives*, also called *type II errors*), while the remaining 3,530 were correctly classified as 5s (*true positives*). A perfect classifier would only have true positives and true negatives, so its confusion matrix would have nonzero values only on its main diagonal (top left to bottom right):

```

>>> y_train_perfect_predictions = y_train_5 # pretend we reached perfection
>>> confusion_matrix(y_train_5, y_train_perfect_predictions)
array([[54579,     0],
       [     0, 5421]])

```

The confusion matrix gives you a lot of information, but sometimes you may prefer a more concise metric. An interesting one to look at is the accuracy of the positive predictions; this is called the *precision* of the classifier (Equation 3-1).

*Equation 3-1. Precision*

$$\text{precision} = \frac{TP}{TP + FP}$$

$TP$  is the number of true positives, and  $FP$  is the number of false positives.

Now consider a model that only makes positive predictions when it's extremely confident. Let's push this to the extreme and suppose that it always makes negative predictions, except for a single positive prediction on the instance it's most confident about. If this one prediction is correct, then the classifier has 100% precision ( $\text{precision} = 1/1 = 100\%$ ). Obviously, such a classifier would not be very useful, since it would ignore all but one positive instance. For this reason, precision is typically used along with another metric named *recall*, also called *sensitivity* or the *true positive rate* (TPR): this is the ratio of positive instances that are correctly detected by the classifier (Equation 3-2).

*Equation 3-2. Recall*

$$\text{recall} = \frac{TP}{TP + FN}$$

$FN$  is, of course, the number of false negatives.

If you are confused about the confusion matrix, Figure 3-3 may help.

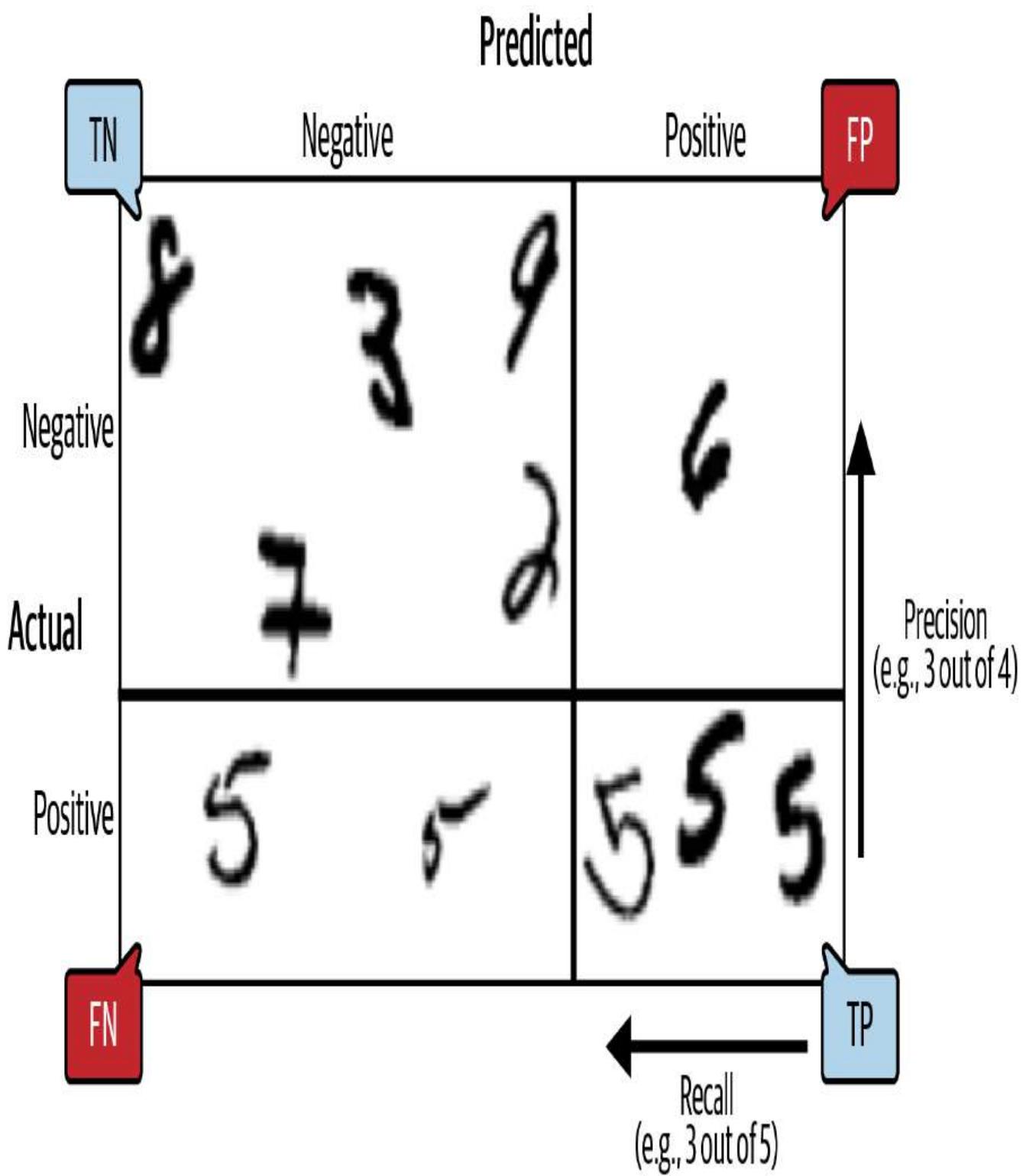


Figure 3-3. An illustrated confusion matrix showing examples of true negatives (top left), false positives (top right), false negatives (lower left), and true positives (lower right)

## Precision and Recall

Scikit-Learn provides several functions to compute classifier metrics, including precision and recall:

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred) # == 3530 / (687 + 3530)
0.8370879772350012
>>> recall_score(y_train_5, y_train_pred) # == 3530 / (1891 + 3530)
0.6511713705958311
```

Now our 5-detector does not look as shiny as it did when we looked at its accuracy. When it claims an image represents a 5, it is correct only 83.7% of the time. Moreover, it only detects 65.1% of the 5s.

It is often convenient to combine precision and recall into a single metric called the  $F_1$  score, especially when you need a single metric to compare two classifiers. The  $F_1$  score is the *harmonic mean* of precision and recall (Equation 3-3). Whereas the regular mean treats all values equally, the harmonic mean gives much more weight to low values. As a result, the classifier will only get a high  $F_1$  score if both recall and precision are high.

*Equation 3-3.  $F_1$  score*

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

To compute the  $F_1$  score, simply call the `f1_score()` function:

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7325171197343846
```

The  $F_1$  score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall. For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a much higher recall but lets a few really bad videos show up in your product (in such cases, you may even want to add a human pipeline to check the classifier's video selection). On the other hand, suppose you train a classifier to detect shoplifters in surveillance images: it is probably fine if your classifier only has 30% precision as long as it has 99% recall. Sure, the security guards will get a few false alerts, but almost all shoplifters will get caught. Similarly, medical diagnosis usually requires a high recall to avoid missing anything important. False positives can be ruled out by follow-up medical tests.

Unfortunately, you can't have it both ways: increasing precision reduces recall, and vice versa. This is called the *precision/recall trade-off*.

## The Precision/Recall Trade-off

To understand this trade-off, let's look at how the `SGDClassifier` makes its classification decisions. For each instance, it computes a score based on a *decision function*. If that score is greater than a threshold, it assigns the instance to the positive class; otherwise it assigns it to the negative class. [Figure 3-4](#) shows a few digits positioned from the lowest score on the left to the highest score on the right. Suppose the *decision threshold* is positioned at the central arrow (between the two 5s): you will find 4 true positives (actual 5s) on the right of that threshold, and 1 false positive (actually a 6). Therefore, with that threshold, the precision is 80% (4 out of 5). But out of 6 actual 5s, the classifier only detects 4, so the recall is 67% (4 out of 6). If you raise the threshold (move it to the arrow on the right), the false positive (the 6) becomes a true negative, thereby increasing the precision (up to 100% in this case), but one true positive becomes a false negative, decreasing recall down to 50%. Conversely, lowering the threshold increases recall and reduces precision.

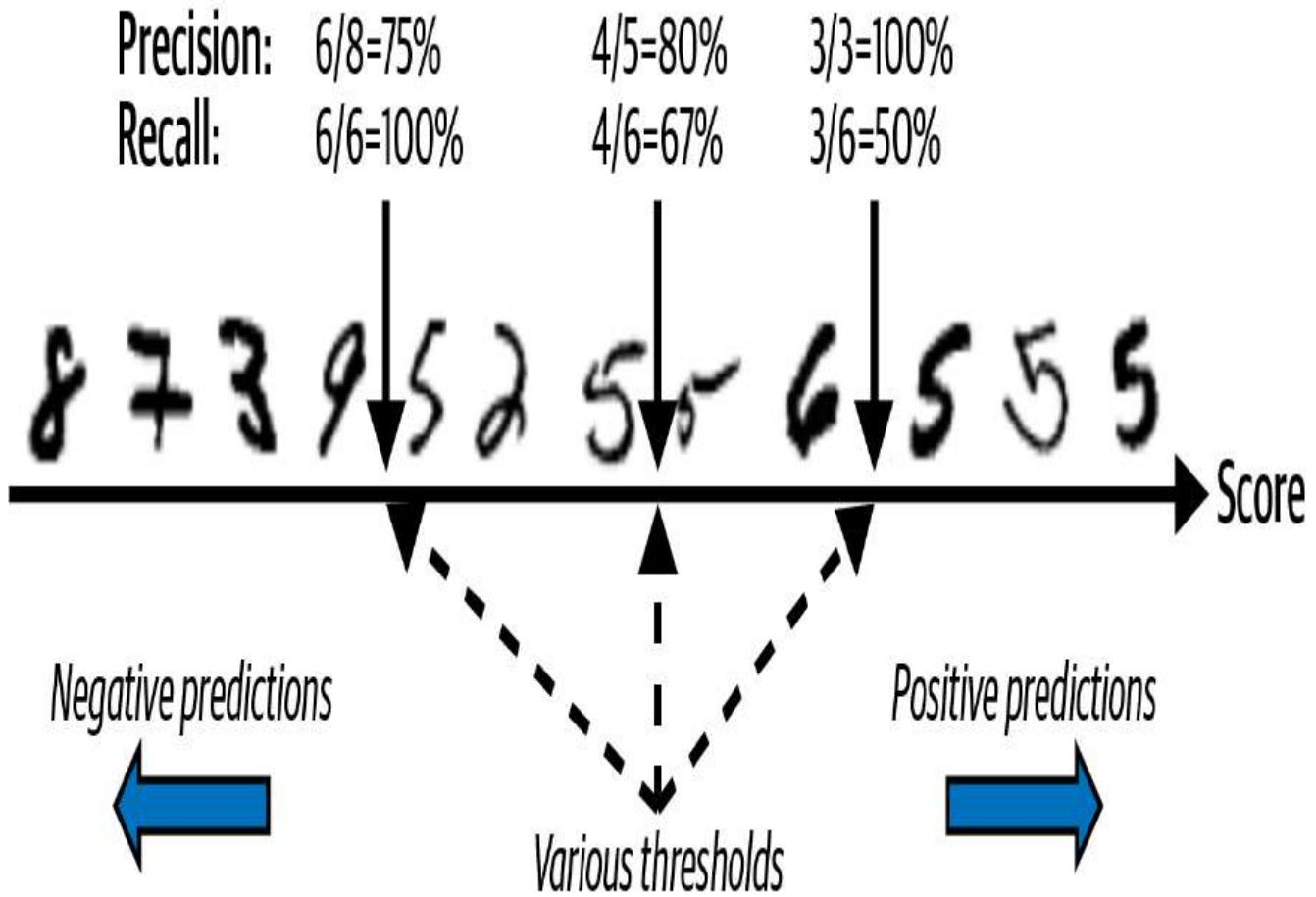


Figure 3-4. The precision/recall trade-off: images are ranked by their classifier score, and those above the chosen decision threshold are considered positive; the higher the threshold, the lower the recall, but (in general) the higher the precision

Instead of calling the classifier's `predict()` method, you can call its `decision_function()` method, which returns a score for each instance. You can then use any threshold you want to make predictions based on those scores:

```
>>> y_scores = sgd_clf.decision_function([some_digit])
>>> y_scores
array([2164.22030239])
>>> threshold = 0
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([ True])
```

The `SGDClassifier` uses a threshold equal to 0, so the preceding code returns the same result as the `predict()` method (i.e., `True`). Let's raise the threshold:

```
>>> threshold = 3000
>>> y_some_digit_pred = (y_scores > threshold)
>>> y_some_digit_pred
array([False])
```

This confirms that raising the threshold decreases recall. The image actually represents a 5, and the classifier detects it when the threshold is 0, but it misses it when the threshold is increased to 3,000.

How do you decide which threshold to use? One option is to use the `cross_val_predict()` function to get the scores of all instances in the training set, but this time specify that you want to return decision scores instead of predictions:

```
y_scores = cross_val_predict(sgd_clf, X_train, y_train_5, cv=3,
                             method="decision_function")
```

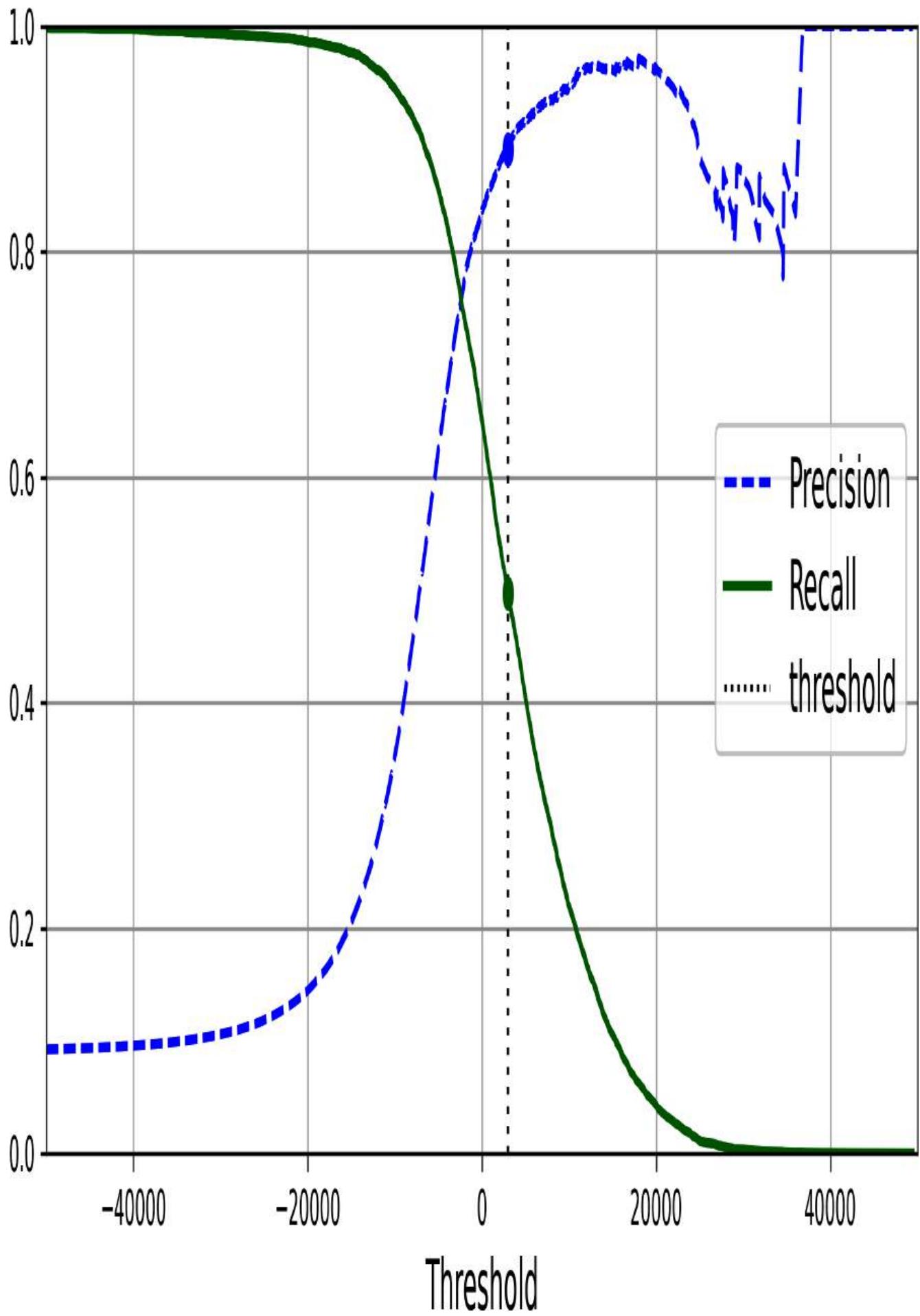
With these scores, use the `precision_recall_curve()` function to compute precision and recall for all possible thresholds (the function adds a last precision of 1 and a last recall of 0, corresponding to an infinite threshold):

```
from sklearn.metrics import precision_recall_curve

precisions, recalls, thresholds = precision_recall_curve(y_train_5, y_scores)
```

Finally, use Matplotlib to plot precision and recall as functions of the threshold value ([Figure 3-5](#)). Let's show the threshold of 3,000 we selected:

```
plt.plot(thresholds, precisions[:-1], "b--", label="Precision", linewidth=2)
plt.plot(thresholds, recalls[:-1], "g-", label="Recall", linewidth=2)
plt.vlines(threshold, 0, 1.0, "k", "dotted", label="threshold")
[...] # beautify the figure: add grid, legend, axis, labels, and circles
plt.show()
```



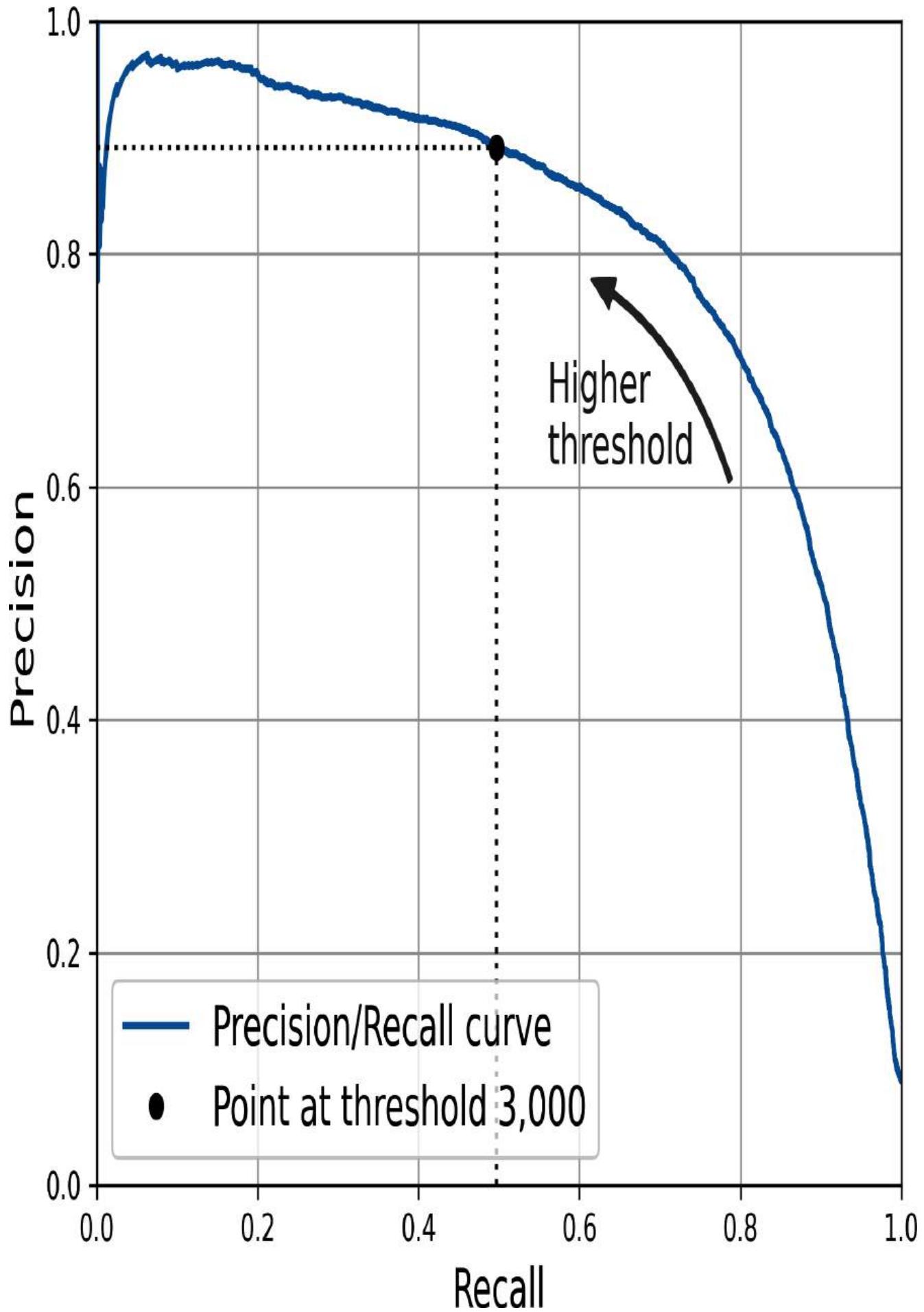
*Figure 3-5. Precision and recall versus the decision threshold*

## NOTE

You may wonder why the precision curve is bumpier than the recall curve in [Figure 3-5](#). The reason is that precision may sometimes go down when you raise the threshold (although in general it will go up). To understand why, look back at [Figure 3-4](#) and notice what happens when you start from the central threshold and move it just one digit to the right: precision goes from  $4/5$  (80%) down to  $3/4$  (75%). On the other hand, recall can only go down when the threshold is increased, which explains why its curve looks smooth.

At this threshold value, precision is near 90% and recall is around 50%. Another way to select a good precision/recall trade-off is to plot precision directly against recall, as shown in [Figure 3-6](#) (the same threshold is shown):

```
plt.plot(recalls, precisions, linewidth=2, label="Precision/Recall curve")
[...] # beautify the figure: add labels, grid, legend, arrow, and text
plt.show()
```



*Figure 3-6. Precision versus recall*

You can see that precision really starts to fall sharply at around 80% recall. You will probably want to select a precision/recall trade-off just before that drop—for example, at around 60% recall. But of course, the choice depends on your project.

Suppose you decide to aim for 90% precision. You could use the first plot to find the threshold you need to use, but that's not very precise. Alternatively, you can search for the lowest threshold that gives you at least 90% precision. For this, you can use the NumPy array's `argmax()` method. This returns the first index of the maximum value, which in this case means the first `True` value:

```
>>> idx_for_90_precision = (precisions >= 0.90).argmax()
>>> threshold_for_90_precision = thresholds[idx_for_90_precision]
>>> threshold_for_90_precision
3370.0194991439557
```

To make predictions (on the training set for now), instead of calling the classifier's `predict()` method, you can run this code:

```
y_train_pred_90 = (y_scores >= threshold_for_90_precision)
```

Let's check these predictions' precision and recall:

```
>>> precision_score(y_train_5, y_train_pred_90)
0.9000345901072293
>>> recall_at_90_precision = recall_score(y_train_5, y_train_pred_90)
>>> recall_at_90_precision
0.4799852425751706
```

Great, you have a 90% precision classifier! As you can see, it is fairly easy to create a classifier with virtually any precision you want: just set a high enough threshold, and you're done. But wait, not so fast—a high-precision classifier is not very useful if its recall is too low! For many applications, 48% recall wouldn't be great at all.

**TIP**

If someone says, “Let's reach 99% precision”, you should ask, “At what recall?”

Since Scikit-Learn 1.5, there are two new classes you can use to more easily adjust the decision threshold:

- The `FixedThresholdClassifier` class lets you wrap a binary classifier and set the desired threshold manually. If the underlying classifier has a `predict_proba()` method, then the threshold should be a value between 0 and 1 (the default is 0.5). Otherwise, it should be a decision score, comparable to the output of the model’s `decision_function()` (the default is 0).
- The `TunedThresholdClassifierCV` class uses  $k$ -fold cross-validation to automatically find the optimal threshold for a given metric. By default, it tries to find the threshold that maximizes the model’s *balanced accuracy*: that’s the average of each class’s recall. However, you can select another metric to optimize for (see the documentation for the full list of options).

## The ROC Curve

The *receiver operating characteristic* (ROC) curve is another common tool used with binary classifiers. It is very similar to the precision/recall curve, but instead of plotting precision versus recall, the ROC curve plots the *true positive rate* (another name for recall) against the *false positive rate* (FPR). The FPR (also called the *fall-out*) is the ratio of negative instances that are incorrectly classified as positive. It is equal to  $1 -$  the *true negative rate* (TNR), which is the ratio of negative instances that are correctly classified as negative. The TNR is also called *specificity*. Hence, the ROC curve plots *sensitivity* (recall) versus  $1 - \text{specificity}$ .

To plot the ROC curve, you first use the `roc_curve()` function to compute the TPR and FPR for various threshold values:

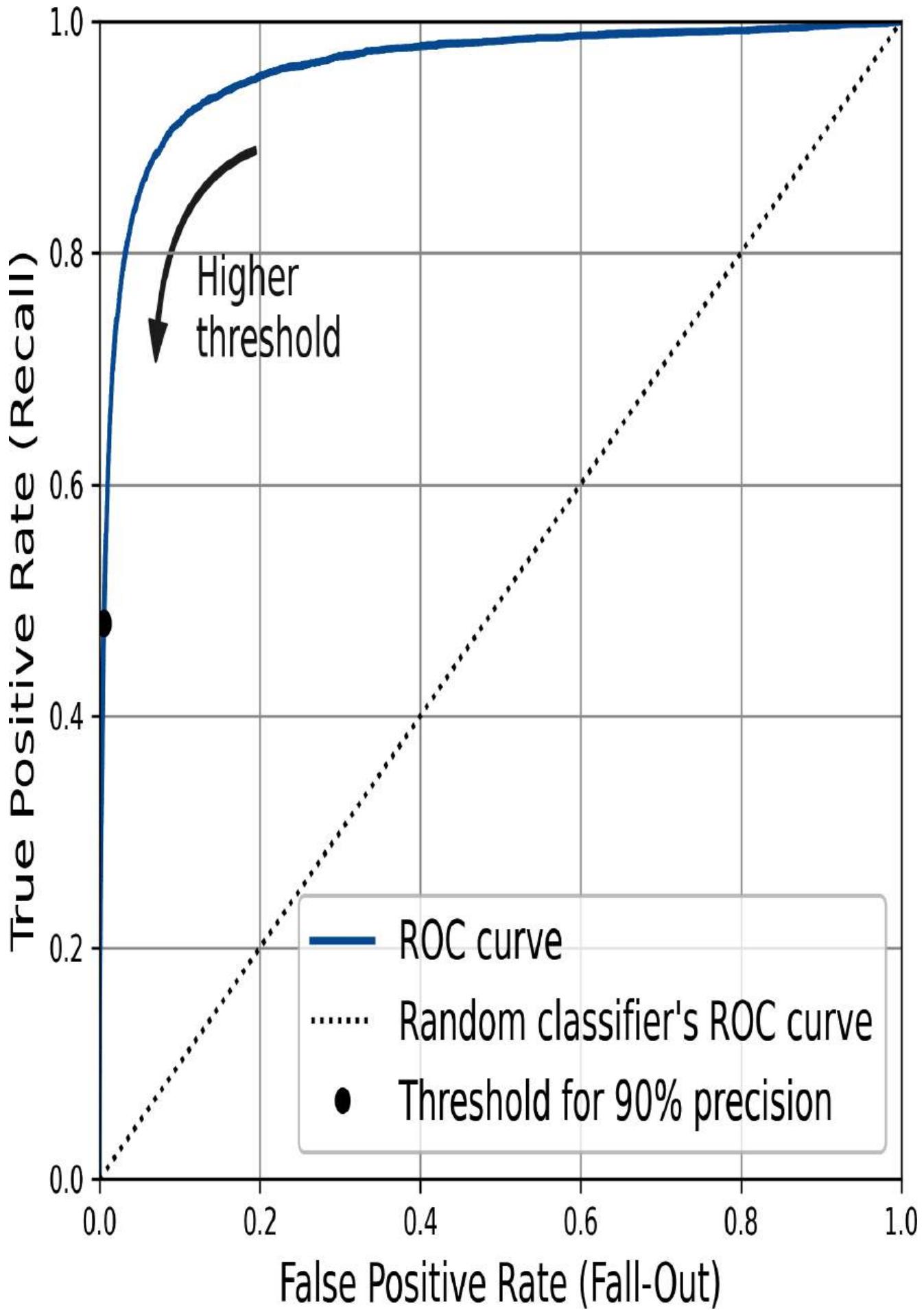
```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_train_5, y_scores)
```

Then you can plot the FPR against the TPR using Matplotlib. The following code produces the plot in [Figure 3-7](#). To find the point that corresponds to 90% precision, we need to look for the index of the desired threshold. Since thresholds are listed in decreasing order in this case, we use `<=` instead of `>=` on the first line:

```
idx_for_threshold_at_90 = (thresholds <= threshold_for_90_precision).argmax()
tpr_90, fpr_90 = tpr[idx_for_threshold_at_90], fpr[idx_for_threshold_at_90]

plt.plot(fpr, tpr, linewidth=2, label="ROC curve")
plt.plot([0, 1], [0, 1], 'k:', label="Random classifier's ROC curve")
plt.plot([fpr_90], [tpr_90], "ko", label="Threshold for 90% precision")
```

```
[...] # beautify the figure: add labels, grid, legend, arrow, and text  
plt.show()
```



*Figure 3-7. A ROC curve plotting the false positive rate against the true positive rate for all possible thresholds; the black circle highlights the chosen ratio (at 90% precision and 48% recall)*

Once again there is a trade-off: the higher the recall (TPR), the more false positives (FPR) the classifier produces. The dotted line represents the ROC curve of a purely random classifier; a good classifier stays as far away from that line as possible (toward the top-left corner).

One way to compare classifiers is to measure the *area under the curve* (AUC). A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5. Scikit-Learn provides a function to estimate the ROC AUC:

```
>>> from sklearn.metrics import roc_auc_score  
>>> roc_auc_score(y_train_5, y_scores)  
0.9604938554008616
```

### TIP

Since the ROC curve is so similar to the precision/recall (PR) curve, you may wonder how to decide which one to use. As a rule of thumb, you should prefer the PR curve whenever the positive class is rare or when you care more about the false positives than the false negatives. Otherwise, use the ROC curve. For example, looking at the previous ROC curve (and the ROC AUC score), you may think that the classifier is really good. But this is mostly because there are few positives (5s) compared to the negatives (non-5s). In contrast, the PR curve makes it clear that the classifier has room for improvement: the curve could really be closer to the top-right corner (see [Figure 3-6](#) again).

Let's now create a `RandomForestClassifier`, whose PR curve and  $F_1$  score we can compare to those of the `SGDClassifier`:

```
from sklearn.ensemble import RandomForestClassifier  
  
forest_clf = RandomForestClassifier(random_state=42)
```

The `precision_recall_curve()` function expects labels and scores for each instance, so we need to train the random forest classifier and make it assign a score to each instance. But the `RandomForestClassifier` class does not have a `decision_function()` method, due to the way it works (we will cover this in [Chapter 6](#)). Luckily, it has a `predict_proba()` method that returns estimated class probabilities for each instance, and we can just use the probability of the positive class as a score, so `precision_recall_curve()` will work.<sup>4</sup> We can call the

`cross_val_predict()` function to train the `RandomForestClassifier` using cross-validation and make it predict class probabilities for every image as follows:

```
y_probas_forest = cross_val_predict(forest_clf, X_train, y_train_5, cv=3,
                                     method="predict_proba")
```

Let's look at the estimated class probabilities for the first two images in the training set:

```
>>> y_probas_forest[:2]
array([[0.11, 0.89],
       [0.99, 0.01]])
```

The model predicts that the first image is positive with 89% probability, and it predicts that the second image is negative with 99% probability. Since each image is either positive or negative, the estimated probabilities in each row add up to 100%.

## WARNING

These are *estimated* probabilities, not actual probabilities. For example, if you look at all the images that the model classified as positive with an estimated probability between 50% and 60%, roughly 94% of them are actually positive. So, the model's estimated probabilities were much too low in this case—but models can be overconfident as well. The `CalibratedClassifierCV` class from the `sklearn.calibration` package can calibrate the estimated probabilities using cross-validation, making them much closer to actual probabilities (see the notebook for a code example). This is important in some scenarios, such as medical diagnosis, financial risk assessment, or fraud detection.

The second column contains the estimated probabilities for the positive class, so let's pass them to the `precision_recall_curve()` function:

```
y_scores_forest = y_probas_forest[:, 1]
precisions_forest, recalls_forest, thresholds_forest = precision_recall_curve(
    y_train_5, y_scores_forest)
```

Now we're ready to plot the PR curve. It is useful to plot the first PR curve as well to see how they compare ([Figure 3-8](#)):

```
plt.plot(recalls_forest, precisions_forest, "b-", linewidth=2,
         label="Random Forest")
plt.plot(recalls, precisions, "--", linewidth=2, label="SGD")
[...] # beautify the figure: add labels, grid, and legend
plt.show()
```

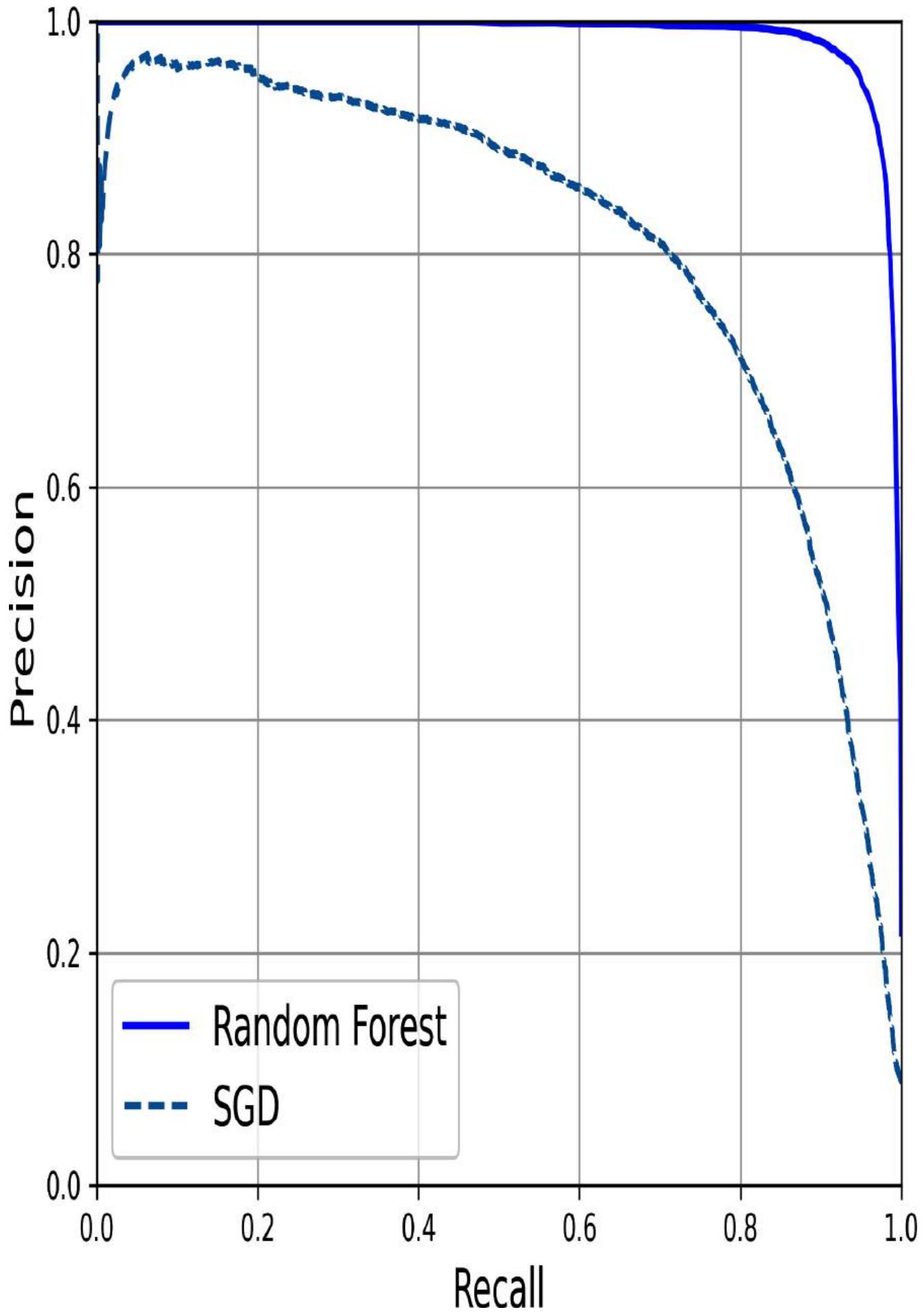


Figure 3-8. Comparing PR curves: the random forest classifier is superior to the SGD classifier because its PR curve is much closer to the top-right corner, and it has a greater AUC

As you can see in [Figure 3-8](#), the `RandomForestClassifier`'s PR curve looks much better than the `SGDClassifier`'s: it comes much closer to the top-right corner. Its  $F_1$  score and ROC AUC score are also significantly better:

```
>>> y_train_pred_forest = y_probas_forest[:, 1] >= 0.5 # positive proba ≥ 50%
>>> f1_score(y_train_5, y_train_pred_forest)
0.9274509803921569
>>> roc_auc_score(y_train_5, y_scores_forest)
0.9983436731328145
```

Try measuring the precision and recall scores: you should find about 99.0% precision and 87.3% recall. Not too bad!

You now know how to train binary classifiers, choose the appropriate metric for your task, evaluate your classifiers using cross-validation, select the precision/recall trade-off that fits your needs, and use several metrics and curves to compare various models. You're ready to try to detect more than just the 5s.

## Multiclass Classification

Whereas binary classifiers distinguish between two classes, *multiclass classifiers* (also called *multinomial classifiers*) can distinguish between more than two classes.

Some Scikit-Learn classifiers (e.g., `LogisticRegression`, `RandomForestClassifier`, and `GaussianNB`) are capable of handling multiple classes natively. Others are strictly binary classifiers (e.g., `SGDClassifier` and `SVC`). However, there are various strategies that you can use to perform multiclass classification with multiple binary classifiers.

One way to create a system that can classify the digit images into 10 classes (from 0 to 9) is to train 10 binary classifiers, one for each digit (a 0-detector, a 1-detector, a 2-detector, and so on). Then when you want to classify an image, you get the decision score from each classifier for that image and you select the class whose classifier outputs the highest score. This is called the *one-versus-the-rest* (OvR) strategy, or sometimes *one-versus-all* (OvA).

Another strategy is to train a binary classifier for every pair of digits: one to distinguish 0s and 1s, another to distinguish 0s and 2s, another for 1s and 2s, and so on. This is

called the *one-versus-one* (OvO) strategy. If there are  $N$  classes, you need to train  $N \times (N - 1) / 2$  classifiers. For the MNIST problem, this means training 45 binary classifiers! When you want to classify an image, you have to run the image through all 45 classifiers and see which class wins the most duels. The main advantage of OvO is that each classifier only needs to be trained on the part of the training set containing the two classes that it must distinguish.

Some algorithms (such as support vector machine classifiers) scale poorly with the size of the training set. For these algorithms OvO is preferred because it is faster to train many classifiers on small training sets than to train few classifiers on large training sets. For most binary classification algorithms, however, OvR is preferred.

Scikit-Learn detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs OvR or OvO, depending on the algorithm. Let's try this with a support vector machine classifier using the `sklearn.svm.SVC` class (see the online chapter on SVMs at <https://homl.info/svm-p>). We'll only train on the first 2,000 images, or else it will take a very long time:

```
from sklearn.svm import SVC

svm_clf = SVC(random_state=42)
svm_clf.fit(X_train[:2000], y_train[:2000]) # y_train, not y_train_5
```

That was easy! We trained the `SVC` using the original target classes from 0 to 9 (`y_train`), instead of the 5-versus-the-rest target classes (`y_train_5`). Since there are 10 classes (i.e., more than 2), Scikit-Learn used the OvO strategy and trained 45 binary classifiers. Now let's make a prediction on an image:

```
>>> svm_clf.predict([some_digit])
array(['5'], dtype=object)
```

That's correct! This code actually made 45 predictions—one per pair of classes—and it selected the class that won the most duels.<sup>5</sup> If you call the `decision_function()` method, you will see that it returns 10 scores per instance: one per class. Each class gets a score equal to the number of won duels plus or minus a small tweak ( $\text{max} \pm 0.33$ ) to break ties, based on the classifier scores:

```
>>> some_digit_scores = svm_clf.decision_function([some_digit])
>>> some_digit_scores.round(2)
array([[ 3.79,  0.73,  6.06,  8.3 , -0.29,  9.3 ,  1.75,  2.77,  7.21,
        4.82]])
```

The highest score is 9.3, and it's indeed the one corresponding to class 5:

```
>>> class_id = some_digit_scores.argmax()
>>> class_id
5
```

When a classifier is trained, it stores the list of target classes in its `classes_` attribute, ordered by value. In the case of MNIST, the index of each class in the `classes_` array conveniently matches the class itself (e.g., the class at index 5 happens to be class '5'), but in general you won't be so lucky; you will need to look up the class label like this:

```
>>> svm_clf.classes_
array(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'], dtype=object)
>>> svm_clf.classes_[class_id]
'5'
```

If you want to force Scikit-Learn to use one-versus-one or one-versus-the-rest, you can use the `OneVsOneClassifier` or `OneVsRestClassifier` classes. Simply create an instance and pass a classifier to its constructor (it doesn't even have to be a binary classifier). For example, this code creates a multiclass classifier using the OvR strategy, based on an SVC:

```
from sklearn.multiclass import OneVsRestClassifier

ovr_clf = OneVsRestClassifier(SVC(random_state=42))
ovr_clf.fit(X_train[:2000], y_train[:2000])
```

Let's make a prediction, and check the number of trained classifiers:

```
>>> ovr_clf.predict([some_digit])
array(['5'], dtype='<U1')
>>> len(ovr_clf.estimators_)
10
```

Training an `SGDClassifier` on a multiclass dataset and using it to make predictions is just as easy:

```
>>> sgd_clf = SGDClassifier(random_state=42)
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array(['3'], dtype='<U1')
```

Oops, that's incorrect. Prediction errors do happen! This time Scikit-Learn used the OvR strategy under the hood: since there are 10 classes, it trained 10 binary classifiers. The `decision_function()` method now returns one value per class. Let's look at the scores that the SGD classifier assigned to each class:

```
>>> sgd_clf.decision_function([some_digit]).round()
array([[-31893., -34420., -9531., 1824., -22320., -1386., -26189.,
       -16148., -4604., -12051.]])
```

You can see that the classifier is not very confident about its prediction: almost all scores are very negative, while class 3 has a score of +1,824, and class 5 is not too far behind at -1,386. Of course, you'll want to evaluate this classifier on more than one image. Since there are roughly the same number of images in each class, the accuracy metric is fine. As usual, you can use the `cross_val_score()` function to evaluate the model:

```
>>> cross_val_score(sgd_clf, X_train, y_train, cv=3, scoring="accuracy")
array([0.87365, 0.85835, 0.8689])
```

It gets over 85.8% on all test folds. If you used a random classifier, you would get 10% accuracy, so this is not such a bad score, but you can still do much better. Simply scaling the inputs (as discussed in [Chapter 2](#)) increases accuracy above 89.1%:

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler()
>>> X_train_scaled = scaler.fit_transform(X_train.astype("float64"))
>>> cross_val_score(sgd_clf, X_train_scaled, y_train, cv=3, scoring="accuracy")
array([0.8983, 0.891 , 0.9018])
```

## Error Analysis

If this were a real project, you would now follow the steps in your machine learning project checklist (see [\[Link to Come\]](#)). You'd explore data preparation options, try out multiple models, shortlist the best ones, fine-tune their hyperparameters using `GridSearchCV`, and automate as much as possible. Here, we will assume that you have found a promising model and you want to find ways to improve it. One way to do this is to analyze the types of errors it makes.

First, look at the confusion matrix. For this, you first need to make predictions using the `cross_val_predict()` function; then you can pass the labels and predictions to the

`confusion_matrix()` function, just like you did earlier. However, since there are now 10 classes instead of 2, the confusion matrix will contain quite a lot of numbers, and it may be hard to read.

A colored diagram of the confusion matrix is much easier to analyze. To plot such a diagram, use the `ConfusionMatrixDisplay.from_predictions()` function like this:

```
from sklearn.metrics import ConfusionMatrixDisplay

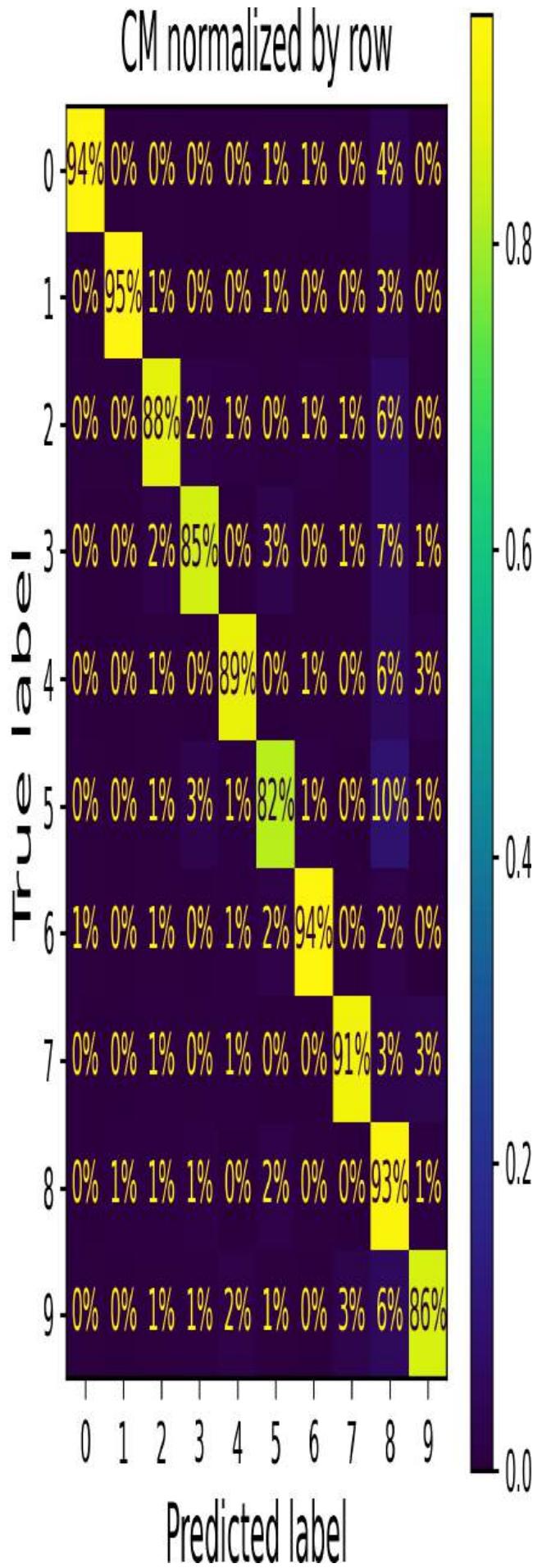
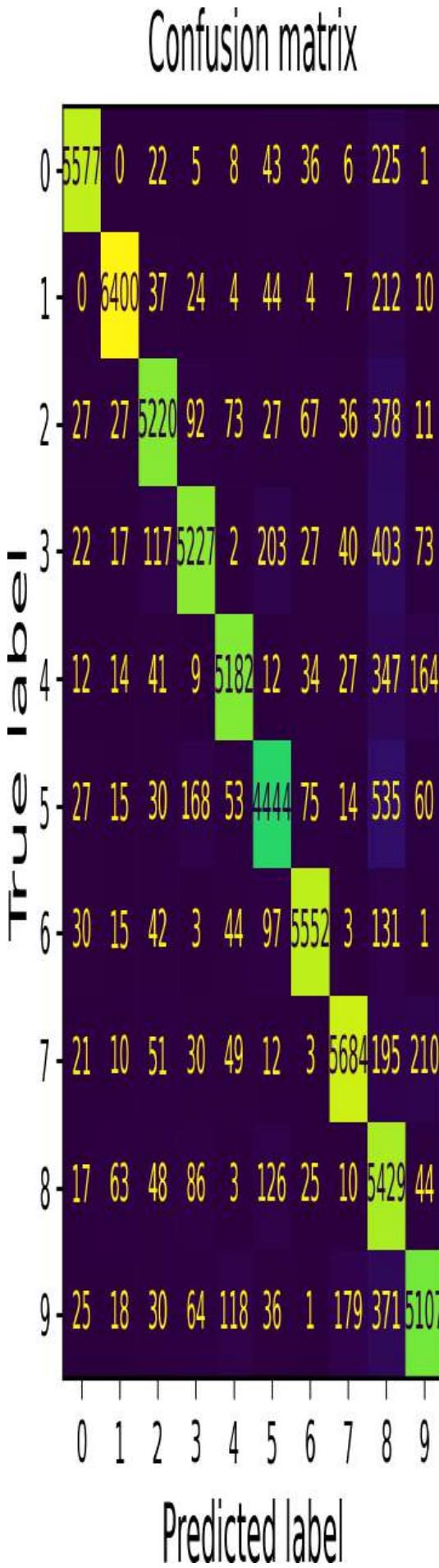
y_train_pred = cross_val_predict(sgd_clf, X_train_scaled, y_train, cv=3)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred)
plt.show()
```

This produces the left diagram in [Figure 3-9](#). This confusion matrix looks pretty good: most images are on the main diagonal, which means that they were classified correctly. Notice that the cell on the diagonal in row #5 and column #5 looks slightly darker than the other digits. This could be because the model made more errors on 5s, or because there are fewer 5s in the dataset than the other digits. That's why it's important to normalize the confusion matrix by dividing each value by the total number of images in the corresponding (true) class (i.e., divide by the row's sum). This can be done simply by setting `normalize="true"`. We can also specify the `values_format=".0%"` argument to show percentages with no decimals. The following code produces the diagram on the right in [Figure 3-9](#):

```
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                       normalize="true", values_format=".0%")
plt.show()
```

Now we can easily see that only 82% of the images of 5s were classified correctly. The most common error the model made with images of 5s was to misclassify them as 8s: this happened for 10% of all 5s. But only 2% of 8s got misclassified as 5s; confusion matrices are generally not symmetrical! If you look carefully, you will notice that many digits have been misclassified as 8s, but this is not immediately obvious from this diagram. If you want to make the errors stand out more, you can try putting zero weight on the correct predictions. The following code does just that and produces the diagram on the left in [Figure 3-10](#):

```
sample_weight = (y_train_pred != y_train)
ConfusionMatrixDisplay.from_predictions(y_train, y_train_pred,
                                       sample_weight=sample_weight,
                                       normalize="true", values_format=".0%")
plt.show()
```



*Figure 3-9. Confusion matrix (left) and the same CM normalized by row (right)*

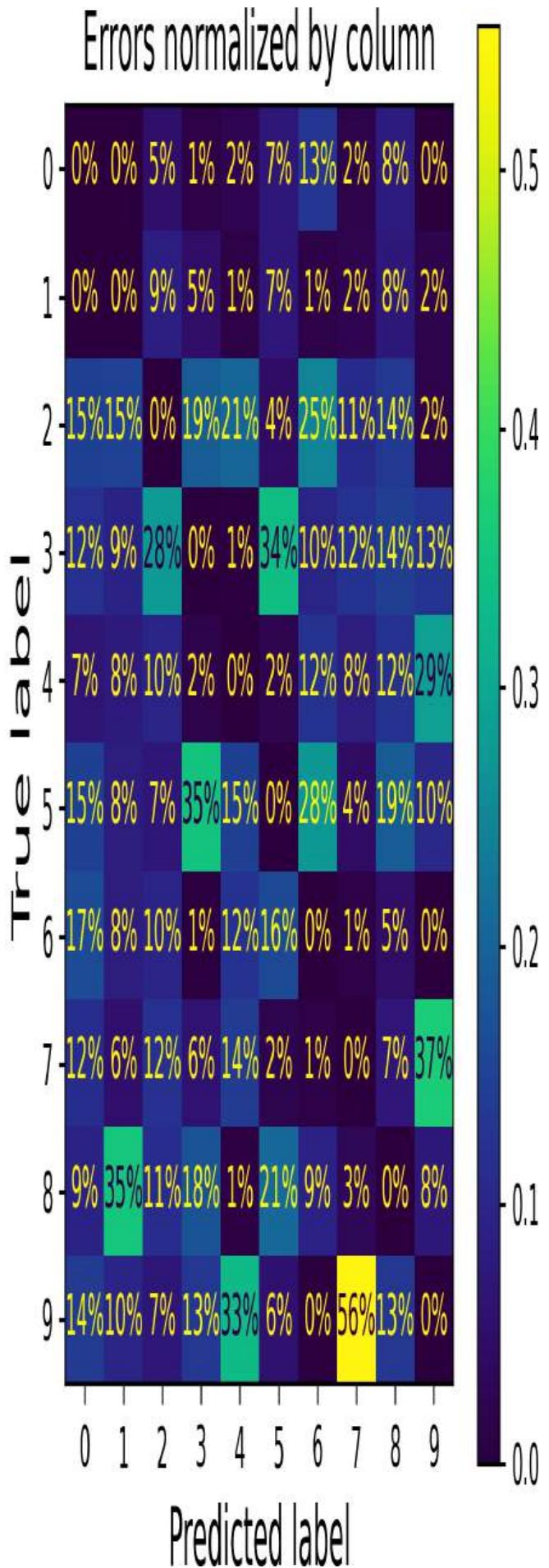
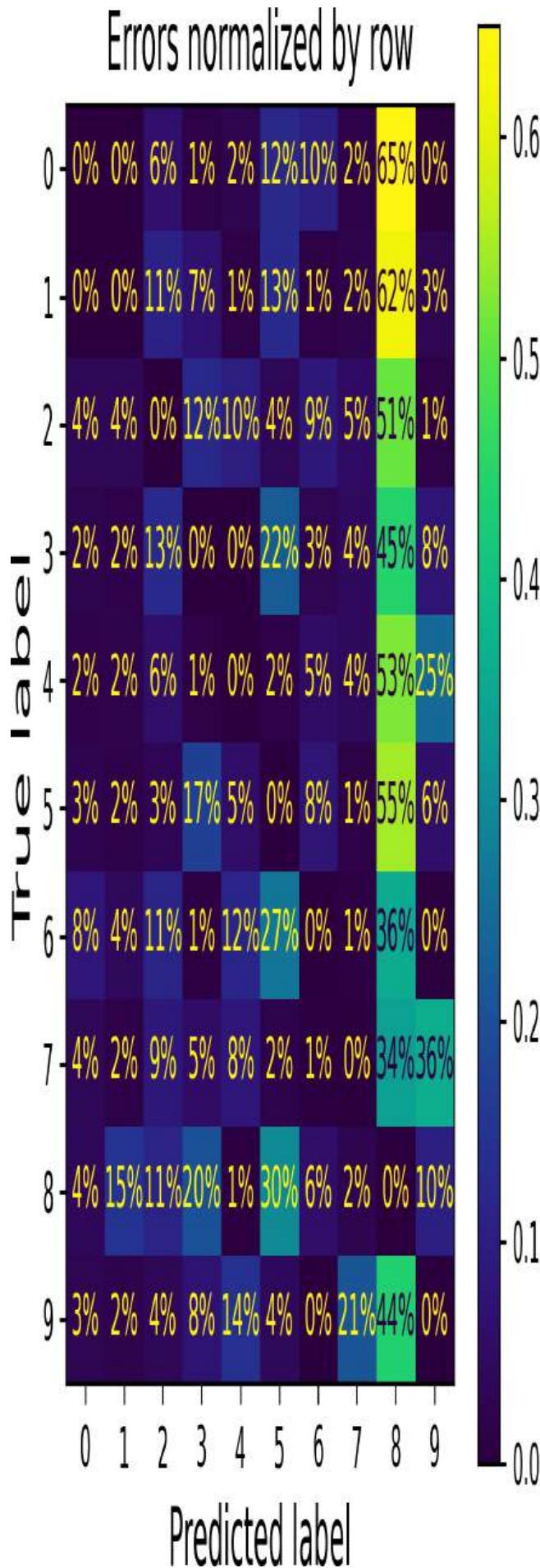


Figure 3-10. Confusion matrix with errors only, normalized by row (left) and by column (right)

Now you can see much more clearly the kinds of errors the classifier makes. The column for class 8 is now really bright, which confirms that many images got misclassified as 8s. In fact this is the most common misclassification for almost all classes. But be careful how you interpret the percentages in this diagram: remember that we've excluded the correct predictions. For example, the 36% in row #7, column #9 in the left grid does *not* mean that 36% of all images of 7s were misclassified as 9s. It means that 36% of the *errors* the model made on images of 7s were misclassifications as 9s. In reality, only 3% of images of 7s were misclassified as 9s, as you can see in the diagram on the right in [Figure 3-9](#).

It is also possible to normalize the confusion matrix by column rather than by row: if you set `normalize="pred"`, you get the diagram on the right in [Figure 3-10](#). For example, you can see that 56% of misclassified 7s are actually 9s.

Analyzing the confusion matrix often gives you insights into ways to improve your classifier. Looking at these plots, it seems that your efforts should be spent on reducing the false 8s. For example, you could try to gather more training data for digits that look like 8s (but are not) so that the classifier can learn to distinguish them from real 8s. Or you could engineer new features that would help the classifier—for example, writing an algorithm to count the number of closed loops (e.g., 8 has two, 6 has one, 5 has none). Or you could preprocess the images (e.g., using Scikit-Image, Pillow, or OpenCV) to make some patterns, such as closed loops, stand out more.

Analyzing individual errors can also be a good way to gain insights into what your classifier is doing and why it is failing. For example, let's plot examples of 3s and 5s in a confusion matrix style ([Figure 3-11](#)):

```
cl_a, cl_b = '3', '5'  
X_aa = X_train[(y_train == cl_a) & (y_train_pred == cl_a)]  
X_ab = X_train[(y_train == cl_a) & (y_train_pred == cl_b)]  
X_ba = X_train[(y_train == cl_b) & (y_train_pred == cl_a)]  
X_bb = X_train[(y_train == cl_b) & (y_train_pred == cl_b)]  
[...] # plot all images in X_aa, X_ab, X_ba, X_bb in a confusion matrix style
```

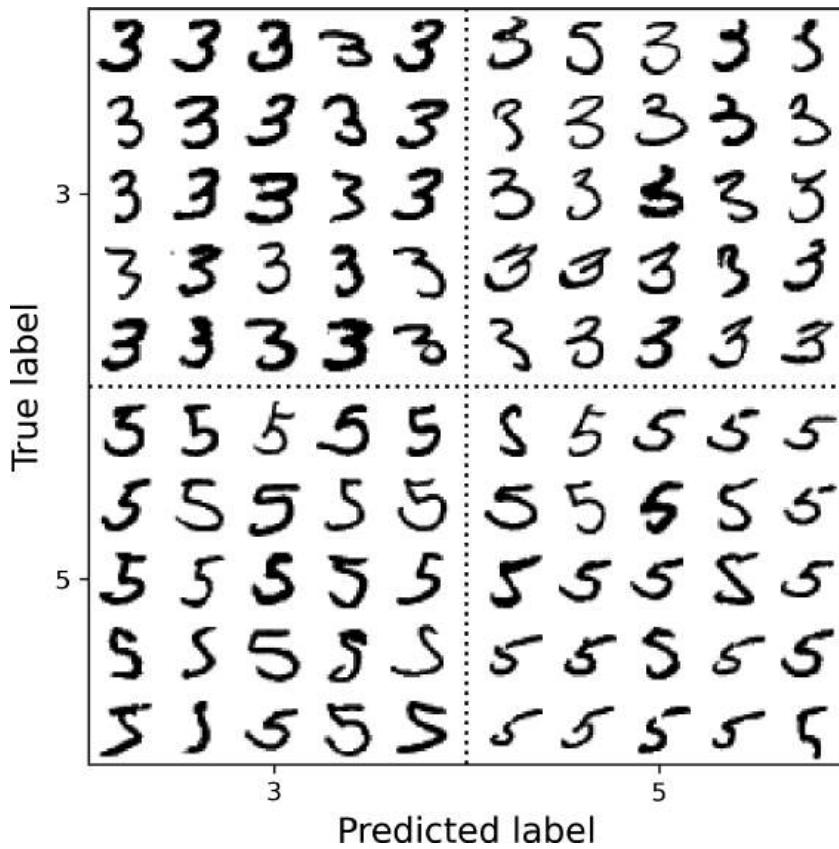


Figure 3-11. Some images of 3s and 5s organized like a confusion matrix

As you can see, some of the digits that the classifier gets wrong (i.e., in the bottom-left and top-right blocks) are so badly written that even a human would have trouble classifying them. However, most misclassified images seem like obvious errors to us. It may be hard to understand why the classifier made the mistakes it did, but remember that the human brain is a fantastic pattern recognition system, and our visual system does a lot of complex preprocessing before any information even reaches our consciousness. So, the fact that this task feels simple does not mean that it is. Recall that we used a simple `SGDClassifier`, which is just a linear model: all it does is assign a weight per class to each pixel, and when it sees a new image it just sums up the weighted pixel intensities to get a score for each class. Since 3s and 5s differ by only a few pixels, this model will easily confuse them.

The main difference between 3s and 5s is the position of the small line that joins the top line to the bottom arc. If you draw a 3 with the junction slightly shifted to the left, the classifier might classify it as a 5, and vice versa. In other words, this classifier is quite sensitive to image shifting and rotation. One way to reduce the 3/5 confusion is to preprocess the images to ensure that they are well centered and not too rotated. However, this may not be easy since it requires predicting the correct rotation of each image. A much simpler approach consists of augmenting the training set with slightly shifted and rotated variants of the training images. This will force the model to learn to

be more tolerant to such variations. This is called *data augmentation* (we'll cover this in [Link to Come]; also see exercise 2 at the end of this chapter).

## Multilabel Classification

Until now, each instance has always been assigned to just one class. But in some cases you may want your classifier to output multiple classes for each instance. Consider a face-recognition classifier: what should it do if it recognizes several people in the same picture? It should attach one tag per person it recognizes. Say the classifier has been trained to recognize three faces: Alice, Bob, and Charlie. Then when the classifier is shown a picture of Alice and Charlie, it should output [True, False, True] (meaning “Alice yes, Bob no, Charlie yes”). Such a classification system that outputs multiple binary tags is called a *multilabel classification* system.

We won't go into face recognition just yet, but let's look at a simpler example, just for illustration purposes:

```
import numpy as np
from sklearn.neighbors import KNeighborsClassifier

y_train_large = (y_train >= '7')
y_train_odd = (y_train.astype('int8') % 2 == 1)
y_multilabel = np.c_[y_train_large, y_train_odd]

knn_clf = KNeighborsClassifier()
knn_clf.fit(X_train, y_multilabel)
```

This code creates a `y_multilabel` array containing two target labels for each digit image: the first indicates whether or not the digit is large (7, 8, or 9), and the second indicates whether or not it is odd. Then the code creates a `KNeighborsClassifier` instance, which supports multilabel classification (not all classifiers do), and trains this model using the multiple targets array. Now you can make a prediction, and notice that it outputs two labels:

```
>>> knn_clf.predict([some_digit])
array([[False,  True]])
```

And it gets it right! The digit 5 is indeed not large (`False`) and odd (`True`).

There are many ways to evaluate a multilabel classifier, and selecting the right metric really depends on your project. One approach is to measure the  $F_1$  score for each

individual label (or any other binary classifier metric discussed earlier), then simply compute the average score. The following code computes the average  $F_1$  score across all labels:

```
>>> y_train_knn_pred = cross_val_predict(knn_clf, X_train, y_multilabel, cv=3)
>>> f1_score(y_multilabel, y_train_knn_pred, average="macro")
0.976410265560605
```

This approach assumes that all labels are equally important, which may not be the case. In particular, if you have many more pictures of Alice than of Bob or Charlie, you may want to give more weight to the classifier’s score on pictures of Alice. One simple option is to give each label a weight equal to its *support* (i.e., the number of instances with that target label). To do this, simply set `average="weighted"` when calling the `f1_score()` function.<sup>6</sup>

If you wish to use a classifier that does not natively support multilabel classification, such as `SVC`, one possible strategy is to train one model per label. However, this strategy may have a hard time capturing the dependencies between the labels. For example, a large digit (7, 8, or 9) is twice more likely to be odd than even, but the classifier for the “odd” label does not know what the classifier for the “large” label predicted. To solve this issue, the models can be organized in a chain: when a model makes a prediction, it uses the input features plus all the predictions of the models that come before it in the chain.

The good news is that Scikit-Learn has a class called `ClassifierChain` that does just that! By default it will use the true labels for training, feeding each model the appropriate labels depending on their position in the chain. But if you set the `cv` hyperparameter, it will use cross-validation to get “clean” (out-of-sample) predictions from each trained model for every instance in the training set, and these predictions will then be used to train all the models later in the chain. Note that the order of the classifiers in the chain may affect the final performance. Here’s an example showing how to create and train a `ClassifierChain` using the cross-validation strategy. As earlier, we’ll just use the first 2,000 images in the training set to speed things up:

```
from sklearn.multioutput import ClassifierChain

chain_clf = ClassifierChain(SVC(), cv=3, random_state=42)
chain_clf.fit(X_train[:2000], y_multilabel[:2000])
```

Now we can use this `ClassifierChain` to make predictions:

```
>>> chain_clf.predict([some_digit])
array([[0., 1.]])
```

## Multioutput Classification

The last type of classification task we'll discuss here is called *multioutput–multiclass classification* (or just *multioutput classification*). It is a generalization of multilabel classification where each label can be multiclass (i.e., it can have more than two possible values).

To illustrate this, let's build a system that removes noise from images. It will take as input a noisy digit image, and it will (hopefully) output a clean digit image, represented as an array of pixel intensities, just like the MNIST images. Notice that the classifier's output is multilabel (one label per pixel) and each label can have multiple values (pixel intensity ranges from 0 to 255). This is thus an example of a multioutput classification system.

### NOTE

The line between classification and regression is sometimes blurry, such as in this example. Arguably, predicting pixel intensity is more akin to regression than to classification. Moreover, multioutput systems are not limited to classification tasks; you could even have a system that outputs multiple labels per instance, including both class labels and value labels.

Let's start by creating the training and test sets by taking the MNIST images and adding noise to their pixel intensities, using a random number generator's `integers()` method. The target images will be the original images:

```
rng = np.random.default_rng(seed=42)
noise_train = rng.integers(0, 100, (len(X_train), 784))
X_train_mod = X_train + noise_train
noise_test = rng.integers(0, 100, (len(X_test), 784))
X_test_mod = X_test + noise_test
y_train_mod = X_train
y_test_mod = X_test
```

Let's take a peek at the first image from the test set (Figure 3-12). Yes, we're snooping on the test data, so you should be frowning right now.

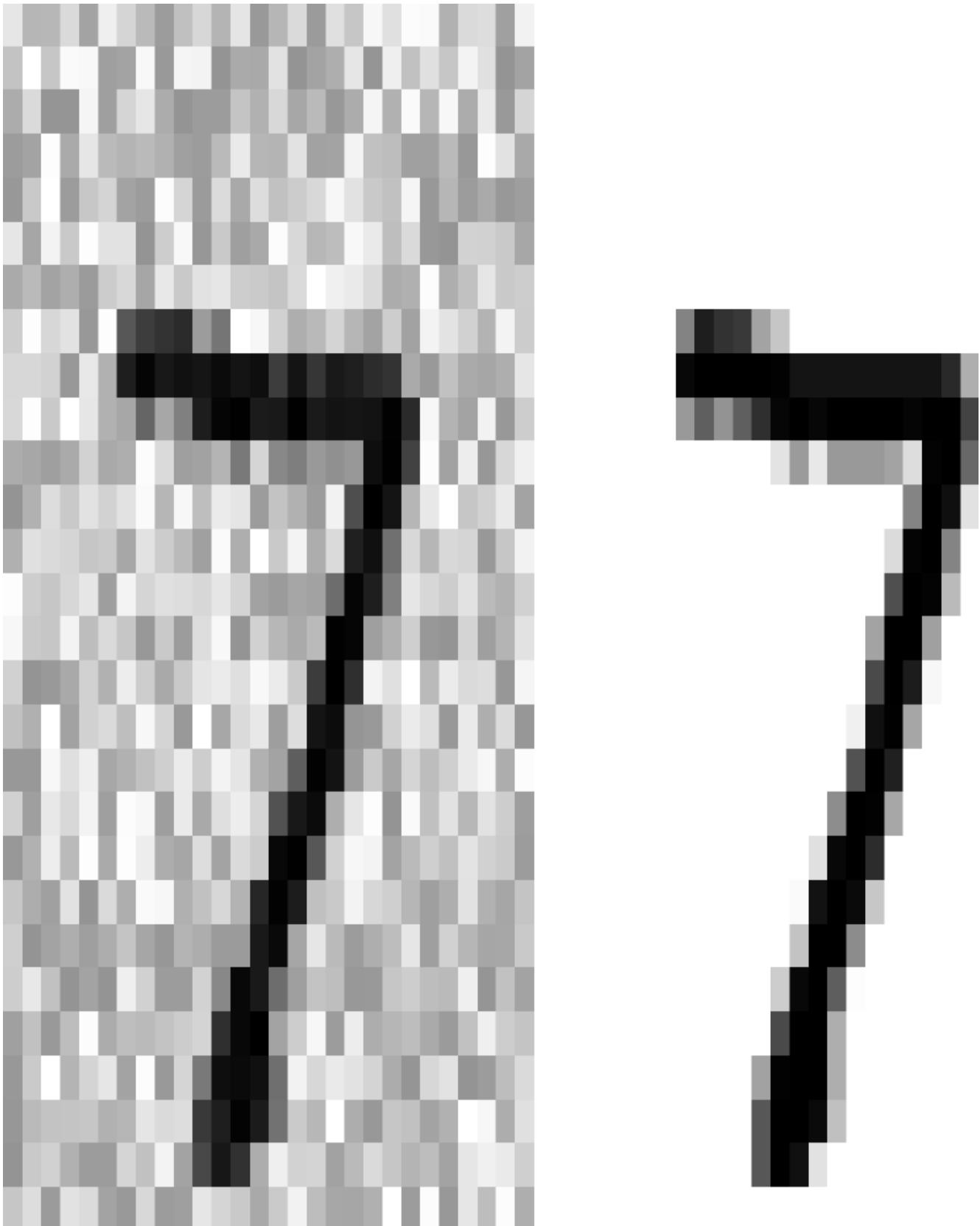


Figure 3-12. A noisy image (left) and the target clean image (right)

On the left is the noisy input image, and on the right is the clean target image. Now let's train the classifier and make it clean up this image ([Figure 3-13](#)):

```
knn_clf = KNeighborsClassifier()  
knn_clf.fit(X_train_mod, y_train_mod)  
clean_digit = knn_clf.predict([X_test_mod[0]])  
plot_digit(clean_digit)  
plt.show()
```

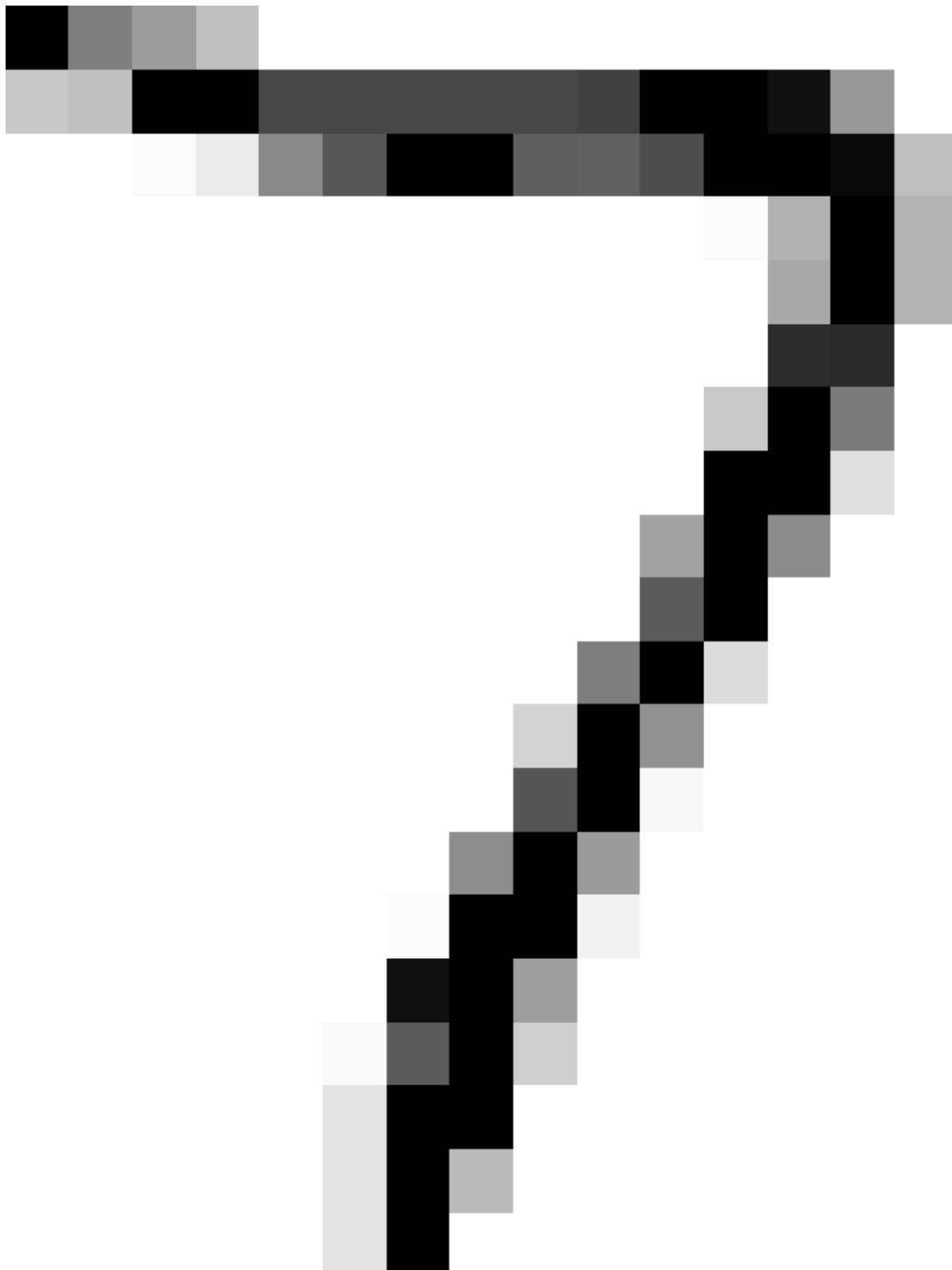


Figure 3-13. The cleaned-up image

Looks close enough to the target! This concludes our tour of classification. You now know how to select good metrics for classification tasks, pick the appropriate precision/recall trade-off, compare classifiers, and more generally build good classification systems for a variety of tasks. In the next chapters, you'll learn how all these machine learning models you've been using actually work.

## Exercises

1. Try to build a classifier for the MNIST dataset that achieves over 97% accuracy on the test set. Hint: the `KNeighborsClassifier` works quite well for this task; you just need to find good hyperparameter values (try a grid search on the `weights` and `n_neighbors` hyperparameters).
2. Write a function that can shift an MNIST image in any direction (left, right, up, or down) by one pixel.<sup>7</sup> Then, for each image in the training set, create four shifted copies (one per direction) and add them to the training set. Finally, train your best model on this expanded training set and measure its accuracy on the test set. You should observe that your model performs even better now! This technique of artificially growing the training set is called *data augmentation* or *training set expansion*.
3. Tackle the Titanic dataset. A great place to start is on [Kaggle](#). Alternatively, you can download the data from <https://homl.info/titanic.tgz> and unzip this tarball like you did for the housing data in [Chapter 2](#). This will give you two CSV files, `train.csv` and `test.csv`, which you can load using `pandas.read_csv()`. The goal is to train a classifier that can predict the `Survived` column based on the other columns.
4. Build a spam classifier (a more challenging exercise):
  - a. Download examples of spam and ham from [Apache SpamAssassin's public datasets](#).
  - b. Unzip the datasets and familiarize yourself with the data format.
  - c. Split the data into a training set and a test set.
  - d. Write a data preparation pipeline to convert each email into a feature vector. Your preparation pipeline should transform an email into a (sparse) vector that indicates the presence or absence of each

possible word. For example, if all emails only ever contain four words, “Hello”, “how”, “are”, “you”, then the email “Hello you Hello Hello you” would be converted into a vector [1, 0, 0, 1] (meaning [“Hello” is present, “how” is absent, “are” is absent, “you” is present]), or [3, 0, 0, 2] if you prefer to count the number of occurrences of each word.

You may want to add hyperparameters to your preparation pipeline to control whether or not to strip off email headers, convert each email to lowercase, remove punctuation, replace all URLs with “URL”, replace all numbers with “NUMBER”, or even perform *stemming* (i.e., trim off word endings; there are Python libraries available to do this).

- e. Finally, try out several classifiers and see if you can build a great spam classifier, with both high recall and high precision.

Solutions to these exercises are available at the end of this chapter’s notebook, at <https://homl.info/colab-p>.

---

- <sup>1</sup> By default Scikit-Learn caches downloaded datasets in a directory called `scikit_learn_data` in your home directory.
- <sup>2</sup> Datasets returned by `fetch_openml()` are not always shuffled or split.
- <sup>3</sup> Shuffling may be a bad idea in some contexts—for example, if you are working on time series data (such as stock market prices or weather conditions). We will explore this in [Link to Come].
- <sup>4</sup> Scikit-Learn classifiers always have either a `decision_function()` method or a `predict_proba()` method, or sometimes both.
- <sup>5</sup> In case of a tie, the first class is selected, unless you set the `break_ties` hyperparameters to `True`, in which case ties are broken using the output of the `decision_function()`.
- <sup>6</sup> Scikit-Learn offers a few other averaging options and multilabel classifier metrics; see the documentation for more details.
- <sup>7</sup> You can use the `shift()` function from the `scipy.ndimage.interpolation` module. For example, `shift(image, [2, 1], cval=0)` shifts the image two pixels down and one pixel to the right.

# Chapter 4. Training Models

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 4th chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

So far we have treated machine learning models and their training algorithms mostly like black boxes. If you went through some of the exercises in the previous chapters, you may have been surprised by how much you can get done without knowing anything about what’s under the hood: you optimized a regression system, you improved a digit image classifier, and you even built a spam classifier from scratch, all without knowing how they actually work. Indeed, in many situations you don’t really need to know the implementation details.

However, having a good understanding of how things work can help you quickly home in on the appropriate model, the right training algorithm to use, and a good set of hyperparameters for your task. Understanding what’s under the hood will also help you debug issues and perform error analysis more efficiently. Lastly, most of the topics discussed in this chapter will be essential in understanding, building, and training neural networks (discussed in [Part II](#) of this book).

In this chapter we will start by looking at the linear regression model, one of the simplest models there is. We will discuss two very different ways to train it:

- Using a “closed-form” equation<sup>1</sup> that directly computes the model parameters that best fit the model to the training set (i.e., the model parameters that minimize the cost function over the training set).
- Using an iterative optimization approach called gradient descent (GD) that gradually tweaks the model parameters to minimize the cost function over the training set, eventually converging to the same set of parameters as the first method. We will look at a few variants of gradient descent that we will use

again and again when we study neural networks in [Part II](#): batch GD, mini-batch GD, and stochastic GD.

Next we will look at polynomial regression, a more complex model that can fit nonlinear datasets. Since this model has more parameters than linear regression, it is more prone to overfitting the training data. We will explore how to detect whether or not this is the case using learning curves, and then we will look at several regularization techniques that can reduce the risk of overfitting the training set.

Finally, we will examine two more models that are commonly used for classification tasks: logistic regression and softmax regression.

### WARNING

There will be quite a few math equations in this chapter using basic concepts of linear algebra and calculus. To understand these equations, you need to be familiar with vectors and matrices—how to transpose them, multiply them, and invert them—as well as partial derivatives. If these concepts are unfamiliar, please review the introductory Jupyter notebooks on linear algebra and calculus provided in the [online supplemental material](#). If you are truly allergic to maths, you can just skip the equations, the text should still help you grasp most of the concepts. That said, learning the mathematical formalism is extremely useful, as it will allow you to read ML papers. Although it may seem daunting at first, it's actually not that hard, and this chapter includes code that should help you make sense of the equations.

## Linear Regression

In [Chapter 1](#) we looked at a simple regression model of life satisfaction:

$$\text{life\_satisfaction} = \theta_0 + \theta_1 \times \text{GDP\_per\_capita}$$

This model is just a linear function of the input feature `GDP_per_capita`.  $\theta_0$  and  $\theta_1$  are the model's parameters.

More generally, a linear model makes a prediction by simply computing a weighted sum of the input features, plus a constant called the *bias term* (also called the *intercept term*), as shown in [Equation 4-1](#).

*Equation 4-1. Linear regression model prediction*

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

In this equation:

- $\hat{y}$  is the predicted value.
- $n$  is the number of features.
- $x_i$  is the  $i^{\text{th}}$  feature value.
- $\theta_j$  is the  $j^{\text{th}}$  model parameter, including the bias term  $\theta_0$  and the feature weights  $\theta_1, \theta_2, \dots, \theta_n$ .

This can be written much more concisely using a vectorized form, as shown in [Equation 4-2](#).

*Equation 4-2. Linear regression model prediction (vectorized form)*

$$\hat{y} = h_{\theta}(\mathbf{x}) = \boldsymbol{\theta} \cdot \mathbf{x}$$

In this equation:

- $h_{\theta}$  is the hypothesis function, using the model parameters  $\boldsymbol{\theta}$ .
- $\boldsymbol{\theta}$  is the model's *parameter vector*, containing the bias term  $\theta_0$  and the feature weights  $\theta_1$  to  $\theta_n$ .
- $\mathbf{x}$  is the instance's *feature vector*, containing  $x_0$  to  $x_n$ , with  $x_0$  always equal to 1.
- $\boldsymbol{\theta} \cdot \mathbf{x}$  is the dot product of the vectors  $\boldsymbol{\theta}$  and  $\mathbf{x}$ , which is equal to  $\theta_0x_0 + \theta_1x_1 + \theta_2x_2 + \dots + \theta_nx_n$ .

### NOTE

In machine learning, vectors are often represented as *column vectors*, which are 2D arrays with a single column. If  $\boldsymbol{\theta}$  and  $\mathbf{x}$  are column vectors, then the prediction is  $\hat{y} = \boldsymbol{\theta}^T \mathbf{x}$ , where  $\boldsymbol{\theta}^T$  is the *transpose* of  $\boldsymbol{\theta}$  (a row vector instead of a column vector) and  $\boldsymbol{\theta}^T \mathbf{x}$  is the matrix multiplication of  $\boldsymbol{\theta}^T$  and  $\mathbf{x}$ . It is of course the same prediction, except that it is now represented as a single-cell matrix rather than a scalar value. In this book I will use this notation to avoid switching between dot products and matrix multiplications.

OK, that's the linear regression model—but how do we train it? Well, recall that training a model means setting its parameters so that the model best fits the training set. For this purpose, we first need a measure of how well (or poorly) the model fits the training data. In [Chapter 2](#) we saw that the most common performance measure of a

regression model is the root mean squared error ([Equation 2-1](#)). Therefore, to train a linear regression model, we need to find the value of  $\theta$  that minimizes the RMSE. In practice, it is simpler to minimize the mean squared error (MSE) than the RMSE, and it leads to the same result (because the value that minimizes a positive function also minimizes its square root).

## WARNING

Learning algorithms will often optimize a different loss function during training than the performance measure used to evaluate the final model. This is generally because the function is easier to optimize and/or because it has extra terms needed during training only (e.g., for regularization). A good performance metric is as close as possible to the final business objective. A good training loss is easy to optimize and strongly correlated with the metric. For example, classifiers are often trained using a cost function such as the log loss (as you will see later in this chapter) but evaluated using precision/recall. The log loss is easy to minimize, and doing so will usually improve precision/recall.

The MSE of a linear regression hypothesis  $h_{\theta}$  on a training set  $\mathbf{X}$  is calculated using [Equation 4-3](#).

*Equation 4-3. MSE cost function for a linear regression model*

$$\text{MSE}(\mathbf{X}, h_{\theta}) = \frac{1}{m} \sum_{i=1}^m \left( \theta^T \mathbf{x}^{(i)} - y^{(i)} \right)^2$$

Most of these notations were presented in [Chapter 2](#) (see “[Notations](#)”). The only difference is that we write  $h_{\theta}$  instead of just  $h$  to make it clear that the model is parametrized by the vector  $\theta$ . To simplify notations, we will just write  $\text{MSE}(\theta)$  instead of  $\text{MSE}(\mathbf{X}, h_{\theta})$ .

## The Normal Equation

To find the value of  $\theta$  that minimizes the MSE, there exists a *closed-form solution*—in other words, a mathematical equation that gives the result directly. This is called the *Normal equation* ([Equation 4-4](#)).

*Equation 4-4. Normal equation*

$$\hat{\theta} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

In this equation:

- $\hat{\theta}$  is the value of  $\theta$  that minimizes the cost function.
- $y$  is the vector of target values containing  $y^{(1)}$  to  $y^{(m)}$ .

Let's generate some linear-looking data to test this equation on (Figure 4-1):

```
import numpy as np

rng = np.random.default_rng(seed=42)
m = 200 # number of instances
X = 2 * rng.random((m, 1)) # column vector
y = 4 + 3 * X + rng.standard_normal((m, 1)) # column vector
```

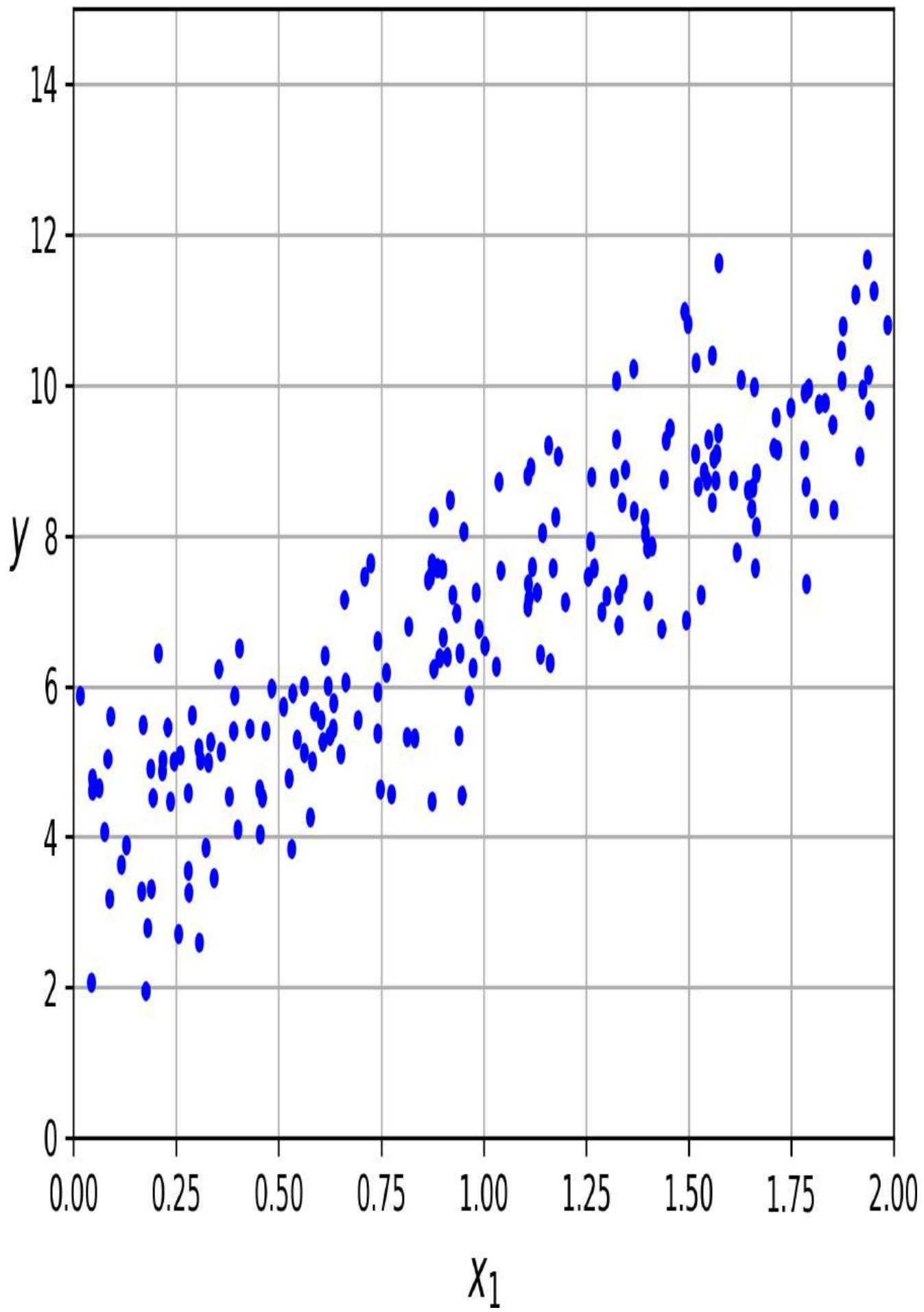


Figure 4-1. A randomly generated linear dataset

Now let's compute  $\hat{\theta}$  using the Normal equation. We will use the `inv()` function from NumPy's linear algebra module (`np.linalg`) to compute the inverse of a matrix, and the `@` operator for matrix multiplication:

```
from sklearn.preprocessing import add_dummy_feature

X_b = add_dummy_feature(X) # add x0 = 1 to each instance
theta_best = np.linalg.inv(X_b.T @ X_b) @ X_b.T @ y
```

## NOTE

The `@` operator performs matrix multiplication. If `A` and `B` are NumPy arrays, then `A @ B` is equivalent to `np.matmul(A, B)`. Many other libraries, like TensorFlow, PyTorch, and JAX, support the `@` operator as well. However, you cannot use `@` on pure Python arrays (i.e., lists of lists).

The function that we used to generate the data is  $y = 4 + 3x_1 + \text{Gaussian noise}$ . Let's see what the equation found:

```
>>> theta_best
array([[3.69084138],
       [3.32960458]])
```

We would have hoped for  $\theta_0 = 4$  and  $\theta_1 = 3$  instead of  $\theta_0 = 3.6908$  and  $\theta_1 = 3.3296$ . Close enough, but the noise made it impossible to recover the exact parameters of the original function. The smaller and noisier the dataset, the harder it gets.

Now we can make predictions using  $\hat{\theta}$ :

```
>>> X_new = np.array([[0], [2]])
>>> X_new_b = add_dummy_feature(X_new) # add x0 = 1 to each instance
>>> y_predict = X_new_b @ theta_best
>>> y_predict
array([[ 3.69084138],
       [10.35005055]])
```

Let's plot this model's predictions (Figure 4-2):

```
import matplotlib.pyplot as plt

plt.plot(X_new, y_predict, "r-", label="Predictions")
```

```
plt.plot(X, y, "b..")
[...] # beautify the figure: add labels, axis, grid, and legend
plt.show()
```

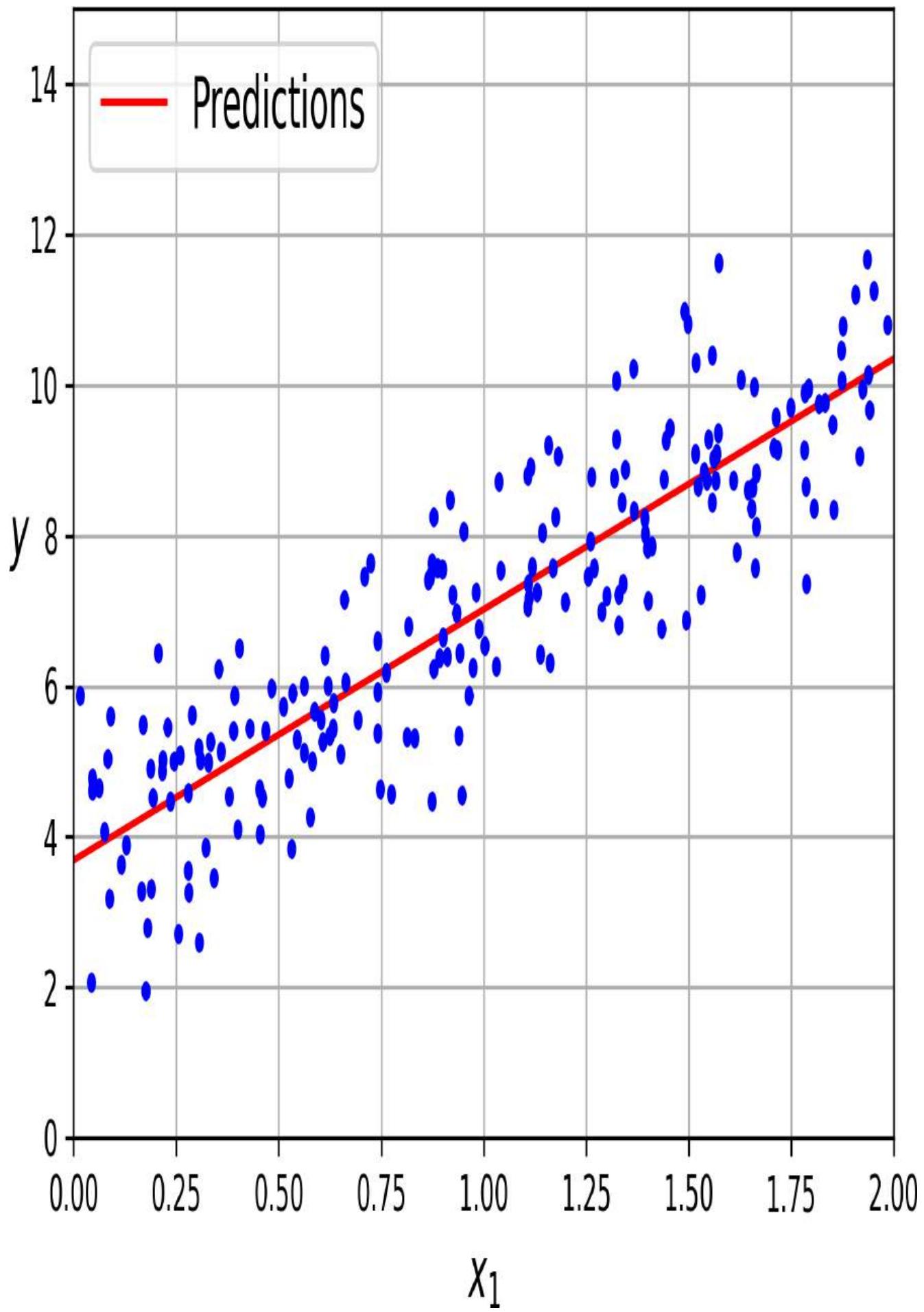


Figure 4-2. Linear regression model predictions

Performing linear regression using Scikit-Learn is relatively straightforward:

```
>>> from sklearn.linear_model import LinearRegression  
>>> lin_reg = LinearRegression()  
>>> lin_reg.fit(X, y)  
>>> lin_reg.intercept_, lin_reg.coef_  
(array([3.69084138]), array([[3.32960458]]))  
>>> lin_reg.predict(X_new)  
array([[ 3.69084138],  
       [10.35005055]])
```

Notice that Scikit-Learn separates the bias term (`intercept_`) from the feature weights (`coef_`). The `LinearRegression` class is based on the `scipy.linalg.lstsq()` function (the name stands for “least squares”), which you could call directly:

```
>>> theta_best_svd, residuals, rank, s = np.linalg.lstsq(X_b, y, rcond=1e-6)  
>>> theta_best_svd  
array([[3.69084138],  
       [3.32960458]])
```

This function computes  $\hat{\theta} = \mathbf{X}^+ \mathbf{y}$ , where  $\mathbf{X}^+$  is the *pseudoinverse* of  $\mathbf{X}$  (specifically, the Moore–Penrose inverse). You can use `np.linalg.pinv()` to compute the pseudoinverse directly:

```
>>> np.linalg.pinv(X_b) @ y  
array([[3.69084138],  
       [3.32960458]])
```

The pseudoinverse itself is computed using a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix  $\mathbf{X}$  into the matrix multiplication of three matrices  $\mathbf{U} \Sigma \mathbf{V}^\top$  (see `numpy.linalg.svd()`). The pseudoinverse is computed as  $\mathbf{X}^+ = \mathbf{V} \Sigma^+ \mathbf{U}^\top$ . To compute the matrix  $\Sigma^+$ , the algorithm takes  $\Sigma$  and sets to zero all values smaller than a tiny threshold value, then it replaces all the nonzero values with their inverse, and finally it transposes the resulting matrix. This approach is more efficient than computing the Normal equation, plus it handles edge cases nicely: indeed, the Normal equation may not work if the matrix  $\mathbf{X}^\top \mathbf{X}$  is not invertible (i.e., singular), such as if  $m < n$  or if some features are redundant, but the pseudoinverse is always defined.

## Computational Complexity

The Normal equation computes the inverse of  $\mathbf{X}^\top \mathbf{X}$ , which is an  $(n + 1) \times (n + 1)$  matrix (where  $n$  is the number of features). The *computational complexity* of inverting such a matrix is typically about  $O(n^{2.4})$  to  $O(n^3)$ , depending on the implementation. In other words, if you double the number of features, you multiply the computation time by roughly  $2^{2.4} = 5.3$  to  $2^3 = 8$ .

The SVD approach used by Scikit-Learn's `LinearRegression` class is about  $O(n^2)$ . If you double the number of features, you multiply the computation time by roughly 4.

### WARNING

Both the Normal equation and the SVD approach get very slow when the number of features grows large (e.g., 100,000). On the positive side, both are linear with regard to the number of instances in the training set (they are  $O(m)$ ), so they handle large training sets efficiently, provided they can fit in memory.

Also, once you have trained your linear regression model (using the Normal equation or any other algorithm), predictions are very fast: the computational complexity is linear with regard to both the number of instances you want to make predictions on and the number of features. In other words, making predictions on twice as many instances (or twice as many features) will take roughly twice as much time.

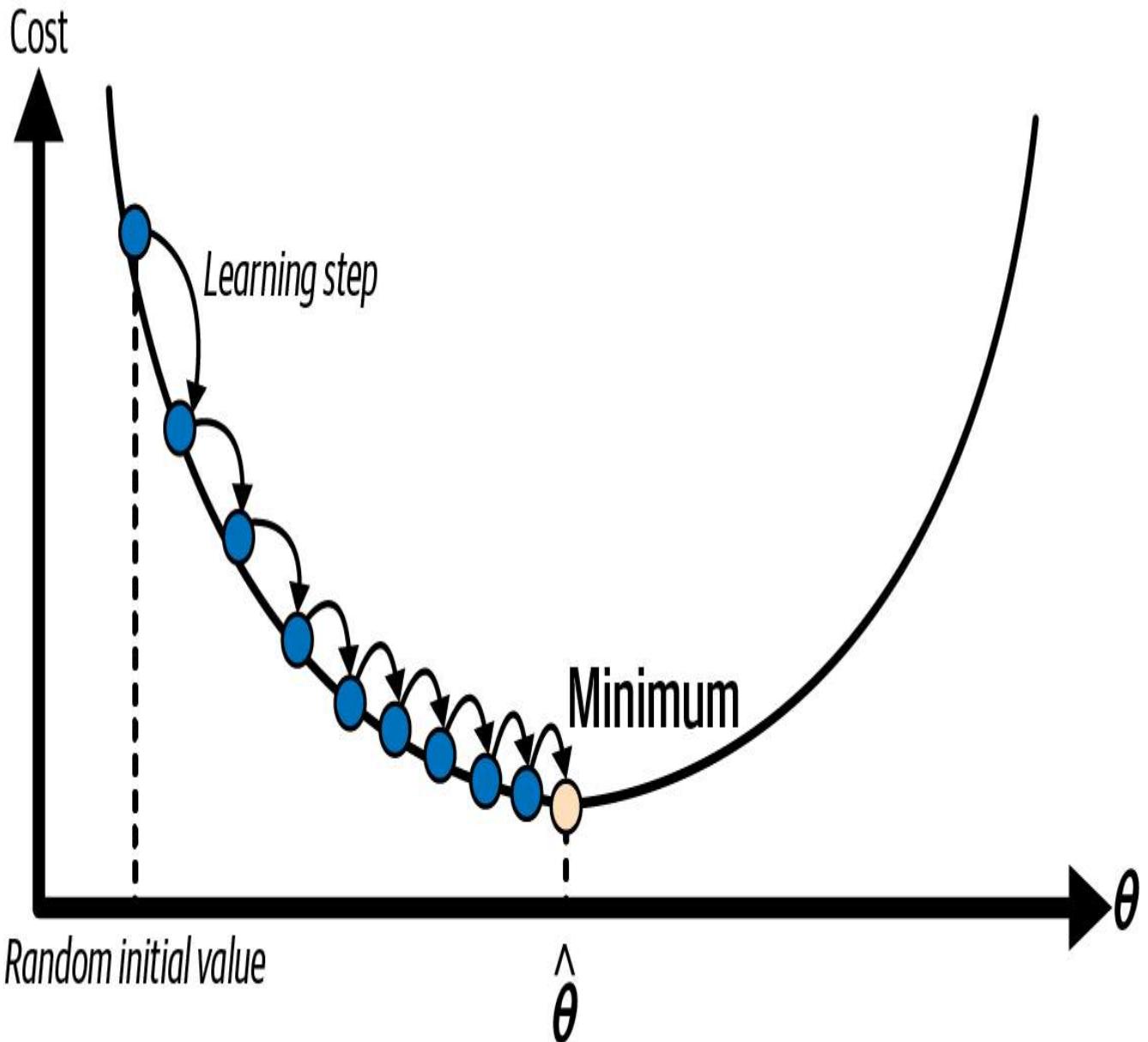
Now we will look at a very different way to train a linear regression model, which is better suited for cases where there are a large number of features or too many training instances to fit in memory.

## Gradient Descent

*Gradient descent* is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of gradient descent is to tweak parameters iteratively in order to minimize a cost function.

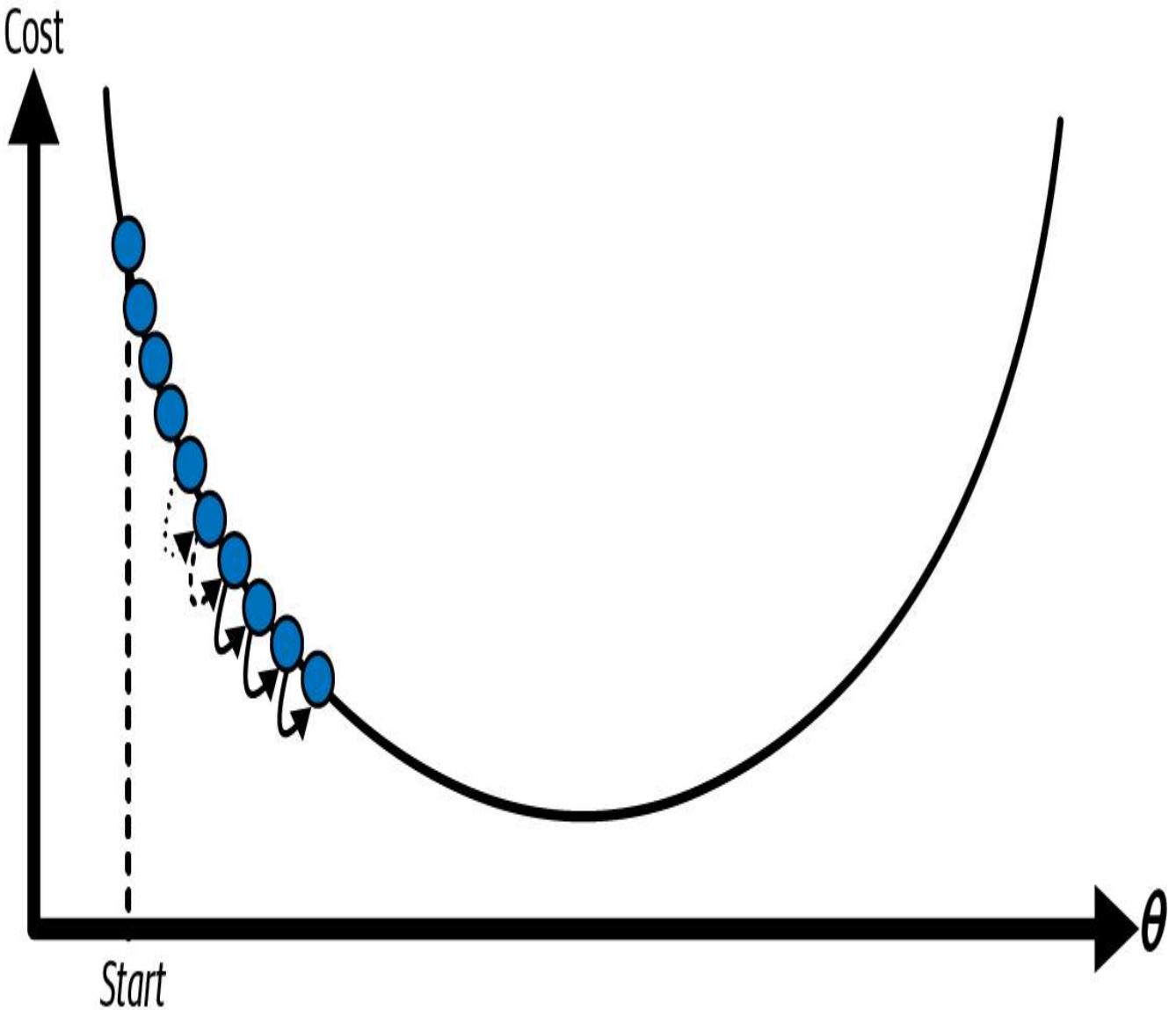
Suppose you are lost in the mountains in a dense fog, and you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope. This is exactly what gradient descent does: it measures the local gradient of the error function with regard to the parameter vector  $\theta$ , and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum!

In practice, you start by filling  $\theta$  with random values (this is called *random initialization*). Then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm *converges* to a minimum (see [Figure 4-3](#)).



*Figure 4-3.* In this depiction of gradient descent, the model parameters are initialized randomly and get tweaked repeatedly to minimize the cost function; the learning step size is proportional to the slope of the cost function, so the steps gradually get smaller as the cost approaches the minimum

An important parameter in gradient descent is the size of the steps, determined by the *learning rate* hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see [Figure 4-4](#)).



*Figure 4-4. Learning rate too small*

On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before. This might make the algorithm diverge, with larger and larger values, failing to find a good solution (see [Figure 4-5](#)).

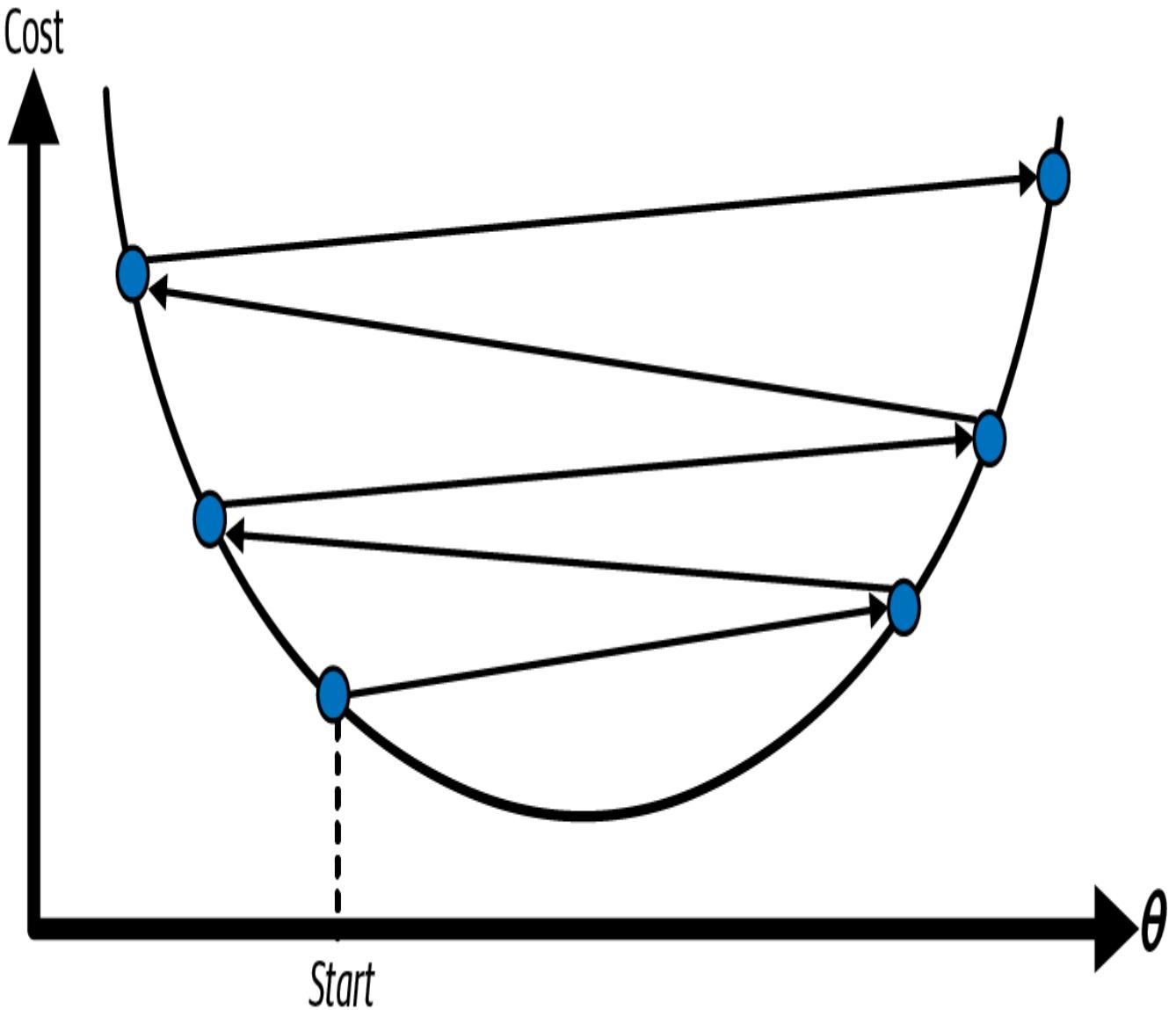


Figure 4-5. Learning rate too high

Additionally, not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrain, making convergence to the minimum difficult. [Figure 4-6](#) shows the two main challenges with gradient descent. If the random initialization starts the algorithm on the left, then it will converge to a *local minimum*, which is not as good as the *global minimum*. If it starts on the right, then it will take a very long time to cross the plateau. And if you stop too early, you will never reach the global minimum.

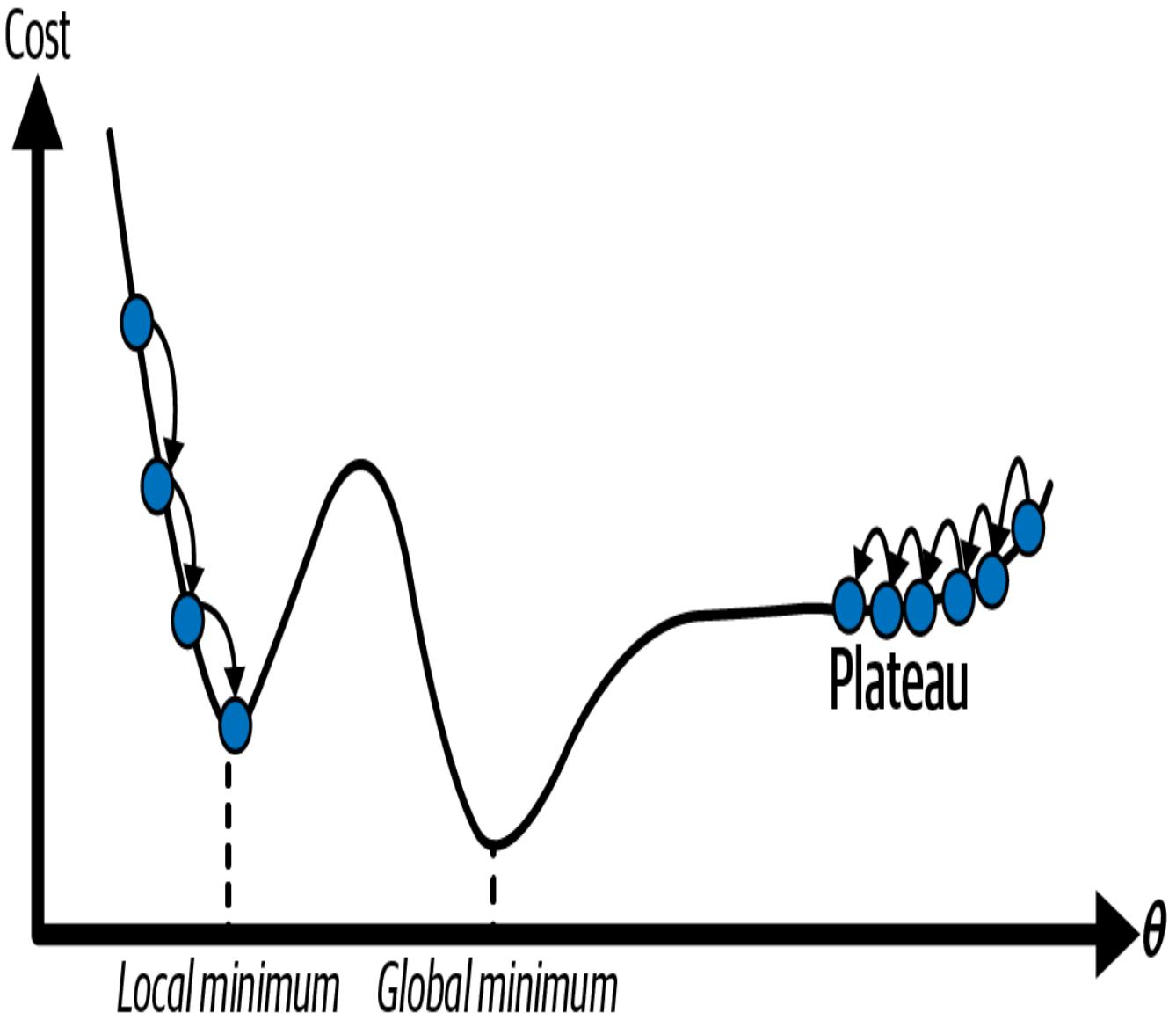
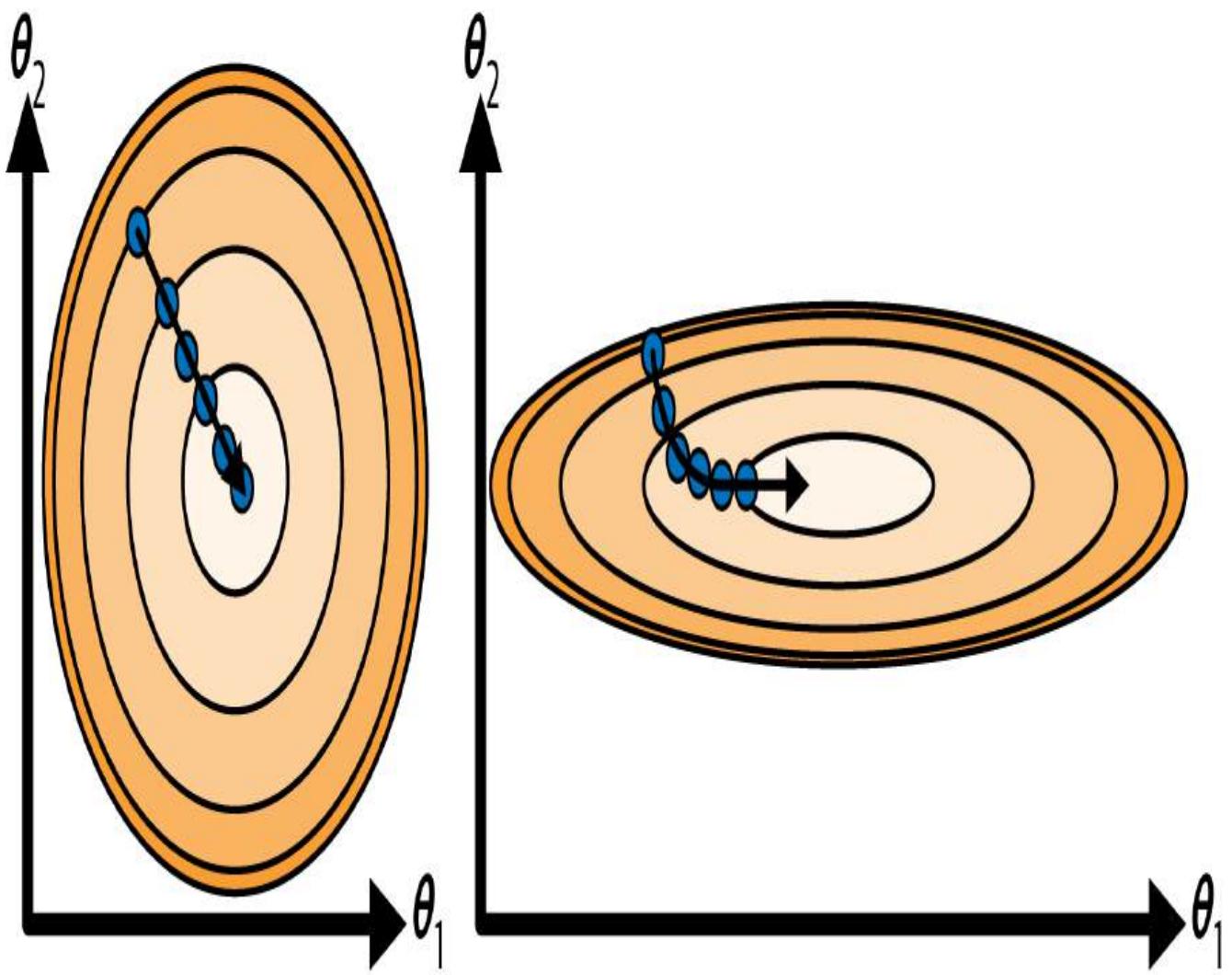
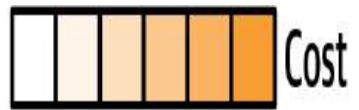


Figure 4-6. Gradient descent pitfalls

Fortunately, the MSE cost function for a linear regression model happens to be a *convex function*, which means that if you pick any two points on the curve, the line segment joining them is never below the curve. This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly.<sup>2</sup> These two facts have a great consequence: gradient descent is guaranteed to approach arbitrarily closely the global minimum (if you wait long enough and if the learning rate is not too high).

While the cost function has the shape of a bowl, it can be an elongated bowl if the features have very different scales. Figure 4-7 shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).<sup>3</sup>



*Figure 4-7. Gradient descent with (left) and without (right) feature scaling*

As you can see, on the left the gradient descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley. It will eventually reach the minimum, but it will take a long time.

### WARNING

When using gradient descent, you should ensure that all features have a similar scale (e.g., using Scikit-Learn's `StandardScaler` class), or else it will take much longer to converge.

This diagram also illustrates the fact that training a model means searching for a combination of model parameters that minimizes a cost function (over the training set). It is a search in the model's *parameter space*. The more parameters a model has, the more dimensions this space has, and the harder the search is: searching for a needle in a 300-dimensional haystack is much trickier than in 3 dimensions. Fortunately, since the cost function is convex in the case of linear regression, the needle is simply at the bottom of the bowl.

## Batch Gradient Descent

Most models have more than one model parameter. Therefore, to implement gradient descent, you need to compute the gradient of the cost function with regard to each model parameter  $\theta_j$ . In other words, you need to calculate how much the cost function will change if you change  $\theta_j$  just a little bit. This is called a *partial derivative*. It is like asking, "What is the slope of the mountain towards the east?" and then asking the same question facing north (and so on for all other dimensions, if you can imagine a universe with more than three dimensions). [Equation 4-5](#) computes the partial derivative of the MSE with regard to parameter  $\theta_j$ , noted  $\partial \text{MSE}(\boldsymbol{\theta}) / \partial \theta_j$ .

*Equation 4-5. Partial derivatives of the cost function*

$$\frac{\partial}{\partial \theta_j} \text{MSE}(\boldsymbol{\theta}) = \frac{2}{m} \sum_{i=1}^m (\boldsymbol{\theta}^\top \mathbf{x}^{(i)} - y^{(i)}) x_j^{(i)}$$

Instead of computing these partial derivatives individually, you can use [Equation 4-6](#) to compute them all in one go. The gradient vector, noted  $\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta})$ , contains all the partial derivatives of the cost function (one for each model parameter).

*Equation 4-6. Gradient vector of the cost function*

$$\nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial \theta_0} \text{MSE}(\boldsymbol{\theta}) \\ \frac{\partial}{\partial \theta_1} \text{MSE}(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial \theta_n} \text{MSE}(\boldsymbol{\theta}) \end{pmatrix} = \frac{2}{m} \mathbf{X}^\top (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})$$

## WARNING

Notice that this formula involves calculations over the full training set  $\mathbf{X}$ , at each gradient descent step! This is why the algorithm is called *batch gradient descent*: it uses the whole batch of training data at every step (actually, *full gradient descent* would probably be a better name). As a result, it is terribly slow on very large training sets (we will look at some much faster gradient descent algorithms shortly). However, gradient descent scales well with the number of features; training a linear regression model when there are hundreds of thousands of features is much faster using gradient descent than using the Normal equation or SVD decomposition.

Once you have the gradient vector, which points uphill, just go in the opposite direction to go downhill. This means subtracting  $\nabla_{\theta} \text{MSE}(\theta)$  from  $\theta$ . This is where the learning rate  $\eta$  comes into play:<sup>4</sup> multiply the gradient vector by  $\eta$  to determine the size of the downhill step ([Equation 4-7](#)).

*Equation 4-7. Gradient descent step*

$$\theta^{(\text{next step})} = \theta - \eta \nabla_{\theta} \text{MSE}(\theta)$$

Let's look at a quick implementation of this algorithm:

```
eta = 0.1 # learning rate
n_epochs = 1000
m = len(X_b) # number of instances

rng = np.random.default_rng(seed=42)
theta = rng.standard_normal((2, 1)) # randomly initialized model parameters

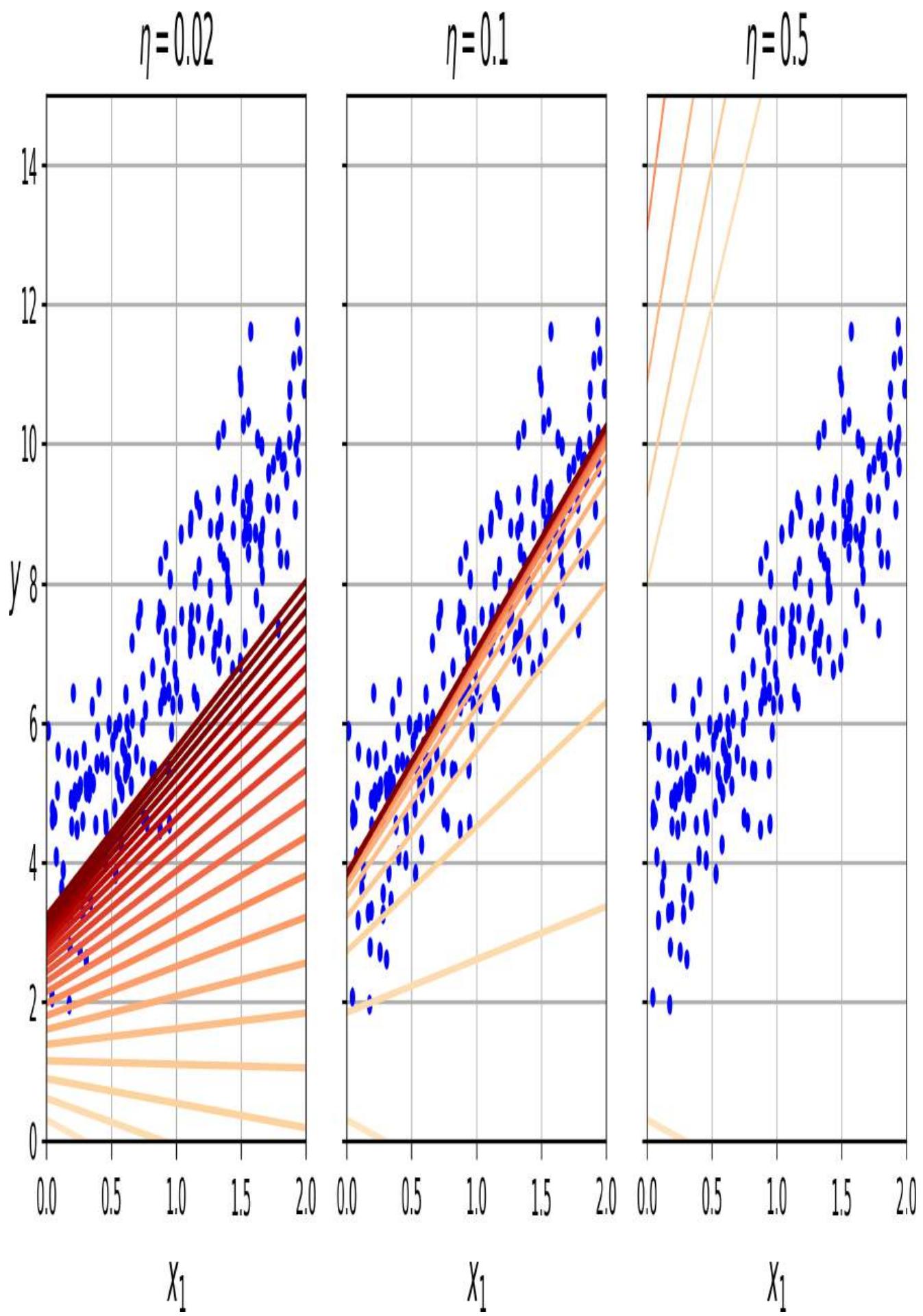
for epoch in range(n_epochs):
    gradients = 2 / m * X_b.T @ (X_b @ theta - y)
    theta = theta - eta * gradients
```

That wasn't too hard! Each iteration over the training set is called an *epoch*. Let's look at the resulting `theta`:

```
>>> theta
array([[3.69084138],
       [3.32960458]])
```

Hey, that's exactly what the Normal equation found! Gradient descent worked perfectly. But what if you had used a different learning rate (`eta`)? [Figure 4-8](#) shows the first 20 steps of gradient descent using three different learning rates. The line at the bottom of

each plot represents the random starting point, then each epoch is represented by a darker and darker line.



*Figure 4-8. Gradient descent with various learning rates*

On the left, the learning rate is too low: the algorithm will eventually reach the solution, but it will take a long time. In the middle, the learning rate looks pretty good: in just a few epochs, it has already converged to the solution. On the right, the learning rate is too high: the algorithm diverges, jumping all over the place and actually getting further and further away from the solution at every step.

To find a good learning rate, you can use grid search (see [Chapter 2](#)). However, you may want to limit the number of epochs so that grid search can eliminate models that take too long to converge.

You may wonder how to set the number of epochs. If it is too low, you will still be far away from the optimal solution when the algorithm stops; but if it is too high, you will waste time while the model parameters do not change anymore. A simple solution is to set a very large number of epochs but to interrupt the algorithm when the gradient vector becomes tiny—that is, when its norm becomes smaller than a tiny number  $\epsilon$  (called the *tolerance*)—because this happens when gradient descent has (almost) reached the minimum.

## CONVERGENCE RATE

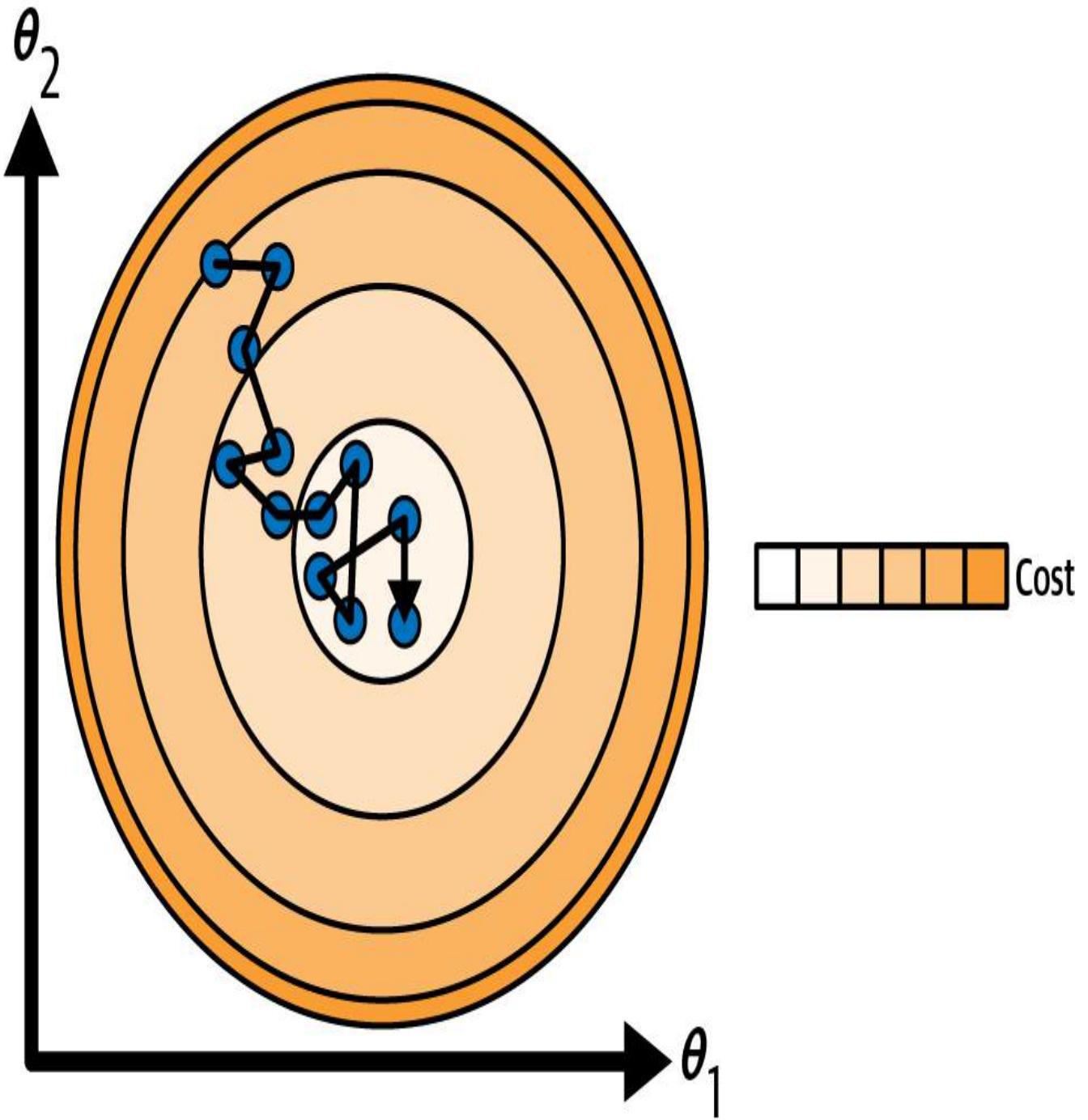
When the cost function is convex and its slope does not change abruptly (as is the case for the MSE cost function), batch gradient descent with a fixed learning rate will eventually converge to the optimal solution, but you may have to wait a while: it can take  $O(1/\epsilon)$  iterations to reach the optimum within a range of  $\epsilon$ , depending on the shape of the cost function. If you divide the tolerance by 10 to have a more precise solution, then the algorithm may have to run about 10 times longer.

## Stochastic Gradient Descent

The main problem with batch gradient descent is the fact that it uses the whole training set to compute the gradients at every step, which makes it very slow when the training set is large. At the opposite extreme, *stochastic gradient descent* picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration. It also makes it possible to train on huge training sets, since only one instance needs to be in memory

at each iteration (stochastic GD can be implemented as an out-of-core algorithm; see [Chapter 1](#)).

On the other hand, due to its stochastic (i.e., random) nature, this algorithm is much less regular than batch gradient descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down (see [Figure 4-9](#)). Once the algorithm stops, the final parameter values will be good, but not optimal.



*Figure 4-9. With stochastic gradient descent, each training step is much faster but also much more stochastic than when using batch gradient descent*

When the cost function is very irregular (as in [Figure 4-6](#)), this can actually help the algorithm jump out of local minima, so stochastic gradient descent has a better chance of finding the global minimum than batch gradient descent does.

Therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum. One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick

progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum. This process is akin to *simulated annealing*, an algorithm inspired by the process in metallurgy of annealing, where molten metal is slowly cooled down. The function that determines the learning rate at each iteration is called the *learning schedule*. If the learning rate is reduced too quickly, you may get stuck in a local minimum, or even end up frozen halfway to the minimum. If the learning rate is reduced too slowly, you may jump around the minimum for a long time and end up with a suboptimal solution if you halt training too early.

This code implements stochastic gradient descent using a simple learning schedule:

```

n_epochs = 50
t0, t1 = 5, 50 # learning schedule hyperparameters

def learning_schedule(t):
    return t0 / (t + t1)

rng = np.random.default_rng(seed=42)
theta = rng.standard_normal((2, 1)) # randomly initialized model parameters

for epoch in range(n_epochs):
    for iteration in range(m):
        random_index = rng.integers(m)
        xi = X_b[random_index : random_index + 1]
        yi = y[random_index : random_index + 1]
        gradients = 2 * xi.T @ (xi @ theta - yi) # for SGD, do not divide by m
        eta = learning_schedule(epoch * m + iteration)
        theta = theta - eta * gradients

```

By convention we iterate by rounds of  $m$  iterations; each round is called an *epoch*, as earlier. While the batch gradient descent code iterated 1,000 times through the whole training set, this code goes through the training set only 50 times and reaches a pretty good solution:

```

>>> theta
array([[3.69826475],
       [3.30748311]])

```

**Figure 4-10** shows the first 20 steps of training (notice how irregular the steps are).

Note that since instances are picked randomly, some instances may be picked several times per epoch, while others may not be picked at all. If you want to be sure that the algorithm goes through every instance at each epoch, another approach is to shuffle the training set (making sure to shuffle the input features and the labels jointly), then go

through it instance by instance, then shuffle it again, and so on. However, this approach is more complex, and it generally does not improve the result.

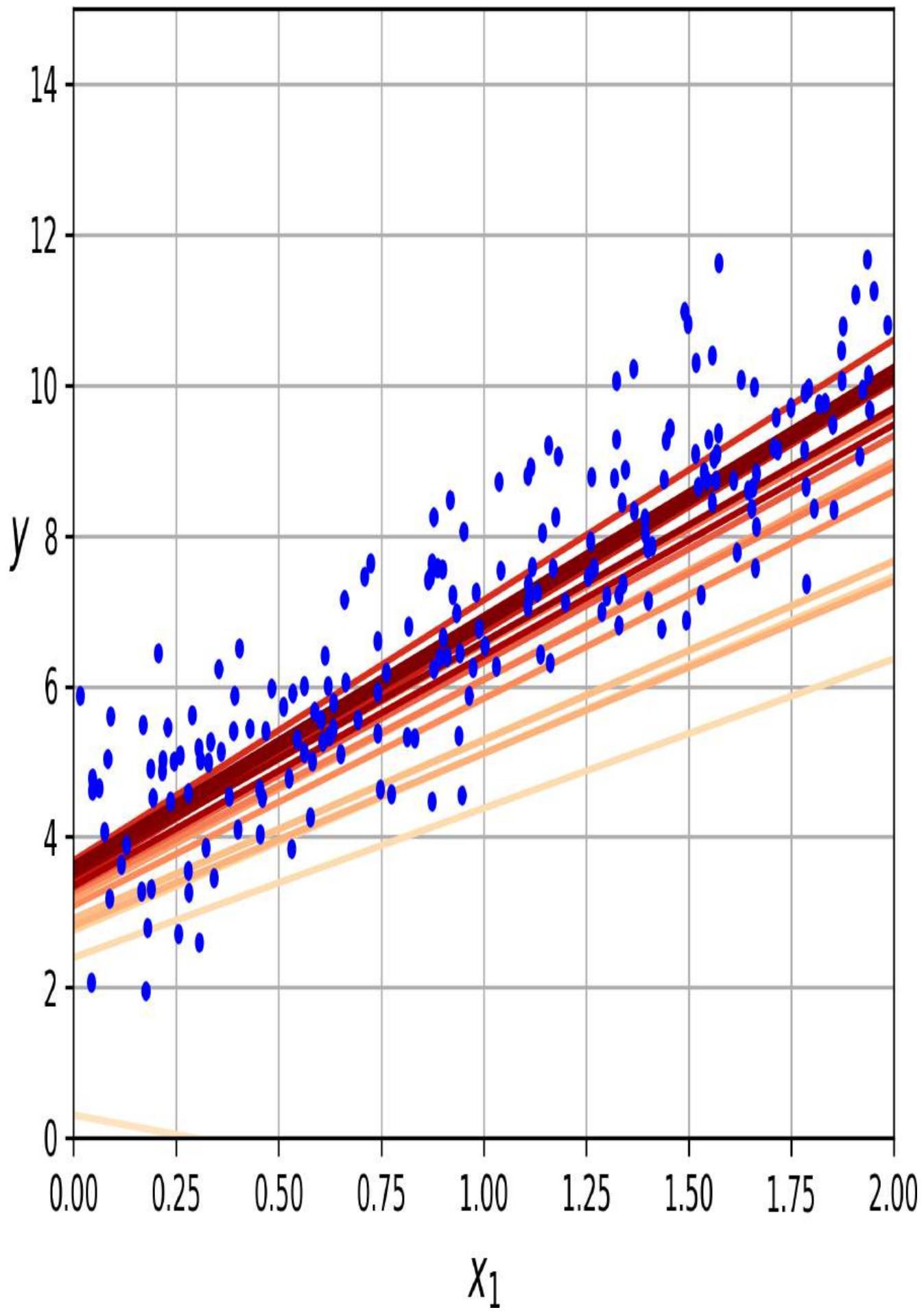


Figure 4-10. The first 20 steps of stochastic gradient descent

## WARNING

When using stochastic gradient descent, the training instances must be independent and identically distributed (IID) to ensure that the parameters get pulled toward the global optimum, on average. A simple way to ensure this is to shuffle the instances during training (e.g., pick each instance randomly, or shuffle the training set at the beginning of each epoch). If you do not shuffle the instances—for example, if the instances are sorted by label—then SGD will start by optimizing for one label, then the next, and so on, and it will not settle close to the global minimum.

To perform linear regression using stochastic GD with Scikit-Learn, you can use the `SGDRegressor` class, which defaults to optimizing the MSE cost function. The following code runs for maximum 1,000 epochs (`max_iter`) or until the loss drops by less than  $10^{-5}$  (`tol`) during 100 epochs (`n_iter_no_change`). It starts with a learning rate of 0.01 (`eta0`), using the default learning schedule (different from the one we used). Lastly, it does not use any regularization (`penalty=None`; more details on this shortly):

```
from sklearn.linear_model import SGDRegressor

sgd_reg = SGDRegressor(max_iter=1000, tol=1e-5, penalty=None, eta0=0.01,
                      n_iter_no_change=100, random_state=42)
sgd_reg.fit(X, y.ravel()) # y.ravel() because fit() expects 1D targets
```

Once again, you find a solution quite close to the one returned by the Normal equation:

```
>>> sgd_reg.intercept_, sgd_reg.coef_
(array([3.68899733]), array([3.33054574]))
```

## TIP

All Scikit-Learn estimators can be trained using the `fit()` method, but some estimators also have a `partial_fit()` method that you can call to run a single round of training on one or more instances (it ignores hyperparameters like `max_iter` or `tol`). Repeatedly calling `partial_fit()` will gradually train the model. This is useful when you need more control over the training process. Other models have a `warm_start` hyperparameter instead (and some have both): if you set `warm_start=True`, calling the `fit()` method on a trained model will not reset the model; it will just continue training where it left off, respecting hyperparameters like `max_iter` and `tol`. Note that `fit()` resets the iteration counter used by the learning schedule, while `partial_fit()` does not.

## Mini-Batch Gradient Descent

The last gradient descent algorithm we will look at is called *mini-batch gradient descent*. It is straightforward once you know batch and stochastic gradient descent: at each step, instead of computing the gradients based on the full training set (as in batch GD) or based on just one instance (as in stochastic GD), mini-batch GD computes the gradients on small random sets of instances called *mini-batches*. The main advantage of mini-batch GD over stochastic GD is that you can get a performance boost from hardware acceleration of matrix operations, especially when using *graphical processing units* (GPUs).

The algorithm's progress in parameter space is less erratic than with stochastic GD, especially with fairly large mini-batches. As a result, mini-batch GD will end up walking around a bit closer to the minimum than stochastic GD—but it may be harder for it to escape from local minima (in the case of problems that suffer from local minima, unlike linear regression with the MSE cost function). [Figure 4-11](#) shows the paths taken by the three gradient descent algorithms in parameter space during training. They all end up near the minimum, but batch GD's path actually stops at the minimum, while both stochastic GD and mini-batch GD continue to walk around. However, don't forget that batch GD takes a lot of time to take each step, and stochastic GD and mini-batch GD would also reach the minimum if you used a good learning schedule.

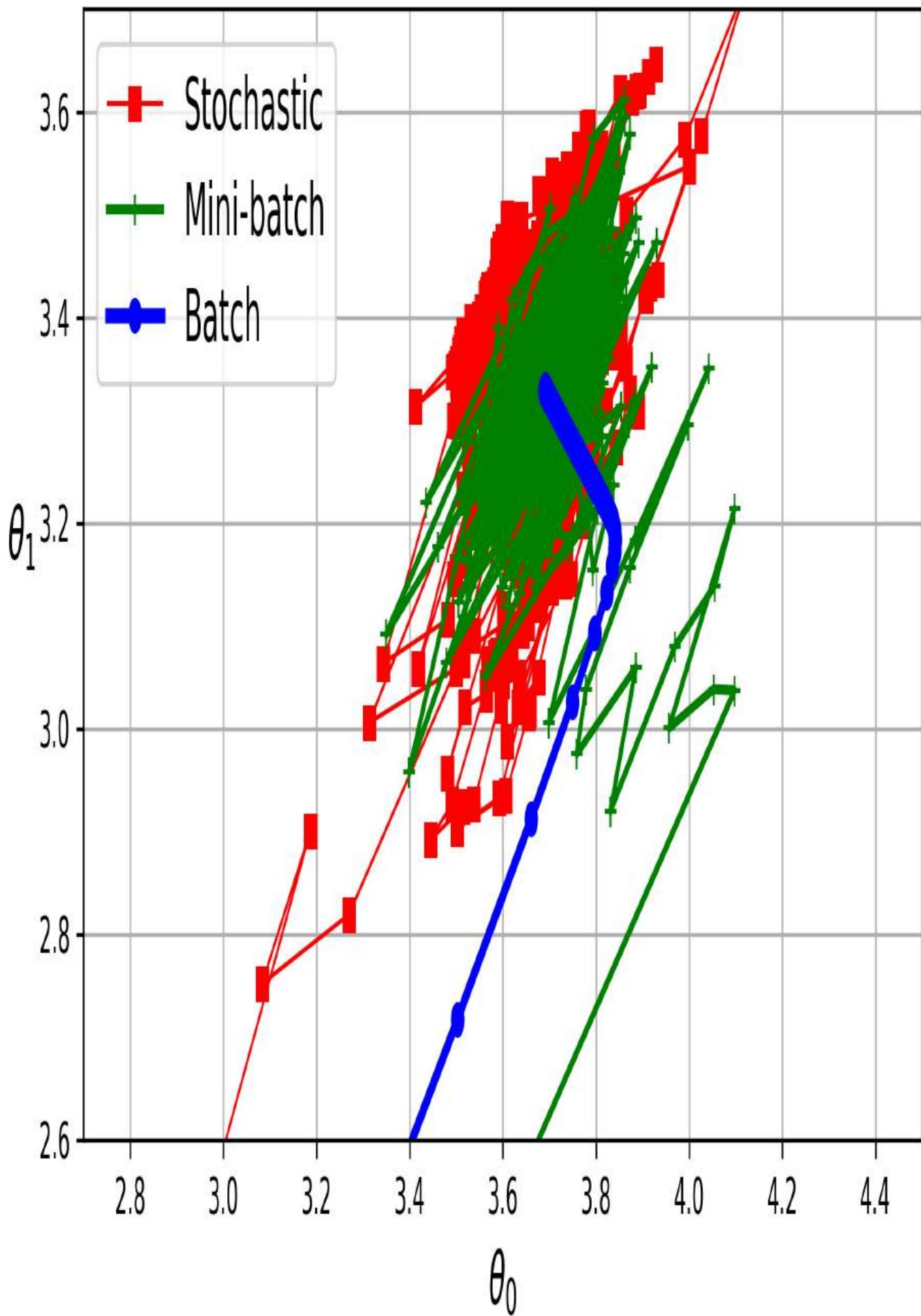


Figure 4-11. Gradient descent paths in parameter space

Table 4-1 compares the algorithms we've discussed so far for linear regression<sup>5</sup> (recall that  $m$  is the number of training instances and  $n$  is the number of features).

Table 4-1. Comparison of algorithms for linear regression

Algorithm	Large $m$	Out-of-core support	Large $n$	Hyperparams
Normal equation	Fast	No	Slow	0
SVD	Fast	No	Slow	0
Batch GD	Slow	No	Fast	2
Stochastic GD	Fast	Yes	Fast	$\geq 2$
Mini-batch GD	Fast	Yes	Fast	$\geq 2$

There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

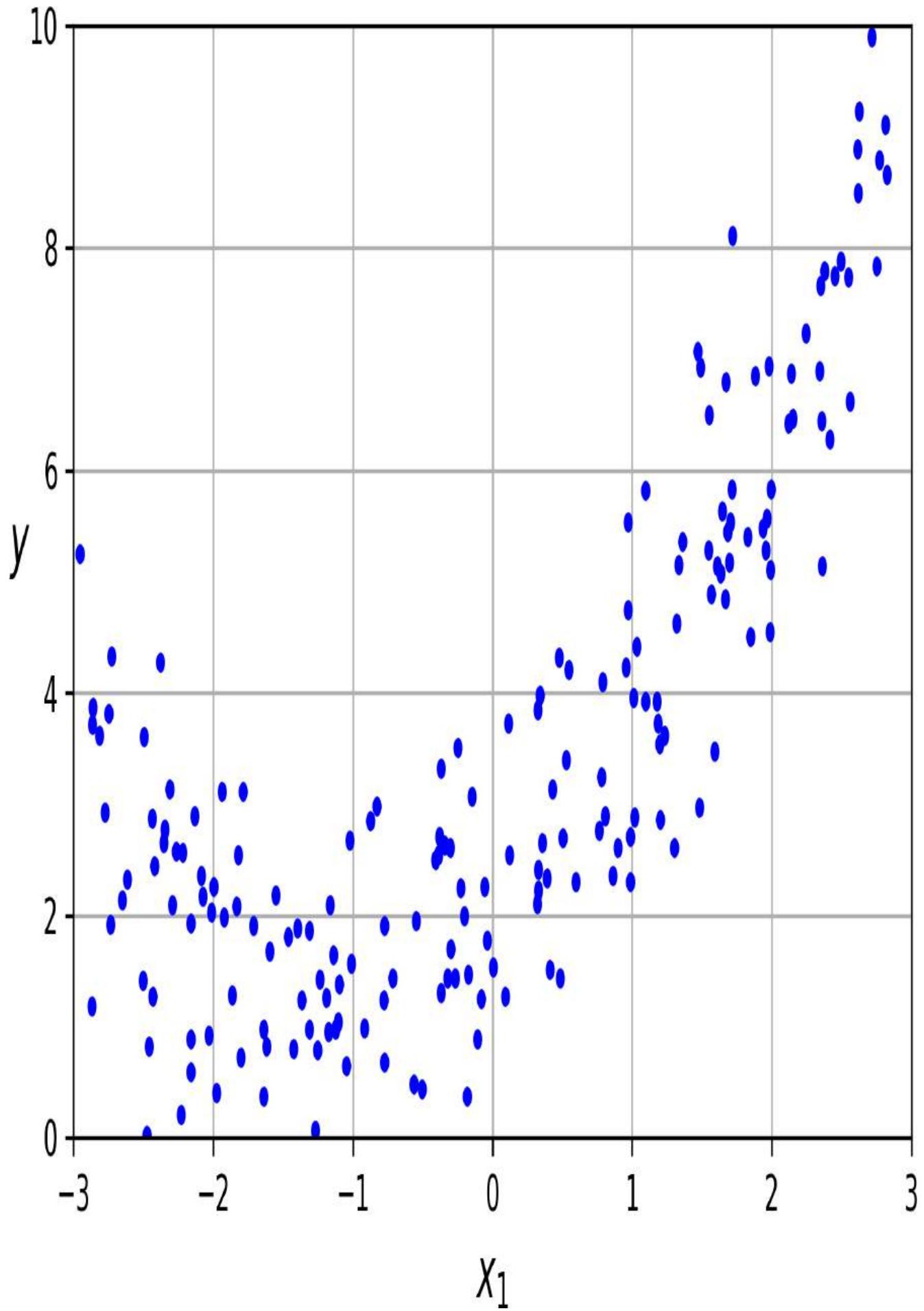
## Polynomial Regression

What if your data is more complex than a straight line? Surprisingly, you can use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called *polynomial regression*.

Let's look at an example. First, we'll generate some nonlinear data (see Figure 4-12), based on a simple *quadratic equation*—that's an equation of the form  $y = ax^2 + bx + c$ —plus some noise:

```
rng = np.random.default_rng(seed=42)
m = 200 # number of instances
```

```
X = 6 * rng.random((m, 1)) - 3
y = 0.5 * X ** 2 + X + 2 + rng.standard_normal((m, 1))
```



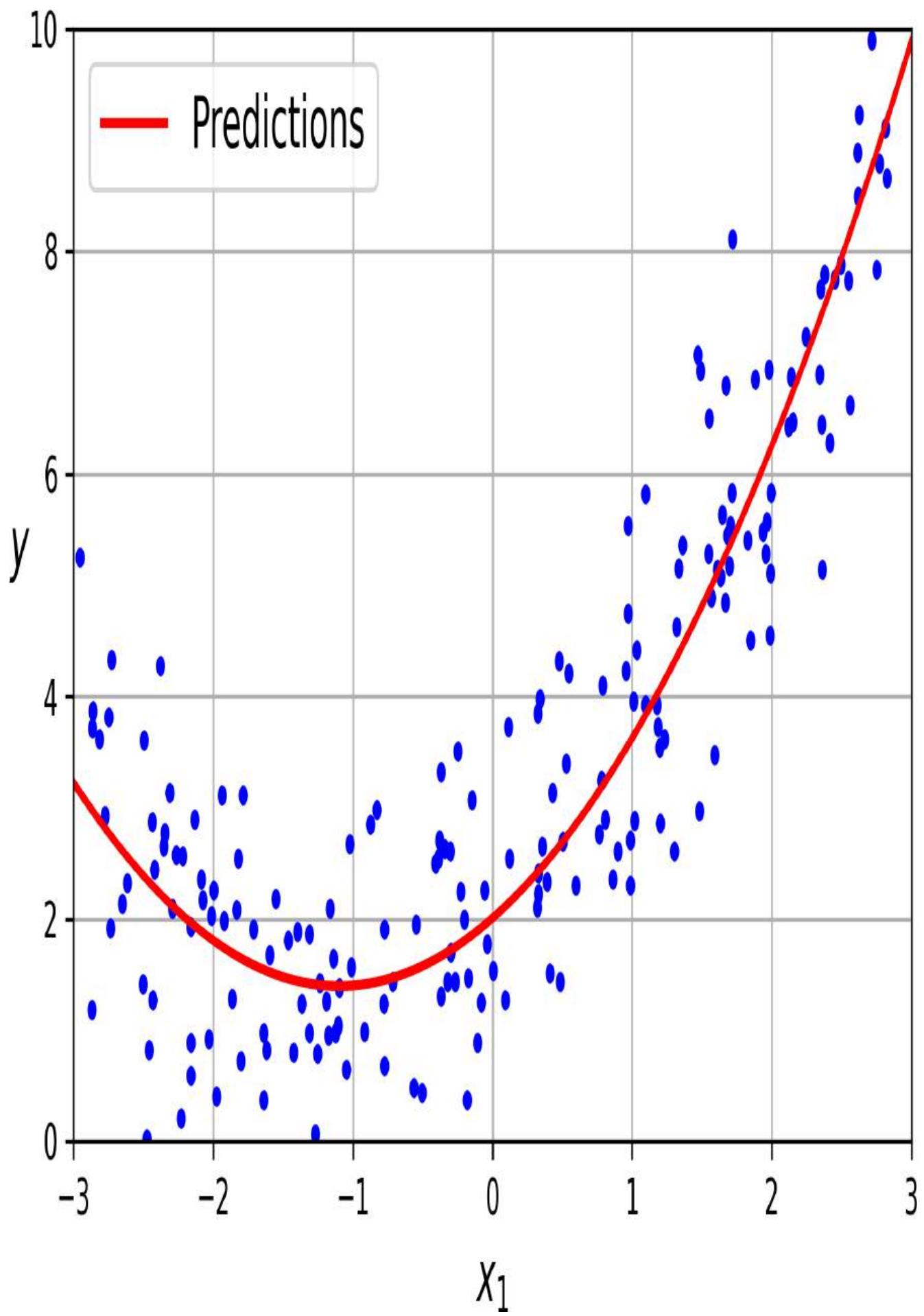
*Figure 4-12. Generated nonlinear and noisy dataset*

Clearly, a straight line will never fit this data properly. So let's use Scikit-Learn's `PolynomialFeatures` class to transform our training data, adding the square (second-degree polynomial) of each feature in the training set as a new feature (in this case there is just one feature):

```
>>> from sklearn.preprocessing import PolynomialFeatures  
>>> poly_features = PolynomialFeatures(degree=2, include_bias=False)  
>>> X_poly = poly_features.fit_transform(X)  
>>> X[0]  
array([1.64373629])  
>>> X_poly[0]  
array([1.64373629, 2.701869])
```

`X_poly` now contains the original feature of `X` plus the square of this feature. Now we can fit a `LinearRegression` model to this extended training data ([Figure 4-13](#)):

```
>>> lin_reg = LinearRegression()  
>>> lin_reg.fit(X_poly, y)  
>>> lin_reg.intercept_, lin_reg.coef_  
(array([2.00540719]), array([[1.11022126, 0.50526985]]))
```



*Figure 4-13. Polynomial regression model predictions*

Not bad: the model estimates  $\hat{y} = 0.56x_1^2 + 0.93x_1 + 1.78$  when in fact the original function was  $y = 0.5x_1^2 + 1.0x_1 + 2.0 + \text{Gaussian noise}$ .

Note that when there are multiple features, polynomial regression is capable of finding relationships between features, which is something a plain linear regression model cannot do. This is made possible by the fact that `PolynomialFeatures` also adds all combinations of features up to the given degree. For example, if there were two features  $a$  and  $b$ , `PolynomialFeatures` with `degree=3` would not only add the features  $a^2$ ,  $a^3$ ,  $b^2$ , and  $b^3$ , but also the combinations  $ab$ ,  $a^2b$ , and  $ab^2$ .

### WARNING

`PolynomialFeatures(degree=d)` transforms an array containing  $n$  features into an array containing  $(n + d)! / d!n!$  features, where  $n!$  is the *factorial* of  $n$ , equal to  $1 \times 2 \times 3 \times \dots \times n$ . Beware of the combinatorial explosion of the number of features!

## Learning Curves

If you perform high-degree polynomial regression, you will likely fit the training data much better than with plain linear regression. For example, [Figure 4-14](#) applies a 300-degree polynomial model to the preceding training data, and compares the result with a pure linear model and a quadratic model (second-degree polynomial). Notice how the 300-degree polynomial model wiggles around to get as close as possible to the training instances.

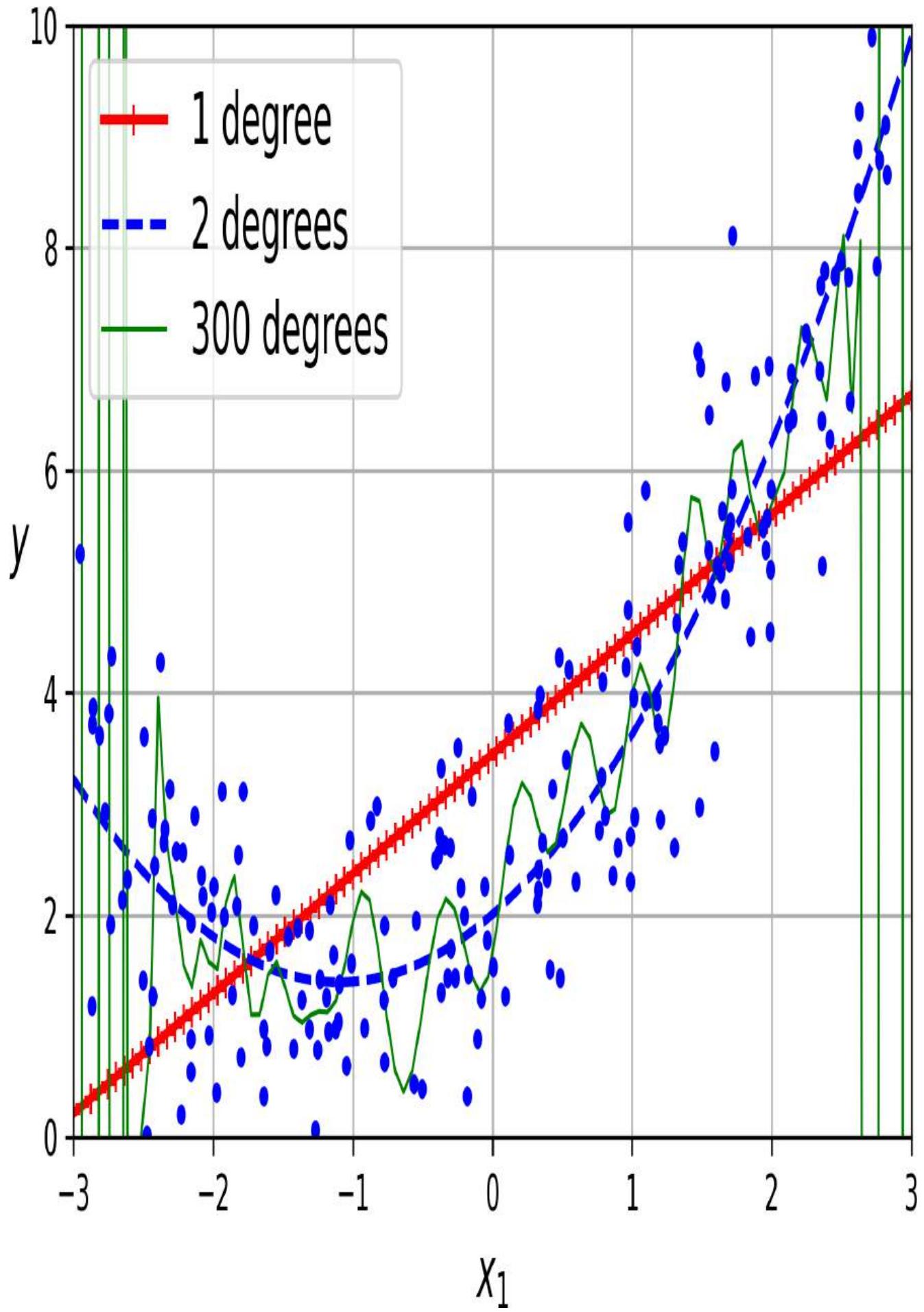


Figure 4-14. High-degree polynomial regression

This high-degree polynomial regression model is severely overfitting the training data, while the linear model is underfitting it. The model that will generalize best in this case is the quadratic model, which makes sense because the data was generated using a quadratic model. But in general you won't know what function generated the data, so how can you decide how complex your model should be? How can you tell that your model is overfitting or underfitting the data?

In [Chapter 2](#) you used cross-validation to get an estimate of a model's generalization performance. If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting. If it performs poorly on both, then it is underfitting. This is one way to tell when a model is too simple or too complex.

Another way to tell is to look at the *learning curves*, which are plots of the model's training error and validation error as a function of the training iteration: just evaluate the model at regular intervals during training on both the training set and the validation set, and plot the results. If the model cannot be trained incrementally (i.e., if it does not support `partial_fit()` or `warm_start`), then you must train it several times on gradually larger subsets of the training set.

Scikit-Learn has a useful `learning_curve()` function to help with this: it trains and evaluates the model using cross-validation. By default it retrains the model on growing subsets of the training set, but if the model supports incremental learning you can set `exploit_incremental_learning=True` when calling `learning_curve()` and it will train the model incrementally instead. The function returns the training set sizes at which it evaluated the model, and the training and validation scores it measured for each size and for each cross-validation fold. Let's use this function to look at the learning curves of the plain linear regression model (see [Figure 4-15](#)):

```
from sklearn.model_selection import learning_curve

train_sizes, train_scores, valid_scores = learning_curve(
    LinearRegression(), X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error")
train_errors = -train_scores.mean(axis=1)
valid_errors = -valid_scores.mean(axis=1)

plt.plot(train_sizes, train_errors, "r-+", linewidth=2, label="train")
plt.plot(train_sizes, valid_errors, "b-", linewidth=3, label="valid")
[...] # beautify the figure: add labels, axis, grid, and legend
plt.show()
```

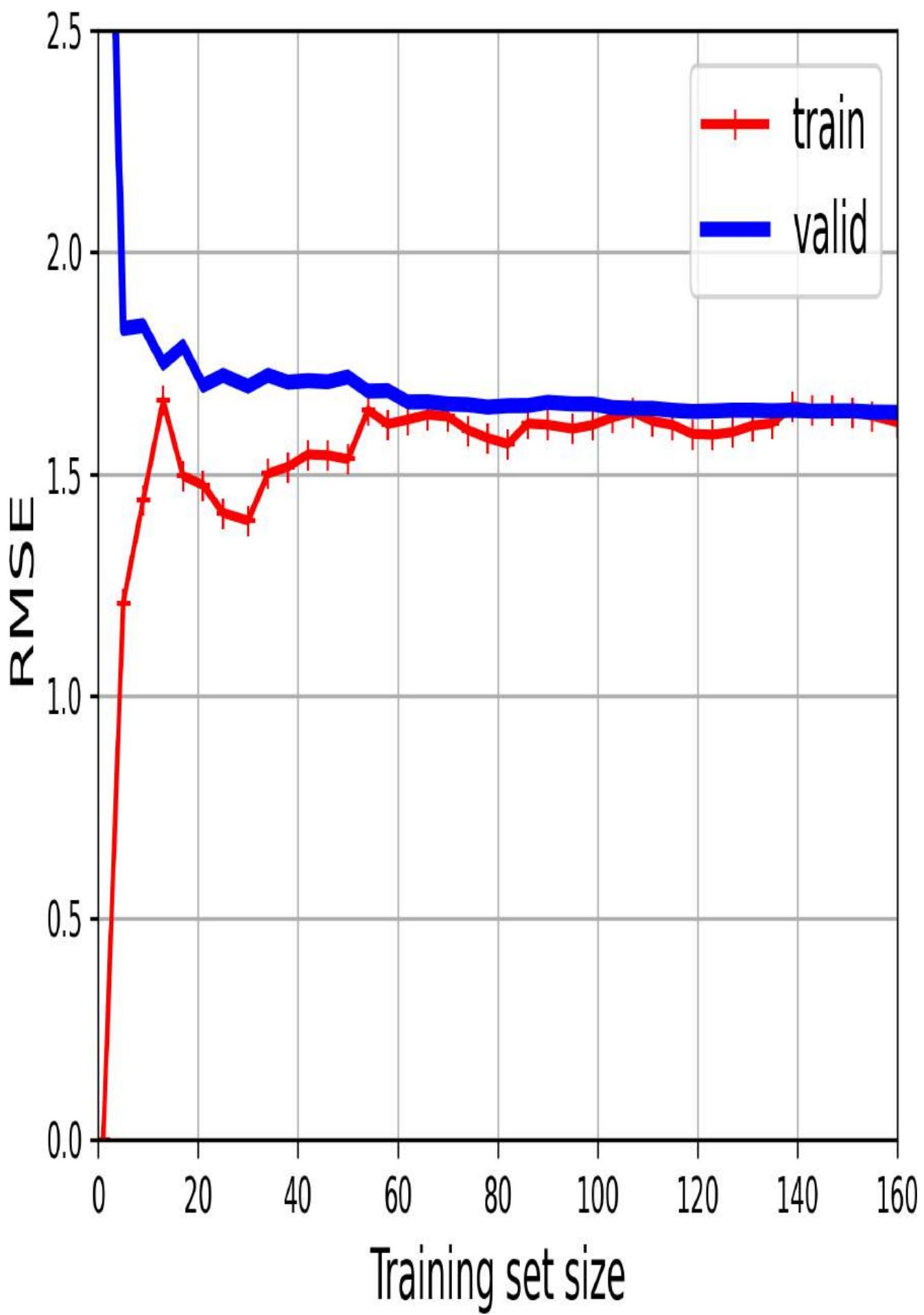


Figure 4-15. Learning curves

This model is underfitting, it's too simple for the data. How can we tell? Well, let's look at the training error. When there are just one or two instances in the training set, the model can fit them perfectly, which is why the curve starts at zero. But as new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all. So the error on the training data goes up until it reaches a plateau, at which point adding new instances to the training set doesn't make the average error much better or worse. Now let's look at the validation error. When the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite large. Then, as the model is shown more training examples, it learns, and thus the validation error slowly goes down. However, once again a straight line cannot do a good job of modeling the data, so the error ends up at a plateau, very close to the other curve.

These learning curves are typical of a model that's underfitting. Both curves have reached a plateau; they are close and fairly high.

### TIP

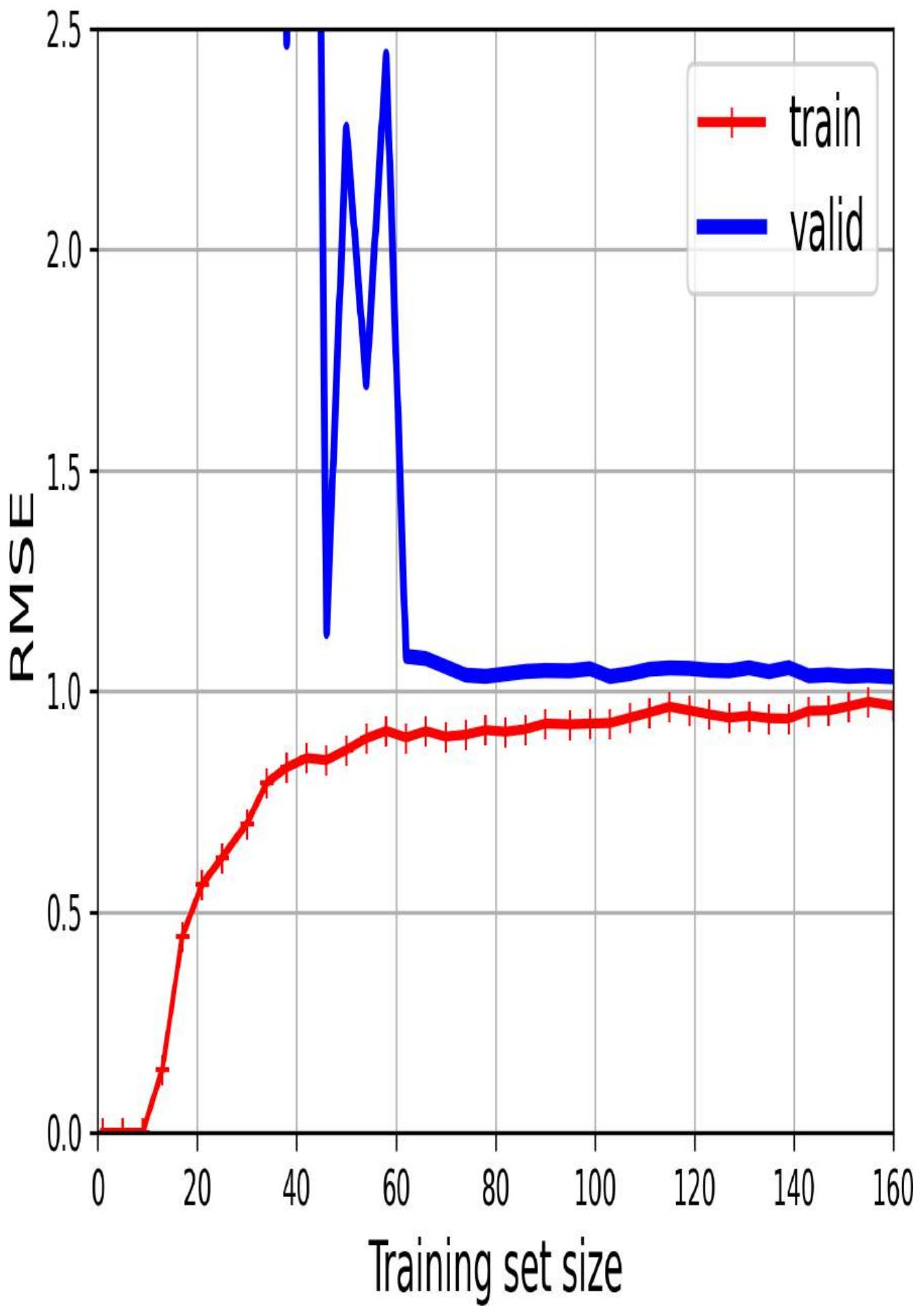
If your model is underfitting the training data, adding more training examples will not help. You need to use a better model or come up with better features.

Now let's look at the learning curves of a 10th-degree polynomial model on the same data (Figure 4-16):

```
from sklearn.pipeline import make_pipeline

polynomial_regression = make_pipeline(
    PolynomialFeatures(degree=10, include_bias=False),
    LinearRegression())

train_sizes, train_scores, valid_scores = learning_curve(
    polynomial_regression, X, y, train_sizes=np.linspace(0.01, 1.0, 40), cv=5,
    scoring="neg_root_mean_squared_error")
[...] # same as earlier
```



*Figure 4-16. Learning curves for the 10th-degree polynomial model*

These learning curves look a bit like the previous ones, but there are two very important differences:

- The error on the training data is much lower than before.
- There is a gap between the curves. This means that the model performs better on the training data than on the validation data, which is the hallmark of an overfitting model. If you used a much larger training set, however, the two curves would continue to get closer.

**TIP**

One way to improve an overfitting model is to feed it more training data until the validation error gets close enough to the training error.

## THE BIAS/VARIANCE TRADE-OFF

An important theoretical result of statistics and machine learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

### *Bias*

This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.<sup>6</sup>

### *Variance*

This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance and thus overfit the training data.

### *Irreducible error*

This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity (or increasing regularization) increases its bias and reduces its variance (see [Figure 4-17](#)). This is why it is called a trade-off.

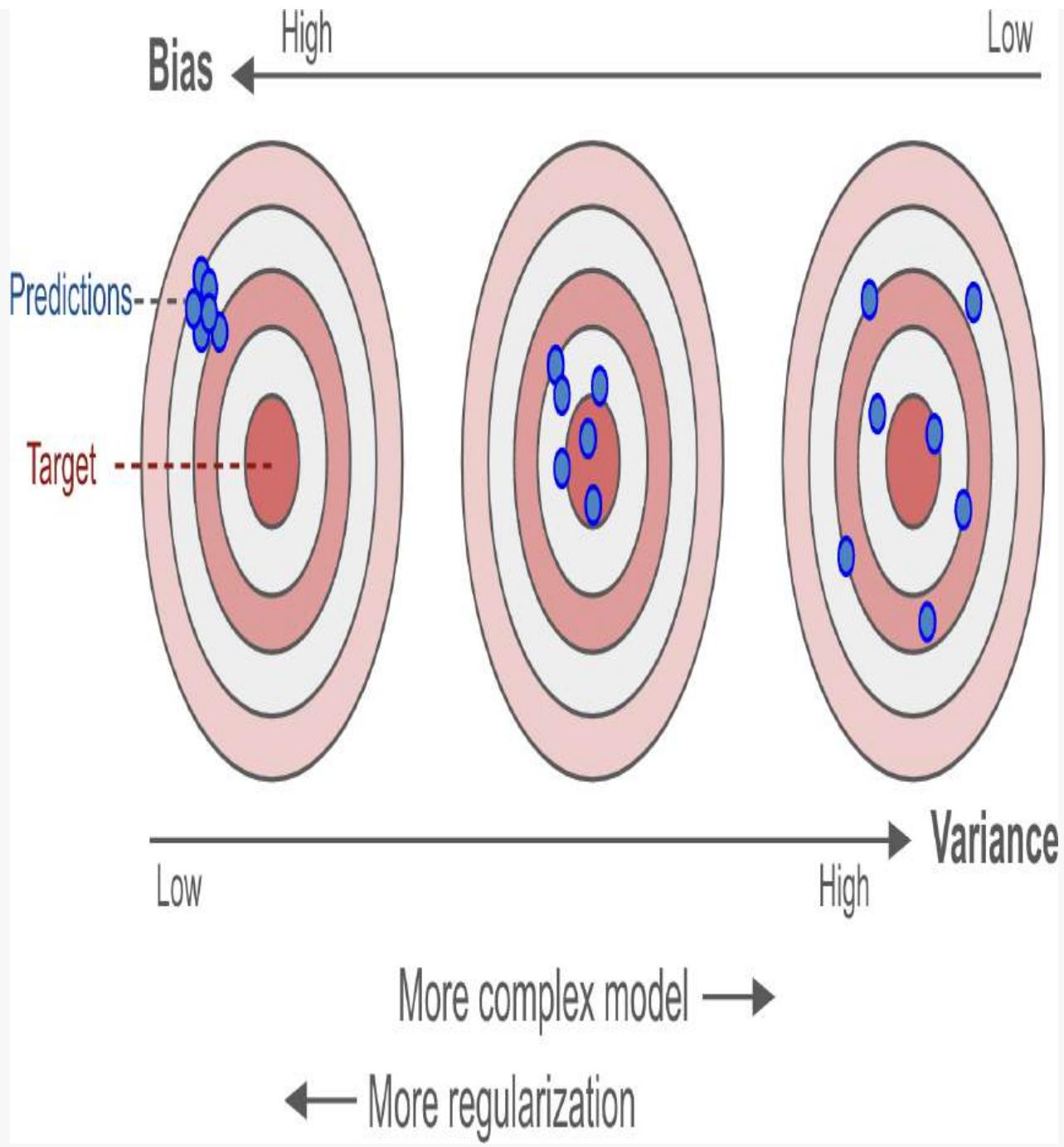


Figure 4-17. Bias/variance trade-off

## Regularized Linear Models

As you saw in Chapters 1 and 2, a good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data. A simple way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, regularization is typically achieved by constraining the weights of the model. We will now look at ridge regression, lasso regression, and elastic net regression, which implement three different ways to constrain the weights.

## Ridge Regression

*Ridge regression* (also called *Tikhonov regularization*) is a regularized version of linear regression: a *regularization term* equal to  $\frac{\alpha}{m} \sum_{i=1}^n \theta_i^2$  is added to the MSE. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. This constraint makes the model less flexible, preventing it from stretching itself too much to fit every data point: this reduces the risk of overfitting. Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to use the unregularized MSE (or the RMSE) to evaluate the model's performance.

The hyperparameter  $\alpha$  controls how much you want to regularize the model. If  $\alpha = 0$ , then ridge regression is just linear regression. If  $\alpha$  is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean. [Equation 4-8](#) presents the ridge regression cost function.<sup>7</sup>

*Equation 4-8. Ridge regression cost function*

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + \frac{\alpha}{m} \sum_{i=1}^n \theta_i^2$$

Note that the bias term  $\theta_0$  is not regularized (the sum starts at  $i = 1$ , not 0). If we define  $\mathbf{w}$  as the vector of feature weights ( $\theta_1$  to  $\theta_n$ ), then the regularization term is equal to  $\alpha(\|\mathbf{w}\|_2)^2 / m$ , where  $\|\mathbf{w}\|_2$  represents the  $\ell_2$  norm of the weight vector.<sup>8</sup> For batch gradient descent, just add  $2\alpha\mathbf{w} / m$  to the part of the MSE gradient vector that corresponds to the feature weights, without adding anything to the gradient of the bias term (see [Equation 4-6](#)).

### WARNING

It is important to scale the data (e.g., using a `StandardScaler`) before performing ridge regression, as it is sensitive to the scale of the input features. This is true of most regularized models.

[Figure 4-18](#) shows several ridge models that were trained on some very noisy linear data using different  $\alpha$  values. On the left, plain ridge models are used, leading to linear

predictions. On the right, the data is first expanded using `PolynomialFeatures(degree=10)`, then it is scaled using a `StandardScaler`, and finally the ridge models are applied to the resulting features: this is polynomial regression with ridge regularization. Note how increasing  $\alpha$  leads to flatter (i.e., less extreme, more reasonable) predictions, thus reducing the model's variance but increasing its bias.

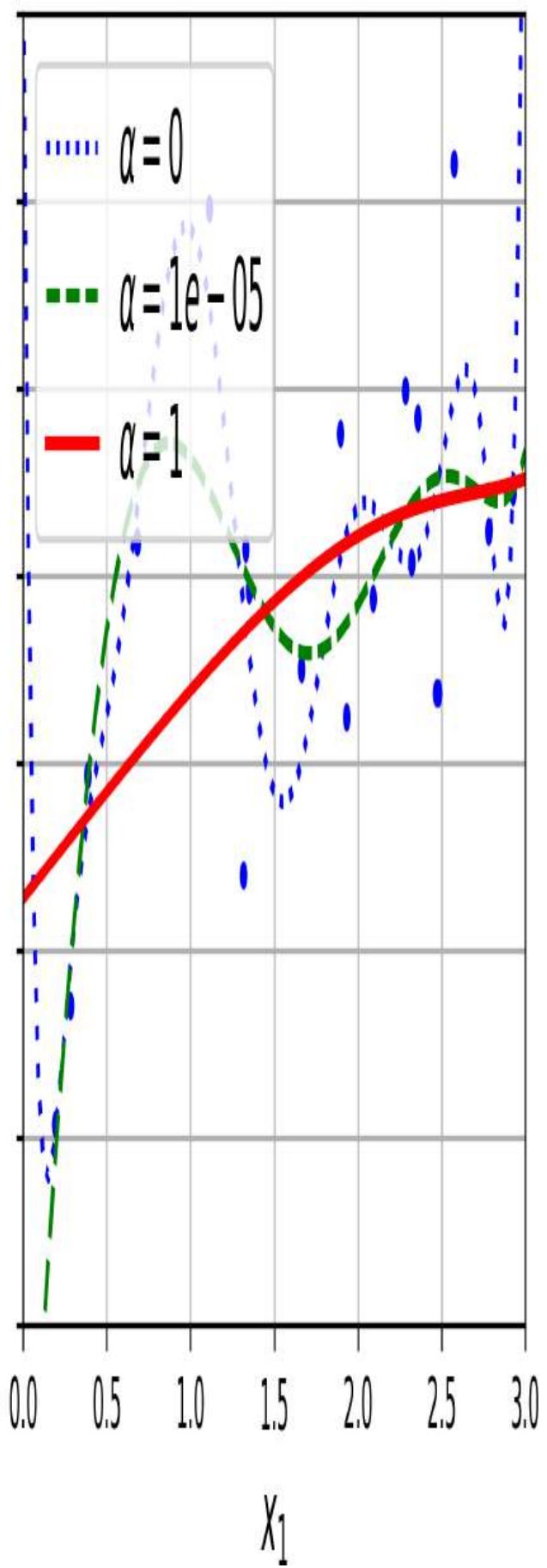
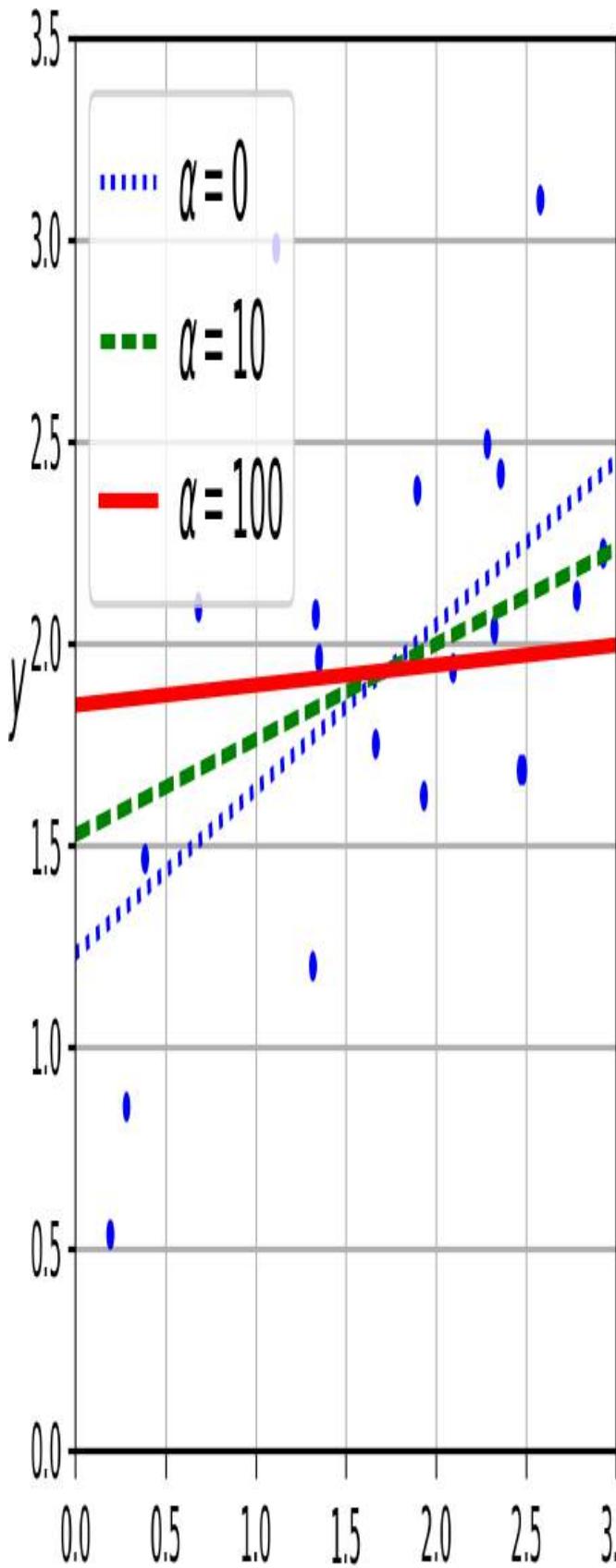


Figure 4-18. Linear (left) and a polynomial (right) models, both with various levels of ridge regularization

As with linear regression, we can perform ridge regression either by computing a closed-form equation or by performing gradient descent. The pros and cons are the same. [Equation 4-9](#) shows the closed-form solution, where  $\mathbf{A}$  is the  $(n + 1) \times (n + 1)$  *identity matrix*,<sup>9</sup> except with a 0 in the top-left cell, corresponding to the bias term.

*Equation 4-9. Ridge regression closed-form solution*

$$\hat{\boldsymbol{\theta}} = (\mathbf{X}^\top \mathbf{X} + \alpha \mathbf{A})^{-1} \mathbf{X}^\top \mathbf{y}$$

Here is how to perform ridge regression with Scikit-Learn using a closed-form solution (a variant of [Equation 4-9](#) that uses a matrix factorization technique by André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=0.1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([1.84414523])
```

And using stochastic gradient descent:<sup>10</sup>

```
>>> sgd_reg = SGDRegressor(penalty="l2", alpha=0.1 / m, tol=None,
...                         max_iter=1000, eta0=0.01, random_state=42)
...
>>> sgd_reg.fit(X, y.ravel()) # y.ravel() because fit() expects 1D targets
>>> sgd_reg.predict([[1.5]])
array([1.83659707])
```

The `penalty` hyperparameter sets the type of regularization term to use. Specifying "`l2`" indicates that you want SGD to add a regularization term to the MSE cost function equal to `alpha` times the square of the  $\ell_2$  norm of the weight vector. This is just like ridge regression, except there's no division by  $m$  in this case; that's why we passed `alpha=0.1 / m`, to get the same result as `Ridge(alpha=0.1)`.

## TIP

The `RidgeCV` class also performs ridge regression, but it automatically tunes hyperparameters using cross-validation. It's roughly equivalent to using `GridSearchCV`, but it's optimized for ridge regression and runs *much* faster. Several other estimators (mostly linear) also have efficient CV variants, such as `LassoCV` and `ElasticNetCV`.

## Lasso Regression

*Least absolute shrinkage and selection operator regression* (usually simply called *lasso regression*) is another regularized version of linear regression: just like ridge regression, it adds a regularization term to the cost function, but it uses the  $\ell_1$  norm of the weight vector instead of the square of the  $\ell_2$  norm (see [Equation 4-10](#)). Notice that the  $\ell_1$  norm is multiplied by  $2\alpha$ , whereas the  $\ell_2$  norm was multiplied by  $\alpha / m$  in ridge regression. These factors were chosen to ensure that the optimal  $\alpha$  value is independent from the training set size: different norms lead to different factors (see [Scikit-Learn issue #15657](#) for more details).

*Equation 4-10. Lasso regression cost function*

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + 2\alpha \sum_{i=1}^n |\theta_i|$$

[Figure 4-19](#) shows the same thing as [Figure 4-18](#) but replaces the ridge models with lasso models and uses different  $\alpha$  values.

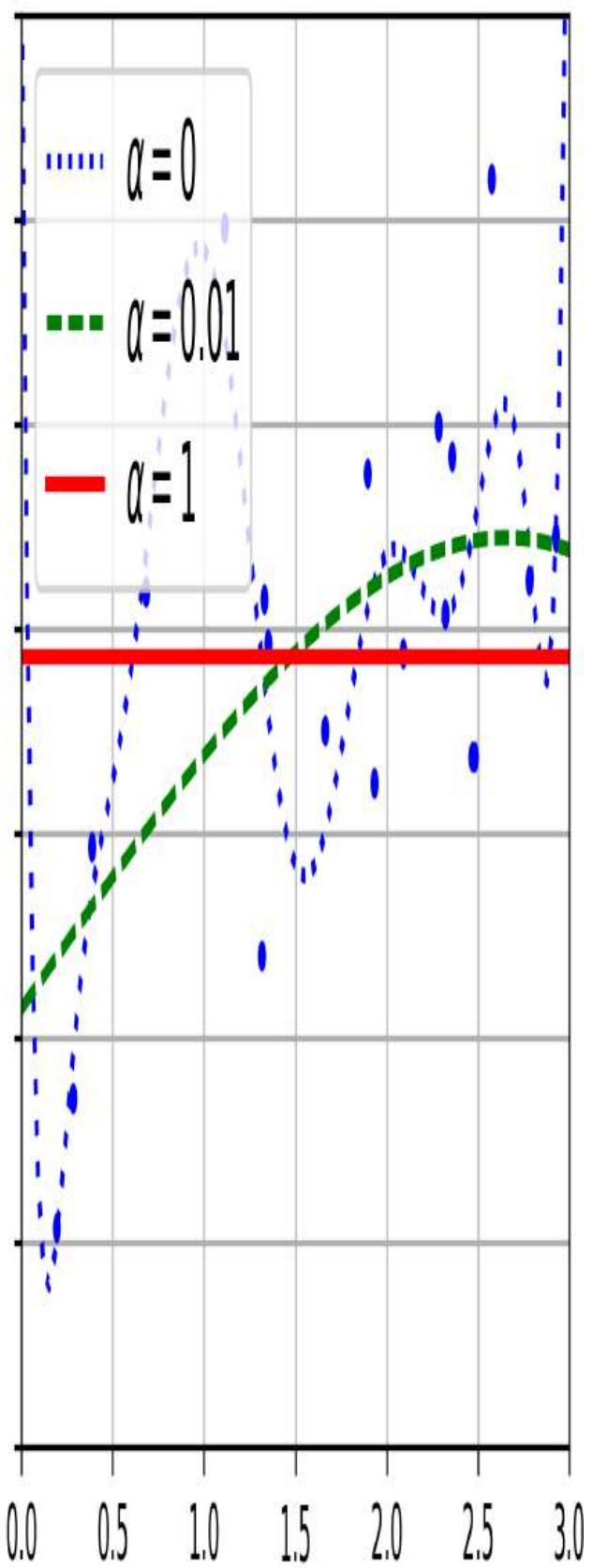
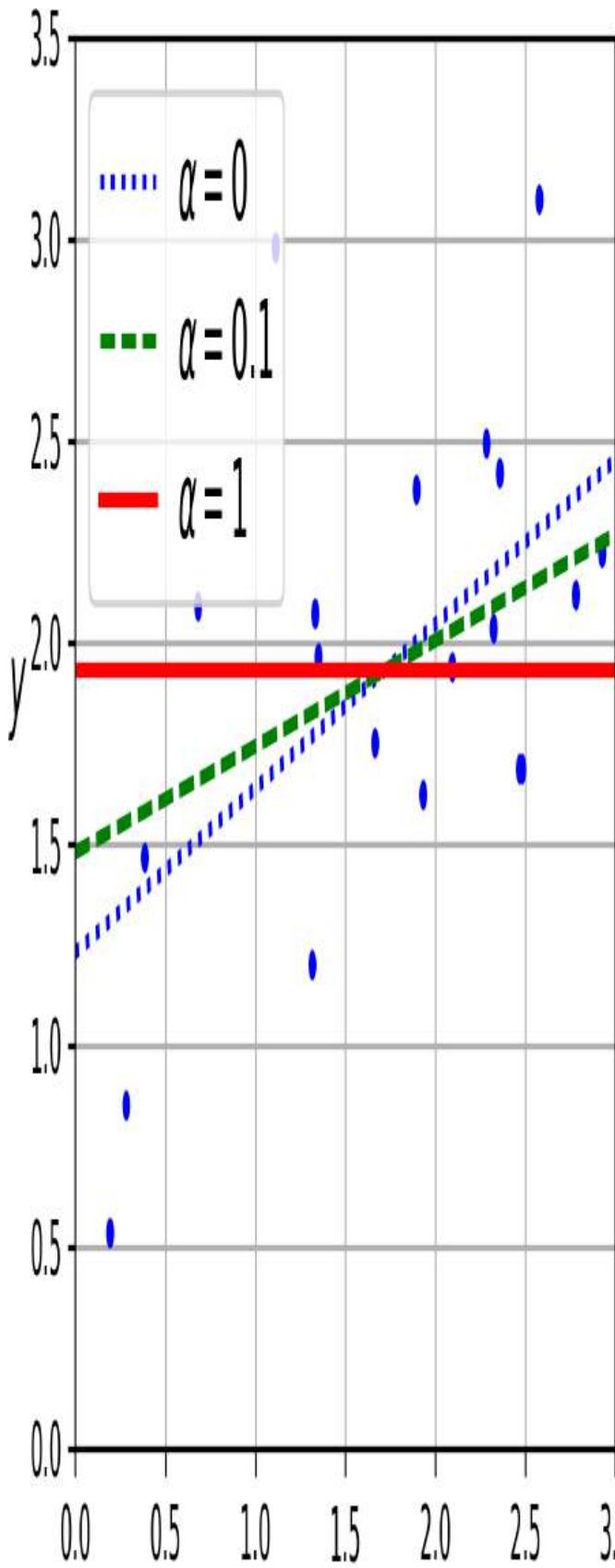


Figure 4-19. Linear (left) and polynomial (right) models, both using various levels of lasso regularization

An important characteristic of lasso regression is that it tends to eliminate the weights of the least important features (i.e., set them to zero). For example, the dashed line in the righthand plot in [Figure 4-19](#) (with  $\alpha = 0.01$ ) looks roughly cubic: all the weights for the high-degree polynomial features are equal to zero. In other words, lasso regression automatically performs feature selection and outputs a *sparse model* with few nonzero feature weights. Of course, there's a trade-off: if you increase  $\alpha$  too much, the model will be very sparse, but its performance will plummet.

You can get a sense of why the  $\ell_1$  norm induces sparsity by looking at [Figure 4-20](#): the axes represent two model parameters, and the background contours represent different loss functions. In the top-left plot, the contours represent the  $\ell_1$  loss ( $|\theta_1| + |\theta_2|$ ), which drops linearly as you get closer to any axis. For example, if you initialize the model parameters to  $\theta_1 = 2$  and  $\theta_2 = 0.5$ , running gradient descent will decrement both parameters equally (as represented by the dashed yellow line); therefore  $\theta_2$  will reach 0 first (since it was closer to 0 to begin with). After that, gradient descent will roll down the gutter until it reaches  $\theta_1 = 0$  (with a bit of bouncing around, since the gradients of  $\ell_1$  never get close to 0: they are either  $-1$  or  $1$  for each parameter). In the top-right plot, the contours represent lasso regression's cost function (i.e., an MSE cost function plus an  $\ell_1$  loss). The small white circles show the path that gradient descent takes to optimize some model parameters that were initialized around  $\theta_1 = 0.25$  and  $\theta_2 = -1$ : notice once again how the path quickly reaches  $\theta_2 = 0$ , then rolls down the gutter and ends up bouncing around the global optimum (represented by the red square). If we increased  $\alpha$ , the global optimum would move left along the dashed yellow line, while if we decreased  $\alpha$ , the global optimum would move right (in this example, the optimal parameters for the unregularized MSE are  $\theta_1 = 2$  and  $\theta_2 = 0.5$ ).

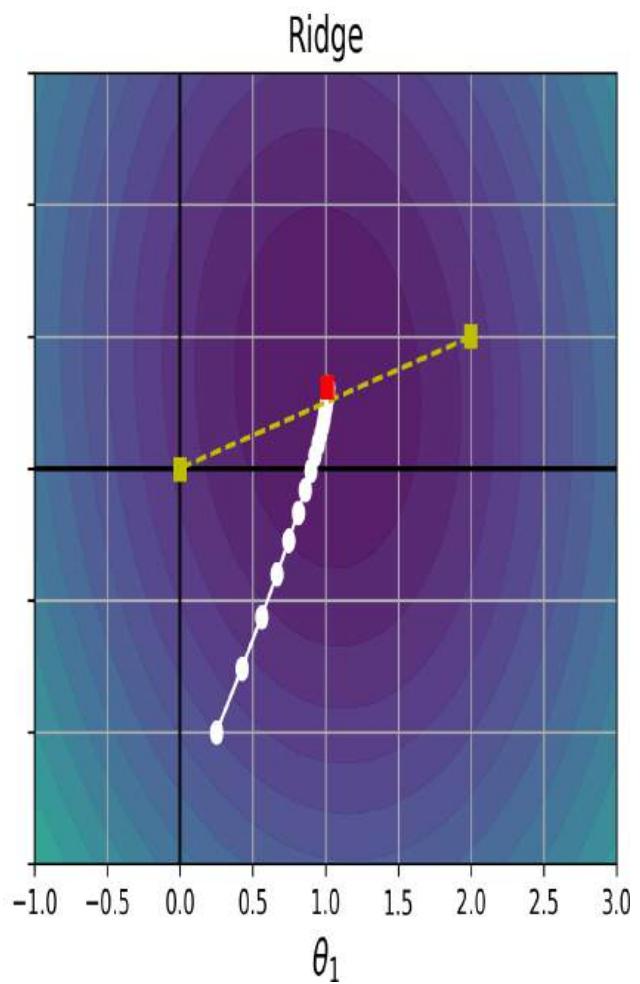
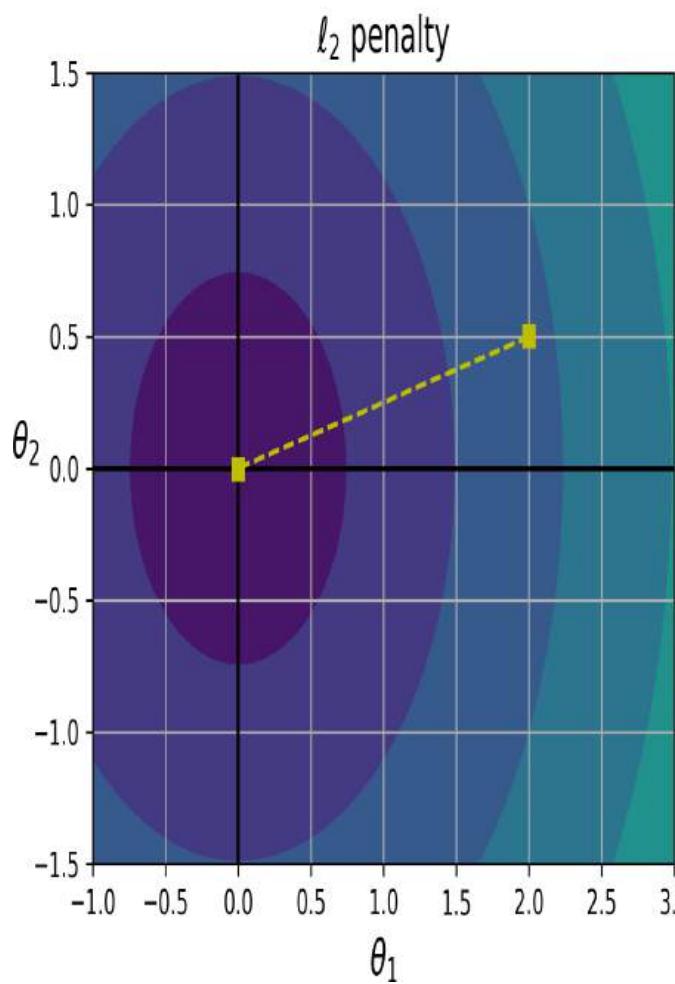
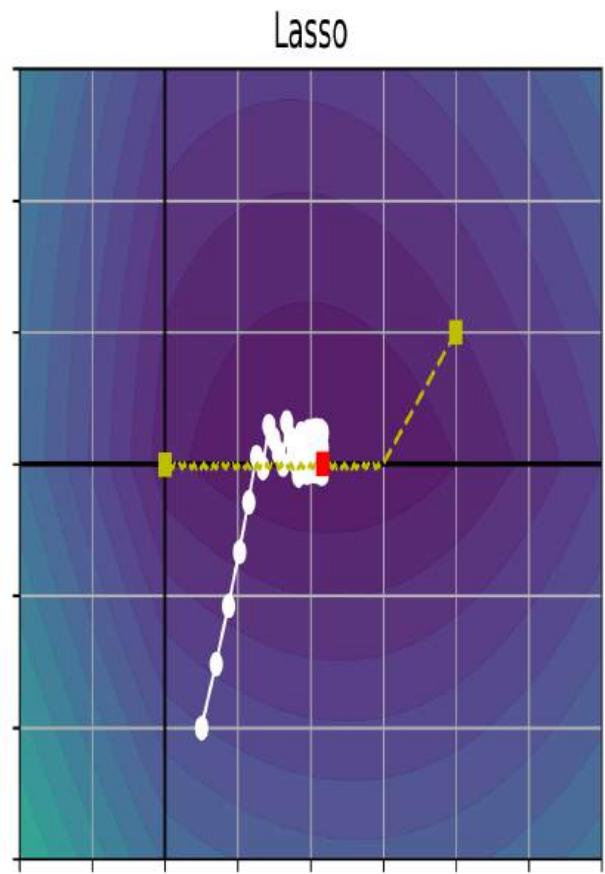
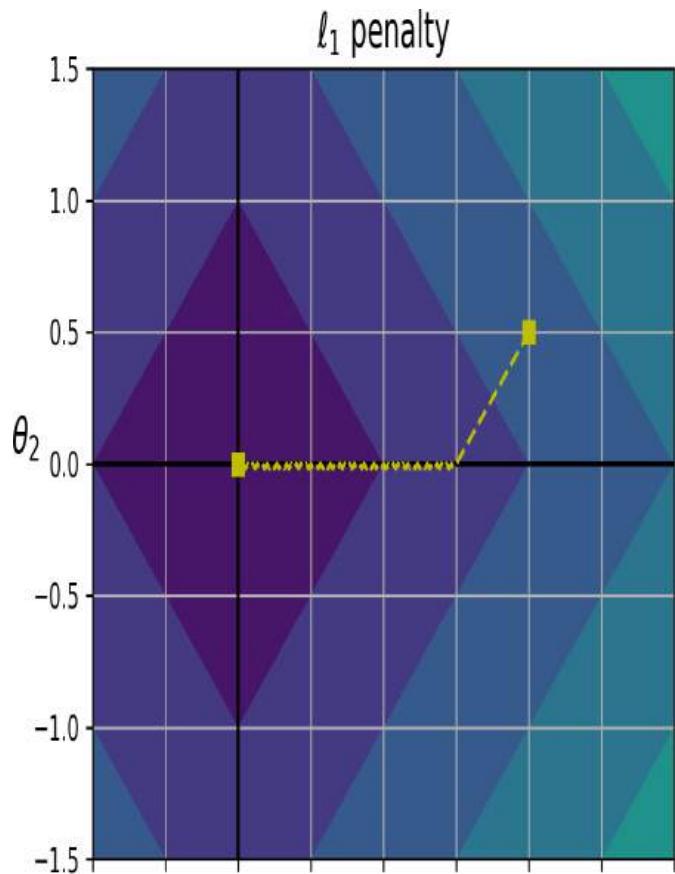


Figure 4-20. Lasso versus ridge regularization

The two bottom plots show the same thing but with an  $\ell_2$  penalty instead. In the bottom-left plot, you can see that the  $\ell_2$  loss decreases as we get closer to the origin, so gradient descent just takes a straight path toward that point. In the bottom-right plot, the contours represent ridge regression's cost function (i.e., an MSE cost function plus an  $\ell_2$  loss). As you can see, the gradients get smaller as the parameters approach the global optimum, so gradient descent naturally slows down. This limits the bouncing around, which helps ridge converge faster than lasso regression. Also note that the optimal parameters (represented by the red square) get closer and closer to the origin when you increase  $\alpha$ , but they never get eliminated entirely.

### TIP

To keep gradient descent from bouncing around the optimum at the end when using lasso regression, you need to gradually reduce the learning rate during training. It will still bounce around the optimum, but the steps will get smaller and smaller, so it will converge.

The lasso cost function is not differentiable at  $\theta_i = 0$  (for  $i = 1, 2, \dots, n$ ), but gradient descent still works if you use a *subgradient vector*  $\mathbf{g}$ <sup>11</sup> instead when any  $\theta_i = 0$ . **Equation 4-11** shows a subgradient vector equation you can use for gradient descent with the lasso cost function.

*Equation 4-11. Lasso regression subgradient vector*

$$g(\boldsymbol{\theta}, J) = \nabla_{\boldsymbol{\theta}} \text{MSE}(\boldsymbol{\theta}) + 2\alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{where } \text{sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

Here is a small Scikit-Learn example using the `Lasso` class:

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([1.87550211])
```

Note that you could instead use `SGDRegressor(penalty="l1", alpha=0.1)`.

## Elastic Net Regression

*Elastic net regression* is a middle ground between ridge regression and lasso regression. The regularization term is a weighted sum of both ridge and lasso's regularization terms, and you can control the mix ratio  $r$ . When  $r = 0$ , elastic net is equivalent to ridge regression, and when  $r = 1$ , it is equivalent to lasso regression (Equation 4-12).

*Equation 4-12. Elastic net cost function*

$$J(\boldsymbol{\theta}) = \text{MSE}(\boldsymbol{\theta}) + r(2\alpha \sum_{i=1}^n |\theta_i|) + (1 - r) \left( \frac{\alpha}{m} \sum_{i=1}^n \theta_i^2 \right)$$

So when should you use elastic net regression, or ridge, lasso, or plain linear regression (i.e., without any regularization)? It is almost always preferable to have at least a little bit of regularization, so generally you should avoid plain linear regression. Ridge is a good default, but if you suspect that only a few features are useful, you should prefer lasso or elastic net because they tend to reduce the useless features' weights down to zero, as discussed earlier. In general, elastic net is preferred over lasso because lasso may behave erratically when the number of features is greater than the number of training instances or when several features are strongly correlated.

Here is a short example that uses Scikit-Learn's `ElasticNet` (`l1_ratio` corresponds to the mix ratio  $r$ ):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([1.8645014])
```

## Early Stopping

A very different way to regularize iterative learning algorithms such as gradient descent is to stop training as soon as the validation error reaches a minimum. This is very popular technique called *early stopping*. Figure 4-21 shows a complex model (in this case, a high-degree polynomial regression model) being trained with batch gradient descent on the quadratic dataset we used earlier. As the epochs go by, the algorithm learns, and its prediction error (RMSE) on the training set goes down, along with its prediction error on the validation set. After a while, though, the validation error stops decreasing and starts to go back up. This indicates that the model has started to overfit the training data. With early stopping you just stop training as soon as the validation

error reaches the minimum. It is such a simple and efficient regularization technique that Geoffrey Hinton called it a “beautiful free lunch”.

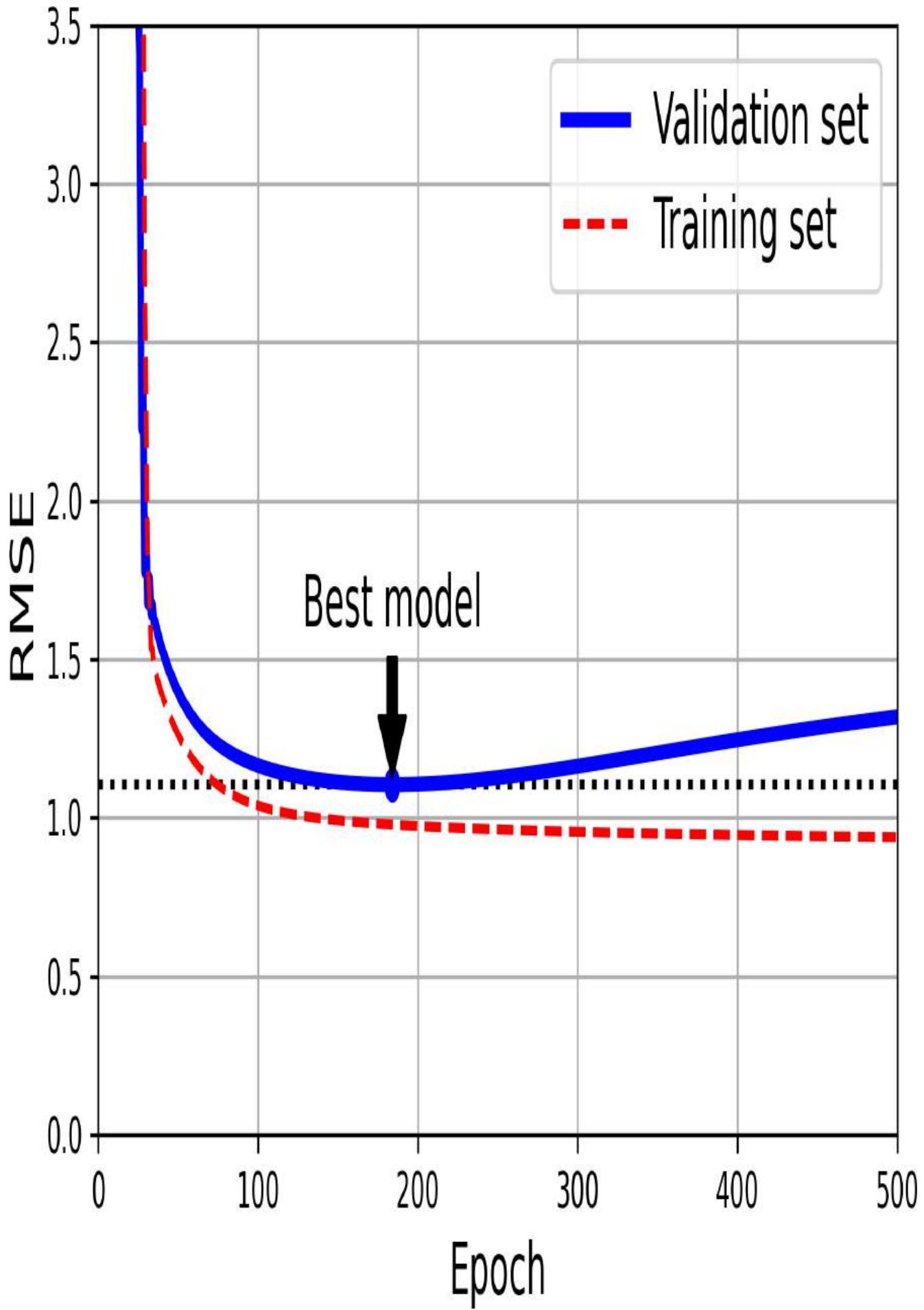


Figure 4-21. Early stopping regularization

## TIP

With stochastic and mini-batch gradient descent, the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not. One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum.

Here is a basic implementation of early stopping:

```
from copy import deepcopy
from sklearn.metrics import root_mean_squared_error
from sklearn.preprocessing import StandardScaler

X_train, y_train, X_valid, y_valid = [...] # split the quadratic dataset

preprocessing = make_pipeline(PolynomialFeatures(degree=90, include_bias=False),
                             StandardScaler())
X_train_prep = preprocessing.fit_transform(X_train)
X_valid_prep = preprocessing.transform(X_valid)
sgd_reg = SGDRegressor(penalty=None, eta0=0.002, random_state=42)
n_epochs = 500
best_valid_rmse = float('inf')

for epoch in range(n_epochs):
    sgd_reg.partial_fit(X_train_prep, y_train)
    y_valid_predict = sgd_reg.predict(X_valid_prep)
    val_error = root_mean_squared_error(y_valid, y_valid_predict)
    if val_error < best_valid_rmse:
        best_valid_rmse = val_error
        best_model = deepcopy(sgd_reg)
```

This code first adds the polynomial features and scales all the input features, both for the training set and for the validation set (the code assumes that you have split the original training set into a smaller training set and a validation set). Then it creates an `SGDRegressor` model with no regularization and a small learning rate. In the training loop, it calls `partial_fit()` instead of `fit()`, to perform incremental learning. At each epoch, it measures the RMSE on the validation set. If it is lower than the lowest RMSE seen so far, it saves a copy of the model in the `best_model` variable. This implementation does not actually stop training, but it lets you revert to the best model after training. Note that the model is copied using `copy.deepcopy()`, because it copies

both the model’s hyperparameters *and* the learned parameters. In contrast, `sklearn.base.clone()` only copies the model’s hyperparameters.

## Logistic Regression

As discussed in [Chapter 1](#), some regression algorithms can be used for classification (and vice versa). *Logistic regression* (also called *logit regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than a given threshold (typically 50%), then the model predicts that the instance belongs to that class (called the *positive class*, labeled “1”), and otherwise it predicts that it does not (i.e., it belongs to the *negative class*, labeled “0”). This makes it a binary classifier.

### Estimating Probabilities

So how does logistic regression work? Just like a linear regression model, a logistic regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the linear regression model does, it outputs the *logistic* of this result (see [Equation 4-13](#)).

*Equation 4-13. Logistic regression model estimated probability (vectorized form)*

$$\hat{p} = h_{\Theta}(\mathbf{x}) = \sigma(\boldsymbol{\Theta}^T \mathbf{x})$$

The logistic—noted  $\sigma(\cdot)$ —is a *sigmoid function* (i.e., S-shaped) that outputs a number between 0 and 1. It is defined as shown in [Equation 4-14](#) and [Figure 4-22](#).

*Equation 4-14. Logistic function*

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

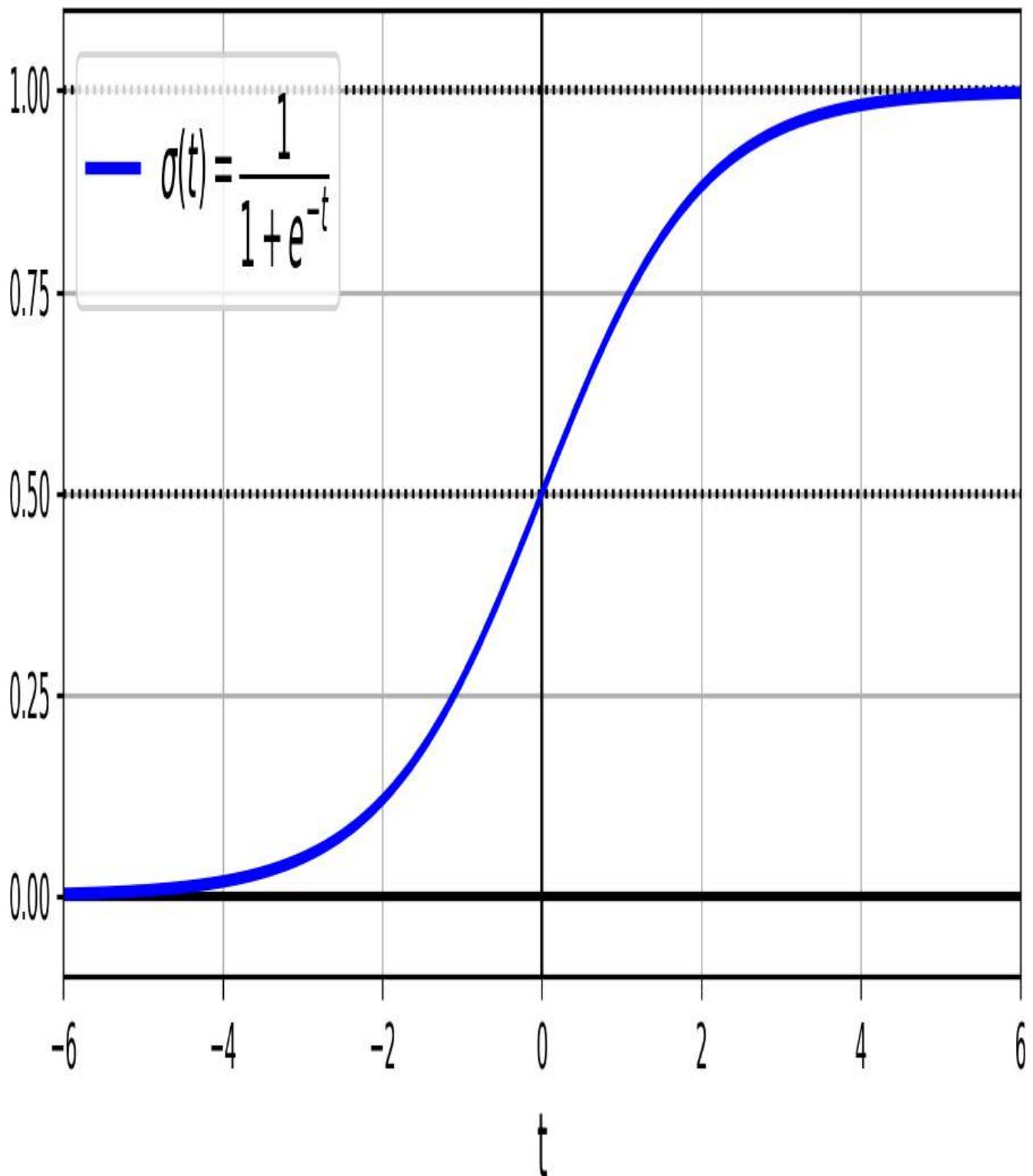


Figure 4-22. Logistic function

Once the logistic regression model has estimated the probability  $\hat{p} = h_{\theta}(x)$  that an instance  $x$  belongs to the positive class, it can make its prediction  $\hat{y}$  easily (see [Equation 4-15](#)).

[Equation 4-15. Logistic regression model prediction using a 50% threshold probability](#)

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5 \\ 1 & \text{if } \hat{p} \geq 0.5 \end{cases}$$

Notice that  $\sigma(t) < 0.5$  when  $t < 0$ , and  $\sigma(t) \geq 0.5$  when  $t \geq 0$ , so a logistic regression model using the default threshold of 50% probability predicts 1 if  $\boldsymbol{\theta}^\top \mathbf{x}$  is positive and 0 if it is negative.

### NOTE

The score  $t$  is often called the *logit*. The name comes from the fact that the logit function, defined as  $\text{logit}(p) = \log(p / (1 - p))$ , is the inverse of the logistic function. Indeed, if you compute the logit of the estimated probability  $p$ , you will find that the result is  $t$ . The logit is also called the *log-odds*, since it is the log of the ratio between the estimated probability for the positive class and the estimated probability for the negative class.

## Training and Cost Function

Now you know how a logistic regression model estimates probabilities and makes predictions. But how is it trained? The objective of training is to set the parameter vector  $\boldsymbol{\theta}$  so that the model estimates high probabilities for positive instances ( $y = 1$ ) and low probabilities for negative instances ( $y = 0$ ). This idea is captured by the cost function shown in [Equation 4-16](#) for a single training instance  $\mathbf{x}$ .

*Equation 4-16. Cost function of a single training instance*

$$c(\boldsymbol{\theta}) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1 \\ -\log(1 - \hat{p}) & \text{if } y = 0 \end{cases}$$

This cost function makes sense because  $-\log(t)$  grows very large when  $t$  approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive instance, and it will also be large if the model estimates a probability close to 1 for a negative instance. On the other hand,  $-\log(t)$  is close to 0 when  $t$  is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want.

The cost function over the whole training set is the average cost over all training instances. It can be written in a single expression called the *log loss*, shown in [Equation 4-17](#).

*Equation 4-17. Logistic regression cost function (log loss)*

$$J(\boldsymbol{\Theta}) = -\frac{1}{m} \sum_{i=1}^m [y^{(i)} \log(\hat{p}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{p}^{(i)})]$$

## WARNING

The log loss was not just pulled out of a hat. It can be shown mathematically (using Bayesian inference) that minimizing this loss will result in the model with the *maximum likelihood* of being optimal, assuming that the instances follow a Gaussian distribution around the mean of their class. When you use the log loss, this is the implicit assumption you are making. The more wrong this assumption is, the more biased the model will be. Similarly, when we used the MSE to train linear regression models, we were implicitly assuming that the data was purely linear, plus some Gaussian noise. So, if the data is not linear (e.g., if it's quadratic) or if the noise is not Gaussian (e.g., if outliers are not exponentially rare), then the model will be biased.

The bad news is that there is no known closed-form equation to compute the value of  $\boldsymbol{\Theta}$  that minimizes this cost function (there is no equivalent of the Normal equation). But the good news is that this cost function is convex, so gradient descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough). The partial derivatives of the cost function with regard to the  $j^{\text{th}}$  model parameter  $\theta_j$  are given by [Equation 4-18](#).

*Equation 4-18. Logistic cost function partial derivatives*

$$\frac{\partial}{\partial \theta_j} J(\boldsymbol{\Theta}) = \frac{1}{m} \sum_{i=1}^m \left( \sigma(\boldsymbol{\Theta}^\top \mathbf{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

This equation looks very much like [Equation 4-5](#): for each instance it computes the prediction error and multiplies it by the  $j^{\text{th}}$  feature value, and then it computes the average over all training instances. Once you have the gradient vector containing all the partial derivatives, you can use it in the batch gradient descent algorithm. That's it: you now know how to train a logistic regression model. For stochastic GD you would take one instance at a time, and for mini-batch GD you would use a mini-batch at a time.

## Decision Boundaries

We can use the iris dataset to illustrate logistic regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: *Iris setosa*, *Iris versicolor*, and *Iris virginica* (see [Figure 4-23](#)).



Figure 4-23. Flowers of three iris plant species<sup>12</sup>

Let's try to build a classifier to detect the *Iris virginica* type based only on the petal width feature. The first step is to load the data and take a quick peek:

```

>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> list(iris)
['data', 'target', 'frame', 'target_names', 'DESCR', 'feature_names',
 'filename', 'data_module']
>>> iris.data.head(3)
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0              5.1             3.5            1.4             0.2
1              4.9             3.0            1.4             0.2
2              4.7             3.2            1.3             0.2
>>> iris.target.head(3) # note that the instances are not shuffled
0    0
1    0
2    0
Name: target, dtype: int64
>>> iris.target_names
array(['setosa', 'versicolor', 'virginica'], dtype='<U10')

```

Next we'll split the data and train a logistic regression model on the training set:

```

from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

X = iris.data[["petal width (cm)"]].values
y = iris.target_names[iris.target] == 'virginica'
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

log_reg = LogisticRegression(random_state=42)
log_reg.fit(X_train, y_train)

```

Let's look at the model's estimated probabilities for flowers with petal widths varying from 0 cm to 3 cm (Figure 4-24):<sup>13</sup>

```

X_new = np.linspace(0, 3, 1000).reshape(-1, 1) # reshape to get a column vector
y_proba = log_reg.predict_proba(X_new)
decision_boundary = X_new[y_proba[:, 1] >= 0.5][0, 0]

plt.plot(X_new, y_proba[:, 0], "b--", linewidth=2,
          label="Not Iris virginica proba")
plt.plot(X_new, y_proba[:, 1], "g-", linewidth=2, label="Iris virginica proba")
plt.plot([decision_boundary, decision_boundary], [0, 1], "k:", linewidth=2,
          label="Decision boundary")
[...] # beautify the figure: add grid, labels, axis, legend, arrows, and samples
plt.show()

```

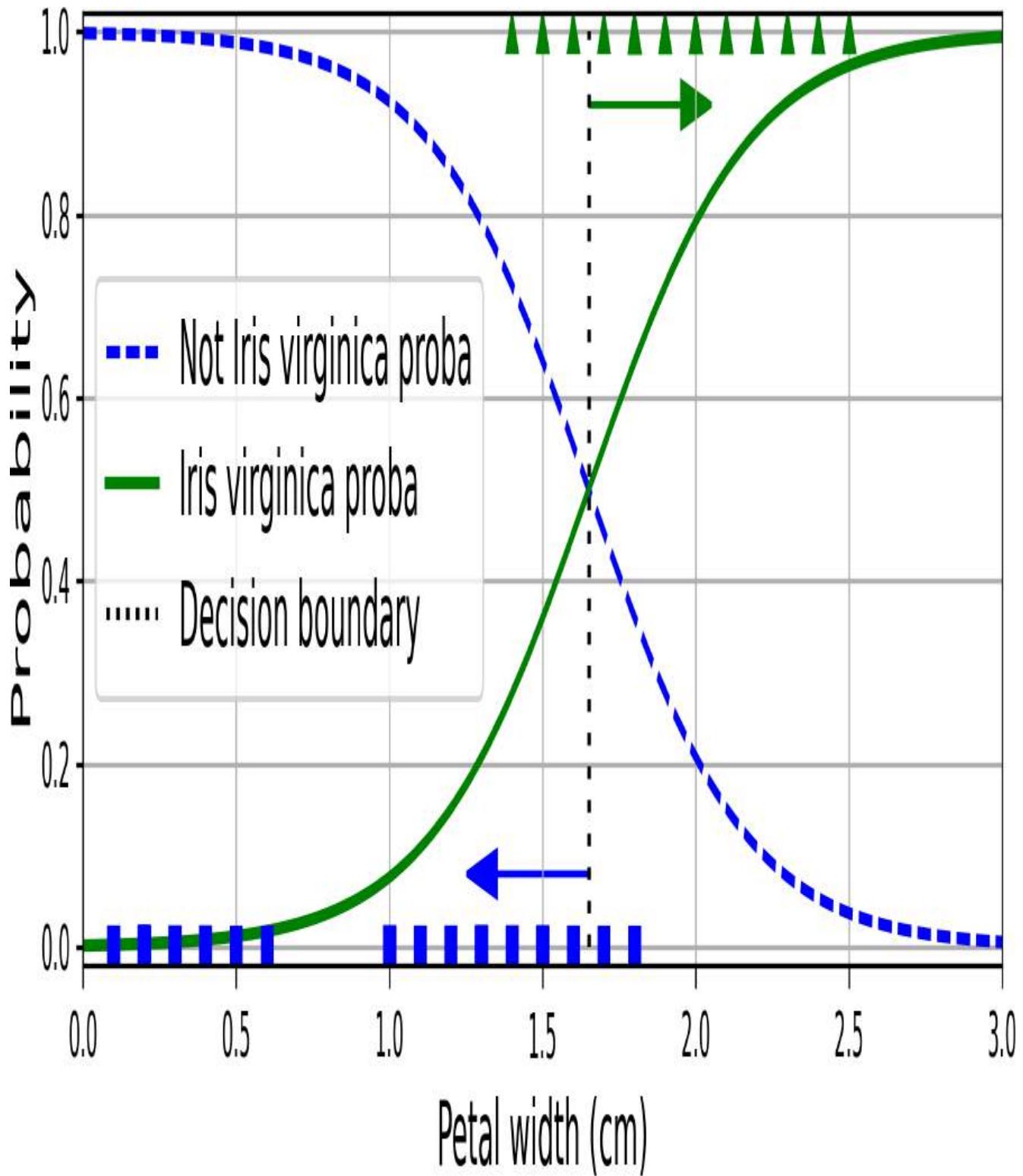


Figure 4-24. Estimated probabilities and decision boundary

The petal width of *Iris virginica* flowers (represented as triangles) ranges from 1.4 cm to 2.5 cm, while the other iris flowers (represented by squares) generally have a smaller petal width, ranging from 0.1 cm to 1.8 cm. Notice that there is a bit of overlap. Above about 2 cm the classifier is highly confident that the flower is an *Iris virginica* (it outputs a high probability for that class), while below 1 cm it is highly confident that

it is not an *Iris virginica* (high probability for the “Not Iris virginica” class). In between these extremes, the classifier is unsure. However, if you ask it to predict the class (using the `predict()` method rather than the `predict_proba()` method), it will return whichever class is the most likely. Therefore, there is a *decision boundary* at around 1.6 cm where both probabilities are equal to 50%: if the petal width is greater than 1.6 cm the classifier will predict that the flower is an *Iris virginica*, and otherwise it will predict that it is not (even if it is not very confident):

```
>>> decision_boundary  
1.6516516516517  
>>> log_reg.predict([[1.7], [1.5]])  
array([ True, False])
```

Figure 4-25 shows the same dataset, but this time displaying two features: petal width and length. Once trained, the logistic regression classifier can, based on these two features, estimate the probability that a new flower is an *Iris virginica*. The dashed line represents the points where the model estimates a 50% probability: this is the model’s decision boundary. Note that it is a linear boundary.<sup>14</sup> Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right). All the flowers beyond the top-right line have over 90% chance of being *Iris virginica*, according to the model.

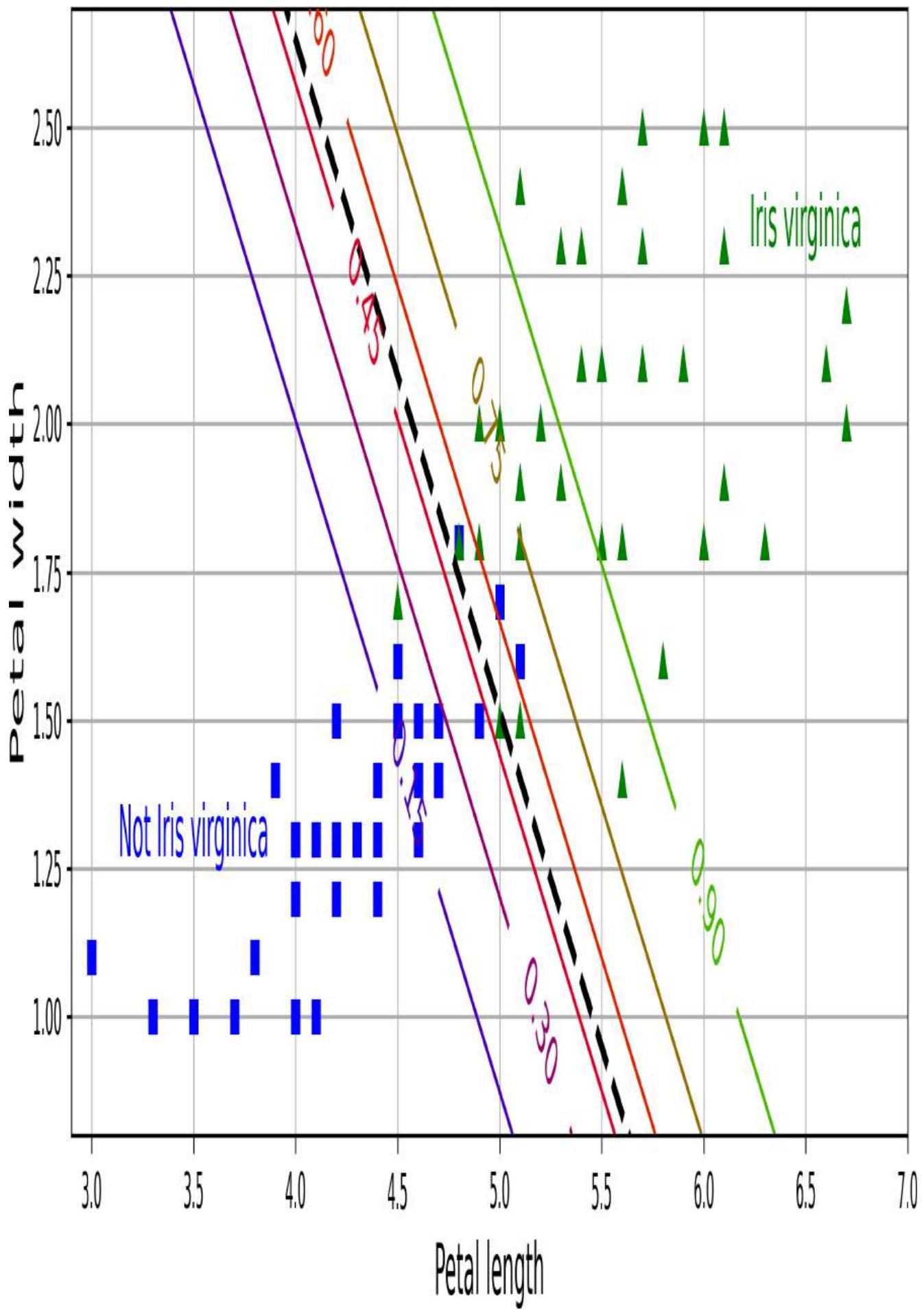


Figure 4-25. Linear decision boundary

## NOTE

The hyperparameter controlling the regularization strength of a Scikit-Learn LogisticRegression model is not `alpha` (as in other linear models), but its inverse: `C`. The higher the value of `C`, the *less* the model is regularized.

Just like the other linear models, logistic regression models can be regularized using  $\ell_1$  or  $\ell_2$  penalties. Scikit-Learn actually adds an  $\ell_2$  penalty by default.

## Softmax Regression

The logistic regression model can be generalized to support multiple classes directly, without having to train and combine multiple binary classifiers (as discussed in [Chapter 3](#)). This is called *softmax regression*, or *multinomial logistic regression*.

The idea is simple: when given an instance  $\mathbf{x}$ , the softmax regression model first computes a score  $s_k(\mathbf{x})$  for each class  $k$ , then estimates the probability of each class by applying the *softmax function* (also called the *normalized exponential*) to the scores. The equation to compute  $s_k(\mathbf{x})$  should look familiar, as it is just like the equation for linear regression prediction (see [Equation 4-19](#)).

Equation 4-19. Softmax score for class  $k$

$$s_k(\mathbf{x}) = (\boldsymbol{\theta}^{(k)})^\top \mathbf{x}$$

Note that each class has its own dedicated parameter vector  $\boldsymbol{\theta}^{(k)}$ . All these vectors are typically stored as rows in a *parameter matrix*  $\boldsymbol{\Theta}$ .

Once you have computed the score of every class for the instance  $\mathbf{x}$ , you can estimate the probability  $\hat{p}_k$  that the instance belongs to class  $k$  by running the scores through the softmax function ([Equation 4-20](#)). The function computes the exponential of every score, then normalizes them (dividing by the sum of all the exponentials). The scores are generally called logits or log-odds (although they are actually unnormalized log-odds).

Equation 4-20. Softmax function

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{\exp(s_k(\mathbf{x}))}{\sum_{j=1}^K \exp(s_j(\mathbf{x}))}$$

In this equation:

- $K$  is the number of classes.
- $\mathbf{s}(\mathbf{x})$  is a vector containing the scores of each class for the instance  $\mathbf{x}$ .
- $\sigma(\mathbf{s}(\mathbf{x}))_k$  is the estimated probability that the instance  $\mathbf{x}$  belongs to class  $k$ , given the scores of each class for that instance.

Just like the logistic regression classifier, by default the softmax regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score), as shown in [Equation 4-21](#).

*Equation 4-21. Softmax regression classifier prediction*

$$\hat{y} = \underset{k}{\operatorname{argmax}} \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}} s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \left( (\boldsymbol{\theta}^{(k)})^\top \mathbf{x} \right)$$

The *argmax* operator returns the value of a variable that maximizes a function. In this equation, it returns the value of  $k$  that maximizes the estimated probability  $\sigma(\mathbf{s}(\mathbf{x}))_k$ .

### TIP

The softmax regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput), so it should be used only with mutually exclusive classes, such as different species of plants. You cannot use it to recognize multiple people in one picture.

Now that you know how the model estimates probabilities and makes predictions, let's take a look at training. The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes).

Minimizing the cost function shown in [Equation 4-22](#), called the *cross entropy*, should lead to this objective because it penalizes the model when it estimates a low probability for a target class. Cross entropy is frequently used to measure how well a set of estimated class probabilities matches the target classes.

*Equation 4-22. Cross entropy cost function*

$$J(\Theta) = -\frac{1}{m} \sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(\hat{p}_k^{(i)})$$

In this equation,  $y_k^{(i)}$  is the target probability that the  $i^{\text{th}}$  instance belongs to class  $k$ . In general, it is either equal to 1 or 0, depending on whether the instance belongs to the class or not.

Notice that when there are just two classes ( $K = 2$ ), this cost function is equivalent to the logistic regression cost function (log loss; see [Equation 4-17](#)).

## CROSS ENTROPY

Cross entropy originated from Claude Shannon's *information theory*. Suppose you want to efficiently transmit information about the weather every day. If there are eight options (sunny, rainy, etc.), you could encode each option using 3 bits, because  $2^3 = 8$ . However, if you think it will be sunny almost every day, it would be much more efficient to code "sunny" on just one bit (0) and the other seven options on four bits (starting with a 1). Cross entropy measures the average number of bits you actually send per option. If your assumption about the weather is perfect, cross entropy will be equal to the entropy of the weather itself (i.e., its intrinsic unpredictability). But if your assumption is wrong (e.g., if it rains often), cross entropy will be greater by an amount called the *Kullback–Leibler (KL) divergence*.

The cross entropy between two probability distributions  $p$  and  $q$  is defined as  $H(p,q) = -\sum_x p(x) \log q(x)$  (at least when the distributions are discrete). For more details, check out [my video on the subject](#).

The gradient vector of this cost function with regard to  $\Theta^{(k)}$  is given by [Equation 4-23](#).

*Equation 4-23. Cross entropy gradient vector for class  $k$*

$$\nabla_{\Theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^m (\hat{p}_k^{(i)} - y_k^{(i)}) \mathbf{x}^{(i)}$$

Now you can compute the gradient vector for every class, then use gradient descent (or any other optimization algorithm) to find the parameter matrix  $\Theta$  that minimizes the cost function.

Let's use softmax regression to classify the iris plants into all three classes. Scikit-Learn's `LogisticRegression` classifier uses softmax regression automatically when

you train it on more than two classes (assuming you use `solver="lbfgs"`, which is the default). It also applies  $\ell_2$  regularization by default, which you can control using the hyperparameter `C`: decrease `C` to increase regularization, as mentioned earlier.

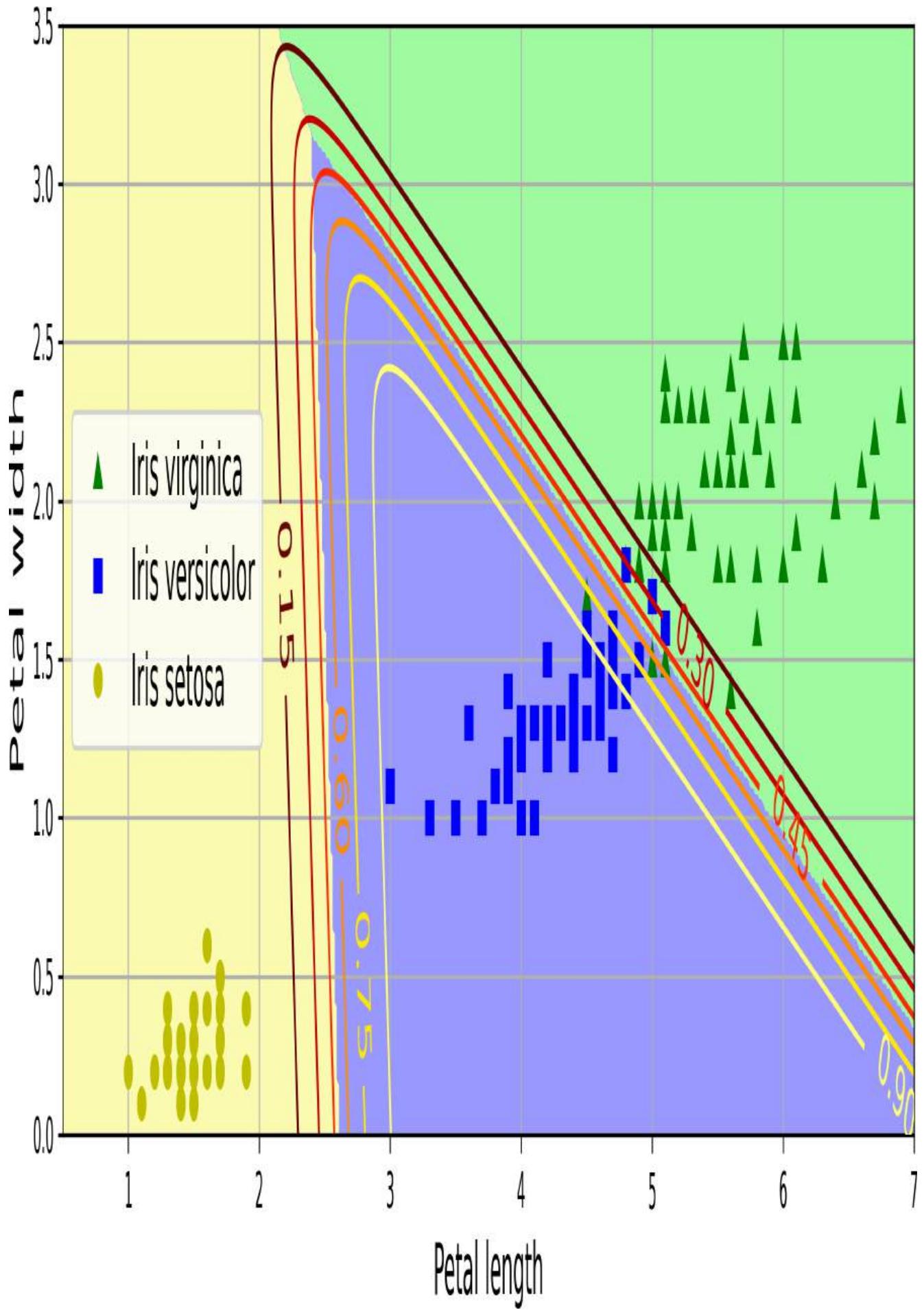
```
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = iris["target"]
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

softmax_reg = LogisticRegression(C=30, random_state=42)
softmax_reg.fit(X_train, y_train)
```

So the next time you find an iris with petals that are 5 cm long and 2 cm wide, you can ask your model to tell you what type of iris it is, and it will answer *Iris virginica* (class 2) with 96% probability (or *Iris versicolor* with 4% probability):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]]).round(2)
array([[0.  , 0.04, 0.96]])
```

**Figure 4-26** shows the resulting decision boundaries, represented by the background colors. Notice that the decision boundaries between any two classes are linear. The figure also shows the probabilities for the *Iris versicolor* class, represented by the curved lines (e.g., the line labeled with 0.30 represents the 30% probability boundary). Notice that the model can predict a class that has an estimated probability below 50%. For example, at the point where all decision boundaries meet, all classes have an equal estimated probability of 33%.



*Figure 4-26. Softmax regression decision boundaries*

In this chapter, you learned various ways to train linear models, both for regression and for classification. You used a closed-form equation to solve linear regression, as well as gradient descent, and you learned how various penalties can be added to the cost function during training to regularize the model. Along the way, you also learned how to plot learning curves and analyze them, and how to implement early stopping. Finally, you learned how logistic regression and softmax regression work. We've opened up the first machine learning black boxes! In the next chapters we will open many more, starting with support vector machines.

## Exercises

1. Which linear regression training algorithm can you use if you have a training set with millions of features?
2. Suppose the features in your training set have very different scales. Which algorithms might suffer from this, and how? What can you do about it?
3. Can gradient descent get stuck in a local minimum when training a logistic regression model?
4. Do all gradient descent algorithms lead to the same model, provided you let them run long enough?
5. Suppose you use batch gradient descent and you plot the validation error at every epoch. If you notice that the validation error consistently goes up, what is likely going on? How can you fix this?
6. Is it a good idea to stop mini-batch gradient descent immediately when the validation error goes up?
7. Which gradient descent algorithm (among those we discussed) will reach the vicinity of the optimal solution the fastest? Which will actually converge? How can you make the others converge as well?
8. Suppose you are using polynomial regression. You plot the learning curves and you notice that there is a large gap between the training error and the validation error. What is happening? What are three ways to solve this?

9. Suppose you are using ridge regression and you notice that the training error and the validation error are almost equal and fairly high. Would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter  $\alpha$  or reduce it?
10. Why would you want to use:
- Ridge regression instead of plain linear regression (i.e., without any regularization)?
  - Lasso instead of ridge regression?
  - Elastic net instead of lasso regression?
11. Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime. Should you implement two logistic regression classifiers or one softmax regression classifier?
12. Implement batch gradient descent with early stopping for softmax regression without using Scikit-Learn, only NumPy. Use it on a classification task such as the iris dataset.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.

---

<sup>1</sup> A closed-form equation is only composed of a finite number of constants, variables, and standard operations: for example,  $a = \sin(b - c)$ . No infinite sums, no limits, no integrals, etc.

<sup>2</sup> Technically speaking, its derivative is *Lipschitz continuous*.

<sup>3</sup> Since feature 1 is smaller, it takes a larger change in  $\theta_1$  to affect the cost function, which is why the bowl is elongated along the  $\theta_1$  axis.

<sup>4</sup> Eta ( $\eta$ ) is the seventh letter of the Greek alphabet.

<sup>5</sup> While the Normal equation can only perform linear regression, the gradient descent algorithms can be used to train many other models, as you'll see.

<sup>6</sup> This notion of bias is not to be confused with the bias term of linear models.

<sup>7</sup> It is common to use the notation  $J(\theta)$  for cost functions that don't have a short name; I'll often use this notation throughout the rest of this book. The context will make it clear which cost function is being discussed.

<sup>8</sup> Norms are discussed in [Chapter 2](#).

- 9 A square matrix full of 0s except for 1s on the main diagonal (top left to bottom right).
- 10 Alternatively, you can use the `Ridge` class with the "`sag`" solver. Stochastic average GD is a variant of stochastic GD. For more details, see the presentation "[Minimizing Finite Sums with the Stochastic Average Gradient Algorithm](#)" by Mark Schmidt et al. from the University of British Columbia.
- 11 You can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point.
- 12 Photos reproduced from the corresponding Wikipedia pages. *Iris virginica* photo by Frank Mayfield ([Creative Commons BY-SA 2.0](#)), *Iris versicolor* photo by D. Gordon E. Robertson ([Creative Commons BY-SA 3.0](#)), *Iris setosa* photo public domain.
- 13 NumPy's `reshape()` function allows one dimension to be `-1`, which means "automatic": the value is inferred from the length of the array and the remaining dimensions.
- 14 It is the set of points  $\mathbf{x}$  such that  $\theta_0 + \theta_1x_1 + \theta_2x_2 = 0$ , which defines a straight line.

# Chapter 5. Decision Trees

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 5th chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

*Decision trees* are versatile machine learning algorithms that can perform both classification and regression tasks, and even multioutput tasks. They are powerful algorithms, capable of fitting complex datasets. For example, in [Chapter 2](#) you trained a `DecisionTreeRegressor` model on the California housing dataset, fitting it perfectly (actually, overfitting it).

Decision trees are also the fundamental components of random forests (see [Chapter 6](#)), which are among the most powerful machine learning algorithms available today.

In this chapter we will start by discussing how to train, visualize, and make predictions with decision trees. Then we will go through the CART training algorithm used by Scikit-Learn, and we will explore how to regularize trees and use them for regression tasks. Finally, we will discuss some of the limitations of decision trees.

## Training and Visualizing a Decision Tree

To understand decision trees, let’s build one and take a look at how it makes predictions. The following code trains a `DecisionTreeClassifier` on the iris dataset (see [Chapter 4](#)):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris(as_frame=True)
X_iris = iris.data[["petal length (cm)", "petal width (cm)"]].values
y_iris = iris.target
```

```
tree_clf = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf.fit(X_iris, y_iris)
```

You can visualize the trained decision tree by first using the `export_graphviz()` function to output a graph definition file called `iris_tree.dot`:

```
from sklearn.tree import export_graphviz

export_graphviz(
    tree_clf,
    out_file="iris_tree.dot",
    feature_names=["petal length (cm)", "petal width (cm)"],
    class_names=iris.target_names,
    rounded=True,
    filled=True
)
```

Then you can use `graphviz.Source.from_file()` to load and display the file in a Jupyter notebook:

```
from graphviz import Source

Source.from_file("iris_tree.dot")
```

**Graphviz** is an open source graph visualization software package. It also includes a `dot` command-line tool to convert `.dot` files to a variety of formats, such as PDF or PNG.

Your first decision tree looks like [Figure 5-1](#).

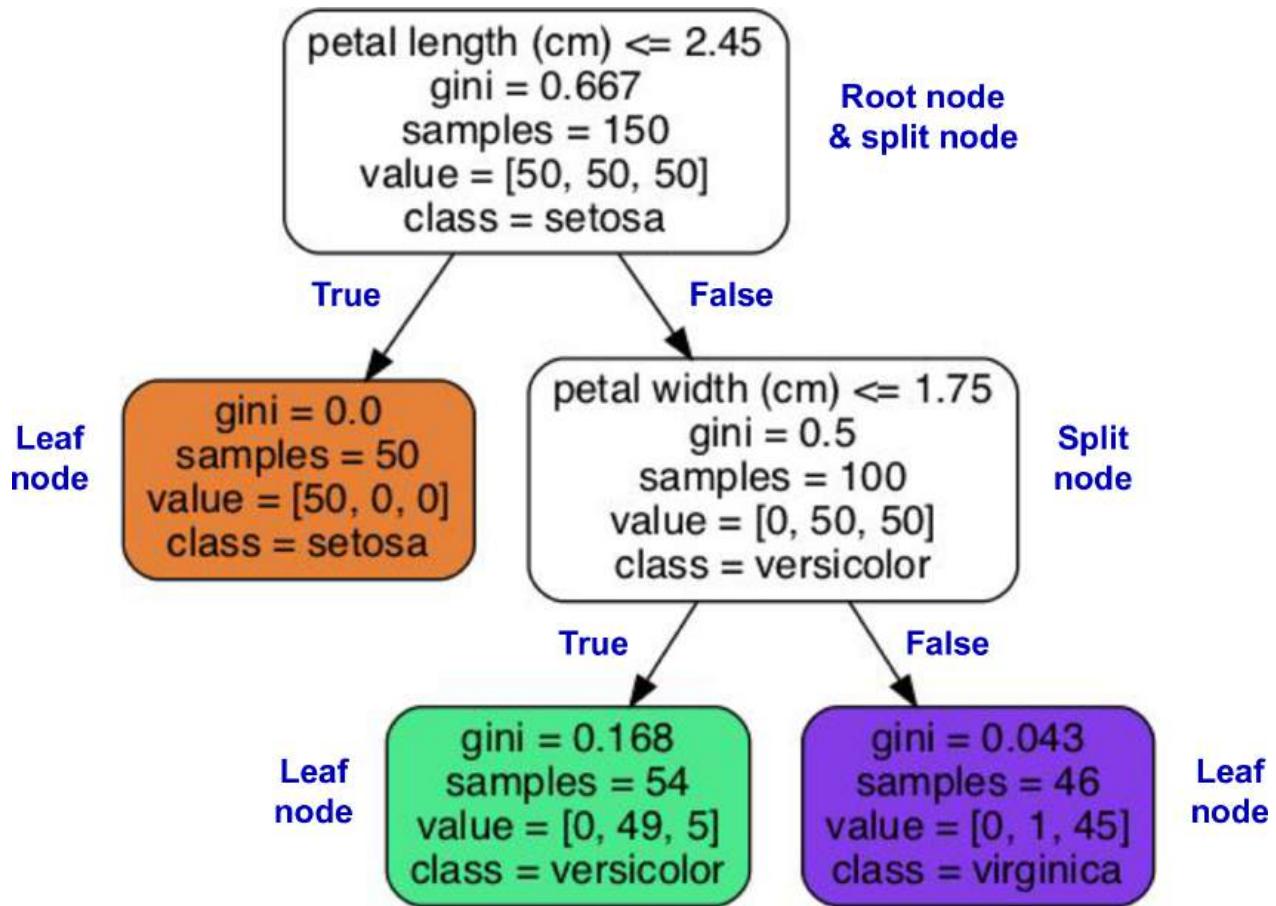


Figure 5-1. Iris decision tree

## Making Predictions

Let's see how the tree represented in Figure 5-1 makes predictions. Suppose you find an iris flower and you want to classify it based on its petals. You start at the *root node* (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm. If it is, then you move down to the root's left child node (depth 1, left). In this case, it is a *leaf node* (i.e., it does not have any child nodes), so it does not ask any questions: simply look at the predicted class for that node, and the decision tree predicts that your flower is an *Iris setosa* (`class=setosa`).

Now suppose you find another flower, and this time the petal length is greater than 2.45 cm. You again start at the root but now move down to its right child node (depth 1, right). This is not a leaf node, it's a *split node*, so it asks another question: is the petal width smaller than 1.75 cm? If it is, then your flower is most likely an *Iris versicolor* (depth 2, left). If not, it is likely an *Iris virginica* (depth 2, right). It's really that simple.

## NOTE

One of the many qualities of decision trees is that they require very little data preparation. In fact, they don't require feature scaling or centering at all.

A node's `samples` attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), and of those 100, 54 have a petal width smaller than 1.75 cm (depth 2, left). A node's `value` attribute tells you how many training instances of each class this node applies to: for example, the bottom-right node applies to 0 *Iris setosa*, 1 *Iris versicolor*, and 45 *Iris virginica*. Finally, a node's `gini` attribute measures its *Gini impurity*: a node is “pure” (`gini=0`) if all training instances it applies to belong to the same class. For example, since the depth-1 left node applies only to *Iris setosa* training instances, its Gini impurity is 0. Conversely, the other nodes all apply to instances of multiple classes, so they are “impure”. [Equation 5-1](#) shows how the training algorithm computes the Gini impurity  $G_i$  of the  $i^{\text{th}}$  node. The more classes and the more mixed they are, the larger the impurity. For example, the depth-2 left node has a Gini impurity equal to  $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$ .

*Equation 5-1. Gini impurity*

$$G_i = 1 - \sum_{k=1}^n p_{i,k}^2$$

In this equation:

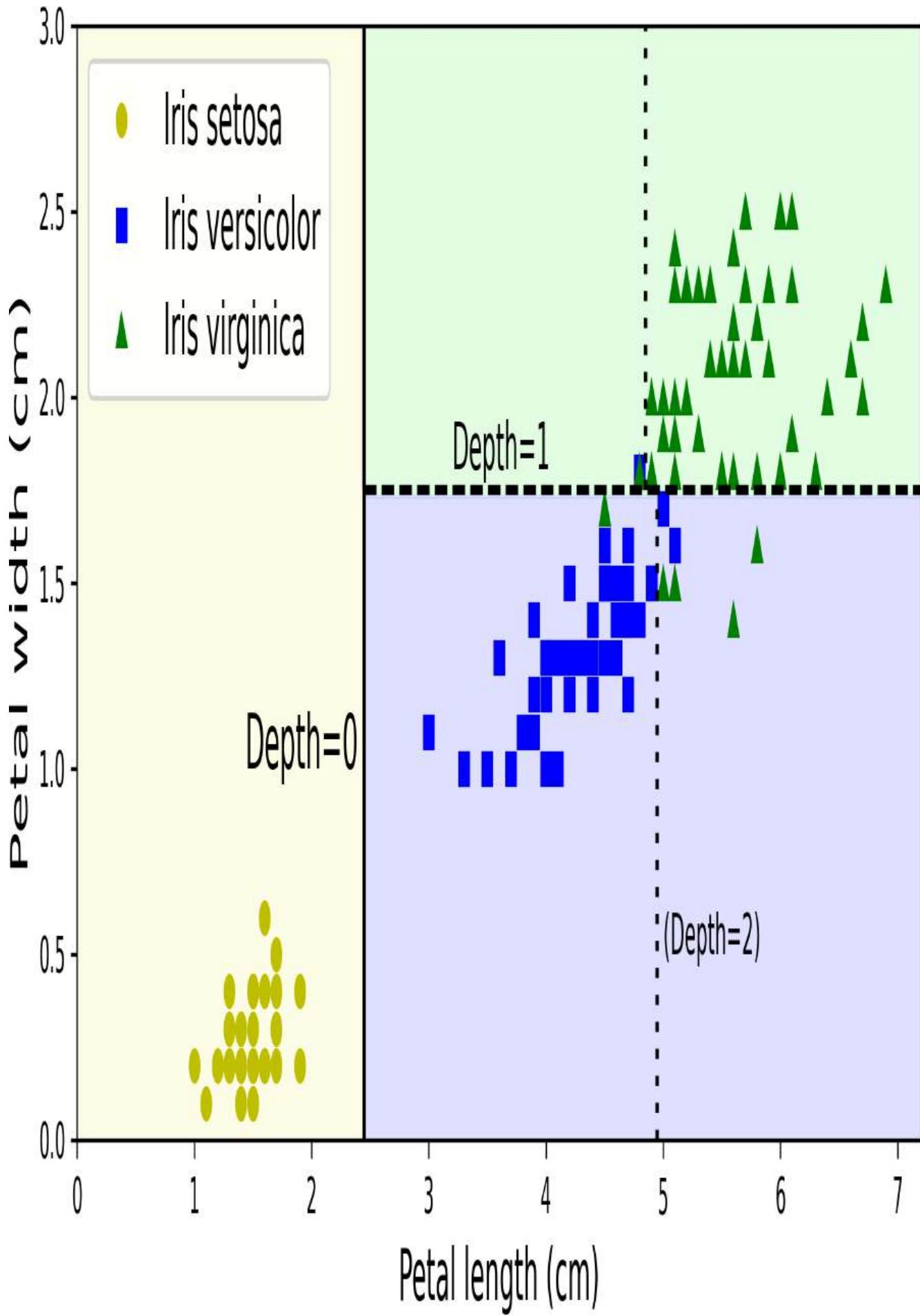
- $G_i$  is the Gini impurity of the  $i^{\text{th}}$  node.
- $p_{i,k}$  is the ratio of class  $k$  instances among the training instances in the  $i^{\text{th}}$  node.

## NOTE

Scikit-Learn uses the CART algorithm, which produces only *binary trees*, meaning trees where split nodes always have exactly two children (i.e., questions only have yes/no answers). However, other algorithms, such as ID3, can produce decision trees with nodes that have more than two children.

[Figure 5-2](#) shows this decision tree's decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0): petal length = 2.45 cm.

Since the lefthand area is pure (only *Iris setosa*), it cannot be split any further. However, the righthand area is impure, so the depth-1 right node splits it at petal width = 1.75 cm (represented by the dashed line). Since `max_depth` was set to 2, the decision tree stops right there. If you set `max_depth` to 3, then the two depth-2 nodes would each add another decision boundary (represented by the two vertical dotted lines).



*Figure 5-2. Decision tree decision boundaries*

**TIP**

The tree structure, including all the information shown in [Figure 5-1](#), is available via the classifier’s `tree_` attribute. Type `help(tree_clf.tree_)` for details, and see the [this chapter’s notebook](#) for an example.

## MODEL INTERPRETATION: WHITE BOX VERSUS BLACK BOX

Decision trees are intuitive, and their decisions are easy to interpret. Such models are often called *white box models*. In contrast, as you will see, random forests and neural networks are generally considered *black box models*. They make great predictions, and you can easily check the calculations that they performed to make these predictions; nevertheless, it is usually hard to explain in simple terms why the predictions were made. For example, if a neural network says that a particular person appears in a picture, it is hard to know what contributed to this prediction: Did the model recognize that person’s eyes? Their mouth? Their nose? Their shoes? Or even the couch that they were sitting on? Conversely, decision trees provide nice, simple classification rules that can even be applied manually if need be (e.g., for flower classification). The field of *interpretable ML* aims at creating ML systems that can explain their decisions in a way humans can understand. This is important in many domains, for example in healthcare, to let a doctor review the diagnosis; in finance, to let analysts understand the risks; in a judicial system, to let a human make the final call; or in human resources, to ensure decisions aren’t biased.

## Estimating Class Probabilities

A decision tree can also estimate the probability that an instance belongs to a particular class  $k$ . First it traverses the tree to find the leaf node for this instance, and then it returns the proportion of instances of class  $k$  among the training instances that would also reach this leaf node. For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The corresponding leaf node is the depth-2 left node, so the decision tree outputs the following probabilities: 0% for *Iris setosa* (0/54), 90.7% for *Iris versicolor* (49/54), and 9.3% for *Iris virginica* (5/54). And if you ask it

to predict the class, it outputs *Iris versicolor* (class 1) because it has the highest probability. Let's check this:

```
>>> tree_clf.predict_proba([[5, 1.5]]).round(3)
array([[0.    , 0.907, 0.093]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Perfect! Notice that the estimated probabilities would be identical anywhere else in the bottom-right rectangle of Figure 5-2—for example, if the petals were 6 cm long and 1.5 cm wide (even though it seems obvious that it would most likely be an *Iris virginica* in this case).

## The CART Training Algorithm

Scikit-Learn uses the *Classification and Regression Tree* (CART) algorithm to train decision trees (also called “growing” trees). The algorithm works by first splitting the training set into two subsets using a single feature  $k$  and a threshold  $t_k$  (e.g., “petal length  $\leq 2.45$  cm”). How does it choose  $k$  and  $t_k$ ? It searches for the pair  $(k, t_k)$  that produces the purest subsets, weighted by their size. Equation 5-2 gives the cost function that the algorithm tries to minimize.

Equation 5-2. CART cost function for classification

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

where      |  $G_{\text{left/right}}$  measures the impurity of the left/right subset  
                |  $m_{\text{left/right}}$  is the number of instances in the left/right subset  
                 $m = m_{\text{left}} + m_{\text{right}}$

Once the CART algorithm has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. It stops recursing once it reaches the maximum depth (defined by the `max_depth` hyperparameter), or if it cannot find a split that will reduce impurity. A few other hyperparameters (described in a moment) control additional stopping conditions: `min_samples_split`, `min_samples_leaf`, `max_leaf_nodes`, and more.

## WARNING

As you can see, the CART algorithm is a *greedy algorithm*: it greedily searches for an optimum split at the top level, then repeats the process at each subsequent level. It does not check whether or not the split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a solution that's reasonably good but not guaranteed to be optimal.

Unfortunately, finding the optimal tree is known to be an *NP-complete* problem.<sup>1</sup> It requires  $O(\exp(m))$  time,<sup>2</sup> making the problem intractable even for small training sets. This is why we must settle for a “reasonably good” solution when training decision trees.

## Computational Complexity

Making predictions requires traversing the decision tree from the root to a leaf. Decision trees are generally approximately balanced, so traversing the decision tree requires going through roughly  $O(\log_2(m))$  nodes, where  $m$  is the number of training instances, and  $\log_2(m)$  is the *binary logarithm* of  $m$ , equal to  $\log(m) / \log(2)$ . Since each node only requires checking the value of one feature, the overall prediction complexity is  $O(\log_2(m))$ , independent of the number of features. So predictions are very fast, even when dealing with large training sets.

By default, the training algorithm compares all features on all samples at each node, which results in a training complexity of  $O(n \times m \log_2(m))$ .

It’s possible to set a maximum tree depth using the `max_depth` hyperparameter, and/or set a maximum number of features to consider at each node (the features are then chosen randomly). Doing so will help speed up training considerably, and it can also reduce the risk of overfitting (but as always, going too far would result in underfitting).

## Gini Impurity or Entropy?

By default, the `DecisionTreeClassifier` class uses the Gini impurity measure, but you can select the *entropy* impurity measure instead by setting the `criterion` hyperparameter to "entropy". The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. Entropy later spread to a wide variety of domains, including in Shannon’s information theory, where it measures the average information content of a message, as we saw in [Chapter 4](#). Entropy is zero when all messages are identical. In machine learning, entropy is frequently used as an impurity measure: a set’s entropy is zero when

it contains instances of only one class. [Equation 5-3](#) shows the definition of the entropy of the  $i^{\text{th}}$  node. For example, the depth-2 left node in [Figure 5-1](#) has an entropy equal to  $-(49/54) \log_2 (49/54) - (5/54) \log_2 (5/54) \approx 0.445$ .

*Equation 5-3. Entropy*

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^n p_{i,k} \log_2 (p_{i,k})$$

So, should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.<sup>3</sup>

## Regularization Hyperparameters

Decision trees make very few assumptions about the training data (as opposed to linear models, which assume that the data is linear, for example). If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely—indeed, most likely overfitting it. Such a model is often called a *nonparametric model*, not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. In contrast, a *parametric model*, such as a linear model, has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but increasing the risk of underfitting).

To avoid overfitting the training data, you need to restrict the decision tree’s freedom during training. As you know by now, this is called regularization. The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the decision tree. In Scikit-Learn, this is controlled by the `max_depth` hyperparameter. The default value is `None`, which means unlimited. Reducing `max_depth` will regularize the model and thus reduce the risk of overfitting.

The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the decision tree:

`max_features`

`max_features`  
Maximum number of features that are evaluated for splitting at each node

`max_leaf_nodes`

Maximum number of leaf nodes

`min_samples_split`

Minimum number of samples a node must have before it can be split

`min_samples_leaf`

Minimum number of samples a leaf node must have to be created

`min_weight_fraction_leaf`

Same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances

`min_impurity_decrease`

Only split a node if this split results in at least this reduction in impurity

`ccp_alpha`

Controls *minimal cost-complexity pruning* (MCCP), i.e., pruning subtrees that don't reduce impurity enough compared to their number of leaves; a larger `ccp_alpha` value leads to more pruning, resulting in a smaller tree (the default is 0—no pruning)

To limit the model's complexity and thereby regularize the model, you can increase `min_*` hyperparameters or `ccp_alpha`, or decrease `max_*` hyperparameters. Tuning `max_depth` is usually a good default: it provides effective regularization, and it keeps the tree small and easy to interpret. Setting `min_samples_leaf` is also a good idea, especially for small datasets. And `max_features` is great when working with high-dimensional datasets.

## NOTE

Other algorithms work by first training the decision tree without restrictions, then *pruning* (deleting) unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not statistically significant. Standard statistical tests, such as the  $\chi^2$  test (chi-squared test), are used to estimate the probability that the improvement is purely the result of chance (which is called the *null hypothesis*). If this probability, called the *p-value*, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.

Let's test regularization on the moons dataset: this is a toy dataset for binary classification in which the data points are shaped as two interleaving crescent moons (see [Figure 5-3](#)). You can generate this dataset using the `make_moons()` function.

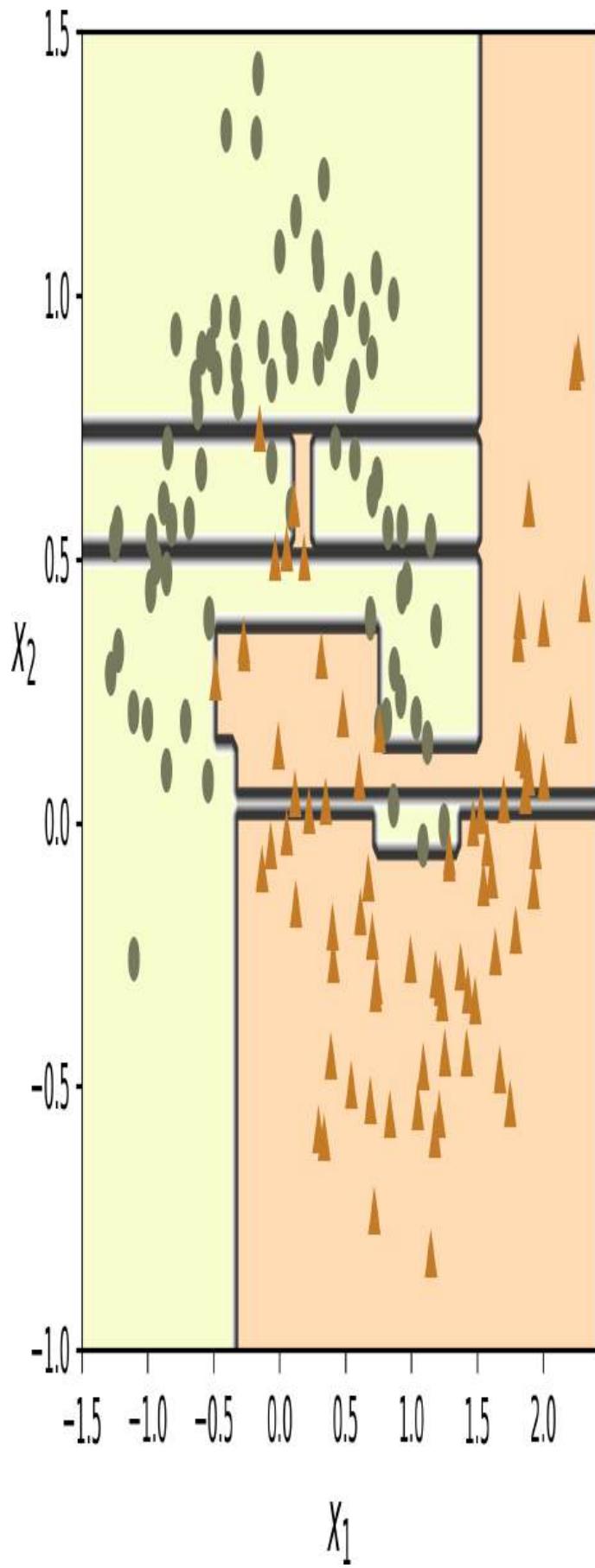
We'll train one decision tree without regularization, and another with `min_samples_leaf=5`. Here's the code; [Figure 5-3](#) shows the decision boundaries of each tree:

```
from sklearn.datasets import make_moons

X_moons, y_moons = make_moons(n_samples=150, noise=0.2, random_state=42)

tree_clf1 = DecisionTreeClassifier(random_state=42)
tree_clf2 = DecisionTreeClassifier(min_samples_leaf=5, random_state=42)
tree_clf1.fit(X_moons, y_moons)
tree_clf2.fit(X_moons, y_moons)
```

No restrictions



min\_samples\_leaf = 5

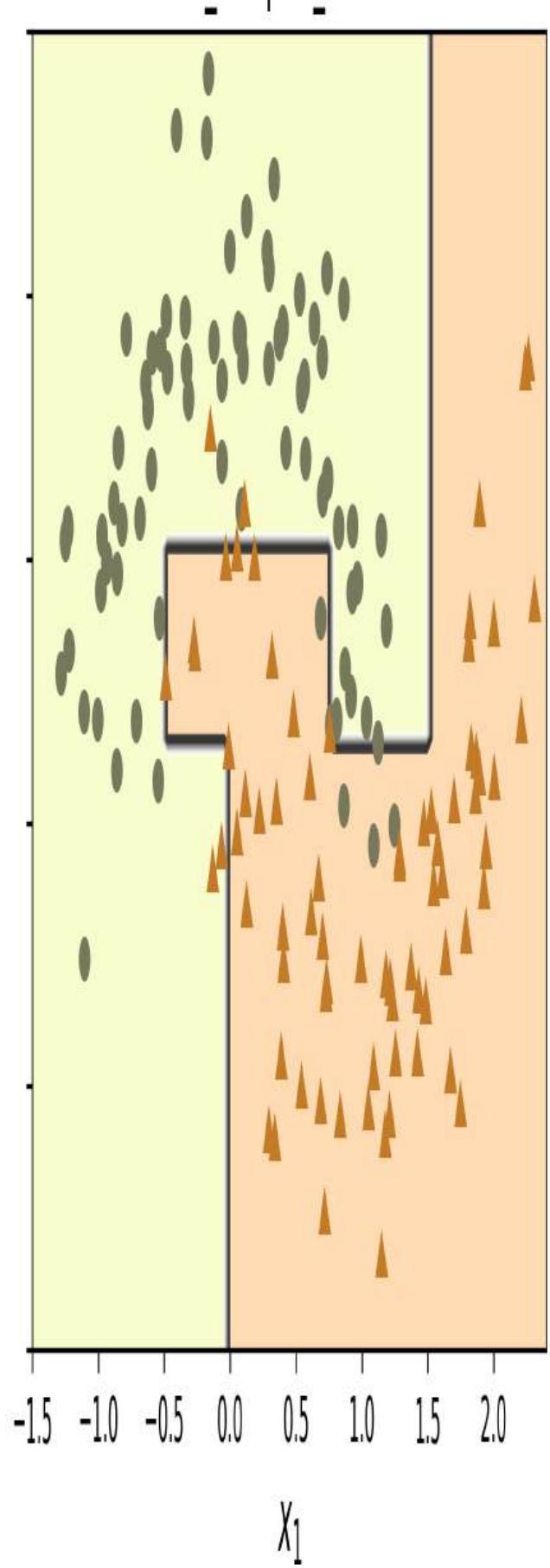


Figure 5-3. Decision boundaries of an unregularized tree (left) and a regularized tree (right)

The unregularized model on the left is clearly overfitting, and the regularized model on the right will probably generalize better. We can verify this by evaluating both trees on a test set generated using a different random seed:

```
>>> X_moons_test, y_moons_test = make_moons(n_samples=1000, noise=0.2,
...                                         random_state=43)
...
>>> tree_clf1.score(X_moons_test, y_moons_test)
0.898
>>> tree_clf2.score(X_moons_test, y_moons_test)
0.92
```

Indeed, the second tree has a better accuracy on the test set.

## Regression

Decision trees are also capable of performing regression tasks. While linear regression only works well with linear data, decision trees can fit all sorts of complex datasets. Let's build a regression tree using Scikit-Learn's `DecisionTreeRegressor` class, training it on a noisy quadratic dataset with `max_depth=2`:

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

rng = np.random.default_rng(seed=42)
X_quad = rng.random((200, 1)) - 0.5 # a single random input feature
y_quad = X_quad ** 2 + 0.025 * rng.standard_normal((200, 1))

tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg.fit(X_quad, y_quad)
```

The resulting tree is represented in Figure 5-4.

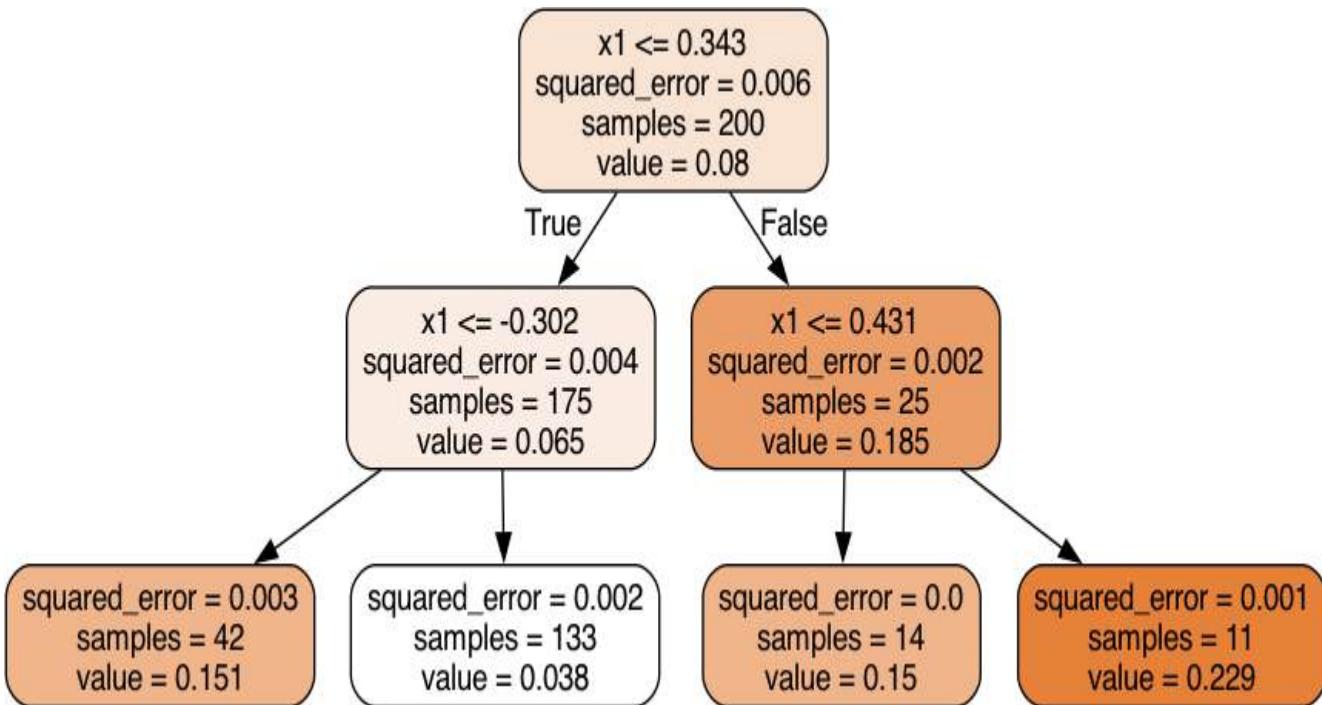


Figure 5-4. A decision tree for regression

This tree looks very similar to the classification tree you built earlier. The main difference is that instead of predicting a class in each node, it predicts a value. For example, suppose you want to make a prediction for a new instance with  $x_1 = 0.2$ . The root node asks whether  $x_1 \leq 0.343$ . Since it is, the algorithm goes to the left child node, which asks whether  $x_1 \leq -0.302$ . Since it is not, the algorithm goes to the right child node. This is a leaf node, and it predicts **value=0.038**. This prediction is the average target value of the 133 training instances associated with this leaf node, and it results in a mean squared error equal to 0.002 over these 133 instances.

This model's predictions are represented on the left in [Figure 5-5](#). If you set `max_depth=3`, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.

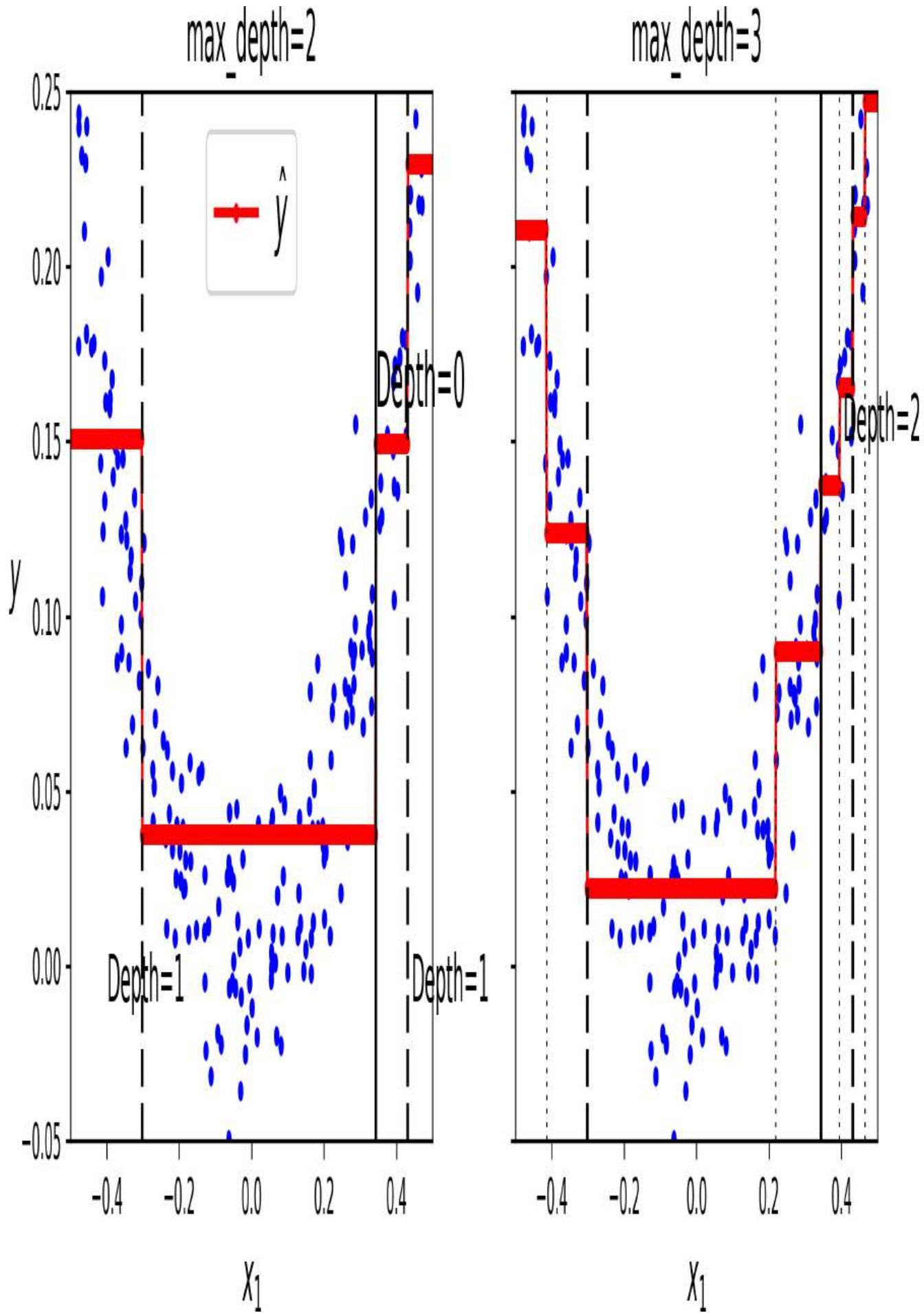


Figure 5-5. Predictions of two decision tree regression models

The CART algorithm works as described earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. [Equation 5-4](#) shows the cost function that the algorithm tries to minimize.

[Equation 5-4. CART cost function for regression](#)

$$J(k, t_k) = \frac{m_{\text{left}}}{m} \text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m} \text{MSE}_{\text{right}} \text{ where } \begin{cases} \text{MSE}_{\text{node}} = \frac{\sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2}{m_{\text{node}}} \\ \hat{y}_{\text{node}} = \frac{\sum_{i \in \text{node}} y^{(i)}}{m_{\text{node}}} \end{cases}$$

Just like for classification tasks, decision trees are prone to overfitting when dealing with regression tasks. Without any regularization (i.e., using the default hyperparameters), you get the predictions on the left in [Figure 5-6](#). These predictions are obviously overfitting the training set very badly. Just setting `min_samples_leaf=10` results in a much more reasonable model, represented on the right in [Figure 5-6](#).

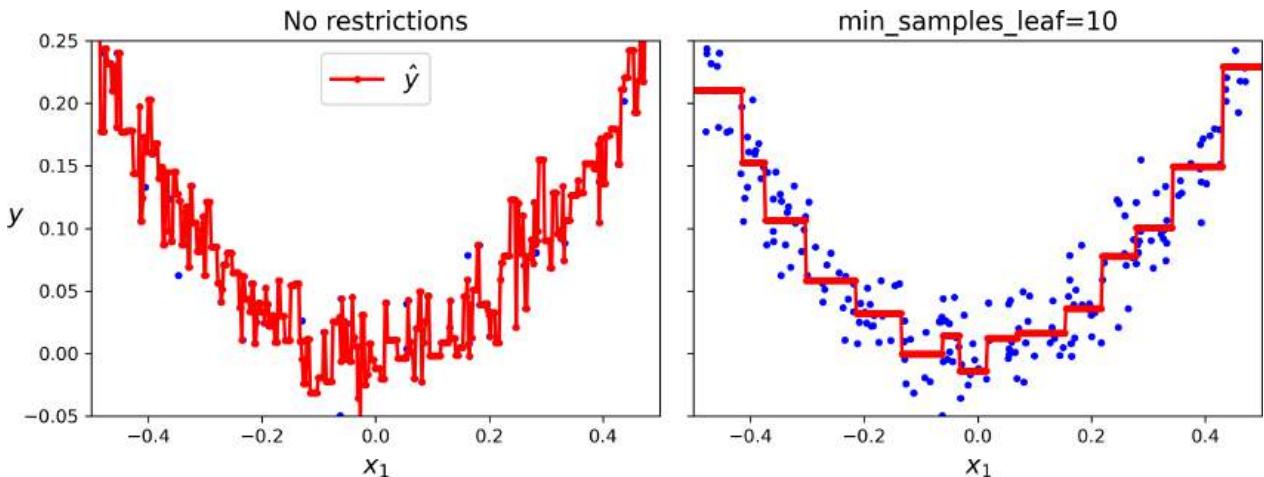
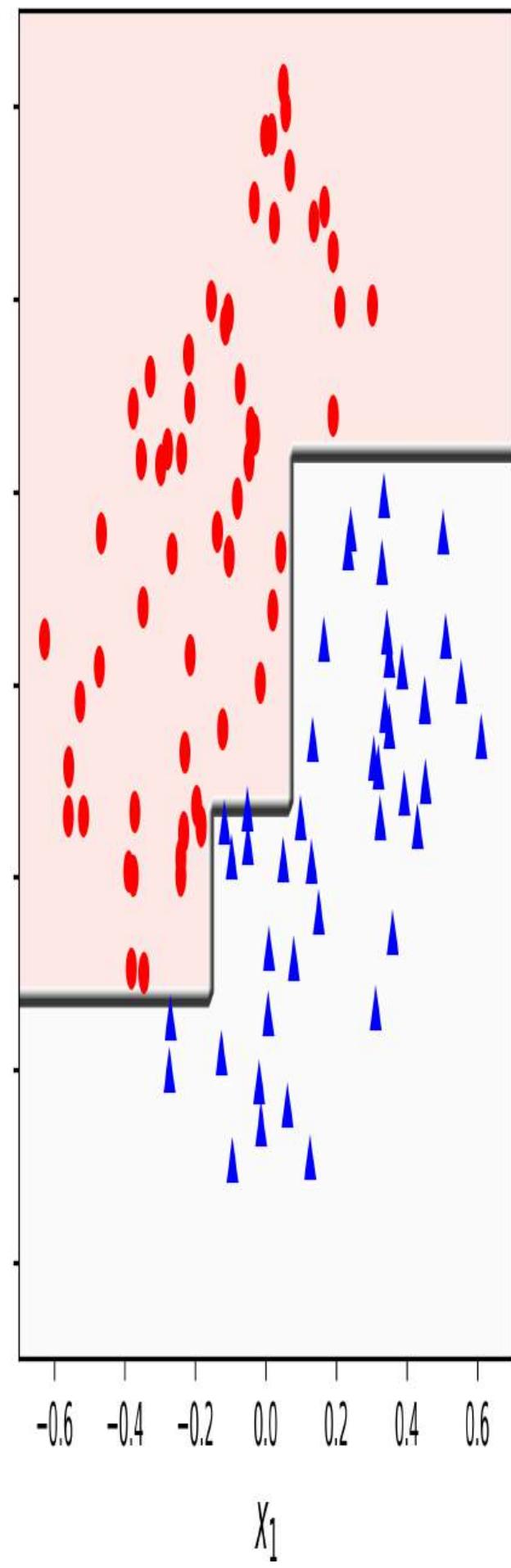
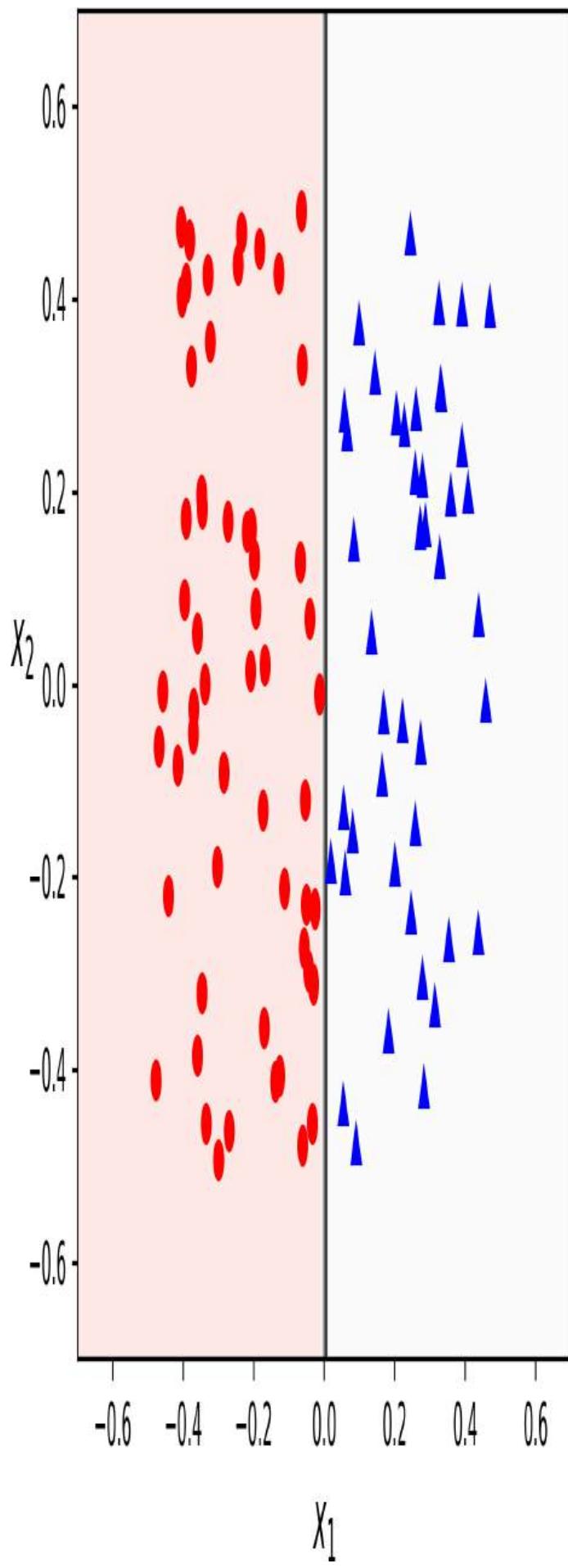


Figure 5-6. Predictions of an unregularized regression tree (left) and a regularized tree (right)

## Sensitivity to Axis Orientation

Hopefully by now you are convinced that decision trees have a lot going for them: they are relatively easy to understand and interpret, simple to use, versatile, and powerful. However, they do have a few limitations. First, as you may have noticed, decision trees love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to the data's orientation. For example, [Figure 5-7](#) shows a simple linearly separable dataset: on the left, a decision tree can split it easily, while on the

right, after the dataset is rotated by  $45^\circ$ , the decision boundary looks unnecessarily convoluted. Although both decision trees fit the training set perfectly, it is very likely that the model on the right will not generalize well.



*Figure 5-7. Sensitivity to training set rotation*

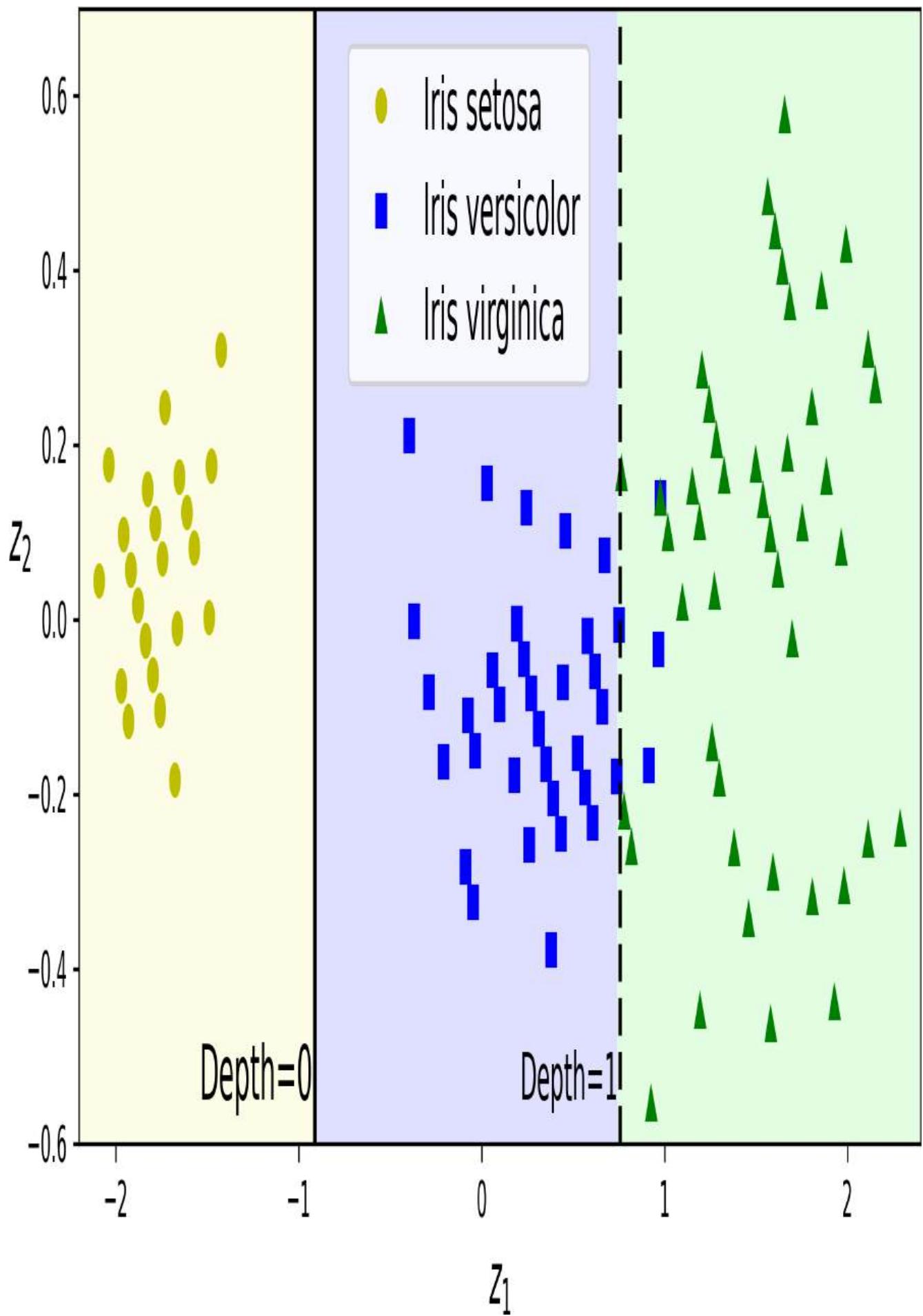
One way to limit this problem is to scale the data, then apply a principal component analysis transformation. We will look at PCA in detail in [Chapter 7](#), but for now you only need to know that it rotates the data in a way that reduces the correlation between the features, which often (not always) makes things easier for trees.

Let's create a small pipeline that scales the data and rotates it using PCA, then train a `DecisionTreeClassifier` on that data. [Figure 5-8](#) shows the decision boundaries of that tree: as you can see, the rotation makes it possible to fit the dataset pretty well using only one feature,  $z_1$ , which is a linear function of the original petal length and width.

Here's the code:

```
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

pca_pipeline = make_pipeline(StandardScaler(), PCA())
X_iris_rotated = pca_pipeline.fit_transform(X_iris)
tree_clf_pca = DecisionTreeClassifier(max_depth=2, random_state=42)
tree_clf_pca.fit(X_iris_rotated, y_iris)
```



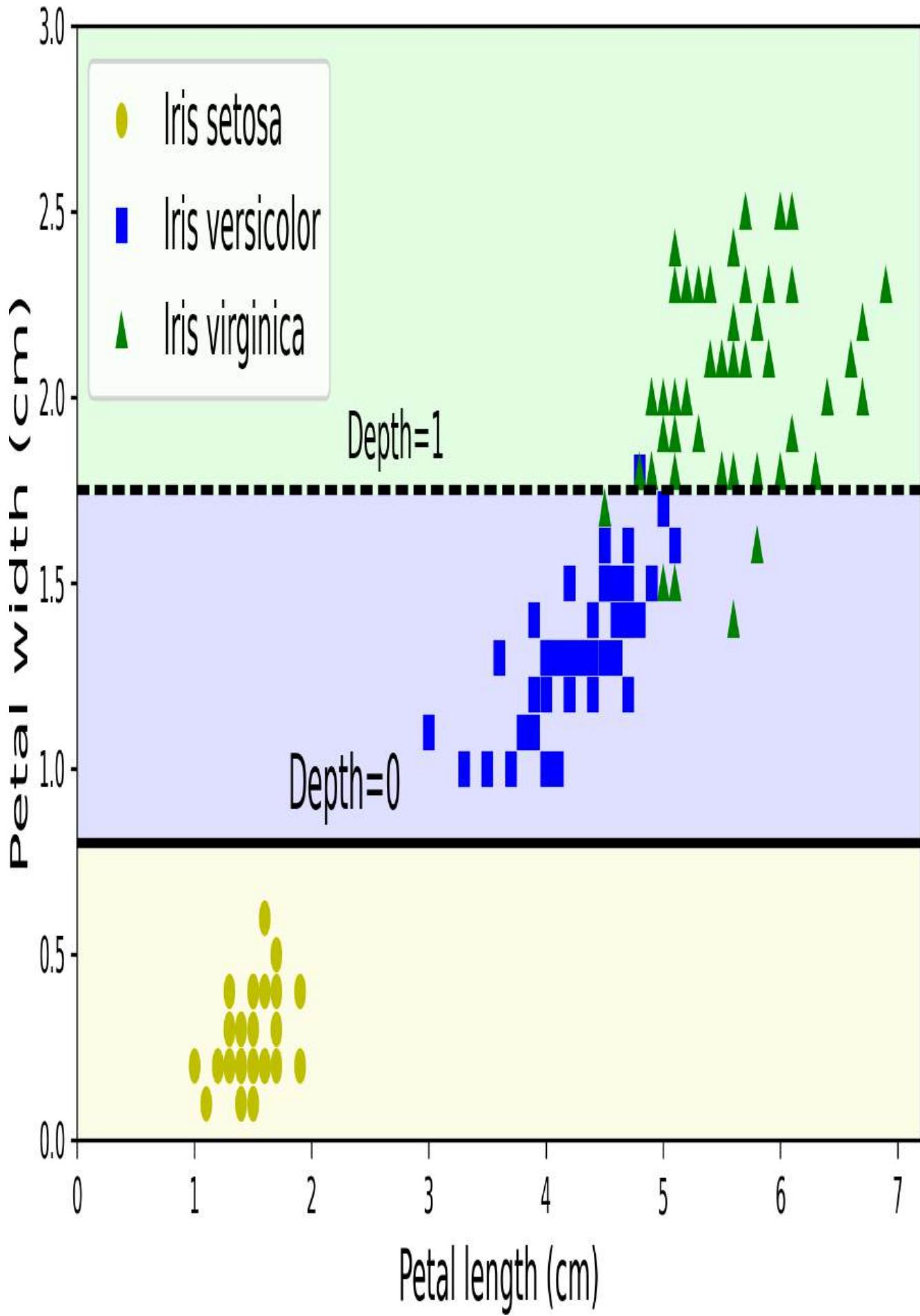
*Figure 5-8. A tree's decision boundaries on the scaled and PCA-rotated iris dataset*

**TIP**

The `DecisionTreeClassifier` and `DecisionTreeRegressor` classes both support missing values natively, no need for an imputer.

## Decision Trees Have a High Variance

More generally, the main issue with decision trees is that they have quite a high variance: small changes to the hyperparameters or to the data may produce very different models. In fact, since the training algorithm used by Scikit-Learn is stochastic—it randomly selects the set of features to evaluate at each node—even retraining the same decision tree on the exact same data may produce a very different model, such as the one represented in [Figure 5-9](#) (unless you set the `random_state` hyperparameter). As you can see, it looks very different from the previous decision tree ([Figure 5-2](#)).



*Figure 5-9. Retraining the same model on the same data may produce a very different model*

Luckily, by averaging predictions over many trees, it's possible to reduce variance significantly. Such an *ensemble* of trees is called a *random forest*, and it's one of the most powerful types of models available today, as you will see in the next chapter.

## Exercises

1. What is the approximate depth of a decision tree trained (without restrictions) on a training set with one million instances?
2. Is a node's Gini impurity generally lower or higher than its parent's? Is it *generally* lower/higher, or *always* lower/higher?
3. If a decision tree is overfitting the training set, is it a good idea to try decreasing `max_depth`?
4. If a decision tree is underfitting the training set, is it a good idea to try scaling the input features?
5. If it takes one hour to train a decision tree on a training set containing one million instances, roughly how much time will it take to train another decision tree on a training set containing ten million instances? Hint: consider the CART algorithm's computational complexity.
6. If it takes one hour to train a decision tree on a given training set, roughly how much time will it take if you double the number of features?
7. Train and fine-tune a decision tree for the moons dataset by following these steps:
  - a. Use `make_moons(n_samples=10000, noise=0.4)` to generate a moons dataset.
  - b. Use `train_test_split()` to split the dataset into a training set and a test set.
  - c. Use grid search with cross-validation (with the help of the `GridSearchCV` class) to find good hyperparameter values for a `DecisionTreeClassifier`. Hint: try various values for `max_leaf_nodes`.

- d. Train it on the full training set using these hyperparameters, and measure your model’s performance on the test set. You should get roughly 85% to 87% accuracy.

8. Grow a forest by following these steps:

- a. Continuing the previous exercise, generate 1,000 subsets of the training set, each containing 100 instances selected randomly. Hint: you can use Scikit-Learn’s `ShuffleSplit` class for this.
- b. Train one decision tree on each subset, using the best hyperparameter values found in the previous exercise. Evaluate these 1,000 decision trees on the test set. Since they were trained on smaller sets, these decision trees will likely perform worse than the first decision tree, achieving only about 80% accuracy.
- c. Now comes the magic. For each test set instance, generate the predictions of the 1,000 decision trees, and keep only the most frequent prediction (you can use SciPy’s `mode()` function for this). This approach gives you *majority-vote predictions* over the test set.
- d. Evaluate these predictions on the test set: you should obtain a slightly higher accuracy than your first model (about 0.5 to 1.5% higher). Congratulations, you have trained a random forest classifier!

Solutions to these exercises are available at the end of this chapter’s notebook, at <https://homl.info/colab-p>.

---

<sup>1</sup> P is the set of problems that can be solved in *polynomial time* (i.e., a polynomial of the dataset size). NP is the set of problems whose solutions can be verified in polynomial time. An NP-hard problem is a problem that can be reduced to a known NP-hard problem in polynomial time. An NP-complete problem is both NP and NP-hard. A major open mathematical question is whether or not P = NP. If P ≠ NP (which seems likely), then no polynomial algorithm will ever be found for any NP-complete problem (except perhaps one day on a quantum computer).

<sup>2</sup> This *big O notation* means that as  $m$  (i.e., the number of training instances) gets larger, the computation time becomes proportional to the exponential of  $m$  (it’s actually an upper bound, but we make it as small as we can). This tells us how “fast” the computation grows with  $m$ , and  $O(\exp(m))$  is very fast.

<sup>3</sup> See Sebastian Raschka’s [interesting analysis](#) for more details.

# Chapter 6. Ensemble Learning and Random Forests

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 6th chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

Suppose you pose a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert’s answer. This is called the *wisdom of the crowd*. Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*; thus, this technique is called *ensemble learning*, and an ensemble learning algorithm is called an *ensemble method*.

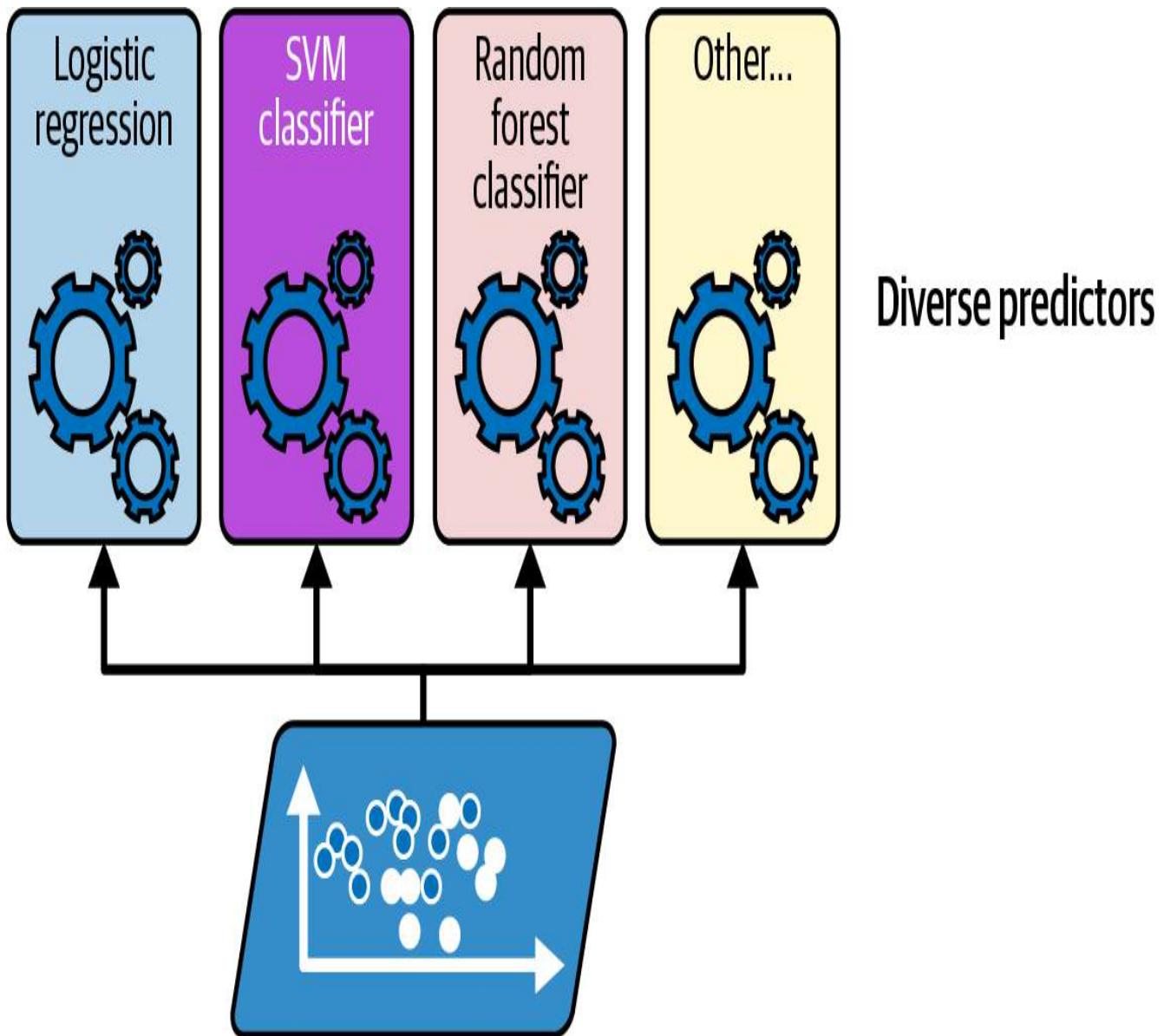
As an example of an ensemble method, you can train a group of decision tree classifiers, each on a different random subset of the training set. You can then obtain the predictions of all the individual trees, and the class that gets the most votes is the ensemble’s prediction (see the last exercise in [Chapter 5](#)). Such an ensemble of decision trees is called a *random forest*, and despite its simplicity, this is one of the most powerful machine learning algorithms available today.

As discussed in [Chapter 2](#), you will often use ensemble methods near the end of a project, once you have already built a few good predictors, to combine them into an even better predictor. In fact, the winning solutions in machine learning competitions often involve several ensemble methods—most famously in the [Netflix Prize competition](#). There are some downsides, however: ensemble learning requires much more computing resources than using a single model (both for training and for inference), it can be more complex to deploy and manage, and the predictions are harder to interpret. But the pros often outweigh the cons.

In this chapter we will examine the most popular ensemble methods, including voting classifiers, bagging and pasting ensembles, random forests, boosting, and stacking ensembles.

## Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a logistic regression classifier, an SVM classifier, a random forest classifier, a  $k$ -nearest neighbors classifier, and perhaps a few more (see [Figure 6-1](#)).



*Figure 6-1. Training diverse classifiers*

A very simple way to create an even better classifier is to aggregate the predictions of each classifier: the class that gets the most votes is the ensemble's prediction. This majority-vote classifier is called a *hard voting* classifier (see [Figure 6-2](#)).

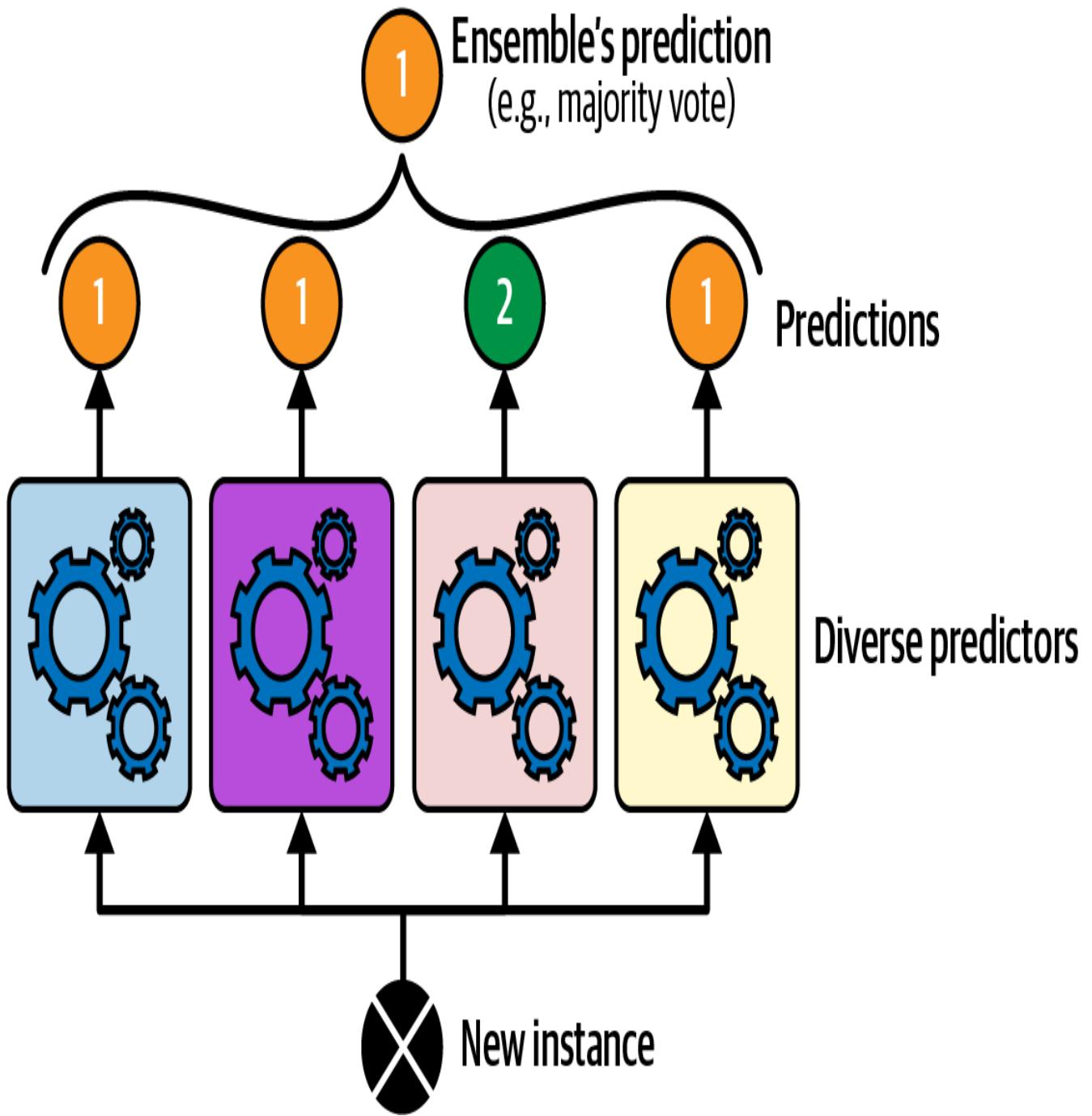


Figure 6-2. Hard voting classifier predictions

Somewhat surprisingly, this voting classifier often achieves a higher accuracy than the best classifier in the ensemble. In fact, even if each classifier is a *weak learner* (meaning it does only slightly better than random guessing), the ensemble can still be a *strong learner* (achieving high accuracy), provided there are a sufficient number of weak learners in the ensemble and they are sufficiently diverse (i.e., if they focus on different aspects of the data and make different kinds of errors).

How is this possible? The following analogy can help shed some light on this mystery. Suppose you have a slightly biased coin that has a 51% chance of coming up heads and

49% chance of coming up tails. If you toss it 1,000 times, you will generally get more or less 510 heads and 490 tails, and hence a majority of heads. If you do the math, you will find that the probability of obtaining a majority of heads after 1,000 tosses is close to 75%. The more you toss the coin, the higher the probability (e.g., with 10,000 tosses, the probability climbs over 97%). This is due to the *law of large numbers*: as you keep tossing the coin, the ratio of heads gets closer and closer to the probability of heads (51%). [Figure 6-3](#) shows 10 series of biased coin tosses. You can see that as the number of tosses increases, the ratio of heads approaches 51%. Eventually all 10 series end up so close to 51% that they are consistently above 50%.

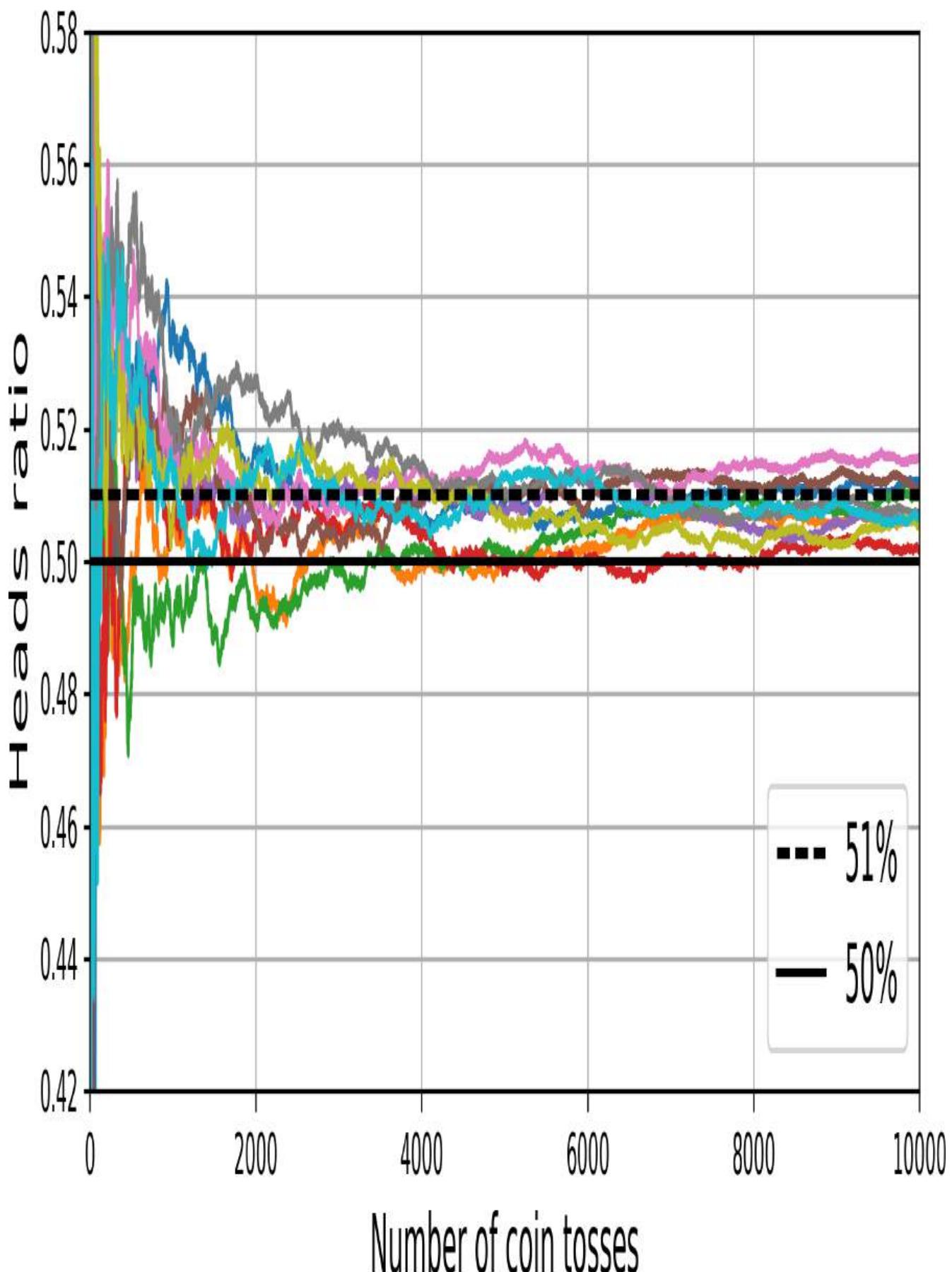


Figure 6-3. The law of large numbers

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case because they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

### TIP

Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy. You can also play with the model hyperparameters to get diverse models, or train the models on different subsets of the data, as we will see.

Scikit-Learn provides a `VotingClassifier` class that's quite easy to use: just give it a list of name/predictor pairs, and use it like a normal classifier. Let's try it on the moons dataset (introduced in [Chapter 5](#)). We will load and split the moons dataset into a training set and a test set, then we'll create and train a voting classifier composed of three diverse classifiers:

```
from sklearn.datasets import make_moons
from sklearn.ensemble import RandomForestClassifier, VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

voting_clf = VotingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(random_state=42))
    ]
)
voting_clf.fit(X_train, y_train)
```

When you fit a `VotingClassifier`, it clones every estimator and fits the clones. The original estimators are available via the `estimators` attribute, while the fitted clones are available via the `estimators_` attribute. If you prefer a dict rather than a list, you can use

`named_estimators` or `named_estimators_` instead. To begin, let's look at each fitted classifier's accuracy on the test set:

```
>>> for name, clf in voting_clf.named_estimators_.items():
...     print(name, "=", clf.score(X_test, y_test))
...
lr = 0.864
rf = 0.896
svc = 0.896
```

When you call the voting classifier's `predict()` method, it performs hard voting. For example, the voting classifier predicts class 1 for the first instance of the test set, because two out of three classifiers predict that class:

```
>>> voting_clf.predict(X_test[:1])
array([1])
>>> [clf.predict(X_test[:1]) for clf in voting_clf.estimators_]
[array([1]), array([1]), array([0])]
```

Now let's look at the performance of the voting classifier on the test set:

```
>>> voting_clf.score(X_test, y_test)
0.912
```

There you have it! The voting classifier outperforms all the individual classifiers.

If all classifiers are able to estimate class probabilities (i.e., if they all have a `predict_proba()` method), then you should generally tell Scikit-Learn to predict the class with the highest class probability, averaged over all the individual classifiers. This is called *soft voting*. It often achieves higher performance than hard voting because it gives more weight to highly confident votes. All you need to do is set the voting classifier's `voting` hyperparameter to "soft", and ensure that all classifiers can estimate class probabilities. This is not the case for the SVC class by default, so you need to set its `probability` hyperparameter to `True` (this will make the SVC class use cross-validation to estimate class probabilities, slowing down training, and it will add a `predict_proba()` method). Let's try that:

```
>>> voting_clf.voting = "soft"
>>> voting_clf.named_estimators["svc"].probability = True
>>> voting_clf.fit(X_train, y_train)
>>> voting_clf.score(X_test, y_test)
0.92
```

We reach 92% accuracy simply by using soft voting—not bad!

### TIP

Soft voting works best when the estimated probabilities are well-calibrated. If they are not, you can use `sklearn.calibration.CalibratedClassifierCV` to calibrate them (see [Chapter 3](#)).

## Bagging and Pasting

One way to get a diverse set of classifiers is to use very different training algorithms, as just discussed. Another approach is to use the same training algorithm for every predictor but train them on different random subsets of the training set. When sampling is performed *with replacement*,<sup>1</sup> this method is called *bagging*<sup>2</sup> (short for *bootstrap aggregating*<sup>3</sup>). When sampling is performed *without replacement*, it is called *pasting*.<sup>4</sup>

In other words, both bagging and pasting allow training instances to be sampled several times across multiple predictors, but only bagging allows training instances to be sampled several times for the same predictor. This sampling and training process is represented in [Figure 6-4](#).

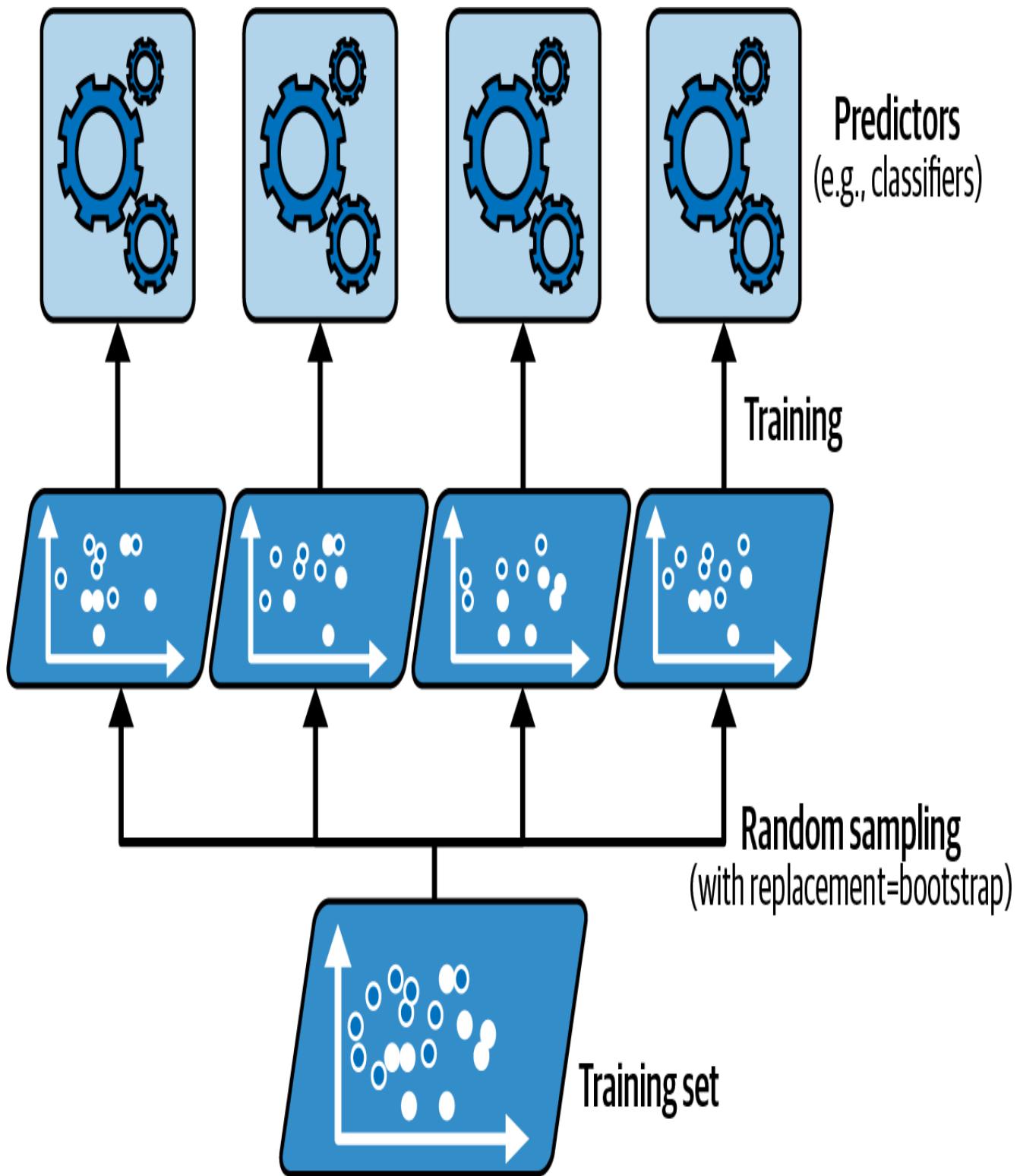


Figure 6-4. Bagging and pasting involve training several predictors on different random samples of the training set

Once all predictors are trained, the ensemble can make a prediction for a new instance by simply aggregating the predictions of all predictors. For classification, the aggregation function is typically the *statistical mode* (i.e., the most frequent prediction, just like with a hard voting classifier), and for regression it's usually just the average. Each individual

predictor has a higher bias than if it were trained on the original training set, but aggregation reduces both bias and variance.<sup>5</sup>

To get an intuition of why this is the case, imagine that you trained two regressors to predict house prices. The first underestimates the prices by \$40,000 on average, while the second overestimates them by \$50,000 on average. Assuming these regressors are 100% independent and their predictions follow a Normal distribution, if you compute the average of the two predictions, the result will overestimate the prices by only  $(-40,000 + 50,000)/2 = \$5,000$  on average: that's a much lower bias! Similarly, if both predictors have a \$10,000 standard deviation (i.e., a variance of 100,000,000), then the average prediction will have a variance of  $(10,000^2 + 10,000^2)/2^2 = 50,000,000$  (i.e., the standard deviation will be \$7,071). The variance is halved!

In practice, the ensemble often ends up with a similar bias but a lower variance than a single predictor trained on the original training set. Therefore it works best with high-variance and low-bias models (e.g., ensembles of decision trees, not ensembles of linear regressors).

### TIP

Prefer bagging when your dataset is noisy or your model is prone to overfitting (e.g., deep decision tree). Otherwise, prefer pasting as it avoids redundancy during training, making it a bit more computationally efficient.

As you can see in [Figure 6-4](#), predictors can all be trained in parallel, via different CPU cores or even different servers. Similarly, predictions can be made in parallel. This is one of the reasons bagging and pasting are such popular methods: they scale very well.

## Bagging and Pasting in Scikit-Learn

Scikit-Learn offers a simple API for both bagging and pasting: `BaggingClassifier` class (or `BaggingRegressor` for regression). The following code trains an ensemble of 500 decision tree classifiers:<sup>6</sup> each is trained on 100 training instances randomly sampled from the training set with replacement (this is an example of bagging, but if you want to use pasting instead, just set `bootstrap=False`). The `n_jobs` parameter tells Scikit-Learn the number of CPU cores to use for training and predictions, and `-1` tells Scikit-Learn to use all available cores:

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
```

```
bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
                            max_samples=100, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
```

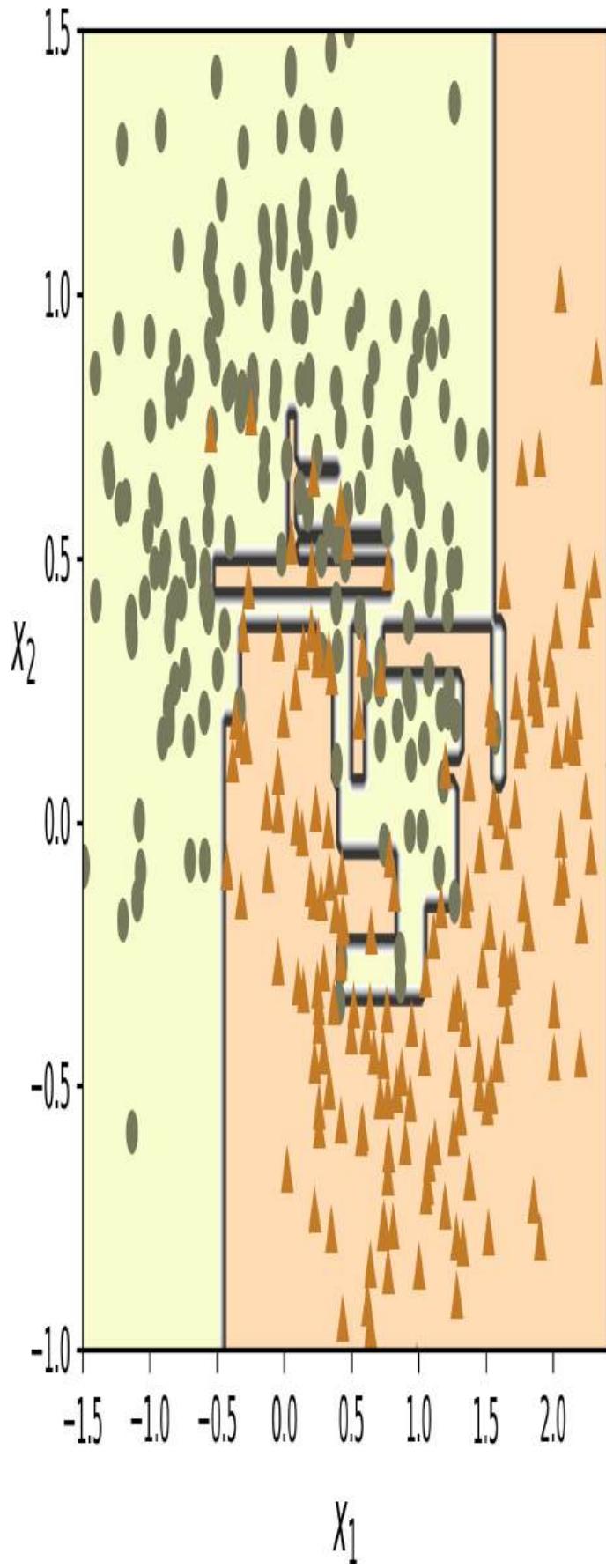
## NOTE

A `BaggingClassifier` automatically performs soft voting instead of hard voting if the base classifier can estimate class probabilities (i.e., if it has a `predict_proba()` method), which is the case with decision tree classifiers.

**Figure 6-5** compares the decision boundary of a single decision tree with the decision boundary of a bagging ensemble of 500 trees (from the preceding code), both trained on the moons dataset. As you can see, the ensemble's predictions will likely generalize much better than the single decision tree's predictions: the ensemble has a comparable bias but a smaller variance (it makes roughly the same number of errors on the training set, but the decision boundary is less irregular).

Bagging introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting; but the extra diversity also means that the predictors end up being less correlated, so the ensemble's variance is reduced. Overall, bagging often results in better models, which explains why it's generally preferred. But if you have spare time and CPU power, you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

Decision Tree



Decision Trees with Bagging

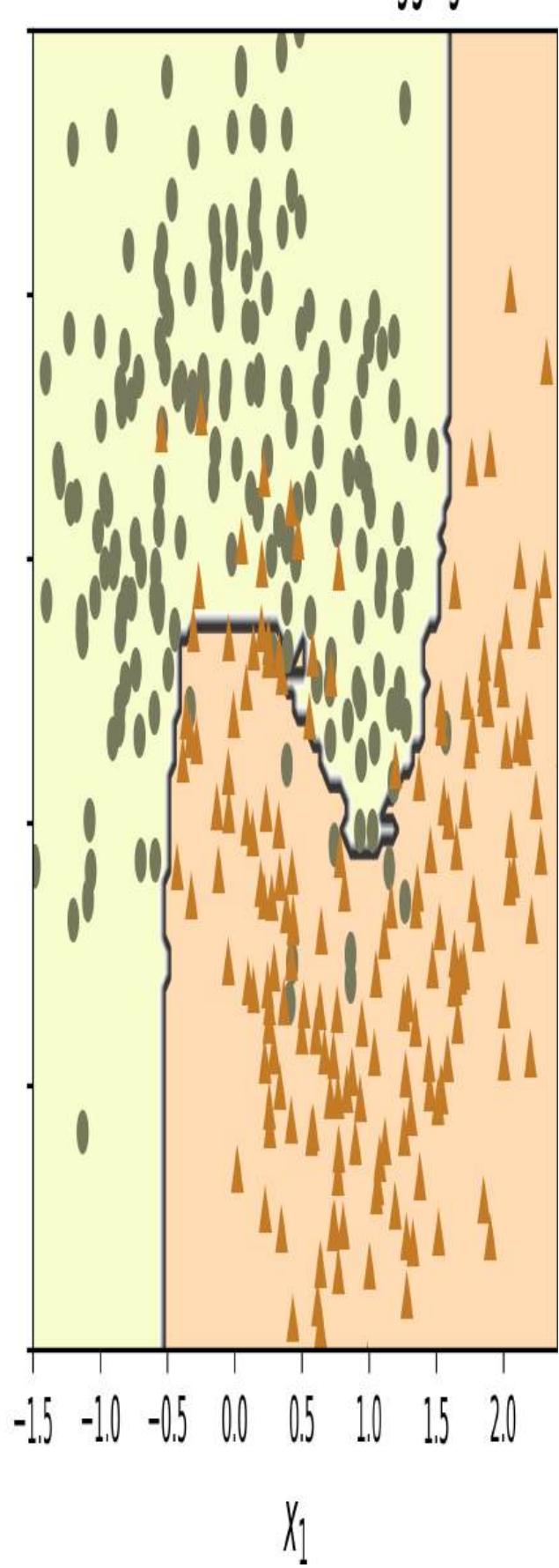


Figure 6-5. A single decision tree (left) versus a bagging ensemble of 500 trees (right)

## Out-of-Bag Evaluation

With bagging, some training instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a `BaggingClassifier` samples  $m$  training instances with replacement (`bootstrap=True`), where  $m$  is the size of the training set. With this process, it can be shown mathematically that only about 63% of the training instances are sampled on average for each predictor.<sup>7</sup> The remaining 37% of the training instances that are not sampled are called *out-of-bag* (OOB) instances. Note that they are not the same 37% for all predictors.

A bagging ensemble can be evaluated using OOB instances, without the need for a separate validation set: indeed, if there are enough estimators, then each instance in the training set will likely be an OOB instance of several estimators, so these estimators can be used to make a fair ensemble prediction for that instance. Once you have a prediction for each instance, you can compute the ensemble's prediction accuracy (or any other metric).

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic OOB evaluation after training. The following code demonstrates this. The resulting evaluation score is available in the `oob_score_` attribute:

```
>>> bag_clf = BaggingClassifier(DecisionTreeClassifier(), n_estimators=500,
...                               oob_score=True, n_jobs=-1, random_state=42)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.896
```

According to this OOB evaluation, this `BaggingClassifier` is likely to achieve about 89.6% accuracy on the test set. Let's verify this:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.912
```

We get 91.2% accuracy on the test. The OOB evaluation was a bit too pessimistic.

The OOB decision function for each training instance is also available through the `oob_decision_function_` attribute. Since the base estimator has a `predict_proba()` method, the decision function returns the class probabilities for each training instance.

For example, the OOB evaluation estimates that the first training instance has a 67.6% probability of belonging to the positive class and a 32.4% probability of belonging to the negative class:

```
>>> bag_clf.oob_decision_function_[:3] # probas for the first 3 instances
array([[0.32352941, 0.67647059],
       [0.3375     , 0.6625     ],
       [1.         , 0.         ]])
```

## Random Patches and Random Subspaces

The `BaggingClassifier` class supports sampling the features as well. Sampling is controlled by two hyperparameters: `max_features` and `bootstrap_features`. They work the same way as `max_samples` and `bootstrap`, but for feature sampling instead of instance sampling. Thus, each predictor will be trained on a random subset of the input features.

This technique is particularly useful when you are dealing with high-dimensional inputs (such as images), as it can considerably speed up training. Sampling both training instances and features is called the *random patches* method.<sup>8</sup> Keeping all training instances (by setting `bootstrap=False` and `max_samples=1.0`) but sampling features (by setting `bootstrap_features` to `True` and/or `max_features` to a value smaller than `1.0`) is called the *random subspaces* method.<sup>9</sup>

Sampling features results in even more predictor diversity, trading a bit more bias for a lower variance.

## Random Forests

As we have discussed, a *random forest*<sup>10</sup> is an ensemble of decision trees, generally trained via the bagging method (or sometimes pasting), typically with `max_samples` set to the size of the training set. Instead of building a `BaggingClassifier` and passing it a `DecisionTreeClassifier`, you can use the `RandomForestClassifier` class, which is more convenient and optimized for decision trees<sup>11</sup> (similarly, there is a `RandomForestRegressor` class for regression tasks). The following code trains a random forest classifier with 500 trees, each limited to maximum 16 leaf nodes, using all available CPU cores:

```
from sklearn.ensemble import RandomForestClassifier
```

```
rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16,
                                  n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)

y_pred_rf = rnd_clf.predict(X_test)
```

With a few exceptions, a `RandomForestClassifier` has all the hyperparameters of a `DecisionTreeClassifier` (to control how trees are grown), plus all the hyperparameters of a `BaggingClassifier` to control the ensemble itself.

The `RandomForestClassifier` class introduces extra randomness when growing trees: instead of searching for the very best feature when splitting a node (see [Chapter 5](#)), it searches for the best feature among a random subset of features. By default, it samples  $\sqrt{n}$  features (where  $n$  is the total number of features). The algorithm results in greater tree diversity, which (again) trades a higher bias for a lower variance, generally yielding an overall better model. So, the following `BaggingClassifier` is equivalent to the previous `RandomForestClassifier`:

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(max_features="sqrt", max_leaf_nodes=16),
    n_estimators=500, n_jobs=-1, random_state=42)
```

## Extra-Trees

When you are growing a tree in a random forest, at each node only a random subset of the features is considered for splitting (as discussed earlier). It is possible to make trees even more random by also using random thresholds for each feature rather than searching for the best possible thresholds (like regular decision trees do). For this, simply set `splitter="random"` when creating a `DecisionTreeClassifier`.

A forest of such extremely random trees is called an *extremely randomized trees*<sup>12</sup> (or *extra-trees* for short) ensemble. Once again, this technique trades more bias for a lower variance, so they may perform better than regular random forests if you encounter overfitting, especially with noisy and/or high-dimensional datasets. Extra-trees classifiers are also much faster to train than regular random forests, because finding the best possible threshold for each feature at every node is one of the most time-consuming tasks of growing a tree in a random forest.

You can create an extra-trees classifier using Scikit-Learn's `ExtraTreesClassifier` class. Its API is identical to the `RandomForestClassifier` class, except `bootstrap` defaults to `False`. Similarly, the `ExtraTreesRegressor` class has the same API as the `RandomForestRegressor` class, except `bootstrap` defaults to `False`.

## TIP

Just like decision tree classes, random forest classes and extra-trees classes in recent Scikit-Learn versions support missing values natively, no need for an imputer.

## Feature Importance

Yet another great quality of random forests is that they make it easy to measure the relative importance of each feature. Scikit-Learn measures a feature's importance by looking at how much the tree nodes that use that feature reduce impurity on average, across all trees in the forest. More precisely, it is a weighted average, where each node's weight is equal to the number of training samples that are associated with it (see [Chapter 5](#)).

Scikit-Learn computes this score automatically for each feature after training, then it scales the results so that the sum of all importances is equal to 1. You can access the result using the `feature_importances_` variable. For example, the following code trains a `RandomForestClassifier` on the iris dataset (introduced in [Chapter 4](#)) and outputs each feature's importance. It seems that the most important features are the petal length (44%) and width (42%), while sepal length and width are rather unimportant in comparison (11% and 2%, respectively):

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris(as_frame=True)
>>> rnd_clf = RandomForestClassifier(n_estimators=500, random_state=42)
>>> rnd_clf.fit(iris.data, iris.target)
>>> for score, name in zip(rnd_clf.feature_importances_, iris.data.columns):
...     print(round(score, 2), name)
...
0.11  sepal length (cm)
0.02  sepal width (cm)
0.44  petal length (cm)
0.42  petal width (cm)
```

Similarly, if you train a random forest classifier on the MNIST dataset (introduced in [Chapter 3](#)) and plot each pixel's importance, you get the image represented in [Figure 6-6](#).

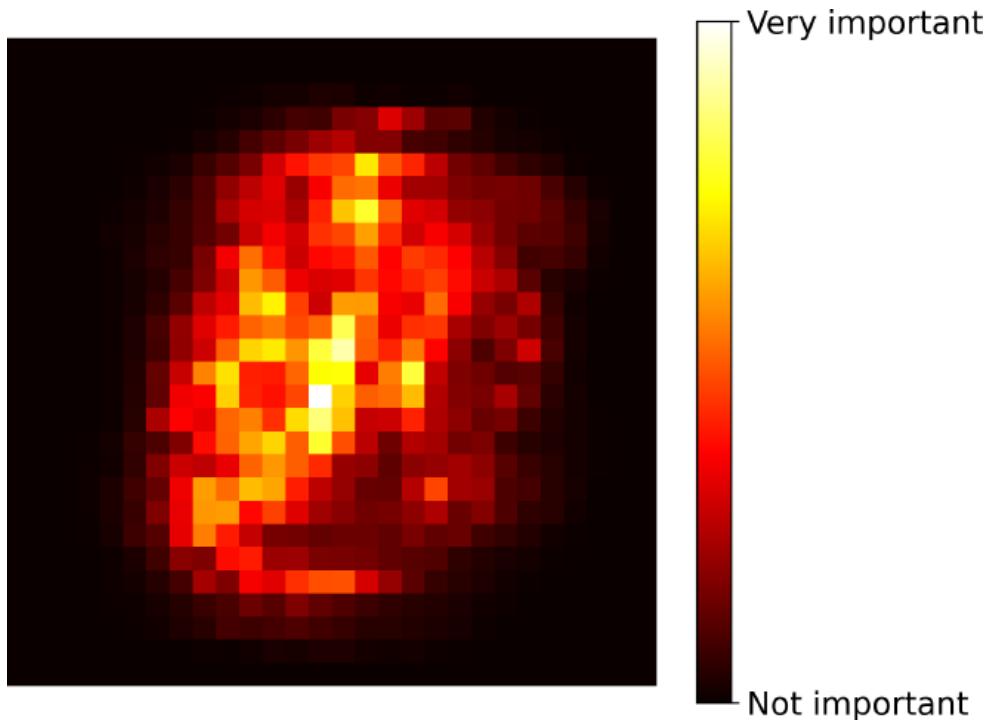


Figure 6-6. MNIST pixel importance (according to a random forest classifier)

Random forests are very handy to get a quick understanding of what features actually matter, in particular if you need to perform feature selection.

## Boosting

*Boosting* (originally called *hypothesis boosting*) refers to any ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. There are many boosting methods available, but by far the most popular are *AdaBoost*<sup>13</sup> (short for *adaptive boosting*) and *gradient boosting*. Let's start with AdaBoost.

## AdaBoost

One way for a new predictor to correct its predecessor is to pay a bit more attention to the training instances that the predecessor underfit. This results in new predictors focusing more and more on the hard cases. This is the technique used by AdaBoost.

For example, when training an AdaBoost classifier, the algorithm first trains a base classifier (such as a decision tree) and uses it to make predictions on the training set. The algorithm then increases the relative weight of misclassified training instances. Then it trains a second classifier, using the updated weights, and again makes predictions on the training set, updates the instance weights, and so on (see Figure 6-7).

Figure 6-8 shows the decision boundaries of five consecutive predictors on the moons dataset (in this example, each predictor is a highly regularized SVM classifier with an RBF kernel).<sup>14</sup> The first classifier gets many instances wrong, so their weights get boosted. The second classifier therefore does a better job on these instances, and so on. The plot on the right represents the same sequence of predictors, except that the learning rate is halved (i.e., the misclassified instance weights are boosted much less at every iteration). As you can see, this sequential learning technique has some similarities with gradient descent, except that instead of tweaking a single predictor's parameters to minimize a cost function, AdaBoost adds predictors to the ensemble, gradually making it better.

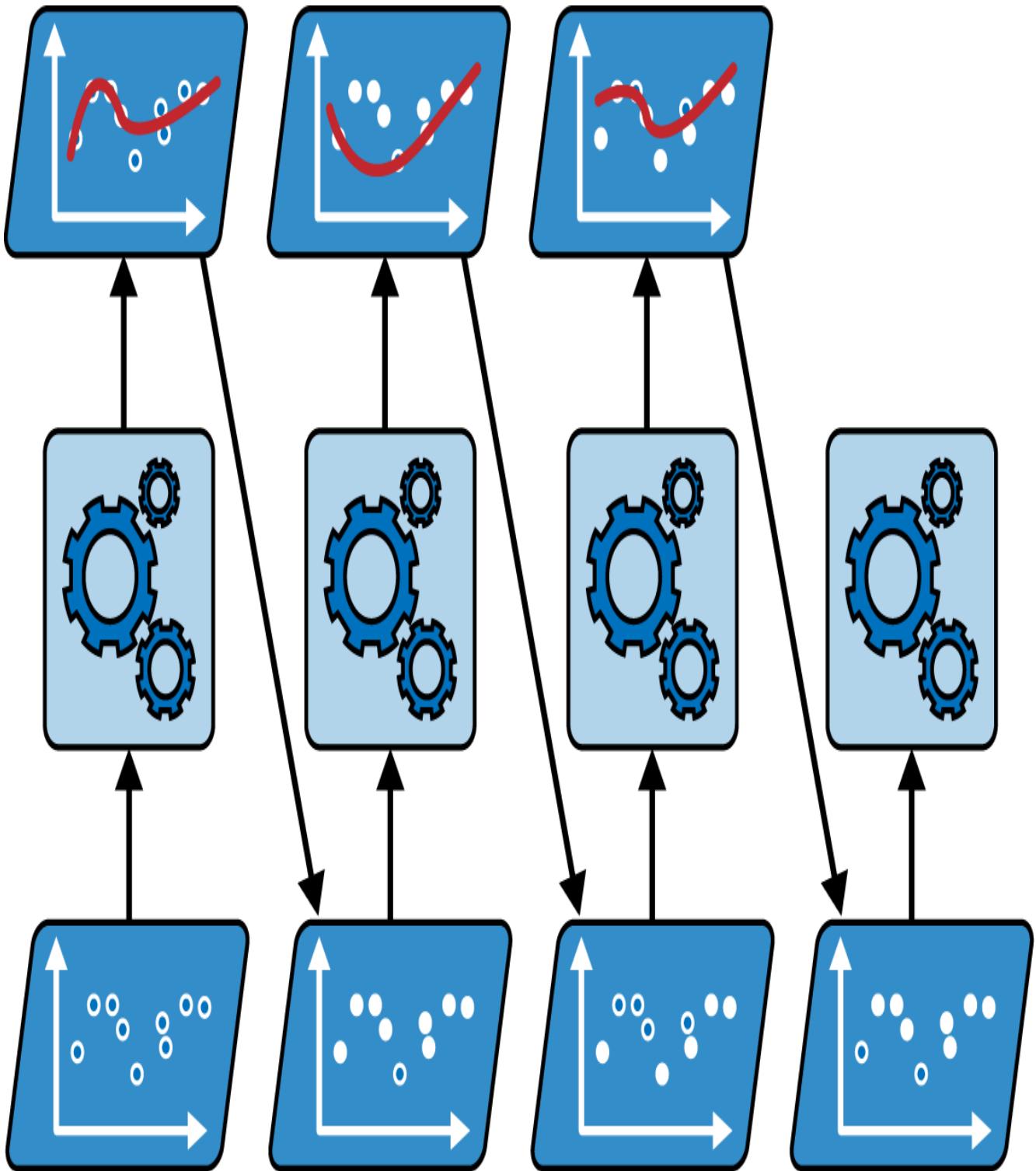


Figure 6-7. AdaBoost sequential training with instance weight updates

Once all predictors are trained, the ensemble makes predictions very much like bagging or pasting, except that predictors have different weights depending on their overall accuracy on the weighted training set.

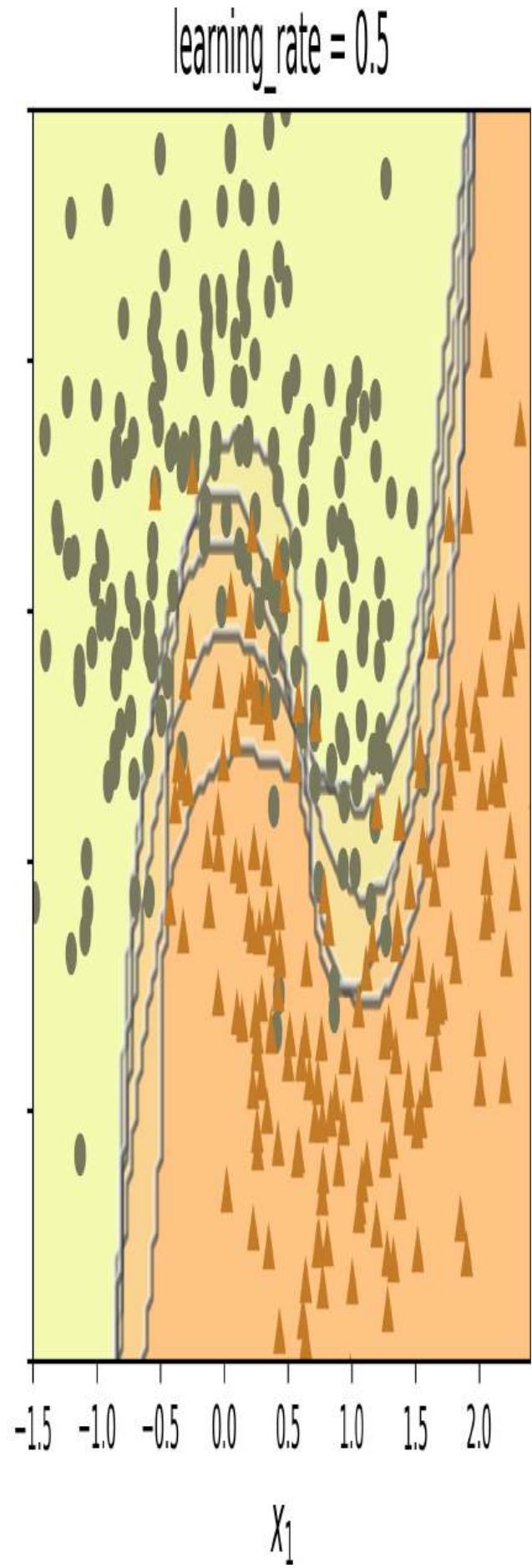
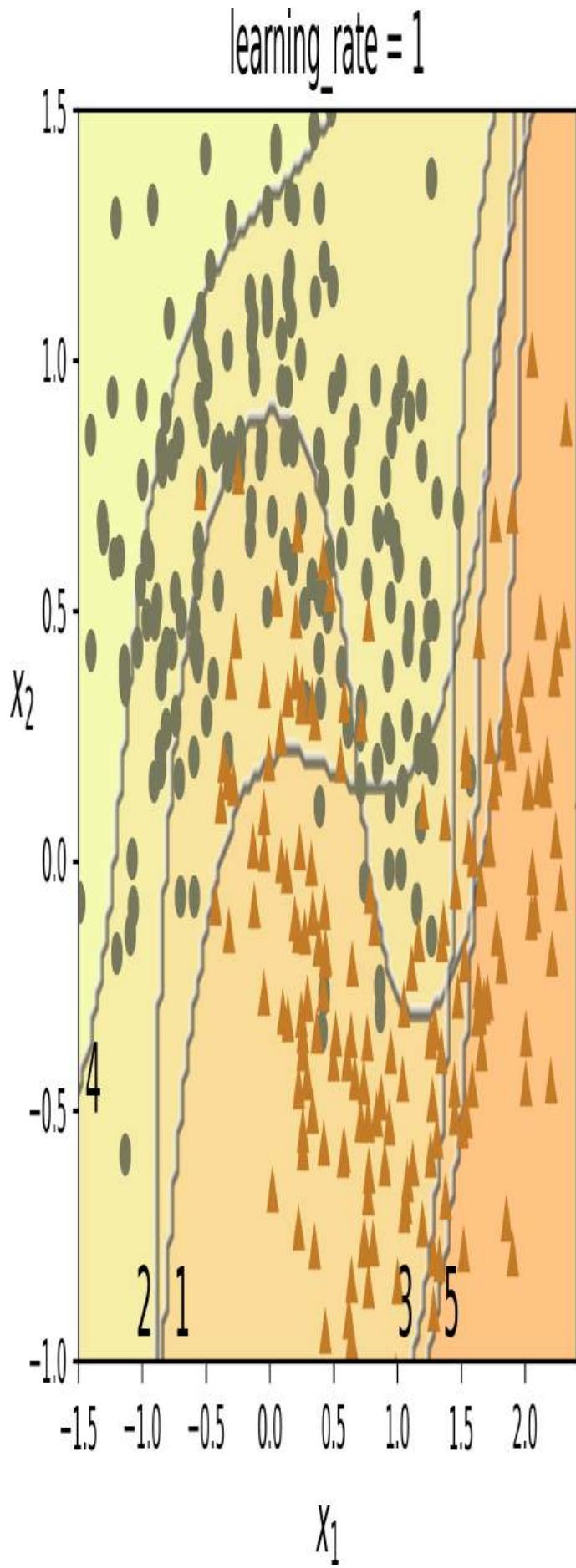


Figure 6-8. Decision boundaries of consecutive predictors

## WARNING

There is one important drawback to this sequential learning technique: training cannot be parallelized since each predictor can only be trained after the previous predictor has been trained and evaluated. As a result, it does not scale as well as bagging or pasting.

Let's take a closer look at the AdaBoost algorithm. Each instance weight  $w^{(i)}$  is initially set to  $1/m$ , so their sum is 1. A first predictor is trained, and its weighted error rate  $r_1$  is computed on the training set; see [Equation 6-1](#).

*Equation 6-1. Weighted error rate of the  $j^{\text{th}}$  predictor*

$$r_j = \sum_{\substack{i=1 \\ \hat{y}_j^{(i)} \neq y^{(i)}}}^m w^{(i)} \quad \text{where } \hat{y}_j^{(i)} \text{ is the } j^{\text{th}} \text{ predictor's prediction for the } i^{\text{th}} \text{ instance}$$

The predictor's weight  $\alpha_j$  is then computed using [Equation 6-2](#), where  $\eta$  is the learning rate hyperparameter (defaults to 1).<sup>15</sup> The more accurate the predictor is, the higher its weight will be. If it is just guessing randomly, then its weight will be close to zero. However, if it is most often wrong (i.e., less accurate than random guessing), then its weight will be negative.

*Equation 6-2. Predictor weight*

$$\alpha_j = \eta \log \frac{1-r_j}{r_j}$$

Next, the AdaBoost algorithm updates the instance weights, using [Equation 6-3](#), which boosts the weights of the misclassified instances and encourages the next predictor to pay more attention to them.

*Equation 6-3. Weight update rule*

for  $i = 1, 2, \dots, m$

$$w^{(i)} \leftarrow \begin{cases} w^{(i)} & \text{if } \hat{y}^{(i)} = y^{(i)} \\ w^{(i)} \exp(\alpha_j) & \text{if } \hat{y}^{(i)} \neq y^{(i)} \end{cases}$$

Then all the instance weights are normalized to ensure that their sum is once again 1 (i.e., they are divided by  $\sum_{i=1}^m w^{(i)}$ ).

Finally, a new predictor is trained using the updated weights, and the whole process is repeated: the new predictor's weight is computed, the instance weights are updated, then another predictor is trained, and so on. The algorithm stops when the desired number of predictors is reached, or when a perfect predictor is found.

To make predictions, AdaBoost simply computes the predictions of all the predictors and weighs them using the predictor weights  $\alpha_j$ . The predicted class is the one that receives the majority of weighted votes (see [Equation 6-4](#)).

*Equation 6-4. AdaBoost predictions*

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{j=1}^N \alpha_j \quad \text{where } N \text{ is the number of predictors}$$
$$\hat{y}_j(\mathbf{x}) = k$$

Scikit-Learn uses a multiclass version of AdaBoost called [SAMME](#)<sup>16</sup> (which stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*). When there are just two classes, SAMME is equivalent to AdaBoost.

The following code trains an AdaBoost classifier based on 30 *decision stumps* using Scikit-Learn's `AdaBoostClassifier` class (as you might expect, there is also an `AdaBoostRegressor` class). A decision stump is a decision tree with `max_depth=1`—in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

```
from sklearn.ensemble import AdaBoostClassifier  
  
ada_clf = AdaBoostClassifier(  
    DecisionTreeClassifier(max_depth=1), n_estimators=30,  
    learning_rate=0.5, random_state=42, algorithm="SAMME")  
ada_clf.fit(X_train, y_train)
```

### TIP

If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base estimator.

## Gradient Boosting

Another very popular boosting algorithm is [gradient boosting](#).<sup>17</sup> Just like AdaBoost, gradient boosting works by sequentially adding predictors to an ensemble, each one

correcting its predecessor. However, instead of tweaking the instance weights at every iteration like AdaBoost does, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

Let's go through a simple regression example, using decision trees as the base predictors; this is called *gradient tree boosting*, or *gradient boosted regression trees* (GBRT).

First, let's generate a noisy quadratic dataset and fit a `DecisionTreeRegressor` to it:

```
import numpy as np
from sklearn.tree import DecisionTreeRegressor

m = 100 # number of instances
rng = np.random.default_rng(seed=42)
X = rng.random((m, 1)) - 0.5
noise = 0.05 * rng.standard_normal(m)
y = 3 * X[:, 0] ** 2 + noise # y = 3x2 + Gaussian noise

tree_reg1 = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg1.fit(X, y)
```

Next, we'll train a second `DecisionTreeRegressor` on the residual errors made by the first predictor:

```
y2 = y - tree_reg1.predict(X)
tree_reg2 = DecisionTreeRegressor(max_depth=2, random_state=43)
tree_reg2.fit(X, y2)
```

And then we'll train a third regressor on the residual errors made by the second predictor:

```
y3 = y2 - tree_reg2.predict(X)
tree_reg3 = DecisionTreeRegressor(max_depth=2, random_state=44)
tree_reg3.fit(X, y3)
```

Now we have an ensemble containing three trees. It can make predictions on a new instance simply by adding up the predictions of all the trees:

```
>>> X_new = np.array([[-0.4], [0.], [0.5]])
>>> sum(tree.predict(X_new) for tree in (tree_reg1, tree_reg2, tree_reg3))
array([0.57356534, 0.0405142 , 0.66914249])
```

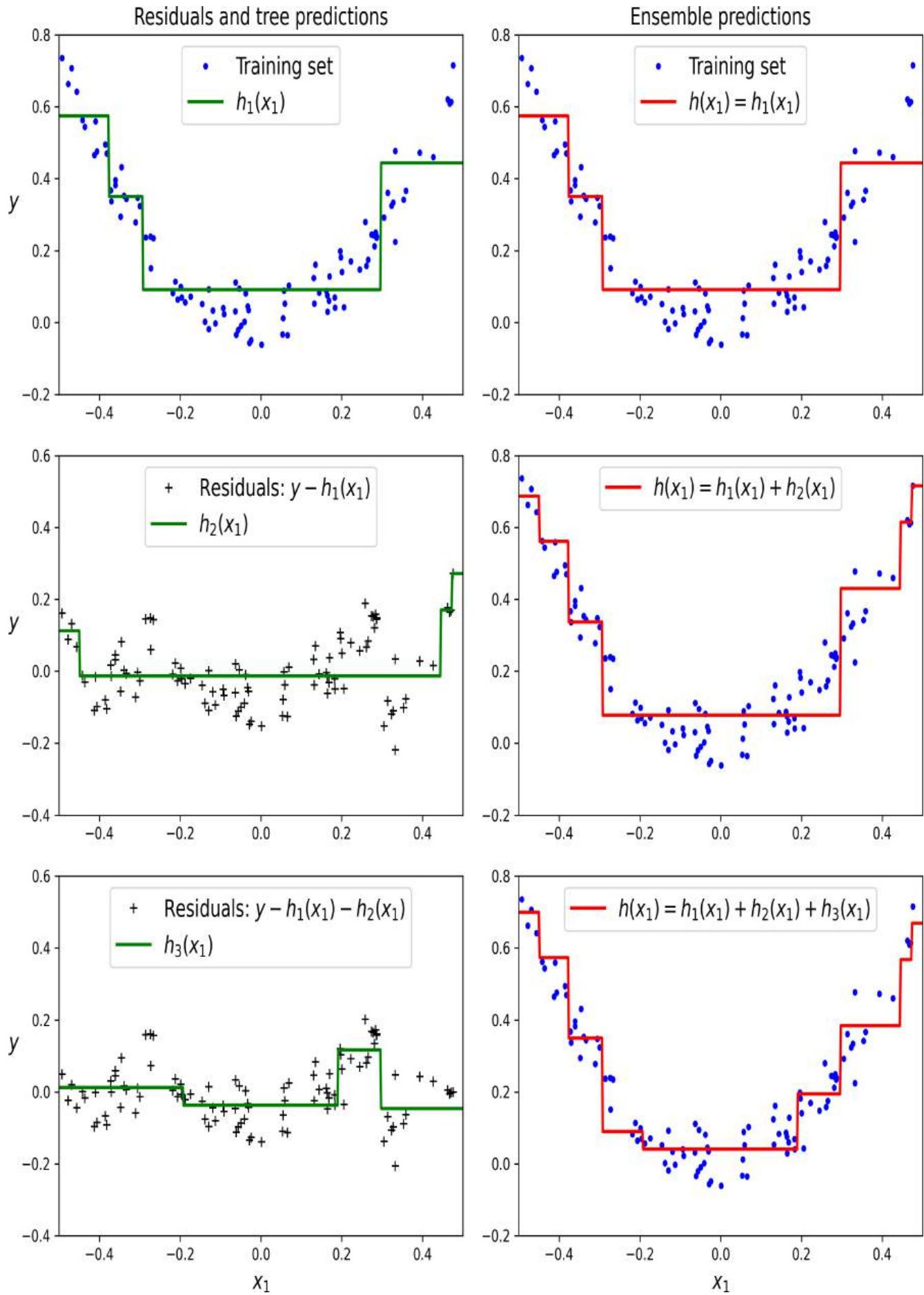
Figure 6-9 represents the predictions of these three trees in the left column, and the ensemble's predictions in the right column. In the first row, the ensemble has just one tree, so its predictions are exactly the same as the first tree's predictions. In the second row, a

new tree is trained on the residual errors of the first tree. On the right you can see that the ensemble's predictions are equal to the sum of the predictions of the first two trees. Similarly, in the third row another tree is trained on the residual errors of the second tree. You can see that the ensemble's predictions gradually get better as trees are added to the ensemble.

You can use Scikit-Learn's `GradientBoostingRegressor` class to train GBRT ensembles more easily (there's also a `GradientBoostingClassifier` class for classification). Much like the `RandomForestRegressor` class, it has hyperparameters to control the growth of decision trees (e.g., `max_depth`, `min_samples_leaf`), as well as hyperparameters to control the ensemble training, such as the number of trees (`n_estimators`). The following code creates the same ensemble as the previous one:

```
from sklearn.ensemble import GradientBoostingRegressor

gbrt = GradientBoostingRegressor(max_depth=2, n_estimators=3,
                                  learning_rate=1.0, random_state=42)
gbrt.fit(X, y)
```



*Figure 6-9. In this depiction of gradient boosting, the first predictor (top left) is trained normally, then each consecutive predictor (middle left and lower left) is trained on the previous predictor's residuals; the right column shows the resulting ensemble's predictions*

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as `0.05`, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called *shrinkage*. [Figure 6-10](#) shows two GBRT ensembles trained with different hyperparameters: the one on the left does not have enough trees to fit the training set, while the one on the right has about the right amount. If we added more trees, the GBRT would start to overfit the training set. As usual, you can use cross-validation to find the optimal learning rate, using `GridSearchCV` or `RandomizedSearchCV`.

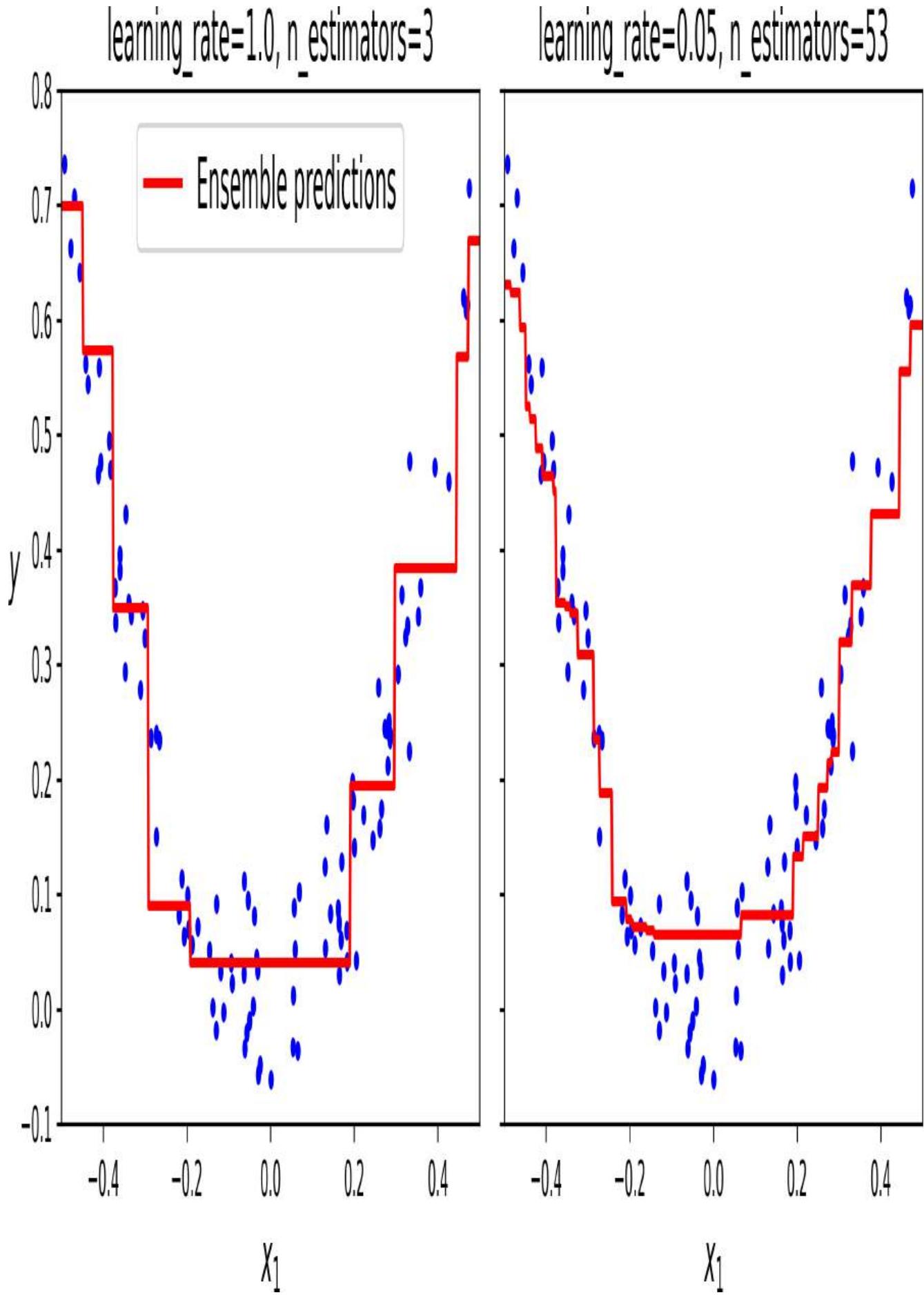


Figure 6-10. GBRT ensembles with not enough predictors (left) and just enough (right)

To find the optimal number of trees, you could also perform cross-validation, but there's a simpler way: if you set the `n_iter_no_change` hyperparameter to an integer value, say 10, then the `GradientBoostingRegressor` will automatically stop adding more trees during training if it sees that the last 10 trees didn't help. This is simply early stopping (introduced in [Chapter 4](#)), but with a little bit of patience: it tolerates having no progress for a few iterations before it stops. Let's train the ensemble using early stopping:

```
gbrt_best = GradientBoostingRegressor(  
    max_depth=2, learning_rate=0.05, n_estimators=500,  
    n_iter_no_change=10, random_state=42)  
gbrt_best.fit(X, y)
```

If you set `n_iter_no_change` too low, training may stop too early and the model will underfit. But if you set it too high, it will overfit instead. We also set a fairly small learning rate and a high number of estimators, but the actual number of estimators in the trained ensemble is much lower, thanks to early stopping:

```
>>> gbrt_best.n_estimators_  
53
```

When `n_iter_no_change` is set, the `fit()` method automatically splits the training set into a smaller training set and a validation set: this allows it to evaluate the model's performance each time it adds a new tree. The size of the validation set is controlled by the `validation_fraction` hyperparameter, which is 10% by default. The `tol` hyperparameter determines the maximum performance improvement that still counts as negligible. It defaults to 0.0001.

The `GradientBoostingRegressor` class also supports a `subsample` hyperparameter, which specifies the fraction of training instances to be used for training each tree. For example, if `subsample=0.25`, then each tree is trained on 25% of the training instances, selected randomly. As you can probably guess by now, this technique trades a higher bias for a lower variance. It also speeds up training considerably. This is called *stochastic gradient boosting*.

## Histogram-Based Gradient Boosting

Scikit-Learn also provides another GBRT implementation, optimized for large datasets: *histogram-based gradient boosting* (HGB). It works by binning the input features, replacing them with integers. The number of bins is controlled by the `max_bins`

hyperparameter, which defaults to 255 and cannot be set any higher than this. Binning can greatly reduce the number of possible thresholds that the training algorithm needs to evaluate. Moreover, working with integers makes it possible to use faster and more memory-efficient data structures. And the way the bins are built removes the need for sorting the features when training each tree.

As a result, this implementation has a computational complexity of  $O(b \times m)$  instead of  $O(n \times m \times \log(m))$ , where  $b$  is the number of bins,  $m$  is the number of training instances, and  $n$  is the number of features. In practice, this means that HGB can train hundreds of times faster than regular GBRT on large datasets. However, binning causes a precision loss, which acts as a regularizer: depending on the dataset, this may help reduce overfitting, or it may cause underfitting.

Scikit-Learn provides two classes for HGB: `HistGradientBoostingRegressor` and `HistGradientBoostingClassifier`. They're similar to `GradientBoostingRegressor` and `GradientBoostingClassifier`, with a few notable differences:

- Early stopping is automatically activated if the number of instances is greater than 10,000. You can turn early stopping always on or always off by setting the `early_stopping` hyperparameter to `True` or `False`.
- Subsampling is not supported.
- `n_estimators` is renamed to `max_iter`.
- The only decision tree hyperparameters that can be tweaked are `max_leaf_nodes`, `min_samples_leaf`, `max_depth`, and `max_features`.

The HGB classes support missing values natively, as well as categorical features. This simplifies preprocessing quite a bit. However, the categorical features must be represented as integers ranging from 0 to a number lower than `max_bins`. You can use an `OrdinalEncoder` for this. For example, here's how to build and train a complete pipeline for the California housing dataset introduced in [Chapter 2](#):

```
from sklearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.ensemble import HistGradientBoostingRegressor
from sklearn.preprocessing import OrdinalEncoder

hgb_reg = make_pipeline(
    make_column_transformer((OrdinalEncoder(), ["ocean_proximity"]),
                           remainder="passthrough",
                           force_int_remainder_cols=False),
    HistGradientBoostingRegressor(categorical_features=[0], random_state=42)
```

```
)  
hgb_reg.fit(housing, housing_labels)
```

The whole pipeline is almost as short as the imports! No need for an imputer, scaler, or a one-hot encoder, it's really convenient. Note that `categorical_features` must be set to the categorical column indices (or a Boolean array). Without any hyperparameter tuning, this model yields an RMSE of about 47,600, which is not too bad.

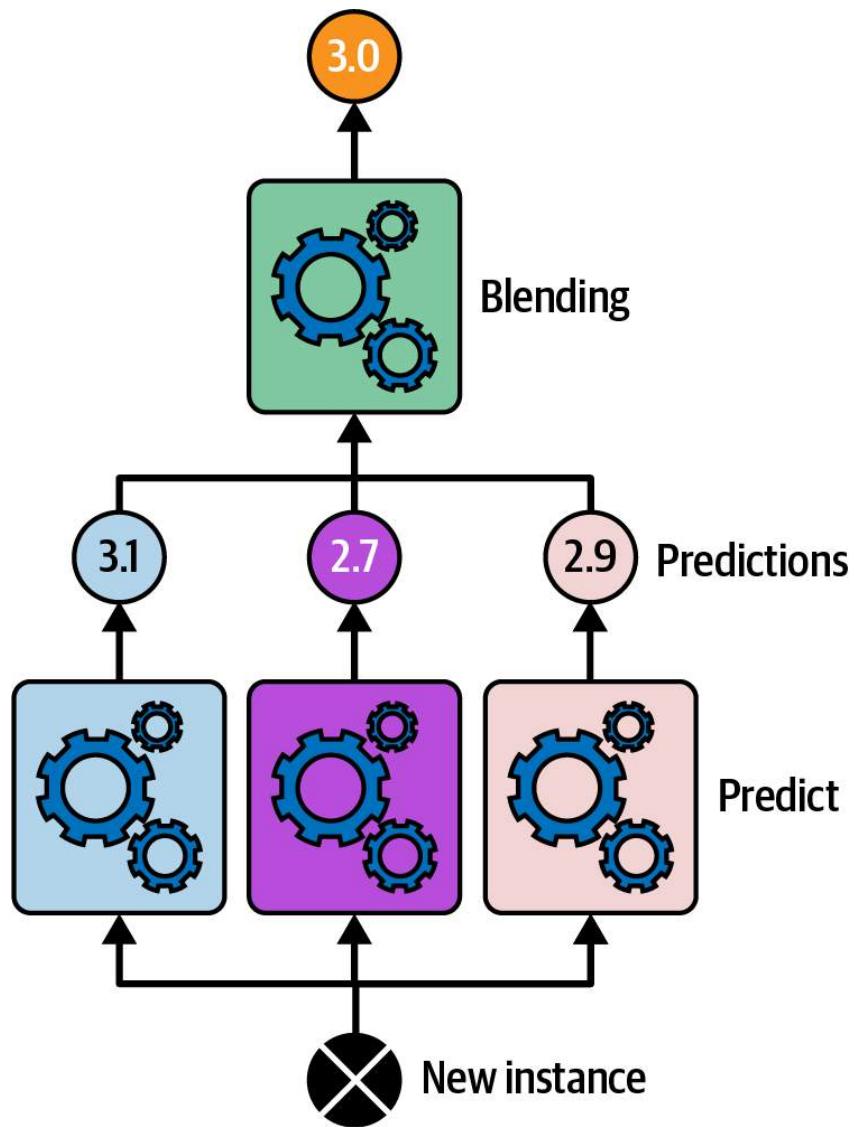
In short, HGB is a great choice when you have a fairly large dataset, especially when it contains categorical features and missing values: it performs well, doesn't require much preprocessing work, and training is fast. However, it can be a bit less accurate than GBRT, due to the binning, so you might want to try both.

### TIP

Several other optimized implementations of gradient boosting are available in the Python ML ecosystem: in particular, [XGBoost](#), [CatBoost](#), and [LightGBM](#). These libraries have been around for several years. They are all specialized for gradient boosting, their APIs are very similar to Scikit-Learn's, and they provide many additional features, including hardware acceleration using GPUs; you should definitely check them out! Moreover, [Yggdrasil Decision Forests \(YDF\)](#) provides optimized implementations of a variety of random forest algorithms.

## Stacking

The last ensemble method we will discuss in this chapter is called *stacking* (short for *stacked generalization*).<sup>18</sup> It is based on a simple idea: instead of using trivial functions (such as hard voting) to aggregate the predictions of all predictors in an ensemble, why don't we train a model to perform this aggregation? [Figure 6-11](#) shows such an ensemble performing a regression task on a new instance. Each of the bottom three predictors predicts a different value (3.1, 2.7, and 2.9), and then the final predictor (called a *blender*, or a *meta learner*) takes these predictions as inputs and makes the final prediction (3.0).



*Figure 6-11. Aggregating predictions using a blending predictor*

To train the blender, you first need to build the blending training set. You can use `cross_val_predict()` on every predictor in the ensemble to get out-of-sample predictions for each instance in the original training set (Figure 6-12), and use these as the input features to train the blender; and the targets can simply be copied from the original training set. Note that regardless of the number of features in the original training set (just one in this example), the blending training set will contain one input feature per predictor (three in this example). Once the blender is trained, the base predictors must be retrained one last time on the full original training set (since `cross_val_predict()` does not give access to the trained estimators).

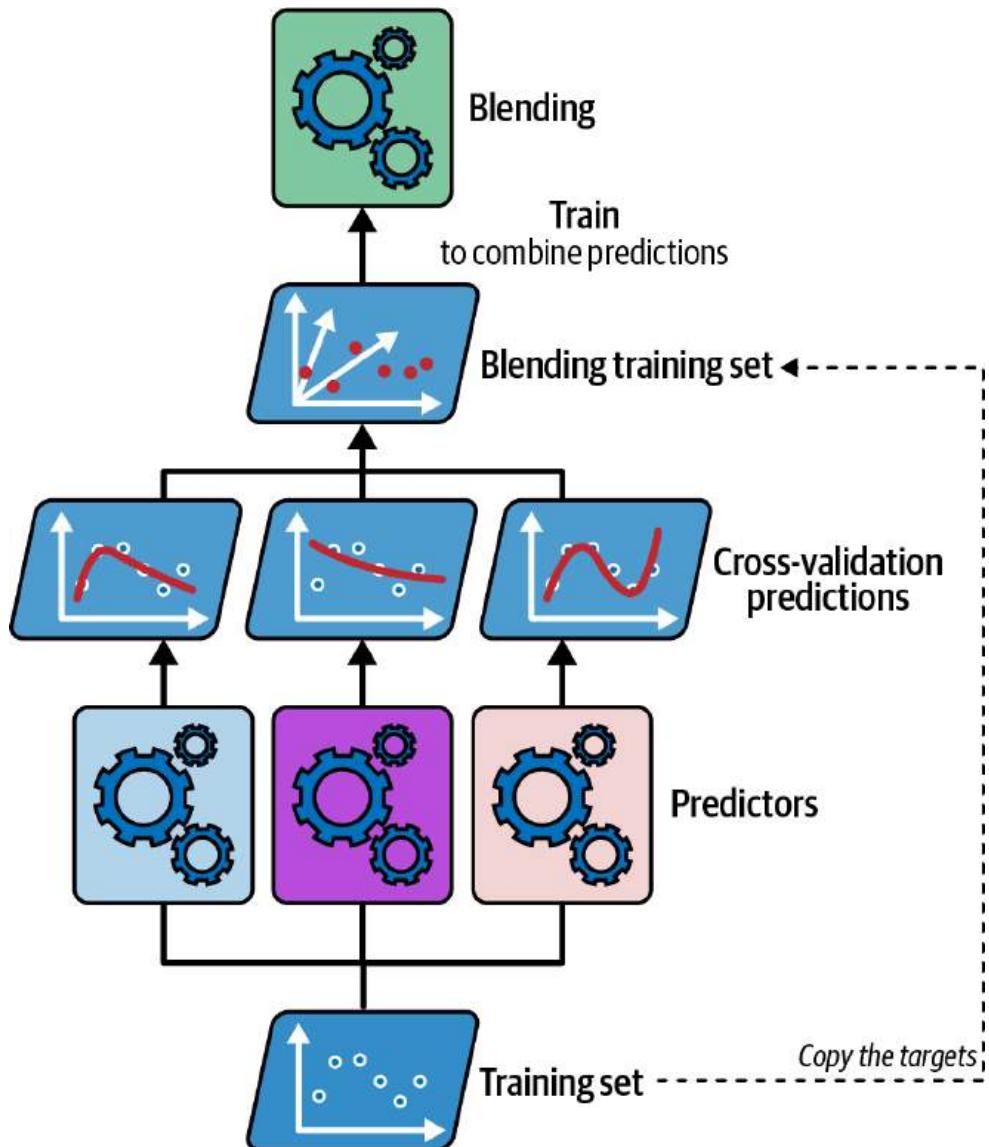


Figure 6-12. Training the blender in a stacking ensemble

It is actually possible to train several different blenders this way (e.g., one using linear regression, another using random forest regression) to get a whole layer of blenders, and then add another blender on top of that to produce the final prediction, as shown in [Figure 6-13](#). You may be able to squeeze out a few more drops of performance by doing this, but it will cost you in both training time and system complexity.

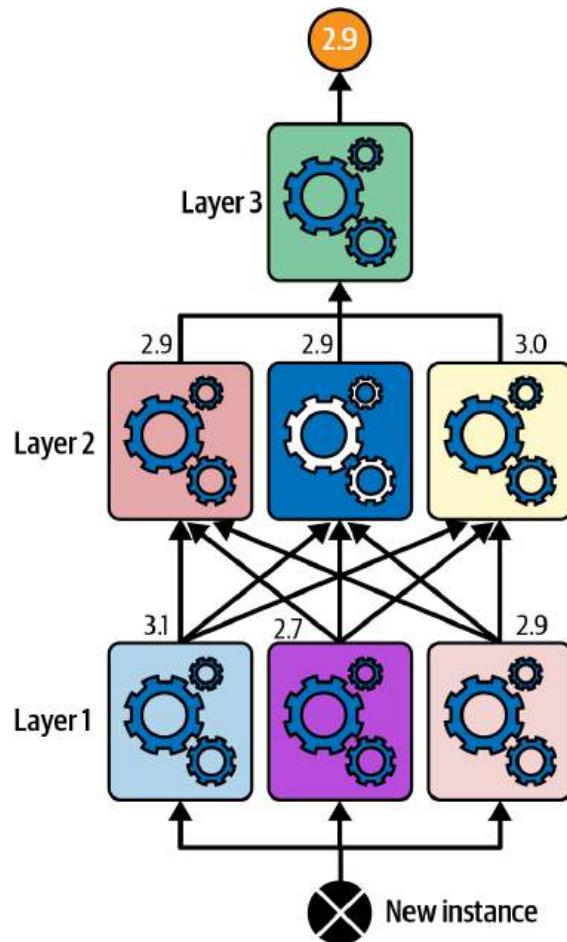


Figure 6-13. Predictions in a multilayer stacking ensemble

Scikit-Learn provides two classes for stacking ensembles: `StackingClassifier` and `StackingRegressor`. For example, we can replace the `VotingClassifier` we used at the beginning of this chapter on the moons dataset with a `StackingClassifier`:

```
from sklearn.ensemble import StackingClassifier

stacking_clf = StackingClassifier(
    estimators=[
        ('lr', LogisticRegression(random_state=42)),
        ('rf', RandomForestClassifier(random_state=42)),
        ('svc', SVC(probability=True, random_state=42))
    ],
    final_estimator=RandomForestClassifier(random_state=43),
    cv=5 # number of cross-validation folds
)
stacking_clf.fit(X_train, y_train)
```

For each predictor, the stacking classifier will call `predict_proba()` if available; if not it will fall back to `decision_function()` or, as a last resort, call `predict()`. If you don't provide a final estimator, `StackingClassifier` will use `LogisticRegression` and `StackingRegressor` will use `RidgeCV`.

If you evaluate this stacking model on the test set, you will find 92.8% accuracy, which is a bit better than the voting classifier using soft voting, which got 92%. Depending on your use case, this may or may not be worth the extra complexity and computational cost (since there's an extra model to run after all the others).

In conclusion, ensemble methods are versatile, powerful, and fairly simple to use. They can overfit if you're not careful, but that's true of every powerful model. **Table 6-1** summarizes all the techniques we discussed in this chapter, and when to use each one.

*Table 6-1. When to use each ensemble learning method*

<b>Ensemble method</b>	<b>When to use it</b>	<b>Example use cases</b>
Hard Voting	Balanced classification dataset with multiple strong but diverse classifiers.	Spam detection, sentiment analysis, disease classification
Soft Voting	Classification dataset with probabilistic models, where confidence scores matter.	Medical diagnosis, credit risk analysis, fake news detection
Bagging	Structured or semi-structured dataset with high variance and overfitting-prone models.	Financial risk modeling, e-commerce recommendation
Pasting	Structured or semi-structured dataset where more independent models are needed	Customer segmentation, protein classification
Random Forest	High-dimensional structured datasets with potentially noisy features.	Customer churn prediction, genetic data analysis, fraud detection
Extra-Trees	Large structured datasets with many features, where speed is critical and reducing variance is important.	Real-time fraud detection, sensor data analysis
AdaBoost	Small to medium-sized, low-noise, structured datasets with weak learners (e.g., decision stumps), where interpretability is helpful.	Credit scoring, anomaly detection, predictive maintenance
Gradient Boosting	Medium to large structured datasets where high predictive power is required, even at the cost of extra tuning.	Housing price prediction, risk assessment, demand forecasting

Ensemble method	When to use it	Example use cases
Histogram-Based Gradient Boosting (HGB)	Large structured datasets where training speed and scalability are key.	Click-through rate prediction, ranking algorithms, real-time bidding in advertising
Stacking	Complex, high-dimensional datasets where combining multiple diverse models can maximize accuracy.	Recommendation engines, autonomous vehicle decision-making, Kaggle competitions

### TIP

Random forests, AdaBoost, GBRT, and HGB are among the first models you should test for most machine learning tasks, particularly with heterogeneous tabular data. Moreover, as they require very little preprocessing, they're great for getting a prototype up and running quickly.

So far, we have looked only at supervised learning techniques. In the next chapter, we will turn to the most common unsupervised learning task: dimensionality reduction.

## Exercises

1. If you have trained five different models on the exact same training data, and they all achieve 95% precision, is there any chance that you can combine these models to get better results? If so, how? If not, why?
2. What is the difference between hard and soft voting classifiers?
3. Is it possible to speed up training of a bagging ensemble by distributing it across multiple servers? What about pasting ensembles, boosting ensembles, random forests, or stacking ensembles?
4. What is the benefit of out-of-bag evaluation?

5. What makes extra-trees ensembles more random than regular random forests? How can this extra randomness help? Are extra-trees classifiers slower or faster than regular random forests?
6. If your AdaBoost ensemble underfits the training data, which hyperparameters should you tweak, and how?
7. If your gradient boosting ensemble overfits the training set, should you increase or decrease the learning rate?
8. Load the MNIST dataset (introduced in [Chapter 3](#)), and split it into a training set, a validation set, and a test set (e.g., use 50,000 instances for training, 10,000 for validation, and 10,000 for testing). Then train various classifiers, such as a random forest classifier, an extra-trees classifier, and an SVM classifier. Next, try to combine them into an ensemble that outperforms each individual classifier on the validation set, using soft or hard voting. Once you have found one, try it on the test set. How much better does it perform compared to the individual classifiers?
9. Run the individual classifiers from the previous exercise to make predictions on the validation set, and create a new training set with the resulting predictions: each training instance is a vector containing the set of predictions from all your classifiers for an image, and the target is the image's class. Train a classifier on this new training set. Congratulations—you have just trained a blender, and together with the classifiers it forms a stacking ensemble! Now evaluate the ensemble on the test set. For each image in the test set, make predictions with all your classifiers, then feed the predictions to the blender to get the ensemble's predictions. How does it compare to the voting classifier you trained earlier? Now try again using a `StackingClassifier` instead. Do you get better performance? If so, why?

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.

---

<sup>1</sup> Imagine picking a card randomly from a deck of cards, writing it down, then placing it back in the deck before picking the next card: the same card could be sampled multiple times.

<sup>2</sup> Leo Breiman, “Bagging Predictors”, *Machine Learning* 24, no. 2 (1996): 123–140.

<sup>3</sup> In statistics, resampling with replacement is called *bootstrapping*.

- <sup>4</sup> Leo Breiman, “Pasting Small Votes for Classification in Large Databases and On-Line”, *Machine Learning* 36, no. 1–2 (1999): 85–103.
- <sup>5</sup> Bias and variance were introduced in [Chapter 4](#). Recall that a high bias means that the average prediction is far off target, while a high variance means that the predictions are very scattered. We want both to be low.
- <sup>6</sup> `max_samples` can alternatively be set to a float between 0.0 and 1.0, in which case the max number of sampled instances is equal to the size of the training set times `max_samples`.
- <sup>7</sup> As  $m$  grows, this ratio approaches  $1 - \exp(-1) \approx 63\%$ .
- <sup>8</sup> Gilles Louppe and Pierre Geurts, “Ensembles on Random Patches”, *Lecture Notes in Computer Science* 7523 (2012): 346–361.
- <sup>9</sup> Tin Kam Ho, “The Random Subspace Method for Constructing Decision Forests”, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 20, no. 8 (1998): 832–844.
- <sup>10</sup> Tin Kam Ho, “Random Decision Forests”, *Proceedings of the Third International Conference on Document Analysis and Recognition* 1 (1995): 278.
- <sup>11</sup> The `BaggingClassifier` class remains useful if you want a bag of something other than decision trees.
- <sup>12</sup> Pierre Geurts et al., “Extremely Randomized Trees”, *Machine Learning* 63, no. 1 (2006): 3–42.
- <sup>13</sup> Yoav Freund and Robert E. Schapire, “A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting”, *Journal of Computer and System Sciences* 55, no. 1 (1997): 119–139.
- <sup>14</sup> This is just for illustrative purposes. SVMs are generally not good base predictors for AdaBoost; they are slow and tend to be unstable with it.
- <sup>15</sup> The original AdaBoost algorithm does not use a learning rate hyperparameter.
- <sup>16</sup> For more details, see Ji Zhu et al., “Multi-Class AdaBoost”, *Statistics and Its Interface* 2, no. 3 (2009): 349–360.
- <sup>17</sup> Gradient boosting was first introduced in Leo Breiman’s [1997 paper](#) “Arcing the Edge” and was further developed in the [1999 paper](#) “Greedy Function Approximation: A Gradient Boosting Machine” by Jerome H. Friedman.
- <sup>18</sup> David H. Wolpert, “Stacked Generalization”, *Neural Networks* 5, no. 2 (1992): 241–259.

# Chapter 7. Dimensionality Reduction

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 7th chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

Many machine learning problems involve thousands or even millions of features for each training instance. Not only do all these features make training extremely slow, but they can also make it much harder to find a good solution, as you will see. This problem is often referred to as the *curse of dimensionality*.

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images (introduced in [Chapter 3](#)): the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information. As we saw in the previous chapter, ([Figure 6-6](#)) confirms that these pixels are utterly unimportant for the classification task.

Additionally, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information, removing redundancy and sometimes even noise.

### WARNING

Reducing dimensionality can also drop some useful information, just like compressing an image to JPEG can degrade its quality: it can make your system perform slightly worse, especially if you reduce dimensionality too much. Moreover, some models—such as neural networks—can handle high-dimensional data efficiently and learn to reduce its dimensionality while preserving the useful information for the task at hand. In short, adding an extra preprocessing step for dimensionality reduction will not always help.

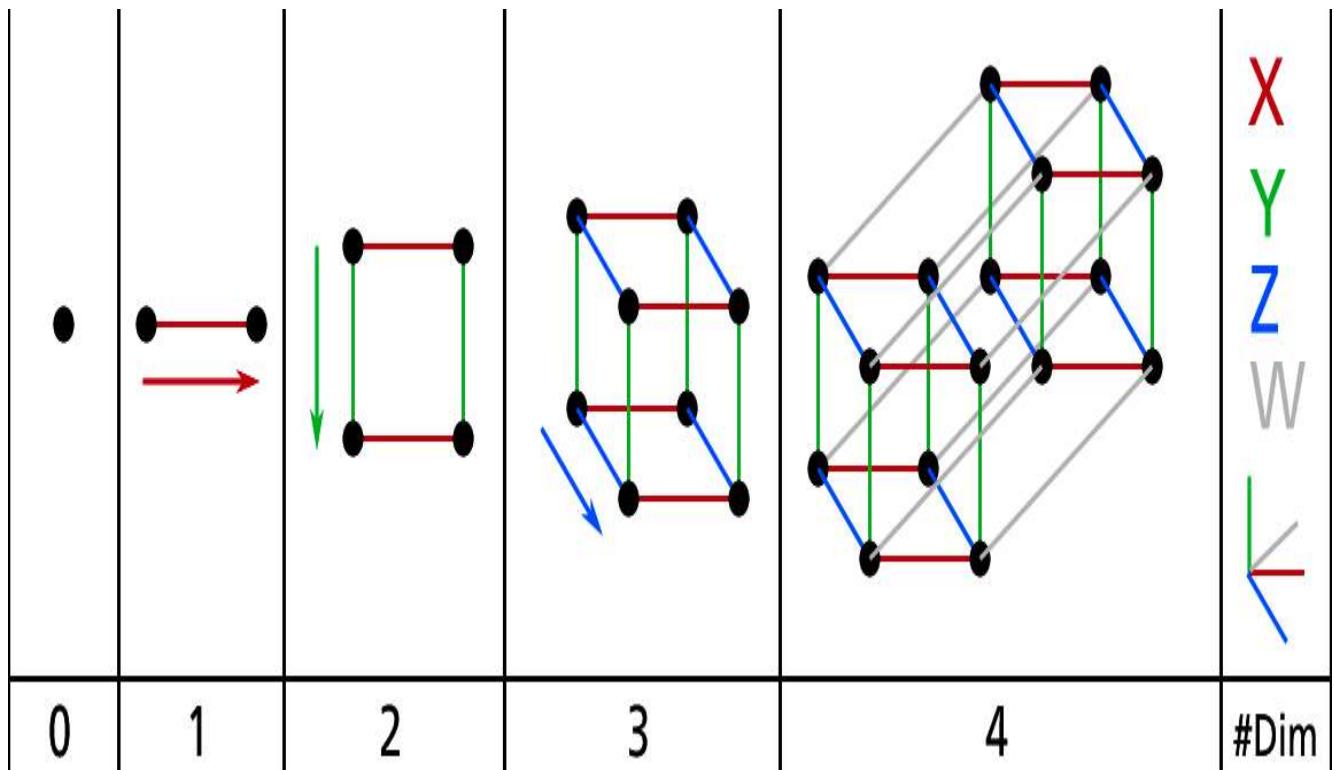
Apart from speeding up training and possibly improving your model’s performance, dimensionality reduction is also extremely useful for data visualization. Reducing the

number of dimensions down to two (or three) makes it possible to plot a condensed view of a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns, such as clusters. Moreover, data visualization is essential to communicate your conclusions to people who are not data scientists—in particular, decision makers who will use your results.

In this chapter we will first discuss the curse of dimensionality and get a sense of what goes on in high-dimensional space. Then we will consider the two main approaches to dimensionality reduction (projection and manifold learning), and we will go through three of the most popular dimensionality reduction techniques: PCA, random projection, and locally linear embedding (LLE).

## The Curse of Dimensionality

We are so used to living in three dimensions<sup>1</sup> that our intuition fails us when we try to imagine a high-dimensional space. Even a basic 4D hypercube is incredibly hard to picture in our minds (see [Figure 7-1](#)), let alone a 200-dimensional ellipsoid bent in a 1,000-dimensional space.



*Figure 7-1. Point, segment, square, cube, and tesseract (0D to 4D hypercubes)<sup>2</sup>*

It turns out that many things behave very differently in high-dimensional space. For example, if you pick a random point in a unit square (a  $1 \times 1$  square), it will have only

about a 0.4% chance of being located less than 0.001 from a border (in other words, it is very unlikely that a random point will be “extreme” along any dimension). But in a 10,000-dimensional unit hypercube, this probability is greater than 99.999999%. Most points in a high-dimensional hypercube are very close to the border.<sup>3</sup>

Here is a more troublesome difference: if you pick two points randomly in a unit square, the distance between these two points will be, on average, roughly 0.52. If you pick two random points in a 3D unit cube, the average distance will be roughly 0.66. But what about two points picked randomly in a 1,000,000-dimensional unit hypercube?

The average distance, believe it or not, will be about 408.25 (roughly  $\sqrt{\frac{1,000,000}{6}}$ )! This is counterintuitive: how can two points be so far apart when they both lie within the same unit hypercube? Well, there’s just plenty of space in high dimensions.

As a result, high-dimensional datasets are often very sparse: most training instances are likely to be far away from each other, so training methods based on distance or similarity (such as  $k$ -nearest neighbors) will be much less effective. And some types of models will not be usable at all because they scale poorly with the dataset’s dimensionality (e.g., SVMs or dense neural networks). And new instances will likely be far away from any training instance, making predictions much less reliable than in lower dimensions since they will be based on much larger extrapolations. Since patterns in the data will become harder to identify, models will tend to fit the noise more frequently than in lower dimensions; regularization will become all the more important. Lastly, models will become even harder to interpret.

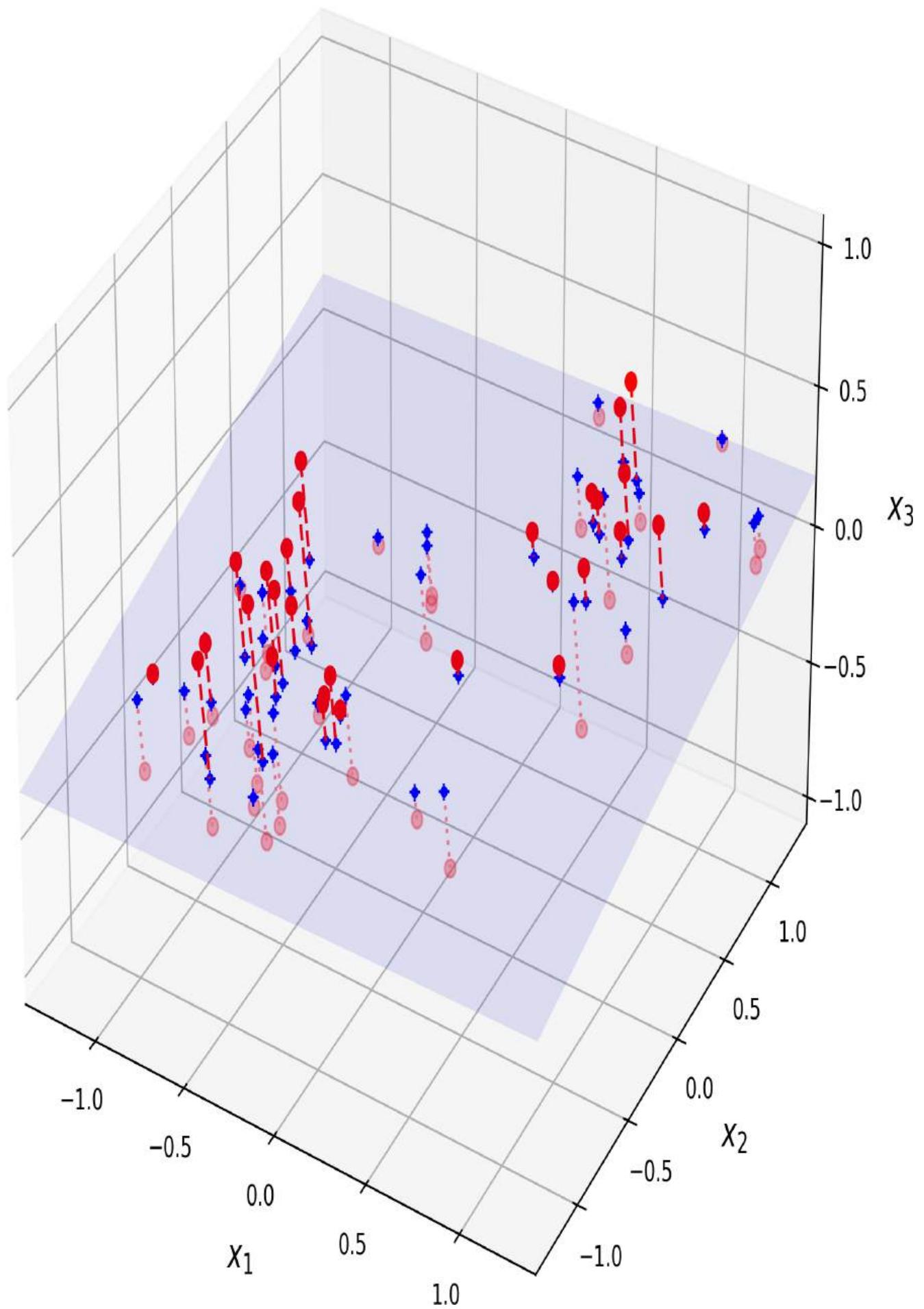
In theory, some of these issues can be resolved by increasing the size of the training set to reach a sufficient density of training instances. Unfortunately, in practice, the number of training instances required to reach a given density grows exponentially with the number of dimensions. With just 100 features—significantly fewer than in the MNIST problem—all ranging from 0 to 1, you would need more training instances than atoms in the observable universe in order for training instances to be within 0.1 of each other on average, assuming they were spread out uniformly across all dimensions.

## Main Approaches for Dimensionality Reduction

Before we dive into specific dimensionality reduction algorithms, let’s take a look at the two main approaches to reducing dimensionality: projection and manifold learning.

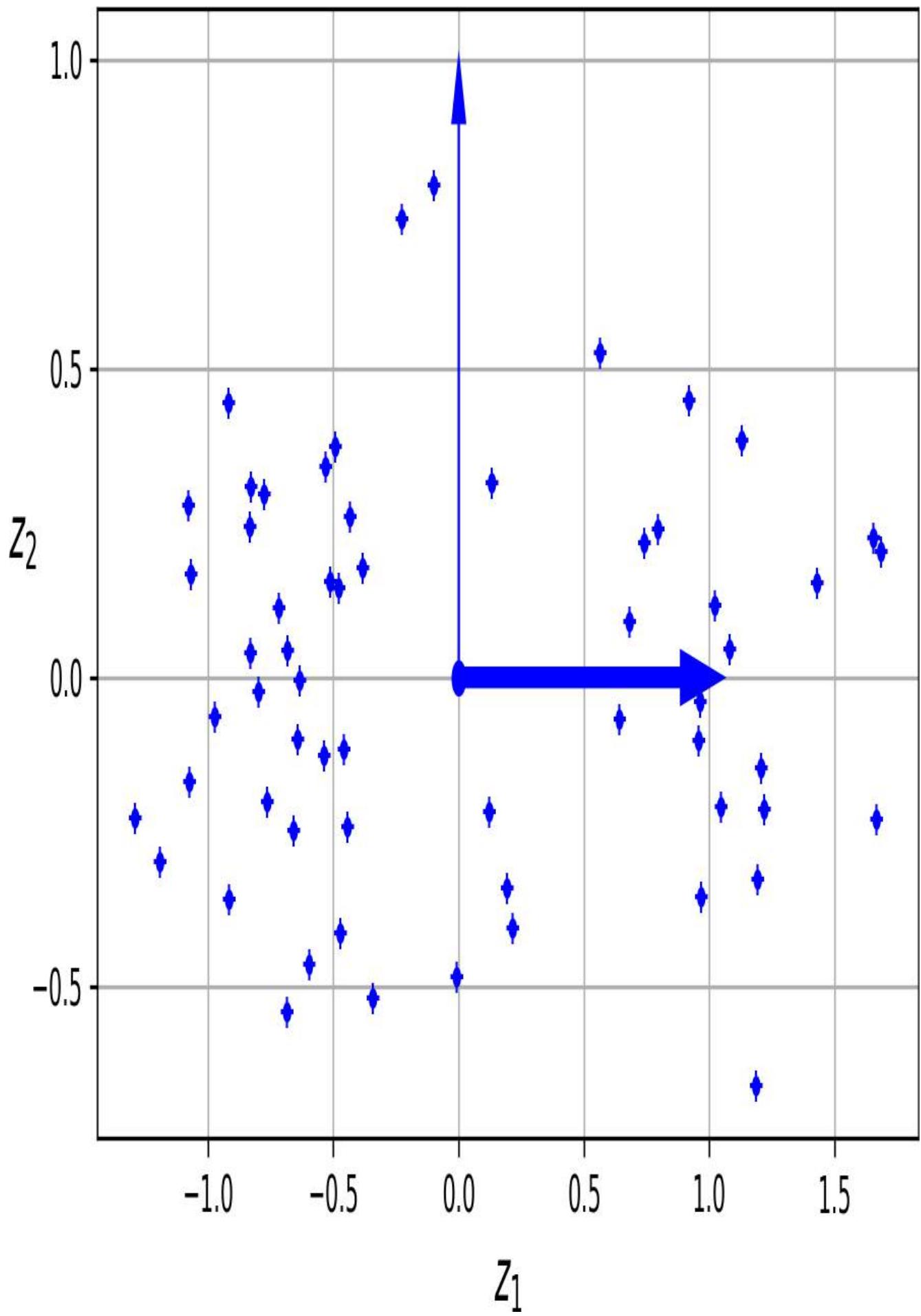
### Projection

In most real-world problems, training instances are *not* spread out uniformly across all dimensions. Many features are almost constant, while others are highly correlated (as discussed earlier for MNIST). As a result, all training instances lie within (or close to) a much lower-dimensional *subspace* of the high-dimensional space. This sounds very abstract, so let's look at an example. In [Figure 7-2](#) you can see a 3D dataset represented by small spheres.



*Figure 7-2. A 3D dataset lying close to a 2D subspace*

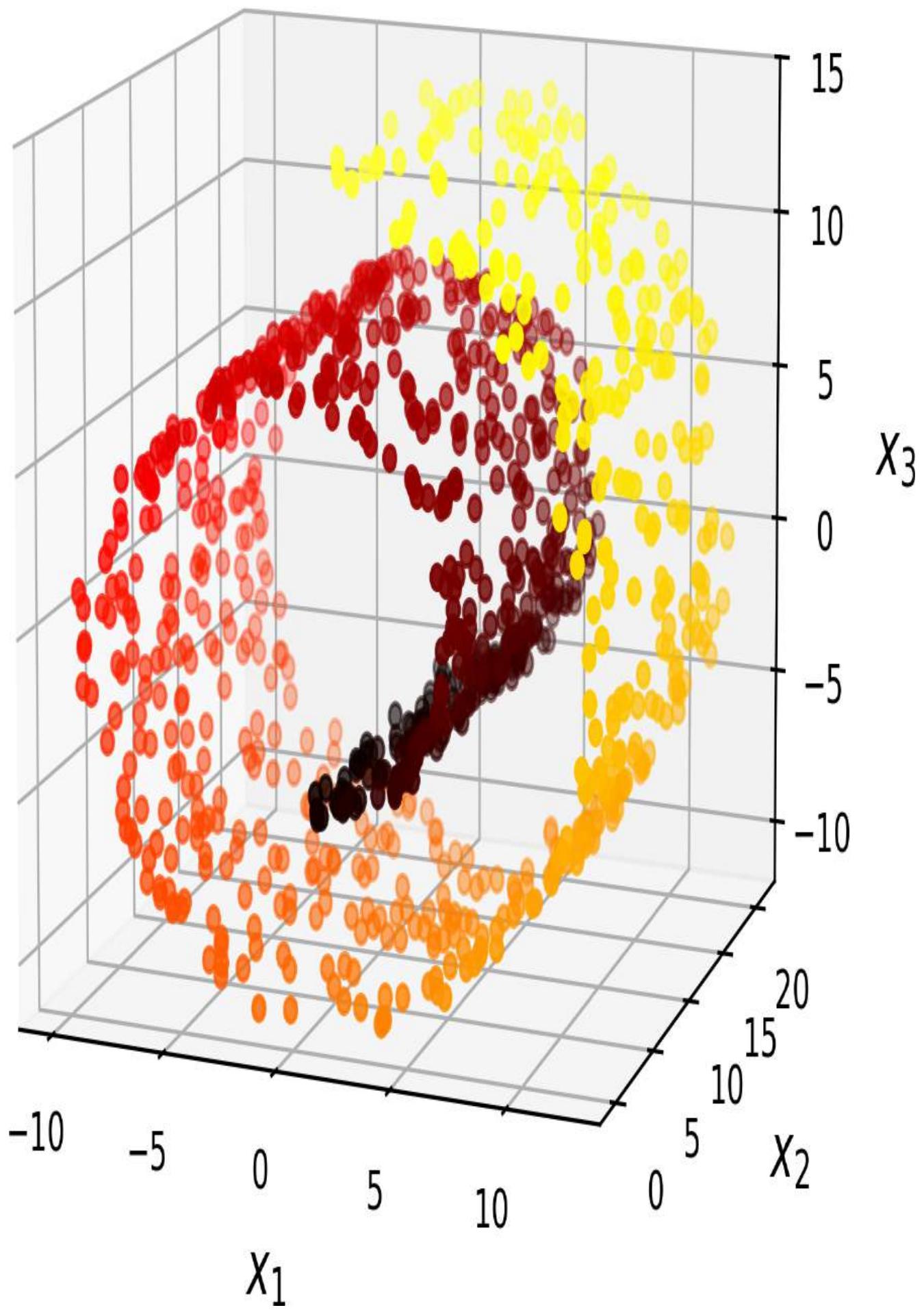
Notice that all training instances lie close to a plane: this is a lower-dimensional (2D) subspace of the higher-dimensional (3D) space. If we project every training instance perpendicularly onto this subspace (as represented by the short dashed lines connecting the instances to the plane), we get the new 2D dataset shown in [Figure 7-3](#). Ta-da! We have just reduced the dataset's dimensionality from 3D to 2D. Note that the axes correspond to new features  $z_1$  and  $z_2$ : they are the coordinates of the projections on the plane.



*Figure 7-3. The new 2D dataset after projection*

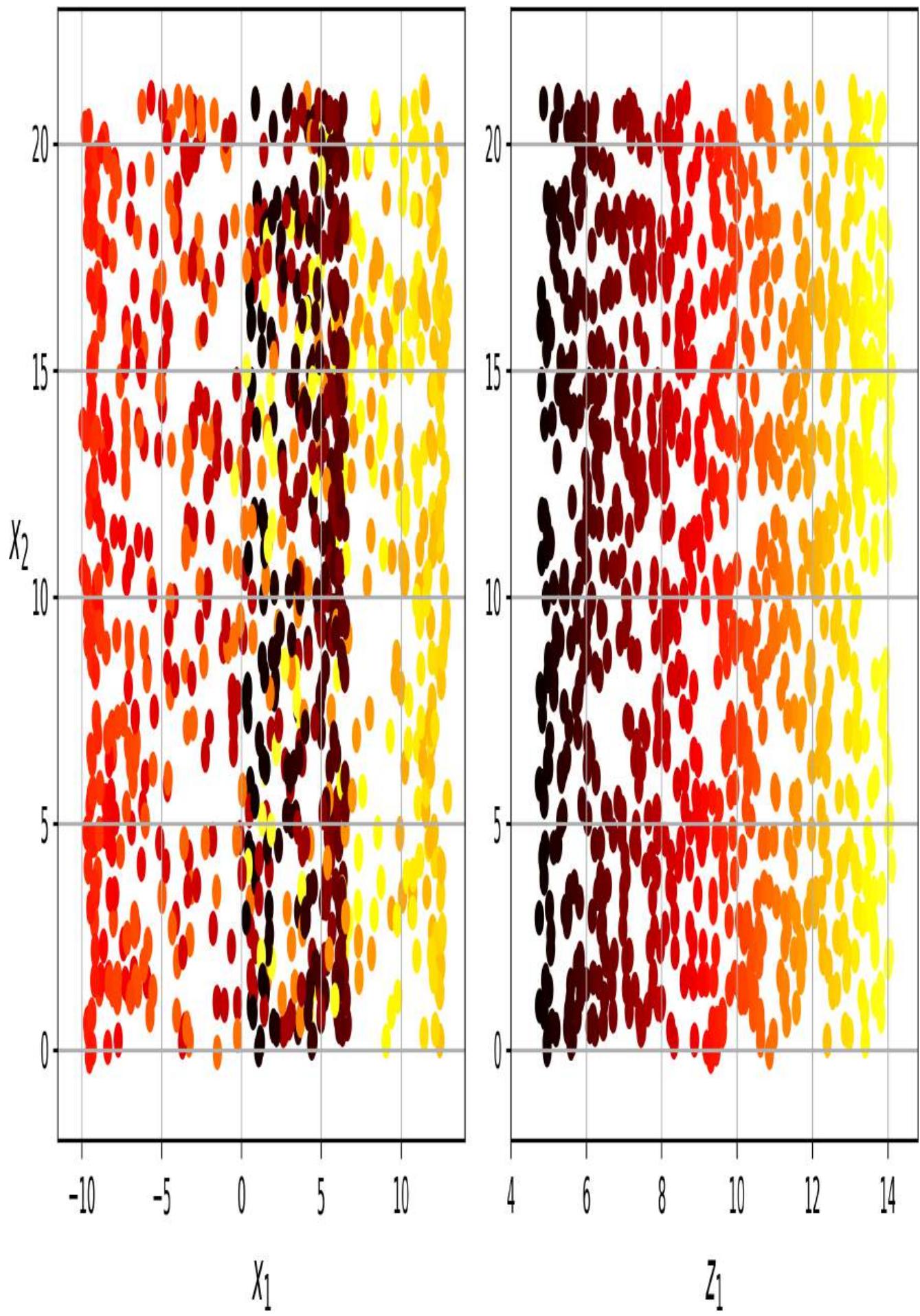
## Manifold Learning

Although projection is fast and often works well, it's not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the famous Swiss roll toy dataset represented in [Figure 7-4](#).



*Figure 7-4. Swiss roll dataset*

Simply projecting onto a plane (e.g., by dropping  $x_3$ ) would squash different layers of the Swiss roll together, as shown on the left side of [Figure 7-5](#). What you probably want instead is to unroll the Swiss roll to obtain the 2D dataset on the righthand side of [Figure 7-5](#).



*Figure 7-5. Squashing by projecting onto a plane (left) versus unrolling the Swiss roll (right)*

The Swiss roll is an example of a 2D *manifold*. Put simply, a 2D manifold is a 2D shape that can be bent and twisted in a higher-dimensional space. More generally, a  $d$ -dimensional manifold is a part of an  $n$ -dimensional space (where  $d < n$ ) that locally resembles a  $d$ -dimensional hyperplane. In the case of the Swiss roll,  $d = 2$  and  $n = 3$ : it locally resembles a 2D plane, but it is rolled in the third dimension.

Many dimensionality reduction algorithms (e.g., LLE, Isomap, t-SNE, or UMAP), work by modeling the manifold on which the training instances lie; this is called *manifold learning*. It relies on the *manifold assumption*, also called the *manifold hypothesis*, which holds that most real-world high-dimensional datasets lie close to a much lower-dimensional manifold. This assumption is very often empirically observed.

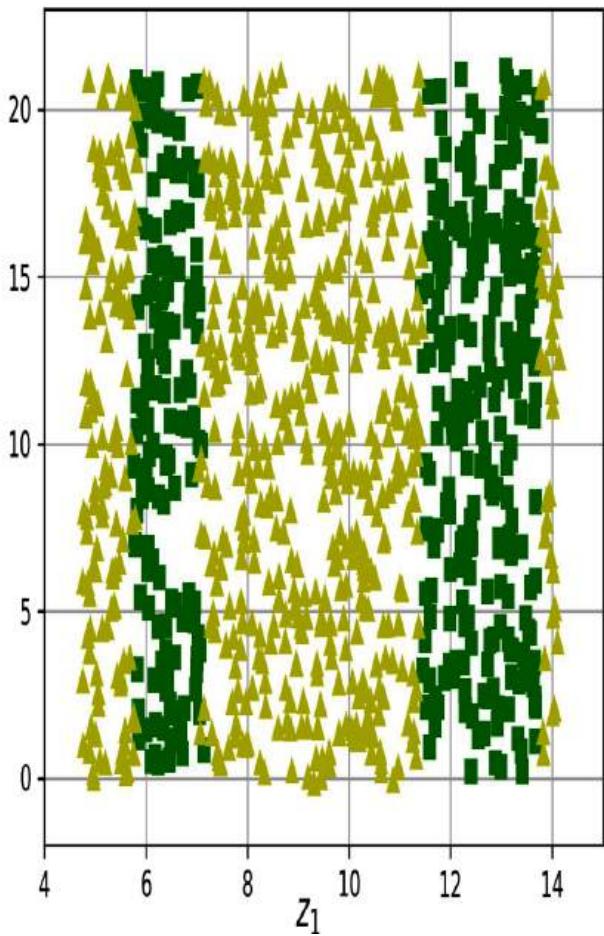
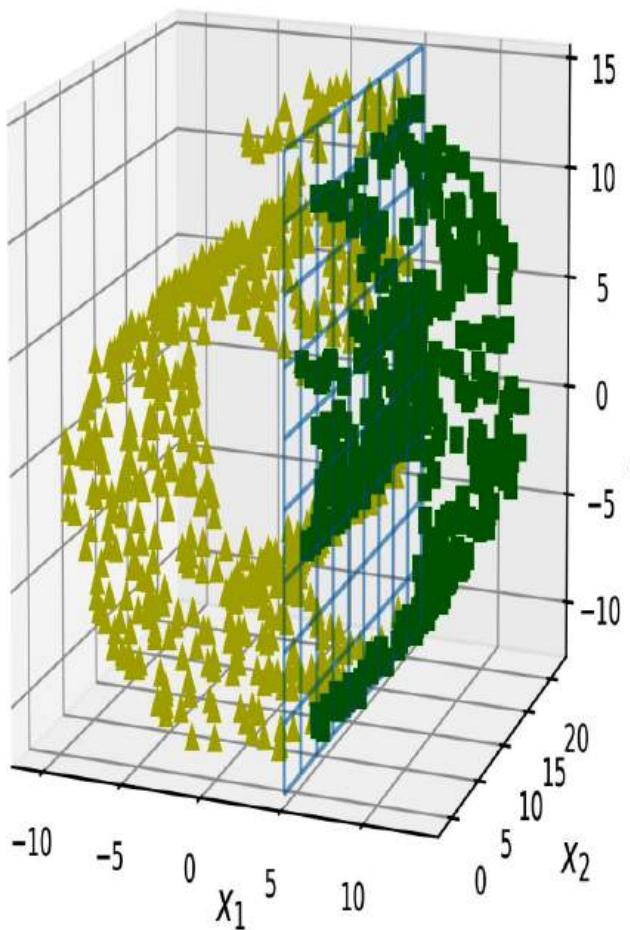
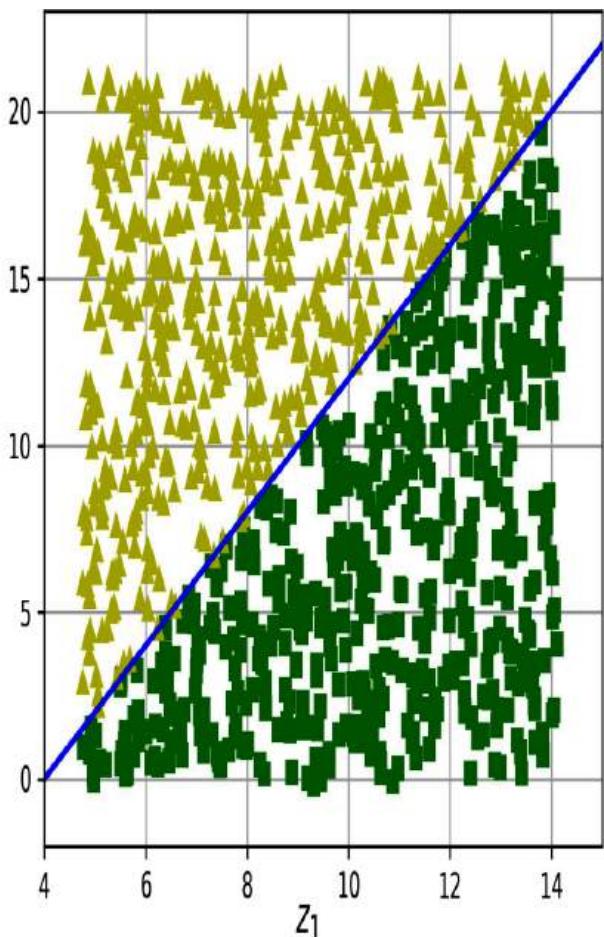
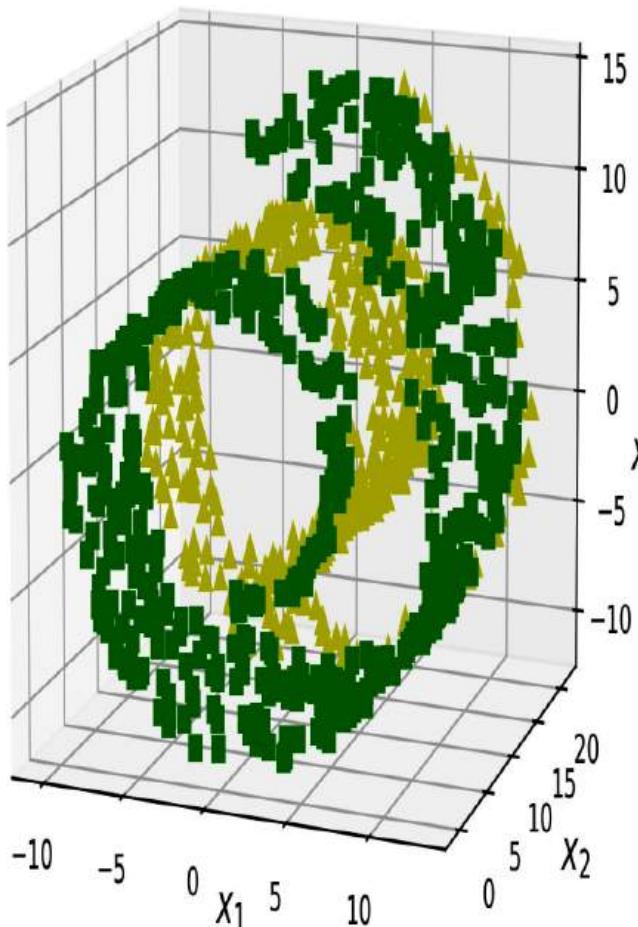
Once again, think about the MNIST dataset: all handwritten digit images have some similarities. They are made of connected lines, the borders are white, and they are more or less centered. If you randomly generated images, only a ridiculously tiny fraction of them would look like handwritten digits. In other words, the degrees of freedom available to you if you try to create a digit image are dramatically lower than the degrees of freedom you have if you are allowed to generate any image you want. These constraints tend to squeeze the dataset into a lower-dimensional manifold.

The manifold assumption is often accompanied by another implicit assumption: that the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of [Figure 7-6](#) the Swiss roll is split into two classes: in the 3D space (on the left) the decision boundary would be fairly complex, but in the 2D unrolled manifold space (on the right) the decision boundary is a straight line.

However, this implicit assumption does not always hold. For example, in the bottom row of [Figure 7-6](#), the decision boundary is located at  $x_1 = 5$ . This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

In short, reducing the dimensionality of your training set before training a model will usually speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset. Dimensionality reduction is typically more effective when the dataset is small relative to the number of features, especially if it's noisy, or many features are highly correlated to one another (i.e., redundant). And if you have domain knowledge about the process that generated the data, and you know it's simple, then the manifold assumption certainly holds, and dimensionality reduction is likely to help.

Hopefully you now have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds. The rest of this chapter will go through some of the most popular algorithms for dimensionality reduction.



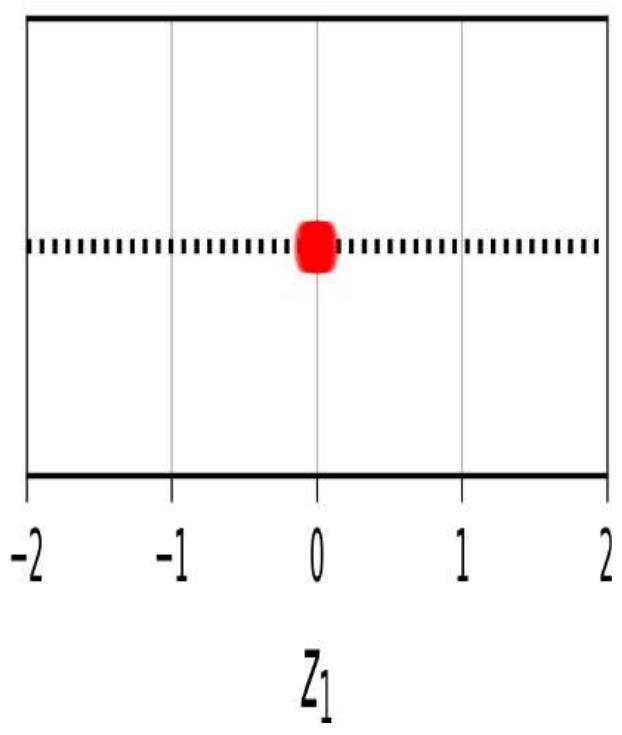
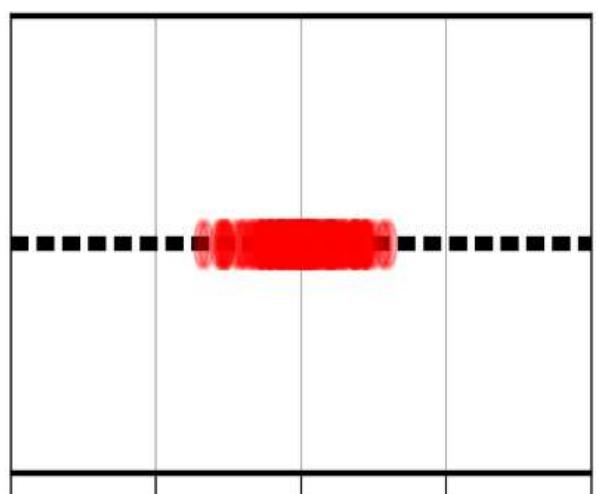
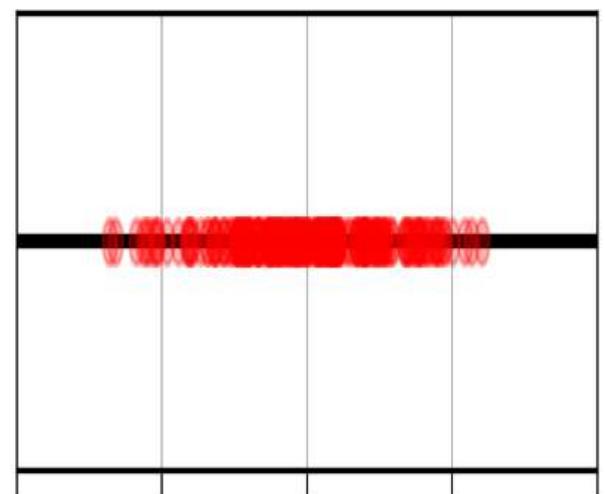
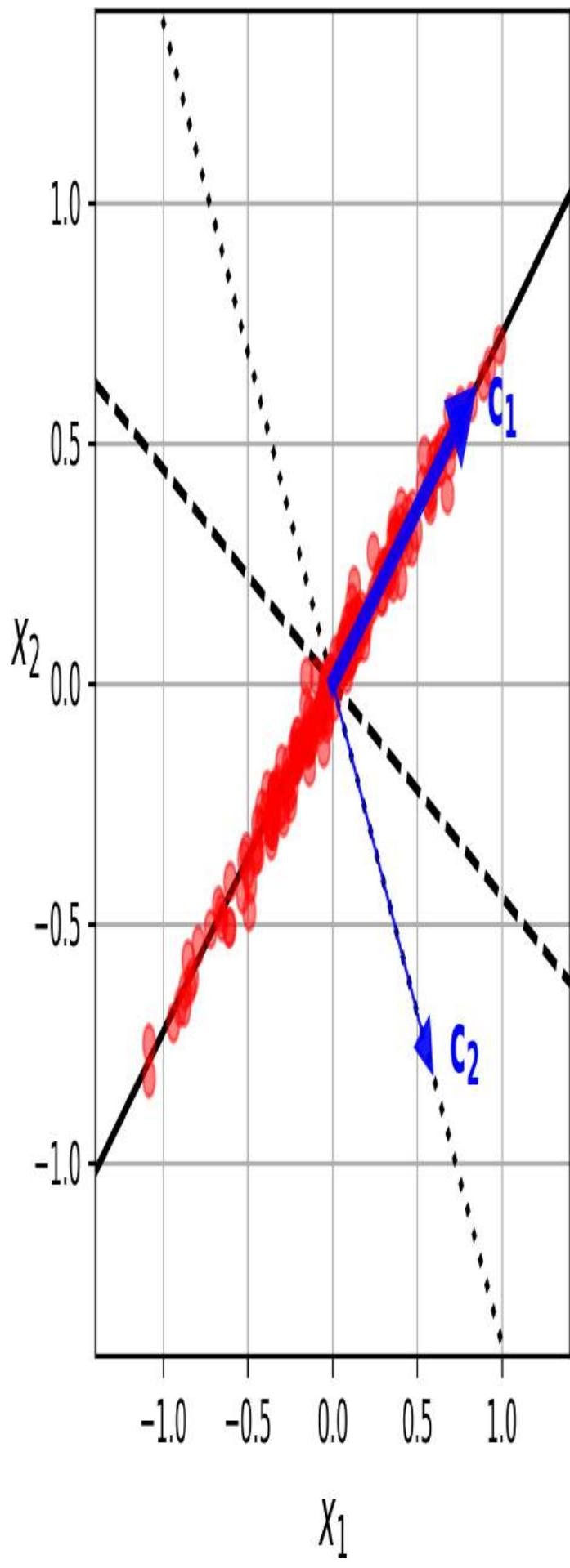
*Figure 7-6. The decision boundary may not always be simpler with lower dimensions*

## PCA

*Principal component analysis* (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it, just like in [Figure 7-2](#).

### Preserving the Variance

Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left in [Figure 7-7](#), along with three different axes (i.e., 1D hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the projection onto the solid line preserves the maximum variance (top), while the projection onto the dotted line preserves very little variance (bottom) and the projection onto the dashed line preserves an intermediate amount of variance (middle).



*Figure 7-7. Selecting the subspace on which to project*

It seems reasonable to select the axis that preserves the maximum amount of variance, as it will most likely lose less information than the other projections. Consider your shadow on the ground when the Sun is directly overhead: it's a small blob that doesn't look anything like you. But your shadow on a wall at sunrise is much larger and it *does* look like you. Another way to justify choosing the axis that maximizes the variance is that it is also the axis that minimizes the mean squared distance between the original dataset and its projection onto that axis. This is the rather simple idea behind PCA, introduced way back in 1901!<sup>4</sup>

## Principal Components

PCA identifies the axis that accounts for the largest amount of variance in the training set. In [Figure 7-7](#), it is the solid line. It also finds a second axis, orthogonal to the first one, that accounts for the largest amount of the remaining variance. In this 2D example there is no choice: it is the dotted line. If it were a higher-dimensional dataset, PCA would also find a third axis, orthogonal to both previous axes, and a fourth, a fifth, and so on—as many axes as the number of dimensions in the dataset.

The  $i^{\text{th}}$  axis is called the  $i^{\text{th}}$  *principal component* (PC) of the data. In [Figure 7-7](#), the first PC is the axis on which vector  $\mathbf{c}_1$  lies, and the second PC is the axis on which vector  $\mathbf{c}_2$  lies. In [Figure 7-2](#) the first two PCs are on the projection plane, and the third PC is the axis orthogonal to that plane. After the projection, in [Figure 7-3](#), the first PC corresponds to the  $z_1$  axis, and the second PC corresponds to the  $z_2$  axis.

### NOTE

For each principal component, PCA finds a zero-centered unit vector pointing along the direction of the PC. Unfortunately, its direction is not guaranteed: if you perturb the training set slightly and run PCA again, the unit vector may point in the opposite direction. In fact, a pair of unit vectors may even rotate or swap if the variances along these two axes are very close. So if you use PCA as a preprocessing step before a model, make sure you always retrain the model entirely every time you update the PCA transformer: if you don't and if the PCA's output doesn't align with the previous version, the model will be very confused.

So how can you find the principal components of a training set? Luckily, there is a standard matrix factorization technique called *singular value decomposition* (SVD) that can decompose the training set matrix  $\mathbf{X}$  into the product of three matrices  $\mathbf{U} \Sigma \mathbf{V}^T$ ,

where  $\mathbf{V}$  contains the unit vectors that define all the principal components that you are looking for, in the correct order, as shown in [Equation 7-1](#).<sup>5</sup>

*Equation 7-1. Principal components matrix*

$$\mathbf{V} = \begin{pmatrix} | & | & & | \\ \mathbf{c}_1 & \mathbf{c}_2 & \cdots & \mathbf{c}_n \\ | & | & & | \end{pmatrix}$$

The following Python code uses NumPy's `svd()` function to obtain all the principal components of the 3D training set represented in [Figure 7-2](#), then it extracts the two unit vectors that define the first two PCs:

```
import numpy as np

X = [...] # create a small 3D dataset
X_centered = X - X.mean(axis=0)
U, s, Vt = np.linalg.svd(X_centered)
c1 = Vt[0]
c2 = Vt[1]
```

### WARNING

PCA assumes that the dataset is centered around the origin. As you will see, Scikit-Learn's PCA classes take care of centering the data for you. If you implement PCA yourself (as in the preceding example), or if you use other libraries, don't forget to center the data first.

## Projecting Down to $d$ Dimensions

Once you have identified all the principal components, you can reduce the dimensionality of the dataset down to  $d$  dimensions by projecting it onto the hyperplane defined by the first  $d$  principal components (we will discuss how to choose the number of dimensions  $d$  shortly). Selecting this hyperplane ensures that the projection will preserve as much variance as possible. For example, in [Figure 7-2](#) the 3D dataset is projected down to the 2D plane defined by the first two principal components, preserving a large part of the dataset's variance. As a result, the 2D projection looks very much like the original 3D dataset.

To project the training set onto the hyperplane and obtain a reduced dataset  $\mathbf{X}_{d\text{-proj}}$  of dimensionality  $d$ , compute the matrix multiplication of the training set matrix  $\mathbf{X}$  by the

matrix  $\mathbf{W}_d$ , defined as the matrix containing the first  $d$  columns of  $\mathbf{V}$ , as shown in [Equation 7-2](#).

*Equation 7-2. Projecting the training set down to  $d$  dimensions*

$$\mathbf{X}_{d\text{-proj}} = \mathbf{X}\mathbf{W}_d$$

The following Python code projects the training set onto the plane defined by the first two principal components:

```
W2 = Vt[:2].T  
X2D = X_centered @ W2
```

There you have it! You now know how to reduce the dimensionality of any dataset by projecting it down to any number of dimensions, while preserving as much variance as possible.

## Using Scikit-Learn

Scikit-Learn's PCA class uses SVD to implement PCA, just like we did earlier in this chapter. The following code applies PCA to reduce the dimensionality of the dataset down to two dimensions (note that it automatically takes care of centering the data):

```
from sklearn.decomposition import PCA  
  
pca = PCA(n_components=2)  
X2D = pca.fit_transform(X)
```

After fitting the PCA transformer to the dataset, its `components_` attribute holds the transpose of  $\mathbf{W}_d$ : it contains one row for each of the first  $d$  principal components.

## Explained Variance Ratio

Another useful piece of information is the *explained variance ratio* of each principal component, available via the `explained_variance_ratio_` variable. The ratio indicates the proportion of the dataset's variance that lies along each principal component. For example, let's look at the explained variance ratios of the first two components of the 3D dataset represented in [Figure 7-2](#):

```
>>> pca.explained_variance_ratio_  
array([0.82279334, 0.10821224])
```

This output tells us that about 82% of the dataset's variance lies along the first PC, and about 11% lies along the second PC. This leaves about 7% for the third PC, so it is reasonable to assume that the third PC probably carries little information.

## Choosing the Right Number of Dimensions

Instead of arbitrarily choosing the number of dimensions to reduce down to, it is simpler to choose the number of dimensions that add up to a sufficiently large portion of the variance—say, 95% (An exception to this rule, of course, is if you are reducing dimensionality for data visualization, in which case you will want to reduce the dimensionality down to 2 or 3).

The following code loads and splits the MNIST dataset (introduced in [Chapter 3](#)) and performs PCA without reducing dimensionality, then computes the minimum number of dimensions required to preserve 95% of the training set's variance:

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', as_frame=False)
X_train, y_train = mnist.data[:60_000], mnist.target[:60_000]
X_test, y_test = mnist.data[60_000:], mnist.target[60_000:]

pca = PCA()
pca.fit(X_train)
cumsum = np.cumsum(pca.explained_variance_ratio_)
d = np.argmax(cumsum >= 0.95) + 1 # d equals 154
```

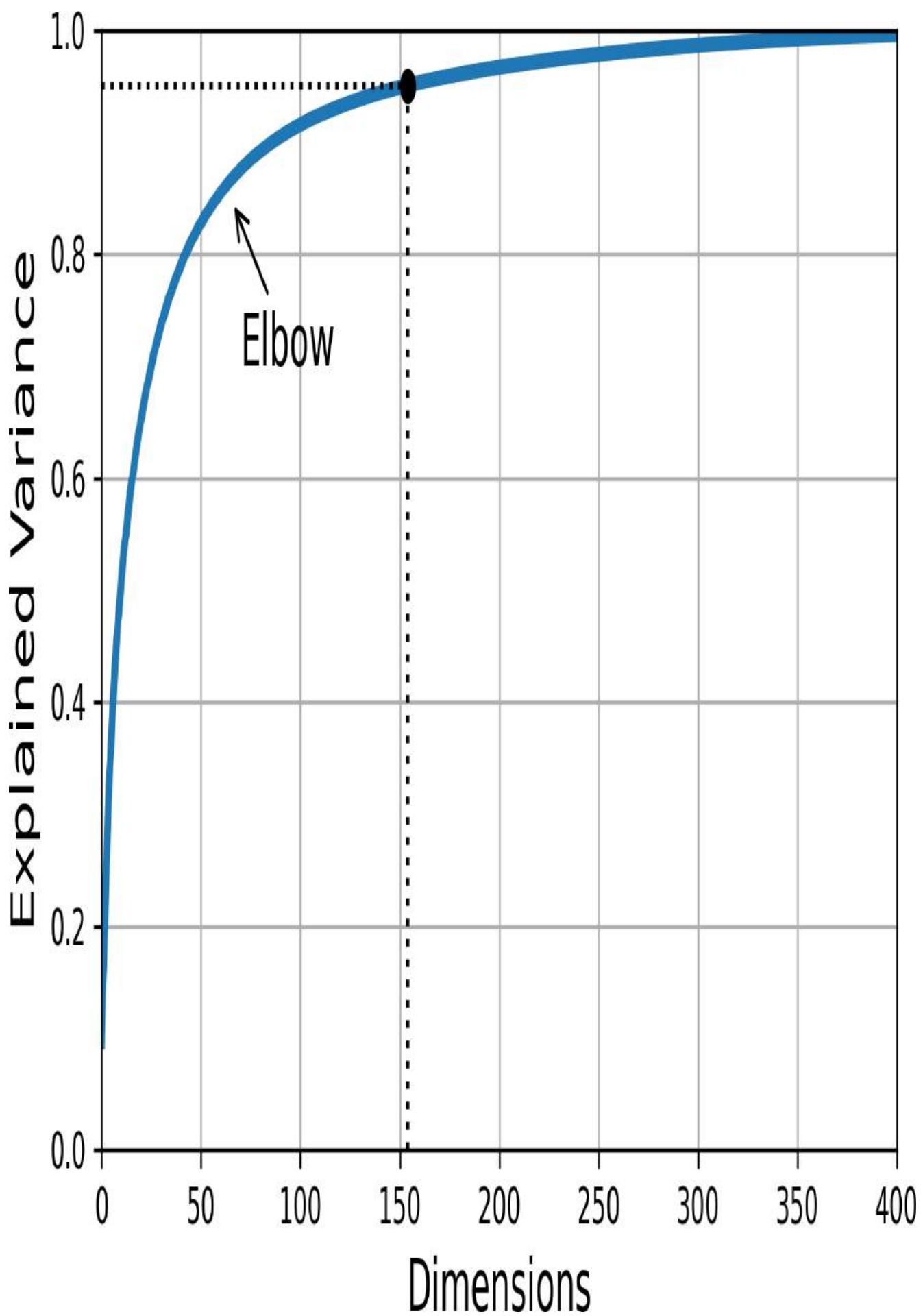
You could then set `n_components=d` and run PCA again, but there's a better option. Instead of specifying the number of principal components you want to preserve, you can set `n_components` to be a float between 0.0 and 1.0, indicating the ratio of variance you wish to preserve:

```
pca = PCA(n_components=0.95)
X_reduced = pca.fit_transform(X_train)
```

The actual number of components is determined during training, and it is stored in the `n_components_` attribute:

```
>>> pca.n_components_
154
```

Yet another option is to plot the explained variance as a function of the number of dimensions (simply plot `cumsum`; see [Figure 7-8](#)). There will usually be an elbow in the curve, where the explained variance stops growing fast. In this case, you can see that reducing the dimensionality down to about 100 dimensions wouldn't lose too much explained variance.



*Figure 7-8. Explained variance as a function of the number of dimensions*

Alternatively, if you are using dimensionality reduction as a preprocessing step for a supervised learning task (e.g., classification), then you can tune the number of dimensions as you would any other hyperparameter (see Chapter 2). For example, the following code example creates a two-step pipeline, first reducing dimensionality using PCA, then classifying using a random forest. Next, it uses `RandomizedSearchCV` to find a good combination of hyperparameters for both PCA and the random forest classifier. This example does a quick search, tuning only 2 hyperparameters, training on just 1,000 instances, and running for just 10 iterations, but feel free to do a more thorough search if you have the time:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.pipeline import make_pipeline

clf = make_pipeline(PCA(random_state=42),
                    RandomForestClassifier(random_state=42))
param_distrib = {
    "pca__n_components": np.arange(10, 80),
    "randomforestclassifier__n_estimators": np.arange(50, 500)
}
rnd_search = RandomizedSearchCV(clf, param_distrib, n_iter=10, cv=3,
                                 random_state=42)
rnd_search.fit(X_train[:1000], y_train[:1000])
```

Let's look at the best hyperparameters found:

```
>>> print(rnd_search.best_params_)
{'randomforestclassifier__n_estimators': 475, 'pca__n_components': 57}
```

It's interesting to note how low the optimal number of components is: we reduced a 784-dimensional dataset to just 57 dimensions! This is tied to the fact that we used a random forest, which is a pretty powerful model. If we used a linear model instead, such as an `SGDClassifier`, the search would find that we need to preserve more dimensions (about 70).

## NOTE

You may also care about the model's size and speed, not just its performance. The fewer dimensions, the smaller the model, and the faster training and inference will be. But if you shrink the data too much, then you will lose too much signal and your model will underfit. You need to choose the right balance of speed, size, and performance for your particular use case.

## PCA for Compression

After dimensionality reduction, the training set takes up much less space. For example, after applying PCA to the MNIST dataset while preserving 95% of its variance, we are left with 154 features, instead of the original 784 features. So the dataset is now less than 20% of its original size, and we only lost 5% of its variance! This is a reasonable compression ratio, and it's easy to see how such a size reduction would speed up a classification algorithm tremendously.

It is also possible to decompress the reduced dataset back to 784 dimensions by applying the inverse transformation of the PCA projection. This won't give you back the original data, since the projection lost a bit of information (within the 5% variance that was dropped), but it will likely be close to the original data. The mean squared distance between the original data and the reconstructed data (compressed and then decompressed) is called the *reconstruction error*.

The `inverse_transform()` method lets us decompress the reduced MNIST dataset back to 784 dimensions:

```
X_recovered = pca.inverse_transform(X_reduced)
```

Figure 7-9 shows a few digits from the original training set (on the left), and the corresponding digits after compression and decompression. You can see that there is a slight image quality loss, but the digits are still mostly intact.

Original

4 8 0 3 /

5 7 1 A 3

1 9 1 0 8

0 9 9 1 9

5 1 7 1 3

Compressed + Decompressed

4 8 0 3 /

5 7 1 A 3

1 9 1 0 8

0 9 9 1 9

5 1 7 1 3

Figure 7-9. MNIST compression that preserves 95% of the variance

The equation for the inverse transformation is shown in [Equation 7-3](#).

*Equation 7-3. PCA inverse transformation, back to the original number of dimensions*

$$\mathbf{X}_{\text{recovered}} = \mathbf{X}_{d-\text{proj}} \mathbf{W}_d^T$$

## Randomized PCA

If you set the `svd_solver` hyperparameter to "randomized", Scikit-Learn uses a stochastic algorithm called *randomized PCA* that quickly finds an approximation of the first  $d$  principal components. Its computational complexity is  $O(m \times d^2) + O(d^3)$ , instead of  $O(m \times n^2) + O(n^3)$  for the full SVD approach, so it is dramatically faster than full SVD when  $d$  is much smaller than  $n$ :

```
rnd_pca = PCA(n_components=154, svd_solver="randomized", random_state=42)
X_reduced = rnd_pca.fit_transform(X_train)
```

### TIP

By default, `svd_solver` is set to "auto": if the input data has few features ( $n < 1,000$ ) and at least 10 times more samples ( $m > 10n$ ), then the "covariance\_eigh" solver is used, which is very fast in these conditions. Otherwise, if  $\max(m, n) > 500$  and `n_components` is an integer smaller than 80% of  $\min(m, n)$ , it uses the "randomized" solver. In other cases, it uses the full SVD approach. If you want to force Scikit-Learn to use full SVD, trading compute time for a slightly more precise result, you can set `svd_solver="full"`.

## Incremental PCA

One problem with the preceding implementations of PCA is that they require the whole training set to fit in memory in order for the algorithm to run. Fortunately, *incremental PCA* (IPCA) algorithms have been developed that allow you to split the training set into mini-batches and feed these in one mini-batch at a time. This is useful for large training sets and for applying PCA online (i.e., on the fly, as new instances arrive).

The following code splits the MNIST training set into 100 mini-batches (using NumPy's `array_split()` function) and feeds them to Scikit-Learn's `IncrementalPCA` class<sup>6</sup> to reduce the dimensionality of the MNIST dataset down to 154 dimensions, just like before. Note that you must call the `partial_fit()` method with each mini-batch, rather than the `fit()` method with the whole training set:

```

from sklearn.decomposition import IncrementalPCA

n_batches = 100
inc_pca = IncrementalPCA(n_components=154)
for X_batch in np.array_split(X_train, n_batches):
    inc_pca.partial_fit(X_batch)

X_reduced = inc_pca.transform(X_train)

```

Alternatively, you can use NumPy's `memmap` class, which allows you to manipulate a large array stored in a binary file on disk as if it were entirely in memory; the class loads only the data it needs in memory, when it needs it. To demonstrate this, let's first create a memory-mapped (`memmap`) file and copy the MNIST training set to it, then call `flush()` to ensure that any data still in the cache gets saved to disk. In real life, `X_train` would typically not fit in memory, so you would load it chunk by chunk and save each chunk to the right part of the `memmap` array:

```

filename = "my_mnist.memmap"
X mmap = np.memmap(filename, dtype='float32', mode='write', shape=X_train.shape)
X mmap[:] = X_train # could be a loop instead, saving the data chunk by chunk
X mmap.flush()

```

Next, we can load the `memmap` file and use it like a regular NumPy array. Let's use the `IncrementalPCA` class to reduce its dimensionality. Since this algorithm uses only a small part of the array at any given time, memory usage remains under control. This makes it possible to call the usual `fit()` method instead of `partial_fit()`, which is quite convenient:

```

X mmap = np.memmap(filename, dtype="float32", mode="readonly").reshape(-1, 784)
batch_size = X mmap.shape[0] // n_batches
inc_pca = IncrementalPCA(n_components=154, batch_size=batch_size)
inc_pca.fit(X mmap)

```

### WARNING

Only the raw binary data is saved to disk, so you need to specify the data type and shape of the array when you load it. If you omit the shape, `np.memmap()` returns a 1D array.

For very high-dimensional datasets, PCA can be too slow. As you saw earlier, even if you use randomized PCA its computational complexity is still  $O(m \times d^2) + O(d^3)$ , so the target number of dimensions  $d$  must not be too large. If you are dealing with a dataset

with tens of thousands of features or more (e.g., images), then training may become much too slow: in this case, you should consider using random projection instead.

## Random Projection

As its name suggests, the random projection algorithm projects the data to a lower-dimensional space using a random linear projection. This may sound crazy, but it turns out that such a random projection is actually very likely to preserve distances fairly well, as was demonstrated mathematically by William B. Johnson and Joram Lindenstrauss in a famous lemma. So, two similar instances will remain similar after the projection, and two very different instances will remain very different.

Obviously, the more dimensions you drop, the more information is lost, and the more distances get distorted. So how can you choose the optimal number of dimensions? Well, Johnson and Lindenstrauss came up with an equation that determines the minimum number of dimensions to preserve in order to ensure—with high probability—that distances won’t change by more than a given tolerance. For example, if you have a dataset containing  $m = 5,000$  instances with  $n = 20,000$  features each, and you don’t want the squared distance between any two instances to change by more than  $\varepsilon = 10\%$ ,<sup>7</sup> then you should project the data down to  $d$  dimensions, with  $d \geq 4 \log(m) / (\frac{1}{2} \varepsilon^2 - \frac{1}{3} \varepsilon^3)$ , which is 7,300 dimensions. That’s quite a significant dimensionality reduction! Notice that the equation does not use  $n$ , it only relies on  $m$  and  $\varepsilon$ . This equation is implemented by the `johnson_lindenstrauss_min_dim()` function:

```
>>> from sklearn.random_projection import johnson_lindenstrauss_min_dim
>>> m, ε = 5_000, 0.1
>>> d = johnson_lindenstrauss_min_dim(m, eps=ε)
>>> d
7300
```

Now we can just generate a random matrix  $\mathbf{P}$  of shape  $[d, n]$ , where each item is sampled randomly from a Gaussian distribution with mean 0 and variance  $1 / d$ , and use it to project a dataset from  $n$  dimensions down to  $d$ :

```
n = 20_000
rng = np.random.default_rng(seed=42)
P = rng.standard_normal((d, n)) / np.sqrt(d) # std dev = sqrt(variance)

X = rng.standard_normal((m, n)) # generate a fake dataset
X_reduced = X @ P.T
```

That's all there is to it! It's simple and efficient, and training is almost instantaneous: the only thing the algorithm needs to create the random matrix is the dataset's shape. The data itself is not used at all. This makes random projection particularly well suited for very high dimensional data such as text or genomics with millions of features, or very sparse data, for which even randomized PCA may take too long to train and require too much memory. At inference time, random projection is just as fast as PCA (i.e., one matrix multiplication). That said, random projection is not a silver bullet: it loses a bit more signal than PCA, so there's a trade-off between training speed and performance.

Scikit-Learn offers a `GaussianRandomProjection` class to do exactly what we just did: when you call its `fit()` method, it uses `johnson_lindenstrauss_min_dim()` to determine the output dimensionality, then it generates a random matrix, which it stores in the `components_` attribute. Then when you call `transform()`, it uses this matrix to perform the projection. When creating the transformer, you can set `eps` to tweak  $\varepsilon$  (it defaults to 0.1), and `n_components` to force a specific target dimensionality  $d$  (you will probably want to fine-tune these hyperparameters using cross-validation). The following code example gives the same result as the preceding code (you can also verify that `gaussian_rnd_proj.components_` is equal to  $P$ ):

```
from sklearn.random_projection import GaussianRandomProjection

gaussian_rnd_proj = GaussianRandomProjection(eps=ε, random_state=42)
X_reduced = gaussian_rnd_proj.fit_transform(X) # same result as above
```

Scikit-Learn also provides a second random projection transformer, known as `SparseRandomProjection`. It determines the target dimensionality in the same way, generates a random matrix of the same shape, and performs the projection identically. The main difference is that the random matrix is sparse. This means it uses much less memory: about 25 MB instead of almost 1.2 GB in the preceding example! And it's also much faster, both to generate the random matrix and to reduce dimensionality: about 50% faster in this case. Moreover, if the input is sparse, the transformation keeps it sparse (unless you set `dense_output=True`). Lastly, it enjoys the same distance-preserving property as the previous approach, and the quality of the dimensionality reduction is comparable (only very slightly less accurate). In short, it's usually preferable to use this transformer instead of the first one, especially for large or sparse datasets.

The ratio  $r$  of nonzero items in the sparse random matrix is called its *density*. By default, it is equal to  $\frac{1}{\sqrt{n}}$ . With 20,000 features, this means that only 1 in  $\sim 141$  cells in

the random matrix is nonzero: that's quite sparse! You can set the `density` hyperparameter to another value if you prefer. Each cell in the sparse random matrix has a probability  $r$  of being nonzero, and each nonzero value is either  $-v$  or  $+v$  (both equally likely), where  $v = \frac{1}{\sqrt{dr}}$ .

If you want to perform the inverse transform, you first need to compute the pseudo-inverse of the components matrix using SciPy's `pinv()` function, then multiply the reduced data by the transpose of the pseudo-inverse:

```
components_pinv = np.linalg.pinv(gaussian_rnd_proj.components_)
X_recovered = X_reduced @ components_pinv.T
```

### WARNING

Computing the pseudo-inverse may take a very long time if the components matrix is large, as the computational complexity of `pinv()` is  $O(dn^2)$  if  $d < n$ , or  $O(nd^2)$  otherwise.

In summary, random projection is a simple, fast, memory-efficient, and surprisingly powerful dimensionality reduction algorithm that you should keep in mind, especially when you deal with high-dimensional datasets.

### NOTE

Random projection is not always used to reduce the dimensionality of large datasets. For example, a [2017 paper<sup>8</sup>](#) by Sanjoy Dasgupta et al. showed that the brain of a fruit fly implements an analog of random projection to map dense low-dimensional olfactory inputs to sparse high-dimensional binary outputs: for each odor, only a small fraction of the output neurons get activated, but similar odors activate many of the same neurons. This is similar to a well-known algorithm called *locality sensitive hashing* (LSH), which is typically used in search engines to group similar documents.

## LLE

*Locally linear embedding (LLE)*<sup>9</sup> is a *nonlinear dimensionality reduction* (NLDR) technique. It is a manifold learning technique that does not rely on projections, unlike PCA and random projection. In a nutshell, LLE first determines how each training instance linearly relates to its nearest neighbors, then it looks for a low-dimensional representation of the training set where these local relationships are best preserved (more details shortly). This approach makes it particularly good at unrolling twisted

manifolds, especially when there is not too much noise. However, it does not scale well so it is mostly for small or medium sized datasets.

The following code makes a Swiss roll, then uses Scikit-Learn's `LocallyLinearEmbedding` class to unroll it:

```
from sklearn.datasets import make_swiss_roll
from sklearn.manifold import LocallyLinearEmbedding

X_swiss, t = make_swiss_roll(n_samples=1000, noise=0.2, random_state=42)
lle = LocallyLinearEmbedding(n_components=2, n_neighbors=10, random_state=42)
X_unrolled = lle.fit_transform(X_swiss)
```

The variable `t` is a 1D NumPy array containing the position of each instance along the rolled axis of the Swiss roll. We don't use it in this example, but it can be used as a target for a nonlinear regression task.

The resulting 2D dataset is shown in [Figure 7-10](#). As you can see, the Swiss roll is completely unrolled, and the distances between instances are locally well preserved. However, distances are not preserved on a larger scale: the unrolled Swiss roll should be a rectangle, not this kind of stretched and twisted band. Nevertheless, LLE did a pretty good job of modeling the manifold.

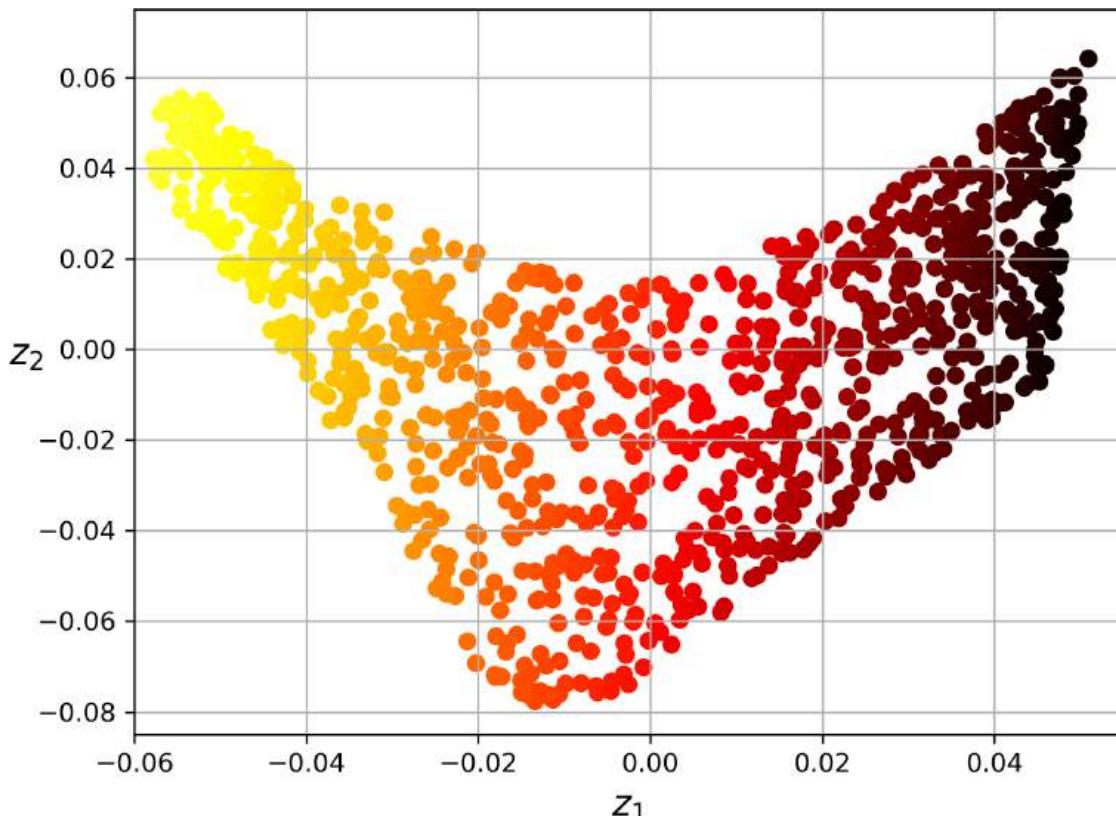


Figure 7-10. Unrolled Swiss roll using LLE

Here's how LLE works: for each training instance  $\mathbf{x}^{(i)}$ , the algorithm identifies its  $k$ -nearest neighbors (in the preceding code  $k = 10$ ), then tries to reconstruct  $\mathbf{x}^{(i)}$  as a linear function of these neighbors. More specifically, it tries to find the weights  $w_{i,j}$  such that the squared distance between  $\mathbf{x}^{(i)}$  and  $\sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)}$  is as small as possible, assuming  $w_{i,j} = 0$  if  $\mathbf{x}^{(j)}$  is not one of the  $k$ -nearest neighbors of  $\mathbf{x}^{(i)}$ . Thus the first step of LLE is the constrained optimization problem described in [Equation 7-4](#), where  $\mathbf{W}$  is the weight matrix containing all the weights  $w_{i,j}$ . The second constraint simply normalizes the weights for each training instance  $\mathbf{x}^{(i)}$ .

*Equation 7-4. LLE step 1: linearly modeling local relationships*

$$\begin{aligned}\widehat{\mathbf{W}} &= \underset{\mathbf{W}}{\operatorname{argmin}} \sum_{i=1}^m \left( \mathbf{x}^{(i)} - \sum_{j=1}^m w_{i,j} \mathbf{x}^{(j)} \right)^2 \\ \text{subject to } &\begin{cases} w_{i,j} = 0 & \text{if } \mathbf{x}^{(j)} \text{ is not one of the } k \text{ n.n. of } \mathbf{x}^{(i)} \\ \sum_{j=1}^m w_{i,j} = 1 & \text{for } i = 1, 2, \dots, m \end{cases}\end{aligned}$$

After this step, the weight matrix  $\widehat{\mathbf{W}}$  (containing the weights  $\widehat{w}_{i,j}$ ) encodes the local linear relationships between the training instances. The second step is to map the training instances into a  $d$ -dimensional space (where  $d < n$ ) while preserving these local relationships as much as possible. If  $\mathbf{z}^{(i)}$  is the image of  $\mathbf{x}^{(i)}$  in this  $d$ -dimensional space, then we want the squared distance between  $\mathbf{z}^{(i)}$  and  $\sum_{j=1}^m \widehat{w}_{i,j} \mathbf{z}^{(j)}$  to be as small as possible. This idea leads to the unconstrained optimization problem described in [Equation 7-5](#). It looks very similar to the first step, but instead of keeping the instances fixed and finding the optimal weights, we are doing the reverse: keeping the weights fixed and finding the optimal position of the instances' images in the low-dimensional space. Note that  $\mathbf{Z}$  is the matrix containing all  $\mathbf{z}^{(i)}$ .

*Equation 7-5. LLE step 2: reducing dimensionality while preserving relationships*

$$\widehat{\mathbf{Z}} = \underset{\mathbf{Z}}{\operatorname{argmin}} \sum_{i=1}^m \left( \mathbf{z}^{(i)} - \sum_{j=1}^m \widehat{w}_{i,j} \mathbf{z}^{(j)} \right)^2$$

Scikit-Learn's LLE implementation has the following computational complexity:  $O(m \log(m)n \log(k))$  for finding the  $k$ -nearest neighbors,  $O(mnk^3)$  for optimizing the weights,

and  $O(dm^2)$  for constructing the low-dimensional representations. Unfortunately, the  $m^2$  in the last term makes this algorithm scale poorly to very large datasets.

As you can see, LLE is quite different from the projection techniques, and it's significantly more complex, but it can also construct much better low-dimensional representations, especially if the data is nonlinear.

## Other Dimensionality Reduction Techniques

Before we conclude this chapter, let's take a quick look at a few other popular dimensionality reduction techniques available in Scikit-Learn:

### `sklearn.manifold.MDS`

*Multidimensional scaling* (MDS) reduces dimensionality while trying to preserve the distances between the instances. Random projection does that for high-dimensional data, but it doesn't work well on low-dimensional data.

### `sklearn.manifold.Isomap`

*Isomap* creates a graph by connecting each instance to its nearest neighbors, then reduces dimensionality while trying to preserve the *geodesic distances* between the instances. The geodesic distance between two nodes in a graph is the number of nodes on the shortest path between these nodes. This approach works best when the data lies on a fairly smooth and low-dimensional manifold with a single global structure (e.g., the Swiss roll).

### `sklearn.manifold.TSNE`

*t-distributed stochastic neighbor embedding* (t-SNE) reduces dimensionality while trying to keep similar instances close and dissimilar instances apart. It is mostly used for visualization, in particular to visualize clusters of instances in high-dimensional space. For example, in the exercises at the end of this chapter you will use t-SNE to visualize a 2D map of the MNIST images. However, it is not meant to be used as a preprocessing stage for an ML model.

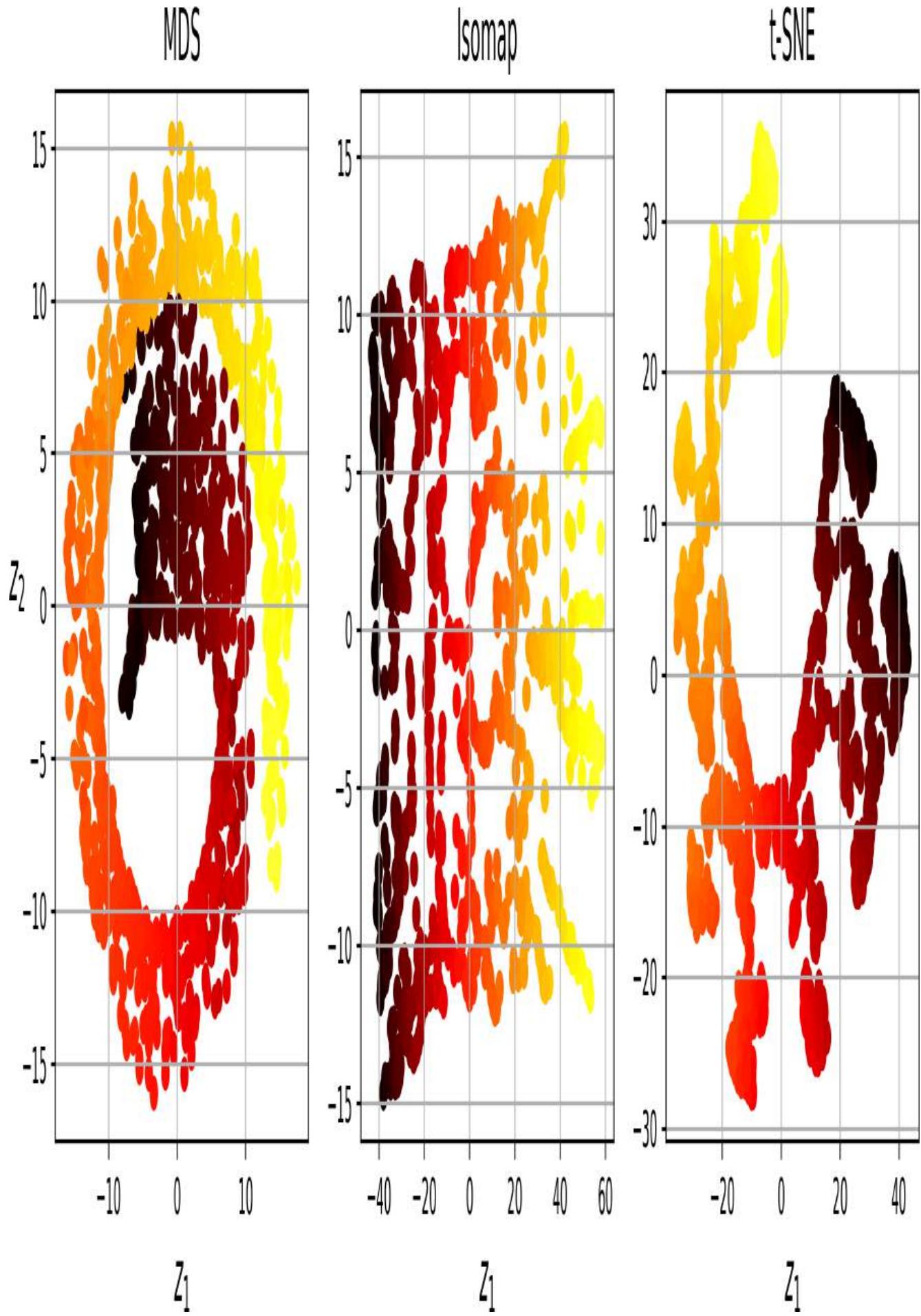
### `sklearn.discriminant_analysis.LinearDiscriminantAnalysis`

*Linear discriminant analysis* (LDA) is a linear classification algorithm that, during training, learns the most discriminative axes between the classes. These axes can then be used to define a hyperplane onto which to project the data. The benefit of this approach is that the projection will keep classes as far apart as possible, so LDA is a good technique to reduce dimensionality before running another classification algorithm (unless LDA alone is sufficient).

### TIP

*Uniform Manifold Approximation and Projection* (UMAP) is another popular dimensionality reduction technique for visualization. While t-SNE is better at preserving the local structure, especially clusters, UMAP tries to preserve both the local and global structures. Moreover, it scales better to large datasets. Sadly, it is not available in Scikit-Learn, but there's a good implementation in the [umap-learn package](#).

Figure 7-11 shows the results of MDS, Isomap, and t-SNE on the Swiss roll. MDS manages to flatten the Swiss roll without losing its global curvature, while Isomap drops it entirely. Depending on the downstream task, preserving the large-scale structure may be good or bad. t-SNE does a reasonable job of flattening the Swiss roll, preserving a bit of curvature, and it also amplifies clusters, tearing the roll apart. Again, this might be good or bad, depending on the downstream task.



*Figure 7-11. Using various techniques to reduce the Swiss roll to 2D*

## Exercises

1. What are the main motivations for reducing a dataset’s dimensionality? What are the main drawbacks?
2. What is the curse of dimensionality?
3. Once a dataset’s dimensionality has been reduced, is it possible to reverse the operation? If so, how? If not, why?
4. Can PCA be used to reduce the dimensionality of a highly nonlinear dataset?
5. Suppose you perform PCA on a 1,000-dimensional dataset, setting the explained variance ratio to 95%. How many dimensions will the resulting dataset have?
6. In what cases would you use regular PCA, incremental PCA, randomized PCA, or random projection?
7. How can you evaluate the performance of a dimensionality reduction algorithm on your dataset?
8. Does it make any sense to chain two different dimensionality reduction algorithms?
9. Load the MNIST dataset (introduced in [Chapter 3](#)) and split it into a training set and a test set (take the first 60,000 instances for training, and the remaining 10,000 for testing). Train a random forest classifier on the dataset and time how long it takes, then evaluate the resulting model on the test set. Next, use PCA to reduce the dataset’s dimensionality, with an explained variance ratio of 95%. Train a new random forest classifier on the reduced dataset and see how long it takes. Was training much faster? Next, evaluate the classifier on the test set. How does it compare to the previous classifier? Try again with an `SGDClassifier`. How much does PCA help now?
10. Use t-SNE to reduce the first 5,000 images of the MNIST dataset down to 2 dimensions and plot the result using Matplotlib. You can use a scatterplot using 10 different colors to represent each image’s target class. Alternatively, you can replace each dot in the scatterplot with the corresponding instance’s class

(a digit from 0 to 9), or even plot scaled-down versions of the digit images themselves (if you plot all digits the visualization will be too cluttered, so you should either draw a random sample or plot an instance only if no other instance has already been plotted at a close distance). You should get a nice visualization with well-separated clusters of digits. Try using other dimensionality reduction algorithms, such as PCA, LLE, or MDS, and compare the resulting visualizations.

Solutions to these exercises are available at the end of this chapter’s notebook, at <https://homl.info/colab-p>.

---

- <sup>1</sup> Well, four dimensions if you count time, and a few more if you are a string theorist.
- <sup>2</sup> Watch a rotating tesseract projected into 3D space at <https://homl.info/30>. Image by Wikipedia user NerdBoy1392 (Creative Commons BY-SA 3.0). Reproduced from <https://en.wikipedia.org/wiki/Tesseract>.
- <sup>3</sup> Fun fact: anyone you know is probably an extremist in at least one dimension (e.g., how much sugar they put in their coffee), if you consider enough dimensions.
- <sup>4</sup> Karl Pearson, “On Lines and Planes of Closest Fit to Systems of Points in Space”, *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2, no. 11 (1901): 559–572.
- <sup>5</sup> The proof that SVD happens to give us exactly the principal components we need for PCA requires some prerequisite math knowledge such as eigenvectors and covariance matrices. If you are curious, you will find all the details in this [2014 paper by Jonathon Shlens](#).
- <sup>6</sup> Scikit-Learn uses the [algorithm](#) described in David A. Ross et al., “Incremental Learning for Robust Visual Tracking”, *International Journal of Computer Vision* 77, no. 1–3 (2008): 125–141.
- <sup>7</sup>  $\varepsilon$  is the Greek letter epsilon, often used for tiny values.
- <sup>8</sup> Sanjoy Dasgupta et al., “A neural algorithm for a fundamental computing problem”, *Science* 358, no. 6364 (2017): 793–796.
- <sup>9</sup> Sam T. Roweis and Lawrence K. Saul, “Nonlinear Dimensionality Reduction by Locally Linear Embedding”, *Science* 290, no. 5500 (2000): 2323–2326.

# Chapter 8. Unsupervised Learning Techniques

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 8th chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

Although most of the applications of machine learning today are based on supervised learning (and as a result, this is where most of the investments go to), the vast majority of the available data is unlabeled: we have the input features  $\mathbf{X}$ , but we do not have the labels  $\mathbf{y}$ . The computer scientist Yann LeCun famously said that “if intelligence was a cake, unsupervised learning would be the cake, supervised learning would be the icing on the cake, and reinforcement learning would be the cherry on the cake.” In other words, there is a huge potential in unsupervised learning that we have only barely started to sink our teeth into.

Say you want to create a system that will take a few pictures of each item on a manufacturing production line and detect which items are defective. You can fairly easily create a system that will take pictures automatically, and this might give you thousands of pictures every day. You can then build a reasonably large dataset in just a few weeks. But wait, there are no labels! If you want to train a regular binary classifier that will predict whether an item is defective or not, you will need to label every single picture as “defective” or “normal”. This will generally require human experts to sit down and manually go through all the pictures. This is a long, costly, and tedious task, so it will usually only be done on a small subset of the available pictures. As a result, the labeled dataset will be quite small, and the classifier’s performance will be disappointing. Moreover, every time the company makes any change to its products, the whole process will need to be started over from scratch. Wouldn’t it be great if the

algorithm could just exploit the unlabeled data without needing humans to label every picture? Enter unsupervised learning.

In [Chapter 7](#) we looked at the most common unsupervised learning task: dimensionality reduction. In this chapter we will look at a few more unsupervised tasks:

### *Clustering*

The goal is to group similar instances together into *clusters*. Clustering is a great tool for data analysis, customer segmentation, recommender systems, search engines, image segmentation, semi-supervised learning, dimensionality reduction, and more.

### *Anomaly detection (also called outlier detection)*

The objective is to learn what “normal” data looks like, and then use that to detect abnormal instances. These instances are called *anomalies*, or *outliers*, while the normal instances are called *inliers*. Anomaly detection is useful in a wide variety of applications, such as fraud detection, detecting defective products in manufacturing, identifying new trends in time series, or removing outliers from a dataset before training another model, which can significantly improve the performance of the resulting model.

### *Density estimation*

This is the task of estimating the *probability density function* (PDF) of the random process that generated the dataset.<sup>1</sup> Density estimation is commonly used for anomaly detection: instances located in very low-density regions are likely to be anomalies. It is also useful for data analysis and visualization.

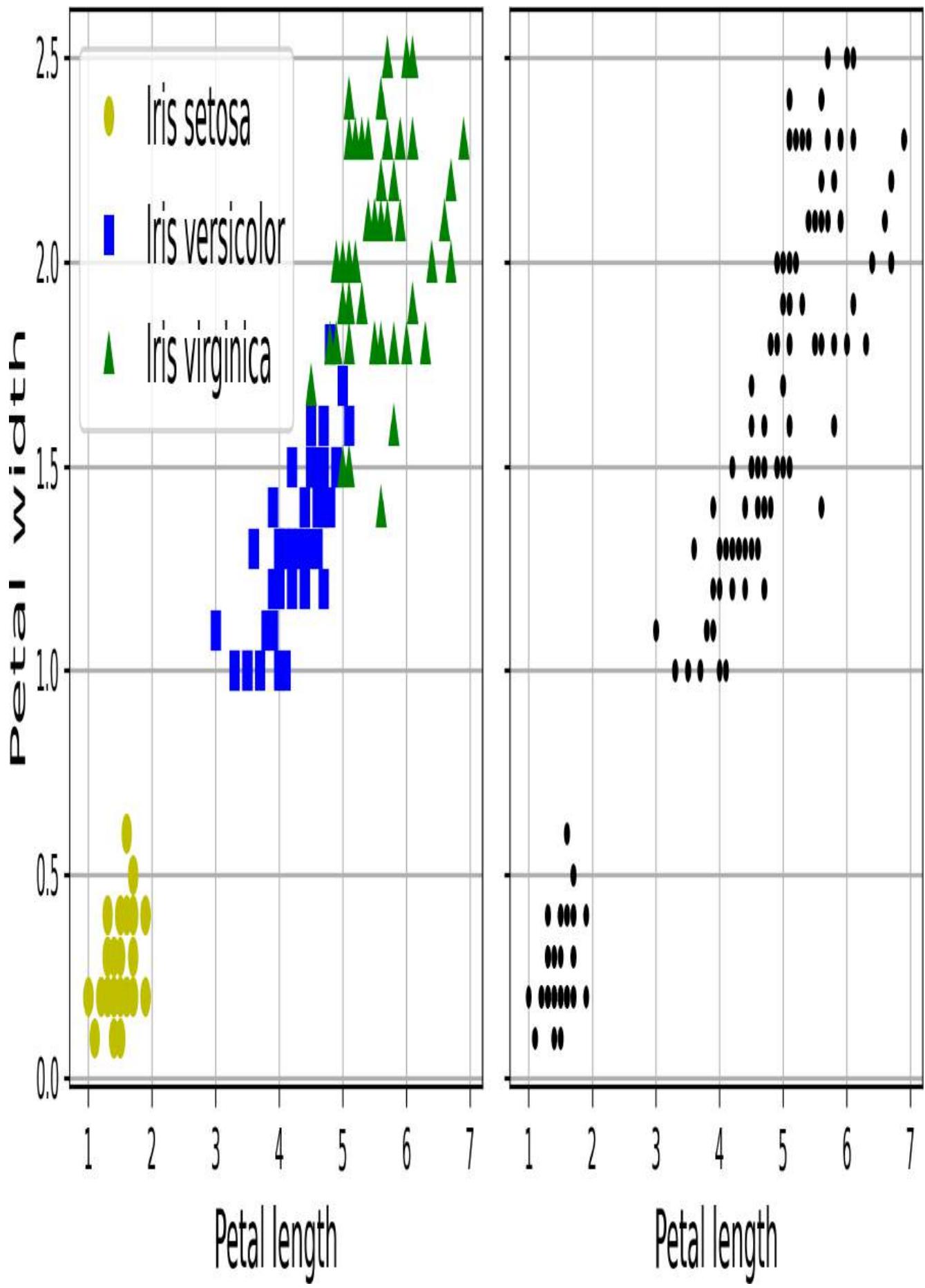
Ready for some cake? We will start with two clustering algorithms, *k*-means and DBSCAN, then we’ll discuss Gaussian mixture models and see how they can be used for density estimation, clustering, and anomaly detection.

## **Clustering Algorithms: k-means and DBSCAN**

As you enjoy a hike in the mountains, you stumble upon a plant you have never seen before. You look around and you notice a few more. They are not identical, yet they are

sufficiently similar for you to know that they most likely belong to the same species (or at least the same genus). You may need a botanist to tell you what species that is, but you certainly don't need an expert to identify groups of similar-looking objects. This is called *clustering*: it is the task of identifying similar instances and assigning them to *clusters*, or groups of similar instances.

Just like in classification, each instance gets assigned to a group. However, unlike classification, clustering is an unsupervised task, there are no labels, so the algorithm needs to figure out on its own how to group instances. Consider [Figure 8-1](#): on the left is the iris dataset (introduced in [Chapter 4](#)), where each instance's species (i.e., its class) is represented with a different marker. It is a labeled dataset, for which classification algorithms such as logistic regression, SVMs, or random forest classifiers are well suited. On the right is the same dataset, but without the labels, so you cannot use a classification algorithm anymore. This is where clustering algorithms step in: many of them can easily detect the lower-left cluster. It is also quite easy to see with our own eyes, but it is not so obvious that the upper-right cluster is composed of two distinct subclusters. That said, the dataset has two additional features (sepal length and width) that are not represented here, and clustering algorithms can make good use of all features, so in fact they identify the three clusters fairly well (e.g., using a Gaussian mixture model, only 5 instances out of 150 are assigned to the wrong cluster).



*Figure 8-1. Classification (left) versus clustering (right): in clustering, the dataset is unlabeled so the algorithm must identify the clusters without guidance*

Clustering is used in a wide variety of applications, including:

#### *Customer segmentation*

You can cluster your customers based on their purchases and their activity on your website. This is useful to understand who your customers are and what they need, so you can adapt your products and marketing campaigns to each segment. For example, customer segmentation can be useful in *recommender systems* to suggest content that other users in the same cluster enjoyed.

#### *Data analysis*

When you analyze a new dataset, it can be helpful to run a clustering algorithm, and then analyze each cluster separately.

#### *Dimensionality reduction*

Once a dataset has been clustered, it is usually possible to measure each instance's *affinity* with each cluster; affinity is any measure of how well an instance fits into a cluster. Each instance's feature vector  $\mathbf{x}$  can then be replaced with the vector of its cluster affinities. If there are  $k$  clusters, then this vector is  $k$ -dimensional. The new vector is typically much lower-dimensional than the original feature vector, but it can preserve enough information for further processing.

#### *Feature engineering*

The cluster affinities can often be useful as extra features. For example, we used  $k$ -means in [Chapter 2](#) to add geographic cluster affinity features to the California housing dataset, and they helped us get better performance.

#### *Anomaly detection (also called outlier detection)*

Any instance that has a low affinity to all the clusters is likely to be an anomaly. For example, if you have clustered the users of your website based on their behavior, you can detect users with unusual behavior, such as an unusual number of requests per second.

## *Semi-supervised learning*

If you only have a few labels, you could perform clustering and propagate the labels to all the instances in the same cluster. This technique can greatly increase the number of labels available for a subsequent supervised learning algorithm, and thus improve its performance.

## *Search engines*

Some search engines let you search for images that are similar to a reference image. To build such a system, you would first apply a clustering algorithm to all the images in your database; similar images would end up in the same cluster. Then when a user provides a reference image, all you'd need to do is use the trained clustering model to find this image's cluster, and you could then simply return all the images from this cluster.

## *Image segmentation*

By clustering pixels according to their color, then replacing each pixel's color with the mean color of its cluster, it is possible to considerably reduce the number of different colors in an image. Image segmentation is used in many object detection and tracking systems, as it makes it easier to detect the contour of each object.

There is no universal definition of what a cluster is: it really depends on the context, and different algorithms will capture different kinds of clusters. Some algorithms look for instances centered around a particular point, called a *centroid*. Others look for continuous regions of densely packed instances: these clusters can take on any shape. Some algorithms are hierarchical, looking for clusters of clusters. And the list goes on.

In this section, we will look at two popular clustering algorithms,  $k$ -means and DBSCAN, and explore some of their applications, such as nonlinear dimensionality reduction, semi-supervised learning, and anomaly detection.

## **k-means**

Consider the unlabeled dataset represented in [Figure 8-2](#): you can clearly see five blobs of instances. The  $k$ -means algorithm is a simple algorithm capable of clustering this

kind of dataset very quickly and efficiently, often in just a few iterations. It was proposed by Stuart Lloyd at Bell Labs in 1957 as a technique for pulse-code modulation, but it was only **published** outside of the company in 1982.<sup>2</sup> In 1965, Edward W. Forgy had published virtually the same algorithm, so  $k$ -means is sometimes referred to as the Lloyd–Forgy algorithm.

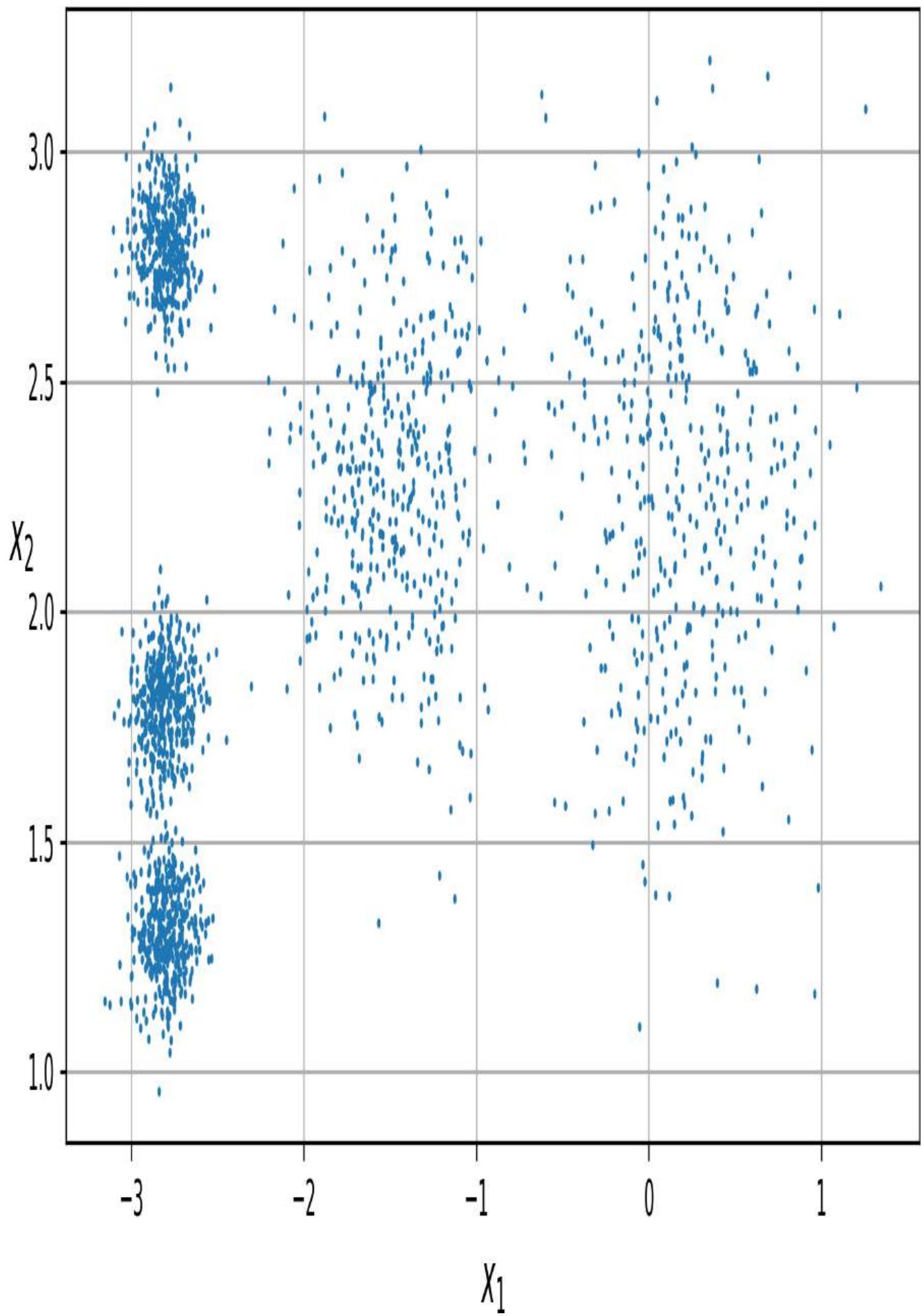


Figure 8-2. An unlabeled dataset composed of five blobs of instances

Let's train a  $k$ -means clusterer on this dataset. It will try to find each blob's center and assign each instance to the closest blob:

```
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

X, y = make_blobs([...]) # make the blobs: y contains the cluster IDs, but we
                        # will not use them; that's what we want to predict
k = 5
kmeans = KMeans(n_clusters=k, random_state=42)
y_pred = kmeans.fit_predict(X)
```

Note that you have to specify the number of clusters  $k$  that the algorithm must find. In this example, it is pretty obvious from looking at the data that  $k$  should be set to 5, but in general it is not that easy. We will discuss this shortly.

Each instance will be assigned to one of the five clusters. In the context of clustering, an instance's *label* is the index of the cluster to which the algorithm assigns this instance; this is not to be confused with the class labels in classification, which are used as targets (remember that clustering is an unsupervised learning task). The `KMeans` instance preserves the predicted labels of the instances it was trained on, available via the `labels_` instance variable:

```
>>> y_pred
array([4, 0, 1, ..., 2, 1, 0], dtype=int32)
>>> y_pred is kmeans.labels_
True
```

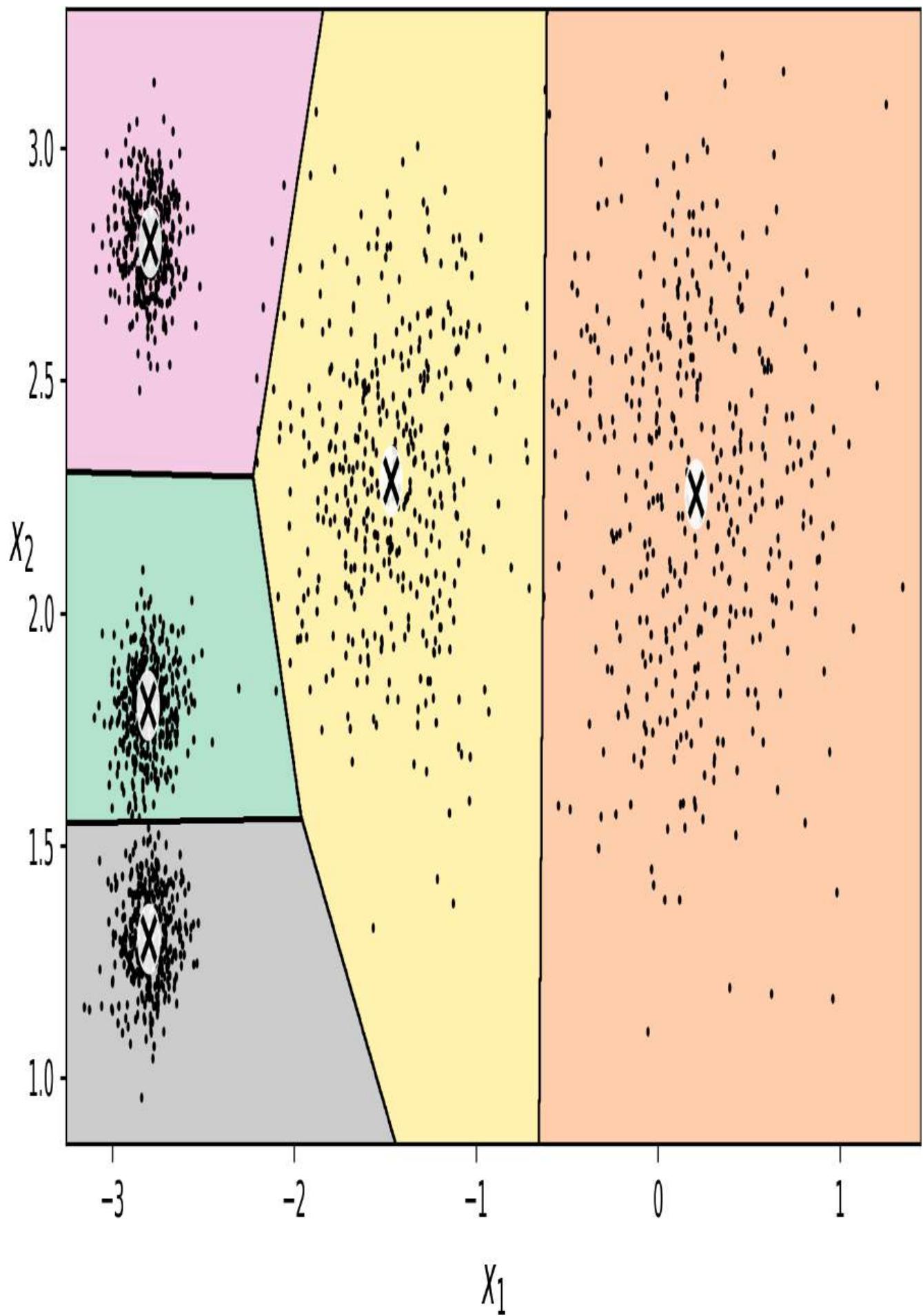
We can also take a look at the five centroids that the algorithm found:

```
>>> kmeans.cluster_centers_
array([[-2.80389616,  1.80117999],
       [ 0.20876306,  2.25551336],
       [-2.79290307,  2.79641063],
       [-1.46679593,  2.28585348],
       [-2.80037642,  1.30082566]])
```

You can easily assign new instances to the cluster whose centroid is closest:

```
>>> import numpy as np
>>> X_new = np.array([[0, 2], [3, 2], [-3, 3], [-3, 2.5]])
>>> kmeans.predict(X_new)
array([1, 1, 2, 2], dtype=int32)
```

If you plot the cluster's decision boundaries, you get a Voronoi tessellation: see [Figure 8-3](#), where each centroid is represented with an  $\textcircled{X}$ .



*Figure 8-3. k-means decision boundaries (Voronoi tessellation)*

The vast majority of the instances were clearly assigned to the appropriate cluster, but a few instances were probably mislabeled, especially near the boundary between the top-left cluster and the central cluster. Indeed, the  $k$ -means algorithm does not behave very well when the blobs have very different diameters because all it cares about when assigning an instance to a cluster is the distance to the centroid.

Instead of assigning each instance to a single cluster, which is called *hard clustering*, it can be useful to give each instance a score per cluster, which is called *soft clustering*. The score can be the distance between the instance and the centroid or a similarity score (or affinity), such as the Gaussian radial basis function we used in [Chapter 2](#). In the `KMeans` class, the `transform()` method measures the distance from each instance to every centroid:

```
>>> kmeans.transform(X_new).round(2)
array([[2.81, 0.33, 2.9 , 1.49, 2.89],
       [5.81, 2.8 , 5.85, 4.48, 5.84],
       [1.21, 3.29, 0.29, 1.69, 1.71],
       [0.73, 3.22, 0.36, 1.55, 1.22]])
```

In this example, the first instance in `X_new` is located at a distance of about 2.81 from the first centroid, 0.33 from the second centroid, 2.90 from the third centroid, 1.49 from the fourth centroid, and 2.89 from the fifth centroid. If you have a high-dimensional dataset and you transform it this way, you end up with a  $k$ -dimensional dataset: this transformation can be a very efficient nonlinear dimensionality reduction technique. Alternatively, you can use these distances as extra features to train another model, as in [Chapter 2](#).

## The k-means algorithm

So, how does the algorithm work? Well, suppose you were given the centroids. You could easily label all the instances in the dataset by assigning each of them to the cluster whose centroid is closest. Conversely, if you were given all the instance labels, you could easily locate each cluster's centroid by computing the mean of the instances in that cluster. But you are given neither the labels nor the centroids, so how can you proceed? Start by placing the centroids randomly (e.g., by picking  $k$  instances at random from the dataset and using their locations as centroids). Then label the instances, update the centroids, label the instances, update the centroids, and so on until the centroids stop moving. The algorithm is guaranteed to converge in a finite number of steps (usually quite small). That's because the mean squared distance between the instances and their

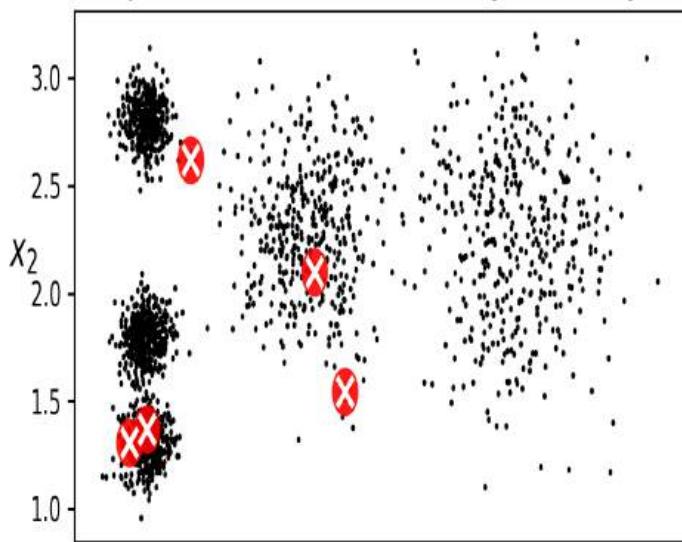
closest centroids can only go down at each step, and since it cannot be negative, it's guaranteed to converge.

You can see the algorithm in action in [Figure 8-4](#): the centroids are initialized randomly (top left), then the instances are labeled (top right), then the centroids are updated (center left), the instances are relabeled (center right), and so on. As you can see, in just three iterations the algorithm has reached a clustering that seems close to optimal.

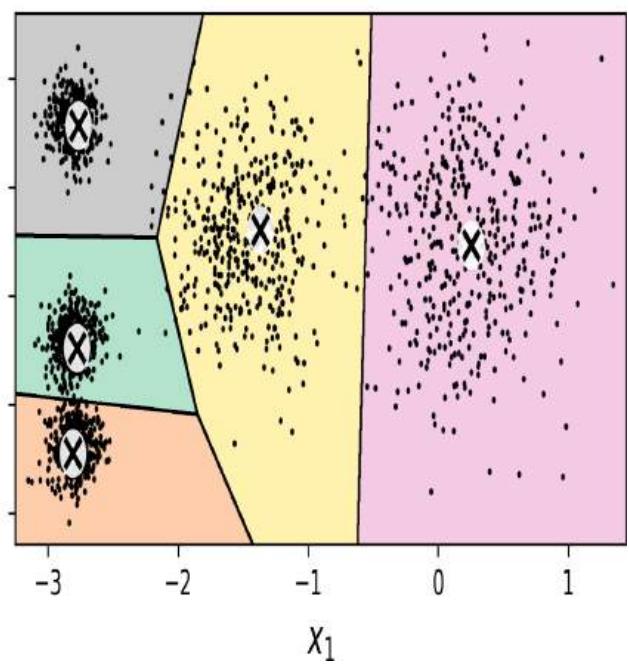
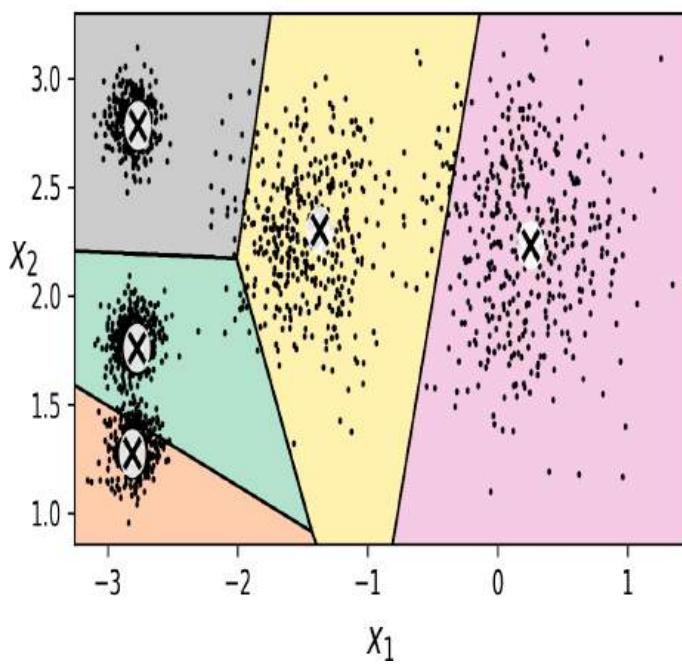
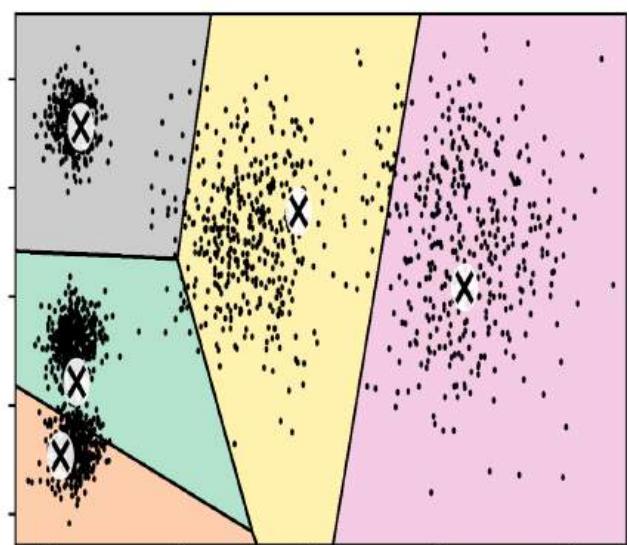
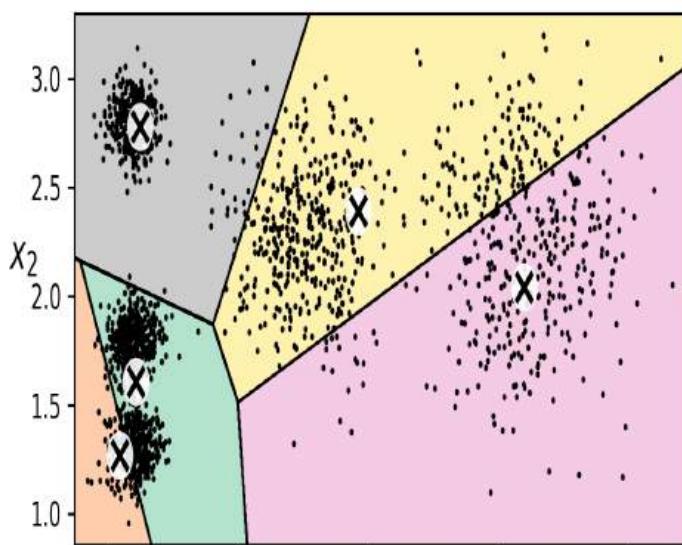
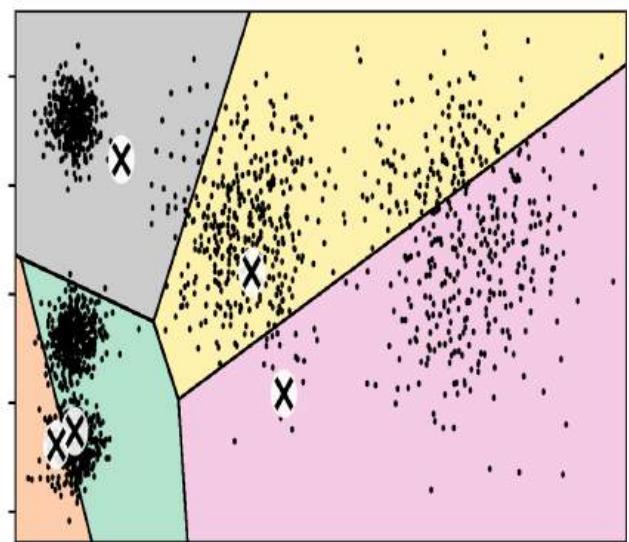
### NOTE

The computational complexity of the algorithm is generally linear with regard to the number of instances  $m$ , the number of clusters  $k$ , and the number of dimensions  $n$ . However, this is only true when the data has a clustering structure. If it does not, then in the worst-case scenario the complexity can increase exponentially with the number of instances. In practice, this rarely happens, and  $k$ -means is generally one of the fastest clustering algorithms.

Update the centroids (initially randomly)



Label the instances



*Figure 8-4. The k-means algorithm*

Although the algorithm is guaranteed to converge, it may not converge to the right solution (i.e., it may converge to a local optimum): whether it does or not depends on the centroid initialization. **Figure 8-5** shows two suboptimal solutions that the algorithm can converge to if you are not lucky with the random initialization step.

## Solution 1

## Solution 2 (with a different random init)

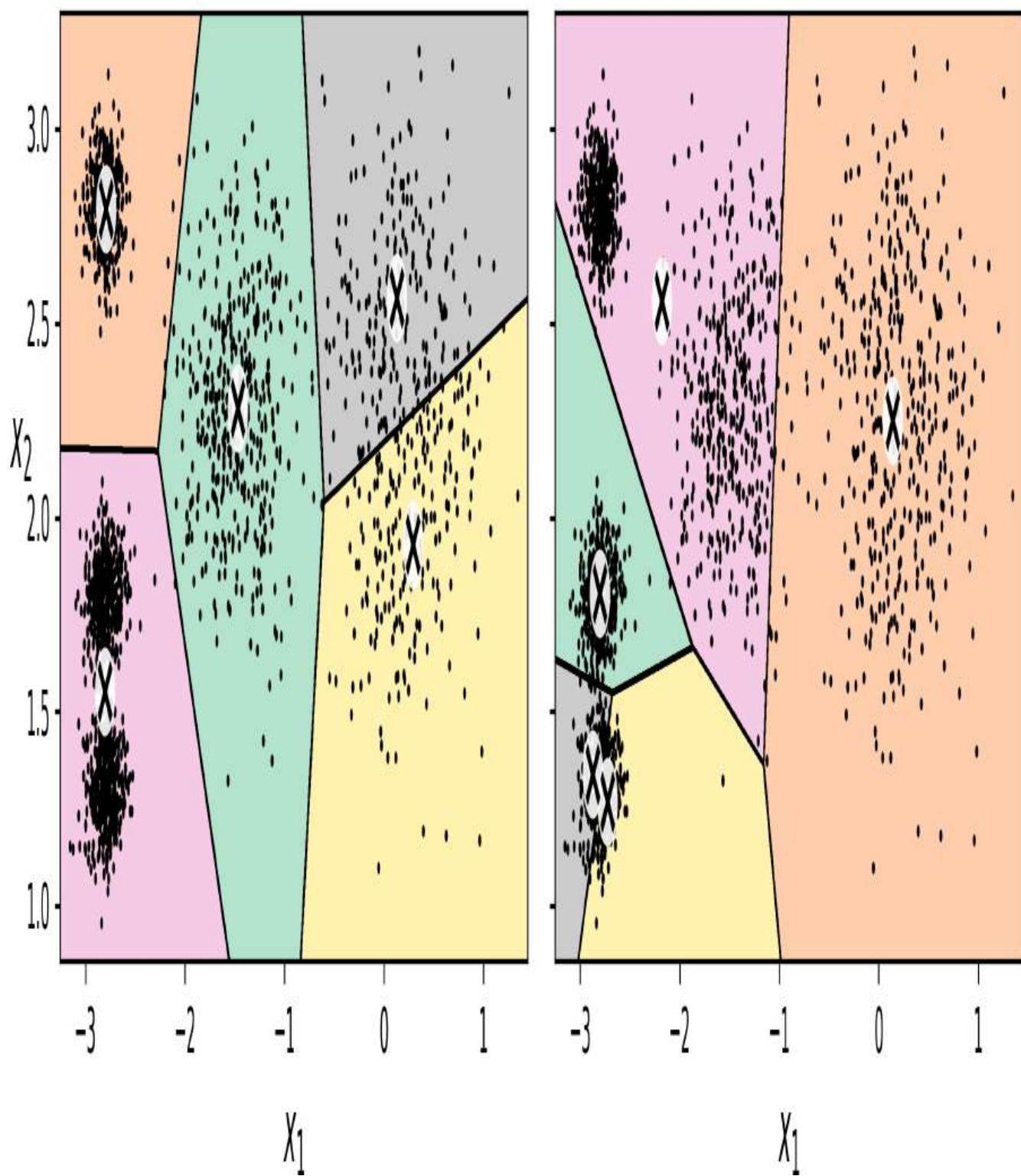


Figure 8-5. Suboptimal solutions due to unlucky centroid initializations

Let's take a look at a few ways you can mitigate this risk by improving the centroid initialization.

## Centroid initialization methods

If you happen to know approximately where the centroids should be (e.g., if you ran another clustering algorithm earlier), then you can set the `init` hyperparameter to a NumPy array containing the list of centroids:

```
good_init = np.array([[-3, 3], [-3, 2], [-3, 1], [-1, 2], [0, 2]])
kmeans = KMeans(n_clusters=5, init=good_init, random_state=42)
kmeans.fit(X)
```

Another solution is to run the algorithm multiple times with different random initializations and keep the best solution. The number of random initializations is controlled by the `n_init` hyperparameter: by default it is equal to 10 when using `init="random"`, which means that the whole algorithm described earlier runs 10 times when you call `fit()`, and Scikit-Learn keeps the best solution. But how exactly does it know which solution is the best? It uses a performance metric! That metric is called the model’s *inertia*, which is defined in [Equation 8-1](#).

*Equation 8-1.* A model’s inertia is the sum of all squared distances between each instance  $\mathbf{x}^{(i)}$  and the closest centroid  $\mathbf{c}^{(i)}$  predicted by the model

$$\text{inertia} = \sum_i \| \mathbf{x}^{(i)} - \mathbf{c}^{(i)} \|^2$$

The inertia is roughly equal to 219.6 for the model on the left in [Figure 8-5](#), 600.4 for the model on the right in [Figure 8-5](#), and only 211.6 for the model in [Figure 8-3](#). The `KMeans` class runs the initialization algorithm `n_init` times and keeps the model with the lowest inertia. In this example, the model in [Figure 8-3](#) will be selected (unless we are very unlucky with `n_init` consecutive random initializations). If you are curious, a model’s inertia is accessible via the `inertia_` instance variable:

```
>>> kmeans.inertia_
211.59853725816836
```

The `score()` method returns the negative inertia (it’s negative because a predictor’s `score()` method must always respect Scikit-Learn’s “greater is better” rule: if a predictor is better than another, its `score()` method should return a greater score):

```
>>> kmeans.score(X)
-211.5985372581684
```

An important improvement to the  $k$ -means algorithm,  $k\text{-means}++$ , was proposed in a [2006 paper](#) by David Arthur and Sergei Vassilvitskii.<sup>3</sup> They introduced a smarter initialization step that tends to select centroids that are distant from one another—this change makes the  $k$ -means algorithm much more likely to locate all important clusters, and less likely to converge to a suboptimal solution (just like spreading out fishing boats increases the chance of locating more schools of fish). The paper showed that the additional computation required for the smarter initialization step is well worth it because it makes it possible to drastically reduce the number of times the algorithm needs to be run to find the optimal solution. The  $k$ -means++ initialization algorithm works like this:

1. Take one centroid  $\mathbf{c}^{(1)}$ , chosen uniformly at random from the dataset.
2. Take a new centroid  $\mathbf{c}^{(i)}$ , choosing an instance  $\mathbf{x}^{(i)}$  with probability  $D(\mathbf{x}^{(i)})^2 / \sum_{j=1}^m D(\mathbf{x}^{(j)})^2$ , where  $D(\mathbf{x}^{(i)})$  is the distance between the instance  $\mathbf{x}^{(i)}$  and the closest centroid that was already chosen. This probability distribution ensures that instances farther away from already chosen centroids are much more likely to be selected as centroids.
3. Repeat the previous step until all  $k$  centroids have been chosen.

When you set `init="k-means++"` (which is the default), the `KMeans` class actually uses a variant of  $k$ -means++ called *greedy k-means++*: instead of sampling a single centroid at each iteration, it samples multiple and picks the best one. When using this algorithm, `n_init` defaults to 1.

## Accelerated k-means and mini-batch k-means

Another improvement to the  $k$ -means algorithm was proposed in a [2003 paper](#) by Charles Elkan.<sup>4</sup> On some large datasets with many clusters, the algorithm can be accelerated by avoiding many unnecessary distance calculations. Elkan achieved this by exploiting the triangle inequality (i.e., that a straight line is always the shortest distance between two points<sup>5</sup>) and by keeping track of lower and upper bounds for distances between instances and centroids. However, Elkan’s algorithm does not always accelerate training, and sometimes it can even slow down training significantly; it depends on the dataset. Still, if you want to give it a try, set `algorithm="elkan"`.

Yet another important variant of the  $k$ -means algorithm was proposed in a [2010 paper](#) by David Sculley.<sup>6</sup> Instead of using the full dataset at each iteration, the algorithm is

capable of using mini-batches, moving the centroids just slightly at each iteration. This speeds up the algorithm and makes it possible to cluster huge datasets that do not fit in memory. Scikit-Learn implements this algorithm in the `MiniBatchKMeans` class, which you can use just like the `KMeans` class:

```
from sklearn.cluster import MiniBatchKMeans  
  
minibatch_kmeans = MiniBatchKMeans(n_clusters=5, random_state=42)  
minibatch_kmeans.fit(X)
```

If the dataset does not fit in memory, the simplest option is to use the `memmap` class, as we did for incremental PCA in [Chapter 7](#). Alternatively, you can pass one mini-batch at a time to the `partial_fit()` method, but this will require much more work, since you will need to perform multiple initializations and select the best one yourself.

## Finding the optimal number of clusters

So far, we've set the number of clusters  $k$  to 5 because it was obvious by looking at the data that this was the correct number of clusters. But in general, it won't be so easy to know how to set  $k$ , and the result might be quite bad if you set it to the wrong value. As you can see in [Figure 8-6](#), for this dataset setting  $k$  to 3 or 8 results in fairly bad models.

You might be thinking that you could just pick the model with the lowest inertia. Unfortunately, it is not that simple. The inertia for  $k=3$  is about 653.2, which is much higher than for  $k=5$  (211.7). But with  $k=8$ , the inertia is just 127.1. The inertia is not a good performance metric when trying to choose  $k$  because it keeps getting lower as we increase  $k$ . Indeed, the more clusters there are, the closer each instance will be to its closest centroid, and therefore the lower the inertia will be. Let's plot the inertia as a function of  $k$ . When we do this, the curve often contains an inflection point called the *elbow* (see [Figure 8-7](#)).

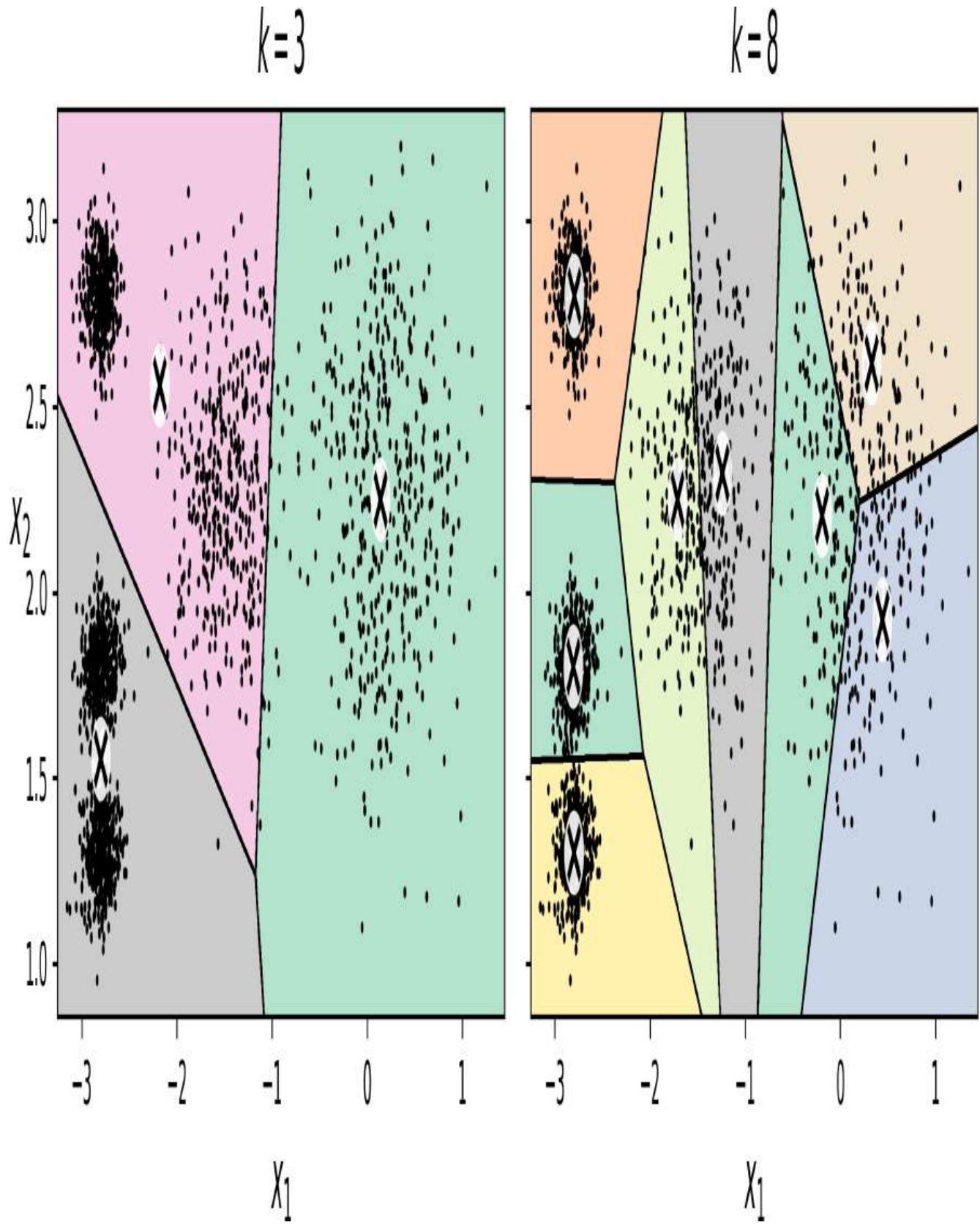


Figure 8-6. Bad choices for the number of clusters: when  $k$  is too small, separate clusters get merged (left), and when  $k$  is too large, some clusters get chopped into multiple pieces (right)

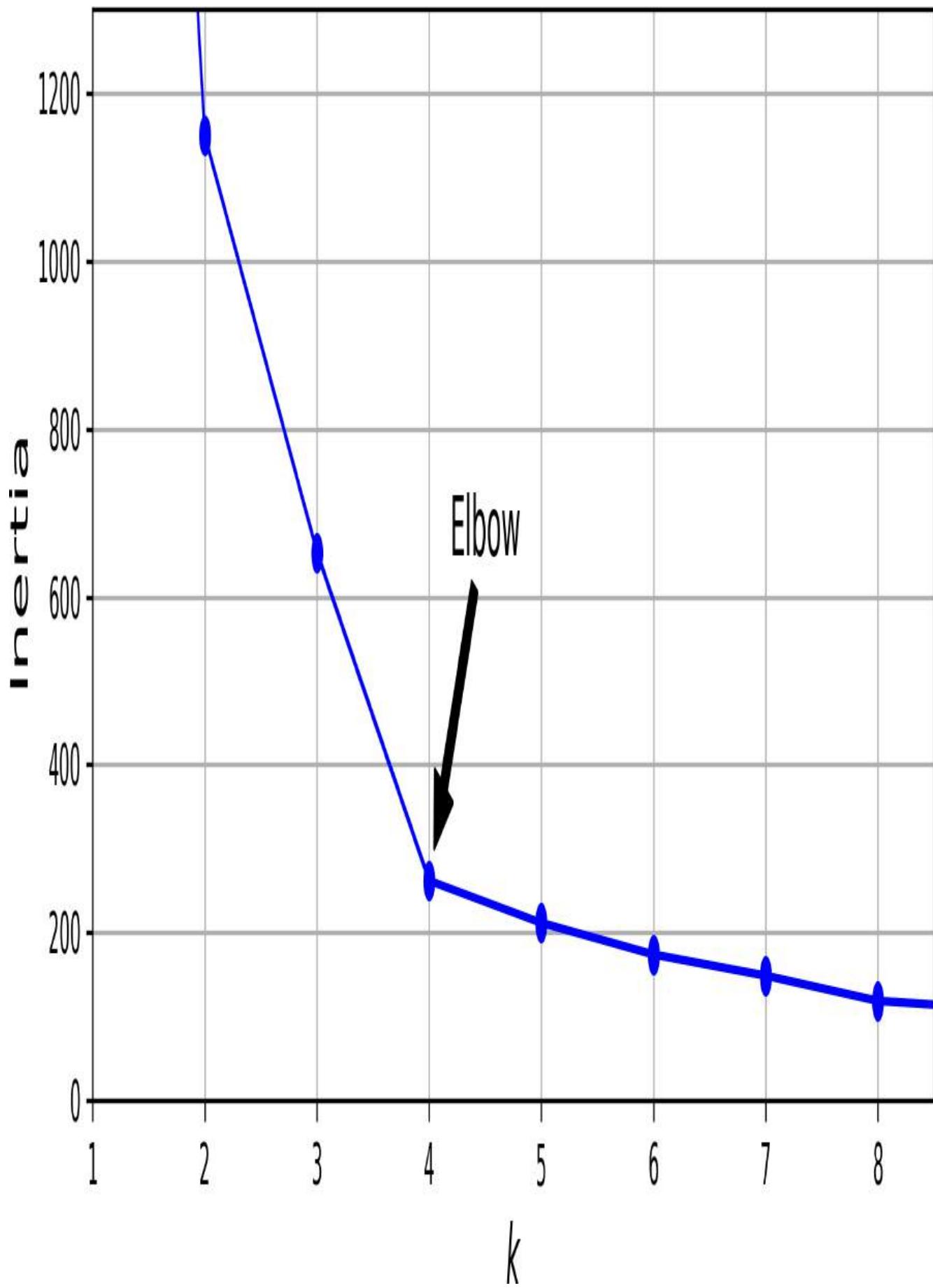


Figure 8-7. Plotting the inertia as a function of the number of clusters  $k$

As you can see, the inertia drops very quickly as we increase  $k$  up to 4, but then it decreases much more slowly as we keep increasing  $k$ . This curve has roughly the shape of an arm, and there is an elbow at  $k = 4$ . So, if we did not know better, we might think 4 was a good choice: any lower value would be dramatic, while any higher value would not help much, and we might just be splitting perfectly good clusters in half for no good reason.

This technique for choosing the best value for the number of clusters is rather coarse. A more precise (but also more computationally expensive) approach is to use the *silhouette score*, which is the mean *silhouette coefficient* over all the instances. An instance's silhouette coefficient is equal to  $(b - a) / \max(a, b)$ , where  $a$  is the mean distance to the other instances in the same cluster (i.e., the mean intra-cluster distance) and  $b$  is the mean nearest-cluster distance (i.e., the mean distance to the instances of the next closest cluster, defined as the one that minimizes  $b$ , excluding the instance's own cluster). The silhouette coefficient can vary between  $-1$  and  $+1$ . A coefficient close to  $+1$  means that the instance is well inside its own cluster and far from other clusters, while a coefficient close to  $0$  means that it is close to a cluster boundary; finally, a coefficient close to  $-1$  means that the instance may have been assigned to the wrong cluster.

To compute the silhouette score, you can use Scikit-Learn's `silhouette_score()` function, giving it all the instances in the dataset and the labels they were assigned:

```
>>> from sklearn.metrics import silhouette_score  
>>> silhouette_score(X, kmeans.labels_)  
0.655517642572828
```

Let's compare the silhouette scores for different numbers of clusters (see [Figure 8-8](#)).

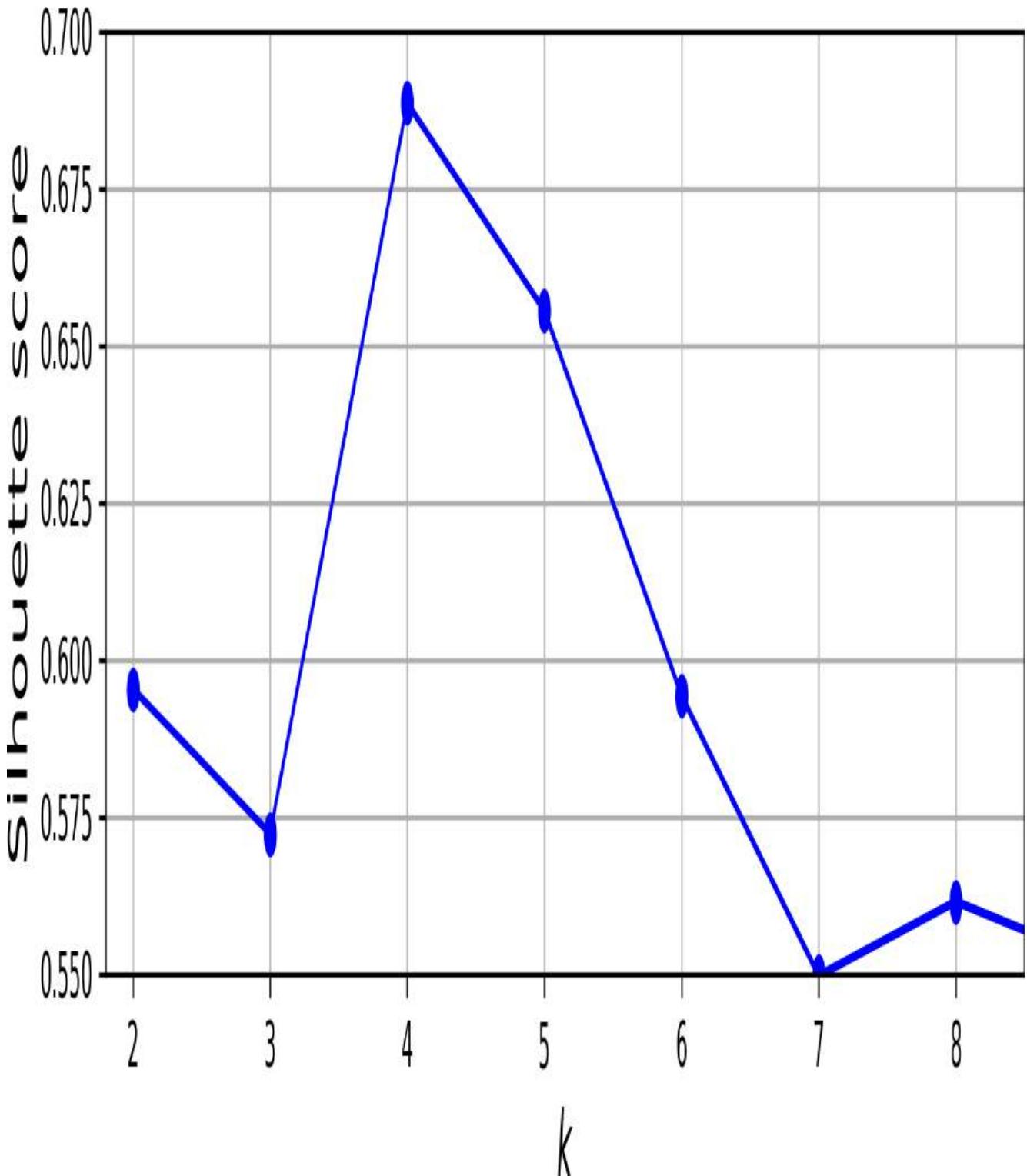
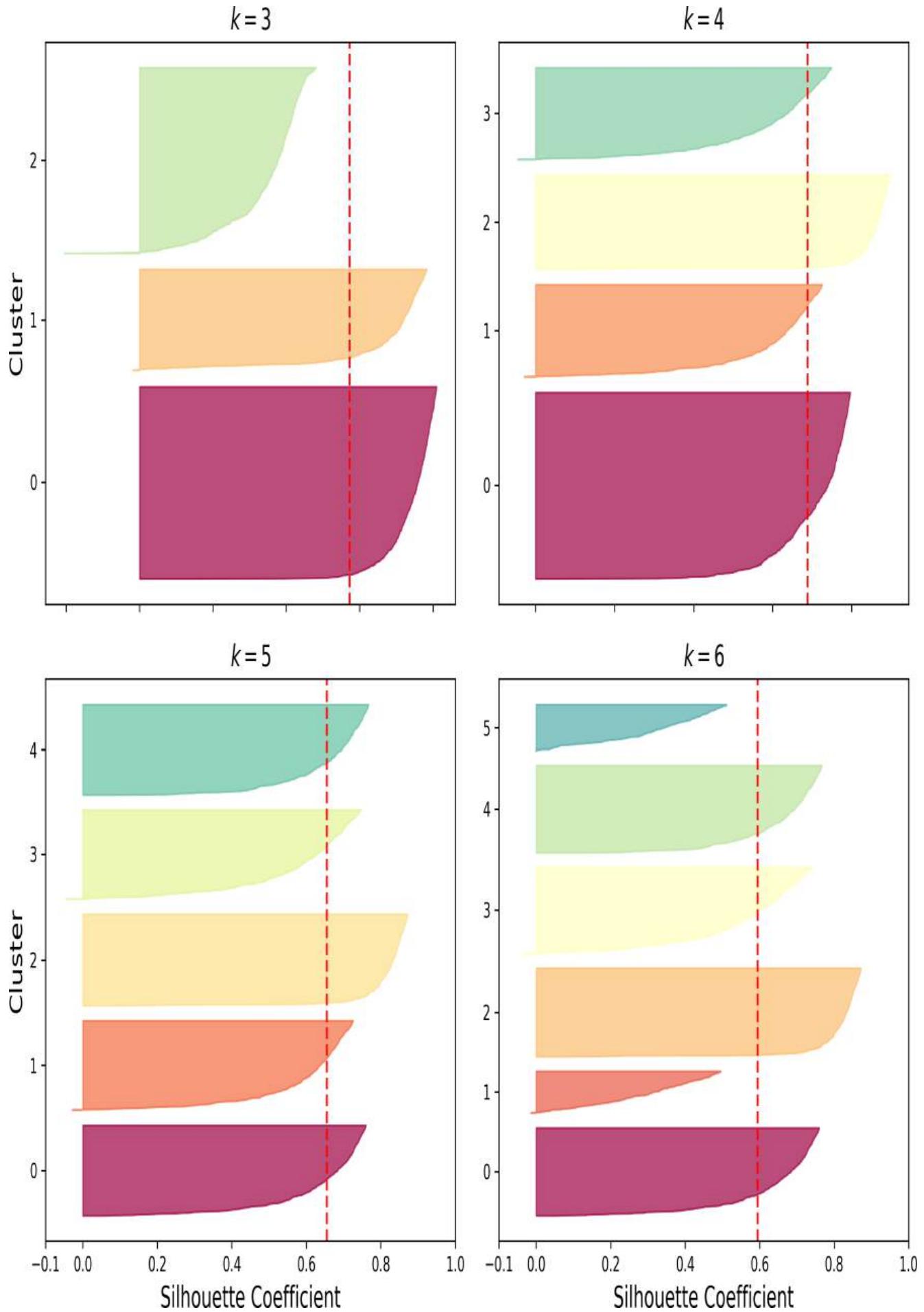


Figure 8-8. Selecting the number of clusters  $k$  using the silhouette score

As you can see, this visualization is much richer than the previous one: although it confirms that  $k = 4$  is a very good choice, it also highlights the fact that  $k = 5$  is quite good as well, and much better than  $k = 6$  or  $7$ . This was not visible when comparing inertias.

An even more informative visualization is obtained when we plot every instance's silhouette coefficient, sorted by the clusters they are assigned to and by the value of the coefficient. This is called a *silhouette diagram* (see [Figure 8-9](#)). Each diagram contains one knife shape per cluster. The shape's height indicates the number of instances in the cluster, and its width represents the sorted silhouette coefficients of the instances in the cluster (wider is better).

The vertical dashed lines represent the mean silhouette score for each number of clusters. When most of the instances in a cluster have a lower coefficient than this score (i.e., if many of the instances stop short of the dashed line, ending to the left of it), then the cluster is rather bad since this means its instances are much too close to other clusters. Here we can see that when  $k = 3$  or  $6$ , we get bad clusters. But when  $k = 4$  or  $5$ , the clusters look pretty good: most instances extend beyond the dashed line, to the right and closer to 1.0. When  $k = 4$ , the cluster at index 0 (at the bottom) is rather big. When  $k = 5$ , all clusters have similar sizes. So, even though the overall silhouette score from  $k = 4$  is slightly greater than for  $k = 5$ , it seems like a good idea to use  $k = 5$  to get clusters of similar sizes.



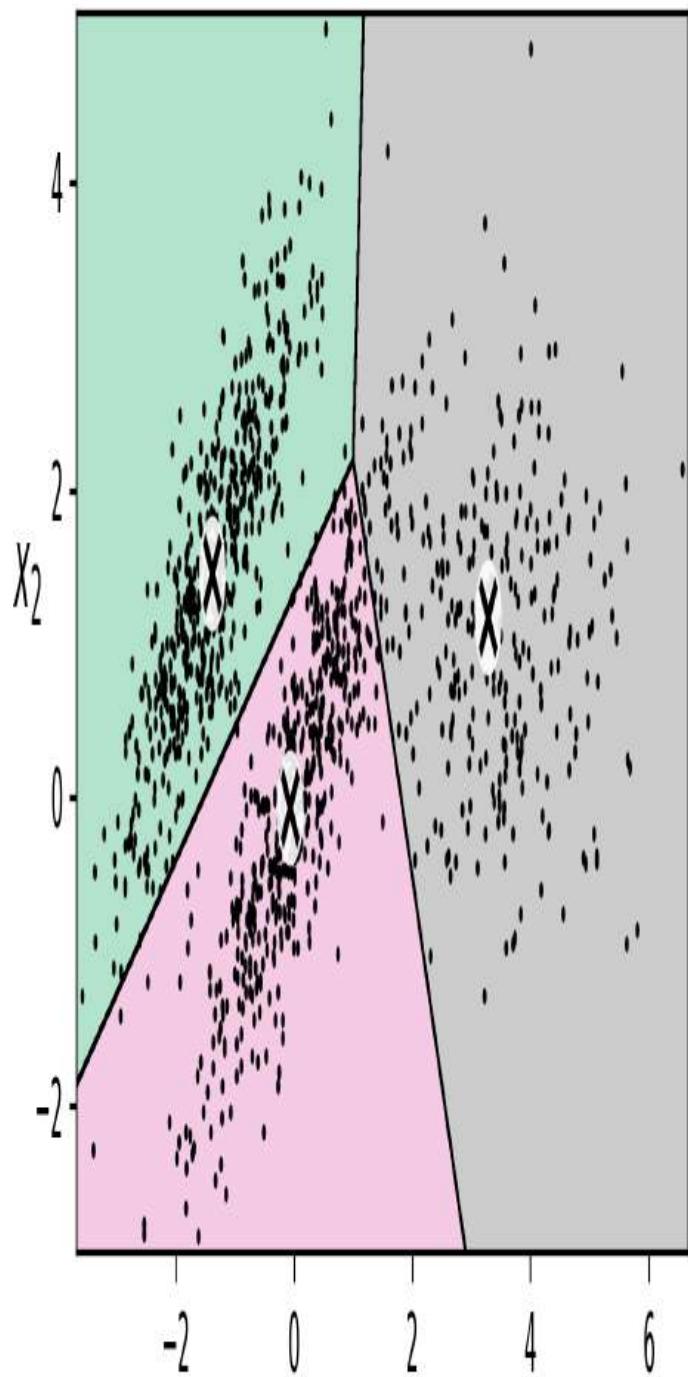
*Figure 8-9. Analyzing the silhouette diagrams for various values of  $k$*

## Limits of k-means

Despite its many merits, most notably being fast and scalable,  $k$ -means is not perfect. As we saw, it is necessary to run the algorithm several times to avoid suboptimal solutions, plus you need to specify the number of clusters, which can be quite a hassle. Moreover,  $k$ -means does not behave very well when the clusters have varying sizes, different densities, or nonspherical shapes. For example, [Figure 8-10](#) shows how  $k$ -means clusters a dataset containing three ellipsoidal clusters of different dimensions, densities, and orientations.

As you can see, neither of these solutions is any good. The solution on the left is better, but it still chops off 25% of the middle cluster and assigns it to the cluster on the right. The solution on the right is just terrible, even though its inertia is lower. So, depending on the data, different clustering algorithms may perform better. On these types of elliptical clusters, Gaussian mixture models work great.

Inertia = 2242.6



Inertia = 2179.6

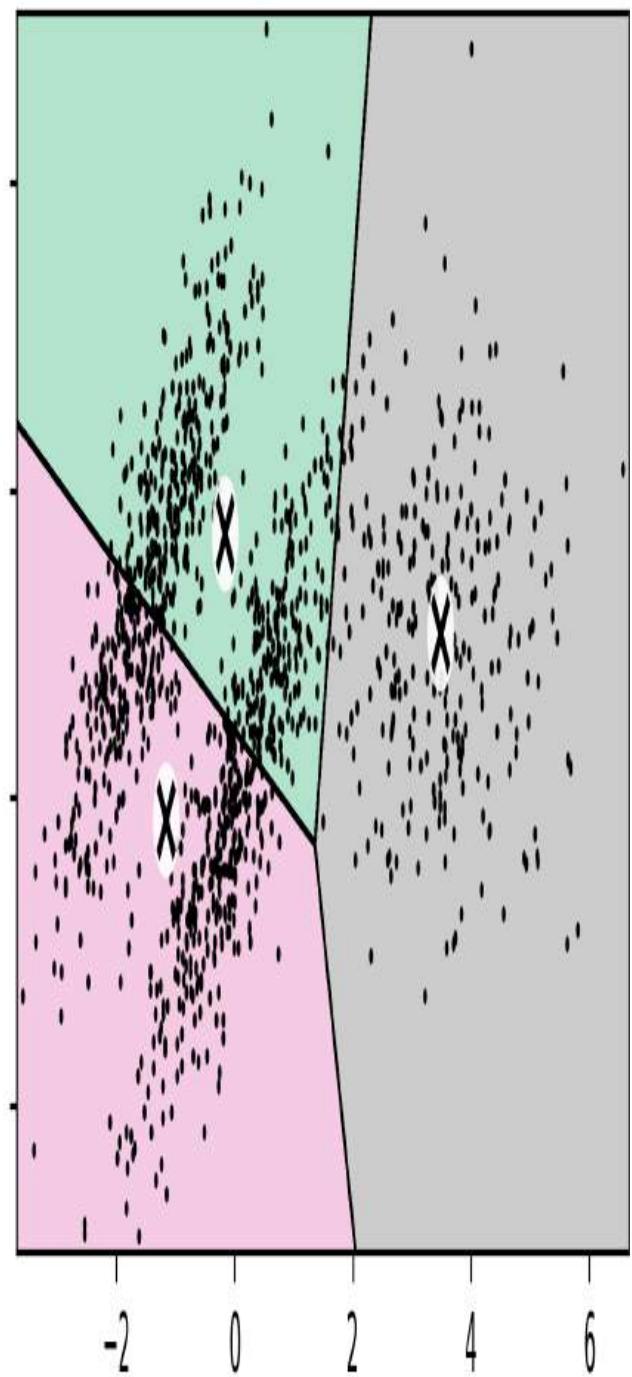


Figure 8-10. k-means fails to cluster these ellipsoidal blobs properly

## TIP

It is important to scale the input features (see [Chapter 2](#)) before you run  $k$ -means, or the clusters may be very stretched and  $k$ -means will perform poorly. Scaling the features does not guarantee that all the clusters will be nice and spherical, but it generally helps  $k$ -means.

Now let's look at a few ways we can benefit from clustering. We will use  $k$ -means, but feel free to experiment with other clustering algorithms.

## Using Clustering for Image Segmentation

*Image segmentation* is the task of partitioning an image into multiple segments. There are several variants:

- In *color segmentation*, pixels with a similar color get assigned to the same segment. This is sufficient in many applications. For example, if you want to analyze satellite images to measure how much total forest area there is in a region, color segmentation may be just fine.
- In *semantic segmentation*, all pixels that are part of the same object type get assigned to the same segment. For example, in a self-driving car's vision system, all pixels that are part of a pedestrian's image might be assigned to the "pedestrian" segment (there would be one segment containing all the pedestrians).
- In *instance segmentation*, all pixels that are part of the same individual object are assigned to the same segment. In this case there would be a different segment for each pedestrian.

The state of the art in semantic or instance segmentation today is achieved using complex architectures based on convolutional neural networks (see [\[Link to Come\]](#)) or transformers (see [\[Link to Come\]](#)). In this chapter we are going to focus on the (much simpler) color segmentation task, using  $k$ -means.

We'll start by importing the Pillow package (successor to the Python Imaging Library, PIL), which we'll then use to load the *ladybug.png* image (see the upper-left image in [Figure 8-11](#)), assuming it's located at `filepath`:

```
>>> import PIL  
>>> image = np.asarray(PIL.Image.open(filepath))
```

```
>>> image.shape  
(533, 800, 3)
```

The image is represented as a 3D array. The first dimension's size is the height; the second is the width; and the third is the number of color channels, in this case red, green, and blue (RGB). In other words, for each pixel there is a 3D vector containing the intensities of red, green, and blue as unsigned 8-bit integers between 0 and 255. Some images may have fewer channels (such as grayscale images, which only have one), and some images may have more channels (such as images with an additional *alpha channel* for transparency, or satellite images, which often contain channels for additional light frequencies (like infrared)).

The following code reshapes the array to get a long list of RGB colors, then it clusters these colors using  $k$ -means with eight clusters. It creates a `segmented_img` array containing the nearest cluster center for each pixel (i.e., the mean color of each pixel's cluster), and lastly it reshapes this array to the original image shape. The third line uses advanced NumPy indexing; for example, if the first 10 labels in `kmeans_.labels_` are equal to 1, then the first 10 colors in `segmented_img` are equal to `kmeans.cluster_centers_[1]`:

```
X = image.reshape(-1, 3)  
kmeans = KMeans(n_clusters=8, random_state=42).fit(X)  
segmented_img = kmeans.cluster_centers_[kmeans.labels_]  
segmented_img = segmented_img.reshape(image.shape)
```

This outputs the image shown in the upper right of [Figure 8-11](#). You can experiment with various numbers of clusters, as shown in the figure. When you use fewer than eight clusters, notice that the ladybug's flashy red color fails to get a cluster of its own: it gets merged with colors from the environment. This is because  $k$ -means prefers clusters of similar sizes. The ladybug is small—much smaller than the rest of the image—so even though its color is flashy,  $k$ -means fails to dedicate a cluster to it.

Original image



10 colors



8 colors



6 colors



4 colors



2 colors



Figure 8-11. Image segmentation using k-means with various numbers of color clusters

That wasn't too hard, was it? Now let's look at another application of clustering.

## Using Clustering for Semi-Supervised Learning

Another use case for clustering is in semi-supervised learning, when we have plenty of unlabeled instances and very few labeled instances. For example, clustering can help choose which additional instances to label (e.g., near the cluster centroids). It can also be used to propagate the most common label in each cluster to the unlabeled instances in that cluster. Let's try these ideas on the digits dataset, which is a simple MNIST-like dataset containing 1,797 grayscale  $8 \times 8$  images representing the digits 0 to 9. First, let's load and split the dataset (it's already shuffled):

```
from sklearn.datasets import load_digits

X_digits, y_digits = load_digits(return_X_y=True)
X_train, y_train = X_digits[:1400], y_digits[:1400]
X_test, y_test = X_digits[1400:], y_digits[1400:]
```

We will pretend we only have labels for 50 instances. To get a baseline performance, let's train a logistic regression model on these 50 labeled instances:

```
from sklearn.linear_model import LogisticRegression

n_labeled = 50
log_reg = LogisticRegression(max_iter=10_000)
log_reg.fit(X_train[:n_labeled], y_train[:n_labeled])
```

We can then measure the accuracy of this model on the test set (note that the test set must be labeled):

```
>>> log_reg.score(X_test, y_test)
0.7581863979848866
```

The model's accuracy is just 75.8%. That's not great: indeed, if you try training the model on the full training set, you will find that it will reach about 90.9% accuracy. Let's see how we can do better. First, let's cluster the training set into 50 clusters. Then, for each cluster, we'll find the image closest to the centroid. We'll call these images the *representative images*:

```
k = 50
kmeans = KMeans(n_clusters=k, random_state=42)
X_digits_dist = kmeans.fit_transform(X_train)
representative_digit_idx = X_digits_dist.argmin(axis=0)
X_representative_digits = X_train[representative_digit_idx]
```

Figure 8-12 shows the 50 representative images.

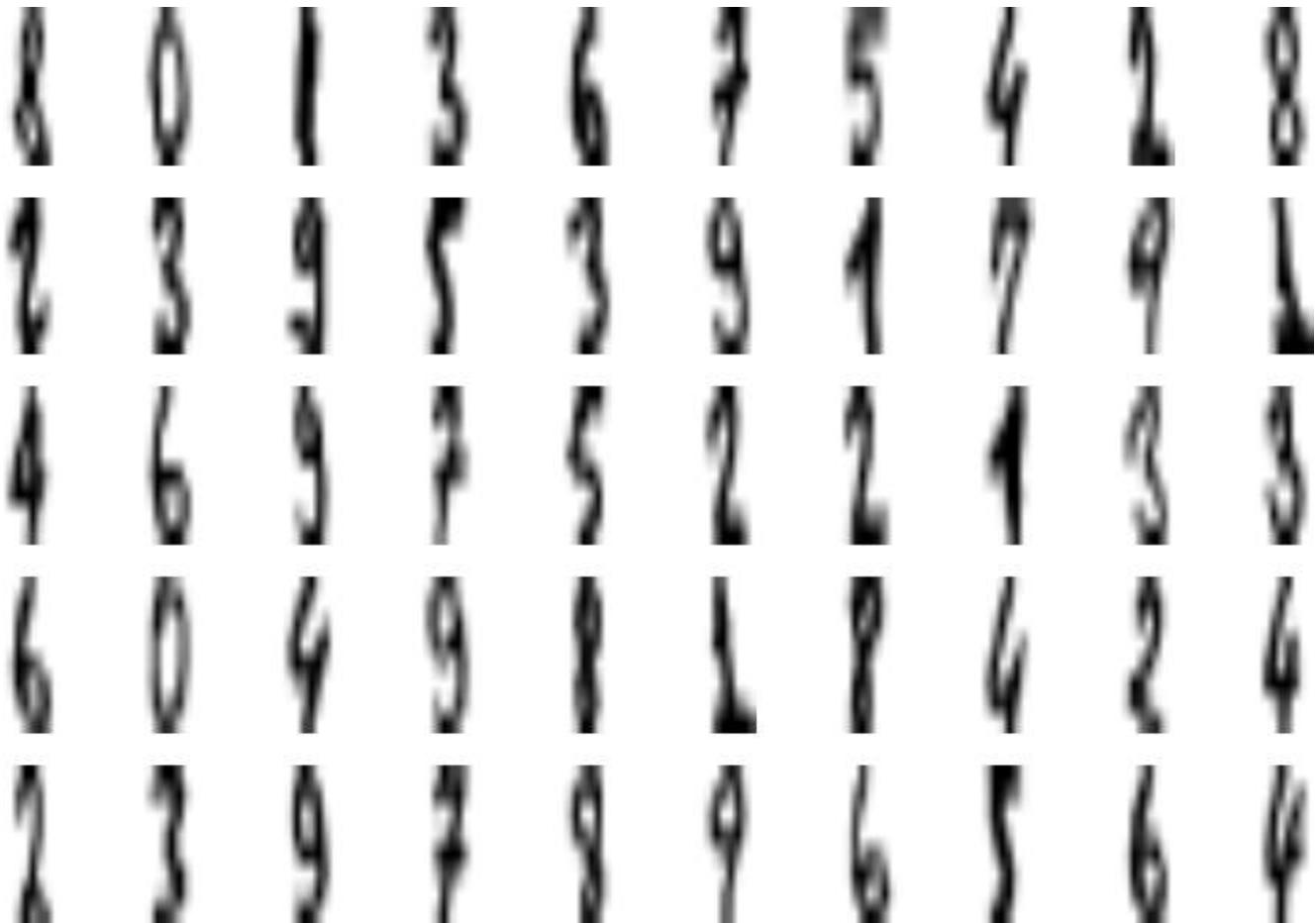


Figure 8-12. Fifty representative digit images (one per cluster)

Let's look at each image and manually label them:

```
y_representative_digits = np.array([8, 0, 1, 3, 6, 7, 5, 4, 2, 8, ..., 6, 4])
```

Now we have a dataset with just 50 labeled instances, but instead of being random instances, each of them is a representative image of its cluster. Let's see if the performance is any better:

```
>>> log_reg = LogisticRegression(max_iter=10_000)
>>> log_reg.fit(X_representative_digits, y_representative_digits)
>>> log_reg.score(X_test, y_test)
0.8312342569269522
```

Wow! We jumped from 75.8% accuracy to 83.1%, although we are still only training the model on 50 instances. Since it is often costly and painful to label instances, especially when it has to be done manually by experts, it is a good idea to label representative instances rather than just random instances.

But perhaps we can go one step further: what if we propagated the labels to all the other instances in the same cluster? This is called *label propagation*:

```
y_train_propagated = np.empty(len(X_train), dtype=np.int64)
for i in range(k):
    y_train_propagated[kmeans.labels_ == i] = y_representative_digits[i]
```

Now let's train the model again and look at its performance:

```
>>> log_reg = LogisticRegression()
>>> log_reg.fit(X_train, y_train_propagated)
>>> log_reg.score(X_test, y_test)
0.871536523929471
```

We got another significant accuracy boost! Let's see if we can do even better by ignoring the 50% of instances that are farthest from their cluster center: this should eliminate some outliers. The following code first computes the distance from each instance to its closest cluster center, then for each cluster it sets the 50% largest distances to  $-1$ . Lastly, it creates a set without these instances marked with a  $-1$  distance:

```
percentile_closest = 50

X_cluster_dist = X_digits_dist[np.arange(len(X_train)), kmeans.labels_]
for i in range(k):
    in_cluster = (kmeans.labels_ == i)
    cluster_dist = X_cluster_dist[in_cluster]
    cutoff_distance = np.percentile(cluster_dist, percentile_closest)
    above_cutoff = (X_cluster_dist > cutoff_distance)
    X_cluster_dist[in_cluster & above_cutoff] = -1

partially_propagated = (X_cluster_dist != -1)
X_train_partially_propagated = X_train[partially_propagated]
y_train_partially_propagated = y_train_propagated[partially_propagated]
```

Now let's train the model again on this partially propagated dataset and see what accuracy we get:

```
>>> log_reg = LogisticRegression(max_iter=10_000)
>>> log_reg.fit(X_train_partially_propagated, y_train_partially_propagated)
>>> log_reg.score(X_test, y_test)
0.8841309823677582
```

Nice! With just 50 labeled instances (only 5 examples per class on average!) we got 88.4% accuracy, pretty close to the performance we got on the fully labeled digits dataset. This is partly thanks to the fact that we dropped some outliers, and partly because the propagated labels are actually pretty good—their accuracy is about 98.9%, as the following code shows:

```
>>> (y_train_partially_propagated == y_train[partially_propagated]).mean()
0.9887798036465638
```

## TIP

Scikit-Learn also offers two classes that can propagate labels automatically: `LabelSpreading` and `LabelPropagation` in the `sklearn.semi_supervised` package. Both classes construct a similarity matrix between all the instances, and iteratively propagate labels from labeled instances to similar unlabeled instances. There's also a very different class called `SelfTrainingClassifier` in the same package: you give it a base classifier (such as a `RandomForestClassifier`) and it trains it on the labeled instances, then uses it to predict labels for the unlabeled samples. It then updates the training set with the labels it is most confident about, and repeats this process of training and labeling until it cannot add labels anymore. These techniques are not magic bullets, but they can occasionally give your model a little boost.

## ACTIVE LEARNING

To continue improving your model and your training set, the next step could be to do a few rounds of *active learning*, which is when a human expert interacts with the learning algorithm, providing labels for specific instances when the algorithm requests them. There are many different strategies for active learning, but one of the most common ones is called *uncertainty sampling*. Here is how it works:

1. The model is trained on the labeled instances gathered so far, and this model is used to make predictions on all the unlabeled instances.
2. The instances for which the model is most uncertain (i.e., where its estimated probability is lowest) are given to the expert for labeling.
3. You iterate this process until the performance improvement stops being worth the labeling effort.

Other active learning strategies include labeling the instances that would result in the largest model change or the largest drop in the model's validation error, or the instances that different models disagree on (e.g., an SVM and a random forest).

Before we move on to Gaussian mixture models, let's take a look at DBSCAN, another popular clustering algorithm that illustrates a very different approach based on local density estimation. This approach allows the algorithm to identify clusters of arbitrary shapes.

## DBSCAN

The *density-based spatial clustering of applications with noise* (DBSCAN) algorithm defines clusters as continuous regions of high density. Here is how it works:

- For each instance, the algorithm counts how many instances are located within a small distance  $\epsilon$  (epsilon) from it. This region is called the instance's  $\epsilon$ -neighborhood.
- If an instance has at least `min_samples` instances in its  $\epsilon$ -neighborhood (including itself), then it is considered a *core instance*. In other words, core instances are those that are located in dense regions.
- All instances in the neighborhood of a core instance belong to the same cluster. This neighborhood may include other core instances; therefore, a long

sequence of neighboring core instances forms a single cluster.

- Any instance that is not a core instance and does not have one in its neighborhood is considered an anomaly.

This algorithm works well if all the clusters are well separated by low-density regions. The DBSCAN class in Scikit-Learn is as simple to use as you might expect. Let's test it on the moons dataset, introduced in [Chapter 5](#):

```
from sklearn.cluster import DBSCAN
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=1000, noise=0.05, random_state=42)
dbSCAN = DBSCAN(eps=0.05, min_samples=5)
dbSCAN.fit(X)
```

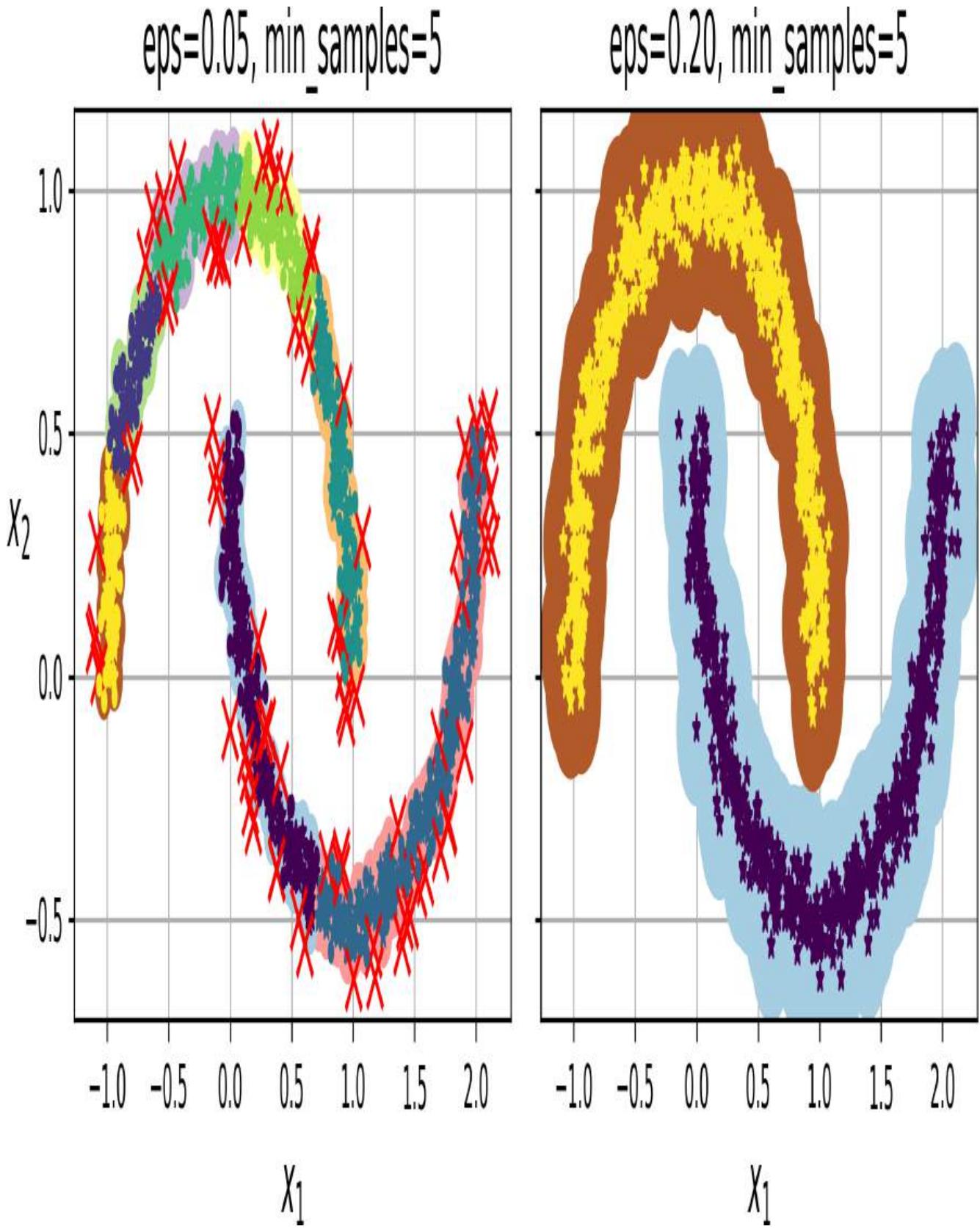
The labels of all the instances are now available in the `labels_` instance variable:

```
>>> dbSCAN.labels_
array([ 0,  2, -1, -1,  1,  0,  0,  0,  2,  5, [...], 3,  3,  4,  2,  6,  3])
```

Notice that some instances have a cluster index equal to  $-1$ , which means that they are considered as anomalies by the algorithm. The indices of the core instances are available in the `core_sample_indices_` instance variable, and the core instances themselves are available in the `components_` instance variable:

```
>>> dbSCAN.core_sample_indices_
array([ 0,  4,  5,  6,  7,  8, 10, 11, [...], 993, 995, 997, 998, 999])
>>> dbSCAN.components_
array([[[-0.02137124,  0.40618608],
       [-0.84192557,  0.53058695],
       [...],
       [ 0.79419406,  0.60777171]])
```

This clustering is represented in the lefthand plot of [Figure 8-13](#). As you can see, it identified quite a lot of anomalies, plus seven different clusters. How disappointing! Fortunately, if we widen each instance's neighborhood by increasing `eps` to 0.2, we get the clustering on the right, which looks perfect. Let's continue with this model.



*Figure 8-13. DBSCAN clustering using two different neighborhood radii*

Surprisingly, the DBSCAN class does not have a `predict()` method, although it has a `fit_predict()` method. In other words, it cannot predict which cluster a new instance belongs to. This decision was made because different classification algorithms can be

better for different tasks, so the authors decided to let the user choose which one to use. Moreover, it's not hard to implement. For example, let's train a `KNeighborsClassifier`:

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=50)
knn.fit(dbSCAN.components_, dbSCAN.labels_[dbSCAN.core_sample_indices_])
```

Now, given a few new instances, we can predict which clusters they most likely belong to and even estimate a probability for each cluster:

```
>>> X_new = np.array([[-0.5, 0], [0, 0.5], [1, -0.1], [2, 1]])
>>> knn.predict(X_new)
array([1, 0, 1, 0])
>>> knn.predict_proba(X_new)
array([[0.18, 0.82],
       [1.  , 0.  ],
       [0.12, 0.88],
       [1.  , 0.  ]])
```

Note that we only trained the classifier on the core instances, but we could also have chosen to train it on all the instances, or all but the anomalies: this choice depends on the final task.

The decision boundary is represented in [Figure 8-14](#) (the crosses represent the four instances in `X_new`). Notice that since there is no anomaly in the training set, the classifier always chooses a cluster, even when that cluster is far away. It is fairly straightforward to introduce a maximum distance, in which case the two instances that are far away from both clusters are classified as anomalies. To do this, use the `kneighbors()` method of the `KNeighborsClassifier`. Given a set of instances, it returns the distances and the indices of the  $k$ -nearest neighbors in the training set (two matrices, each with  $k$  columns):

```
>>> y_dist, y_pred_idx = knn.kneighbors(X_new, n_neighbors=1)
>>> y_pred = dbSCAN.labels_[dbSCAN.core_sample_indices_][y_pred_idx]
>>> y_pred[y_dist > 0.2] = -1
>>> y_pred.ravel()
array([-1,  0,  1, -1])
```

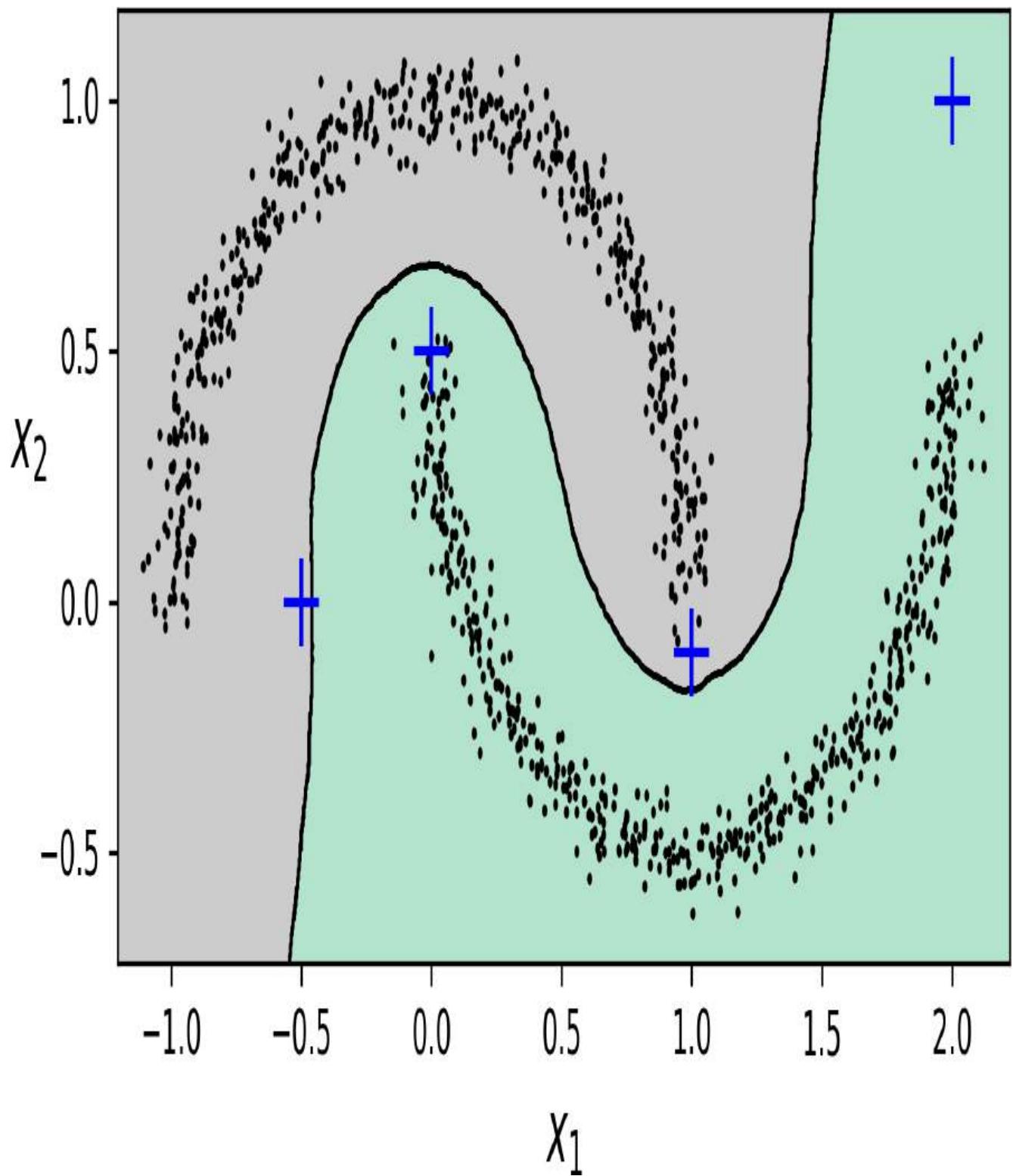


Figure 8-14. Decision boundary between two clusters

In short, DBSCAN is a very simple yet powerful algorithm capable of identifying any number of clusters of any shape. It is robust to outliers, and it has just two hyperparameters (`eps` and `min_samples`). If the density varies significantly across the clusters, however, or if there's no sufficiently low-density region around some clusters,

DBSCAN can struggle to capture all the clusters properly. Moreover, its computational complexity is roughly  $O(m^2n)$ , so it does not scale well to large datasets.

### TIP

You may also want to try *hierarchical DBSCAN* (HDBSCAN), using `sklearn.cluster.HDBSCAN`: it is often better than DBSCAN at finding clusters of varying densities.

## Other Clustering Algorithms

Scikit-Learn implements several more clustering algorithms that you should take a look at. I cannot cover them all in detail here, but here is a brief overview:

### *Agglomerative clustering*

A hierarchy of clusters is built from the bottom up. Think of many tiny bubbles floating on water and gradually attaching to each other until there's one big group of bubbles. Similarly, at each iteration, agglomerative clustering connects the nearest pair of clusters (starting with individual instances). If you drew a tree with a branch for every pair of clusters that merged, you would get a binary tree of clusters, where the leaves are the individual instances. This approach can capture clusters of various shapes; it also produces a flexible and informative cluster tree instead of forcing you to choose a particular cluster scale, and it can be used with any pairwise distance. It can scale nicely to large numbers of instances if you provide a connectivity matrix, which is a sparse  $m \times m$  matrix that indicates which pairs of instances are neighbors (e.g., returned by `sklearn.neighbors.kneighbors_graph()`). Without a connectivity matrix, the algorithm does not scale well to large datasets.

### *BIRCH*

The balanced iterative reducing and clustering using hierarchies (BIRCH) algorithm was designed specifically for very large datasets, and it can be faster than batch  $k$ -means, with similar results, as long as the number of features is not too large (<20). During training, it builds a tree structure containing just enough information to quickly assign each new instance to a cluster, without having to store all the

instances in the tree: this approach allows it to use limited memory while handling huge datasets.

### *Mean-shift*

This algorithm starts by placing a circle centered on each instance; then for each circle it computes the mean of all the instances located within it, and it shifts the circle so that it is centered on the mean. Next, it iterates this mean-shifting step until all the circles stop moving (i.e., until each of them is centered on the mean of the instances it contains). Mean-shift shifts the circles in the direction of higher density, until each of them has found a local density maximum. Finally, all the instances whose circles have settled in the same place (or close enough) are assigned to the same cluster. Mean-shift has some of the same features as DBSCAN, like how it can find any number of clusters of any shape, it has very few hyperparameters (just one—the radius of the circles, called the *bandwidth*), and it relies on local density estimation. But unlike DBSCAN, mean-shift tends to chop clusters into pieces when they have internal density variations. Unfortunately, its computational complexity is  $O(m^2n)$ , so it is not suited for large datasets.

### *Affinity propagation*

In this algorithm, instances repeatedly exchange messages between one another until every instance has elected another instance (or itself) to represent it. These elected instances are called *exemplars*. Each exemplar and all the instances that elected it form one cluster. In real-life politics, you typically want to vote for a candidate whose opinions are similar to yours, but you also want them to win the election, so you might choose a candidate you don't fully agree with, but who is more popular. You typically evaluate popularity through polls. Affinity propagation works in a similar way, and it tends to choose exemplars located near the center of clusters, similar to  $k$ -means. But unlike with  $k$ -means, you don't have to pick a number of clusters ahead of time: it is determined during training. Moreover, affinity propagation can deal nicely with clusters of different sizes. Sadly, this algorithm has a computational complexity of  $O(m^2)$ , so it is not suited for large datasets.

## *Spectral clustering*

This algorithm takes a similarity matrix between the instances and creates a low-dimensional embedding from it (i.e., it reduces the matrix's dimensionality), then it uses another clustering algorithm in this low-dimensional space (Scikit-Learn's implementation uses  $k$ -means). Spectral clustering can capture complex cluster structures, and it can also be used to cut graphs (e.g., to identify clusters of friends on a social network). It does not scale well to large numbers of instances, and it does not behave well when the clusters have very different sizes.

Now let's dive into Gaussian mixture models, which can be used for density estimation, clustering, and anomaly detection.

## Gaussian Mixtures

A *Gaussian mixture model* (GMM) is a probabilistic model that assumes that the instances were generated from a mixture of several Gaussian distributions whose parameters are unknown. All the instances generated from a single Gaussian distribution form a cluster that typically looks like an ellipsoid. Each cluster can have a different ellipsoidal shape, size, density, and orientation, just like in [Figure 8-10](#).<sup>7</sup> When you observe an instance, you know it was generated from one of the Gaussian distributions, but you are not told which one, and you do not know what the parameters of these distributions are.

There are several GMM variants. In the simplest variant, implemented in the `GaussianMixture` class, you must know in advance the number  $k$  of Gaussian distributions. The dataset  $\mathbf{X}$  is assumed to have been generated through the following probabilistic process:

- For each instance, a cluster is picked randomly from among  $k$  clusters. The probability of choosing the  $j^{\text{th}}$  cluster is the cluster's weight  $\phi^{(j)}$ .<sup>8</sup> The index of the cluster chosen for the  $i^{\text{th}}$  instance is noted  $z^{(i)}$ .
- If the  $i^{\text{th}}$  instance was assigned to the  $j^{\text{th}}$  cluster (i.e.,  $z^{(i)} = j$ ), then the location  $\mathbf{x}^{(i)}$  of this instance is sampled randomly from the Gaussian distribution with mean  $\boldsymbol{\mu}^{(j)}$  and covariance matrix  $\boldsymbol{\Sigma}^{(j)}$ . This is noted  $\mathbf{x}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}^{*(j)}, \boldsymbol{\Sigma}^{(j)})$ .

So what can you do with such a model? Well, given the dataset  $\mathbf{X}$ , you typically want to start by estimating the weights  $\phi$  and all the distribution parameters  $\mu^{(1)}$  to  $\mu^{(k)}$  and  $\Sigma^{(1)}$  to  $\Sigma^{(k)}$ . Scikit-Learn's `GaussianMixture` class makes this super easy:

```
from sklearn.mixture import GaussianMixture

gm = GaussianMixture(n_components=3, n_init=10, random_state=42)
gm.fit(X)
```

Let's look at the parameters that the algorithm estimated:

```
>>> gm.weights_
array([0.40005972, 0.20961444, 0.39032584])
>>> gm.means_
array([[[-1.40764129, 1.42712848],
       [ 3.39947665, 1.05931088],
       [ 0.05145113, 0.07534576]]])
>>> gm.covariances_
array([[[[ 0.63478217, 0.72970097],
        [ 0.72970097, 1.16094925]],
       [[ 1.14740131, -0.03271106],
        [-0.03271106, 0.95498333]],
       [[ 0.68825143, 0.79617956],
        [ 0.79617956, 1.21242183]]]])
```

Great, it worked fine! Indeed, two of the three clusters were generated with 500 instances each, while the third cluster only contains 250 instances. So the true cluster weights are 0.4, 0.4, and 0.2, respectively, and that's roughly what the algorithm found (in a different order). Similarly, the true means and covariance matrices are quite close to those found by the algorithm. But how? This class relies on the *expectation-maximization* (EM) algorithm, which has many similarities with the  $k$ -means algorithm: it also initializes the cluster parameters randomly, then it repeats two steps until convergence, first assigning instances to clusters (this is called the *expectation step*) and then updating the clusters (this is called the *maximization step*). Sounds familiar, right? In the context of clustering, you can think of EM as a generalization of  $k$ -means that not only finds the cluster centers ( $\mu^{(1)}$  to  $\mu^{(k)}$ ), but also their size, shape, and orientation ( $\Sigma^{(1)}$  to  $\Sigma^{(k)}$ ), as well as their relative weights ( $\phi^{(1)}$  to  $\phi^{(k)}$ ). Unlike  $k$ -means, though, EM uses soft cluster assignments, not hard assignments. For each instance, during the expectation step, the algorithm estimates the probability that it belongs to each cluster (based on the current cluster parameters). Then, during the maximization

step, each cluster is updated using *all* the instances in the dataset, with each instance weighted by the estimated probability that it belongs to that cluster. These probabilities are called the *responsibilities* of the clusters for the instances. During the maximization step, each cluster's update will mostly be impacted by the instances it is most responsible for.

## WARNING

Unfortunately, just like  $k$ -means, EM can end up converging to poor solutions, so it needs to be run several times, keeping only the best solution. This is why we set `n_init` to 10. Be careful: by default `n_init` is set to 1.

You can check whether or not the algorithm converged and how many iterations it took:

```
>>> gm.converged_
True
>>> gm.n_iter_
4
```

Now that you have an estimate of the location, size, shape, orientation, and relative weight of each cluster, the model can easily assign each instance to the most likely cluster (hard clustering) or estimate the probability that it belongs to a particular cluster (soft clustering). Just use the `predict()` method for hard clustering, or the `predict_proba()` method for soft clustering:

```
>>> gm.predict(X)
array([2, 2, 0, ..., 1, 1, 1])
>>> gm.predict_proba(X).round(3)
array([[0.    , 0.023, 0.977],
       [0.001, 0.016, 0.983],
       [1.    , 0.    , 0.    ],
       ...,
       [0.    , 1.    , 0.    ],
       [0.    , 1.    , 0.    ],
       [0.    , 1.    , 0.    ]])
```

A Gaussian mixture model is a *generative model*, meaning you can sample new instances from it (note that they are ordered by cluster index):

```
>>> X_new, y_new = gm.sample(6)
>>> X_new
array([-2.32491052,  1.04752548],
```

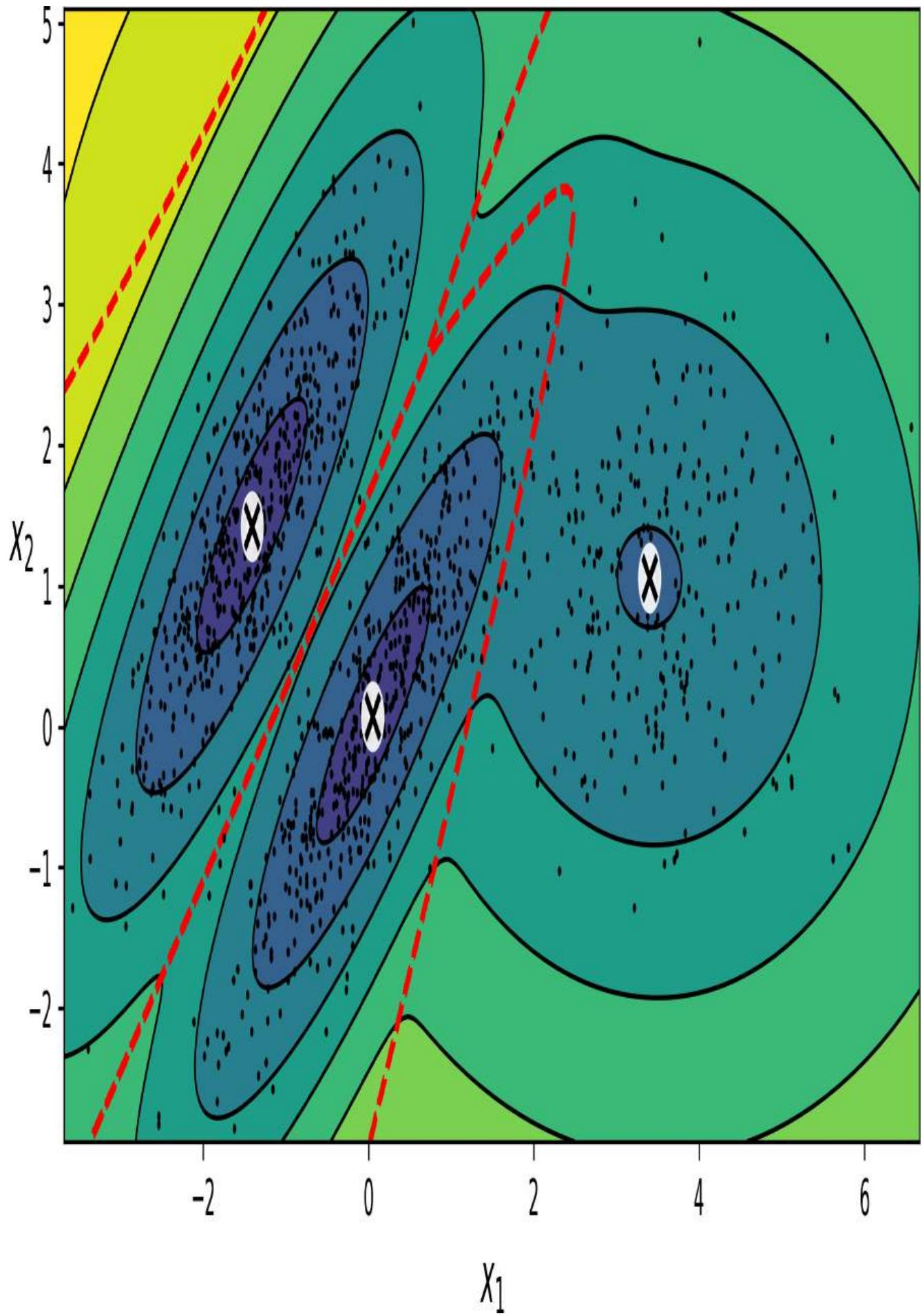
```
[ -1.16654983,  1.62795173],  
[ 1.84860618,  2.07374016],  
[ 3.98304484,  1.49869936],  
[ 3.8163406 ,  0.53038367],  
[ 0.38079484, -0.56239369]])  
  
=> y_new  
array([0, 0, 1, 1, 1, 2])
```

It is also possible to estimate the density of the model at any given location. This is achieved using the `score_samples()` method: for each instance it is given, this method estimates the log of the *probability density function* (PDF) at that location. The greater the score, the higher the density:

```
>>> gm.score_samples(X).round(2)  
array([-2.61, -3.57, -3.33, ..., -3.51, -4.4 , -3.81])
```

If you compute the exponential of these scores, you get the value of the PDF at the location of the given instances. These are not probabilities, but probability *densities*: they can take on any positive value, not just a value between 0 and 1. To estimate the probability that an instance will fall within a particular region, you would have to integrate the PDF over that region (if you do so over the entire space of possible instance locations, the result will be 1).

Figure 8-15 shows the cluster means, the decision boundaries (dashed lines), and the density contours of this model.



*Figure 8-15. Cluster means, decision boundaries, and density contours of a trained Gaussian mixture model*

Nice! The algorithm clearly found an excellent solution. Of course, we made its task easy by generating the data using a set of 2D Gaussian distributions (unfortunately, real-life data is not always so Gaussian and low-dimensional). We also gave the algorithm the correct number of clusters. When there are many dimensions, or many clusters, or few instances, EM can struggle to converge to the optimal solution. You might need to reduce the difficulty of the task by limiting the number of parameters that the algorithm has to learn. One way to do this is to limit the range of shapes and orientations that the clusters can have. This can be achieved by imposing constraints on the covariance matrices. To do this, set the `covariance_type` hyperparameter to one of the following values:

`"spherical"`

All clusters must be spherical, but they can have different diameters (i.e., different variances).

`"diag"`

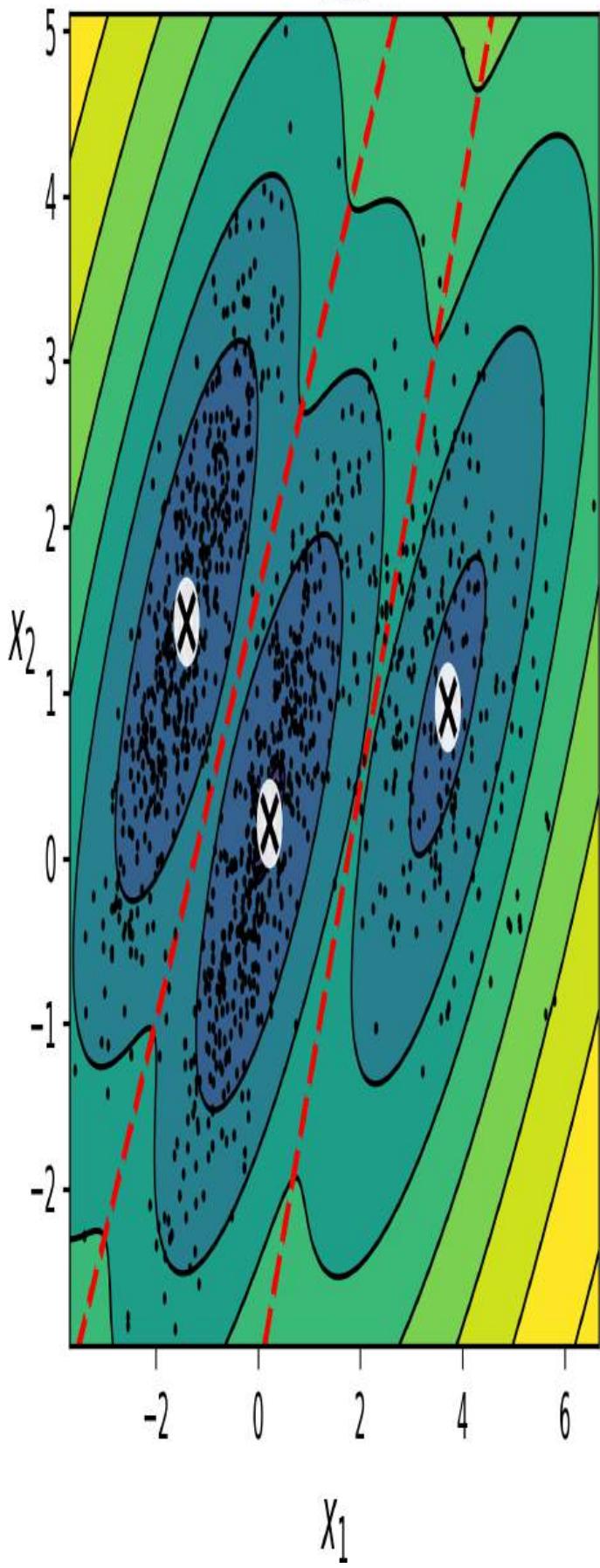
Clusters can take on any ellipsoidal shape of any size, but the ellipsoid's axes must be parallel to the coordinate axes (i.e., the covariance matrices must be diagonal).

`"tied"`

All clusters must have the same ellipsoidal shape, size, and orientation (i.e., all clusters share the same covariance matrix).

By default, `covariance_type` is equal to `"full"`, which means that each cluster can take on any shape, size, and orientation (it has its own unconstrained covariance matrix). [Figure 8-16](#) plots the solutions found by the EM algorithm when `covariance_type` is set to `"tied"` or `"spherical"`.

covariance\_type="tied"



covariance\_type="spherical"

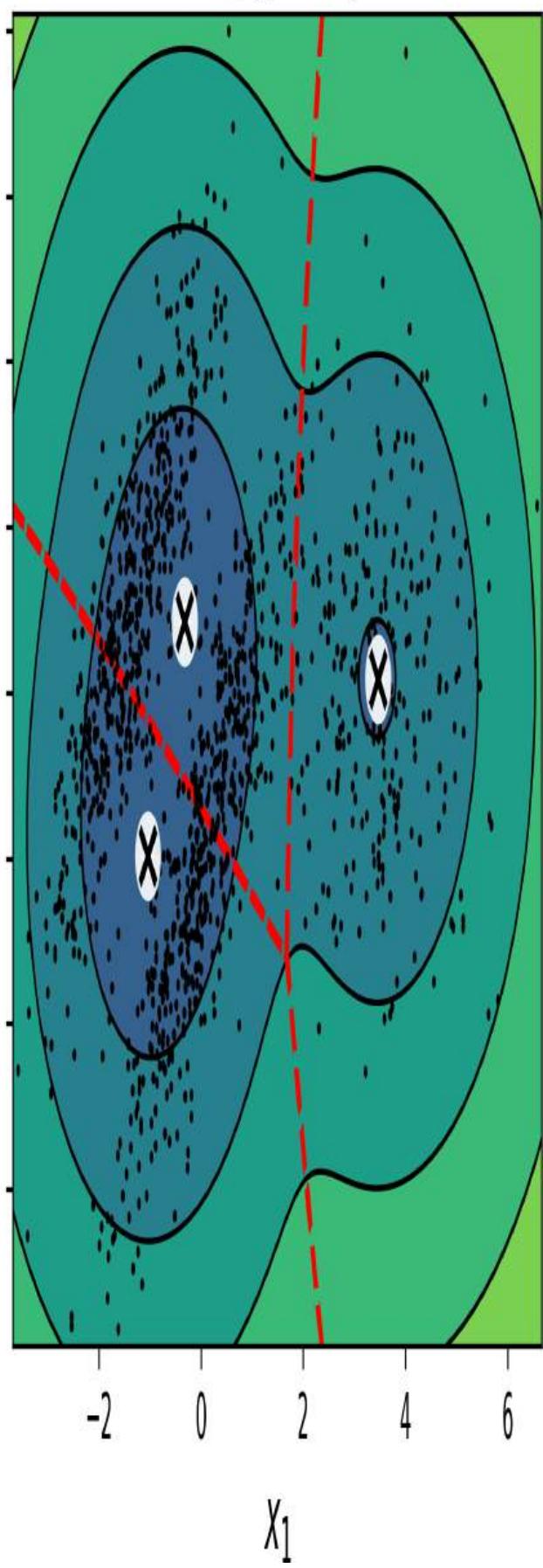


Figure 8-16. Gaussian mixtures for tied clusters (left) and spherical clusters (right)

## NOTE

The computational complexity of training a `GaussianMixture` model depends on the number of instances  $m$ , the number of dimensions  $n$ , the number of clusters  $k$ , and the constraints on the covariance matrices. If `covariance_type` is "spherical" or "diag", it is  $O(kmn)$ , assuming the data has a clustering structure. If `covariance_type` is "tied" or "full", it is  $O(kmn^2 + kn^3)$ , so it will not scale to large numbers of features.

Gaussian mixture models can also be used for anomaly detection. We'll see how in the next section.

## Using Gaussian Mixtures for Anomaly Detection

Using a Gaussian mixture model for anomaly detection is quite simple: any instance located in a low-density region can be considered an anomaly. You must define what density threshold you want to use. For example, in a manufacturing company that tries to detect defective products, the ratio of defective products is usually well known. Say it is equal to 2%. You then set the density threshold to be the value that results in having 2% of the instances located in areas below that threshold density. If you notice that you get too many false positives (i.e., perfectly good products that are flagged as defective), you can lower the threshold. Conversely, if you have too many false negatives (i.e., defective products that the system does not flag as defective), you can increase the threshold. This is the usual precision/recall trade-off (see [Chapter 3](#)). Here is how you would identify the outliers using the second percentile lowest density as the threshold (i.e., approximately 2% of the instances will be flagged as anomalies):

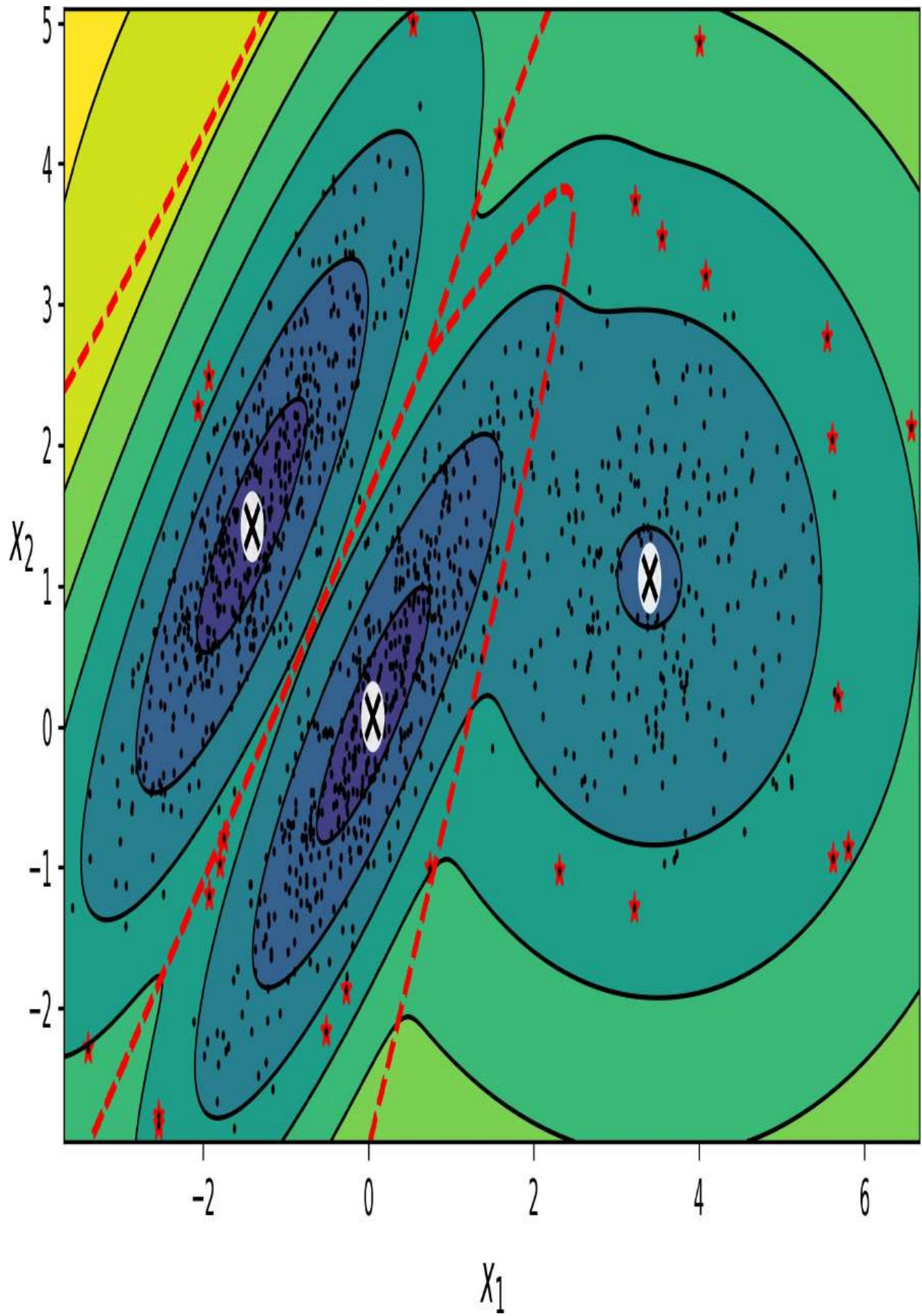
```
densities = gm.score_samples(X)
density_threshold = np.percentile(densities, 2)
anomalies = X[densities < density_threshold]
```

[Figure 8-17](#) represents these anomalies as stars.

A closely related task is *novelty detection*: it differs from anomaly detection in that the algorithm is assumed to be trained on a “clean” dataset, uncontaminated by outliers, whereas anomaly detection does not make this assumption. Indeed, outlier detection is often used to clean up a dataset.

## TIP

Gaussian mixture models try to fit all the data, including the outliers; if you have too many of them this will bias the model's view of “normality”, and some outliers may wrongly be considered as normal. If this happens, you can try to fit the model once, use it to detect and remove the most extreme outliers, then fit the model again on the cleaned-up dataset. Another approach is to use robust covariance estimation methods (see the `EllipticEnvelope` class).



*Figure 8-17. Anomaly detection using a Gaussian mixture model*

Just like  $k$ -means, the `GaussianMixture` algorithm requires you to specify the number of clusters. So how can you find that number?

## Selecting the Number of Clusters

With  $k$ -means, you can use the inertia or the silhouette score to select the appropriate number of clusters. But with Gaussian mixtures, it is not possible to use these metrics because they are not reliable when the clusters are not spherical or have different sizes. Instead, you can try to find the model that minimizes a *theoretical information criterion*, such as the *Bayesian information criterion* (BIC) or the *Akaike information criterion* (AIC), defined in [Equation 8-2](#).

*Equation 8-2. Bayesian information criterion (BIC) and Akaike information criterion (AIC)*

$$\begin{aligned} BIC &= \log(m)p - 2 \log(\widehat{\mathcal{L}}) \\ AIC &= 2p - 2 \log(\widehat{\mathcal{L}}) \end{aligned}$$

In these equations:

- $m$  is the number of instances, as always.
- $p$  is the number of parameters learned by the model.
- $\widehat{\mathcal{L}}$  is the maximized value of the *likelihood function* of the model.

Both the BIC and the AIC penalize models that have more parameters to learn (e.g., more clusters) and reward models that fit the data well. They often end up selecting the same model. When they differ, the model selected by the BIC tends to be simpler (fewer parameters) than the one selected by the AIC, but tends to not fit the data quite as well (this is especially true for larger datasets).

## LIKELIHOOD FUNCTION

The terms “probability” and “likelihood” are often used interchangeably in everyday language, but they have very different meanings in statistics. Given a statistical model with some parameters  $\theta$ , the word “probability” is used to describe how plausible a future outcome  $x$  is (knowing the parameter values  $\theta$ ), while the word “likelihood” is used to describe how plausible a particular set of parameter values  $\theta$  are, after the outcome  $x$  is known.

Consider a 1D mixture model of two Gaussian distributions centered at  $-4$  and  $+1$ . For simplicity, this toy model has a single parameter  $\theta$  that controls the standard deviations of both distributions. The top-left contour plot in [Figure 8-18](#) shows the entire model  $f(x; \theta)$  as a function of both  $x$  and  $\theta$ . To estimate the probability distribution of a future outcome  $x$ , you need to set the model parameter  $\theta$ . For example, if you set  $\theta$  to  $1.3$  (the horizontal line), you get the probability density function  $f(x; \theta=1.3)$  shown in the lower-left plot. Say you want to estimate the probability that  $x$  will fall between  $-2$  and  $+2$ . You must calculate the integral of the PDF on this range (i.e., the surface of the shaded region). But what if you don’t know  $\theta$ , and instead if you have observed a single instance  $x=2.5$  (the vertical line in the upper-left plot)? In this case, you get the likelihood function  $L(\theta|x=2.5)=f(x=2.5; \theta)$ , represented in the upper-right plot.

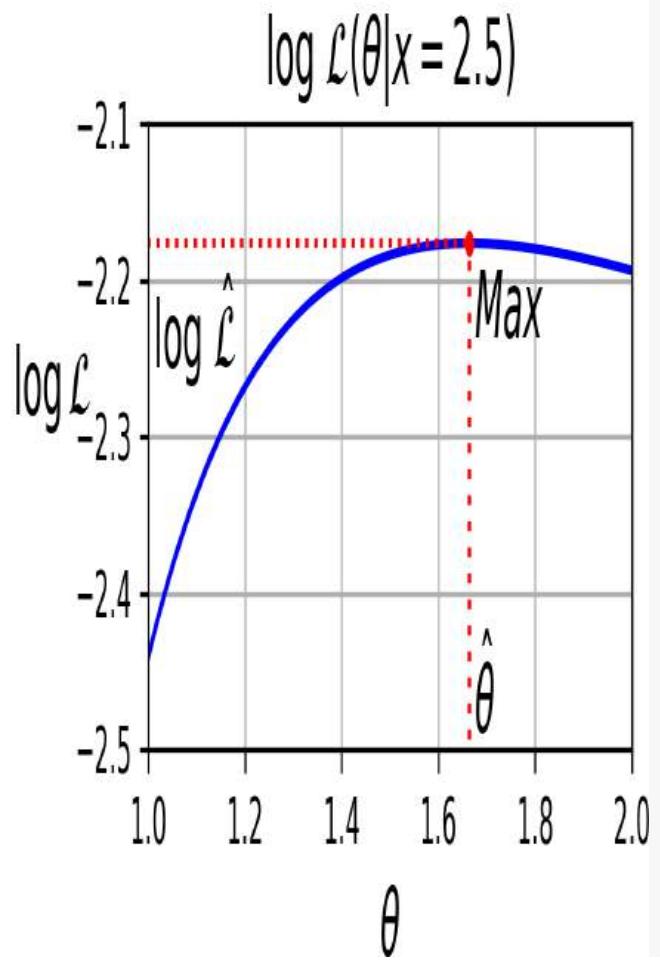
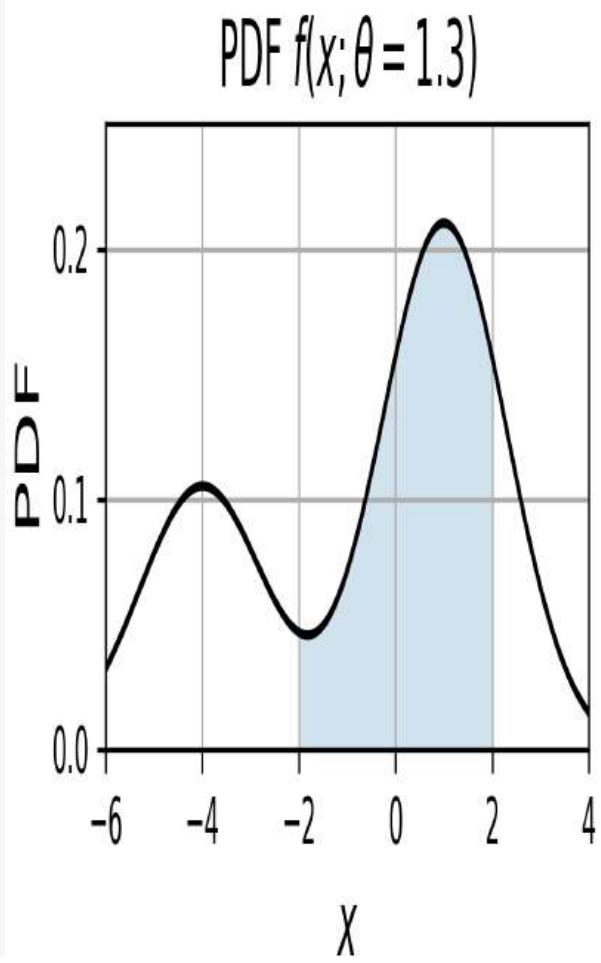
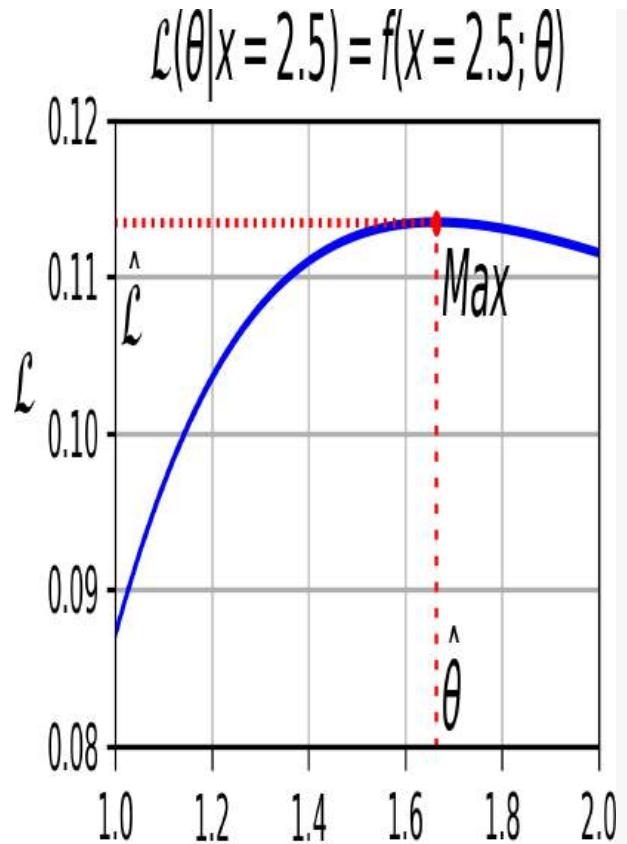
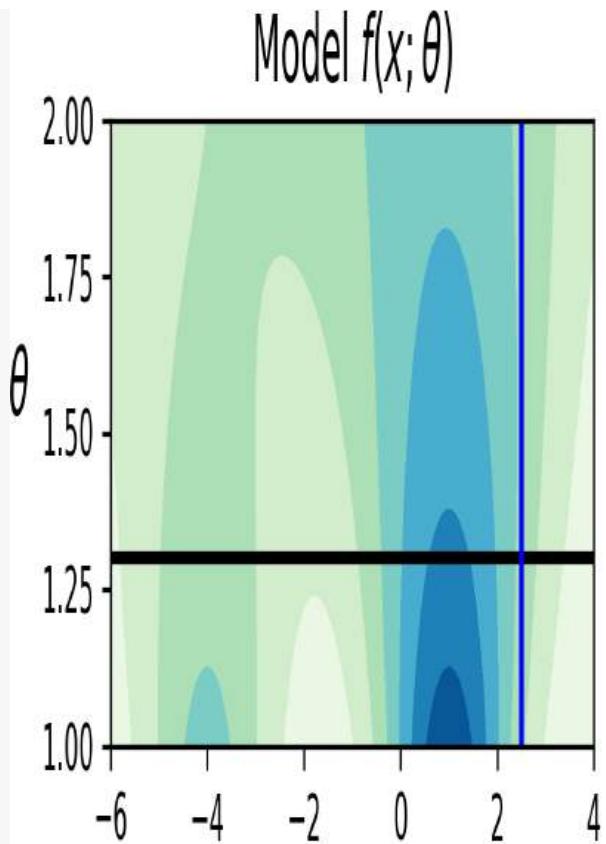


Figure 8-18. A model's parametric function (top left), and some derived functions: a PDF (lower left), a likelihood function (top right), and a log likelihood function (lower right)

In short, the PDF is a function of  $x$  (with  $\theta$  fixed), while the likelihood function is a function of  $\theta$  (with  $x$  fixed). It is important to understand that the likelihood function is *not* a probability distribution: if you integrate a probability distribution over all possible values of  $x$ , you always get 1, but if you integrate the likelihood function over all possible values of  $\theta$  the result can be any positive value.

Given a dataset  $\mathbf{X}$ , a common task is to try to estimate the most likely values for the model parameters. To do this, you must find the values that maximize the likelihood function, given  $\mathbf{X}$ . In this example, if you have observed a single instance  $x=2.5$ , the *maximum likelihood estimate* (MLE) of  $\theta$  is  $\hat{\theta} \approx 1.66$ . If a prior probability distribution  $g$  over  $\theta$  exists, it is possible to take it into account by maximizing  $\mathcal{L}(\theta|x)g(\theta)$  rather than just maximizing  $\mathcal{L}(\theta|x)$ . This is called *maximum a-posteriori* (MAP) estimation. Since MAP constrains the parameter values, you can think of it as a regularized version of MLE.

Notice that maximizing the likelihood function is equivalent to maximizing its logarithm (represented in the lower-right plot in Figure 8-18). Indeed, the logarithm is a strictly increasing function, so if  $\theta$  maximizes the log likelihood, it also maximizes the likelihood. It turns out that it is generally easier to maximize the log likelihood. For example, if you observed several independent instances  $x^{(1)}$  to  $x^{(m)}$ , you would need to find the value of  $\theta$  that maximizes the product of the individual likelihood functions. But it is equivalent, and much simpler, to maximize the sum (not the product) of the log likelihood functions, thanks to the magic of the logarithm which converts products into sums:  $\log(ab) = \log(a) + \log(b)$ .

Once you have estimated  $\hat{\theta}$ , the value of  $\theta$  that maximizes the likelihood function, then you are ready to compute  $\widehat{\mathcal{L}} = \mathcal{L}(\hat{\theta}, \mathbf{X})$ , which is the value used to compute the AIC and BIC; you can think of it as a measure of how well the model fits the data.

To compute the BIC and AIC, call the `bic()` and `aic()` methods:

```
>>> gm.bic(X)
8189.733705221636
>>> gm.aic(X)
8102.508425106598
```

Figure 8-19 shows the BIC for different numbers of clusters  $k$ . As you can see, both the BIC and the AIC are lowest when  $k=3$ , so it is most likely the best choice.

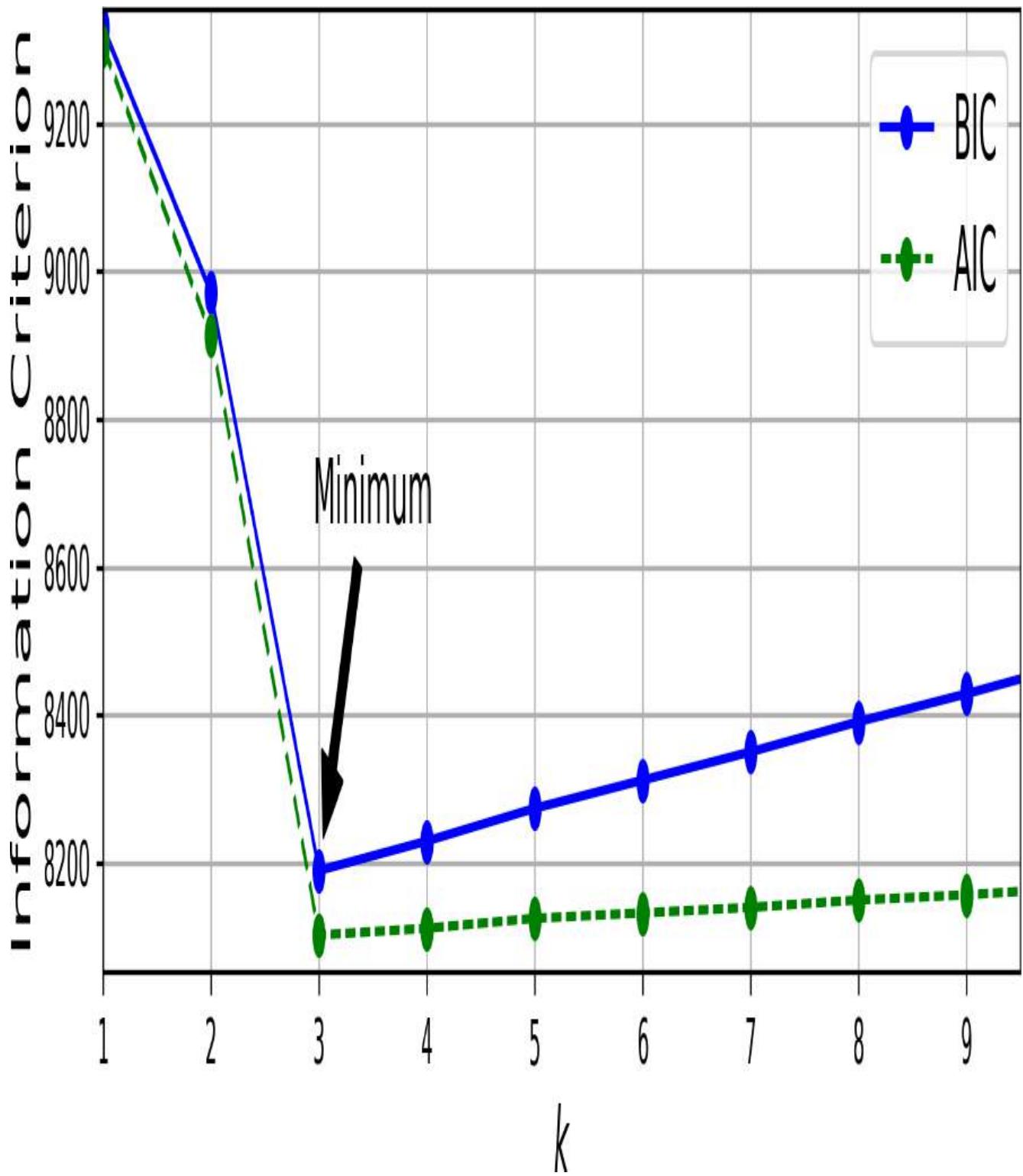


Figure 8-19. AIC and BIC for different numbers of clusters  $k$

## Bayesian Gaussian Mixture Models

Rather than manually searching for the optimal number of clusters, you can use the `BayesianGaussianMixture` class, which is capable of giving weights equal (or close) to zero to unnecessary clusters. Set the number of clusters `n_components` to a value that you have good reason to believe is greater than the optimal number of clusters (this assumes some minimal knowledge about the problem at hand), and the algorithm will eliminate the unnecessary clusters automatically. For example, let's set the number of clusters to 10 and see what happens:

```
>>> from sklearn.mixture import BayesianGaussianMixture
>>> bgm = BayesianGaussianMixture(n_components=10, n_init=10, max_iter=500,
...                                 random_state=42)
...
>>> bgm.fit(X)
>>> bgm.weights_.round(2)
array([0.4 , 0.21, 0.39, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

Perfect: the algorithm automatically detected that only three clusters are needed, and the resulting clusters are almost identical to the ones in [Figure 8-15](#).

A final note about Gaussian mixture models: although they work great on clusters with ellipsoidal shapes, they don't do so well with clusters of very different shapes. For example, let's see what happens if we use a Bayesian Gaussian mixture model to cluster the moons dataset (see [Figure 8-20](#)).

Oops! The algorithm desperately searched for ellipsoids, so it found eight different clusters instead of two. The density estimation is not too bad, so this model could perhaps be used for anomaly detection, but it failed to identify the two moons. To conclude this chapter, let's take a quick look at a few algorithms capable of dealing with arbitrarily shaped clusters.

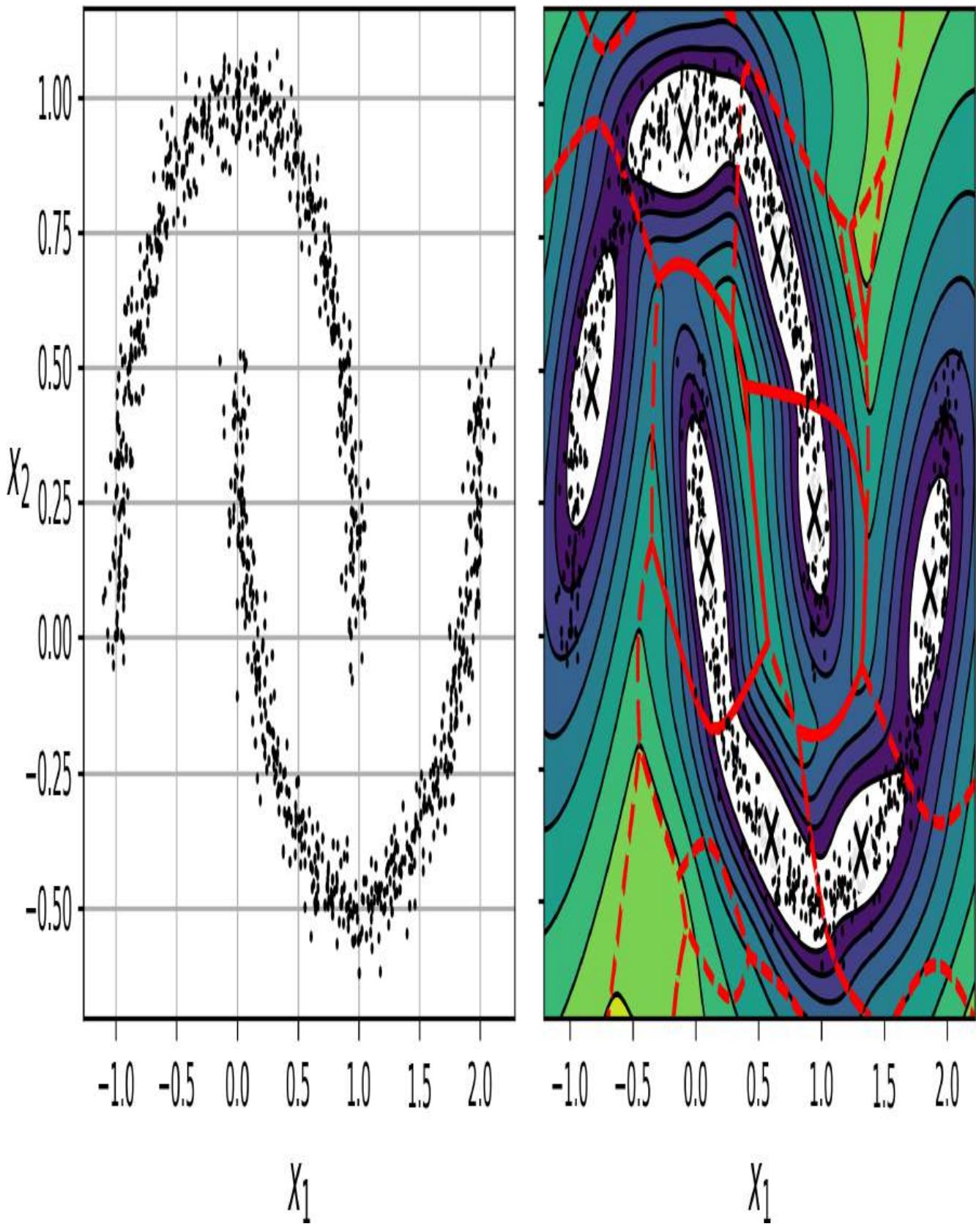


Figure 8-20. Fitting a Gaussian mixture to nonellipsoidal clusters

## Other Algorithms for Anomaly and Novelty Detection

Scikit-Learn implements other algorithms dedicated to anomaly detection or novelty detection:

#### *Fast-MCD (minimum covariance determinant)*

Implemented by the `EllipticEnvelope` class, this algorithm is useful for outlier detection, in particular to clean up a dataset. It assumes that the normal instances (inliers) are generated from a single Gaussian distribution (not a mixture). It also assumes that the dataset is contaminated with outliers that were not generated from this Gaussian distribution. When the algorithm estimates the parameters of the Gaussian distribution (i.e., the shape of the elliptic envelope around the inliers), it is careful to ignore the instances that are most likely outliers. This technique gives a better estimation of the elliptic envelope and thus makes the algorithm better at identifying the outliers.

#### *Isolation forest*

This is an efficient algorithm for outlier detection, especially in high-dimensional datasets. The algorithm builds a random forest in which each decision tree is grown randomly: at each node, it picks a feature randomly, then it picks a random threshold value (between the min and max values) to split the dataset in two. The dataset gradually gets chopped into pieces this way, until all instances end up isolated from the other instances. Anomalies are usually far from other instances, so on average (across all the decision trees) they tend to get isolated in fewer steps than normal instances.

#### *Local outlier factor (LOF)*

This algorithm is also good for outlier detection. It compares the density of instances around a given instance to the density around its neighbors. An anomaly is often more isolated than its  $k$ -nearest neighbors.

#### *One-class SVM*

This algorithm is better suited for novelty detection. Recall that a kernelized SVM classifier separates two classes by first (implicitly) mapping all the instances to a high-dimensional space, then

separating the two classes using a linear SVM classifier within this high-dimensional space (see the online chapter on SVMs at <https://homl.info/svm-p>). Since we just have one class of instances, the one-class SVM algorithm instead tries to separate the instances in high-dimensional space from the origin. In the original space, this will correspond to finding a small region that encompasses all the instances. If a new instance does not fall within this region, it is an anomaly. There are a few hyperparameters to tweak: the usual ones for a kernelized SVM, plus a margin hyperparameter that corresponds to the probability of a new instance being mistakenly considered as novel when it is in fact normal. It works great, especially with high-dimensional datasets, but like all SVMs it does not scale to large datasets.

*PCA and other dimensionality reduction techniques with an `inverse_transform()` method*

If you compare the reconstruction error of a normal instance with the reconstruction error of an anomaly, the latter will usually be much larger. This is a simple and often quite efficient anomaly detection approach (see this chapter's exercises for an example).

## Exercises

1. How would you define clustering? Can you name a few clustering algorithms?
2. What are some of the main applications of clustering algorithms?
3. Describe two techniques to select the right number of clusters when using  $k$ -means.
4. What is label propagation? Why would you implement it, and how?
5. Can you name two clustering algorithms that can scale to large datasets? And two that look for regions of high density?
6. Can you think of a use case where active learning would be useful? How would you implement it?
7. What is the difference between anomaly detection and novelty detection?

8. What is a Gaussian mixture? What tasks can you use it for?
9. Can you name two techniques to find the right number of clusters when using a Gaussian mixture model?
10. The classic Olivetti faces dataset contains 400 grayscale  $64 \times 64$ -pixel images of faces. Each image is flattened to a 1D vector of size 4,096. Forty different people were photographed (10 times each), and the usual task is to train a model that can predict which person is represented in each picture. Load the dataset using the `sklearn.datasets.fetch_olivetti_faces()` function, then split it into a training set, a validation set, and a test set (note that the dataset is already scaled between 0 and 1). Since the dataset is quite small, you will probably want to use stratified sampling to ensure that there are the same number of images per person in each set. Next, cluster the images using  $k$ -means, and ensure that you have a good number of clusters (using one of the techniques discussed in this chapter). Visualize the clusters: do you see similar faces in each cluster?
11. Continuing with the Olivetti faces dataset, train a classifier to predict which person is represented in each picture, and evaluate it on the validation set. Next, use  $k$ -means as a dimensionality reduction tool, and train a classifier on the reduced set. Search for the number of clusters that allows the classifier to get the best performance: what performance can you reach? What if you append the features from the reduced set to the original features (again, searching for the best number of clusters)?
12. Train a Gaussian mixture model on the Olivetti faces dataset. To speed up the algorithm, you should probably reduce the dataset's dimensionality (e.g., use PCA, preserving 99% of the variance). Use the model to generate some new faces (using the `sample()` method), and visualize them (if you used PCA, you will need to use its `inverse_transform()` method). Try to modify some images (e.g., rotate, flip, darken) and see if the model can detect the anomalies (i.e., compare the output of the `score_samples()` method for normal images and for anomalies).
13. Some dimensionality reduction techniques can also be used for anomaly detection. For example, take the Olivetti faces dataset and reduce it with PCA, preserving 99% of the variance. Then compute the reconstruction error for each image. Next, take some of the modified images you built in the previous exercise and look at their reconstruction error: notice how much larger it is. If

you plot a reconstructed image, you will see why: it tries to reconstruct a normal face.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.

---

- <sup>1</sup> If you are not familiar with probability theory, I highly recommend the free online classes by Khan Academy.
- <sup>2</sup> Stuart P. Lloyd, “Least Squares Quantization in PCM”, *IEEE Transactions on Information Theory* 28, no. 2 (1982): 129–137.
- <sup>3</sup> David Arthur and Sergei Vassilvitskii, “k-Means++: The Advantages of Careful Seeding”, *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms* (2007): 1027–1035.
- <sup>4</sup> Charles Elkan, “Using the Triangle Inequality to Accelerate k-Means”, *Proceedings of the 20th International Conference on Machine Learning* (2003): 147–153.
- <sup>5</sup> The triangle inequality is  $AC \leq AB + BC$ , where A, B and C are three points and AB, AC, and BC are the distances between these points.
- <sup>6</sup> David Sculley, “Web-Scale K-Means Clustering”, *Proceedings of the 19th International Conference on World Wide Web* (2010): 1177–1178.
- <sup>7</sup> In contrast, as we saw earlier,  $k$ -means implicitly assumes that clusters all have a similar size and density, and are all roughly round.
- <sup>8</sup> Phi ( $\phi$  or  $\varphi$ ) is the 21st letter of the Greek alphabet.

# **Part II. Neural Networks and Deep Learning**

---

# Chapter 9. Introduction to Artificial Neural Networks

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 9th chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

Birds inspired us to fly, burdock plants inspired Velcro, and nature has inspired countless more inventions. It seems only logical, then, to look at the brain’s architecture for inspiration on how to build an intelligent machine. This is the logic that sparked *artificial neural networks* (ANNs), machine learning models inspired by the networks of biological neurons found in our brains. However, although planes were inspired by birds, they don’t have to flap their wings to fly. Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether (e.g., by saying “units” rather than “neurons”), lest we restrict our creativity to biologically plausible systems.<sup>1</sup>

ANNs are at the very core of deep learning. They are versatile, powerful, and scalable, making them ideal to tackle large and highly complex machine learning tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple’s Siri or Google Assistant) and chatbots (e.g., ChatGPT or Claude), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning how proteins fold (DeepMind’s AlphaFold).

This chapter introduces artificial neural networks, starting with a quick tour of the very first ANN architectures and leading up to multilayer perceptrons (MLPs), which are heavily used today (many other architectures will be explored in the following chapters). In this chapter, we will implement simple MLPs using Scikit-Learn to get our feet wet, and in the next chapter we will switch to PyTorch, as it is a much more flexible and efficient library for neural nets.

Now let's go back in time to the origins of artificial neural networks.

## From Biological to Artificial Neurons

Surprisingly, ANNs have been around for quite a while: they were first introduced back in 1943 by the neurophysiologist Warren McCulloch and the mathematician Walter Pitts. In their [landmark paper<sup>2</sup>](#) “A Logical Calculus of Ideas Immanent in Nervous Activity”, McCulloch and Pitts presented a simplified computational model of how biological neurons might work together in animal brains to perform complex computations using *propositional logic*. This was the first artificial neural network architecture. Since then many other architectures have been invented, as you will see.

The early successes of ANNs led to the widespread belief that we would soon be conversing with truly intelligent machines. When it became clear in the 1960s that this promise would go unfulfilled (at least for quite a while), funding flew elsewhere, and ANNs entered a long winter. In the early 1980s, new architectures were invented and better training techniques were developed, sparking a revival of interest in *connectionism*, the study of neural networks. But progress was slow, and by the 1990s other powerful machine learning techniques had been invented, such as support vector machines. These techniques seemed to offer better results and stronger theoretical foundations than ANNs, so once again the study of neural networks was put on hold.

We are now witnessing yet another wave of interest in ANNs. Will this wave die out like the previous ones did? Well, here are a few good reasons to believe that this time is different and that the renewed interest in ANNs will have a much more profound impact on our lives:

- There is now a huge quantity of data available to train neural networks, and ANNs frequently outperform other ML techniques on very large and complex problems.
- The tremendous increase in computing power since the 1990s now makes it possible to train large neural networks in a reasonable amount of time. This is in part due to Moore’s law (the number of components in integrated circuits has doubled about every 2 years over the last 50 years), but also thanks to the gaming industry, which has stimulated the production of powerful *graphical processing units* (GPUs) by the millions: GPU cards were initially designed to accelerate graphics, but it turns out that neural networks perform similar computations (such as large matrix multiplications), so they can also be

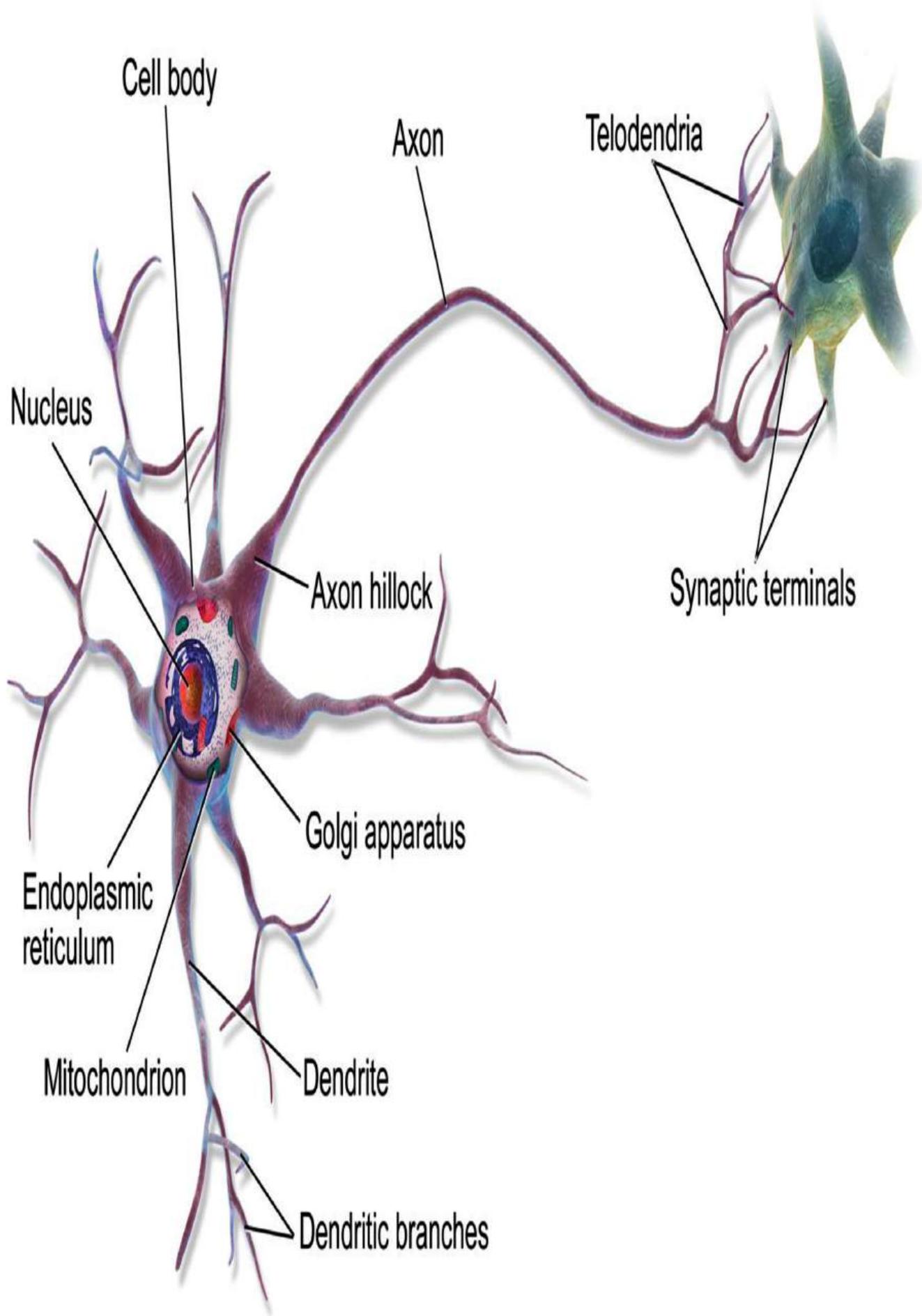
accelerated using GPUs. Moreover, cloud platforms have made this power accessible to everyone.

- The training algorithms have been improved. To be fair they are only slightly different from the ones used in the 1990s, but these relatively small tweaks have had a huge positive impact.
- Some theoretical limitations of ANNs have turned out to be benign in practice. For example, many people thought that ANN training algorithms were doomed because they were likely to get stuck in local optima, but it turns out that this is not a big problem in practice, especially for larger neural networks: the local optima often perform almost as well as the global optimum.
- The invention of the Transformer architecture in 2017 (see [Link to Come]) has been a game changer: it can process and generate all sorts of data (e.g., text, images, audio) unlike earlier more specialized architectures, and it performs great across a wide variety of tasks from robotics to protein folding. Moreover, it scales rather well which has made it possible to train very large *foundation models* that can be reused across many different tasks, possibly with a bit of fine-tuning (that's transfer learning), or just by prompting the model in the right way (that's *in-context learning*, or ICL)—for instance, you can give it a few examples of the task at hand (that's *few-shot learning*, or FSL), or ask it to reason step by step (that's *chain-of-thought* prompting, or CoT). It's a new world!
- ANNs seem to have entered a virtuous circle of funding and progress. Amazing products based on ANNs regularly make the headline news, which pulls more and more attention and funding toward them, resulting in more and more progress and even more amazing products. AI is no longer just powering products in the shadows: since chatbots such as ChatGPT were released, the general public is now directly interacting daily with AI assistants, and the big tech companies are competing fiercely to grab this gigantic market: the pace of innovation is wild.

## Biological Neurons

Before we discuss artificial neurons, let's take a quick look at a biological neuron (represented in [Figure 9-1](#)). It is an unusual-looking cell mostly found in animal brains. It's composed of a *cell body* containing the nucleus and most of the cell's complex components, many branching extensions called *dendrites*, plus one very long extension

called the *axon*. The axon's length may be just a few times longer than the cell body, or up to tens of thousands of times longer. Near its extremity the axon splits off into many branches called *telodendria*, and at the tip of these branches are minuscule structures called *synaptic terminals* (or simply *synapses*), which are connected to the dendrites or cell bodies of other neurons.<sup>3</sup> Biological neurons produce short electrical impulses called *action potentials* (APs, or just *signals*), which travel along the axons and make the synapses release chemical signals called *neurotransmitters*. When a neuron receives a sufficient amount of these neurotransmitters within a few milliseconds, it fires its own electrical impulses (actually, it depends on the neurotransmitters, as some of them inhibit the neuron from firing).



*Figure 9-1. A biological neuron<sup>4</sup>*

Thus, individual biological neurons seem to behave in a simple way, but they're organized in a vast network of billions, with each neuron typically connected to thousands of other neurons. Highly complex computations can be performed by a network of fairly simple neurons, much like a complex anthill can emerge from the combined efforts of simple ants. The architecture of biological neural networks (BNNs)<sup>5</sup> is the subject of active research, but some parts of the brain have been mapped. These efforts show that neurons are often organized in consecutive layers, especially in the cerebral cortex (the outer layer of the brain), as shown in [Figure 9-2](#).

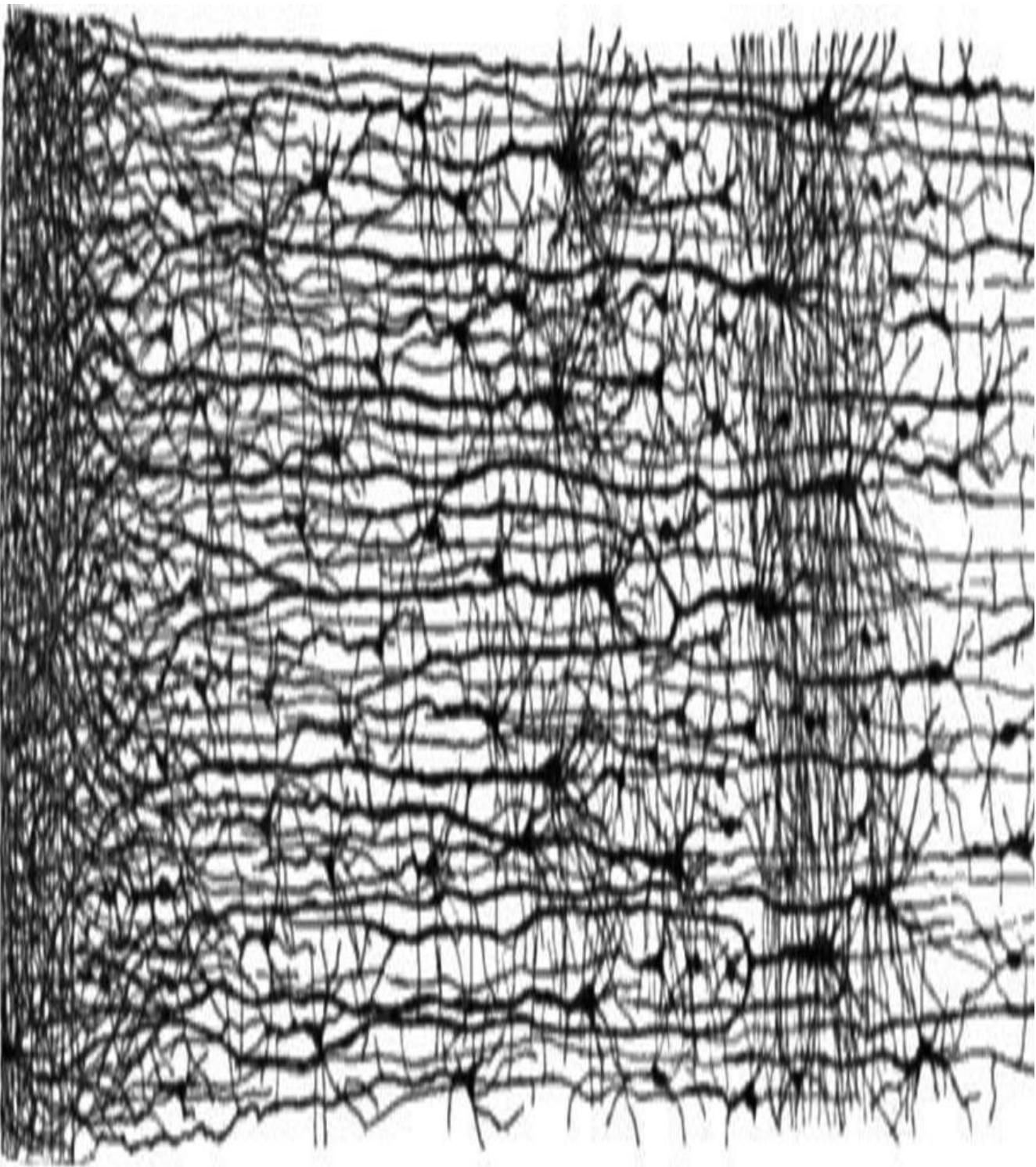
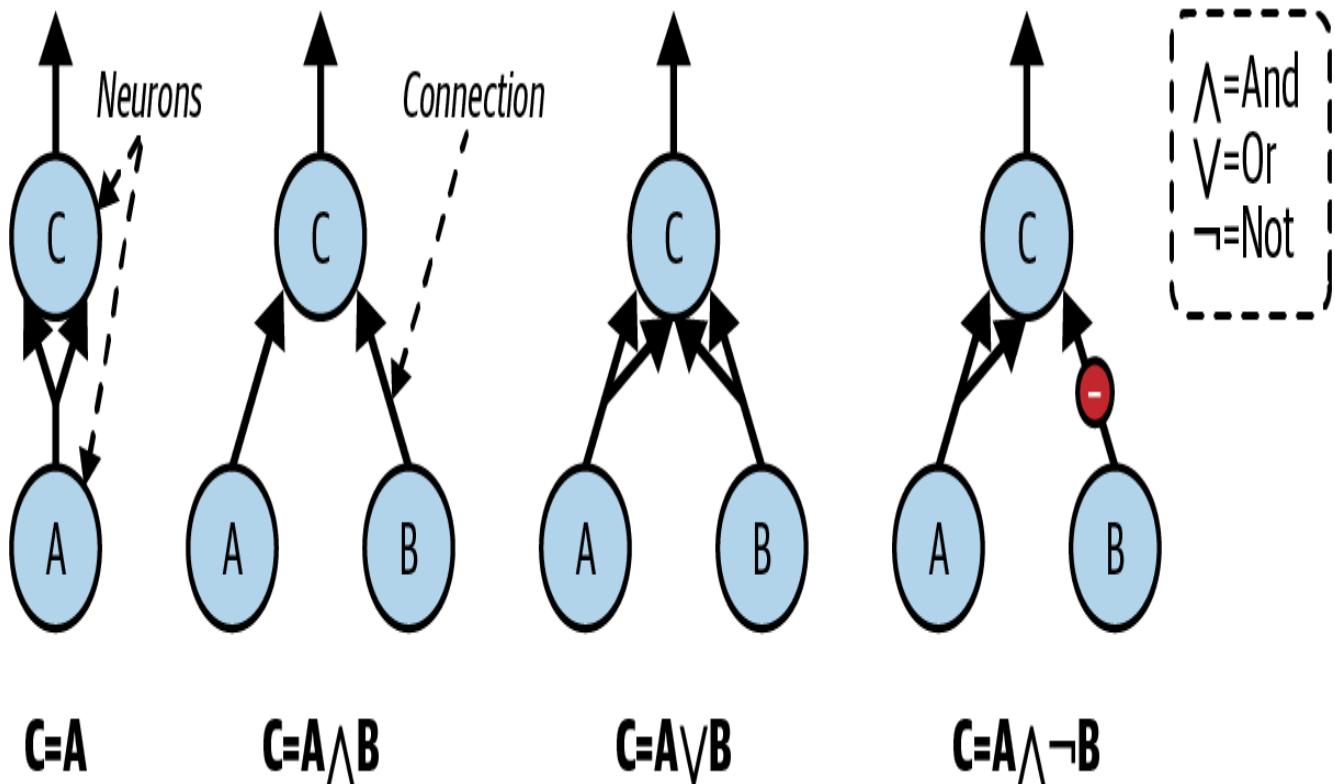


Figure 9-2. Multiple layers in a biological neural network (human cortex)<sup>6</sup>

## Logical Computations with Neurons

McCulloch and Pitts proposed a very simple model of the biological neuron, which later became known as an *artificial neuron*: it has one or more binary (on/off) inputs and one binary output. The artificial neuron activates its output when more than a certain number of its inputs are active. In their paper, McCulloch and Pitts showed that even

with such a simplified model it is possible to build a network of artificial neurons that can compute any logical proposition you want. To see how such a network works, let's build a few ANNs that perform various logical computations (see [Figure 9-3](#)), assuming that a neuron is activated when at least two of its input connections are active.



*Figure 9-3. ANNs performing simple logical computations*

Let's see what these networks do:

- The first network on the left is the identity function: if neuron A is activated, then neuron C gets activated as well (since it receives two input signals from neuron A); but if neuron A is off, then neuron C is off as well.
- The second network performs a logical AND: neuron C is activated only when both neurons A and B are activated (a single input signal is not enough to activate neuron C).
- The third network performs a logical OR: neuron C gets activated if either neuron A or neuron B is activated (or both).
- Finally, if we suppose that an input connection can inhibit the neuron's activity (which is the case with biological neurons), then the fourth network computes a slightly more complex logical proposition: neuron C is activated only if neuron

A is active and neuron B is off. If neuron A is active all the time, then you get a logical NOT: neuron C is active when neuron B is off, and vice versa.

You can imagine how these networks can be combined to compute complex logical expressions (see the exercises at the end of the chapter for an example).

## The Perceptron

The *perceptron* is one of the simplest ANN architectures, invented in 1957 by Frank Rosenblatt. It is based on a slightly different artificial neuron (see [Figure 9-4](#)) called a *threshold logic unit* (TLU), or sometimes a *linear threshold unit* (LTU). The inputs and output are numbers (instead of binary on/off values), and each input connection is associated with a weight. The TLU first computes a linear function of its inputs:  $z = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b = \mathbf{w}^\top \mathbf{x} + b$ . Then it applies a *step function* to the result:  $h_{\mathbf{w}}(\mathbf{x}) = \text{step}(z)$ . So it's almost like logistic regression, except it uses a step function instead of the logistic function.<sup>7</sup> Just like in logistic regression, the model parameters are the input weights  $\mathbf{w}$  and the bias term  $b$ .

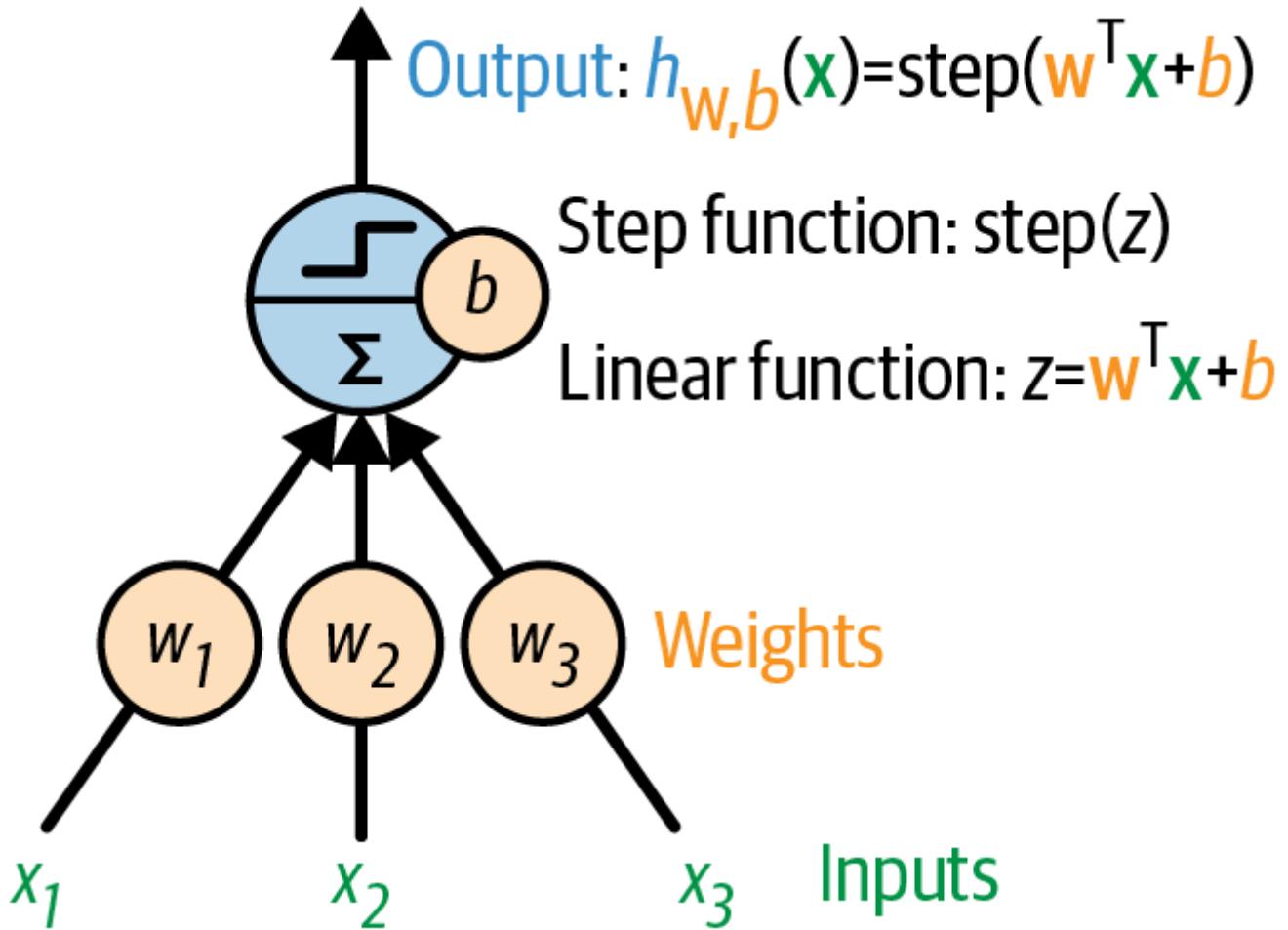


Figure 9-4. TLU: an artificial neuron that computes a weighted sum of its inputs  $w^T x$ , plus a bias term  $b$ , then applies a step function

The most common step function used in perceptrons is the *Heaviside step function* (see [Equation 9-1](#)). Sometimes the sign function is used instead.

[Equation 9-1. Common step functions used in perceptrons \(assuming threshold = 0\)](#)

$$\text{heaviside}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \quad \text{sgn}(z) = \begin{cases} -1 & \text{if } z < 0 \\ 0 & \text{if } z = 0 \\ +1 & \text{if } z > 0 \end{cases}$$

A single TLU can be used for simple linear binary classification. It computes a linear function of its inputs, and if the result exceeds a threshold, it outputs the positive class. Otherwise, it outputs the negative class. This may remind you of logistic regression ([Chapter 4](#)) or linear SVM classification (see the online chapter on SVMs at <https://homl.info/svm-p>). You could, for example, use a single TLU to classify iris flowers based on petal length and width. Training such a TLU would require finding the right values for  $w_1$ ,  $w_2$ , and  $b$  (the training algorithm is discussed shortly).

A perceptron is composed of one or more TLUs organized in a single layer, where every TLU is connected to every input. Such a layer is called a *fully connected layer*, or a *dense layer*. The inputs constitute the *input layer*. And since the layer of TLUs produces the final outputs, it is called the *output layer*. For example, a perceptron with two inputs and three outputs is represented in Figure 9-5.

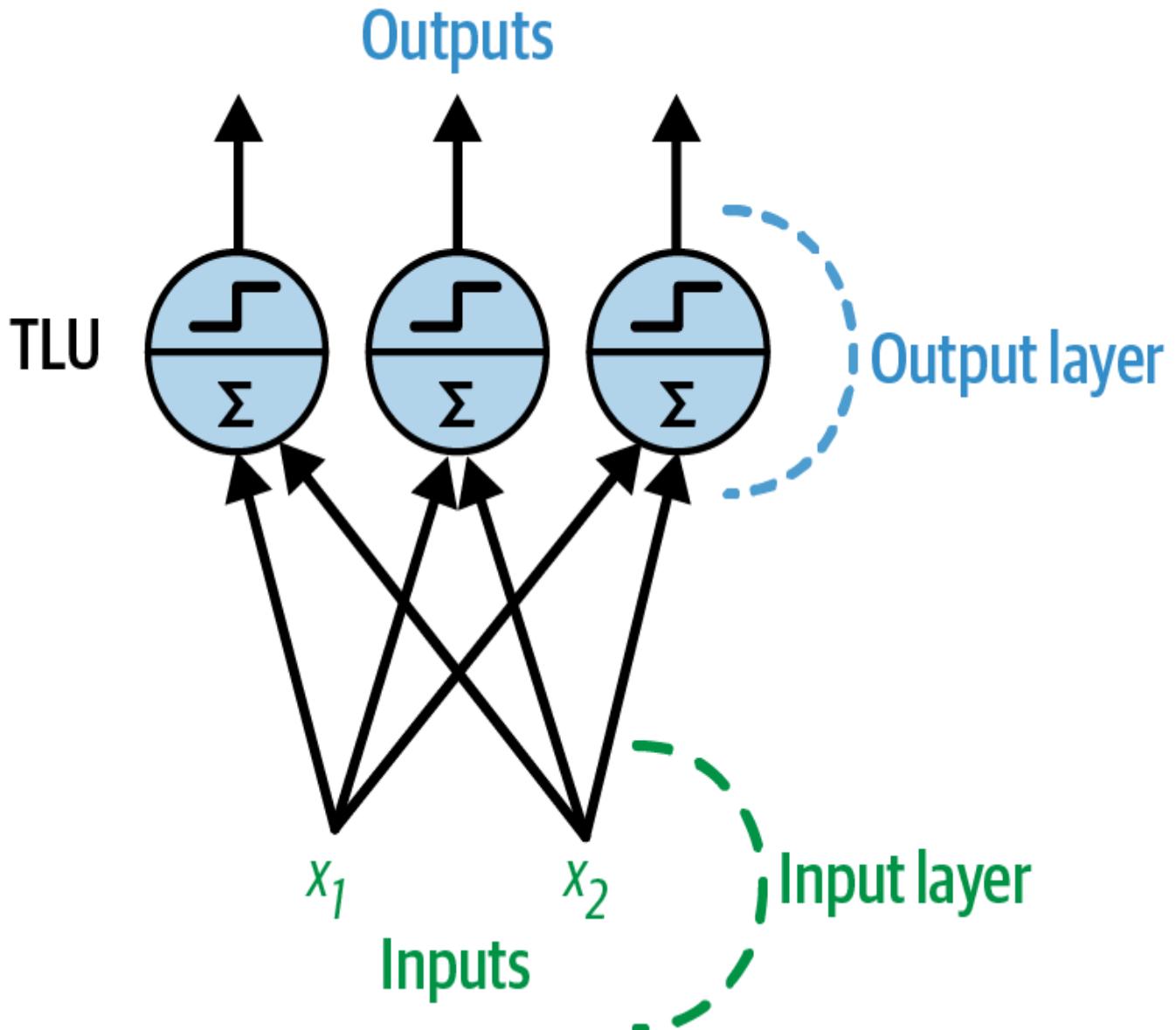


Figure 9-5. Architecture of a perceptron with two inputs and three output neurons

This perceptron can classify instances simultaneously into three different binary classes, which makes it a multilabel classifier. It may also be used for multiclass classification.

Thanks to the magic of linear algebra, Equation 9-2 can be used to efficiently compute the outputs of a layer of artificial neurons for several instances at once.

*Equation 9-2. Computing the outputs of a fully connected layer*

$$\hat{\mathbf{Y}} = \varphi(\mathbf{X}\mathbf{W} + \mathbf{b})$$

In this equation:

- $\hat{\mathbf{Y}}$  is the output matrix. It has one row per instance and one column per neuron.
- $\mathbf{X}$  is the input matrix. It has one row per instance and one column per input feature.
- The weight matrix  $\mathbf{W}$  contains all the connection weights. It has one row per input feature and one column per neuron.<sup>8</sup>
- The bias vector  $\mathbf{b}$  contains all the bias terms: one per neuron.
- The function  $\phi$  is called the *activation function*: when the artificial neurons are TLUs, it is a step function (we will discuss other activation functions shortly).

### NOTE

In mathematics, the sum of a matrix and a vector is undefined. However, in data science, we allow “broadcasting”: adding a vector to a matrix means adding it to every row in the matrix. So,  $\mathbf{X}\mathbf{W} + \mathbf{b}$  first multiplies  $\mathbf{X}$  by  $\mathbf{W}$ —which results in a matrix with one row per instance and one column per output—then adds the vector  $\mathbf{b}$  to every row of that matrix, which adds each bias term to the corresponding output, for every instance. Moreover,  $\phi$  is then applied itemwise to each item in the resulting matrix.

So, how is a perceptron trained? The perceptron training algorithm proposed by Rosenblatt was largely inspired by *Hebb's rule*. In his 1949 book *The Organization of Behavior* (Wiley), Donald Hebb suggested that when a biological neuron triggers another neuron often, the connection between these two neurons grows stronger. Siegrid Löwel later summarized Hebb's idea in the catchy phrase, “Cells that fire together, wire together”; that is, the connection weight between two neurons tends to increase when they fire simultaneously. This rule later became known as Hebb's rule (or *Hebbian learning*). Perceptrons are trained using a variant of this rule that takes into account the error made by the network when it makes a prediction; the perceptron learning rule reinforces connections that help reduce the error. More specifically, the perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction. The rule is shown in [Equation 9-3](#).

Equation 9-3. Perceptron learning rule (weight update)

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i$$

In this equation:

- $w_{i,j}$  is the connection weight between the  $i^{\text{th}}$  input and the  $j^{\text{th}}$  neuron.
- $x_i$  is the  $i^{\text{th}}$  input value of the current training instance.
- $\hat{y}_j$  is the output of the  $j^{\text{th}}$  output neuron for the current training instance.
- $y_j$  is the target output of the  $j^{\text{th}}$  output neuron for the current training instance.
- $\eta$  is the learning rate (see [Chapter 4](#)).

The decision boundary of each output neuron is linear, so perceptrons are incapable of learning complex patterns (just like logistic regression classifiers). However, if the training instances are linearly separable, Rosenblatt demonstrated that this algorithm will converge to a solution.<sup>9</sup> This is called the *perceptron convergence theorem*.

Scikit-Learn provides a `Perceptron` class that can be used pretty much as you would expect—for example, on the iris dataset (introduced in [Chapter 4](#)):

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.linear_model import Perceptron

iris = load_iris(as_frame=True)
X = iris.data[["petal length (cm)", "petal width (cm)"]].values
y = (iris.target == 0) # Iris setosa

per_clf = Perceptron(random_state=42)
per_clf.fit(X, y)

X_new = [[2, 0.5], [3, 1]]
y_pred = per_clf.predict(X_new) # predicts True and False for these 2 flowers
```

You may have noticed that the perceptron learning algorithm strongly resembles stochastic gradient descent (introduced in [Chapter 4](#)). In fact, Scikit-Learn's `Perceptron` class is equivalent to using an `SGDClassifier` with the following hyperparameters: `loss="perceptron"`, `learning_rate="constant"`, `eta0=1` (the learning rate), and `penalty=None` (no regularization).

## NOTE

Contrary to logistic regression classifiers, perceptrons do not output a class probability. This is one reason to prefer logistic regression over perceptrons. Moreover, perceptrons do not use any regularization by default, and training stops as soon as there are no more prediction errors on the training set, so the model typically does not generalize as well as logistic regression or a linear SVM classifier. However, perceptrons may train a bit faster.

In their 1969 monograph *Perceptrons*, Marvin Minsky and Seymour Papert highlighted a number of serious weaknesses of perceptrons—in particular, the fact that they are incapable of solving some trivial problems (e.g., the *exclusive OR* (XOR) classification problem; see the left side of [Figure 9-6](#)). This is true of any other linear classification model (such as logistic regression classifiers), but researchers had expected much more from perceptrons, and some were so disappointed that they dropped neural networks altogether in favor of more formal approaches such as logic, problem solving, and search. The lack of practical applications also didn’t help.

It turns out that some of the limitations of perceptrons can be eliminated by stacking multiple perceptrons. The resulting ANN is called a *multilayer perceptron* (MLP).

## The Multilayer Perceptron and Backpropagation

An MLP can solve the XOR problem, as you can verify by computing the output of the MLP represented on the righthand side of [Figure 9-6](#): with inputs  $(0, 0)$  or  $(1, 1)$ , the network outputs 0, and with inputs  $(0, 1)$  or  $(1, 0)$  it outputs 1. Try verifying that this network indeed solves the XOR problem!<sup>10</sup>

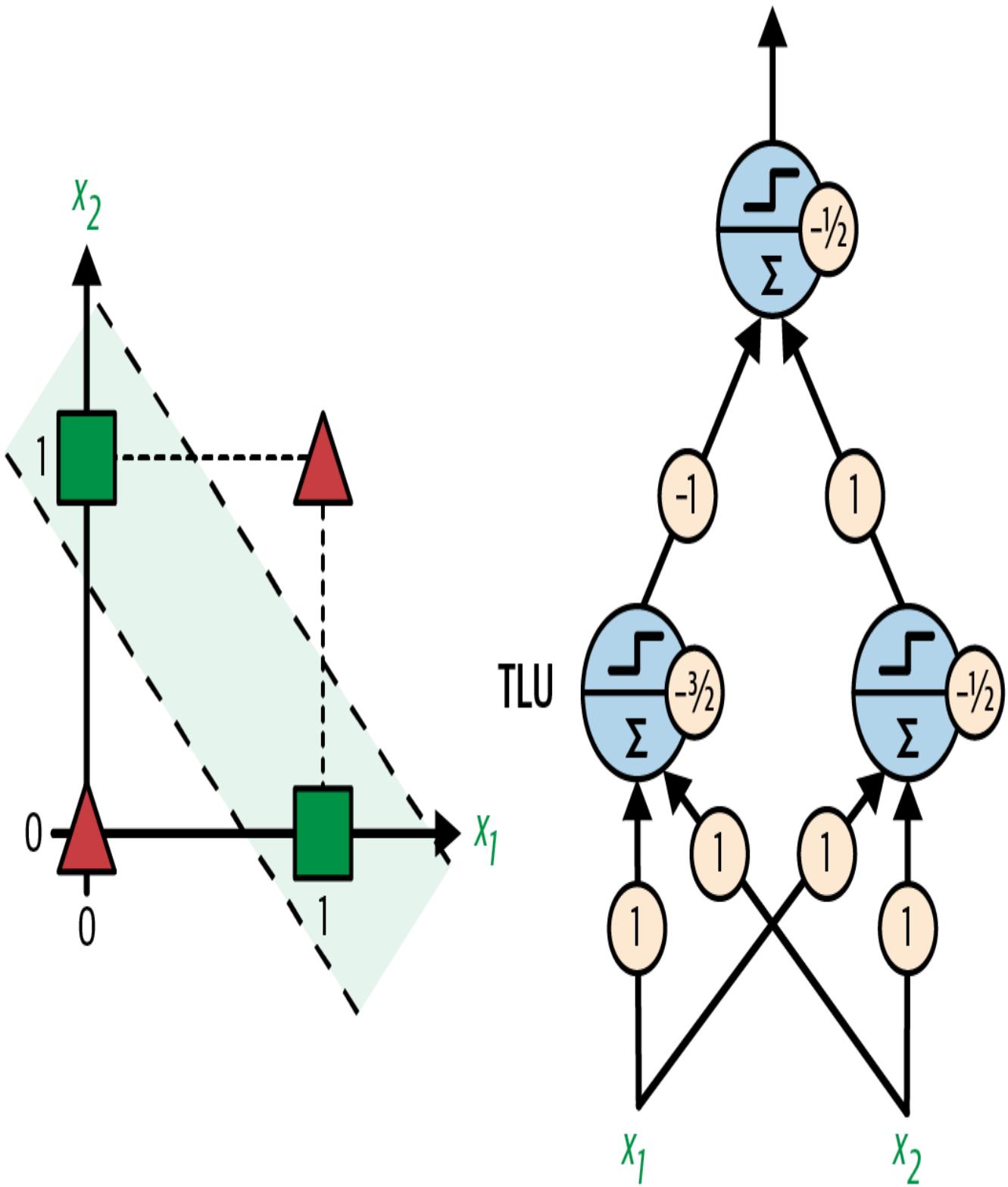


Figure 9-6. XOR classification problem and an MLP that solves it

An MLP is composed of one input layer, one or more layers of artificial neurons (originally TLUs) called *hidden layers*, and one final layer of artificial neurons called the *output layer* (see [Figure 9-7](#)). The layers close to the input layer are usually called the *lower layers*, and the ones close to the outputs are usually called the *upper layers*.

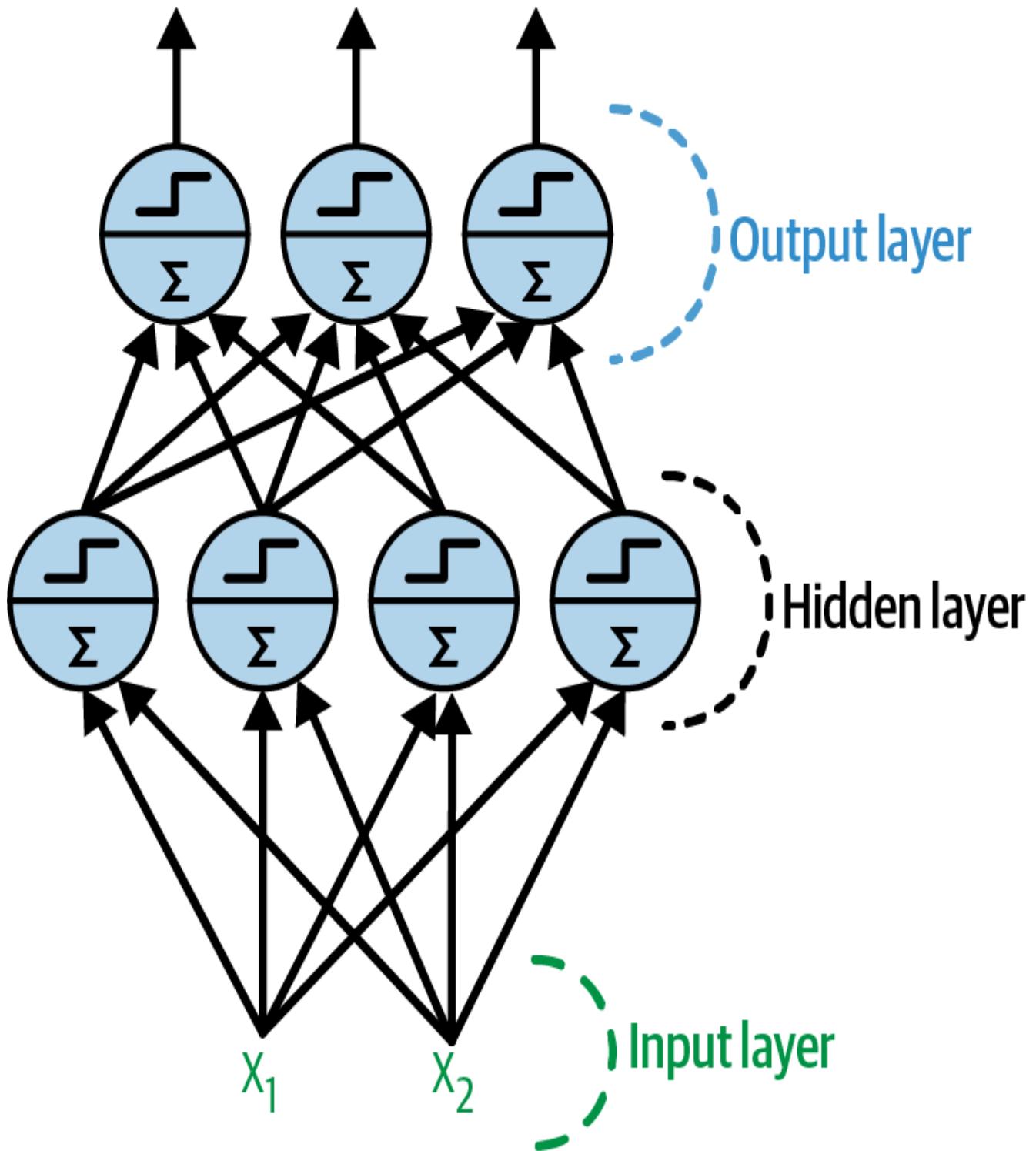


Figure 9-7. Architecture of a multilayer perceptron with two inputs, one hidden layer of four neurons, and three output neurons

#### NOTE

The signal flows only in one direction (from the inputs to the outputs), so this architecture is an example of a *feedforward neural network* (FNN).

When an ANN contains a deep stack of hidden layers,<sup>11</sup> it is called a *deep neural network* (DNN). The field of deep learning studies DNNs, and more generally it is interested in models containing deep stacks of computations. Even so, many people talk about deep learning whenever neural networks are involved (even shallow ones).

For many years researchers struggled to find a way to train MLPs, without success. In the early 1960s several researchers discussed the possibility of using gradient descent to train neural networks, but as we saw in [Chapter 4](#), this requires computing the gradients of the model’s error with regard to the model parameters; it wasn’t clear at the time how to do this efficiently with such a complex model containing so many parameters, especially with the computers they had back then.

Then, in 1970, a researcher named Seppo Linnainmaa introduced in his master’s thesis a technique to compute all the gradients automatically and efficiently. This algorithm is now called *reverse-mode automatic differentiation* (or *reverse-mode autodiff* for short). In just two passes through the network (one forward, one backward), it is able to compute the gradients of the neural network’s error with regard to every single model parameter. In other words, it can find out how each connection weight and each bias should be tweaked in order to reduce the neural network’s error. These gradients can then be used to perform a gradient descent step. If you repeat this process of computing the gradients automatically and taking a gradient descent step, the neural network’s error will gradually drop until it eventually reaches a minimum. This combination of reverse-mode autodiff and gradient descent is now called *backpropagation* (or *backprop* for short).

Here’s an analogy: imagine you are learning to shoot a basketball into the hoop. You throw the ball (that’s the forward pass), and you observe that it went far off to the right side (that’s the error computation), then you consider how you can change your body position to throw the ball a bit less to the right next time (that’s the backward pass): you realize that your arm will need to rotate a bit counterclockwise, and probably your whole upper body as well, which in turn means that your feet should turn too (notice how we’re going down the “layers”). Once you’ve thought it through, you actually move your body: that’s the gradient descent step. The smaller the errors, the smaller the adjustments. As you repeat the whole process many times, the error gradually gets smaller, and after a few hours of practice, you manage to get the ball through the hoop every time. Good job!

## NOTE

There are various autodiff techniques, with different pros and cons. Reverse-mode autodiff is well suited when the function to differentiate has many variables (e.g., connection weights and biases) and few outputs (e.g., one loss). If you want to learn more about autodiff, check out [Link to Come].

Backpropagation can actually be applied to all sorts of computational graphs, not just neural networks: indeed, Linnainmaa's master's thesis was not about neural nets at all, it was more general. It was several more years before backprop started to be used to train neural networks, but it still wasn't mainstream. Then, in 1985, David Rumelhart, Geoffrey Hinton, and Ronald Williams published a [paper<sup>12</sup>](#) analyzing how backpropagation allows neural networks to learn useful internal representations. Their results were so impressive that backpropagation was quickly popularized in the field. Over 40 years later, it is still by far the most popular training technique for neural networks.

Let's run through how backpropagation works again in a bit more detail:

- It handles one mini-batch at a time, and goes through the full training set multiple times. If each mini-batch contains 32 instances, and each instance has 100 features, then the mini-batch will be represented as a matrix with 32 rows and 100 columns. Each pass through the training set is called an *epoch*.
- For each mini-batch, the algorithm computes the output of all the neurons in the first hidden layer using [Equation 9-2](#). If the layer has 50 neurons, then its output is a matrix with one row per sample in the mini-batch (e.g., 32), and 50 columns (i.e., one per neuron). This matrix is then passed on to the next layer, its output is computed and passed to the next layer, and so on until we get the output of the last layer, the output layer. This is the *forward pass*: it is exactly like making predictions, except all intermediate results are preserved since they are needed for the backward pass.
- Next, the algorithm measures the network's output error (i.e., it uses a loss function that compares the desired output and the actual output of the network, and returns some measure of the error).
- Then it computes how much each output layer parameter contributed to the error. This is done analytically by applying the *chain rule* (one of the most fundamental rules in calculus), which makes this step fast and precise. The result is one gradient per parameter.

- The algorithm then measures how much of these error contributions came from each connection in the layer below, again using the chain rule, working backward until it reaches the input layer. As explained earlier, this reverse pass efficiently measures the error gradient across all the connection weights and biases in the network by propagating the error gradient backward through the network (hence the name of the algorithm).
- Finally, the algorithm performs a gradient descent step to tweak all the connection weights and bias terms in the network, using the error gradients it just computed.

### WARNING

It is important to initialize all the hidden layers' connection weights randomly, or else training will fail. For example, if you initialize all weights and biases to zero, then all neurons in a given layer will be perfectly identical, and thus backpropagation will affect them in exactly the same way, so they will remain identical. In other words, despite having hundreds of neurons per layer, your model will act as if it had only one neuron per layer: it won't be too smart. If instead you randomly initialize the weights, you *break the symmetry* and allow backpropagation to train a diverse team of neurons.

In short, backpropagation makes predictions for a mini-batch (forward pass), measures the error, then goes through each layer in reverse to measure the error contribution from each parameter (reverse pass), and finally tweaks the connection weights and biases to reduce the error (gradient descent step).

In order for backprop to work properly, Rumelhart and his colleagues made a key change to the MLP's architecture: they replaced the step function with the logistic function,  $\sigma(z) = 1 / (1 + \exp(-z))$ , also called the *sigmoid* function. This was essential because the step function contains only flat segments, so there is no gradient to work with (gradient descent cannot move on a flat surface), while the sigmoid function has a well-defined nonzero derivative everywhere, allowing gradient descent to make some progress at every step. In fact, the backpropagation algorithm works well with many other activation functions, not just the sigmoid function. Here are two other popular choices:

*The hyperbolic tangent function:  $\tanh(z) = 2\sigma(2z) - 1$*

Just like the sigmoid function, this activation function is S-shaped, continuous, and differentiable, but its output value ranges from  $-1$  to  $1$  (instead of  $0$  to  $1$  in the case of the sigmoid function). That range

tends to make each layer's output more or less centered around 0 at the beginning of training, which often helps speed up convergence.

*The rectified linear unit function:  $\text{ReLU}(z) = \max(0, z)$*

The ReLU function is continuous but unfortunately not differentiable at  $z = 0$  (the slope changes abruptly, which can make gradient descent bounce around), and its derivative is 0 for  $z < 0$ . In practice, however, it works very well and has the advantage of being fast to compute, so it has become the default for most architectures (except transformers, as we will see in [Link to Come]).<sup>13</sup> Importantly, the fact that it does not have a maximum output value helps reduce some issues during gradient descent (we will come back to this in [Chapter 11](#)).

These popular activation functions and their derivatives are represented in [Figure 9-8](#). But wait! Why do we need activation functions in the first place? Well, if you chain several linear transformations, all you get is a linear transformation. For example, if  $f(x) = 2x + 3$  and  $g(x) = 5x - 1$ , then chaining these two linear functions gives you another linear function:  $f(g(x)) = 2(5x - 1) + 3 = 10x + 1$ . So if you don't have some nonlinearity between layers, then even a deep stack of layers is equivalent to a single layer, and you can't solve very complex problems with that. Conversely, a large enough DNN with nonlinear activations can theoretically approximate any continuous function.

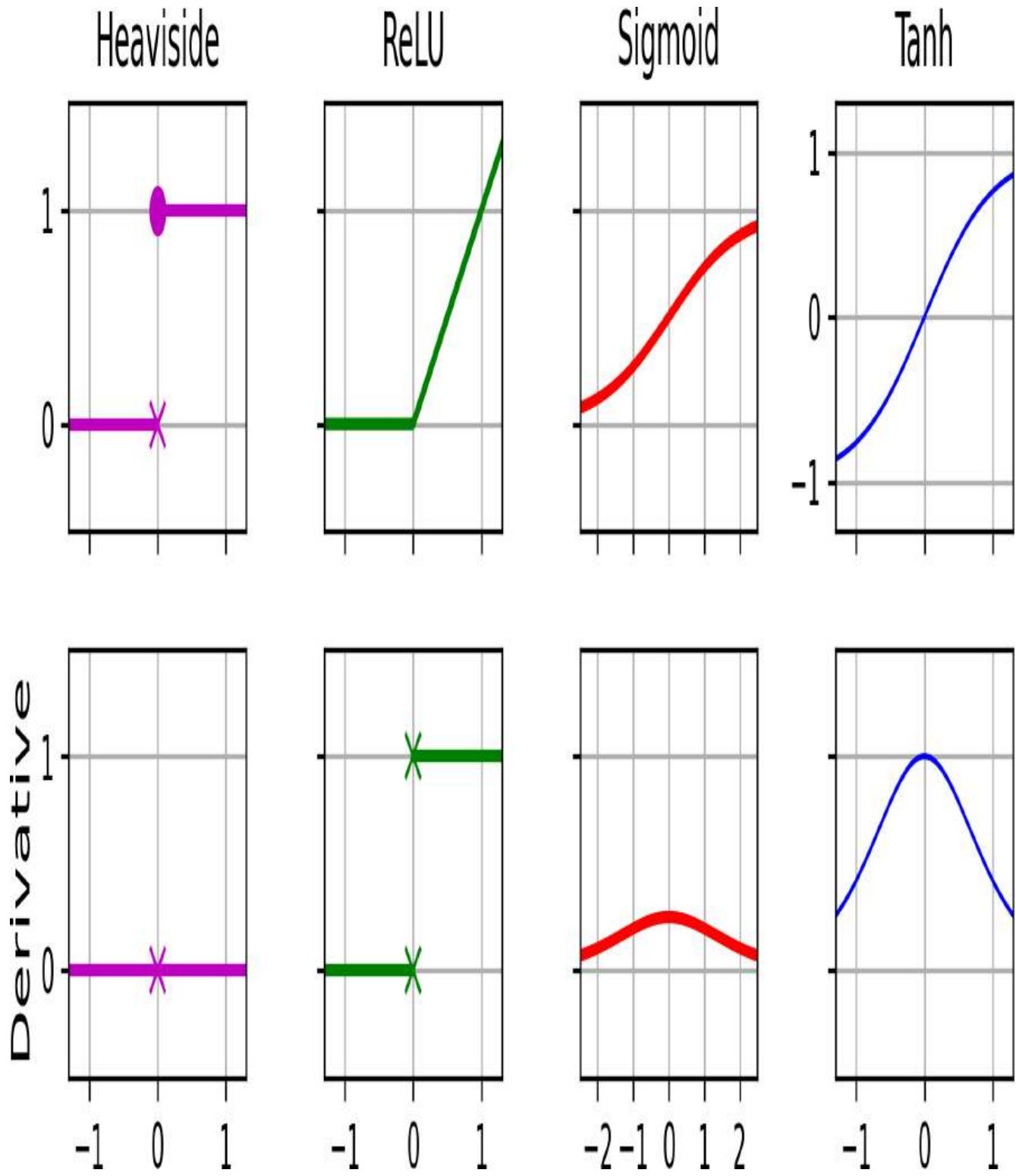


Figure 9-8. Activation functions (left) and their derivatives (right)

OK! You know where neural nets came from, what the MLP architecture looks like, and how it computes its outputs. You've also learned about the backpropagation algorithm. It's time to see MLPs in action!

# Building and Training MLPs with Scikit-Learn

MLPs can tackle a wide range of tasks, but the most common are regression and classification. Scikit-Learn can help with both of these. Let's start with regression.

## Regression MLPs

How would you build an MLP for a regression task? Well, if you want to predict a single value (e.g., the price of a house, given many of its features), then you just need a single output neuron: its output is the predicted value. For multivariate regression (i.e., to predict multiple values at once), you need one output neuron per output dimension. For example, to locate the center of an object in an image, you need to predict 2D coordinates, so you need two output neurons. If you also want to place a bounding box around the object, then you need two more numbers: the width and the height of the object. So, you end up with four output neurons.

Scikit-Learn includes an `MLPRegressor` class, so let's use it to build an MLP with three hidden layers composed of 50 neurons each, and train it on the California housing dataset. For simplicity, we will use Scikit-Learn's `fetch_california_housing()` function to load the data. This dataset is simpler than the one we used in [Chapter 2](#), since it contains only numerical features (there is no `ocean_proximity` feature), and there are no missing values. The targets are also scaled down: each unit represents \$100,000. Let's start by importing everything we will need:

```
from sklearn.datasets import fetch_california_housing
from sklearn.metrics import root_mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPRegressor
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
```

Next, let's fetch the California housing dataset and split it into a training set and a test set:

```
housing = fetch_california_housing()
X_train, X_test, y_train, y_test = train_test_split(
    housing.data, housing.target, random_state=42)
```

Now let's create an `MLPRegressor` model with 3 hidden layers composed of 50 neurons each. The first hidden layer's input size (i.e., the number of rows in its weights matrix) and the output layer's output size (i.e., the number of columns in its weights

matrix) will adjust automatically to the dimensionality of the inputs and targets respectively when training starts. The model uses the ReLU activation function in all hidden layers, and no activation function at all on the output layer. We also set `verbose=True` to get details on the model's progress during training.

```
mlp_reg = MLPRegressor(hidden_layer_sizes=[50, 50, 50], early_stopping=True,  
                      verbose=True, random_state=42)
```

Since neural nets can have a *lot* of parameters, they have a tendency to overfit the training set. To reduce this risk, it's a good idea to use early stopping (introduced in [Chapter 4](#)): when we set `early_stopping=True`, the `MLPRegressor` class automatically sets aside 10% of the training data and uses it to evaluate the model at each epoch (you can adjust the validation set's size by setting `validation_fraction`). If the validation score stops improving for 10 epochs, training automatically stops (you can tweak this number of epochs by setting `n_iter_no_change`).

Now let's creates a pipeline to standardize the input features before sending them to the `MLPRegressor`. This is very important because gradient descent does not converge very well when the features have very different scales, as we saw in [Chapter 4](#). We can then train the model! The `MLPRegressor` class uses a variant of gradient descent called *Adam* (see [Chapter 11](#)) to minimize the mean squared error. It also uses a tiny bit of  $\ell_2$  regularization (you can control its strength via the `alpha` hyperparameter, which defaults to 0.0001):

```
>>> pipeline = make_pipeline(StandardScaler(), mlp_reg)  
>>> pipeline.fit(X_train, y_train)  
Iteration 1, loss = 0.85190332  
Validation score: 0.534299  
Iteration 2, loss = 0.28288639  
Validation score: 0.651094  
[...]  
Iteration 45, loss = 0.12960481  
Validation score: 0.788517  
Validation score did not improve more than tol=0.000100 for 10 consecutive  
epochs. Stopping.
```

And there you go, you just trained your very first MLP! It required 45 epochs, and as you can see the training loss went down at each epoch. This loss corresponds to [Equation 4-8](#) divided by 2, so you must multiply it by 2 to get the MSE (although not exactly because the loss includes the  $\ell_2$  regularization term). The validation score generally went up at each epoch. Like every regressor in Scikit-Learn, `MLPRegressor`

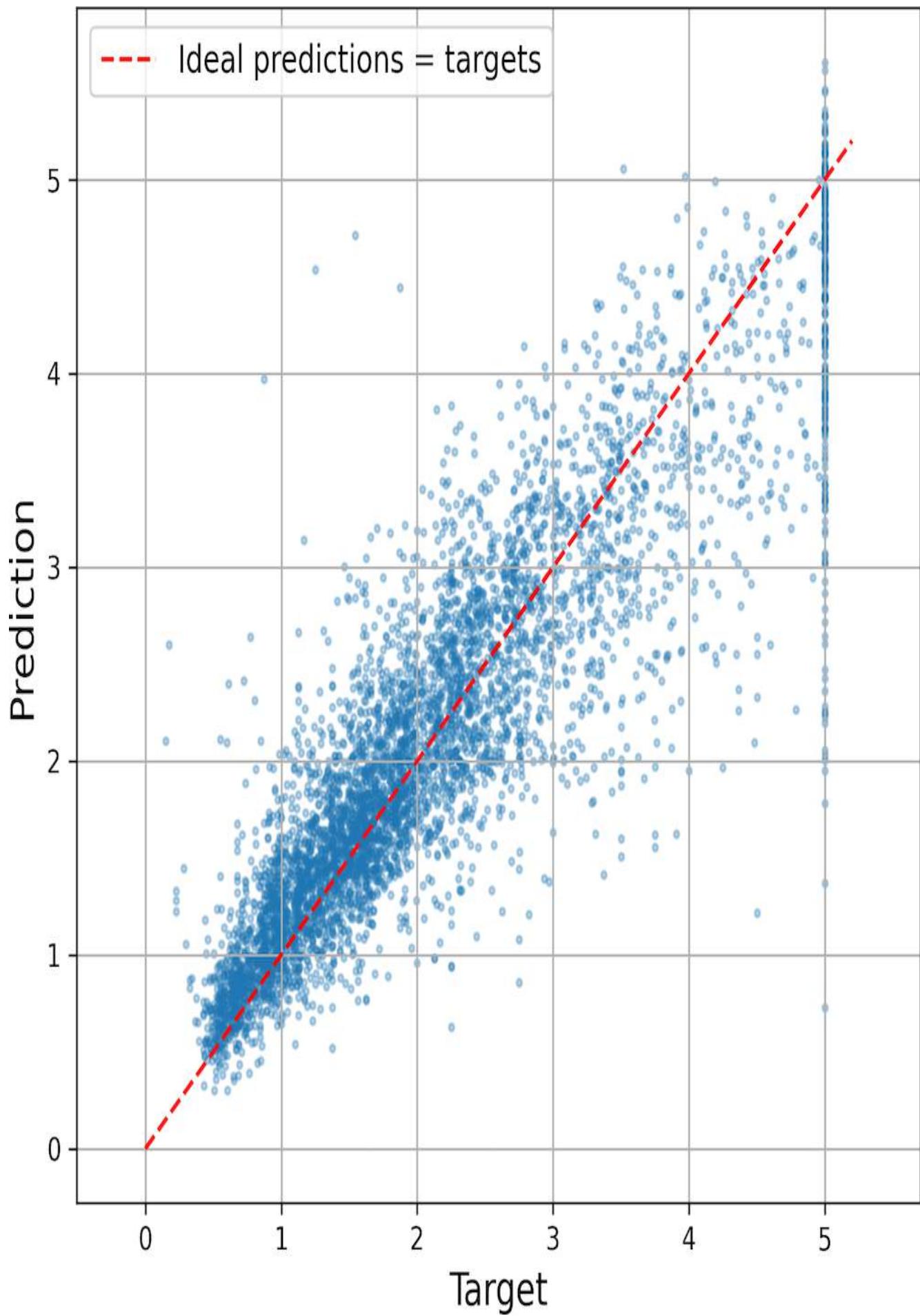
uses the  $R^2$  score by default for evaluation—that's what the `score()` method returns. As we saw in [Chapter 2](#), the  $R^2$  score measures the ratio of the variance that is explained by the model. In this case, it reaches close to 80% on the validation set, which is fairly good for this task:

```
>>> mlp_reg.best_validation_score_
0.791536125425778
```

Let's evaluate the RMSE on the test set:

```
>>> y_pred = pipeline.predict(X_test)
>>> rmse = root_mean_squared_error(y_test, y_pred)
>>> rmse
0.5327699946812925
```

We get a test RMSE of about 0.53, which is comparable to what you would get with a random forest classifier. Not too bad for a first try! [Figure 9-9](#) plots the model's predictions versus the targets (on the test set). The dashed red line represents the ideal predictions (i.e., equal to the targets): most of the predictions are close to the targets, but there are still quite a few errors, especially for larger targets.



*Figure 9-9. MLP regressor's predictions versus the targets*

Note that this MLP does not use any activation function for the output layer, so it's free to output any value it wants. This is generally fine, but if you want to guarantee that the output is always positive, then you should use the ReLU activation function on the output layer, or the *softplus* activation function, which is a smooth variant of ReLU:  $\text{softplus}(z) = \log(1 + \exp(z))$ . Softplus is close to 0 when  $z$  is negative, and close to  $z$  when  $z$  is positive. Finally, if you want to guarantee that the predictions always fall within a given range of values, then you should use the sigmoid function or the hyperbolic tangent, and scale the targets to the appropriate range: 0 to 1 for sigmoid and  $-1$  to  $1$  for tanh. Sadly, the `MLPRegressor` class does not support activation functions in the output layer.

### WARNING

Scikit-Learn does not offer GPU acceleration, and its neural net features are fairly limited. This is why we will switch to PyTorch starting in [Chapter 10](#). That said, it is quite convenient to be able to build and train a standard MLP in just a few lines of code using Scikit-Learn: it lets you tackle many complex tasks very quickly.

In general, the mean squared error is the right loss to use for a regression tasks, but if you have a lot of outliers in the training set, you may sometimes prefer to use the mean absolute error instead, or preferably the *Huber loss*, which is a combination of both: it is quadratic when the error is smaller than a threshold  $\delta$  (typically 1) but linear when the error is larger than  $\delta$ . The linear part makes it less sensitive to outliers than the mean squared error, and the quadratic part allows it to converge faster and be more precise than the mean absolute error. Unfortunately, `MLPRegressor` only supports the MSE loss.

[Table 9-1](#) summarizes the typical architecture of a regression MLP.

*Table 9-1. Typical regression MLP architecture*

Hyperparameter	Typical value
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per target dimension
Hidden activation	ReLU
Output activation	None, or ReLU/softplus (if positive outputs) or sigmoid/tanh (if bounded outputs)
Loss function	MSE, or Huber if outliers

All right, MLPs can tackle regression tasks. What else can they do?

## Classification MLPs

MLPs can also be used for classification tasks. For a binary classification problem, you just need a single output neuron using the sigmoid activation function: the output will be a number between 0 and 1, which you can interpret as the estimated probability of the positive class. The estimated probability of the negative class is equal to one minus that number.

MLPs can also easily handle multilabel binary classification tasks (see [Chapter 3](#)). For example, you could have an email classification system that predicts whether each incoming email is ham or spam, and simultaneously predicts whether it is an urgent or nonurgent email. In this case, you would need two output neurons, both using the sigmoid activation function: the first would output the probability that the email is spam, and the second would output the probability that it is urgent. More generally, you would dedicate one output neuron for each positive class. Note that the output probabilities do not necessarily add up to 1. This lets the model output any combination of labels: you

can have nonurgent ham, urgent ham, nonurgent spam, and perhaps even urgent spam (although that would probably be an error).

If each instance can belong only to a single class, out of three or more possible classes (e.g., classes 0 through 9 for digit image classification), then you need to have one output neuron per class, and you should use the softmax activation function for the whole output layer (see [Figure 9-10](#)). The softmax function (introduced in [Chapter 4](#)) will ensure that all the estimated probabilities are between 0 and 1 and that they add up to 1, since the classes are exclusive. As we saw in [Chapter 3](#), this is called multiclass classification.

Regarding the loss function, since we are predicting probability distributions, the cross-entropy loss (or *x-entropy* or log loss for short, see [Chapter 4](#)) is generally a good choice.

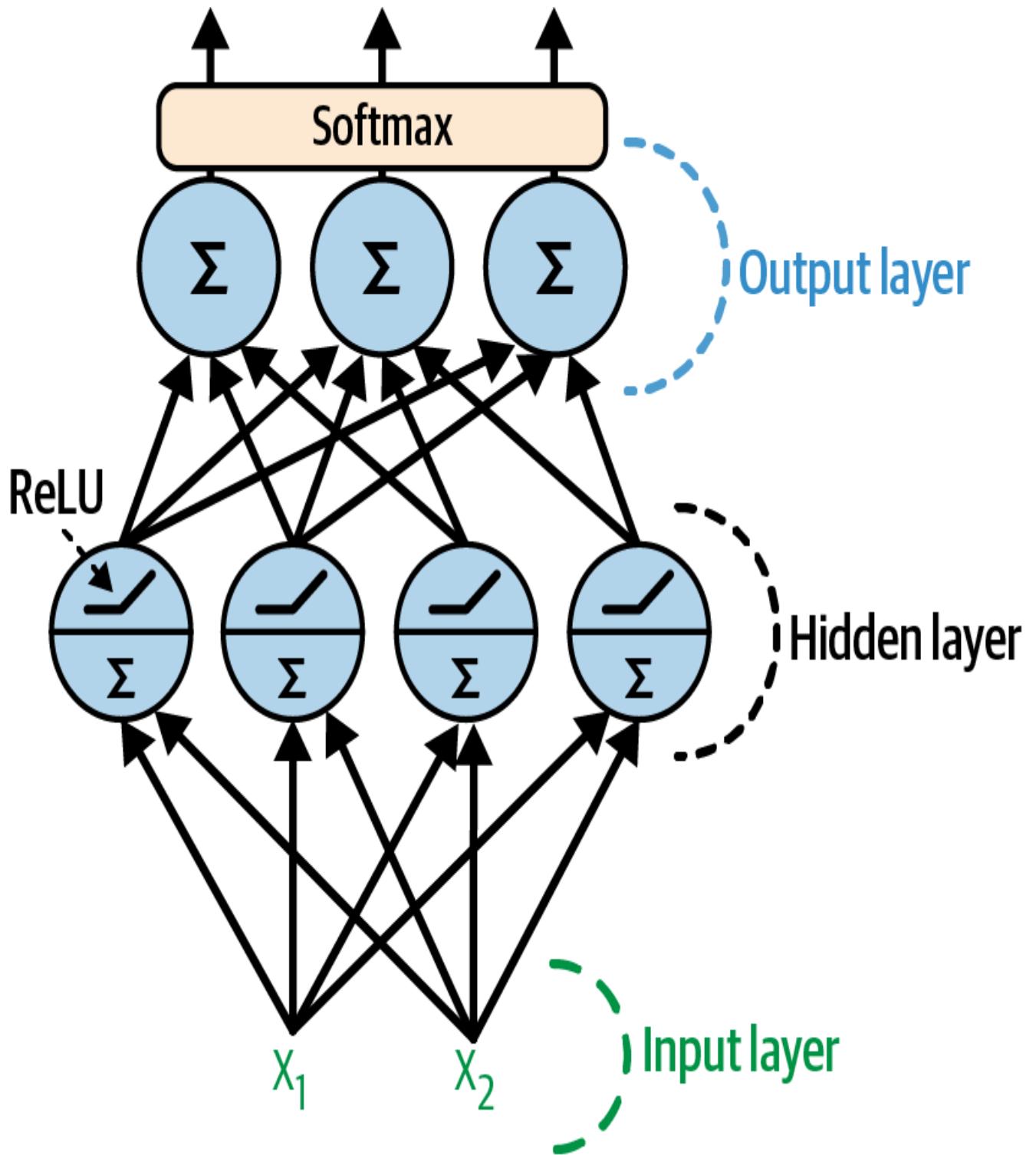


Figure 9-10. A modern MLP (including ReLU and softmax) for classification

Table 9-2 summarizes the typical architecture of a classification MLP.

Table 9-2. Typical classification MLP architecture

Hyperparameter	Binary classification	Multilabel binary classification	Multiclass classification
# hidden layers		Typically 1 to 5 layers, depending on the task	
# output neurons	1	1 per binary label	1 per class
Output layer activation	Sigmoid	Sigmoid	Softmax
Loss function	X-entropy		

As you might expect, Scikit-Learn offers an `MLPClassifier` class in the `sklearn.neural_network` package, which you can use for binary or multiclass classification. It is almost identical to the `MLPRegressor` class, except that its output layer uses the softmax activation function, and it minimizes the cross entropy loss rather than the MSE. Moreover, the `score()` method returns the model's accuracy rather than the  $R^2$  score. Let's try it out.

We could tackle the iris dataset, but that task is too simple for a neural net: a linear model would do just as well and wouldn't risk overfitting. So let's instead tackle a more complex task: Fashion MNIST. This is a drop-in replacement of MNIST (introduced in [Chapter 3](#)). It has the exact same format as MNIST (70,000 grayscale images of  $28 \times 28$  pixels each, with 10 classes), but the images represent fashion items rather than handwritten digits, so each class is much more diverse, and the problem turns out to be significantly more challenging than MNIST. For example, a simple linear model reaches about 92% accuracy on MNIST, but only about 83% on Fashion MNIST. Let's see if we can do better with an MLP.

First, let's load the dataset using the `fetch_openml()` function, very much like we did for MNIST in [Chapter 3](#). Note that the targets are represented as strings '`'0'`', '`'1'`', ..., '`'9'`', so we convert them to integers:

```
from sklearn.datasets import fetch_openml
```

```
fashion_mnist = fetch_openml(name="Fashion-MNIST", as_frame=False)
targets = fashion_mnist.target.astype(int)
```

The data is already shuffled, so we just take the first 60,000 images for training, and the last 10,000 for testing:

```
X_train, y_train = fashion_mnist.data[:60_000], targets[:60_000]
X_test, y_test = fashion_mnist.data[60_000:], targets[60_000:]
```

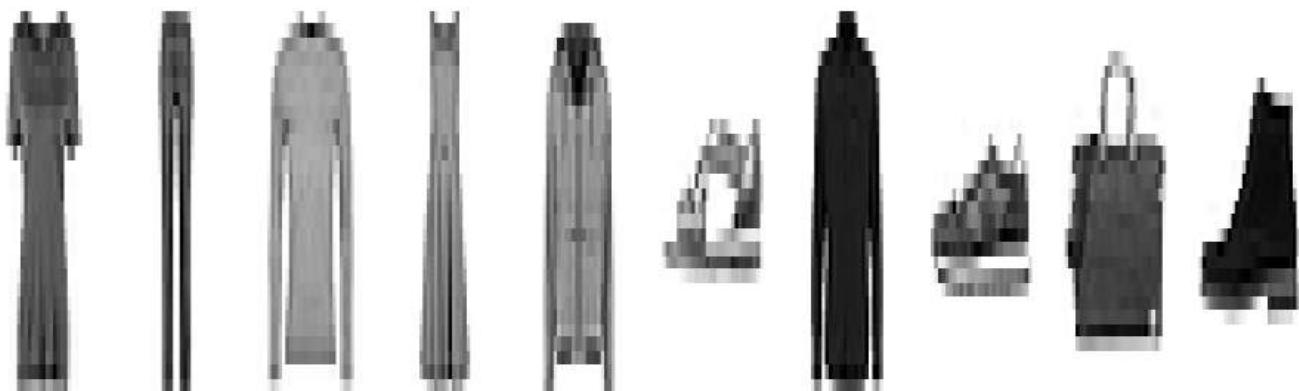
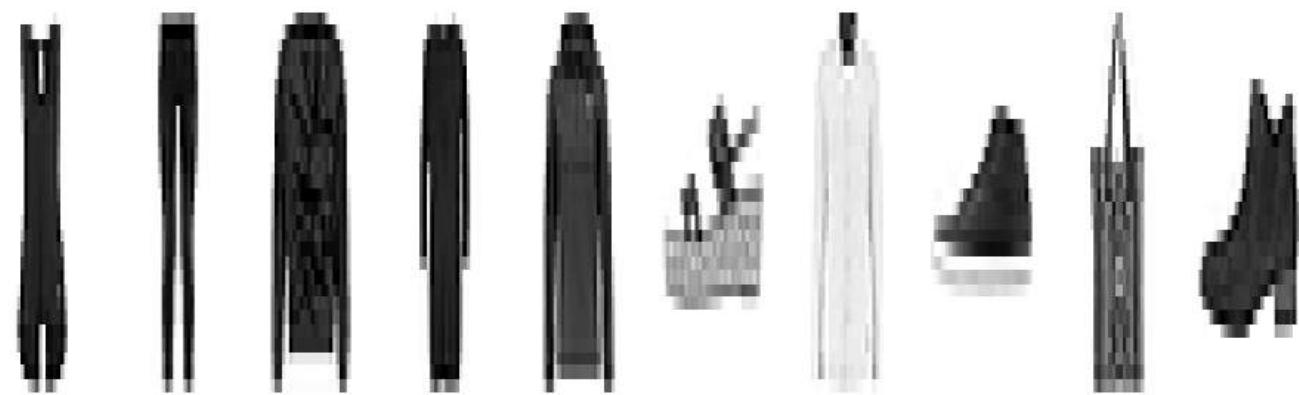
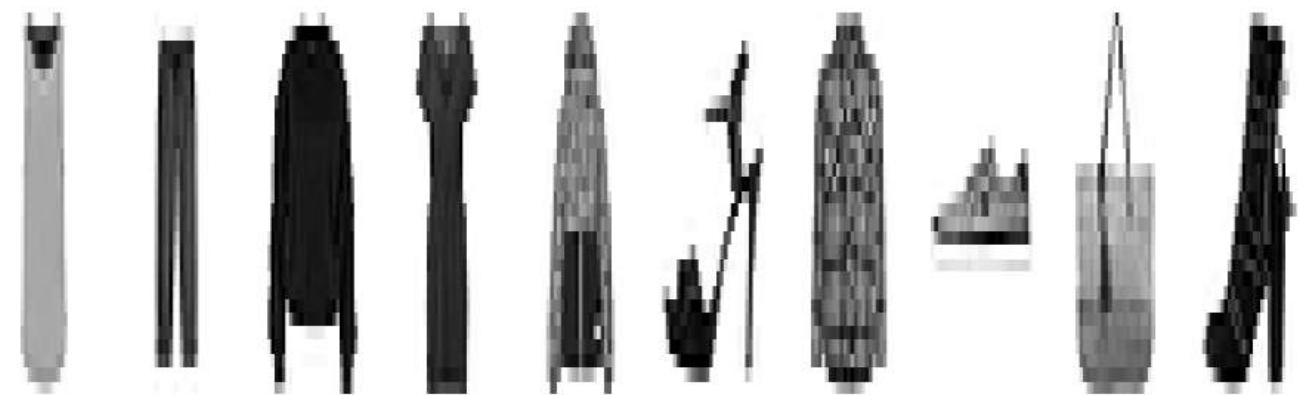
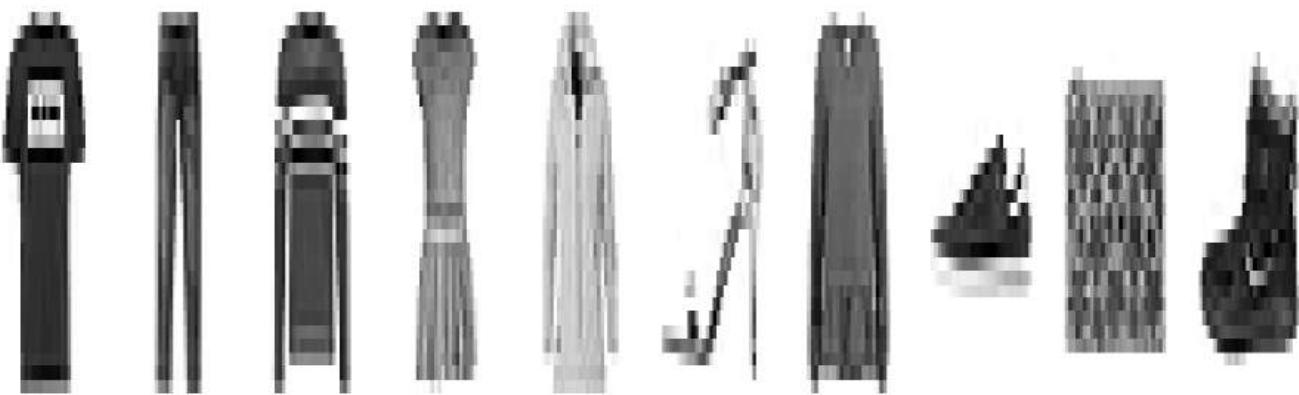
Each image is represented as a 1D integer array containing 784 pixel intensities ranging from 0 to 255. You can use the `plt.imshow()` function to plot an image, but first you need to reshape it to [28, 28]:

```
import matplotlib.pyplot as plt

X_sample = X_train[0].reshape(28, 28) # first image in the training set
plt.imshow(X_sample, cmap="binary")
plt.show()
```

If you run this code, you should see the ankle boot represented in the top right corner of Figure 9-11.

T-shirt/top Trouser Pullover Dress Coat Sandal Shirt Sneaker Bag Ankle boot



*Figure 9-11. First four samples from each class in Fashion MNIST*

With MNIST, when the label is equal to 5, it means that the image represents the handwritten digit 5. Easy. For Fashion MNIST, however, we need the list of class names to know what we are dealing with. Scikit-Learn does not provide it, so let's create it:

```
class_names = ["T-shirt/top", "Trouser", "Pullover", "Dress", "Coat",
               "Sandal", "Shirt", "Sneaker", "Bag", "Ankle boot"]
```

We can now confirm that the first image in the training set represents an ankle boot:

```
>>> class_names[y_train[0]]
'Ankle boot'
```

We're ready to build the classification MLP:

```
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import MinMaxScaler

mlp_clf = MLPClassifier(hidden_layer_sizes=[300, 100], verbose=True,
                        early_stopping=True, random_state=42)
pipeline = make_pipeline(MinMaxScaler(), mlp_clf)
pipeline.fit(X_train, y_train)
accuracy = pipeline.score(X_test, y_test)
```

This code is very similar to the regression code we used earlier, but there are a few differences:

- Of course, it's a classification task so we use an `MLPClassifier` rather than an `MLPRegressor`.
- We use just two hidden layers with 300 and 100 neurons respectively. You can try a different number of hidden layers, and change the number of neurons as well if you want.
- We also use a `MinMaxScaler` instead of a `StandardScaler`. We need it to shrink the pixel intensities down to the 0-1 range rather than 0-255: having features in this range usually works better with the default hyperparameters used by `MLPClassifier`, such as its default learning rate and weight initialization scale. You might wonder why we didn't use a `StandardScaler`? Well some pixels don't vary much across images, for example the pixels

around the edges are almost always white. If we used the `StandardScaler`, these pixels would get scaled up to have the same variance as every other pixel: as a result, we would give more importance to these pixels than they probably deserve. Using the `MinMaxScaler` often works better than the `StandardScaler` for images (but your mileage may vary).

- Lastly, the `score()` function returns the model's accuracy.

If you run this code, you will find that the model reaches over 90% accuracy on the validation set during training (the exact value is given by `mlp_clf.best_validation_score_`), but it starts overfitting a bit towards the end so it ends up at just 89.9% accuracy. When we evaluate the model on the test set, we get 89.3%, which is not bad at all for this task, although we can do better with other neural net architectures such as convolutional neural networks ([Link to Come]).

You probably noticed that training was quite slow. That's because the hidden layers have a *lot* of parameters, so there are many computations to run at each iteration. For example, the first hidden layer has  $784 \times 300$  connection weights, plus 300 bias terms, which adds up to 235,500 parameters! All these parameters give the model quite a lot of flexibility to fit the training data, but it also means that there's a high risk of overfitting, especially when you do not have a lot of training data. This is why it's so important to use regularization techniques such as early stopping and  $\ell_2$  regularization.

Once the model is trained, you can use it to classify new images:

```
>>> X_new = X_test[:15] # let's pretend these are 15 new images
>>> mlp_clf.predict(X_new)
array([9, 2, 1, 1, 6, 1, 4, 6, 5, 7, 4, 5, 5, 3, 4])
```

All these predictions are correct, except for the one at index 12, which should be a 7 (sneaker) instead of a 5 (sandal). You might want to know how confident the model was about these predictions, especially the bad one. For this, you can use `model.predict_proba()` instead of `model.predict()`, like we did in [Chapter 3](#):

```
>>> y_proba = mlp_clf.predict_proba(X_new)
>>> y_proba[12]
array([0., 0., 0., 0., 0., 1., 0., 0., 0., 0.])
```

Mmmh, that's not great: the model is telling us that it's 100% confident that the image represents a sandal (index 5). So not only is the model wrong, it's 100% confident that it's right. In fact, across all 10,000 images in the test set, there are only 33 images that

the model is less than 99.9% confident about, despite the fact that its accuracy is about 90%. That's why you should always treat estimated probabilities with a grain of salt: neural nets have a strong tendency to be overconfident, especially if they are trained for a bit too long.

### TIP

The targets for classification tasks can be class indices (e.g., 3) or class probabilities, typically one-hot vectors (e.g., [0, 0, 0, 1, 0, 0, 0, 0, 0]). But if your model tends to be overconfident, you can try the *label smoothing* technique:<sup>14</sup> reduce the target class's probability slightly (e.g., from 1 down to 0.9) and distribute the rest evenly across the other classes (e.g., [0.1/9, 0.1/9, 0.1/9, 0.9, 0.1/9, 0.1/9, 0.1/9, 0.1/9, 0.1/9]).

Still, getting 90% accuracy on Fashion MNIST is pretty good. You could get even better performance by fine-tuning the hyperparameters, for example using `RandomizedSearchCV`, as we did in [Chapter 2](#). However, the search space is quite large, so it helps to know roughly where to look.

## Hyperparameter Tuning Guidelines

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network architecture, but even in a basic MLP you can change the number of layers, the number of neurons and the type of activation function to use in each layer, the weight initialization logic, the type of optimizer to use, its learning rate, the batch size, and more. What are some good values for these hyperparameters?

### Number of Hidden Layers

For many problems, you can begin with a single hidden layer and get reasonable results. An MLP with just one hidden layer can theoretically model even the most complex functions, provided it has enough neurons. But deep networks have a much higher *parameter efficiency* than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets, allowing them to reach much better performance with the same amount of training data. This is because their layered structure enables them to reuse and compose features across multiple levels: for example, the first layer in a face classifier may learn to recognize low-level features such as dots, arcs or straight lines, while the second layer may learn to combine these

low-level features into higher-level features such as squares or circles, and the third layer may learn to combine these higher-level features into a mouth, an eye or a nose, and the top layer would then be able to use these top-level features to classify faces.

Not only does this hierarchical architecture help DNNs converge faster to a good solution, but it also improves their ability to generalize to new datasets. For example, if you have already trained a model to recognize faces in pictures and you now want to train a new neural network to recognize hairstyles, you can kickstart the training by reusing the lower layers of the first network. Instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the values of the weights and biases of the lower layers of the first network. This way the network will not have to learn from scratch all the low-level structures that occur in most pictures; it will only have to learn the higher-level structures (e.g., hairstyles). This is called *transfer learning*.

In summary, for many problems you can start with just one or two hidden layers and the neural network will work pretty well. For instance, you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total number of neurons, in roughly the same amount of training time. For more complex problems, you can ramp up the number of hidden layers until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers (or even hundreds, but not fully connected ones, as you will see in [Link to Come]), and they need a huge amount of training data. You will rarely have to train such networks from scratch: it is much more common to reuse parts of a pretrained state-of-the-art network that performs a similar task. Training will then be a lot faster and require much less data.

## Number of Neurons per Hidden Layer

The number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, the MNIST task requires  $28 \times 28 = 784$  inputs and 10 output neurons.

As for the hidden layers, it used to be common to size them to form a pyramid, with fewer and fewer neurons at each layer—the rationale being that many low-level features can coalesce into far fewer high-level features. A typical neural network for MNIST might have 3 hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned because it seems that using the same number of neurons in all hidden layers performs just as well in most

cases, or even better; plus, there is only one hyperparameter to tune, instead of one per layer. That said, depending on the dataset, it can sometimes help to make the first hidden layer a bit larger than the others.

Just like the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. Alternatively, you can try building a model with slightly more layers and neurons than you actually need, then use early stopping and other regularization techniques to prevent it from overfitting too much. Vincent Vanhoucke, a Waymo researcher and former Googler, has dubbed this the “stretch pants” approach: instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size. With this approach, you avoid bottleneck layers that could ruin your model. Indeed, if a layer has too few neurons, it will lack the computational capacity to model complex relationships, and it may not even have enough representational power to preserve all the useful information from the inputs. For example, if you apply PCA (introduced in [Chapter 7](#)) to the Fashion MNIST training set, you will find that you need 187 dimensions to preserve 95% of the variance in the data. So if you set the number of neurons in the first hidden layer to some greater number, say 200, you can be confident that this layer will not be a bottleneck. However, you don’t want to add too many neurons, or else the model will have too many parameters to optimize, and it will take more time and data to train.

### TIP

In general you will get more bang for your buck by increasing the number of layers rather than the number of neurons per layer.

That said, bottleneck layers are not always a bad thing. For example, limiting the dimensionality of the first hidden layers forces the neural net to keep only the most important dimensions, which can eliminate some of the noise in the data (but don’t go too far!). Also, having a bottleneck layer near the output layer can force the neural net to learn good representations of the data in the previous layers (i.e., packing more useful information in less space), which can help the neural net generalize, and can also be useful in and of itself for *representation learning*. We will get back to that in [Link to Come].

## Learning Rate

The learning rate is a hugely important hyperparameter. In general, the optimal learning rate is about half of the maximum learning rate (i.e., the learning rate above which the training algorithm diverges, as we saw in [Chapter 4](#)). One way to find a good learning rate is to train the model for a few hundred iterations, starting with a very low learning rate (e.g.,  $10^{-5}$ ) and gradually increasing it up to a very large value (e.g., 10). This is done by multiplying the learning rate by a constant factor at each iteration (e.g., by  $(10 / 10^{-5})^{1/500}$  to go from  $10^{-5}$  to 10 in 500 iterations). If you plot the loss as a function of the learning rate (using a log scale for the learning rate), you should see it dropping at first. But after a while, the learning rate will be too large, so the loss will shoot back up: the optimal learning rate is often a bit lower than the point at which the loss starts to climb (typically about 10 times lower than the turning point). You can then reinitialize your model and train it normally using this good learning rate.

### TIP

To change the learning rate during training when using Scikit-Learn, you must set the MLP’s `warm_start` hyperparameter to `True`, and fit the model one batch at a time using `partial_fit()`, much like we did with the `SGDRegressor` in [Chapter 4](#). Simply update the learning rate at each iteration.

## Batch Size

The batch size can have a significant impact on your model’s performance and training time. The main benefit of using large batch sizes is that hardware accelerators like GPUs can process them efficiently (as we will see in [Chapter 10](#)), so the training algorithm will see more instances per second. Therefore, many researchers and practitioners recommend using the largest batch size that can fit in *VRAM* (video RAM, i.e., the GPU’s memory). There’s a catch, though: large batch sizes can sometimes lead to training instabilities, especially with smaller models and at the beginning of training, and the resulting model may not generalize as well as a model trained with a small batch size. Yann LeCun once tweeted “Friends don’t let friends use mini-batches larger than 32”, citing a [2018 paper<sup>15</sup>](#) by Dominic Masters and Carlo Luschi which concluded that using small batches (from 2 to 32) was preferable because small batches led to better models in less training time.

However, other research points in the opposite direction. For example, in 2017, papers by [Elad Hoffer et al.<sup>16</sup>](#) and [Priya Goyal et al.<sup>17</sup>](#) showed that it is possible to use very large batch sizes (up to 8,192) along with various techniques such as warming up the

learning rate (i.e., starting training with a small learning rate, then ramping it up) and to obtain very short training times, without any generalization gap.

So one strategy is to use a large batch size, possibly with learning rate warmup, and if training is unstable or the final performance is disappointing, then try using a smaller batch size instead.

## Other Hyperparameters

Here are a few more hyperparameters you can tune if you have the computation budget and the time:

### *Optimizer*

Choosing a better optimizer than plain old mini-batch gradient descent (and tuning its hyperparameters) can help speed up training and sometimes reach better performance.

### *Activation function*

We discussed how to choose the activation function earlier in this chapter: in general, the ReLU activation function is a good default for all hidden layers. In some cases, replacing ReLU with another function can help.

### *Number of iterations*

In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

#### TIP

The optimal learning rate depends on the other hyperparameters—especially the batch size—so if you modify any hyperparameter, make sure to tune the learning rate again.

For more best practices regarding tuning neural network hyperparameters, check out the excellent [2018 paper<sup>18</sup>](#) by Leslie Smith. The [Deep Learning Tuning Playbook](#) is also well worth reading. *Machine Learning Yearning* by Andrew Ng also contains a wealth of practical advice.

Lastly, I highly recommend you go through exercise 1 at the end of this chapter. You will use a nice web interface to play with various neural network architectures and visualize their outputs. This will be very useful to better understand MLPs and grow a good intuition of the effects of each hyperparameter (number of layers and neurons, activation functions, and more).

This concludes our introduction to artificial neural networks and their implementation with Scikit-Learn. In the next chapter, we will switch to PyTorch, the leading open-source library for neural networks, and we will use it to train and run MLPs much faster by exploiting the power of Graphical Processing Units (GPUs). We will also start building more complex models, with multiple inputs and outputs.

## Exercises

1. This [neural network playground](#) is a great tool to build your intuitions without writing any code (it was built by the TensorFlow team, but there's nothing TensorFlow-specific about it; in fact, it doesn't even use TensorFlow). In this exercise, you will train several binary classifiers in just a few clicks, and tweak the model's architecture and its hyperparameters to gain some intuition on how neural networks work and what their hyperparameters do. Take some time to explore the following:
  - a. The patterns learned by a neural net. Try training the default neural network by clicking the Run button (top left). Notice how it quickly finds a good solution for the classification task. The neurons in the first hidden layer have learned simple patterns, while the neurons in the second hidden layer have learned to combine the simple patterns of the first hidden layer into more complex patterns. In general, the more layers there are, the more complex the patterns can be.
  - b. Activation functions. Try replacing the tanh activation function with a ReLU activation function, and train the network again. Notice that it finds a solution even faster, but this time the boundaries are linear. This is due to the shape of the ReLU function.
  - c. The risk of local minima. Modify the network architecture to have just one hidden layer with three neurons. Train it multiple times (to reset the network weights, click the Reset button next to the Play button).

Notice that the training time varies a lot, and sometimes it even gets stuck in a local minimum.

- d. What happens when neural nets are too small. Remove one neuron to keep just two. Notice that the neural network is now incapable of finding a good solution, even if you try multiple times. The model has too few parameters and systematically underfits the training set.
  - e. What happens when neural nets are large enough. Set the number of neurons to eight, and train the network several times. Notice that it is now consistently fast and never gets stuck. This highlights an important finding in neural network theory: large neural networks rarely get stuck in local minima, and even when they do these local optima are often almost as good as the global optimum. However, they can still get stuck on long plateaus for a long time.
  - f. The risk of vanishing gradients in deep networks. Select the spiral dataset (the bottom-right dataset under “DATA”), and change the network architecture to have four hidden layers with eight neurons each. Notice that training takes much longer and often gets stuck on plateaus for long periods of time. Also notice that the neurons in the highest layers (on the right) tend to evolve faster than the neurons in the lowest layers (on the left). This problem, called the *vanishing gradients* problem, can be alleviated with better weight initialization and other techniques, better optimizers (such as AdaGrad or Adam), or batch normalization (discussed in [Chapter 11](#)).
  - g. Go further. Take an hour or so to play around with other parameters and get a feel for what they do, to build an intuitive understanding about neural networks.
2. Draw an ANN using the original artificial neurons (like the ones in [Figure 9-3](#)) that computes  $A \oplus B$  (where  $\oplus$  represents the XOR operation). Hint:  $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$ .
  3. Why is it generally preferable to use a logistic regression classifier rather than a classic perceptron (i.e., a single layer of threshold logic units trained using the perceptron training algorithm)? How can you tweak a perceptron to make it equivalent to a logistic regression classifier?

4. Why was the sigmoid activation function a key ingredient in training the first MLPs?
5. Name three popular activation functions. Can you draw them?
6. Suppose you have an MLP composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.
  - a. What is the shape of the input matrix  $\mathbf{X}$ ?
  - b. What are the shapes of the hidden layer's weight matrix  $\mathbf{W}_h$  and bias vector  $\mathbf{b}_h$ ?
  - c. What are the shapes of the output layer's weight matrix  $\mathbf{W}_o$  and bias vector  $\mathbf{b}_o$ ?
  - d. What is the shape of the network's output matrix  $\mathbf{Y}$ ?
  - e. Write the equation that computes the network's output matrix  $\mathbf{Y}$  as a function of  $\mathbf{X}$ ,  $\mathbf{W}_h$ ,  $\mathbf{b}_h$ ,  $\mathbf{W}_o$ , and  $\mathbf{b}_o$ .
7. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST, how many neurons do you need in the output layer, and which activation function should you use? What about for getting your network to predict housing prices, as in [Chapter 2](#)?
8. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
9. Can you list all the hyperparameters you can tweak in a basic MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem?
10. Train a deep MLP on the CoverType dataset. You can load it using `sklearn.datasets.fetch_covtype()`. See if you can get over 93% accuracy on the test set by fine-tuning the hyperparameters, manually and/or using `RandomizedSearchCV`.

Solutions to these exercises are available at the end of this chapter’s notebook, at <https://homl.info/colab-p>.

---

- 1 You can get the best of both worlds by being open to biological inspirations without being afraid to create biologically unrealistic models, as long as they work well.
- 2 Warren S. McCulloch and Walter Pitts, “A Logical Calculus of the Ideas Immanent in Nervous Activity”, *The Bulletin of Mathematical Biology* 5, no. 4 (1943): 115–113.
- 3 They are not actually attached, just so close that they can very quickly exchange chemical signals.
- 4 Image by Bruce Blaus (Creative Commons 3.0). Reproduced from <https://en.wikipedia.org/wiki/Neuron>.
- 5 In the context of machine learning, the phrase “neural networks” generally refers to ANNs, not BNNs.
- 6 Drawing of a cortical lamination by S. Ramon y Cajal (public domain). Reproduced from [https://en.wikipedia.org/wiki/Cerebral\\_cortex](https://en.wikipedia.org/wiki/Cerebral_cortex).
- 7 Logistic regression and the logistic function were introduced in [Chapter 4](#), along with several other concepts that we will heavily rely on in this chapter, including softmax, cross-entropy, gradient descent, early stopping, and more, so please make sure to read it first.
- 8 In some libraries, such as PyTorch, the weight matrix is transposed, so there’s one row per neuron, and one column per input feature.
- 9 Note that this solution is not unique: when data points are linearly separable, there is an infinity of hyperplanes that can separate them.
- 10 For example, when the inputs are  $(0, 1)$  the lower-left neuron computes  $0 \times 1 + 1 \times 1 - 3 / 2 = -1 / 2$ , which is negative, so it outputs 0. The lower-right neuron computes  $0 \times 1 + 1 \times 1 - 1 / 2 = 1 / 2$ , which is positive, so it outputs 1. The output neuron receives the outputs of the first two neurons as its inputs, so it computes  $0 \times (-1) + 1 \times 1 - 1 / 2 = 1 / 2$ . This is positive, so it outputs 1.
- 11 In the 1990s, an ANN with more than two hidden layers was considered deep. Nowadays, it is common to see ANNs with dozens of layers, or even hundreds, so the definition of “deep” is quite fuzzy.
- 12 David Rumelhart et al., “Learning Internal Representations by Error Propagation” (Defense Technical Information Center technical report, September 1985).
- 13 Biological neurons seem to implement a roughly sigmoid (S-shaped) activation function, so researchers stuck to sigmoid functions for a very long time. But it turns out that ReLU generally works better in ANNs. This is one of the cases where the biological analogy was perhaps misleading.
- 14 C. Szegedy et al., “Rethinking the Inception Architecture for Computer Vision”, CVPR 2016: 2818-2826.
- 15 Dominic Masters and Carlo Luschi, “Revisiting Small Batch Training for Deep Neural Networks”, arXiv preprint arXiv:1804.07612 (2018).
- 16 Elad Hoffer et al., “Train Longer, Generalize Better: Closing the Generalization Gap in Large Batch Training of Neural Networks”, *Proceedings of the 31st International Conference on Neural Information Processing*

*Systems* (2017): 1729–1739.

- <sup>17</sup> Priya Goyal et al., “Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour”, arXiv preprint arXiv:1706.02677 (2017).
- <sup>18</sup> Leslie N. Smith, “A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay”, arXiv preprint arXiv:1803.09820 (2018).

# Chapter 10. Building Neural Networks with PyTorch

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 10th chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

PyTorch is a powerful open-source deep learning library developed by Facebook’s AI Research lab (FAIR, now called Meta AI). It is the Python successor of the Torch library, originally written in the Lua programming language. With PyTorch, you can build all sorts of neural network models, and train them at scale using GPUs (or other hardware accelerators, as we will see). In many ways it is similar to NumPy, except it also supports hardware acceleration and autodiff (see [Chapter 9](#)), and includes optimizers and ready-to-use neural net components.

When PyTorch was released in 2016, Google’s TensorFlow library was by far the most popular: it was fast, it scaled well, and it could be deployed across many platforms. But its programming model was complex and static, making it difficult to use and debug. In contrast, PyTorch was designed from the ground up to provide a more flexible, Pythonic approach to building neural networks: in particular, as you will see, it uses dynamic computation graphs (also known as define-by-run), making it intuitive and easy to debug. PyTorch is also beautifully coded and documented, and focuses on its core task: making it easy to build and train high performance neural networks. Last but not least, it leans strongly into the open source culture and benefits from an enthusiastic and dedicated community, and a rich ecosystem. In September 2022, PyTorch’s governance was even transferred to the PyTorch Foundation, a subsidiary of the Linux Foundation. All these qualities resonated well with researchers: PyTorch quickly became the most used framework in academia, and once a majority of deep learning papers were based on PyTorch, a large part of the industry was gradually converted as well.<sup>1</sup>

In this chapter, you will learn how to train, evaluate, fine-tune, optimize, and save neural nets with PyTorch. We will start by getting familiar with the core building blocks of PyTorch, namely tensors and autograd, next we will test the waters by building and training a simple linear regression model, and then we will upgrade this model to a multilayer neural network, first for regression, then for classification. On the way, we will see how to build custom neural networks with multiple inputs or outputs. Finally, we will discuss how to automatically fine-tune hyperparameters using the Optuna library, and how to optimize and export your models. Hop onboard, we're diving into deep learning!

### NOTE

Colab runtimes come with a recent version of PyTorch preinstalled. However, if you prefer to install it on your own machine, please see the installation instructions at <https://homl.info/install-p>.

## PyTorch Fundamentals

The core building block of PyTorch is the **tensor**.<sup>2</sup> It's a multidimensional array with a shape and a data type, used for numerical computations. Isn't that exactly like a NumPy array? Well, yes, it is! But a tensor also has two extra features: it can live on a GPU (or other hardware accelerators, as we will see), and it supports auto-differentiation. Let's start by looking at the basic NumPy-like features.

## PyTorch Tensors

First, let's import the PyTorch library:

```
>>> import torch
```

Next you can create a PyTorch tensor much like you would create a NumPy array. For example, let's create a  $2 \times 3$  array:

```
>>> X = torch.tensor([[1.0, 4.0, 7.0], [2.0, 3.0, 6.0]])
>>> X
tensor([[1., 4., 7.],
        [2., 3., 6.]])
```

Just like a NumPy array, a tensor can contain floats, integers, booleans, or complex numbers—just one data type per tensor. If you initialize a tensor with values of different types, then the most general one will be selected (i.e., complex > float > integer > bool). You can also select the data type explicitly when creating the tensor, for example `dtype=torch.float16` for 16-bit floats. Note that tensors of strings or objects are not supported.

You can get a tensor’s shape and data type like this:

```
>>> X.shape  
torch.Size([2, 3])  
>>> X.dtype  
torch.float32
```

Indexing works just like for NumPy arrays:

```
>>> X[0, 1]  
tensor(4.)  
>>> X[:, 1]  
tensor([4., 3.])
```

You can also run all sorts of computations on tensors, and the API is conveniently similar to NumPy’s: for example, there’s `torch.abs()`, `torch.cos()`, `torch.exp()`, `torch.max()`, `torch.mean()`, `torch.sqrt()`, and so on. PyTorch tensors also have methods for most of these operations, so you can write `X.exp()` instead of `torch.exp(X)`. Let’s try a few operations:

```
>>> 10 * (X + 1.0) # itemwise addition and multiplication  
tensor([[20., 50., 80.],  
       [30., 40., 70.]])  
>>> X.exp() # itemwise exponential  
tensor([[ 2.7183,  54.5982, 1096.6332],  
       [ 7.3891, 20.0855, 403.4288]])  
>>> X.mean()  
tensor(3.8333)  
>>> X.max(dim=0) # max values along dimension 0 (i.e., max value per column)  
torch.return_types.max(values=tensor([2., 4., 7.]), indices=tensor([1, 0, 0]))  
>>> X @ X.T # matrix transpose and matrix multiplication  
tensor([[66., 56.],  
       [56., 49.]])
```

## NOTE

PyTorch prefers the argument name `dim` in operations such as `max()`, but it also supports `axis` (as in NumPy or Pandas).

You can also convert a tensor to a NumPy array using the `numpy()` method, and create a tensor from a NumPy array:

```
>>> import numpy as np
>>> X.numpy()
array([[1., 4., 7.],
       [2., 3., 6.]], dtype=float32)
>>> torch.tensor(np.array([[1., 4., 7.], [2., 3., 6.]]))
tensor([[1., 4., 7.],
       [2., 3., 6.]], dtype=torch.float64)
```

Notice that the default precision for floats is 32-bits in PyTorch, whereas it's 64-bits in NumPy. It's generally better to use 32-bits in deep learning because this takes half the RAM and speeds up computations, and neural nets do not actually need the extra precision offered by 64-bit floats. So when calling the `torch.tensor()` function to convert a NumPy array to a tensor, it's best to specify `dtype=torch.float32`.

Alternatively, you can use `torch.FloatTensor()` which automatically converts the array to 32-bits:

```
>>> torch.FloatTensor(np.array([[1., 4., 7.], [2., 3., 6.]]))
tensor([[1., 4., 7.],
       [2., 3., 6.]])
```

## TIP

Both `torch.tensor()` and `torch.FloatTensor()` make a copy of the given NumPy array. If you prefer, you can use `torch.from_numpy()` which creates a tensor on the CPU that just uses the NumPy array's data directly, without copying it. But beware: modifying the NumPy array will also modify the tensor, and vice versa.

You can also modify a tensor in-place using indexing and slicing, as with a NumPy array:

```
>>> X[:, 1] = -99
>>> X
```

```
tensor([[ 1., -99.,  7.],
       [ 2., -99.,  6.]])
```

PyTorch's API provides many in-place operations, such as `abs_()`, `sqrt_()`, `zero_()`, which modify the input tensor directly: they can sometimes save some memory and speed up your models. For example, the `relu_()` method applies the ReLU activation function in-place by replacing all negative values with 0s:

```
>>> X.relu_()
>>> X
tensor([[1., 0., 7.],
       [2., 0., 6.]])
```

### TIP

PyTorch's in-place operations are easy to spot at a glance because their name always ends with an underscore. With very few exceptions (e.g., `zero_()`), removing the underscore gives you the regular operation (e.g., `abs_()` is in-place, `abs()` is not).

We will cover many more operations as we go, but now let's look at how to use hardware acceleration to make computations much faster.

## Hardware Acceleration

PyTorch tensors can be copied easily to the GPU, assuming your machine has a compatible GPU, and you have the required libraries installed. On Colab, all you need to do is ensure that you are using a GPU runtime: for this, go to the *Runtime* menu and select *Change runtime type*, then make sure a GPU is selected (e.g., an Nvidia T4 GPU). The GPU runtime will automatically have the appropriate PyTorch library installed—compiled with GPU support—as well as the appropriate GPU drivers and any other required library (e.g., Nvidia's CUDA and cuDNN libraries).<sup>3</sup> If you prefer to run the code on your own machine, you will need to ensure that you have all the drivers and libraries required. Please follow the instructions at <https://hml.info/install-p>.

PyTorch has excellent support for Nvidia GPUs, as well as several other hardware accelerators:

- Apple's *Metal Performance Shaders* (MPS) to accelerate computations on Apple silicon such as the M1, M2, and later chips, as well as some Intel Macs with a compatible GPU.

- AMD Instinct accelerators and AMD Radeon GPUs, through the ROCm software stack, or via DirectML on Windows.
- Intel GPUs and CPUs on Linux and Windows via Intel's oneAPI.
- Google TPUs via the `torch_xla` library.

## WARNING

Deep learning without a hardware accelerator can be incredibly slow, especially once we start diving into computer vision and natural language processing, in the following chapters. If you do not have a hardware accelerator, try using Colab or Kaggle, they offer runtimes with free GPUs. Or consider using other cloud services. Or prepare to be very, very patient.

Let's check whether PyTorch can access an Nvidia GPU or Apple's MPS, otherwise let's fallback to the CPU:

```
if torch.cuda.is_available():
    device = "cuda"
elif torch.backends.mps.is_available():
    device = "mps"
else:
    device = "cpu"
```

On a Colab GPU Runtime, `device` will be equal to "`cuda`". Now let's create a tensor on that GPU. To do that, one option is to create the tensor on the CPU, then copy it to the GPU using the `to()` method:

```
>>> M = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
>>> M = M.to(device)
```

## TIP

The `cpu()` and `cuda()` methods are short for `to("cpu")` and `to("cuda")` respectively.

You can always tell which device a tensor lives on by looking at its `device` attribute:

```
>>> M.device
device(type='cuda', index=0)
---
```

Alternatively, we can create the tensor directly on the GPU using the `device` argument:

```
>>> M = torch.tensor([[1., 2., 3.], [4., 5., 6.]], device=device)
```

### TIP

If you have multiple Nvidia GPUs, you can refer to the desired GPU by appending the GPU index: "`cuda:0`" (or just "`cuda`") for GPU #0, "`cuda:1`" for GPU #1, and so on.

Once the tensor is on the GPU, we can run operations on it normally, and they will all take place on the GPU:

```
>>> R = M @ M.T # run some operations on the GPU
>>> R
tensor([[14., 32.],
       [32., 77.]], device='cuda:0')
```

Note that the result `R` also lives on the GPU. This means we can perform multiple operations on the GPU without having to transfer data back and forth between the CPU and the GPU. This is crucial in deep learning because data transfer between devices can often become a performance bottleneck.

How much does a GPU accelerate the computations? Well it depends on the GPU, of course: the more expensive ones are dozens of times faster than the cheap ones. But speed alone is not the only important factor: the data throughput is also crucial, as we just saw. If your model is compute-heavy (e.g., a very deep neural net), the GPU's speed and amount of RAM will typically matter most, but if it is a shallower model, then pumping the training data into the GPU might become the bottleneck. Let's run a little test to compare the speed of a matrix multiplication running on the CPU vs the GPU:<sup>4</sup>

```
>>> M = torch.rand((1000, 1000)) # on the CPU
>>> %timeit M @ M.T
16.2 ms ± 3.24 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> M = torch.rand((1000, 1000), device="cuda") # on the GPU
>>> %timeit M @ M.T
605 µs ± 13.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Wow! The GPU gave us a  $26\times$  speed boost! And that's just using the free Nvidia T4 GPU on Colab, imagine the speedup we could get using a more powerful GPU. Now try playing around with the matrix size: you will notice that the speedup is much less

impressive on smaller matrices (e.g., it's just  $2\times$  for  $100\times 100$  matrices). That's because GPUs work by breaking large operations into smaller operations and running them in parallel across thousands of cores. If the task is small, it cannot be broken up into that many pieces, and the performance gain is therefore smaller. In fact, when running many tiny tasks, it can sometimes be faster to just run the operations on the CPU.

All right, now that we've seen what tensors are and how to use them on the CPU or the GPU, let's look at PyTorch's auto-differentiation feature.

## Autograd

PyTorch comes with an efficient implementation of reverse-mode auto-differentiation (introduced in [Chapter 9](#) and detailed in [Link to Come]), called *autograd*, which stands for automatic gradients. It is quite easy to use. For example, consider a simple function,  $f(x) = x^2$ . Differential calculus tells us that the derivative of this function is  $f'(x) = 2x$ . If we evaluate  $f(5)$  and  $f'(5)$ , we get 25 and 10 respectively. Let's see if PyTorch agrees:

```
>>> x = torch.tensor(5.0, requires_grad=True)
>>> f = x ** 2
>>> f
tensor(25., grad_fn=<PowBackward0>)
>>> f.backward()
>>> x.grad
tensor(10.)
```

Great, we got the correct results:  $f$  is 25, and  $x.grad$  is 10! Note that the `backward()` function automatically computed the gradient  $f(x)$  at the same point  $x = 5.0$ . Let's go through this code line by line:

- First, we created a tensor  $x$ , equal to 5.0, and we told PyTorch that it's a variable (not a constant) by specifying `requires_grad=True`. Knowing this, PyTorch will automatically keep track of all operations involving  $x$ : this is needed because PyTorch must capture the computation graph in order to run backprop on it and obtain the derivative of  $f$  with regards to  $x$ . In this computation graph, the tensor  $x$  is a *leaf node*.
- Then we compute  $f = x ** 2$ . The result is a tensor equal to 25.0, the square of 5.0. But wait, there's more to it:  $f$  also carries a `grad_fn` attribute which represents the operation that created this tensor (`**`, power, hence the name `PowBackward0`), and which tells PyTorch how to backpropagate the gradients

through this particular operation. This `grad_fn` attribute is how PyTorch keeps track of the computation graph.

- Next, we call `f.backward()`: this backpropagates the gradients through the computation graph, starting with `f`, and all the way back to the leaf nodes (just `x` in this case).
- Lastly, we can just read the `x` tensor's `grad` attribute, which was computed during backprop: this gives us the derivative of `f` with regards to `x`. Ta-da!

PyTorch creates a new computation graph on the fly during each forward pass, as the operations are executed. This allows PyTorch to support very dynamic models containing loops and conditionals.

### WARNING

The way PyTorch accumulates gradients in each variable's `grad` attribute can be surprising at first, especially coming from TensorFlow or JAX. In these frameworks, computing the gradients of `f` with regards to `x` just returns the gradients, without affecting `x`. In PyTorch, if you call `backward()` on a tensor, it will accumulate the gradients in every variable that was used to compute it. So if you call `backward()` on two tensors `t1` and `t2` that both used the same variable `v`, then `v.grad` will be the sum of their gradients.

After computing the gradients, you generally want to perform a gradient descent step by subtracting a fraction of the gradients from the model variables (at least when training a neural network). In our simple example, running gradient descent will gradually push `x` towards 0, since that's the value that minimizes  $f(x) = x^2$ . To do a gradient descent step, you must temporarily disable gradient tracking since you don't want to track the gradient descent step itself in the computation graph (in fact, PyTorch would raise an exception if you tried to run an in-place operation on a tracked variable). This can be done by placing the gradient descent step inside a `torch.no_grad()` context, like this:

```
learning_rate = 0.1
with torch.no_grad():
    x -= learning_rate * x.grad # gradient descent step
```

The variable `x` gets decremented by  $0.1 * 10.0 = 1.0$ , down from 5.0 to 4.0.

Another way to avoid gradient computation is to use the variable's `detach()` method: this creates a new tensor detached from the computation graph, with

`requires_grad=False`, but still pointing to the same data in memory. You can then update this detached tensor:

```
x_detached = x.detach()  
x_detached -= learning_rate * x.grad
```

Since `x_detached` and `x` share the same memory, modifying `x_detached` also modifies `x`.

The `detach()` method can be handy when you need to run some computation on a tensor without affecting the gradients (e.g., for evaluation or logging), or when you need fine-grained control over which operations should contribute to gradient computation. Using `no_grad()` is generally preferred when performing inference or doing a gradient descent step, as it provides a convenient context-wide method to disable gradient tracking.

Lastly, before you repeat the whole process (forward pass + backward pass + gradient descent step), it's essential to zero out the gradients of every model parameter (you don't need a `no_grad()` context for this since the gradient tensor has `requires_grad=False`):

```
x.grad.zero_()
```

### WARNING

If you forget to zero out the gradients at each training iteration, the `backward()` method will just accumulate them, causing incorrect gradient descent updates. Since there won't be any explicit error, just low performance (and perhaps infinite or NaN values), this issue may be hard to debug.

Putting everything together, the whole training loop looks like this:

```
learning_rate = 0.1  
x = torch.tensor(5.0, requires_grad=True)  
for iteration in range(100):  
    f = x ** 2 # forward pass  
    f.backward() # backward pass  
    with torch.no_grad():  
        x -= learning_rate * x.grad # gradient descent step  
  
    x.grad.zero_() # reset the gradients
```

If you want to use in-place operations to save memory and speed up your models a bit by avoiding unnecessary copy operations, you have to be careful: in-place operations don't always play nicely with autograd. Firstly, as we saw earlier, you cannot apply an in-place operation to a leaf node (i.e., a tensor with `requires_grad=True`), as PyTorch wouldn't know where to store the computation graph. For example `x.cos_()` or `x += 1` would cause a `RuntimeError`. Secondly, consider the following code, which computes  $z(t) = \exp(t) + 1$  at  $t = 2$  and then tries to compute the gradients:

```
t = torch.tensor(2.0, requires_grad=True)
z = t.exp() # this is an intermediate result
z += 1 # this is an in-place operation
z.backward() # △ RuntimeError!
```

Oh no! Although `z` is computed correctly, the last line causes a `RuntimeError`, complaining that "one of the variables needed for gradient computation has been modified by an in-place operation". Indeed, the intermediate result `z = t.exp()` was lost when we ran the in-place operation `z += 1`, so when the backward pass reached the exponential operation, the gradients could not be computed. A simple fix is to replace `z += 1` with `z = z + 1`. It looks similar, but it's no longer an in-place operation: a new tensor is created and assigned to the same variable, but the original tensor is unchanged and recorded in the computation graph of the final tensor.

Surprisingly, if you replace `exp()` with `cos()` in the previous code example, the gradients will be computed correctly: no error! Why is that? Well, the outcome depends on the way each operation is implemented:

- Some operations—such as `exp()`, `relu()`, `rsqrt()`, `sigmoid()`, `sqrt()`, `tan()`, `tanh()`—save their outputs in the computation graph during the forward pass, then use these outputs to compute the gradients during the backward pass.<sup>5</sup> This means that you must not modify such an operation's output in-place, or else you will get an error during the backward pass (as we just saw).
- Other operations—such as `abs()`, `cos()`, `log()`, `sin()`, `square()`, `var()`—save their inputs instead of their output.<sup>6</sup> Such an operation doesn't care if you modify its output in-place, but you must not modify its inputs in-place before the backward pass (e.g., to compute something else based on the same inputs).
- Some operations—such as `max()`, `min()`, `norm()`, `prod()`, `sgn()`, `std()`—save both the inputs and the outputs, so you must not modify either of them in-place before the backward pass.

- Lastly, a few operations—such as `ceil()`, `floor()`, `mean()`, `round()`, `sum()`—save neither their inputs nor their outputs.<sup>7</sup> You can safely modify them in-place.

### TIP

Implement your models first without any in-place operations, then if you need to save some memory or speed up your model a bit, you can try converting some of the most costly operations to their in-place counterparts. Just make sure that your model still outputs the same result for a given input, and also make sure you don't modify in-place a tensor needed for backprop (you will get a `RuntimeError` in this case).

OK, let's step back a bit. We've discussed all the fundamentals of PyTorch: how to create tensors and use them to perform all sorts of computations, how to accelerate the computations with a GPU, and how to use autograd to compute gradients for gradient descent. Great! Now let's apply what we've learned so far by building and training a simple linear regression model with PyTorch.

## Implementing Linear Regression

We will start by implementing linear regression using tensors and autograd directly, then we will simplify the code using PyTorch's high-level API, and also add GPU support.

### Linear Regression Using Tensors & Autograd

Let's tackle the same California housing dataset as in [Chapter 9](#). I will assume you have already downloaded it using `sklearn.datasets.fetch_california_housing()`, and you have split it into a training set (`X_train` and `y_train`), a validation set (`X_valid` and `y_valid`), and a test set (`X_test` and `y_test`), using `sklearn.model_selection.train_test_split()`. Next, let's convert it to tensors and normalize it. We could use a `StandardScaler` for this, like we did in [Chapter 9](#), but let's just use tensor operations instead, to get a bit of practice:

```
X_train = torch.FloatTensor(X_train)
X_valid = torch.FloatTensor(X_valid)
X_test = torch.FloatTensor(X_test)
means = X_train.mean(dim=0, keepdims=True)
stds = X_train.std(dim=0, keepdims=True)
X_train = (X_train - means) / stds
```

```
X_valid = (X_valid - means) / stds  
X_test = (X_test - means) / stds
```

Let's also convert the targets to tensors. Since our predictions will be column vectors (i.e., matrices with a single column), we need to ensure that our targets are also column vectors.<sup>8</sup> Unfortunately, the NumPy arrays representing the targets are one-dimensional so we need to reshape the tensors to column vectors by adding a second dimension of size 1:<sup>9</sup>

```
y_train = torch.FloatTensor(y_train).reshape(-1, 1)  
y_valid = torch.FloatTensor(y_valid).reshape(-1, 1)  
y_test = torch.FloatTensor(y_test).reshape(-1, 1)
```

Now that the data is ready, let's create the parameters of our linear regression model:

```
torch.manual_seed(42)  
n_features = X_train.shape[1] # there are 8 input features  
w = torch.randn((n_features, 1), requires_grad=True)  
b = torch.tensor(0., requires_grad=True)
```

We now have a weights parameter  $w$  (a column vector with one weight per input dimension, in this case 8), and a bias parameter  $b$  (a single scalar). The weights are initialized randomly, while the bias is initialized to zero. We could have initialized the weights to zero as well in this case, but when we get to neural networks it will be important to initialize the weights randomly to break the symmetry between neurons (as explained in [Chapter 9](#)), so we might as well get into the habit now.

## WARNING

We called `torch.manual_seed()` to ensure that the results are reproducible. However, PyTorch does not guarantee perfectly reproducible results across different releases, platforms, or devices, so if you do not run the code in this chapter with PyTorch 2.5 on a Colab Runtime with an NVidia T4 GPU, you may get different results. Moreover, since a GPU splits each operation into multiple chunks and runs them in parallel, the order in which these chunks finish may vary across runs, and this may slightly affect the result due to floating point precision errors. These minor differences may compound during training, and lead to very different models. To reduce this risk, you can tell PyTorch to use only deterministic algorithms by calling `torch.use_deterministic_algorithms(True)`. However, deterministic algorithms are often slower than stochastic ones, and some operations don't have a deterministic version at all, so you will get an error if your code tries to use one.

Next, let's train our model, very much like we did in [Chapter 4](#), except we will use autodiff to compute the gradients rather than using a closed-form equation. For now we will use Batch Gradient Descent, using the full training set at each training step:

```
learning_rate = 0.4
n_epochs = 20
for epoch in range(n_epochs):
    y_pred = X_train @ w + b
    loss = ((y_pred - y_train) ** 2).mean()
    loss.backward()
    with torch.no_grad():
        b -= learning_rate * b.grad
        w -= learning_rate * w.grad
        b.grad.zero_()
        w.grad.zero_()
    print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}")
```

Let's walk through this code:

- First we define the `learning_rate` hyperparameter. You can experiment with different values to find a value that converges fast and gives a precise result.
- Next, we run 20 epochs. We could implement early stopping to find the right moment to stop and avoid overfitting, like we did in [Chapter 4](#), but we will keep things simple for now.
- Next we run the forward pass: we compute the predictions `y_pred`, and the mean squared error `loss`.
- Then we run `loss.backward()` to compute the gradients of the loss with regards to every model parameter. This is autograd in action.
- Next we use the gradients `b.grad` and `w.grad` to perform a gradient descent step. Notice that we're running this code inside a `with torch.no_grad()` context, as discussed earlier.
- Once we've done the gradient descent step, we reset the gradients to zero (very important!).
- Lastly, we print the epoch number and the current loss at each epoch. The `item()` method extracts the value of a scalar tensor.

And that's it, if you run this code, you should see the training loss going down like this:

```
Epoch 1/20, Loss: 16.158458709716797  
Epoch 2/20, Loss: 4.879374027252197  
Epoch 3/20, Loss: 2.255225896835327  
[...]  
Epoch 20/20, Loss: 0.5684100389480591
```

Congratulations, you just trained your first model using PyTorch! You can now use the model to make predictions for some new data `X_new` (which must be represented as a PyTorch tensor). For example, let's make predictions for the first 3 instances in the test set:

```
>>> X_new = X_test[:3] # pretend these are new instances  
>>> with torch.no_grad():  
...     y_pred = X_new @ w + b # use the trained parameters to make predictions  
...  
>>> y_pred  
tensor([[0.8916],  
       [1.6480],  
       [2.6577]])
```

### TIP

It's best to use a `with torch.no_grad()` context during inference: PyTorch will consume less RAM and run faster since it won't have to keep track of the computation graph.

Implementing linear regression using PyTorch's low-level API wasn't too hard, but using this approach for more complex models would get really messy and difficult. So PyTorch offers a higher-level API to simplify all this. Let's rewrite our model using this higher-level API.

## Linear Regression Using PyTorch's High-Level API

PyTorch provides an implementation of linear regression in the `torch.nn.Linear` class, so let's use it:

```
import torch.nn as nn # by convention, this module is usually imported this way  
  
torch.manual_seed(42) # to get reproducible results  
model = nn.Linear(in_features=n_features, out_features=1)
```

The `nn.Linear` class (short for `torch.nn.Linear`) is one of many *modules* provided by PyTorch. Each module is a subclass of the `nn.Module` class. To build a simple

linear regression model, a single `nn.Linear` module is all you need. However, for most neural networks you will need to assemble many modules, as we will see later in this chapter, so you can think of modules as math LEGO® bricks. Many modules contain model parameters. For example, the `nn.Linear` module contains a `bias` vector (with one bias term per neuron), and a `weight` matrix (with one row per neuron and one column per input dimension, which is the transpose of the weight matrix we used earlier and in [Equation 9-2](#)). Since our model has a single neuron (because `out_features=1`), the `bias` vector contains a single bias term, and the `weight` matrix contains a single row. These parameters are accessible directly as attributes of the `nn.Linear` module:

```
>>> model.bias
Parameter containing:
tensor([0.3117], requires_grad=True)
>>> model.weight
Parameter containing:
tensor([[ 0.2703,  0.2935, -0.0828,  0.3248, -0.0775,  0.0713, -0.1721,  0.2076]], 
      requires_grad=True)
```

Notice that both parameters were automatically initialized randomly (which is why we used `manual_seed()` to get reproducible results). These parameters are instances of the `torch.nn.Parameter` class, which is a subclass of the `tensor.Tensor` class: this means that you can use them exactly like normal tensors. A module's `parameters()` method returns an iterator over all of the module's attributes of type `Parameter`, as well as all the parameters of all its submodules, recursively (if it has any). It does *not* return regular tensors, even those with `requires_grad=True`: that's the main difference between a regular tensor and a `Parameter`.

```
>>> for param in model.parameters():
...     [...] # do something with each parameter
```

There's also a `named_parameters()` method that returns an iterator over pairs of parameter names and values.

A module can be called just like a regular function. For example, let's make some predictions for the first two instances in the training set (since the model is not trained yet, its parameters are random and the predictions are terrible).

```
>>> model(X_train[:2])
tensor([-0.4718,
       0.1131], grad_fn=<AddmmBackward0>)
```

When we use a module as a function, PyTorch internally calls the module's `forward()` method. In the case of the `nn.Linear` module, the `forward()` method computes  $X @ \text{self.weight.T} + \text{self.bias}$  (where  $X$  is the input). That's just what we need for linear regression!

Notice that the result contains the `grad_fn` attribute, showing that autograd did its job and tracked the computation graph while the model was making its predictions.

### TIP

If you pass a custom function to a module's `register_forward_hook()` method, it will be called automatically every time the module itself is called. This is particularly handy for logging or debugging. To remove a hook, just call the `remove()` method on the object returned by `register_forward_hook()`. Note that hooks only work if you call the model like a function, not if you call its `forward()` method directly (which is why you should never do that). You can also register functions to run during the backward pass using `register_backward_hook()`.

Now that we have our model, we need to create an optimizer to update the model parameters, and we must also choose a loss function:

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
mse = nn.MSELoss()
```

PyTorch provides a few different optimizers (we will discuss them in the next chapter). Here we're using the simple Stochastic Gradient Descent (SGD) optimizer, which can be used for SGD, mini-batch GD, or Batch Gradient Descent. To initialize it, we must give it the model parameters and the learning rate.

For the loss function, we create an instance of the `nn.MSELoss` class: this is also a module, so we can use it like a function, giving it the predictions and the targets, and it will compute the MSE. The `nn` module contains many other loss functions and other neural net tools, as we will see. Next, let's write a small function to train our model:

```
def train_bgd(model, optimizer, criterion, X_train, y_train, n_epochs):
    for epoch in range(n_epochs):
        y_pred = model(X_train)
        loss = criterion(y_pred, y_train)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}")
```

Compare this training loop with our earlier training loop: it's very similar, but we're now using higher level constructs rather than working directly with tensors and autograd. Here are a few things to note:

- In PyTorch, the loss function object is commonly referred to as the *criterion*, to distinguish it from the loss value itself (which is computed at each training iteration using the criterion). In this example, it's the `MSELoss` instance.
- The `optimizer.step()` line corresponds to the two lines that updated `b` and `w` in our earlier code.
- And of course the `optimizer.zero_grad()` line corresponds to the two lines that zeroed out `b.grad` and `w.grad`. Notice that we don't need to use `with torch.no_grad()` here since this is done automatically by the optimizer, inside the `step()` and `zero_grad()` functions.

Now let's call this function to train our model!

```
>>> train_bgd(model, optimizer, mse, X_train, y_train, n_epochs)
Epoch 1/20, Loss: 4.3378496170043945
Epoch 2/20, Loss: 0.780293345451355
[...]
Epoch 20/20, Loss: 0.5374288558959961
```

All good, the model is trained, you can now use it to make predictions by simply calling it like a function (preferably inside a `no_grad()` context, as we saw earlier):

```
>>> X_new = X_test[:3] # pretend these are new instances
>>> with torch.no_grad():
...     y_pred = model(X_new) # use the trained model to make predictions
...
>>> y_pred
tensor([[0.8061],
        [1.7116],
        [2.6973]])
```

These predictions are similar to the ones our previous model made, but not exactly the same: that's because the `nn.Linear` module initializes the parameters slightly differently: it uses a uniform random distribution from  $-1/2\sqrt{2}$  to  $+1/2\sqrt{2}$  for both the weights and the bias term (we will discuss initialization methods in [Chapter 11](#)).

Now that you are familiar with PyTorch's high-level API, you are ready to go beyond linear regression and build a multilayer perceptron (introduced in [Chapter 9](#)).

# Implementing a Regression MLP

PyTorch provides a helpful `nn.Sequential` module that chains multiple modules: when you call this module with some inputs, it feeds these inputs to the first module, then feeds the output of the first module to the second module, and so on. Most neural networks contain stacks of modules, and in fact many neural networks are just one big stack of modules: this makes the `nn.Sequential` module one of the most useful modules in PyTorch. The MLP we want to build is just that: a simple stack of modules—two hidden layers and one output layer. So let's build it using the `nn.Sequential` module:

```
torch.manual_seed(42)
model = nn.Sequential(
    nn.Linear(n_features, 50),
    nn.ReLU(),
    nn.Linear(50, 40),
    nn.ReLU(),
    nn.Linear(40, 1)
)
```

Let's go through each layer:

- The first layer must have the right number of inputs for our data: `n_features` (equal to 8 in our case). However, it can have any number of outputs: let's pick 50 (that's a hyperparameter we can tune).
- Next we have a `nn.ReLU` module, which implements the ReLU activation function for the first hidden layer. This module does not contain any model parameters, and it acts itemwise so its output's shape is equal to its input's shape.
- The second hidden layer must have the same number of inputs as the output of the previous layer: in this case, 50. However, it can have any number of outputs. It's common to use the same number of output dimensions in all hidden layers, but in this example I used 40 to make it clear that the output of one layer must match the input of the next layer.
- Then again a `nn.ReLU` module, to implement the second hidden layer's activation function.
- Finally, the output layer must have 40 inputs, but this time its number of outputs is not free: it must match the targets' dimensionality. Since our targets have a

single dimension, we must have just 1 output dimension in the output layer.

Now let's train the model just like we did before:

```
>>> learning_rate = 0.1
>>> optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
>>> mse = nn.MSELoss()
>>> train_bgd(model, optimizer, mse, X_train, y_train, n_epochs)
Epoch 1/20, Loss: 5.045480251312256
Epoch 2/20, Loss: 2.0523128509521484
[...]
Epoch 20/20, Loss: 0.565444827079773
```

That's it, you can tell your friends you trained your first neural network with PyTorch! However, we are still using batch gradient descent (BGD), computing the gradients over the entire training set at each iteration. This works with small datasets, but if we want to be able to scale up to large datasets and large models, we need to switch to mini-batch GD.

## Implementing Mini-Batch Gradient Descent using DataLoaders

To help implement mini-batch GD, PyTorch provides a class named `DataLoader` in the `torch.utils.data` module. It can efficiently load batches of data of the desired size, and shuffle the data at each epoch if we want it to. The `DataLoader` expects the dataset to be represented as an object with at least two methods: `__len__(self)` to get the number of samples in the dataset, and `__getitem__(self, index)` to load the sample at the given index (including the target).

In our case, the training set is available in the `X_train` and `y_train` tensors, so we first need to wrap these tensors in a dataset object with the required API. To help with this, PyTorch provides a `TensorDataset` class. So let's build a `TensorDataset` to wrap our training set, and a `DataLoader` to pull batches from this dataset. During training, we want the dataset to be shuffled, so we specify `shuffle=True`:

```
from torch.utils.data import TensorDataset, DataLoader

train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

Now that we have a larger model and we have the tools to train it one batch at a time, it's a good time to start using hardware acceleration. It's really quite simple: we just need to move the model to the GPU, which will move all of its parameters to the GPU RAM, and then at the start of each iteration during training we must copy each batch to the GPU. To move the model, we can just use its `to()` method, just like we did with tensors:

```
torch.manual_seed(42)
model = nn.Sequential([...]) # create the model just like earlier
model = model.to(device)
```

We can also create the loss function and optimizer, as earlier (but using a lower learning rate, such as 0.02).

### WARNING

Some optimizers have some internal state, as we will see in [Chapter 11](#). The optimizer will usually allocate its state on the same device as the model parameters, so it's important to create the optimizer *after* you have moved the model to the GPU.

Now let's create a `train()` function to implement mini-batch GD:

```
def train(model, optimizer, criterion, train_loader, n_epochs):
    model.train()
    for epoch in range(n_epochs):
        total_loss = 0.
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            loss = criterion(y_pred, y_batch)
            total_loss += loss.item()
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

        mean_loss = total_loss / len(train_loader)
        print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {mean_loss:.4f}")
```

At every epoch, the function iterates through the whole training set, one batch at a time, and processes each batch just like earlier. But what about the very first line: `model.train()`? Well, this switches the model and all of its submodules to *training mode*. For now, this makes no difference at all, but it will be important in [Chapter 11](#)

when we start using layers that behave differently during training and evaluation (e.g., `nn.Dropout` or `nn.BatchNorm1d`). Whenever you want to use the model outside of training (e.g., for evaluation, or to make predictions on new instances), you must first switch the model to *evaluation mode* by running `model.eval()`. Note that `model.training` holds a boolean that indicates the current mode.

### TIP

PyTorch itself does not provide a training loop implementation, you have to build it yourself. As we just saw, it's not that long, and many people enjoy the freedom, clarity, and control this provides. However, if you would prefer to use a well-tested, off-the-shelf training loop with all the bells and whistles you need (such as multi-GPU support), then you can use a library such as *PyTorch Lightning*, *FastAI*, *Catalyst* or *Keras*. These libraries are built on top of PyTorch and include a training loop and many other features (*Keras* supports PyTorch since version 3, and also supports TensorFlow and JAX). Check them out!

Now let's call this `train()` function to train our model on the GPU:

```
>>> train(model, optimizer, mse, train_loader, n_epochs)
Epoch 1/20, Loss: 0.6958
Epoch 2/20, Loss: 0.4480
[...]
Epoch 20/20, Loss: 0.3227
```

It worked great: we actually reached a much lower loss in the same number of epochs! However, you probably noticed that each epoch was much slower. There are two easy tweaks you can make to considerably speed up training:

- If you are using a CUDA device, you should generally set `pin_memory=True` when creating the data loader: this will allocate the data in *page-locked memory* which guarantees a fixed physical memory location in the CPU RAM, and therefore allows direct memory access (DMA) transfers to the GPU, eliminating an extra copy operation that would otherwise be needed. While this could use more CPU RAM since the memory cannot be swapped out to disk, it typically results in significantly faster data transfers and thus faster training.
- The current training loop waits until a batch has been fully processed before it loads the next batch. You can often speed up training by prefetching the next batches on the CPU, while the GPU is still working on the current batch. For this, set the data loader's `num_workers` argument to the number of processes you want to use for data loading and preprocessing. The optimal number

depends on your platform, hardware, and workload, so you should experiment with different values. Note that the overhead of spawning and synchronizing workers can often slow down training rather than speed it up (especially on Windows). In this case, you can try setting `persistent_workers=True` to reuse the same workers across epochs.

OK, time to step back a bit: you know the PyTorch fundamentals (tensors & autograd), you can build neural nets using PyTorch's high-level API, and train them using mini-batch gradient descent, with the help of an optimizer, a criterion, and a data loader. The next step is to learn how to evaluate your model.

## Model Evaluation

Let's write a function to evaluate the model. It takes the model and a `DataLoader` for the dataset that we want to evaluate the model on, as well as a function to compute the metric for a given batch, and lastly a function to aggregate the batch metrics (by default, it just computes the mean):

```
def evaluate(model, data_loader, metric_fn, aggregate_fn=torch.mean):
    model.eval()
    metrics = []
    with torch.no_grad():
        for X_batch, y_batch in data_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            metric = metric_fn(y_pred, y_batch)
            metrics.append(metric)
    return aggregate_fn(torch.stack(metrics))
```

Now let's build a `TensorDataset` and a `DataLoader` for our validation set, and pass it to our `evaluate()` function to compute the validation MSE:

```
>>> valid_dataset = TensorDataset(X_valid, y_valid)
>>> valid_loader = DataLoader(valid_dataset, batch_size=32)
>>> valid_mse = evaluate(model, valid_loader, mse)
>>> valid_mse
tensor(0.4193, device='cuda:0')
```

It works fine. But now suppose we want to use the RMSE instead of the MSE. PyTorch does not have a built-in function for that, but it's easy enough to write:

```

>>> def rmse(y_pred, y_true):
...     return ((y_pred - y_true) ** 2).mean().sqrt()
...
>>> evaluate(model, valid_loader, rmse)
tensor(0.5732, device='cuda:0')

```

But wait a second! The RMSE should be equal to the square root of the MSE, however when we compute the square root of the MSE that we found earlier, we get a different result:

```

>>> valid_mse.sqrt()
tensor(0.6476, device='cuda:0')

```

The reason is that instead of calculating the RMSE over the whole validation set, we computed it over each batch and then computed the mean of all these batch RMSEs. That's not mathematically equivalent to computing the RMSE over the whole validation set. To solve this, we can use the MSE as our `metric_fn`, and use the `aggregate_fn` to compute the square root of the mean MSE:<sup>10</sup>

```

>>> evaluate(model, valid_loader, mse,
...           aggregate_fn=lambda metrics: torch.sqrt(torch.mean(metrics)))
...
tensor(0.6476, device='cuda:0')

```

That's much better!

Rather than implement metrics yourself, you may prefer to use the TorchMetrics library (made by the same team as PyTorch Lightning), which provides many well-tested *streaming metrics*. A streaming metric is an object that keeps track of a given metric, and can be updated one batch at a time. The TorchMetrics library is not preinstalled on Colab, so we have to run `%pip install torchmetrics`, then we can implement the `evaluate_tm()` function like this:

```

import torchmetrics

def evaluate_tm(model, data_loader, metric):
    model.eval()
    metric.reset() # reset the metric at the beginning
    with torch.no_grad():
        for X_batch, y_batch in data_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            metric.update(y_pred, y_batch) # update it at each iteration
    return metric.compute() # compute the final result at the end

```

Then we can create an RMSE streaming metric, move it to the GPU, and use it to evaluate the validation set:

```
>>> rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
>>> evaluate_tm(model, valid_loader, rmse)
tensor(0.6477, device='cuda:0')
```

Sure enough, we get the correct result! Now try updating the `train()` function to evaluate your model's performance during training, both on the training set (during each epoch) and on the validation set (at the end of each epoch). As always, if the performance on the training set is much better than on the validation set, your model is probably overfitting the training set, or there is a bug, such as a data mismatch between the training set and the validation set. This is easier to detect if you plot and analyze the learning curves, much like we did in [Chapter 4](#). For this you can use Matplotlib, or a visualization tool such as TensorBoard (see the notebook for an example).

Now you know how to build, train, and evaluate a regression MLP using PyTorch, and how to use the trained model to make predictions. Great! But so far we have only looked at simple sequential models, composed of a sequence of linear layers and ReLU activation functions. How would you build a more complex, nonsequential model? For this, we will need to build custom modules.

## Building Nonsequential Models Using Custom Modules

One example of a nonsequential neural network is a *Wide & Deep* neural network. This neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al.<sup>11</sup> It connects all or part of the inputs directly to the output layer, as shown in [Figure 10-1](#). This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path). The short path can also be used to provide manually engineered features to the neural network. In contrast, a regular MLP forces all the data to flow through the full stack of layers; thus, simple patterns in the data may end up being distorted by this sequence of transformations.

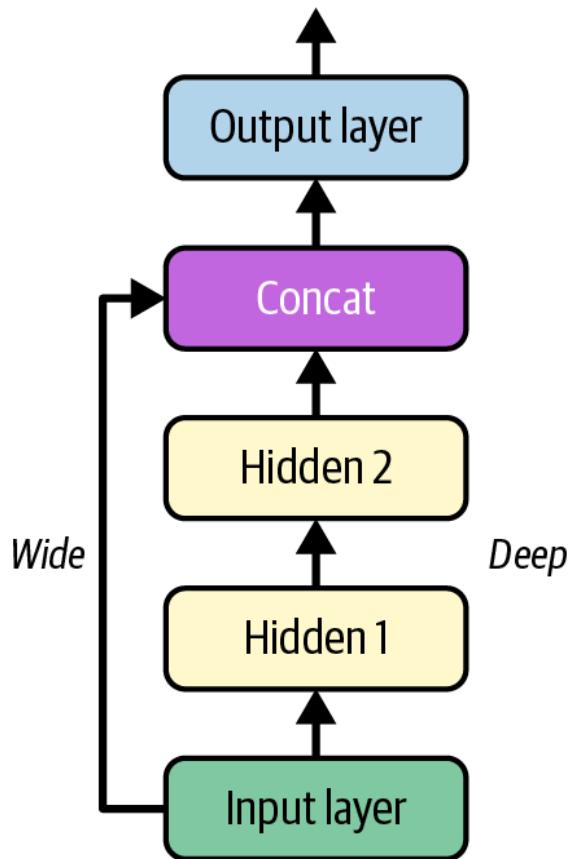


Figure 10-1. Wide & Deep neural network

Let's build such a neural network to tackle the California housing problem. Because it's nonsequential, we have to create a custom module. It's easier than it sounds: just create a class derived from `torch.nn.Module`, then create all the layers you need in the constructor (after calling the base class's `__init__()` method), and define how these layers should be used by the module in the `forward()` method:

```

class WideAndDeep(nn.Module):
    def __init__(self, n_features):
        super().__init__()
        self.deep_stack = nn.Sequential(
            nn.Linear(n_features, 50), nn.ReLU(),
            nn.Linear(50, 40), nn.ReLU(),
        )
        self.output_layer = nn.Linear(40 + n_features, 1)

    def forward(self, X):
        deep_output = self.deep_stack(X)
        wide_and_deep = torch.concat([X, deep_output], dim=1)
        return self.output_layer(wide_and_deep)

```

Notice that we can use any kind of module inside our custom module: in this example, we use a `nn.Sequential` module to build the “deep” part of our model (it's actually

not that deep, this is just a toy example). It's the same MLP as earlier, except we separated the output layer because we need to feed it the concatenation of the model's inputs and the deep part's outputs. For this same reason, the output layer now has  $40 + n\_features$  inputs instead of just 40.

In the `forward()` method, we just feed the input `X` to the deep stack, and we concatenate the input and the deep stack's output, and feed the result to the output layer.

## NOTE

Modules have a `children()` method that returns an iterator over the module's submodules (non-recursively). There's also a `named_children()` method. If your model has a variable number of submodules, you should store them in a `nn.ModuleList` or a `nn.ModuleDict`, which are returned by the `children()` and `named_children()` methods (as opposed to regular Python lists and dicts). Similarly, if your model has a variable number of parameters, you should store them in a `nn.ParameterList` or a `nn.ParameterDict` to ensure they are returned by the `parameters()` and `named_parameters()` methods.

Now we can create an instance of our custom module, move it to the GPU, train it, evaluate it, and use it exactly like our previous models:

```
torch.manual_seed(42)
model = WideAndDeep(n_features).to(device)
learning_rate = 0.002 # the model changed, so did the optimal learning rate
[...] # train, evaluate, and use the model, exactly like earlier
```

But what if you want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path, as illustrated in [Figure 10-2](#)? In this case, one approach is to split the inputs inside the `forward()` method, for example:

```
class WideAndDeepV2(nn.Module):
    [...] # same constructor as earlier, except with adjusted input sizes

    def forward(self, X):
        X_wide = X[:, :5]
        X_deep = X[:, 2:]
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        return self.output_layer(wide_and_deep)
```

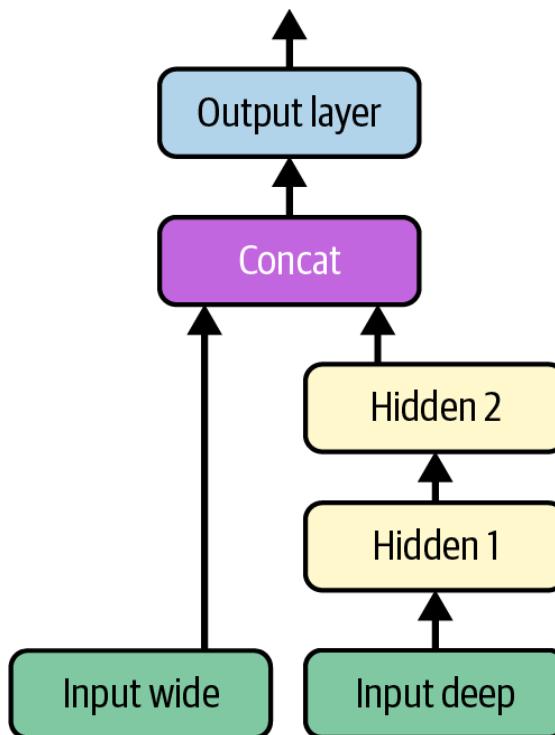
This works fine, however in many cases it's preferable to just let the model take two separate tensors as input. Let's see why and how.

## Building Models with Multiple Inputs

Some models require multiple inputs that cannot easily be combined into a single tensor. For example, the inputs may have a different number of dimensions (e.g., when you want to feed both images and text to the neural network). To make our wide & deep model take two separate inputs, as shown in [Figure 10-2](#), we must start by changing the model's `forward()` method:

```
class WideAndDeepV3(nn.Module):
    [...] # same as WideAndDeepV2

    def forward(self, X_wide, X_deep):
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        return self.output_layer(wide_and_deep)
```



*Figure 10-2. Handling multiple inputs*

Next, we need to create datasets that return the wide and deep inputs separately:

```
train_data_wd = TensorDataset(X_train[:, :5], X_train[:, 2:], y_train)
train_loader_wd = DataLoader(train_data_wd, batch_size=32, shuffle=True)
[...] # same for the validation set and test set
```

Since the data loaders now return three tensors instead of two at each iteration, we need to update the main loop in the evaluation and training functions:

```

for X_batch_wide, X_batch_deep, y_batch in data_loader:
    X_batch_wide = X_batch_wide.to(device)
    X_batch_deep = X_batch_deep.to(device)
    y_batch = y_batch.to(device)
    y_pred = model(X_batch_wide, X_batch_deep)
    [...] # the rest of the function is unchanged

```

Alternatively, since the order of the inputs matches the order of the `forward()` method's arguments, we can use Python's \* operator to unpack all the inputs returned by the `data_loader`, and pass them to the model. The advantage of this implementation is that it will work with models that take any number of inputs, not just two, as long as the order is correct:

```

for *X_batch_inputs, y_batch in data_loader:
    X_batch_inputs = [X.to(device) for X in X_batch_inputs]
    y_batch = y_batch.to(device)
    y_pred = model(*X_batch_inputs)
    [...]

```

When your model has many inputs, it's easy to make a mistake and mix up the order of the inputs, which can lead to hard-to-debug issues. To avoid this, it can be a good idea to name each input. For this, you can define a custom dataset that returns a dictionary from input names to input values, like this:

```

class WideAndDeepDataset(torch.utils.data.Dataset):
    def __init__(self, X_wide, X_deep, y):
        self.X_wide = X_wide
        self.X_deep = X_deep
        self.y = y

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        input_dict = {"X_wide": self.X_wide[idx], "X_deep": self.X_deep[idx]}
        return input_dict, self.y[idx]

```

Then create the datasets and data loaders:

```

train_data_named = WideAndDeepDataset(
    X_wide=X_train[:, :5], X_deep=X_train[:, 2:], y=y_train)
train_loader_named = DataLoader(train_data_named, batch_size=32, shuffle=True)
[...] # same for the validation set and test set

```

Once again, we also need to update the main loop in the evaluation and training functions:

```
for inputs, y_batch in data_loader:  
    inputs = {name: X.to(device) for name, X in inputs.items()}  
    y_batch = y_batch.to(device)  
    y_pred = model(X_wide=inputs["X_wide"], X_deep=inputs["X_deep"])  
    [...] # the rest of the function is unchanged
```

Alternatively, since all the input names match the `forward()` method's argument names, we can use Python's `**` operator to unpack all the tensors in the `inputs` dictionary and pass them as named arguments to the model: `y_pred = model(**inputs)`.

Now that you know how to build sequential and nonsequential models with one or more inputs, let's look at models with multiple outputs.

## Building Models with Multiple Outputs

There are many use cases where you may need a neural net with multiple outputs:

- Firstly, the task may demand it. For instance, you may want to locate and classify the main object in a picture. This is both a regression task and a classification task.
- Similarly, you may have multiple independent tasks based on the same data. Sure, you could train one neural network per task, but in many cases you will get better results on all tasks by training a single neural network with one output per task. This is because the neural network can learn features in the data that are useful across tasks. For example, you could perform *multitask classification* on pictures of faces, using one output to classify the person's facial expression (smiling, surprised, etc.) and another output to identify whether they are wearing glasses or not.
- Another use case is regularization (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize). For example, you may want to add an auxiliary output in a neural network architecture (see [Figure 10-3](#)) to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.

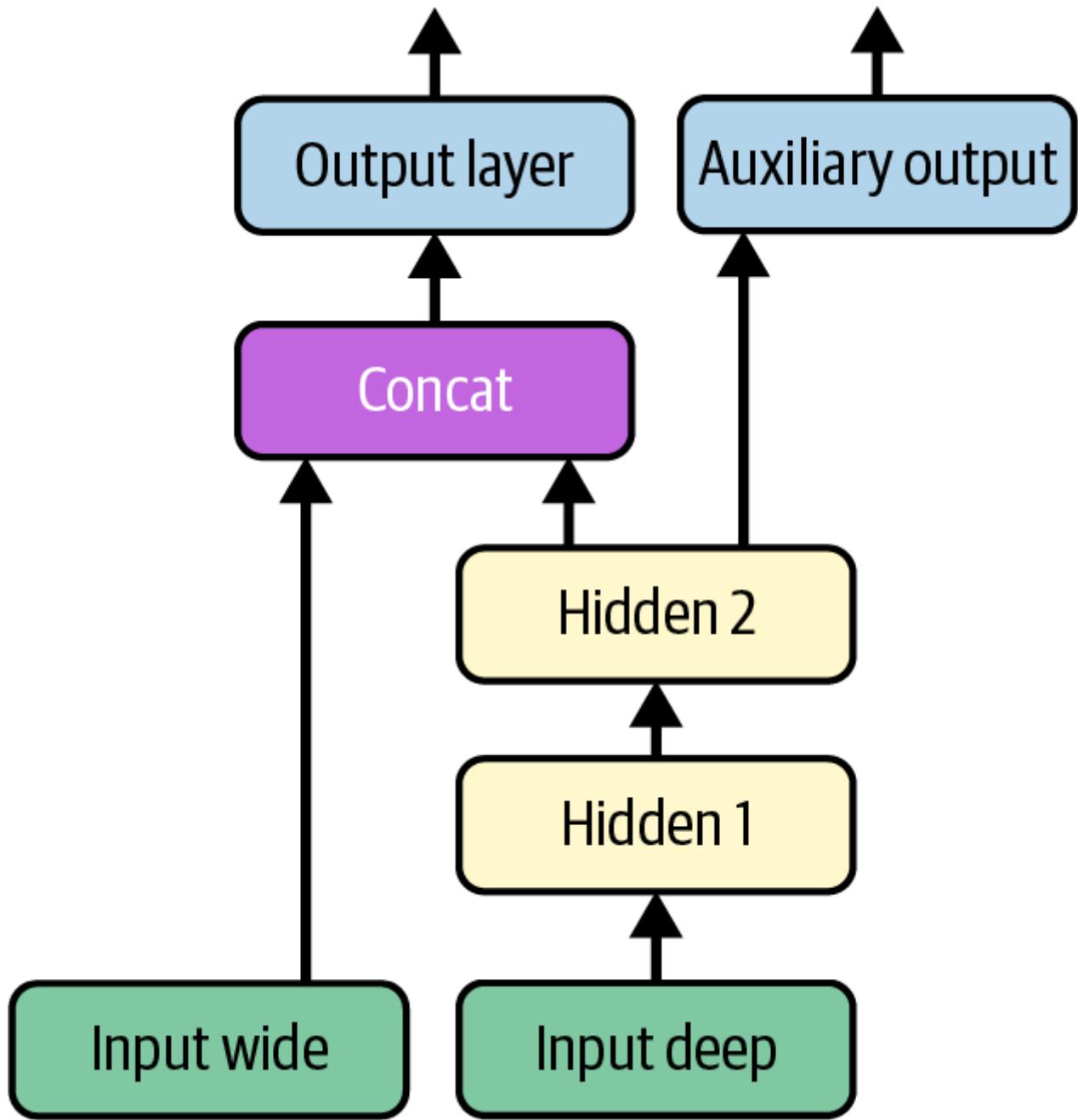


Figure 10-3. Handling multiple outputs, in this example to add an auxiliary output for regularization

Let's add an auxiliary output to our wide & deep model to ensure the deep part can make good predictions on its own. Since the deep stack's output dimension is 40, and the targets have a single dimension, we must add a `nn.Linear` layer for the auxiliary output, to go from 40 dimensions down to 1. We also need to make the `forward()` method compute the auxiliary output, and return both the main output and the auxiliary output:

```

class WideAndDeepV4(nn.Module):
    def __init__(self, n_features):
        [...] # same as earlier
        self.aux_output_layer = nn.Linear(40, 1)
  
```

```

def forward(self, X_wide, X_deep):
    deep_output = self.deep_stack(X_deep)
    wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
    main_output = self.output_layer(wide_and_deep)
    aux_output = self.aux_output_layer(deep_output)
    return main_output, aux_output

```

Next, we need to update the main loop in the training function:

```

for inputs, y_batch in train_loader:
    y_pred, y_pred_aux = model(**inputs)
    main_loss = criterion(y_pred, y_batch)
    aux_loss = criterion(y_pred_aux, y_batch)
    loss = 0.8 * main_loss + 0.2 * aux_loss
    [...] # the rest is unchanged

```

Notice that the model now returns both the main predictions `y_pred` and the auxiliary predictions `y_pred_aux`. In this example, we can use the same targets and the same loss function to compute the main output's loss and the auxiliary output's loss. In other cases, you may have different targets and loss functions for each output, in which case you would need to create a custom dataset to return all the necessary targets. Once we have a loss for each output, we must combine them into a single loss that will be minimized by gradient descent. In general, this final loss is just a weighted sum of all the output losses. In this example, we use a higher weight for the main loss (0.8), because that's what we care about the most, and a lower weight for the auxiliary loss (0.2). This ratio is a regularization hyperparameter that you can tune.

We also need to update the main loop in the evaluation function. However, in this case we can just ignore the auxiliary output, since we only really care about the main output—the auxiliary output is just there for regularization during training.

```

for inputs, y_batch in data_loader:
    y_pred, _ = model(**inputs)
    metric.update(y_pred, y_batch)
    [...] # the rest is unchanged

```

Voilà! You can now build and train all sorts of neural net architectures, combining pre-defined modules and custom modules in any way you please, and with any number of inputs and outputs. The flexibility of neural networks is one of their main qualities. But so far we have only tackled a regression task, so let's now turn to classification.

# Building an Image Classifier with PyTorch

As in [Chapter 9](#), we will tackle the Fashion MNIST dataset, so the first thing we need to do is to download the dataset. We could use the `fetch_openml()` function like we did in [Chapter 9](#), but we will show another method instead, using the TorchVision library.

## Using TorchVision to Load the Dataset

The TorchVision library is an important part of the PyTorch ecosystem: it provides many tools for computer vision, including utility functions to download common datasets, such as MNIST or Fashion MNIST, as well as pretrained models for various computer vision tasks (see [\[Link to Come\]](#)), functions to transform images (e.g., crop, rotate, resize, etc.), and more. It is preinstalled on Colab, so let's go ahead and use it to load Fashion MNIST. It is already split into a training set (60,000 images) and a test set (10,000 images), but we'll hold out the last 5,000 images from the training set for validation, using PyTorch's `random_split()` function:

```
import torchvision
import torchvision.transforms.v2 as T

toTensor = T.Compose([T.ToImage(), T.ToDtype(torch.float32, scale=True)])

train_and_valid_data = torchvision.datasets.FashionMNIST(
    root="datasets", train=True, download=True, transform=toTensor)
test_data = torchvision.datasets.FashionMNIST(
    root="datasets", train=False, download=True, transform=toTensor)

torch.manual_seed(42)
train_data, valid_data = torch.utils.data.random_split(
    train_and_valid_data, [55_000, 5_000])
```

After the imports and before loading the datasets, we create a `toTensor` object. What's that about? Well, by default, the `FashionMNIST` class loads images as PIL (Python Image Library) images, with integer pixel values ranging from 0 to 255. But we need PyTorch float tensors instead, with scaled pixel values. Luckily, TorchVision datasets accept a `transform` argument which lets you pass a preprocessing function that will get executed on the fly whenever the data is accessed (there's also a `target_transform` argument if you need to preprocess the targets). TorchVision provides many transform objects which you can use for this (most of these transforms are PyTorch modules).

In this code, we create a `Compose` transform to chain two transforms: a `ToImage` transform followed by a `ToDtype` transform. `ToImage` converts various formats—

including PIL images, NumPy arrays, and tensors—to TorchVision’s `Image` class, which is a subclass of `Tensor`. The `ToDtype` transform converts the data type, in this case to 32-bit floats. We also set its `scale` argument to `True` to ensure the values get scaled between 0.0 and 1.0.<sup>12</sup>

## NOTE

The version 1 of TorchVision’s transforms API is still available for backward compatibility and can be imported using `import torchvision.transforms`, but you should use version 2 (v2) instead, since it’s faster and has more features.

Next, we load the dataset: first the training and validation data, then the test data. The `root` argument is the path to the directory where TorchVision will create a subdirectory for the Fashion MNIST dataset. The `train` argument indicates whether you want to load the training set (`True` by default) or the test set. The `download` argument indicates whether to download the dataset if it cannot be found locally (`False` by default). And we also set `transform=toTensor` to use our custom preprocessing pipeline.

As usual, we must create data loaders:

```
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
valid_loader = DataLoader(valid_data, batch_size=32)
test_loader = DataLoader(test_data, batch_size=32)
```

Now let’s look at the first image in the training set:

```
>>> X_sample, y_sample = train_data[0]
>>> X_sample.shape
torch.Size([1, 28, 28])
>>> X_sample.dtype
torch.float32
```

In [Chapter 9](#), each image was represented by a 1D array containing 784 pixel intensities, but now each image tensor has 3 dimensions, and its shape is: [1, 28, 28]. The first dimension is the *channel* dimension. For grayscale images, there is a single channel (color images usually have 3 channels, as we will see in [\[Link to Come\]](#)). The other two dimensions are the height and width dimensions. For example, `X_sample[0, 2, 4]` represents the pixel located in channel 0, row 2, column 4. In Fashion MNIST, a larger value means a darker pixel.

## WARNING

PyTorch expects the channel dimension to be first, while many other libraries such as Matplotlib, PIL, TensorFlow, OpenCV, or Scikit-Image expect it to be last. Always make sure to move the channels dimension to the right place depending on the library you are using. ToImage already took care of moving the channel dimension to the first position, otherwise we could have used the `torch.permute()` function.

As for the targets, they are integers from 0 to 9, and we can interpret them using the same `class_names` array as in [Chapter 9](#). In fact, many datasets—including `FashionMNIST`—have a `classes` attribute containing the list of class names. For example, here's how we can tell that the sample image represents an ankle boot:

```
>>> train_and_valid_data.classes[y_sample]  
'Ankle boot'
```

## Building the Classifier

Let's build a custom module for a classification MLP with two hidden layers:

```
class ImageClassifier(nn.Module):  
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_classes):  
        super().__init__()  
        self.mlp = nn.Sequential(  
            nn.Flatten(),  
            nn.Linear(n_inputs, n_hidden1),  
            nn.ReLU(),  
            nn.Linear(n_hidden1, n_hidden2),  
            nn.ReLU(),  
            nn.Linear(n_hidden2, n_classes)  
        )  
  
    def forward(self, X):  
        return self.mlp(X)  
  
torch.manual_seed(42)  
model = ImageClassifier(n_inputs=28 * 28, n_hidden1=300, n_hidden2=100,  
                      n_classes=10)  
xentropy = nn.CrossEntropyLoss()
```

There are a few things to note in this code:

- First, the model is composed of a single sequence of layers, which is why we used the `nn.Sequential` module. We did not have to create a custom module, we could have written `model = nn.Sequential(...)` instead, but it's

generally preferable to wrap your models in custom modules, as it makes your code easier to deploy and reuse, and it's also easier to tune the hyperparameters.

- The model starts with a `nn.Flatten` layer: this layer does not have any parameters, it just reshapes each input sample to a single dimension, which is needed for the `nn.Linear` layers. For example, a batch of 32 Fashion MNIST images has a shape of [32, 1, 28, 28], but after going through the `nn.Flatten` layer, it ends up with a shape of [32, 784] (since  $28 \times 28 = 784$ ).
- The first hidden layer must have the correct number of inputs ( $28 \times 28 = 784$ ), and the output layer must have the correct number of outputs (10, one per class).
- We use a ReLU activation function after each hidden layer, and no activation function at all after the output layer.
- Since this is a multiclass classification task, we use the `nn.CrossEntropyLoss`. It accepts either class indices as targets (as in this example), or class probabilities (such as one-hot vectors).

### TIP

Shape errors are quite common, especially when getting started, so you should familiarize yourself with the error messages: try removing the `nn.Flatten` module, or try messing with the shape of the inputs and/or labels, and see the errors you get.

But wait! Didn't we say in [Chapter 9](#) that we should use the softmax activation function on the output layer, for multiclass classification tasks? Well it turns out that PyTorch's `nn.CrossEntropyLoss` computes the cross-entropy loss directly from the logits (i.e., the class scores, introduced in [Chapter 4](#)), rather than from the class probabilities. This bypasses some costly computations during training (e.g., logarithms and exponentials that cancel out), saving both compute and RAM. It's also more numerically stable. However, the downside is that the model must output logits, which means that we will have to call the softmax function manually on the logits whenever we want class probabilities, as we will see shortly.

## OTHER CLASSIFICATION LOSSES

For multiclass classification, another option is to add the `nn.LogSoftmax` activation function to the output layer, and then use the `nn.NLLLoss` (negative log-likelihood loss). The model then outputs log probabilities (rather than logits), and the loss computes the cross-entropy based on these log probabilities. Whenever you need actual estimated probabilities, just pass the log probabilities through the exponential function. This approach is a bit slower than using `nn.CrossEntropyLoss`, so it's not used as often, but it can sometimes be useful if you want your model to output log probabilities, or when you wish to tweak the probability distribution before computing the loss.

For binary classification tasks, you must use a single output neuron in the output layer and use the `nn.BCEWithLogitsLoss` (BCE stands for binary cross-entropy). The model outputs logits, so you must apply the sigmoid function to get estimated probabilities (for the positive class). Alternatively, you can add the `nn.Sigmoid` activation function to the output layer, and use the `nn.BCELoss`: the model will then output estimated probabilities directly (but it's a bit slower and less numerically stable).

For multilabel binary classification, the only difference is that you must have one neuron per label in the output layer.

Now we can train the model as usual (e.g., using the `train()` function with an SGD optimizer). To evaluate the model, we can use the `Accuracy` streaming metric from the `torchmetrics` library, and move it to the GPU:

```
accuracy = torchmetrics.Accuracy(task="multiclass", num_classes=10).to(device)
```

### WARNING

Training the model will take a few minutes with a GPU (or much longer without one). Handling images requires significantly more compute and memory than handling low-dimensional data.

The model reaches around 92.8% accuracy on the training set, and 87.6% accuracy on the validation set (the results might differ a bit depending on the hardware accelerator you use). This means there's a little bit of overfitting going on, so you may want to reduce the number of neurons or add some regularization (see [Chapter 11](#)).

Now that the model is trained, we can use it to make predictions on new images. As an example, let's make predictions for the first batch in the validation set, and look at the results for the first 3 images:

```
>>> model.eval()
>>> X_new, y_new = next(iter(valid_loader))
>>> X_new = X_new[:3].to(device)
>>> with torch.no_grad():
...     y_pred_logits = model(X_new)
...
>>> y_pred = y_pred_logits.argmax(dim=1) # index of the largest logit
>>> y_pred
tensor([7, 4, 2], device='cuda:0')
>>> [train_and_valid_data.classes[index] for index in y_pred]
['Sneaker', 'Coat', 'Pullover']
```

For each image, the predicted class is the one with the highest logit. In this example, all three predictions are correct!

But what if we want the model's estimated probabilities? For this, we need to compute the softmax of the logits manually, since the model does not include the softmax activation function on the output layer, as we discussed earlier. We could create a `nn.Softmax` module and pass it the logits, but we can also just call the `softmax()` function, which is just one of many functions you will find in the `torch.nn.functional` module (by convention, this module is usually imported as F). It doesn't make much difference, it just avoids creating a module instance that we don't need:

```
>>> import torch.nn.functional as F
>>> y_proba = F.softmax(y_pred_logits, dim=1)
>>> y_proba.round(decimals=3)
tensor([[0.000, 0.000, 0.000, 0.000, 0.000, 0.002, 0.000, 0.957, 0.000, 0.040],
        [0.000, 0.000, 0.003, 0.000, 0.997, 0.000, 0.000, 0.000, 0.000, 0.000],
        [0.001, 0.000, 0.670, 0.001, 0.144, 0.000, 0.182, 0.000, 0.001, 0.000]], device='cuda:0')
```

Just like in [Chapter 9](#), the model is very confident about the first two predictions: 95.7% and 99.7% respectively.<sup>13</sup>

### TIP

If you wish to apply label smoothing during training, just set the `label_smoothing` hyperparameter of the `nn.CrossEntropyLoss` to the amount of smoothing you wish, between 0 and 1 (e.g., 0.05).

It can often be useful to get the model's top  $k$  predictions. For this, we can use the `torch.topk()` function, which returns a tuple containing both the top  $k$  values and their indices:

```
>>> y_top4_logits, y_top4_indices = torch.topk(y_pred_logits, k=4, dim=1)
>>> y_top4_probas = F.softmax(y_top4_logits, dim=1)
>>> y_top4_probas.round(decimals=3)
tensor([[0.9570, 0.0400, 0.0020, 0.0000],
        [0.9970, 0.0030, 0.0000, 0.0000],
        [0.6720, 0.1830, 0.1440, 0.0010]], device='cuda:0')
>>> y_top4_indices
tensor([[7, 9, 5, 8],
        [4, 2, 6, 3],
        [2, 6, 4, 0]], device='cuda:0')
```

For the first image, the model's best guess is class 7 (Sneaker) with 95.7% confidence, its second best guess is class 9 (Ankle boot) with 4% confidence, and so on.

### TIP

The Fashion MNIST dataset is balanced, meaning it has the same number of instances of each class. When dealing with an unbalanced dataset, you should generally give more weight to the rare classes and less weight to the frequent ones, or else your model will be biased towards the more frequent classes. You can do this by setting the `weight` argument of the `nn.CrossEntropyLoss`. For example, if there are three classes with 900, 700, and 400 instances respectively (i.e., 2000 instances in total), then the respective weights should be 2000/900, 2000/700, and 2000/400. It's preferable to normalize these weights to ensure they add up to 1, so in this example you would set `weight=torch.tensor([0.2205, 0.2835, 0.4961])`.

Your PyTorch superpowers are growing: you can now build, train, and evaluate both regression and classification neural nets. The next step is to learn how to fine-tune the model hyperparameters.

## Fine-Tuning Neural Network Hyperparameters with Optuna

We discussed how to manually pick reasonable values for your model's hyperparameters in [Chapter 9](#), but what if you want to go further and automatically search for good hyperparameter values? One option is to convert your PyTorch model to a Scikit-Learn estimator, either by writing your own custom estimator class or by using a wrapper library such as Skorch (<https://skorch.readthedocs.io/>), and then use

`GridSearchCV` or `RandomizedSearchCV` to fine-tune the hyperparameters, as you did in [Chapter 2](#). However, you will usually get better results by using a dedicated fine-tuning library such as Optuna (<https://optuna.org/>), Ray Tune (<https://docs.ray.io/>), or Hyperopt (<https://hyperopt.github.io/hyperopt/>). These libraries offer several powerful tuning strategies, and they’re highly customizable.

Let’s look at an example using Optuna. It is not preinstalled on Colab, so we need to install it using `%pip install optuna` (if you prefer to run the code locally, please follow the installation instructions at <https://hml.info/install-p>). Let’s tune the learning rate and the number of neurons in the hidden layers (for simplicity, we will use the same number of neurons in both hidden layers). First, we need to define a function that Optuna will call many times to perform hyperparameter tuning: this function must take a `Trial` object and use it to ask Optuna for hyperparameter values, and then use these hyperparameter values to build and train a model. Finally, the function must evaluate the model (typically on the validation set) and return the metric:

```
import optuna

def objective(trial):
    learning_rate = trial.suggest_float("learning_rate", 1e-5, 1e-1, log=True)
    n_hidden = trial.suggest_int("n_hidden", 20, 300)
    model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=n_hidden,
                           n_hidden2=n_hidden, n_classes=10).to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
    [...] # train the model, then evaluate it on the validation set
    return validation_accuracy
```

The `suggest_float()` and `suggest_int()` methods let us ask Optuna for a good hyperparameter value in a given range (Optuna also provides a `suggest_categorical()` method). For the `learning_rate` hyperparameter, we ask for a value between  $10^{-5}$  and  $10^{-1}$ , and since we don’t know what the optimal scale is, we add `log=True`: this will make Optuna sample values from a log distribution, which makes it explore all possible scales. If we used the default uniform distribution instead, Optuna would be very unlikely to explore tiny values.

To start hyperparameter tuning, we create a `Study` object and call its `optimize()` method, passing it the objective function we just defined, as well as the number of trials to run (i.e., the number of times Optuna should call the objective function). Since our objective function returns a score—higher is better—we set `direction="maximize"` when creating the study (by default, Optuna tries to *minimize* the objective). To ensure

reproducibility, we also set PyTorch’s random seed, as well as the random seed used by Optuna’s sampler:

```
torch.manual_seed(42)
sampler = optuna.samplers.TPESampler(seed=42)
study = optuna.create_study(direction="maximize", sampler=sampler)
study.optimize(objective, n_trials=5)
```

By default, Optuna uses the *Tree-structured Parzen Estimator* (TPE) algorithm to optimize the hyperparameters: this is a sequential model-based optimization algorithm, meaning it learns from past results to better select promising hyperparameters. In other words, Optuna starts with random hyperparameter values, but it progressively focuses its search on the most promising regions of the hyperparameter space. This allows Optuna to find much better hyperparameters than random search in the same amount of time.

### TIP

You can add more hyperparameters to the search space, such as the batch size, the type of optimizer, the number of hidden layers, or the type of activation function, but remember that the search space will grow exponentially as you add more hyperparameters, so make sure it’s worth the extra search time and compute.

Once Optuna is done, you can look at the best hyperparameters it found, as well as the corresponding validation accuracy:

```
>>> study.best_params
{'learning_rate': 0.008471801418819975, 'n_hidden': 188}
>>> study.best_value
0.8547999858856201
```

This is worse than the performance we got earlier, but that’s because we set `n_trials=5`, which is way too small. If you bump this value up to 50 or more, you will get much better results, but of course it will take hours to run. You can also just run `optimize()` repeatedly and stop once you are happy with the performance.

Optuna can also run trials in parallel across multiple machines, which can offer a near linear speed boost. For this, you will need to setup a SQL database (e.g., SQLite or PostgreSQL) and set the `storage` parameter of the `create_study()` function to point to that database. You also need to set the study’s name via the `study_name` parameter,

and set `load_if_exists=True`. After that, you can copy your hyperparameter tuning script to multiple machines, and run it on each one (if you are using random seeds, make sure they are different on each machine). The scripts will work in parallel, reading and writing the trial results to the database. This has the additional benefit of keeping a full log of all your experiment results.

You may have noticed that we assumed that the `objective()` function had direct access to the training set and validation, presumably via global variables. In general, it's much cleaner to pass them as extra arguments to the `objective()` function, for example like this:

```
def objective(trial, train_loader, valid_loader):
    [...] # the rest of the function remains the same as above

objective_with_data = lambda trial: objective(
    trial, train_loader=train_loader, valid_loader=valid_loader)
study.optimize(objective_with_data, n_trials=5)
```

To set the extra arguments (the dataset loaders in this case), we just create a lambda function when needed and pass it to the `optimize()` method. Alternatively, you can use the `functools.partial()` function which creates a thin wrapper function around the given callable to provide default values for any number of arguments:

```
from functools import partial

objective_with_data = partial(objective, train_loader=train_loader,
                             valid_loader=valid_loader)
```

It's often possible to quickly tell that a trial is absolutely terrible: for example, when the loss shoots up during the first epoch, or when the model barely improves during the first few epochs. In such a case, it's a good idea to interrupt training early, to avoid wasting time and compute. You can simply return the model's current validation accuracy, and hope that Optuna will learn to avoid this region of hyperparameter space. Alternatively, you can interrupt training by raising the `optuna.TrialPruned` exception: this tells Optuna to ignore this trial altogether. In many cases, this leads to a more efficient search because it avoids polluting Optuna's search algorithm with many noisy model evaluations.

Optuna comes with several `Pruner` classes that can detect and prune bad trials. For example, the `MedianPruner` will prune trials whose performance is below the median performance, at regular intervals during training. It starts pruning after a given number

of trials have completed, controlled by `n_startup_trials` (5 by default). For each trial after that, it lets training start for a few epochs, controlled by `n_warmup_steps` (0 by default), then every few epochs (controlled by `interval_steps`), it ensures that the model's performance is better than the median performance at the same epoch in past trials. To use this pruner, create an instance and pass it to the `create_study()` method:

```
pruner = optuna.pruners.MedianPruner(n_startup_trials=5, n_warmup_steps=0,
                                         interval_steps=1)
study = optuna.create_study(direction="maximize", sampler=sampler,
                             pruner=pruner)
```

Then in the `objective()` function, add the following code so it runs after each epoch:

```
for epoch in range(n_epochs):
    [...] # train the model for one epoch
    validation_accuracy = [...] # evaluate the model's validation accuracy
    trial.report(validation_accuracy, epoch)
    if trial.should_prune():
        raise optuna.TrialPruned()
```

The `report()` method informs Optuna of the current validation accuracy and epoch, so it can determine whether or not the trial should be pruned. If `trial.should_prune()` returns `True`, we raise a `TrialPruned` exception.

### TIP

Optuna has many other features well worth exploring, such as visualization tools, persistence tools for trial results and other artifacts, a dashboard for human-in-the-loop optimization, and many other algorithms for hyperparameter search and trial pruning.

Once you are happy with the hyperparameters, you can train the model on the full training set (i.e., the training set + the validation set), then evaluate it on the test set. Hopefully, it will perform great! If it does, you will want to save the model, then load it and use it in production: that's the final topic of this chapter.

## Saving and Loading PyTorch Models

The simplest way to save a PyTorch model is to use the `torch.save()` method, passing it the model and the file path. This uses Python's pickle functionality to serialize the

model object, then compresses the result (zip) and saves it to disk. The convention is to use the `.pt` or `.pth` extension for PyTorch files:

```
torch.save(model, "my_fashion_mnist.pt")
```

Simple! Now you can load the model (e.g., in your production code) just as easily:

```
loaded_model = torch.load("my_fashion_mnist.pt", weights_only=False)
```

### WARNING

If your model uses any custom functions or classes (e.g., `ImageClassifier`), then `torch.save()` only saves references to them, not the code itself. Therefore you must ensure that any custom code is loaded in the Python environment before calling `torch.load()`. Also make sure to use the same version of the code to avoid any mismatch issues.

Setting `weights_only=False` ensures that the whole model object is loaded rather than just the model parameters. Then you can use the loaded model for inference. Don't forget to switch to evaluation mode first using the `eval()` method:

```
loaded_model.eval()  
y_pred_logits = loaded_model(X_new)
```

This is nice and easy, but unfortunately this approach has some very serious drawbacks:

- Firstly, pickle's serialization format is notoriously insecure: while `torch.save()` doesn't save custom code, the pickle format supports it, so a hacker could inject malicious code in a saved PyTorch model: this code would be run automatically by the pickle module when the model is loaded. So always make sure you fully trust the model's source before you load it this way.
- Secondly, pickle is somewhat brittle: it can vary depending on the Python version (e.g., there were big changes between Python 3.7 and 3.8) and it saves specific file paths to locate code, which can break if the loading environment has a different folder structure.

To avoid these issues, it is recommended to save and load the model weights only, rather than the full model object:

```
torch.save(model.state_dict(), "my_fashion_mnist_weights.pt")
```

The state dictionary returned by the `state_dict()` method is just a Python `OrderedDict` containing an entry for each parameter returned by the `named_parameters()` method. It also contains buffers, if the model has any: a buffer is just a regular tensor that was registered with the model (or any of its submodules) using the `register_buffer()` method. Buffers hold extra data that needs to be stored along with the model, but that is not a model parameter. We will see an example in [Chapter 11](#) with the batch-norm layer.

To load these weights, we must first create a model with the exact same structure, then load the weights using `torch.load()` with `weights_only=True`, and finally call the model's `load_state_dict()` method with the loaded weights:

```
new_model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=300, n_hidden2=100,
                            n_classes=10)
loaded_weights = torch.load("my_fashion_mnist_weights.pt", weights_only=True)
new_model.load_state_dict(loaded_weights)
new_model.eval()
```

The saved model contains only data, and the `load()` function makes sure of that, so this is safe, and also much less likely to break between Python version or to cause any deployment issue. However, it only works if you are able to create the exact same model architecture before loading the state dictionary. For this, you need to know the number of layers, the number of neurons per layer, and so on. It's a good idea to save this information along with the state dictionary:

```
model_data = {
    "model_state_dict": model.state_dict(),
    "model_hyperparameters": {"n_inputs": 1 * 28 * 28, "n_hidden1": 300, [...]}
}
torch.save(model_data, "my_fashion_mnist_model.pt")
```

You can then load this dictionary, construct the model based on the saved hyperparameters, and load the state dictionary into this model:

```
loaded_data = torch.load("my_fashion_mnist_model.pt", weights_only=True)
new_model = ImageClassifier(**loaded_data["model_hyperparameters"])
new_model.load_state_dict(loaded_data["model_state_dict"])
new_model.eval()
```

If you want to be able to continue training where it left off, you will also need to save the optimizer’s state dictionary, its hyperparameters, and any other training information you may need, such as the current epoch and the loss history.

### TIP

The `safetensors` library by Hugging Face is another popular way to save model weights safely.

There is yet another way to save and load your model: by first converting it to TorchScript. This also makes it possible to speed up your model’s inference.

## Compiling and Optimizing a PyTorch Model

PyTorch comes with a very nice feature: it can automatically convert your model’s code to *TorchScript*, which you can think of as a statically typed subset of Python. There are two main benefits:

- First, TorchScript code can be compiled and optimized to produce significantly faster models. For example, multiple operations can often be fused into a single more efficient operation. Operations on constants (e.g.,  $2 * 3$ ) can be replaced with their result (e.g., 6): this is called *constant folding*. Unused code can be pruned, and so on.
- Secondly, TorchScript can be serialized, saved to disk, and then loaded and executed in Python or in a C++ environment using the LibTorch library: this makes it possible to run PyTorch models on a wide range of devices, including embedded devices.

There are two ways to convert a PyTorch model to TorchScript. The first way is called *tracing*. PyTorch just runs your model with some sample data and logs every operation that takes place, then converts this log to TorchScript. This is done using the `torch.jit.trace()` function:

```
torchscript_model = torch.jit.trace(model, X_new)
```

This generally works well with static models whose `forward()` method doesn’t use conditionals or loops. However, if you try to trace a model that includes an `if` or `match` statement, then only the branch that is actually executed will be captured by

TorchScript, which is generally not what you want. Similarly, if you use tracing with a model that contains a loop, then the TorchScript code will contain one copy of the operations within that loop for each iteration that was actually executed. Again, not what you generally want.

For such dynamic models, you will probably want to try another approach named *scripting*. In this case, PyTorch actually parses your Python code directly and converts it to TorchScript. This method supports `if` statements and `while` loops properly, as long as the conditions are tensors. It also supports `for` loops when iterating over tensors. However, it only works on a subset of Python. For example, you cannot use global variables, or Python generators (`yield`), or complex list comprehensions, or variable length function arguments (`*args` or `**kwargs`), or `match` statements, moreover types must be fixed (a function cannot return an integer in some cases and a float in others), and you can only call other functions if they also respect these rules, so no standard library, no third-party libraries, etc. (see the documentation for the full list of constraints). This sounds daunting, but for most real-world models, these rules are actually not too hard to respect, and you can save your model like this:

```
torchscript_model = torch.jit.script(model)
```

Regardless of whether you use tracing or scripting to produce your TorchScript model, you can then further optimize it:

```
optimized_model = torch.jit.optimize_for_inference(torchscript_model)
```

TorchScript models can only be used for inference, not for training, since the TorchScript environment doesn't support gradient tracking or parameter updates.

Finally, you can save a TorchScript model using its `save()` method:

```
torchscript_model.save('my_fashion_mnist_torchscript.pt')
```

And then load it using the `torch.jit.load()` function:

```
loaded_torchscript_model = torch.jit.load("my_fashion_mnist_torchscript.pt")
```

One important caveat: TorchScript is no longer under active development—bugs are fixed but no new features are added. It still works fine and it remains one of the best ways to run your PyTorch models in a C++ environment,<sup>14</sup> but since the release of

PyTorch 2.0 in March 2023, the PyTorch team has been focusing its efforts on a new set of compilation tools centered around the `torch.compile()` function, which you can use very easily:

```
compiled_model = torch.compile(model)
```

The resulting model can now be used normally, and it will automatically be compiled and optimized when you use it. This is called Just-In-Time (JIT) compilation, as opposed to Ahead-Of-Time (AOT) compilation. Under the hood, `torch.compile()` relies on *TorchDynamo* (or *Dynamo* for short) which hooks directly into Python bytecode to capture the model's computation graph at inference time. Having access to the bytecode allows Dynamo to efficiently and reliably capture the computation graph, properly handling conditionals and loops, while also benefiting from dynamic information that can be used to better optimize the model. The actual compilation and optimization is performed by default by a backend component named *TorchInductor*, which in turn relies on the Triton language to generate highly efficient GPU code (Nvidia only), or on the OpenMP API for CPU optimization. PyTorch 2.x offers a few other optimization backends, including the XLA backend for Google's TPU devices: just set `device="xla"` when calling `torch.compile()`.

With that you now have all the tools you need to start building and training complex and efficient neural networks. I hope you enjoyed this introduction to PyTorch! We covered a lot, but the adventure is only beginning: in the next chapter we will discuss techniques to train very deep nets. After that, we will dive into other popular neural network architectures: convolutional neural networks for image processing, recurrent neural networks for sequential data, transformers for text (and much more), autoencoders for representation learning, and generative adversarial networks and diffusion models to generate data.<sup>15</sup> Then we will visit reinforcement learning to train autonomous agents, and finally we will learn more about deploying and optimizing your PyTorch models. Let's go!

## Exercises

1. PyTorch is similar to NumPy in many ways, but it offers some extra features. Can you name the most important ones?
2. What is the difference between `torch.exp()` and `torch.exp_()`, or between `torch.relu()` and `torch.relu_()`?

3. What are two ways to create a new tensor on the GPU?
4. What are three ways to perform tensor computations without using autograd?
5. Will the following code cause a `RuntimeError`? What if you replace the second line with `z = t.cos_().exp()`? And what if you replace it with `z = t.exp_().cos_()`?

```
t = torch.tensor(2.0, requires_grad=True)
z = t.cos_().exp_()
z.backward()
```

How about the following code, will it cause an error? And what if you replace the third line with `w = v.cos_() * v.sin_()`? Will `w` have the same value in both cases?

```
u = torch.tensor(2.0, requires_grad=True)
v = u + 1
w = v.cos_() * v.sin_()
w.backward()
```

6. Suppose you create a `Linear(100, 200)` module. How many neurons does it have? What is the shape of its `weight` and `bias` parameters? What input shape does it expect? What output shape does it produce?
7. What are the main steps of a PyTorch training loop?
8. Why is it recommended to create the optimizer *after* the model is moved to the GPU?
9. What `DataLoader` options should you generally set to speed up training when using a GPU?
10. What are the main classification losses provided by PyTorch, and when should you use each of them?
11. Why is it important to call `model.train()` before training and `model.eval()` before evaluation?
12. What is the difference between `torch.jit.trace()` and `torch.jit.script()`?

13. Use autograd to find the gradient vector of  $f(x, y) = \sin(x^2 y)$  at the point  $(x, y) = (1.2, 3.4)$ .
14. Create a custom `Dense` module that replicates the functionality of a `nn.Linear` module followed by a `nn.ReLU` module. Try implementing it first using the `nn.Linear` and `nn.ReLU` modules, and then reimplement it using `nn.Parameter` and the `relu()` function.
15. Build and train a classification MLP on the CoverType dataset:
  - a. Load the dataset using `sklearn.datasets.fetch_covtype()` and create a custom PyTorch `Dataset` for this data.
  - b. Create data loaders for training, validation, and testing.
  - c. Build a custom MLP module to tackle this classification task. You can optionally use the custom `Dense` module from the previous exercise.
  - d. Train this model on the GPU, and try to reach 93% accuracy on the test set. For this, you will likely have to perform hyperparameter search to find the right number of layers and neurons per layer, a good learning rate and batch size, and so on. You can optionally use Optuna for this.

Solutions to these exercises are available at the end of this chapter's notebook, at <https://homl.info/colab-p>.

---

- <sup>1</sup> To be fair, most of TensorFlow's usability issues were fixed in version 2, and Google also launched JAX, which is well designed and extremely fast, so PyTorch still has some healthy competition. The good news is that the APIs of all these libraries have converged quite a bit, so switching from one to the other is much easier than it used to be.
- <sup>2</sup> There are things called tensors in mathematics and physics, but ML tensors are simpler: they're really just multidimensional arrays for numerical computations, plus a few extra features.
- <sup>3</sup> CUDA is Nvidia's proprietary platform to run code on their CUDA-compatible GPUs, and cuDNN is a library built on CUDA to accelerate various deep neural network architectures.
- <sup>4</sup> The `%timeit` magic command only works in Jupyter notebooks and Colab, as well as in the iPython shell; in a regular Python shell or program, you can use the `timeit.timeit()` function instead.
- <sup>5</sup> For example, since the derivative of  $\exp(x)$  is equal to  $\exp(x)$ , it makes a lot of sense to store the output of this operation in the computation graph during the forward pass, then use this output during the backward pass to get the gradients: no need to store additional data, and no need to recompute  $\exp(x)$ .

- 6 For example, the derivative of  $\text{abs}(x)$  is  $-1$  when  $x < 0$  and  $+1$  when  $x > 0$ . If this operation saved its output in the computation graph, the backward pass would be unable to know whether  $x$  was positive or negative (since  $\text{abs}(x)$  is always positive), so it wouldn't be able to compute the gradients. This is why this operation must save its input instead.
- 7 For example, the derivative of  $\text{floor}(x)$  is always zero (at least for non-integer inputs), so the `floor()` operation just saves the shape of the inputs during the forward pass, then during the backward pass it produces gradients of the same shape but full of zeros. For integer inputs, autograd also returns zeros, instead of NaN.
- 8 Column vectors (shape  $[m, 1]$ ) and row vectors (shape  $[1, m]$ ) are often preferred over 1D vectors (shape  $[m]$ ) in Machine Learning, as they avoid ambiguity in some operations such as matrix multiplication or broadcasting, and they make the code more consistent whether there's just 1 feature or more.
- 9 Just like in NumPy, the `reshape()` method allows you to specify  $-1$  for one of the dimensions. This dimension's size is automatically calculated to ensure the new tensor has the same number of cells as the original.
- 10 The mean of the batch MSEs is equal to the overall MSE since all batches have the same size. Well, except the last batch, which is often smaller, but this makes very little difference (the correct RMSE is 0.6477 instead of 0.6476).
- 11 Heng-Tze Cheng et al., “Wide & Deep Learning for Recommender Systems”, *Proceedings of the First Workshop on Deep Learning for Recommender Systems* (2016): 7–10.
- 12 TorchVision includes a `ToTensor` transform which does all this, but it's deprecated so it's recommended to use this pipeline instead.
- 13 The `round()` method does not yet support the `decimals` argument on MPS devices, so if you run this code with MPS acceleration, you should move the predictions to the CPU before calling `round()`.
- 14 Another popular option is exporting your PyTorch model to the open ONNX standard using `torch.onnx.export()`. The ONNX model can then be used for inference in a wide variety of environments.
- 15 A few extra ANN architectures are presented in the online notebook at <https://homl.info/extranns>.

# Chapter 11. Training Deep Neural Networks

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles. This will be the 11th chapter of the final book. The GitHub repo is <https://github.com/ageron/handson-mlp>. If you’d like to be actively involved in reviewing and commenting on this draft, please reach out to the editor at [mcronin@oreilly.com](mailto:mcronin@oreilly.com).

In [Chapter 10](#) you built, trained, and fine-tuned several artificial neural networks using PyTorch. But they were shallow nets, with just a few hidden layers. What if you need to tackle a complex problem, such as detecting hundreds of types of objects in high-resolution images? You may need to train a much deeper ANN, perhaps with dozens or even hundreds of layers, each containing hundreds of neurons, linked by hundreds of thousands of connections. Training a deep neural network isn’t a walk in the park. Here are some of the problems you could run into:

- You may be faced with the problem of gradients growing ever smaller or larger, when flowing backward through the DNN during training. Both of these problems make lower layers very hard to train.
- You might not have enough training data for such a large network, or it might be too costly to label.
- Training may be extremely slow.
- A model with millions of parameters risks severely overfitting the training set, especially if there are not enough training instances or if they are too noisy.

In this chapter we will go through each of these problems and present various techniques to solve them. We will start by exploring the vanishing and exploding gradients problems and some of their most popular solutions, including smart weight initialization, better activation functions, batch-norm, layer-norm, and gradient clipping.

Next, we will look at transfer learning and unsupervised pretraining, which can help you tackle complex tasks even when you have little labeled data. Then we will discuss a variety of optimizers that can speed up training large models tremendously. We will also discuss how you can tweak the learning rate during training to speed up training and produce better models. Finally, we will cover a few popular regularization techniques for large neural networks:  $\ell_1$  and  $\ell_2$  regularization, dropout, MC dropout, and max-norm regularization.

With these tools, you will be able to train all sorts of deep nets. Welcome to *deep learning!*

## The Vanishing/Exploding Gradients Problems

As discussed in [Chapter 9](#), the backpropagation algorithm's second phase works by going from the output layer to the input layer, propagating the error gradient along the way. Once the algorithm has computed the gradient of the cost function with regard to each parameter in the network, it uses these gradients to update each parameter with a gradient descent step.

Unfortunately, gradients often get smaller and smaller as the algorithm progresses down to the lower layers. As a result, the gradient descent update leaves the lower layers' connection weights virtually unchanged, and training never converges to a good solution. This is called the *vanishing gradients* problem. In some cases, the opposite can happen: the gradients can grow bigger and bigger until layers get insanely large weight updates and the algorithm diverges. This is the *exploding gradients* problem, which surfaces most often in recurrent neural networks (see [\[Link to Come\]](#)). More generally, deep neural networks suffer from unstable gradients; different layers may learn at widely different speeds.

This unfortunate behavior was empirically observed long ago, and it was one of the reasons deep neural networks were mostly abandoned in the early 2000s. It wasn't clear what caused the gradients to be so unstable when training a DNN, but some light was shed in a [2010 paper](#) by Xavier Glorot and Yoshua Bengio.<sup>1</sup> The authors found a few suspects, including the combination of the popular sigmoid (logistic) activation function and the weight initialization technique that was most popular at the time (i.e., a normal distribution with a mean of 0 and a standard deviation of 1). In short, they showed that with this activation function and this initialization scheme, the variance of the outputs of each layer is much greater than the variance of its inputs. Going forward in the network, the variance keeps increasing after each layer until the activation

function saturates at the top layers. This saturation is actually made worse by the fact that the sigmoid function has a mean of 0.5, not 0 (the hyperbolic tangent function has a mean of 0 and behaves slightly better than the sigmoid function in deep networks).

Looking at the sigmoid activation function (see [Figure 11-1](#)), you can see that when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0 (i.e., the curve is flat at both extremes). Thus, when backpropagation kicks in it has virtually no gradient to propagate back through the network, and what little gradient exists keeps getting diluted as backpropagation progresses down through the top layers, so there is really nothing left for the lower layers.

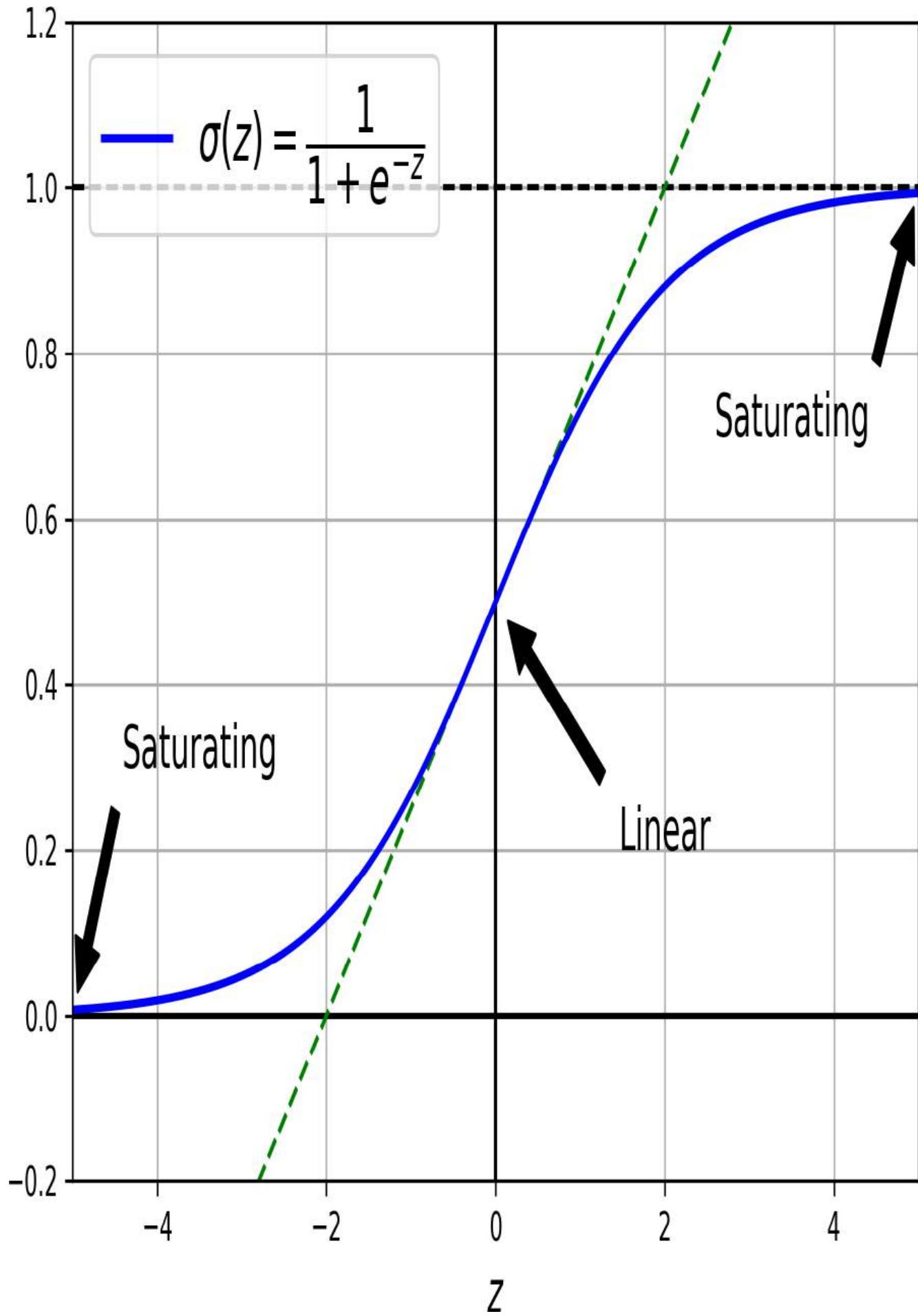


Figure 11-1. Sigmoid activation function saturation

## Glorot Initialization and He Initialization

In their paper, Glorot and Bengio propose a way to significantly alleviate the unstable gradients problem. They point out that we need the signal to flow properly in both directions: in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. We don't want the signal to die out, nor do we want it to explode and saturate. For the signal to flow properly, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs,<sup>2</sup> and we need the gradients to have equal variance before and after flowing through a layer in the reverse direction (please check out the paper if you are interested in the mathematical details). It is actually not possible to guarantee both unless the layer has an equal number of inputs and outputs (these numbers are called the *fan-in* and *fan-out* of the layer), but Glorot and Bengio proposed a good compromise that has proven to work very well in practice: the connection weights of each layer must be initialized randomly as described in [Equation 11-1](#), where  $\text{fan}_{\text{avg}} = (\text{fan}_{\text{in}} + \text{fan}_{\text{out}}) / 2$ . This initialization strategy is called *Xavier initialization* or *Glorot initialization*, after the paper's first author.

*Equation 11-1. Glorot initialization (when using the sigmoid activation function)*

$$\text{Normal distribution with mean 0 and variance } \sigma^2 = \frac{1}{\text{fan}_{\text{avg}}}$$

$$\text{Or a uniform distribution between } -r \text{ and } +r, \text{ with } r = \sqrt{\frac{3}{\text{fan}_{\text{avg}}}}$$

If you replace  $\text{fan}_{\text{avg}}$  with  $\text{fan}_{\text{in}}$  in [Equation 11-1](#), you get an initialization strategy that Yann LeCun proposed in the 1990s. He called it *LeCun initialization*. Genevieve Orr and Klaus-Robert Müller even recommended it in their 1998 book *Neural Networks: Tricks of the Trade* (Springer). LeCun initialization is equivalent to Glorot initialization when  $\text{fan}_{\text{in}} = \text{fan}_{\text{out}}$ . It took over a decade for researchers to realize how important this trick is. Using Glorot initialization can speed up training considerably, and it is one of the tricks that led to the success of deep learning.

Some papers have provided similar strategies for different activation functions, most notably a [2015 paper by Kaiming He et al.](#)<sup>3</sup> These strategies differ only by the scale of the variance and whether they use  $\text{fan}_{\text{avg}}$  or  $\text{fan}_{\text{in}}$ , as shown in [Table 11-1](#) (for the uniform distribution, just use  $r = \sqrt{3\sigma^2}$ ). The initialization strategy proposed for the

ReLU activation function and its variants is called *He initialization* or *Kaiming initialization*, after the paper’s first author. For SELU, use Yann LeCun’s initialization method, preferably with a normal distribution. We will cover all these activation functions shortly.

*Table 11-1. Initialization parameters for each type of activation function*

Initialization	Activation functions	$\sigma^2$ (Normal)
Xavier Glorot	None, tanh, sigmoid, softmax	$1 / fan_{avg}$
Kaiming He	ReLU, Leaky ReLU, ELU, GELU, Swish, Mish	$2 / fan_{in}$
Yann LeCun	SELU	$1 / fan_{in}$

For historical reasons, PyTorch’s `nn.Linear` module initializes its weights using Kaiming uniform initialization, except the weights are scaled down by a factor of  $\sqrt{6}$  (and the bias terms are also initialized randomly). Sadly, this is not the optimal scale for any common activation function.<sup>4</sup> One solution is to simply multiply the weights by  $\sqrt{6}$  (i.e.,  $6^{0.5}$ ) just after creating the `nn.Linear` layer, to get proper Kaiming initialization. To do this, you update the parameter’s `data` attribute. We will also zero out the biases:

```
import torch
import torch.nn as nn

layer = nn.Linear(40, 10)
layer.weight.data *= 6 ** 0.5 # Kaiming init (or 3 ** 0.5 for LeCun init)
torch.zeros_(layer.bias.data)
```

This works, but it’s clearer and less error-prone to use one of the initialization functions available in the `torch.nn.init` module:

```
nn.init.kaiming_uniform_(layer.weight)
nn.init.zeros_(layer.bias)
```

If you want to apply the same initialization method to the weights of every `nn.Linear` layer in a model, you can do so in the model’s constructor, after creating each `nn.Linear` layer. Alternatively, you can write a subclass of the `nn.Linear` class and

tweak its constructor to initialize the weights as you wish. But arguably the simplest option is to write a little function that takes a module, checks whether it's an instance of the `nn.Linear` class, and if so, applies the desired initialization function to its weights. You can then apply this function to the model and all of its submodules by passing it to the model's `apply()` method. For example:

```
def use_he_init(module):
    if isinstance(module, nn.Linear):
        nn.init.kaiming_uniform_(module.weight)
        nn.init.zeros_(module.bias)

model = nn.Sequential(nn.Linear(50, 40), nn.ReLU(), nn.Linear(40, 1), nn.ReLU())
model.apply(use_he_init)
```

### TIP

In a classifier, it's generally a good idea to scale down the weights of the output layer during initialization (e.g., by a factor of 10). Indeed, this will result in smaller logits at the beginning of training, which means they will be closer together, and hence the estimated probabilities will also be closer together. In other words, it encourages the model to be less confident about its predictions when training starts: this will avoid extreme losses and huge gradients that can often make the model's weights bounce around randomly at the start of training, losing time and potentially preventing the model from learning anything.

The `torch.nn.init` module also contains an `orthogonal_()` function which initializes the weights using a random orthogonal matrix, as proposed in a [2014 paper](#) by Andrew Saxe et al.<sup>5</sup> Orthogonal matrices have a number of useful mathematical properties, including the fact that they preserve norms: given an orthogonal matrix  $\mathbf{W}$  and an input vector  $\mathbf{x}$ , the norm of  $\mathbf{Wx}$  is equal to the norm of  $\mathbf{x}$ , and therefore the magnitude of the inputs is preserved in the outputs. When the inputs are standardized, this results in a stable variance through the layer, which prevents the activations and gradients from vanishing or exploding in a deep network (at least at the beginning of training). This initialization technique is much less common than the initialization techniques discussed earlier, but it can work well in recurrent neural nets ([Link to Come]) or generative adversarial networks ([Link to Come]).

And that's it! Scaling the weights properly will give a deep neural net a much better starting point for training.

## Better Activation Functions

One of the insights in the 2010 paper by Glorot and Bengio was that the problems with unstable gradients were in part due to a poor choice of activation function. Until then most people had assumed that if Mother Nature had chosen to use roughly sigmoid activation functions in biological neurons, they must be an excellent choice. But it turns out that other activation functions behave much better in deep neural networks—in particular, the ReLU activation function, mostly because it does not saturate for positive values, and also because it is very fast to compute.

Unfortunately, the ReLU activation function is not perfect. It suffers from a problem known as the *dying ReLUs*: during training, some neurons effectively “die”, meaning they stop outputting anything other than 0. In some cases, you may find that half of your network’s neurons are dead, especially if you used a large learning rate. A neuron dies when its weights get tweaked in such a way that the input of the ReLU function (i.e., the weighted sum of the neuron’s inputs plus its bias term) is negative for all instances in the training set. When this happens, it just keeps outputting zeros, and gradient descent does not affect it anymore because the gradient of the ReLU function is zero when its input is negative.<sup>6</sup>

To solve this problem, you may want to use a variant of the ReLU function, such as the *leaky ReLU*.

## Leaky ReLU

The leaky ReLU activation function is defined as  $\text{LeakyReLU}_\alpha(z) = \max(\alpha z, z)$  (see [Figure 11-2](#)). The hyperparameter  $\alpha$  defines how much the function “leaks”: it is the slope of the function for  $z < 0$ . Having a slope for  $z < 0$  ensures that leaky ReLUs never die; they can go into a long coma, but they have a chance to eventually wake up. A [2015 paper](#) by Bing Xu et al.<sup>7</sup> compared several variants of the ReLU activation function, and one of its conclusions was that the leaky variants always outperformed the strict ReLU activation function. In fact, setting  $\alpha = 0.2$  (a huge leak) seemed to result in better performance than  $\alpha = 0.01$  (a small leak). The paper also evaluated the *randomized leaky ReLU* (RReLU), where  $\alpha$  is picked randomly in a given range during training and is fixed to an average value during testing. RReLU also performed fairly well and seemed to act as a regularizer, reducing the risk of overfitting. Finally, the paper evaluated the *parametric leaky ReLU* (PReLU), where  $\alpha$  is authorized to be learned during training: instead of being a hyperparameter, it becomes a parameter that can be modified by backpropagation like any other parameter. PReLU was reported to strongly outperform ReLU on large image datasets, but on smaller datasets it runs the risk of overfitting the training set.

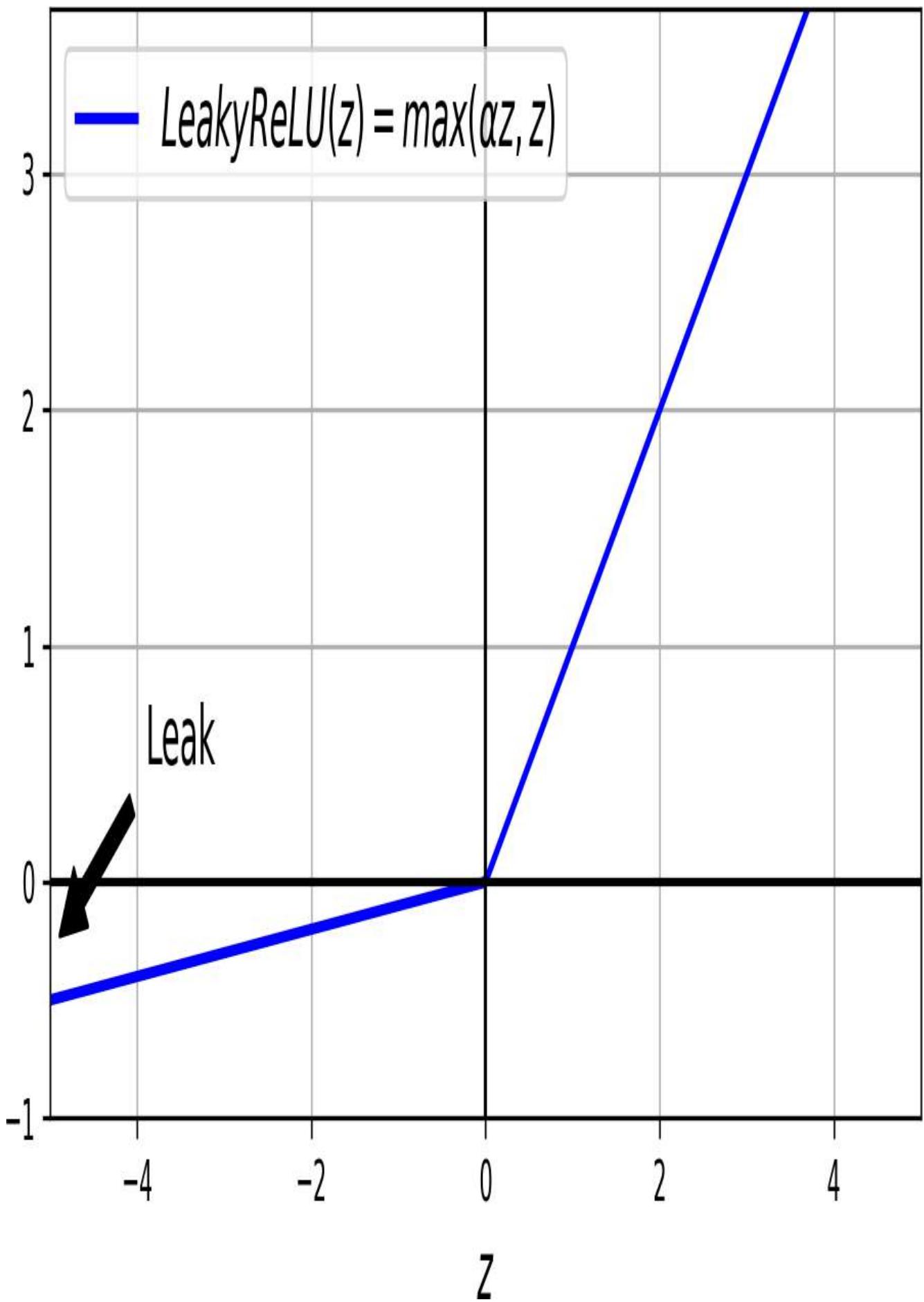


Figure 11-2. Leaky ReLU: like ReLU, but with a small slope for negative values

As you might expect, PyTorch includes modules for each of these activation functions: `nn.LeakyReLU`, `nn.RReLU` and `nn.PReLU`. Just like for other ReLU variants, you should use these along with Kaiming initialization, but the variance should be slightly smaller due to the negative slope: it should be scaled down by a factor of  $1 + \alpha^2$ . PyTorch supports this: you can pass the  $\alpha$  hyperparameter to the `kaiming_uniform_()` and `kaiming_normal_()` functions, along with `nonlinearity="leaky_relu"` to get the appropriately adjusted Kaiming initialization:

```
alpha = 0.2
model = nn.Sequential(nn.Linear(50, 40), nn.LeakyReLU(negative_slope=alpha))
nn.init.kaiming_uniform_(model[0].weight, alpha, nonlinearity="leaky_relu")
```

ReLU, leaky ReLU, and PReLU all suffer from the fact that they are not smooth functions: their derivatives abruptly change (at  $z = 0$ ). As we saw in [Chapter 4](#) when we discussed lasso, this sort of discontinuity can make gradient descent bounce around the optimum, and slow down convergence. So now we will look at some smooth variants of the ReLU activation function, starting with ELU and SELU.

## ELU and SELU

In 2015, a [paper](#) by Djork-Arné Clevert et al.<sup>8</sup> proposed a new activation function, called the *exponential linear unit* (ELU), that outperformed all the ReLU variants in the authors' experiments: training time was reduced, and the neural network performed better on the test set. [Equation 11-2](#) shows this activation function's definition.

Equation 11-2. ELU activation function

$$\text{ELU}_\alpha(z) = \begin{cases} \alpha(\exp(z) - 1) & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$$

The ELU activation function looks a lot like the ReLU function (see [Figure 11-3](#)), with a few major differences:

- It takes on negative values when  $z < 0$ , which allows the unit to have an average output closer to 0 and helps alleviate the vanishing gradients problem. The hyperparameter  $\alpha$  defines the opposite of the value that the ELU function approaches when  $z$  is a large negative number. It is usually set to 1, but you can tweak it like any other hyperparameter.

- It has a nonzero gradient for  $z < 0$ , which avoids the dead neurons problem.
- If  $\alpha$  is equal to 1 then the function is smooth everywhere, including around  $z = 0$ , which helps speed up gradient descent since it does not bounce as much to the left and right of  $z = 0$ .

Using ELU with PyTorch is as easy as using the `nn.ELU` module, along with Kaiming initialization. The main drawback of the ELU activation function is that it is slower to compute than the ReLU function and its variants (due to the use of the exponential function). Its faster convergence rate during training may compensate for that slow computation, but still, at test time an ELU network will be a bit slower than a ReLU network.

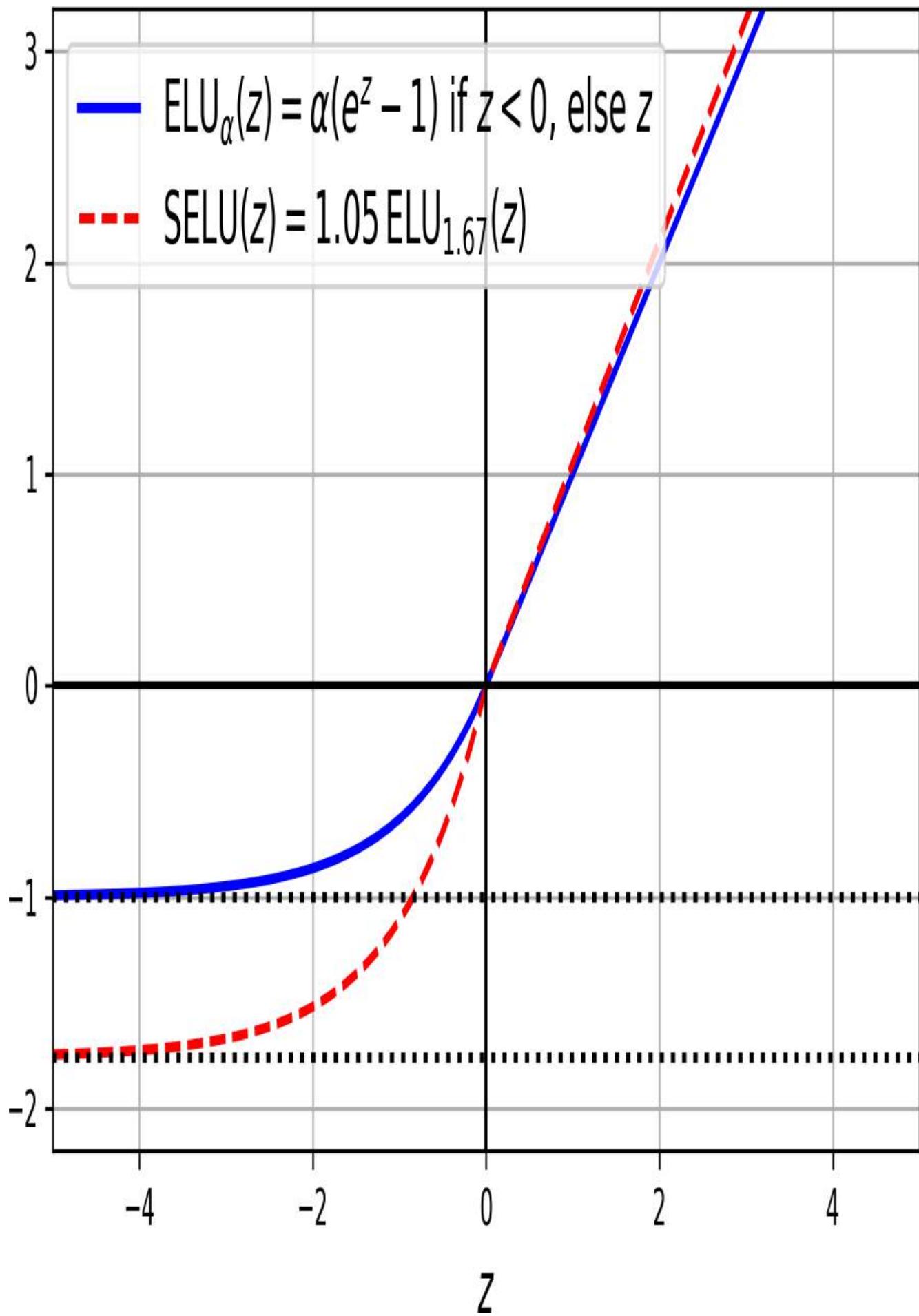


Figure 11-3. ELU and SELU activation functions

Not long after, a [2017 paper](#) by Günter Klambauer et al.<sup>9</sup> introduced the *scaled ELU* (SELU) activation function: as its name suggests, it is a scaled variant of the ELU activation function (about 1.05 times ELU, using  $\alpha \approx 1.67$ ). The authors showed that if you build a neural network composed exclusively of a stack of dense layers (i.e., an MLP), and if all hidden layers use the SELU activation function, then the network will *self-normalize*: the output of each layer will tend to preserve a mean of 0 and a standard deviation of 1 during training, which solves the vanishing/exploding gradients problem. As a result, the SELU activation function may outperform other activation functions for MLPs, especially deep ones. To use it with PyTorch, just use `nn.SELU`. There are, however, a few conditions for self-normalization to happen (see the paper for the mathematical justification):

- The input features must be standardized: mean 0 and standard deviation 1.
- Every hidden layer's weights must be initialized using LeCun normal initialization.
- The self-normalizing property is only guaranteed with plain MLPs. If you try to use SELU in other architectures, like recurrent networks (see [Link to Come]) or networks with *skip connections* (i.e., connections that skip layers, such as in Wide & Deep nets), it will probably not outperform ELU.
- You cannot use regularization techniques like  $\ell_1$  or  $\ell_2$  regularization, batch-norm, layer-norm, max-norm, or regular dropout (these are discussed later in this chapter).

These are significant constraints, so despite its promises, SELU did not gain a lot of traction. Moreover, three more activation functions seem to outperform it quite consistently on most tasks: GELU, Swish, and Mish.

## GELU, Swish, and Mish

*GELU* was introduced in a [2016 paper](#) by Dan Hendrycks and Kevin Gimpel.<sup>10</sup> Once again, you can think of it as a smooth variant of the ReLU activation function. Its definition is given in [Equation 11-3](#), where  $\Phi$  is the standard Gaussian cumulative distribution function (CDF):  $\Phi(z)$  corresponds to the probability that a value sampled randomly from a normal distribution of mean 0 and variance 1 is lower than  $z$ .

Equation 11-3. GELU activation function

$$\text{GELU}(z) = z \Phi(z)$$

As you can see in [Figure 11-4](#), GELU resembles ReLU: it approaches 0 when its input  $z$  is very negative, and it approaches  $z$  when  $z$  is very positive. However, whereas all the activation functions we've discussed so far were both convex and monotonic,<sup>11</sup> the GELU activation function is neither: from left to right, it starts by going straight, then it wiggles down, reaches a low point around  $-0.17$  (near  $z \approx -0.75$ ), and finally bounces up and ends up going straight toward the top right. This fairly complex shape and the fact that it has a curvature at every point may explain why it works so well, especially for complex tasks: gradient descent may find it easier to fit complex patterns. In practice, it often outperforms every other activation function discussed so far. However, it is a bit more computationally intensive, and the performance boost it provides is not always sufficient to justify the extra cost. That said, it is possible to show that it is approximately equal to  $z\sigma(1.702 z)$ , where  $\sigma$  is the sigmoid function: using this approximation also works very well, and it has the advantage of being much faster to compute.

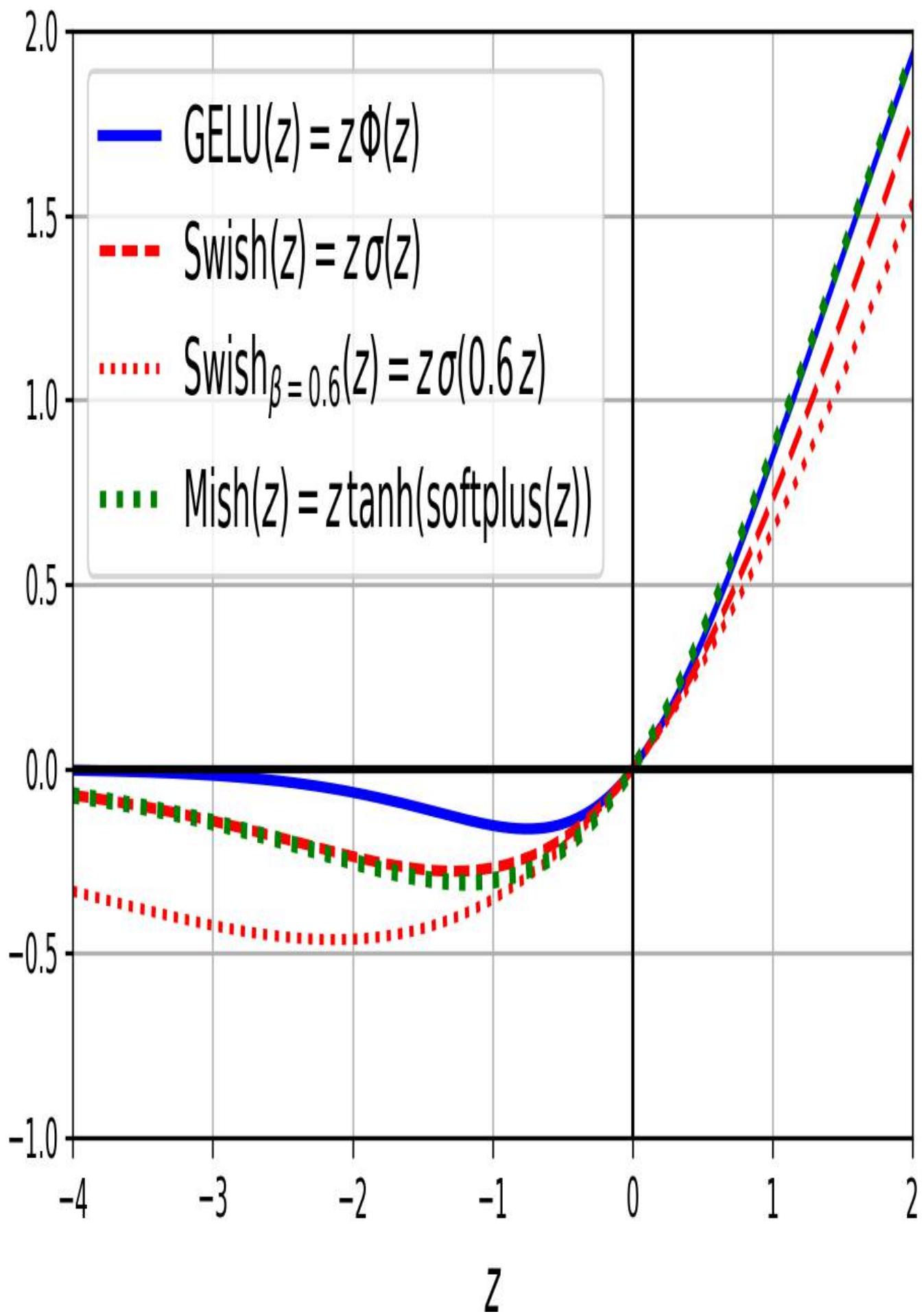


Figure 11-4. GELU, Swish, parametrized Swish, and Mish activation functions

The GELU paper also introduced the *sigmoid linear unit* (SiLU) activation function, which is equal to  $z\sigma(z)$ , but it was outperformed by GELU in the authors' tests.

Interestingly, a [2017 paper](#) by Prajit Ramachandran et al.<sup>12</sup> rediscovered the SiLU function by automatically searching for good activation functions. The authors named it *Swish*, and the name caught on. In their paper, Swish outperformed every other function, including GELU. Ramachandran et al. later generalized Swish by adding an extra scalar hyperparameter  $\beta$  to scale the sigmoid function's input. The generalized Swish function is  $\text{Swish}_\beta(z) = z\sigma(\beta z)$ , so GELU is approximately equal to the generalized Swish function using  $\beta = 1.702$ . You can tune  $\beta$  like any other hyperparameter. Alternatively, it's also possible to make  $\beta$  trainable and let gradient descent optimize it (a bit like PReLU): there is typically a single trainable  $\beta$  parameter for the whole model, or just one per layer, to keep the model efficient and avoid overfitting.

Another quite similar activation function is *Mish*, which was introduced in a [2019 paper](#) by Diganta Misra.<sup>13</sup> It is defined as  $\text{mish}(z) = z\tanh(\text{softplus}(z))$ , where  $\text{softplus}(z) = \log(1 + \exp(z))$ . Just like GELU and Swish, it is a smooth, nonconvex, and nonmonotonic variant of ReLU, and once again the author ran many experiments and found that Mish generally outperformed other activation functions—even Swish and GELU, by a tiny margin. [Figure 11-4](#) shows GELU, Swish (both with the default  $\beta = 1$  and with  $\beta = 0.6$ ), and lastly Mish. As you can see, Mish overlaps almost perfectly with Swish when  $z$  is negative, and almost perfectly with GELU when  $z$  is positive.

### TIP

So, which activation function should you use for the hidden layers of your deep neural networks? ReLU remains a good default for most tasks: it's often just as good as the more sophisticated activation functions, plus it's very fast to compute, and many libraries and hardware accelerators provide ReLU-specific optimizations. However, Swish is probably a better default for complex tasks, and you can even try parametrized Swish with a learnable  $\beta$  parameter for the most complex tasks. Mish may give you slightly better results, but it requires a bit more compute. If you care a lot about runtime latency, then you may prefer leaky ReLU, or parametrized leaky ReLU for complex tasks.

PyTorch supports GELU, Mish, and Swish out of the box (using `nn.GELU`, `nn.Mish`, and `nn.SiLU` respectively). PyTorch also includes simplified and approximated versions of several activation functions, which are much faster to compute and often more stable during training. These simplified versions have names starting with “Hard”, such as

`nn.Hardsigmoid`, `nn.Hardtanh`, and `nn.Hardswish`, and they are often used on mobile devices.

That's all for activation functions! Now, let's look at a completely different way to solve the unstable gradients problem: batch normalization.

## Batch Normalization

Although using Kaiming initialization along with ReLU (or any of its variants) can significantly reduce the danger of the vanishing/exploding gradients problems at the beginning of training, it doesn't guarantee that they won't come back during training.

In a [2015 paper](#),<sup>14</sup> Sergey Ioffe and Christian Szegedy proposed a technique called *batch normalization* (BN) that addresses these problems. The technique consists of adding an operation in the model just before or after the activation function of each hidden layer. This operation simply zero-centers and normalizes each input, then scales and shifts the result using two new parameter vectors per layer: one for scaling, the other for shifting. In other words, the operation lets the model learn the optimal scale and mean of each of the layer's inputs. In many cases, if you add a BN layer as the very first layer of your neural network, you do not need to standardize your training set (no need for `StandardScaler`); the BN layer will do it for you (well, approximately, since it only looks at one batch at a time, and it can also rescale and shift each input feature).

In order to zero-center and normalize the inputs, the algorithm needs to estimate each input's mean and standard deviation. It does so by evaluating the mean and standard deviation of the input over the current mini-batch (hence the name “batch normalization”). The whole operation is summarized step by step in [Equation 11-4](#).

*Equation 11-4. Batch normalization algorithm*

1.  $\boldsymbol{\mu}_B = \frac{1}{m_B} \sum_{i=1}^{m_B} \mathbf{x}^{(i)}$
2.  $\boldsymbol{\sigma}_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (\mathbf{x}^{(i)} - \boldsymbol{\mu}_B)^2$
3.  $\hat{\mathbf{x}}^{(i)} = \frac{\mathbf{x}^{(i)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \epsilon}}$
4.  $\mathbf{z}^{(i)} = \boldsymbol{\gamma} \otimes \hat{\mathbf{x}}^{(i)} + \boldsymbol{\beta}$

In this algorithm:

- $\mu_B$  is the vector of input means, evaluated over the whole mini-batch  $B$  (it contains one mean per input).
- $m_B$  is the number of instances in the mini-batch.
- $\mathbf{x}^{(i)}$  is the input vector of the batch norm layer for instance  $i$ .
- $\sigma_B$  is the vector of input standard deviations, also evaluated over the whole mini-batch (it contains one standard deviation per input).
- $\hat{\mathbf{x}}^{(i)}$  is the vector of zero-centered and normalized inputs for instance  $i$ .
- $\varepsilon$  is a tiny number that avoids division by zero and ensures the gradients don't grow too large (typically  $10^{-5}$ ). This is called a *smoothing term*.
- $\gamma$  is the output scale parameter vector for the layer (it contains one scale parameter per input).
- $\otimes$  represents element-wise multiplication (each input is multiplied by its corresponding output scale parameter).
- $\beta$  is the output shift (offset) parameter vector for the layer (it contains one shift parameter per input). Each input is offset by its corresponding shift parameter.
- $\mathbf{z}^{(i)}$  is the output of the BN operation. It is a rescaled and shifted version of the inputs.

So during training, BN standardizes its inputs, then rescales and offsets them. Good! What about at test time? Well, it's not that simple. Indeed, we may need to make predictions for individual instances rather than for batches of instances: in this case, we will have no way to compute each input's standard deviation. Moreover, even if we do have a batch of instances, it may be too small, or the instances may not be independent and identically distributed, so computing statistics over the batch instances would be unreliable. One solution is to wait until the end of training, then run the whole training set through the neural network and compute the mean and standard deviation of each input of the BN layer. These "final" input means and standard deviations can then be used instead of the batch input means and standard deviations when making predictions. However, most implementations of batch norm estimate these final statistics during training by using a moving average of the layer's batch input means and variances. This

is what PyTorch does automatically when you use its batch norm layers, such as `nn.BatchNorm1d` (which we will discuss in the next section). To sum up, four parameter vectors are learned in each batch norm layer:  $\gamma$  (the output scale vector) and  $\beta$  (the output offset vector) are learned through regular backpropagation, and  $\mu$  (the final input mean vector) and  $\sigma^2$  (the final input variance vector) are estimated using an exponential moving average. Note that  $\mu$  and  $\sigma^2$  are estimated during training, but they are used only after training, once you switch the model to evaluation mode using `model.eval()`:  $\mu$  and  $\sigma^2$  then replace  $\mu_B$  and  $\sigma_B^2$  in [Equation 11-4](#).

Ioffe and Szegedy demonstrated that batch norm considerably improved all the deep neural networks they experimented with, leading to a huge improvement in the ImageNet classification task (ImageNet is a large database of images classified into many classes, commonly used to evaluate computer vision systems). The vanishing gradients problem was strongly reduced, to the point that they could use saturating activation functions such as tanh and even sigmoid. The networks were also much less sensitive to the weight initialization. The authors were able to use much larger learning rates, significantly speeding up the learning process. Specifically, they note that:

*Applied to a state-of-the-art image classification model, batch norm achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. [...] Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.*

Finally, like a gift that keeps on giving, batch norm acts like a regularizer, reducing the need for other regularization techniques (such as dropout, described later in this chapter).

Batch normalization does, however, add some complexity to the model (although it can remove the need for normalizing the input data, as discussed earlier). Moreover, there is a runtime penalty: the neural network makes slower predictions due to the extra computations required at each layer. Fortunately, it's often possible to fuse the BN layer with the previous layer after training, thereby avoiding the runtime penalty. This is done by updating the previous layer's weights and biases so that it directly produces outputs of the appropriate scale and offset. For example, if the previous layer computes  $\mathbf{XW} + \mathbf{b}$ , then the BN layer will compute  $\gamma \otimes (\mathbf{XW} + \mathbf{b} - \mu) / \sigma + \beta$  (ignoring the smoothing term  $\varepsilon$  in the denominator). If we define  $\mathbf{W}' = \gamma \otimes \mathbf{W} / \sigma$  and  $\mathbf{b}' = \gamma \otimes (\mathbf{b} - \mu) / \sigma + \beta$ , the equation simplifies to  $\mathbf{XW}' + \mathbf{b}'$ . So, if we replace the previous layer's weights and

biases (**W** and **b**) with the updated weights and biases (**W'** and **b'**), we can get rid of the BN layer. This is one of the optimizations performed by `optimize_for_inference()` (see [Chapter 10](#)).

## NOTE

You may find that training is rather slow, because each epoch takes much more time when you use batch norm. This is usually counterbalanced by the fact that convergence is much faster with BN, so it will take fewer epochs to reach the same performance. All in all, *wall time* will usually be shorter (this is the time measured by the clock on your wall).

## Implementing batch norm with PyTorch

As with most things with PyTorch, implementing batch norm is straightforward and intuitive. Just add a `nn.BatchNorm1d` layer before or after each hidden layer's activation function, and specify the number of inputs of each BN layer. You may also add a BN layer as the first layer in your model, which removes the need to standardize the inputs manually. For example, let's create a Fashion MNIST image classifier (similar to the one we built in [Chapter 10](#)) using BN as the first layer in the model (after flattening the input images), then again after each hidden layer:

```
model = nn.Sequential(  
    nn.Flatten(),  
    nn.BatchNorm1d(1 * 28 * 28),  
    nn.Linear(1 * 28 * 28, 300),  
    nn.ReLU(),  
    nn.BatchNorm1d(300),  
    nn.Linear(300, 100),  
    nn.ReLU(),  
    nn.BatchNorm1d(100),  
    nn.Linear(100, 10)  
)
```

You can now train the model normally (as you learned in [Chapter 10](#)), and that's it! In this tiny example with just two hidden layers, batch norm is unlikely to have a large impact, but for deeper networks it can make a tremendous difference.

## WARNING

Since batch norm behaves differently during training and during evaluation, it's critical to switch to training mode during training (using `model.train()`), and switch to evaluation mode during evaluation (using `model.eval()`). Forgetting to do so is one of the most common mistakes.

If you look at the parameters of the first BN layer, you will find two: `weight` and `bias`, which correspond to  $\gamma$  and  $\beta$  in [Equation 11-4](#):

```
>>> dict(model[1].named_parameters()).keys()  
dict_keys(['weight', 'bias'])
```

And if you look at the buffers of this same BN layer, you will find three: `running_mean`, `running_var` and `num_batches_tracked`. The first two correspond to the running means  $\mu$  and  $\sigma^2$  discussed earlier, and `num_batches_tracked` simply counts the number of batches seen during training:

```
>>> dict(model[1].named_buffers()).keys()  
dict_keys(['running_mean', 'running_var', 'num_batches_tracked'])
```

The authors of the BN paper argued in favor of adding the BN layers before the activation functions, rather than after (as we just did). There is some debate about this, and it seems to depend on the task, so you can experiment with this to see which option works best on your dataset. If you move the BN layers before the activation functions, you can also remove the bias term from the previous `nn.Linear` layers by setting their `bias` hyperparameter to `False`. Indeed, a batch norm layer already includes one bias term per input. You can also drop the first BN layer to avoid sandwiching the first hidden layer between two BN layers, but this means you should normalize the training set before training. The updated code looks like this:

```
model = nn.Sequential(  
    nn.Flatten(),  
    nn.Linear(1 * 28 * 28, 300, bias=False),  
    nn.BatchNorm1d(300),  
    nn.ReLU(),  
    nn.Linear(300, 100, bias=False),  
    nn.BatchNorm1d(100),  
    nn.ReLU(),  
    nn.Linear(100, 10)  
)
```

The `nn.BatchNorm1d` class has a few hyperparameters you can tweak. The defaults will usually be fine, but you may occasionally need to tweak the `momentum`. This hyperparameter is used by the `BatchNorm1d` layer when it updates the exponential moving averages; given a new value  $\mathbf{v}$  (i.e., a new vector of input means or variances computed over the current batch), the layer updates the running average  $\hat{\mathbf{v}}$  using the following equation:

$$\hat{\mathbf{v}} \leftarrow \mathbf{v} \times \text{momentum} + \hat{\mathbf{v}} \times (1 - \text{momentum})$$

A good momentum value is typically close to 0; for example, 0.01, or 0.001. You want more 0s for smaller mini-batches, and less for larger mini-batches. The default is 0.1, which is good for large batch sizes, but not great for small batch sizes such as 32 or 64.

### WARNING

When people talk about “momentum” in the context of a running mean, they usually refer to the weight of the current running mean in the update equation. Sadly, for historical reasons, PyTorch uses the opposite meaning in the BN layers. However, other parts of PyTorch use the conventional meaning (e.g., in optimizers), so don’t get confused.

Now let’s talk about dimensionality. In the examples above, we flattened the input images before sending them through the first `nn.BatchNorm1d` layer. This is because a `nn.BatchNorm1d` layer works on batches of shape `[batch_size, num_features]` (just like the `nn.Linear` layer does), so you would get an error if you moved it before the `nn.Flatten` layer. However, you could use a `nn.BatchNorm2d` layer before the `nn.Flatten` layer: indeed, it expects its inputs to be image batches of shape `[batch_size, channels, height, width]`, and it computes the batch mean and variance across both the batch dimension (dimension 0) and the spatial dimensions (dimensions 2 and 3). This means that all pixels in the same batch and channel get normalized using the same mean and variance: the `nn.BatchNorm2d` layer only has one weight per channel and one bias per channel (e.g., 3 weights and 3 bias terms for color images with 3 channels for red, green, and blue). This generally works better when dealing with image datasets. There’s also a `nn.BatchNorm3d` layer which expects batches of shape `[batch_size, channels, depth, height, width]`: this is useful for datasets of 3D images such as CT scans.

The `nn.BatchNorm1d` layer can also work on batches of sequences. The convention in PyTorch is to represent batches of sequences as 3D tensors of shape `[batch_size,`

`sequence_length, num_features`]. For example, suppose you work on particle physics and you have a dataset of particle trajectories, where each trajectory is composed of a sequence of 100 points in 3D space, then a batch of 32 trajectories will have a shape of [32, 100, 3]. However, the `nn.BatchNorm1d` layer expects the shape to be [batch\_size, num\_features, sequence\_length], and it computes the batch mean and variance across the first and last dimensions to get one mean and variance per feature. So you must permute the last two dimensions of the data using `X.permute(0, 2, 1)` before letting it go through the `nn.BatchNorm1d` layer. We will discuss sequences further in [Link to Come].

Batch normalization has become one of the most-used layers in deep neural networks, especially deep convolutional neural networks discussed in ([Link to Come]), to the point that it is often omitted in the architecture diagrams: it is assumed that BN is added after every layer. That said, it is not perfect. In particular, the computed statistics for an instance are biased by the other samples in a batch, which may reduce performance (especially for small batch sizes). Moreover, BN struggles with some architectures such as recurrent nets, as we will see in [Link to Come]. For these reasons, batch-norm is more and more often replaced by layer-norm.

## Layer Normalization

Layer normalization (LN) is very similar to batch norm, but instead of normalizing across the batch dimension, LN normalizes across the feature dimensions. This simple idea was introduced by Jimmy Lei Ba et al. in a [2016 paper](#):<sup>15</sup>, and initially applied mostly to recurrent nets. However, in recent years it has been successfully applied to many other architectures, such as convolutional nets, transformers, diffusion nets, and more.

One advantage is that it can compute the required statistics on the fly, at each time step, independently for each instance. This also means that it behaves the same way during training and testing (as opposed to BN), and it does not need to use exponential moving averages to estimate the feature statistics across all instances in the training set, like BN does. Lastly, layer normalization learns a scale and an offset parameter for each input feature, just like BN does.

PyTorch includes a `nn.LayerNorm` module. To create an instance, you must simply indicate the size of the dimensions that you want to normalize over. These must be the last dimension(s) of the inputs. For example, if the inputs are batches of  $100 \times 200$  RGB

images of shape [3, 100, 200], and you want to normalize each image over each of the three color channels separately, you would use the following `nn.LayerNorm` module:

```
inputs = torch.randn(32, 3, 100, 200) # a batch of random RGB images
layer_norm = nn.LayerNorm([100, 200])
result = layer_norm(inputs) # normalizes over the last two dimensions
```

The following code produces the same result:

```
means = inputs.mean(dim=[2, 3], keepdim=True) # shape: [32, 3, 1, 1]
vars_ = inputs.var(dim=[2, 3], keepdim=True, unbiased=False) # shape: same
stds = torch.sqrt(vars_ + layer_norm.eps) # eps is a smoothing term (1e-5)
result = layer_norm.weight * (inputs - means) / stds + layer_norm.bias
# result shape: [32, 3, 100, 200]
```

However, most computer vision architectures that use layer-norm normalize over all channels at once. For this, you must include the size of the channels dimension when creating the `nn.LayerNorm` module:

```
layer_norm = nn.LayerNorm([3, 100, 200])
result = layer_norm(inputs) # normalizes over the last three dimensions
```

And that's all there is to it! Now let's look at one last technique to stabilize gradients during training: gradient clipping.

## Gradient Clipping

Another technique to mitigate the exploding gradients problem is to clip the gradients during backpropagation so that they never exceed some threshold. This is called *gradient clipping*.<sup>16</sup> This technique is generally used in recurrent neural networks, where using batch norm is tricky (as you will see in [Link to Come]).

In PyTorch, gradient clipping is generally implemented by calling either `torch.nn.utils.clip_grad_norm_()` or `torch.nn.utils.clip_grad_value_()` at each iteration during training, right after the gradients are computed (i.e., after `loss.backward()`). Both functions take as a first argument the list of model parameters whose gradients must be clipped—typically all of them (`model.parameters()`). The `clip_grad_norm_()` function clips each gradient vector's norm if it exceeds the given `max_norm` argument. This is a hyperparameter you can tune (a typical default value is 1.0). The `clip_grad_value_()` function independently clips the individual components of the gradient vector between `-clip_value` and `+clip_value`, where

`clip_value` is a hyperparameter you can tune. For example, this training loop clips the norm of each gradient vector to 1.0:

```
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        X_batch, y_batch = X_batch.to(device), y_batch.to(device)
        y_pred = model(X_batch)
        loss = loss_fn(y_pred, y_batch)
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        optimizer.zero_grad()
```

Note that `clip_grad_value_()` will change the orientation of the gradient vector when its components are clipped. For instance, if the original gradient vector is [0.9, 100.0], it points mostly in the direction of the second dimension; but once you clip it by value, you get [0.9, 1.0], which points roughly at the diagonal between the two axes. Despite this reorientation, this approach actually works quite well in practice. If you clipped the same vector by norm, the result would be [0.00899964, 0.9999595]: this would preserve the vector's orientation, but almost eliminate the first component. The best clipping function to use depends on the dataset.

## Reusing Pretrained Layers

It is generally not a good idea to train a very large DNN from scratch without first trying to find an existing neural network that accomplishes a similar task to the one you are trying to tackle (I will discuss how to find them in [Link to Come]). If you find such a neural network, then you can generally reuse most of its layers, except for the top ones. This technique is called *transfer learning*. It will not only speed up training considerably, but also require significantly less training data.

Suppose you have access to a DNN that was trained to classify pictures into 100 different categories, including animals, plants, vehicles, and everyday objects, and you now want to train a DNN to classify specific types of vehicles. These tasks are very similar, even partly overlapping, so you should try to reuse parts of the first network (see [Figure 11-5](#)).

## NOTE

If the input pictures for your new task don't have the same size as the ones used in the original task, you will usually have to add a preprocessing step to resize them to the size expected by the original model. More generally, transfer learning will work best when the inputs have similar low-level features. For example, a neural net trained on regular pictures taken from mobile phones will help with many other tasks on mobile phone pictures, but it will likely not help at all on satellite images or medical images.

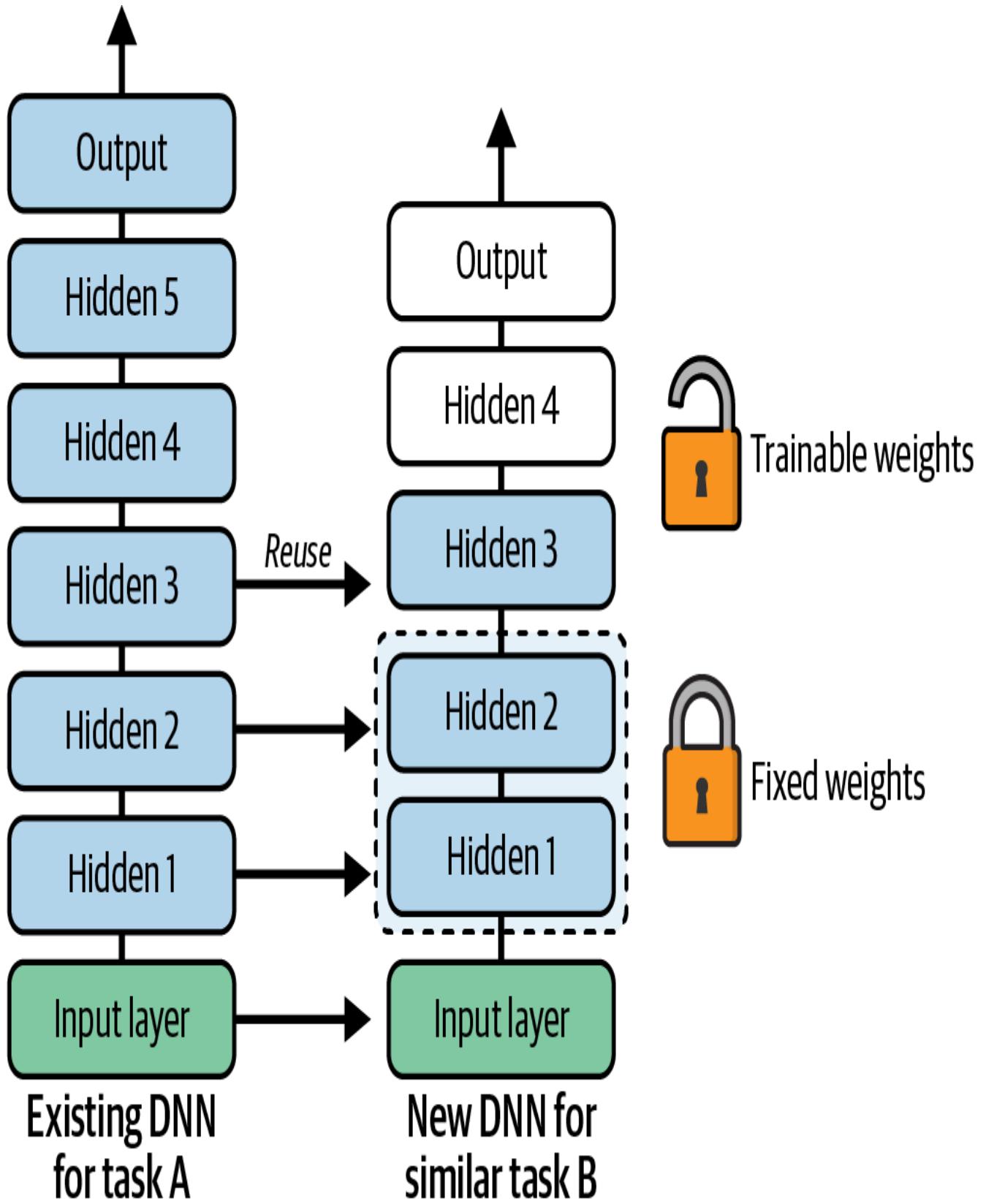


Figure 11-5. Reusing pretrained layers

The output layer of the original model should usually be replaced because it is most likely not useful at all for the new task, and it may not even have the right number of outputs.

Similarly, the upper hidden layers of the original model are less likely to be as useful as the lower layers, since the high-level features that are most useful for the new task may differ significantly from the ones that were most useful for the original task. You want to find the right number of layers to reuse.

### TIP

The more similar the tasks are, the more layers you will want to reuse (starting with the lower layers). For very similar tasks, try to keep all the hidden layers and just replace the output layer.

Try freezing all the reused layers first (i.e., make their parameters non-trainable by setting `requires_grad` to `False` so that gradient descent won't modify them and they will remain fixed), then train your model and see how it performs. Then try unfreezing one or two of the top hidden layers to let backpropagation tweak them and see if performance improves. The more training data you have, the more layers you can unfreeze. It is also useful to reduce the learning rate when you unfreeze reused layers: this will avoid wrecking their fine-tuned weights.

If you still cannot get good performance, and you have little training data, try dropping the top hidden layer(s) and freezing all the remaining hidden layers again. You can iterate until you find the right number of layers to reuse. If you have plenty of training data, you may try replacing the top hidden layers instead of dropping them, and even adding more hidden layers.

## Transfer Learning with PyTorch

Let's look at an example. Suppose the Fashion MNIST dataset only contained eight classes—for example, all the classes except for Pullover and T-shirt/top. Someone built and trained a PyTorch model on that set and got reasonably good performance (~92% accuracy). Let's call this model A. You now want to tackle a different task: you have images of T-shirts and pullovers, and you want to train a binary classifier: positive for T-shirt/top, negative for Pullover. Your dataset is tiny; you only have 20 labeled images! When you train a new model for this task (let's call it model B) with the same architecture as model A, you get 71.6% test accuracy. While drinking your morning coffee, you realize that your task is quite similar to task A, so perhaps transfer learning can help? Let's find out!

First, let's look at model A:

```

torch.manual_seed(42)

model_A = nn.Sequential(
    nn.Flatten(),
    nn.Linear(1 * 28 * 28, 100),
    nn.ReLU(),
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.Linear(100, 100),
    nn.ReLU(),
    nn.Linear(100, 8)
)
[...] # train this model or load pretrained weights

```

We can now reuse the layers we want, for example all layers except for the output layer:

```

import copy

torch.manual_seed(42)
reused_layers = copy.deepcopy(model_A[:-1])
model_B_on_A = nn.Sequential(
    *reused_layers,
    nn.Linear(100, 1) # new output layer for task B
).to(device)

```

In this code, we use Python's `copy.deepcopy()` function to copy all the modules in the `nn.Sequential` module (along with all their data and submodules), except for the last layer. Since we're making a deep copy, all the submodules are copied as well. Then we create `model_B_on_A`, which is a `nn.Sequential` model based on the reused layers of `model_A`, plus a new output layer for task B: it has a single output since task B is binary classification.

You could start training `model_B_on_A` for task B now, but since the new output layer was initialized randomly it will make large errors (at least during the first few epochs), so there will be large error gradients that may wreck the reused weights. To avoid this, one approach is to freeze the reused layers during the first few epochs, giving the new layer some time to learn reasonable weights:

```

for layer in model_B_on_A[:-1]:
    for param in layer.parameters():
        param.requires_grad = False

```

Now you can train `model_B_on_A`. But don't forget that task B is binary classification, so you must switch the loss to `nn.BCEWithLogitsLoss` (or to `nn.BCELoss` if you

prefer to add a `nn.Sigmoid` activation function on the output layer), as we discussed in [Chapter 10](#). Also, if you are using `torchmetrics`, make sure to set `task="binary"` when creating the `Accuracy` metric:

```
xentropy = nn.BCEWithLogitsLoss()  
accuracy = torchmetrics.Accuracy(task="binary").to(device)  
[...] # train model_B_on_A
```

After you have trained the model for a few epochs, you can unfreeze the reused layers (setting `param.requires_grad = True` for all parameters), reduce the learning rate, and continue training to fine-tune the reused layers for task B.

So, what's the final verdict? Well, this model's test accuracy is 92.5%, which is much better than the 71.6% accuracy we reached without pretraining!

Are you convinced? Well, you shouldn't be: I cheated! I tried many configurations until I found one that demonstrated a strong improvement. If you try to change the classes or the random seed, you will see that the improvement generally drops, or even vanishes or reverses. What I did is called "torturing the data until it confesses". When a paper just looks too positive, you should be suspicious: perhaps the flashy new technique does not actually help much (in fact, it may even degrade performance), but the authors tried many variants and reported only the best results—which may be due to sheer luck—without mentioning how many failures they encountered along the way: that's called *p-hacking*. Most of the time, this is not malicious, but it is part of the reason why so many results in science can never be reproduced.

But why did I cheat? It turns out that transfer learning does not work very well with small dense networks, presumably because small networks learn few patterns, and dense networks learn very specific patterns, which are unlikely to be useful for other tasks. Transfer learning works best with deep convolutional neural networks and with transformer architectures. We will revisit transfer learning in [\[Link to Come\]](#) and [\[Link to Come\]](#), using the techniques we just discussed (and this time it will work fine without cheating, I promise!).

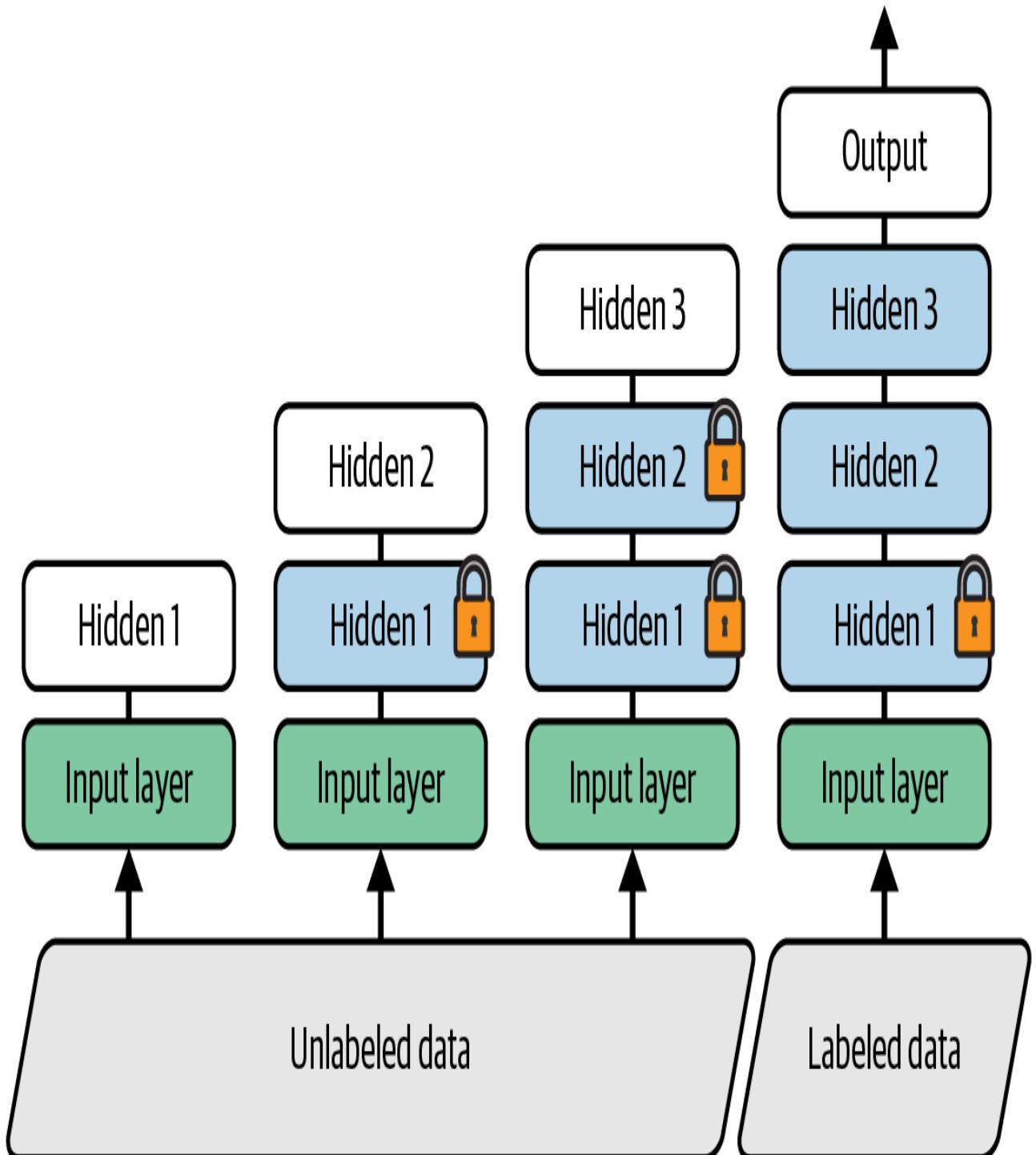
## Unsupervised Pretraining

Suppose you want to tackle a complex task for which you don't have much labeled training data, but unfortunately you cannot find a model trained on a similar task. Don't lose hope! First, you should try to gather more labeled training data, but if you can't, you may still be able to perform *unsupervised pretraining* (see [Figure 11-6](#)). Indeed, it

is often cheap to gather unlabeled training examples, but expensive to label them. If you can gather plenty of unlabeled training data, you can try to use it to train an unsupervised model, such as an autoencoder (see [Link to Come]). Then you can reuse the lower layers of the autoencoder, add the output layer for your task on top, and fine-tune the final network using supervised learning (i.e., with the labeled training examples).

It is this technique that Geoffrey Hinton and his team used in 2006, and which led to the revival of neural networks and the success of deep learning. Until 2010, unsupervised pretraining—typically with restricted Boltzmann machines (RBMs; see the notebook at <https://homl.info/extr-anns>)—was the norm for deep nets, and only after the vanishing gradients problem was alleviated did it become much more common to train DNNs purely using supervised learning. Unsupervised pretraining (today typically using autoencoders or diffusion models rather than RBMs) is still a good option when you have a complex task to solve, no similar model you can reuse, and little labeled training data but plenty of unlabeled training data.

Note that in the early days of deep learning it was difficult to train deep models, so people would use a technique called *greedy layer-wise pretraining* (depicted in [Figure 11-6](#)). They would first train an unsupervised model with a single layer, typically an RBM, then they would freeze that layer and add another one on top of it, then train the model again (effectively just training the new layer), then freeze the new layer and add another layer on top of it, train the model again, and so on. Nowadays, things are much simpler: people generally train the full unsupervised model in one shot and use models such as autoencoders or diffusion models rather than RBMs.



Unsupervised (e.g., autoencoders)

Supervised

Train layer 1

Train layer 2

Train layer 3

Train final model

*Figure 11-6. Greedy layer-wise pretraining used in the early days of deep learning; nowadays the unsupervised part is typically done in one shot on all the data rather than one layer at a time*

## Pretraining on an Auxiliary Task

If you do not have much labeled training data, one last option is to train a first neural network on an auxiliary task for which you can easily obtain or generate labeled training data, then reuse the lower layers of that network for your actual task. The first neural network's lower layers will learn feature detectors that will likely be reusable by the second neural network.

For example, if you want to build a system to recognize faces, you may only have a few pictures of each individual—clearly not enough to train a good classifier. Gathering hundreds of pictures of each person would not be practical. You could, however, use a public dataset containing millions of pictures of people (such as VGGFace2) and train a first neural network to detect whether or not two different pictures feature the same person. Such a network would learn good feature detectors for faces, so reusing its lower layers would allow you to train a good face classifier that uses little training data.

### WARNING

You could also just scrape pictures of random people from the web, but this would probably be illegal. Firstly, photos are usually copyrighted by their creators, and websites like Instagram or Facebook enforce these copyright protections through their terms of service, which prohibit scraping and unauthorized use. Secondly, over 40 countries require explicit consent for collecting and processing personal data, including facial images.

For natural language processing (NLP) applications, you can download a corpus of millions of text documents and automatically generate labeled data from it. For example, you could randomly mask out some words and train a model to predict what the missing words are (e.g., it should predict that the missing word in the sentence “What \_\_\_ you saying?” is probably “are” or “were”). If you can train a model to reach good performance on this task, then it will already know quite a lot about language, and you can certainly reuse it for your actual task and fine-tune it on your labeled data (this is basically how large language models are trained and fine-tuned, as we will see in [Link to Come]).

## NOTE

*Self-supervised learning* is when you automatically generate the labels from the data itself, as in the text-masking example, then you train a model on the resulting “labeled” dataset using supervised learning techniques.

# Faster Optimizers

Training a very large deep neural network can be painfully slow. So far we have seen four ways to speed up training (and reach a better solution): applying a good initialization strategy for the connection weights, using a good activation function, using batch-norm or layer-norm, and reusing parts of a pretrained network (possibly built for an auxiliary task or using unsupervised learning). Another huge speed boost comes from using a faster optimizer than the regular gradient descent optimizer. In this section we will present the most popular optimization algorithms: momentum, Nesterov accelerated gradient, AdaGrad, RMSProp, and finally Adam and its variants.

## Momentum

Imagine a bowling ball rolling down a gentle slope on a smooth surface: it will start out slowly, but it will quickly pick up momentum until it eventually reaches terminal velocity (if there is some friction or air resistance). This is the core idea behind *momentum optimization*, proposed by Boris Polyak in 1964.<sup>17</sup> In contrast, regular gradient descent will take small steps when the slope is gentle and big steps when the slope is steep, but it will never pick up speed. As a result, regular gradient descent is generally much slower to reach the minimum than momentum optimization.

Recall that gradient descent updates the weights  $\theta$  by directly subtracting the gradient of the cost function  $J(\theta)$  with regard to the weights ( $\nabla_{\theta}J(\theta)$ ) multiplied by the learning rate  $\eta$ . The equation is  $\theta \leftarrow \theta - \eta \nabla_{\theta}J(\theta)$ . It does not care about what the earlier gradients were. If the local gradient is tiny, it goes very slowly.

Momentum optimization cares a great deal about what previous gradients were: at each iteration, it subtracts the local gradient from the *momentum vector*  $m$  (multiplied by the learning rate  $\eta$ ), and it updates the weights by adding this momentum vector (see [Equation 11-5](#)). In other words, the gradient is used as a force learning to an acceleration, not as a speed. To simulate some sort of friction mechanism and prevent the momentum from growing too large, the algorithm introduces a new hyperparameter

$\beta$ , called the *momentum coefficient*, which must be set between 0 (high friction) and 1 (no friction). A typical momentum value is 0.9.

*Equation 11-5. Momentum algorithm*

1.  $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta)$
2.  $\theta \leftarrow \theta + \mathbf{m}$

You can verify that if the gradient remains constant, the terminal velocity (i.e., the maximum size of the weight updates) is equal to that gradient multiplied by the learning rate  $\eta$  multiplied by  $1 / (1 - \beta)$  (ignoring the sign). For example, if  $\beta = 0.9$ , then the terminal velocity is equal to 10 times the gradient times the learning rate, so momentum optimization ends up going 10 times faster than gradient descent! In practice, the gradients are not constant, so the speedup is not always as dramatic, but momentum optimization can escape from plateaus much faster than regular gradient descent. We saw in [Chapter 4](#) that when the inputs have very different scales, the cost function will look like an elongated bowl (see [Figure 4-7](#)). Gradient descent goes down the steep slope quite fast, but then it takes a very long time to go down the valley. In contrast, momentum optimization will roll down the valley faster and faster until it reaches the bottom (the optimum). In deep neural networks that don't use batch-norm or layer-norm, the upper layers will often end up having inputs with very different scales, so using momentum optimization helps a lot. It can also help roll past local optima.

### NOTE

Due to the momentum, the optimizer may overshoot a bit, then come back, overshoot again, and oscillate like this many times before stabilizing at the minimum. This is one of the reasons why it's good to have a bit of friction in the system: it reduces these oscillations and thus speeds up convergence.

Implementing momentum optimization in PyTorch is a no-brainer: just use the `SGD` optimizer and set its `momentum` hyperparameter, then lie back and profit!

```
optimizer = torch.optim.SGD(model.parameters(), momentum=0.9, lr=0.05)
```

The one drawback of momentum optimization is that it adds yet another hyperparameter to tune. However, the momentum value of 0.9 usually works well in practice and almost always goes faster than regular gradient descent.

## Nesterov Accelerated Gradient

One small variant to momentum optimization, proposed by [Yurii Nesterov in 1983](#),<sup>18</sup> is almost always faster than regular momentum optimization. The *Nesterov accelerated gradient* (NAG) method, also known as *Nesterov momentum optimization*, measures the gradient of the cost function not at the local position  $\theta$  but slightly ahead in the direction of the momentum, at  $\theta + \beta\mathbf{m}$  (see [Equation 11-6](#)).

*Equation 11-6. Nesterov accelerated gradient algorithm*

1.  $\mathbf{m} \leftarrow \beta\mathbf{m} - \eta\nabla_{\theta}J(\theta + \beta\mathbf{m})$
2.  $\theta \leftarrow \theta + \mathbf{m}$

This small tweak works because in general the momentum vector will be pointing in the right direction (i.e., toward the optimum), so it will be slightly more accurate to use the gradient measured a bit farther in that direction rather than the gradient at the original position, as you can see in [Figure 11-7](#) (where  $\nabla_1$  represents the gradient of the cost function measured at the starting point  $\theta$ , and  $\nabla_2$  represents the gradient at the point located at  $\theta + \beta\mathbf{m}$ ).

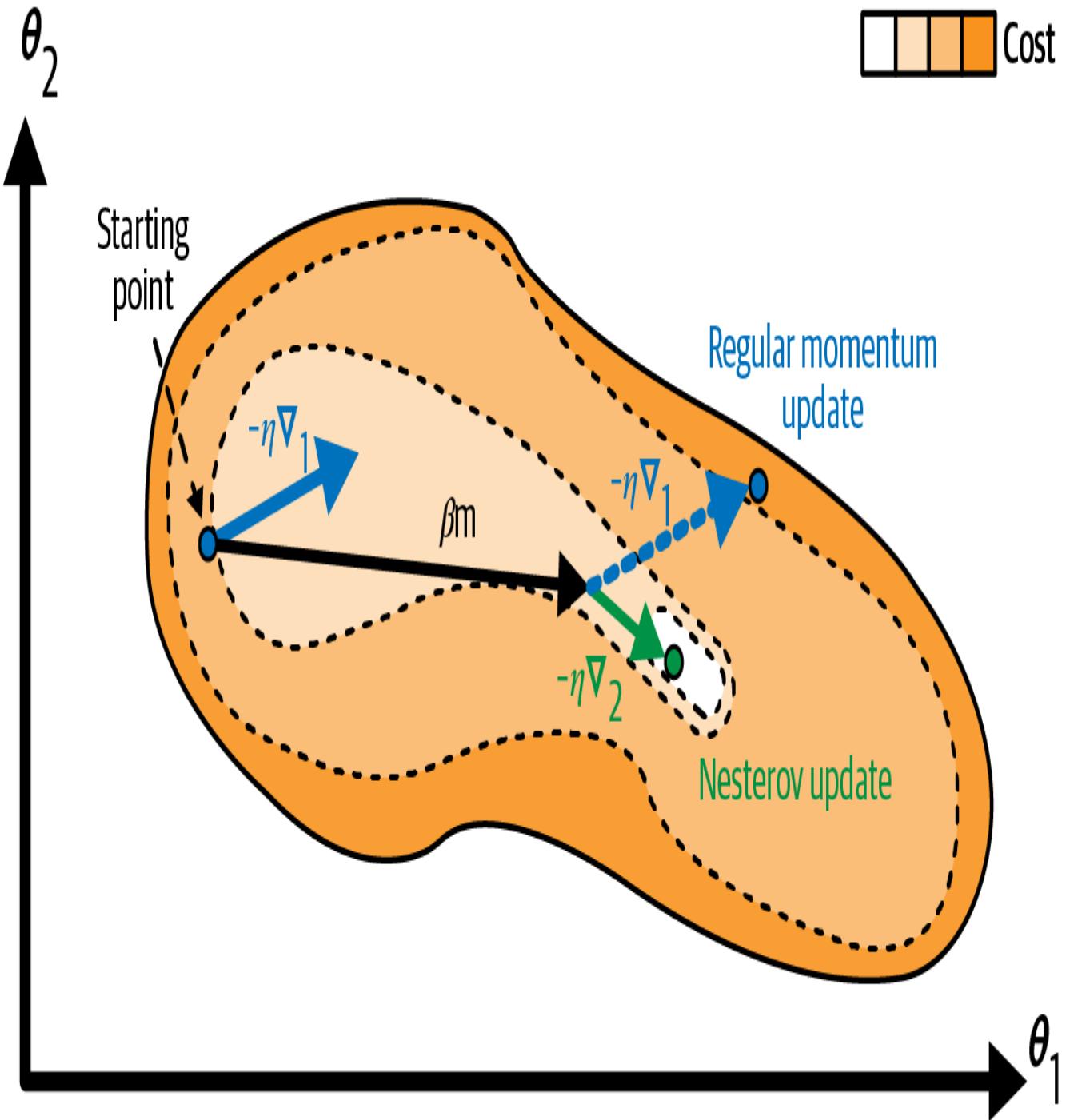


Figure 11-7. Regular versus Nesterov momentum optimization: the former applies the gradients computed before the momentum step, while the latter applies the gradients computed after

As you can see, the Nesterov update ends up closer to the optimum. After a while, these small improvements add up and NAG ends up being significantly faster than regular momentum optimization. Moreover, note that when the momentum pushes the weights across a valley,  $\nabla_1$  continues to push farther across the valley, while  $\nabla_2$  pushes back toward the bottom of the valley. This helps reduce oscillations and thus NAG converges faster.

To use NAG, simply set `nesterov=True` when creating the SGD optimizer:

```
optimizer = torch.optim.SGD(model.parameters(),
                           momentum=0.9, nesterov=True, lr=0.05)
```

## AdaGrad

Consider the elongated bowl problem again: gradient descent starts by quickly going down the steepest slope, which does not point straight toward the global optimum, then it very slowly goes down to the bottom of the valley. It would be nice if the algorithm could correct its direction earlier to point a bit more toward the global optimum. The *AdaGrad algorithm*<sup>19</sup> achieves this correction by scaling down the gradient vector along the steepest dimensions (see [Equation 11-7](#)).

*Equation 11-7. AdaGrad algorithm*

1.  $\mathbf{s} \leftarrow \mathbf{s} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2.  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

The first step accumulates the square of the gradients into the vector  $\mathbf{s}$  (recall that the  $\otimes$  symbol represents the element-wise multiplication). This vectorized form is equivalent to computing  $s_i \leftarrow s_i + (\partial J(\boldsymbol{\theta}) / \partial \theta_i)^2$  for each element  $s_i$  of the vector  $\mathbf{s}$ ; in other words, each  $s_i$  accumulates the squares of the partial derivative of the cost function with regard to parameter  $\theta_i$ . If the cost function is steep along the  $i^{\text{th}}$  dimension, then  $s_i$  will get larger and larger at each iteration.

The second step is almost identical to gradient descent, but with one big difference: the gradient vector is scaled down by a factor of  $\sqrt{\mathbf{s} + \varepsilon}$  (the  $\oslash$  symbol represents the element-wise division, the square root is also computed element-wise, and  $\varepsilon$  is a smoothing term to avoid division by zero, typically set to  $10^{-10}$ ). This vectorized form is equivalent to simultaneously computing  $\theta_i \leftarrow \theta_i - \eta \partial J(\boldsymbol{\theta}) / \partial \theta_i / \sqrt{s_i + \varepsilon}$  for all parameters  $\theta_i$ .

In short, this algorithm decays the learning rate, but it does so faster for steep dimensions than for dimensions with gentler slopes. This is called an *adaptive learning rate*. It helps point the resulting updates more directly toward the global optimum (see [Figure 11-8](#)). One additional benefit is that it requires much less tuning of the learning rate hyperparameter  $\eta$ .

(Steep dimension)

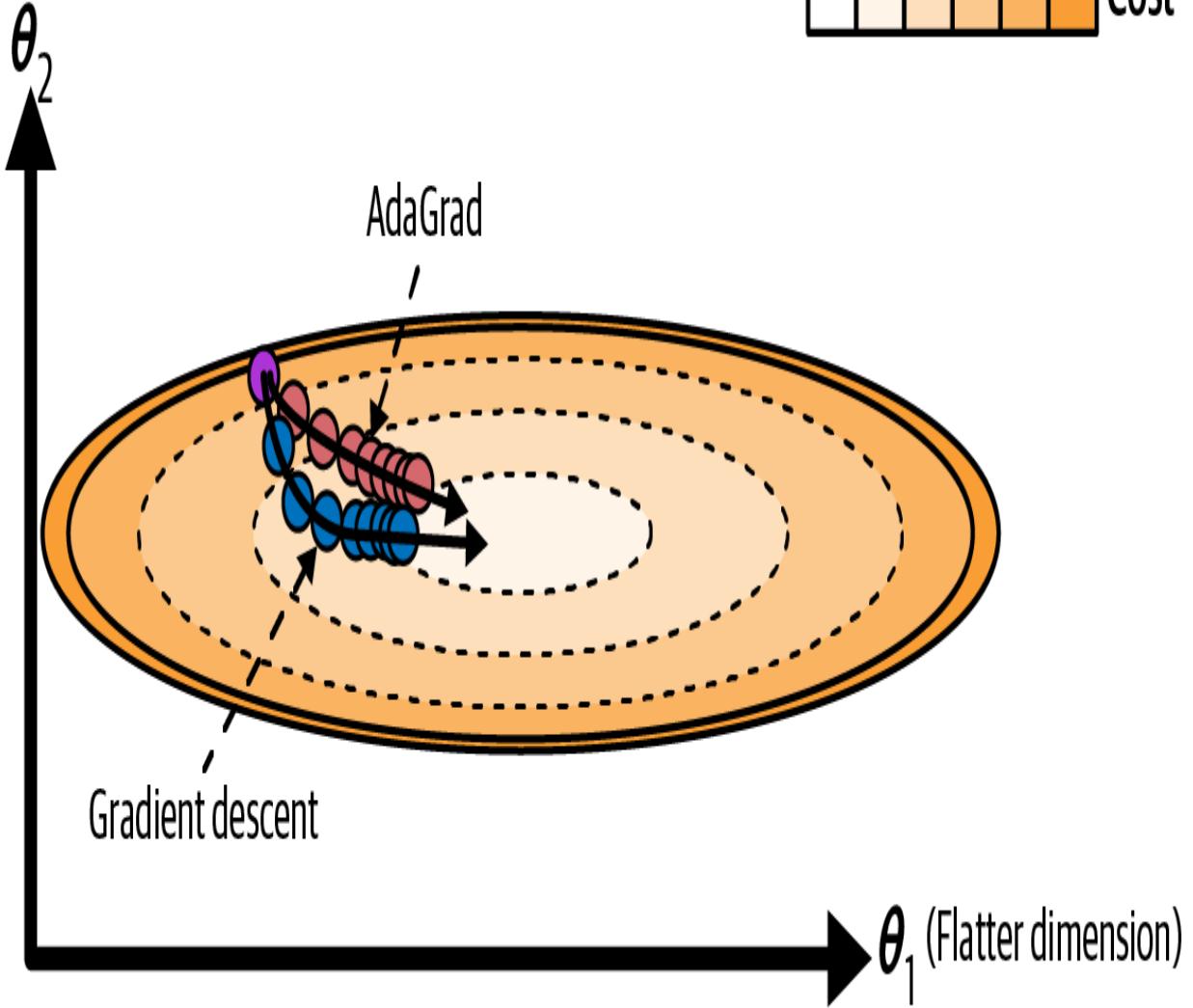
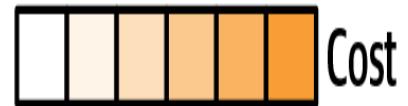


Figure 11-8. AdaGrad versus gradient descent: the former can correct its direction earlier to point to the optimum

AdaGrad frequently performs well for simple quadratic problems, but it often stops too early when training neural networks: the learning rate gets scaled down so much that the algorithm ends up stopping entirely before reaching the global optimum. So even though PyTorch has an `Adagrad` optimizer, you should not use it to train deep neural networks (it may be efficient for simpler tasks such as linear regression, though). Still, understanding AdaGrad is helpful to comprehend the other adaptive learning rate optimizers.

## RMSProp

As we've seen, AdaGrad runs the risk of slowing down a bit too fast and never converging to the global optimum. The *RMSProp* algorithm<sup>20</sup> fixes this by accumulating

only the gradients from the most recent iterations, as opposed to all the gradients since the beginning of training. It does so by using exponential decay in the first step (see [Equation 11-8](#)).

*Equation 11-8. RMSProp algorithm*

1.  $\mathbf{s} \leftarrow \alpha \mathbf{s} + (1 - \alpha) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2.  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \oslash \sqrt{\mathbf{s} + \varepsilon}$

The decay rate  $\alpha$  is typically set to 0.9. Yes, it is once again a new hyperparameter, but this default value often works well, so you may not need to tune it at all.

As you might expect, PyTorch has an `RMSprop` optimizer:

```
optimizer = torch.optim.RMSprop(model.parameters(), alpha=0.9, lr=0.05)
```

Except on very simple problems, this optimizer almost always performs much better than AdaGrad. In fact, it was the preferred optimization algorithm of many researchers until Adam optimization came around.

## Adam

*Adam*,<sup>21</sup> which stands for *adaptive moment estimation*, combines the ideas of momentum optimization and RMSProp: just like momentum optimization, it keeps track of an exponentially decaying average of past gradients; and just like RMSProp, it keeps track of an exponentially decaying average of past squared gradients (see [Equation 11-9](#)). These are estimations of the mean and (uncentered) variance of the gradients. The mean is often called the *first moment* while the variance is often called the *second moment*, hence the name of the algorithm.

*Equation 11-9. Adam algorithm*

1.  $\mathbf{m} \leftarrow \beta_1 \mathbf{m} - (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
2.  $\mathbf{s} \leftarrow \beta_2 \mathbf{s} + (1 - \beta_2) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \otimes \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$
3.  $\widehat{\mathbf{m}} \leftarrow \frac{\mathbf{m}}{1 - \beta_1^t}$
4.  $\widehat{\mathbf{s}} \leftarrow \frac{\mathbf{s}}{1 - \beta_2^t}$
5.  $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \widehat{\mathbf{m}} \oslash \sqrt{\widehat{\mathbf{s}} + \varepsilon}$

In this equation,  $t$  represents the iteration number (starting at 1).

If you just look at steps 1, 2, and 5, you will notice Adam's close similarity to both momentum optimization and RMSProp:  $\beta_1$  corresponds to  $\beta$  in momentum optimization, and  $\beta_2$  corresponds to  $\alpha$  in RMSProp. The only difference is that step 1 computes an exponentially decaying average rather than an exponentially decaying sum, but these are actually equivalent except for a constant factor (the decaying average is just  $1 - \beta_1$  times the decaying sum). Steps 3 and 4 are somewhat of a technical detail: since  $\mathbf{m}$  and  $\mathbf{s}$  are initialized at 0, they will be biased toward 0 at the beginning of training, so these two steps will help boost  $\mathbf{m}$  and  $\mathbf{s}$  at the beginning of training.

The momentum decay hyperparameter  $\beta_1$  is typically initialized to 0.9, while the scaling decay hyperparameter  $\beta_2$  is often initialized to 0.999. As earlier, the smoothing term  $\varepsilon$  is usually initialized to a tiny number such as  $10^{-8}$ . These are the default values for the `Adam` class. Here is how to create an Adam optimizer using PyTorch:

```
optimizer = torch.optim.Adam(model.parameters(), betas=(0.9, 0.999), lr=0.05)
```

Since Adam is an adaptive learning rate algorithm, like AdaGrad and RMSProp, it requires less tuning of the learning rate hyperparameter  $\eta$ . You can often use the default value  $\eta = 0.001$ , making Adam even easier to use than gradient descent.

### TIP

If you are starting to feel overwhelmed by all these different techniques and are wondering how to choose the right ones for your task, don't worry: some practical guidelines are provided at the end of this chapter.

Finally, three variants of Adam are worth mentioning: AdaMax, NAdam, and AdamW.

## AdaMax

The Adam paper also introduced AdaMax. Notice that in step 2 of [Equation 11-9](#), Adam accumulates the squares of the gradients in  $\mathbf{s}$  (with a greater weight for more recent gradients). In step 5, if we ignore  $\varepsilon$  and steps 3 and 4 (which are technical details anyway), Adam scales down the parameter updates by the square root of  $\mathbf{s}$ . In short, Adam scales down the parameter updates by the  $\ell_2$  norm of the time-decayed gradients (recall that the  $\ell_2$  norm is the square root of the sum of squares).

AdaMax replaces the  $\ell_2$  norm with the  $\ell_\infty$  norm (a fancy way of saying the max). Specifically, it replaces step 2 in [Equation 11-9](#) with  $\mathbf{s} \leftarrow \max(\beta_2 \mathbf{s}, \text{abs}(\nabla_{\theta} J(\theta)))$ , it drops step 4, and in step 5 it scales down the gradient updates by a factor of  $\mathbf{s}$ , which is the max of the absolute value of the time-decayed gradients.

In practice, this can make AdaMax more stable than Adam, but it really depends on the dataset, and in general Adam performs better. So, this is just one more optimizer you can try if you experience problems with Adam on some task.

## NAdam

NAdam optimization is Adam optimization plus the Nesterov trick, so it will often converge slightly faster than Adam. In [his report introducing this technique](#),<sup>22</sup> the researcher Timothy Dozat compares many different optimizers on various tasks and finds that NAdam generally outperforms Adam but is sometimes outperformed by RMSProp.

## AdamW

[AdamW](#)<sup>23</sup> is a variant of Adam that integrates a regularization technique called *weight decay*. Weight decay reduces the size of the model's weights at each training iteration by multiplying them by a decay factor such as 0.99. This may remind you of  $\ell_2$  regularization (introduced in [Chapter 4](#)), which also aims to keep the weights small, and indeed it can be shown mathematically that  $\ell_2$  regularization is equivalent to weight decay when using SGD. However, when using Adam or its variants,  $\ell_2$  regularization and weight decay are *not* equivalent: in practice, combining Adam with  $\ell_2$  regularization results in models that often don't generalize as well as those produced by SGD. AdamW fixes this issue by properly combining Adam with weight decay.

### WARNING

Adaptive optimization methods (including RMSProp, Adam, AdaMax, NAdam, and AdamW optimization) are often great, converging fast to a good solution. However, a [2017 paper](#)<sup>24</sup> by Ashia C. Wilson et al. showed that they can lead to solutions that generalize poorly on some datasets. So when you are disappointed by your model's performance, try using NAG instead: your dataset may just be allergic to adaptive gradients.

To use NAdam, AdaMax, or AdamW in PyTorch, replace `torch.optim.Adam` with `torch.optim.NAdam`, `torch.optim.Adamax`, or `torch.optim.AdamW`. For AdamW, you probably want to tune the `weight_decay` hyperparameter.

All the optimization techniques discussed so far only rely on the *first-order partial derivatives* (*Jacobians*, which measure the slope of the loss function along each axis). The optimization literature also contains amazing algorithms based on the *second-order partial derivatives* (the *Hessians*, which are the partial derivatives of the Jacobians, measuring how each Jacobian changes along each axis; in other words, measuring the loss function's curvature). Unfortunately, these algorithms are hard to apply directly to deep neural networks because there are  $n^2$  second-order derivatives per output (where  $n$  is the number of parameters), as opposed to just  $n$  first-order derivatives per output. Since DNNs typically have hundreds of thousands of parameters or more, the second-order optimization algorithms often don't even fit in memory, and even when they do, computing the *Hessian matrix* is just too slow.<sup>25</sup> Luckily, it is possible to use stochastic methods that can efficiently approximate second-order information. One such algorithm is `Shampoo`<sup>26</sup>, which uses accumulated gradient information to approximate the second-order terms, similar to how Adam accumulates first-order statistics. It is not included in the PyTorch library, but you can get it in the PyTorch-Optimizer library (`pip install torch_optimizer`).

## TRAINING SPARSE MODELS

All the optimization algorithms we just discussed produce dense models, meaning that most parameters will be nonzero. If you need a blazingly fast model at runtime, or if you need it to take up less memory, you may prefer to end up with a sparse model instead.

One way to achieve this is to train the model as usual, then get rid of the tiny weights (set them to zero) using `torch.nn.prune.l1_unstructured()`. Or you can get rid of entire neurons, channels, or layers, not just individual weights, using `torch.nn.prune.ln_structured()`, or other functions in the `torch.nn.prune` package.

You should generally also apply fairly strong sparsity inducing regularization during training, such as  $\ell_1$  regularization (you'll see how later in this chapter), since it pushes the optimizer to zero out as many weights as it can (see “[Lasso Regression](#)”). You can also try scaling down random weights during initialization to encourage sparsity.

[Table 11-2](#) compares all the optimizers we've discussed so far (\* is bad, \*\* is average, and \*\*\* is good).

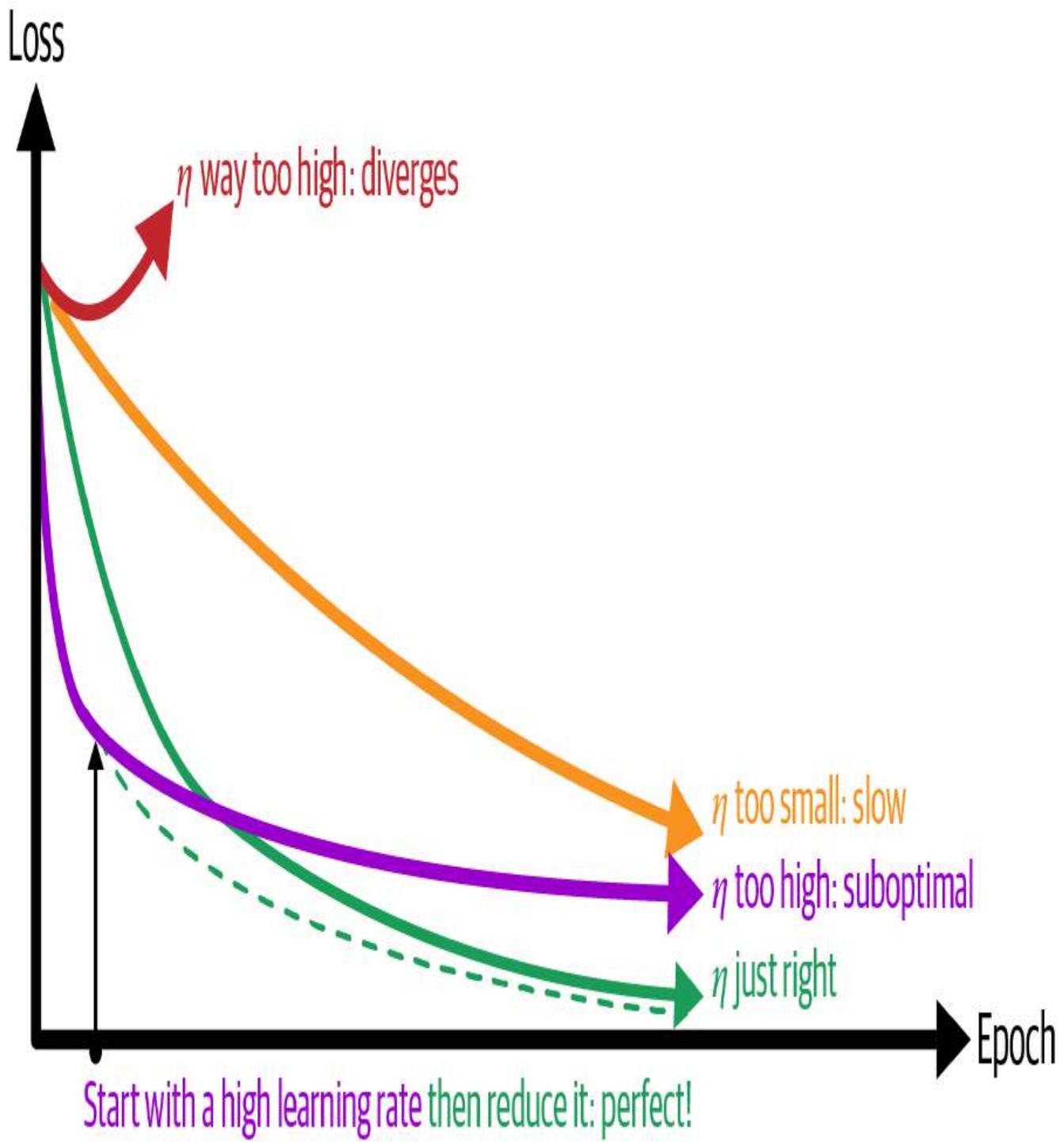
Table 11-2. Optimizer comparison

Class	Convergence speed	Convergence quality
SGD	★	★ ★ ★
SGD(momentum=...)	★ ★	★ ★ ★
SGD(momentum=..., nesterov=True)	★ ★	★ ★ ★
Adagrad	★ ★ ★	★ (stops too early)
RMSprop	★ ★ ★	★ ★ or ★ ★ ★
Adam	★ ★ ★	★ ★ or ★ ★ ★
AdaMax	★ ★ ★	★ ★ or ★ ★ ★
NAdam	★ ★ ★	★ ★ or ★ ★ ★
AdamW	★ ★ ★	★ ★ or ★ ★ ★

## Learning Rate Scheduling

Finding a good learning rate is very important. If you set it too high, training will diverge (as discussed in “[Gradient Descent](#)”). If you set it too low, then training will be painfully slow, and it may also get stuck in a local optimum and produce a suboptimal model. If you set the learning rate fairly high (but not high enough to diverge), then training will often make rapid progress at first, but it will end up dancing around the optimum towards the end of training and thereby produce a suboptimal model. If you find a really good learning rate, you can end up with an excellent model, but training will generally be a bit too slow. Luckily, you can do better than a constant learning rate. In particular, it’s a good idea to start with a fairly high learning rate and then reduce it towards the end of training (or whenever progress stops): this ensures that training

starts fast, while also allowing backprop to settle down towards the end to really fine-tune the model parameters (see [Figure 11-9](#)).



*Figure 11-9. Learning curves for various learning rates  $\eta$*

There are various other strategies to tweak the learning rate during training. These are called *learning schedules* (I briefly introduced this concept in [Chapter 4](#)). The `torch.optim.lr_scheduler` module provides several implementations of common learning schedules. For example, the `ExponentialLR` class implements *exponential*

*scheduling*, whereby the learning rate is multiplied by a constant factor `gamma` at some regular interval, typically at every epoch. As a result, after the  $n^{\text{th}}$  epoch, the learning rate will be equal to the initial learning rate times `gamma` to the power of `_n_`. This factor `gamma` is yet another hyperparameter you can tune. In general you will want to set `gamma` to a value lower than 1, but fairly close to 1 to avoid decreasing the learning rate too fast. For example, if `gamma` is set to 0.9, then after 10 epochs the learning rate will be about 35% of the initial learning rate, and after 20 epochs it will be about 12%.

The `ExponentialLR` constructor expects at least two arguments: the optimizer whose learning rate will be tweaked during training, and the factor `gamma`:

```
model = [...] # build the model
optimizer = torch.optim.SGD(model.parameters(), lr=0.05) # or any other optim.
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.9)
```

Next, you must update the training loop to call `scheduler.step()` at the end of each epoch, to tweak the optimizer's learning rate:

```
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        [...]
        # the rest of the training loop remains unchanged

    scheduler.step()
```

### TIP

If you interrupt training and you later want to resume it where you left off, you should set the `last_epoch` argument of the scheduler's constructor to the last epoch you ran (zero-indexed). The default is `-1`, which makes the scheduler start training from scratch.

Instead of decreasing the learning rate exponentially, you can use the cosine function to go from the maximum learning rate  $\eta_{\max}$  at the start of training, down to the minimum learning rate  $\eta_{\min}$  at the end. This is called *cosine annealing*. Compared to exponential scheduling, cosine annealing ensures that the learning rate remains fairly high during most of training, while getting closer to the minimum near the end (see [Figure 11-10](#)). All in all, cosine annealing generally performs better. The learning rate at epoch  $t$  (zero-indexed) is given by [Equation 11-10](#), where  $T_{\max}$  is the maximum number of epochs.

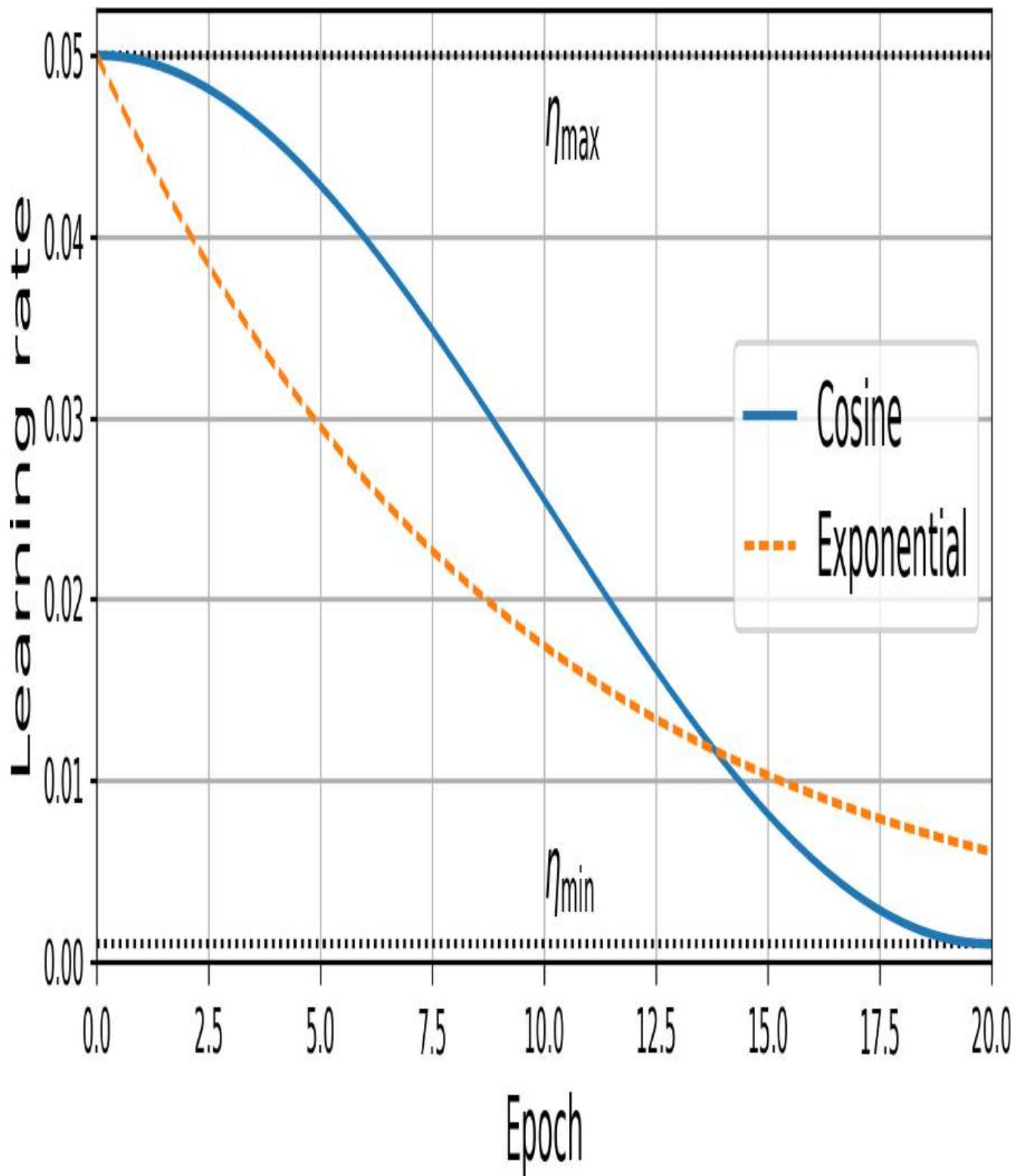


Figure 11-10. Cosine annealing learning schedule

Equation 11-10. Cosine annealing equation

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left( 1 + \cos \left( \frac{t}{T_{\max}} \pi \right) \right)$$

PyTorch includes the `CosineAnnealingLR` scheduler, which you can create as follows (`T_max` is  $T_{\max}$  and `eta_min` is  $\eta_{\min}$ ). You can then use it just like the `ExponentialLR` scheduler:

```
cosine_scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(  
    optimizer, T_max=20, eta_min=0.001)
```

One problem with cosine annealing is that you have to set two new hyperparameters:  $T_{\max}$  and  $\eta_{\min}$ , and it's not easy to know in advance how many epochs to train and when to stop decreasing the learning rate. This is why I generally prefer to use yet another technique called *performance scheduling* (or *adaptive scheduling*), which is implemented by PyTorch's `ReduceLROnPlateau` scheduler: it keeps track of a given metric during training—typically the validation loss—and if this metric stops improving for some time, it multiplies the learning rate by some factor. This scheduler has quite a few hyperparameters, but the default values work well for most of them. You may occasionally need to tweak the following (see the documentation for information on the other hyperparameters):

- `mode`: if the tracked metric must be maximized (such as the validation accuracy) then you must set the `mode` to '`max`'. The default is '`min`', which is fine if the tracked metric must be minimized (such as the validation loss).
- `patience`: the number of consecutive steps (typically epochs) to wait for improvement in the monitored metric before reducing the learning rate. It defaults to 10, which is generally fine. If each epoch is very long, then you may want to reduce this value.
- `factor`: the factor by which the learning rate will be multiplied whenever the monitored metric fails to improve for too long. It defaults to 0.1, again a reasonable default, but perhaps a bit small in some cases.

For example, let's implement performance scheduling based on the validation accuracy (i.e., which we want to maximize):

```
[...] # build the model and optimizer  
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(  
    optimizer, mode="max", patience=2, factor=0.1)
```

The training loop needs to be tweaked again, because we must evaluate the desired metric at each epoch (in this example, we are using the `evaluate_tm()` function that

we defined in [Chapter 10](#)), and we must then pass the result to the scheduler’s `step()` method:

```
metric = torchmetrics.Accuracy(task="multiclass", num_classes=10).to(device)
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        [...] # the rest of the training loop remains unchanged
        val_metric = evaluate_tm(model, valid_loader, metric).item()
        scheduler.step(val_metric)
```

So far, we have always started training with the maximum learning rate. However, this can sometimes cause gradient descent to bounce around randomly at the beginning of training, neither exploding nor making any significant progress. This typically happens with sensitive models, such as recurrent neural networks ([\[Link to Come\]](#)), or when using a very large batch size. In such cases, one solution is to “warmup” the learning rate, starting close to zero and gradually increasing the learning rate over a few epochs, up to the maximum learning rate. During this warmup phase, gradient descent has time to stabilize into a better region of the loss landscape, where it can then make quick progress using a high learning rate.

Why does this work? Well, the loss landscape sometimes resembles the Himalayas: it’s very high up and full of gigantic spikes. If you start with a high learning rate, you might jump from one mountain peak to the next for a very long time. If instead you start with a small learning rate, you will just walk down the mountain and valleys and escape the spiky mountain range altogether until you reach flatter lands. From then on, you can use a large learning rate for the rest of your journey, slowing down only towards the end.

A common way to implement learning rate warmup using PyTorch is to use a `LinearLR` scheduler to increase the learning rate linearly over a few epochs. For example, the following scheduler will increase the learning rate from 10% to 100% of the optimizer’s original learning rate over 3 epochs (i.e., 10% during the first epoch, 40% during the second epoch, 70% during the third epoch, and 100% after that):

```
warmup_scheduler = torch.optim.lr_scheduler.LinearLR(
    optimizer, start_factor=0.1, end_factor=1.0, total_iters=3)
```

If you would like more flexibility, you can write your own custom function and wrap it in a `LambdaLR` scheduler. For example, the following scheduler is equivalent to the `LinearLR` scheduler we just defined:

```
warmup_scheduler = torch.optim.lr_scheduler.LambdaLR(  
    optimizer,  
    lambda epoch: (min(epoch, 3) / 3) * (1.0 - 0.1) + 0.1)
```

You must then insert `warmup_scheduler.step()` at the beginning of each epoch, and make sure you deactivate the scheduler(s) you are using for the rest of training during the warmup phase. And that's all!

```
for epoch in range(n_epochs):  
    warmup_scheduler.step()  
    for X_batch, y_batch in train_loader:  
        [...] # the rest of the training loop is unchanged  
    if epoch >= 3: # deactivate other scheduler(s) during warmup  
        scheduler.step(val_metric)
```

In short, you pretty much always want to cool down the learning rate at the end of training, and you may also want to warm it up at the beginning if gradient descent needs a bit of help getting started. But are there any cases where you may want to tweak the learning rate in the middle of training? Well yes, there are, for example if gradient descent gets stuck in a local optimum or a high plateau. Gradient descent could remain stuck here for a long time, or even forever. Luckily, there's a way to escape this trap: just increase the learning rate for a little while.

You could spend your time staring at the learning curves during training, and manually interrupting it to tweak the learning rate when needed, but you probably have better things to do. Alternatively, you could implement a custom scheduler that monitors the validation metric—much like the `ReduceLROnPlateau` scheduler—and increases the learning rate for a while if the validation metric is stuck in a bad plateau. For this, you could subclass the `LRScheduler` base class. This is beyond the scope of this book, but you can take inspiration from the `ReduceLROnPlateau` scheduler's source code (and get a little bit of help from your favorite AI assistant).

A much simpler option is to use *cosine annealing with warm restarts*, which was introduced in a [2016 paper](#) by Ilya Loshchilov and Frank Hutter.<sup>27</sup> This schedule just repeats the cosine annealing schedule over and over again. Since the learning rate regularly shoots back up, this schedule allows gradient descent to escape local optima and plateaus automatically. The authors recommend starting with a fairly short round of cosine annealing, but then doubling  $T_{\max}$  after each round (see [Figure 11-11](#)). This allows gradient descent to do a lot of quick explorations at the start of training, while also taking the time to properly optimize the model later during training, possibly escaping a plateau or two along the way.

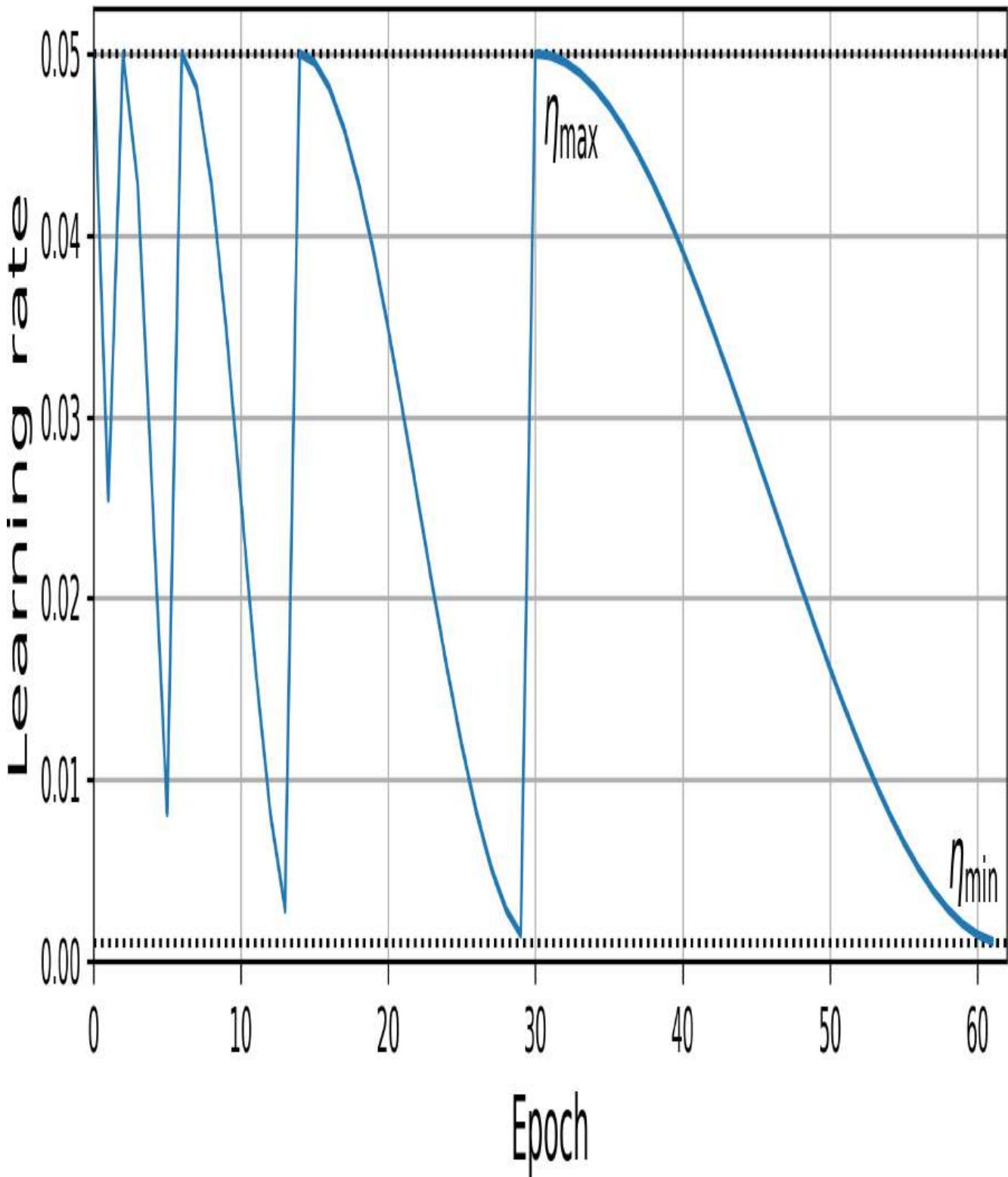


Figure 11-11. Cosine annealing with warm restarts

Conveniently, PyTorch includes a `CosineAnnealingWarmRestarts` scheduler. You must set `T_0`, which is the value of  $T_{\max}$  for the first round of cosine annealing. You may also set `T_mult` to 2 if you want to double  $T_{\max}$  at each round (the default is 1, meaning

$T_{\max}$  stays constant and all rounds have the same length). Finally, you can set `eta_min` (it defaults to 0).

```
cosine_repeat_scheduler = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(  
    optimizer, T_0=2, T_mult=2, eta_min=0.001)
```

Yet another popular learning schedule is *1cycle*, introduced in a [2018 paper](#) by Leslie Smith.<sup>28</sup> It starts by warming up the learning rate, starting at  $\eta_0$  and growing linearly up to  $\eta_1$  halfway through training. Then it decreases the learning rate linearly down to  $\eta_0$  again during the second half of training, finishing the last few epochs by dropping the rate down by several orders of magnitude (still linearly). The maximum learning rate  $\eta_1$  is chosen using the same approach we used to find the optimal learning rate, and the initial learning rate  $\eta_0$  is usually 10 times lower. When using a momentum, we start with a high momentum first (e.g., 0.95), then drop it down to a lower momentum during the first half of training (e.g., down to 0.85, linearly), and then bring it back up to the maximum value (e.g., 0.95) during the second half of training, finishing the last few epochs with that maximum value. Smith did many experiments showing that this approach was often able to speed up training considerably and reach better performance. For example, on the popular CIFAR10 image dataset, this approach reached 91.9% validation accuracy in just 100 epochs, compared to 90.3% accuracy in 800 epochs through a standard approach (using the same neural network architecture). This feat was dubbed *super-convergence*. PyTorch implements this schedule in the `OneCycleLR` scheduler.

### TIP

If you are not sure which learning schedule to use, 1cycle can be a good default, but I tend to have more luck with performance scheduling. If you run into instabilities at the start of training, try adding learning rate warmup. And if training gets stuck on plateaus, try cosine annealing with warm restarts.

We have now covered the most popular learning schedules, but PyTorch offers a few extra schedulers (e.g., a polynomial scheduler, a cyclic scheduler, a scheduler that make it easy to chain other schedulers, and a few more), so make sure to check out the documentation.

Now let's move on to one final topic before we complete this chapter on deep learning training techniques: regularization. Deep learning is highly prone to overfitting, so regularization is key!

# Avoiding Overfitting Through Regularization

*With four parameters I can fit an elephant and with five I can make him wiggle his trunk.*

—John von Neumann, cited by Enrico Fermi in *Nature* 427

With thousands of parameters, you can fit the whole zoo. Deep neural networks typically have tens of thousands of parameters, sometimes even millions or billions. This gives them an incredible amount of freedom and means they can fit a huge variety of complex datasets. But this great flexibility also makes the network prone to overfitting the training set. Regularization is often needed to prevent this.

We already implemented one of the best regularization techniques in [Chapter 4](#): early stopping. Moreover, even though batch-norm and layer-norm were designed to solve the unstable gradients problems, they also act like pretty good regularizers. In this section we will examine other popular regularization techniques for neural networks:  $\ell_1$  and  $\ell_2$  regularization, dropout, MC dropout, and max-norm regularization.

## $\ell_1$ and $\ell_2$ Regularization

Just like you did in [Chapter 4](#) for simple linear models, you can use  $\ell_2$  regularization to constrain a neural network's connection weights, and/or  $\ell_1$  regularization if you want a sparse model (with many weights equal to 0). As we saw earlier (when discussing the AdamW optimizer),  $\ell_2$  regularization is mathematically equivalent to weight decay when using an SGD optimizer (with or without momentum), so if that's the case you can implement  $\ell_2$  regularization by simply setting the optimizer's `weight_decay` argument. For example, here is how to apply  $\ell_2$  regularization to the connection weights of a PyTorch model trained using SGD, with a regularization factor of  $10^{-4}$ :

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.05, weight_decay=1e-4)
[...] # use the optimizer normally during training
```

If instead you are using an Adam optimizer, you should switch to AdamW and set the `weight_decay` argument. This is not exactly equivalent to  $\ell_2$  regularization, but as we saw earlier it's pretty close and it works better.

Note that weight decay is applied to every model parameter, including bias terms, and even parameters of batch-norm and layer-norm layers. Generally that's not a big deal, but penalizing these parameters does not contribute much to regularization and it may

sometimes negatively impact training performance. So how can we apply weight decay to some model parameters and not others? One approach is to implement  $\ell_2$  regularization manually, without relying on the optimizer's weight decay feature. For this, you must tweak the training loop to manually compute the  $\ell_2$  loss based only on the parameters you want, and add this  $\ell_2$  loss to the main loss:

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.05)
params_to_regularize = [
    param for name, param in model.named_parameters()
    if not "bias" in name and not "bn" in name]
for epoch in range(n_epochs):
    for X_batch, y_batch in train_loader:
        [...] # the rest of the training loop is unchanged
        main_loss = loss_fn(y_pred, y_batch)
        l2_loss = sum(param.pow(2.0).sum() for param in params_to_regularize)
        loss = main_loss + 1e-4 * l2_loss
        [...]
```

Another approach is to use PyTorch's *parameter groups* feature, which lets the optimizer apply different hyperparameters to different groups of model parameters. So far, we have always created optimizers by passing them the full list of model parameters: PyTorch automatically put them all in a single parameter group, sharing the same hyperparameters. Instead, we can pass a list of dictionaries to the optimizer, each with a "params" entry containing a list of parameters, and (optionally) some hyperparameter key/value pairs specific to this group of parameters. The group-specific hyperparameters take precedence over the optimizer's global hyperparameters. For example, let's create an optimizer with two parameter groups: the first group will contain all the parameters we want to regularize and it will use weight decay, while the second group will contain all the bias terms and BN parameters, and it will not use weight decay at all.

```
params_bias_and_bn = [
    param for name, param in model.named_parameters()
    if "bias" in name or "bn" in name]
optimizer = torch.optim.SGD([
    {"params": params_to_regularize, "weight_decay": 1e-4},
    {"params": params_bias_and_bn},
], lr=0.05)
[...] # use the optimizer normally during training
```

## TIP

Parameter groups also allow you to apply different learning rates to different parts of your model. This is most common for transfer learning, when you want new layers to be updated faster than reused ones.

Now how about  $\ell_1$  regularization? Well unfortunately PyTorch does not provide any helper for this, so you need to implement it manually, much like we did for  $\ell_2$  regularization. This means tweaking the training loop to compute the  $\ell_1$  loss and adding it to the main loss:

```
l1_loss = sum(param.abs().sum() for param in params_to_regularize)
loss = main_loss + 1e-4 * l1_loss
```

That's all there is to it! Now let's move on to Dropout, which is one of the most popular regularization techniques for deep neural networks.

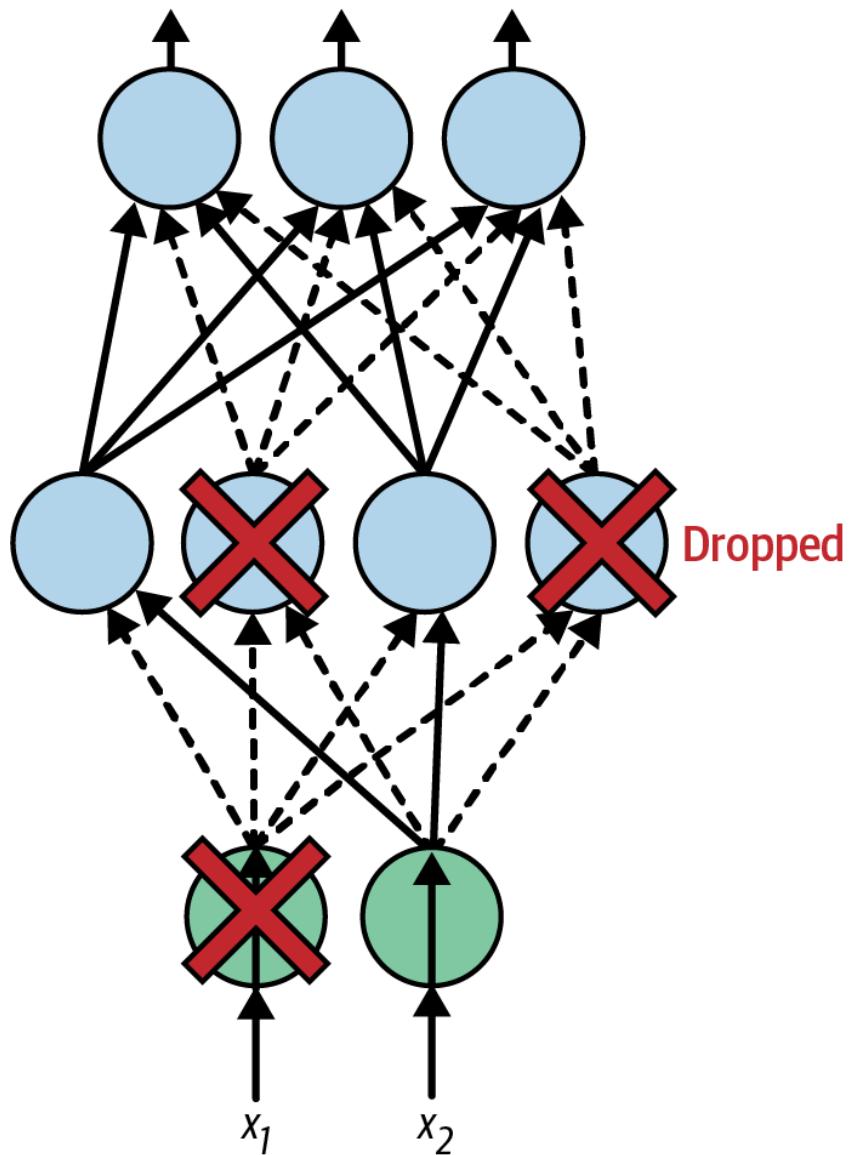
## Dropout

Dropout was proposed in a paper<sup>29</sup> by Geoffrey Hinton et al. in 2012 and further detailed in a 2014 paper<sup>30</sup> by Nitish Srivastava et al., and it has proven to be highly successful: many state-of-the-art neural networks use dropout, as it gives them a 1%–2% accuracy boost. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons, but always excluding the output neurons) has a probability  $p$  of being temporarily “dropped out”, meaning it will be entirely ignored during this training step, but it may be active during the next step (see Figure 11-12). The hyperparameter  $p$  is called the *dropout rate*, and it is typically set between 10% and 50%: closer to 20%–30% in recurrent neural nets (see [Link to Come]), and closer to 40%–50% in convolutional neural networks (see [Link to Come]). After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss shortly).

It's surprising at first that this destructive technique works at all. Would a company perform better if its employees were told to toss a coin every morning to decide whether or not to go to work? Well, who knows; perhaps it would! The company would be forced to adapt its organization; it could not rely on any single person to work the coffee machine or perform any other critical tasks, so this expertise would have to be

spread across several people. Employees would have to learn to cooperate with many of their coworkers, not just a handful of them. The company would become much more resilient. If one person quit, it wouldn't make much of a difference. It's unclear whether this idea would actually work for companies, but it certainly does for neural networks. Neurons trained with dropout cannot co-adapt with their neighboring neurons; they have to be as useful as possible on their own. They also cannot rely excessively on just a few input neurons; they must pay attention to each of their input neurons. They end up being less sensitive to slight changes in the inputs. In the end, you get a more robust network that generalizes better.



*Figure 11-12. With dropout regularization, at each training iteration a random subset of all neurons in one or more layers—except the output layer—are “dropped out”; these neurons output 0 at this iteration (represented by the dashed arrows)*

Another way to understand the power of dropout is to realize that a unique neural network is generated at each training step. Since each neuron can be either present or

absent, there are a total of  $2^N$  possible networks (where  $N$  is the total number of droppable neurons). This is such a huge number that it is virtually impossible for the same neural network to be sampled twice. Once you have run 10,000 training steps, you have essentially trained 10,000 different neural networks, each with just one training instance. These neural networks are obviously not independent because they share many of their weights, but they are nevertheless all different. The resulting neural network can be seen as an averaging ensemble of all these smaller neural networks.

## TIP

Higher layers, which learn more complex feature combinations, benefit more from dropout because they are more prone to overfitting. So you can usually apply dropout only to the neurons of the top hidden layers (e.g., 1 to 3 hidden layers). However, you should avoid dropping the output neurons, as this would be like changing the task during training: it wouldn't help.

There is one small but important technical detail. Suppose  $p = 75\%$ : on average only 25% of all neurons are active at each step during training. This means that after training, each neuron receives four times more inputs than during training, on average. This discrepancy is so large that the model is unlikely to work well. To avoid this issue, a simple solution is to multiply the inputs by four during training, which is the same as dividing them by 25%. More generally, we need to divide the inputs by the *keep probability* ( $1 - p$ ) during training.

To implement dropout using PyTorch, you can use the `nn.Dropout` layer. During training, it randomly drops some inputs (setting them to 0) and divides the remaining inputs by the keep probability. After training, it does nothing at all; it just passes the inputs to the next layer. The following code applies dropout regularization before every `nn.Linear` layer, using a dropout rate of 0.2:

```
model = nn.Sequential(
    nn.Flatten(),
    nn.Dropout(p=0.2), nn.Linear(1 * 28 * 28, 100), nn.ReLU(),
    nn.Dropout(p=0.2), nn.Linear(100, 100), nn.ReLU(),
    nn.Dropout(p=0.2), nn.Linear(100, 100), nn.ReLU(),
    nn.Dropout(p=0.2), nn.Linear(100, 10)
).to(device)
```

## WARNING

Since dropout is only active during training, comparing the training loss and the validation loss can be misleading. In particular, a model may be overfitting the training set and yet have similar training and validation losses. So, make sure to evaluate the training loss without dropout (e.g., after training).

If you observe that the model is overfitting, you can increase the dropout rate. Conversely, you should try decreasing the dropout rate if the model underfits the training set. It can also help to increase the dropout rate for large layers, and reduce it for small ones. Moreover, many state-of-the-art architectures only apply dropout to the last few hidden layers, so you may want to try this if full dropout is too strong.

Dropout does tend to significantly slow down convergence, but it often results in a better model when tuned properly. So, it is generally well worth the extra time and effort, especially for large models.

## WARNING

When using dropout, it's important to switch to training mode during training, and to evaluation mode during evaluation (just like for batch norm).

## TIP

If you want to regularize a self-normalizing network based on the SELU activation function (as discussed earlier), you should use *alpha dropout*: this is a variant of dropout that preserves the mean and standard deviation of its inputs. It was introduced in the same paper as SELU, as regular dropout would break self-normalization. PyTorch implements it in the `nn.AlphaDropout` layer.

## Monte Carlo (MC) Dropout

In 2016, a [paper<sup>31</sup>](#) by Yarin Gal and Zoubin Ghahramani added a few more good reasons to use dropout:

- First, the paper established a profound connection between dropout networks (i.e., neural networks containing `Dropout` layers) and approximate Bayesian inference,<sup>32</sup> giving dropout a solid mathematical justification.
- Second, the authors introduced a powerful technique called *MC dropout*, which can boost the performance of any trained dropout model without having

to retrain it or even modify it at all. It also provides a much better measure of the model’s uncertainty, and it can be implemented in just a few lines of code.

This description of MC dropout sounds like some “one weird trick” clickbait, so let me explain: it is just like regular dropout, except it is active not only during training, but also during evaluation. This means that the predictions are always a bit random (hence the name Monte Carlo). But instead of making a single prediction, you make many predictions and average them out. It turns out that this produces better predictions than the original model.

Below is a full implementation of MC dropout, using the model we trained in the previous section to make predictions for a batch of images:

```
model.eval()
for module in model.modules():
    if isinstance(module, nn.Dropout):
        module.train()

X_new = [...] # some new images, e.g., the first 3 images of the test set
X_new = X_new.to(device)

torch.manual_seed(42)
with torch.no_grad():
    X_new_repeated = X_new.repeat_interleave(100, dim=0)
    y_logits_all = model(X_new_repeated).reshape(3, 100, 10)
    y_probas_all = torch.nn.functional.softmax(y_logits_all, dim=-1)
    y_probas = y_probas_all.mean(dim=1)
```

Let’s go through this code:

- First, we switch the model to evaluation mode as we always do before making predictions, but this time we immediately switch all the dropout layers back to training mode, so they will behave just like during training (i.e., randomly dropping out some of their inputs). In other words, we convert the dropout layers to MC dropout layers.
- Next we load a new batch of images  $X_{\text{new}}$ , and we move it to the GPU. In this example, let’s assume  $X_{\text{new}}$  contains 3 images.
- We then use the `repeat_interleave()` method to create a batch containing 100 copies of each image in  $X_{\text{new}}$ . The images are repeated along the first dimension (`dim=0`) so  $X_{\text{new\_repeated}}$  has a shape of [300, 1, 28, 28].

- Next, we pass this big batch to the model, which predicts 10 logits per image, as usual. This tensor's shape is [300, 10], but we reshape it to [3, 100, 10] to group the predictions for each image. Remember that the dropout layers are active, which means that there's some variability across the predictions, even for copies of the same image.
- Then, we convert these logits to estimated probabilities using the softmax function.
- Lastly, we compute the mean over the second dimension (`dim=1`) to get the average estimated probability for each class and each image, across all 100 predictions. The result is a tensor of shape [3, 10]. These are our final predictions:<sup>33</sup>

```
>>> y_probas.cpu().numpy().round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.03, 0. , 0.97],
       [0.99, 0. , 0. , 0. , 0. , 0. , 0.01, 0. , 0. , 0. ],
       [0.5 , 0.03, 0.03, 0.22, 0.02, 0. , 0.2 , 0. , 0. , 0. ]],
      dtype=float32)
```

### WARNING

Rather than converting the logits to probabilities and then computing the mean probabilities, you may be tempted to do the reverse: first average over the logits and *then* convert the mean logits to probabilities. This is faster but it does not properly reflect the model's uncertainty, so it tends to produce overconfident models.

MC dropout tends to improve the reliability of the model's probability estimates. This means that it's less likely to be confidently wrong, making it safer (you don't want a self-driving car confidently ignoring a stop sign). It's also useful when you're interested in the top  $k$  classes, not just the most likely. Additionally, you can take a look at the **standard deviation of each class probability**:

```
>>> y_std = y_probas_all.std(dim=1)
>>> y_std.cpu().numpy().round(2)
array([[0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.06, 0. , 0.06],
       [0.02, 0. , 0. , 0. , 0. , 0. , 0.02, 0. , 0. , 0. ],
       [0.16, 0.03, 0.04, 0.15, 0.03, 0. , 0.09, 0. , 0. , 0. ]],
      dtype=float32)
```

There's a standard deviation of 0.06 for the probability estimate of class 9 (ankle boot) for the first image. This adds a grain of salt to the estimated probability of 97% for this class: in fact, the model is really saying “mmh, I'm guessing over 90%”. If you were building a risk-sensitive system (e.g., a medical or financial system), you may want to consider only the predictions with both a high estimated probability *and* a low standard deviation.

## NOTE

The number of Monte Carlo samples you use (100 in this example) is a hyperparameter you can tweak. The higher it is, the more accurate the predictions and their uncertainty estimates are. But also the slower the predictions are. Moreover, above a certain number of samples, you will notice little improvement. Your job is to find the right trade-off between latency, throughput, and accuracy, depending on your application.

If you want to train an MC dropout model from scratch rather than reuse an existing dropout model, you should probably use a custom `McDropout` module rather than using `nn.Dropout` and hacking around with `train()` and `eval()`, as this is a bit brittle (e.g., it won't play nicely with the evaluation function). Here is a 3-line implementation:

```
class McDropout(nn.Dropout):
    def forward(self, input):
        return F.dropout(input, self.p, training=True)
```

In short, MC dropout is a great technique that boosts dropout models and provides better uncertainty estimates. And of course, since it is just regular dropout during training, it also acts like a regularizer.

## Max-Norm Regularization

Another fairly popular regularization technique for neural networks is called *max-norm regularization*: for each neuron, it constrains the weights  $\mathbf{w}$  of the incoming connections such that  $\|\mathbf{w}\|_2 \leq r$ , where  $r$  is the max-norm hyperparameter and  $\|\cdot\|_2$  is the  $\ell_2$  norm.

Reducing  $r$  increases the amount of regularization and helps reduce overfitting. Max-norm regularization can also help alleviate the unstable gradients problems (if you are not using batch-norm or layer-norm).

Rather than adding a regularization loss term to the overall loss function, max-norm regularization is typically implemented by computing  $\|\mathbf{w}\|_2$  after each training step and

rescaling  $\mathbf{w}$  if needed ( $\mathbf{w} \leftarrow \mathbf{w} r / \|\mathbf{w}\|_2$ ). Here's a common way to implement this in PyTorch:

```
def apply_max_norm(model, max_norm=2, epsilon=1e-8, dim=1):
    with torch.no_grad():
        for name, param in model.named_parameters():
            if 'bias' not in name:
                actual_norm = param.norm(p=2, dim=dim, keepdim=True)
                target_norm = torch.clamp(actual_norm, 0, max_norm)
                param *= target_norm / (epsilon + actual_norm)
```

This function iterates through all of the model's weight matrices (i.e., all parameters except for the bias terms), and for each one of them it uses the `norm()` method to compute the  $\ell_2$  norm of each row (`dim=1`). A `nn.Linear` layer has weights of shape [*number of neurons, number of inputs*], so using `dim=1` means that we will get one norm per neuron, as desired. Then the function uses `torch.clamp()` to compute the target norm for each neuron's weights: this creates a copy of the `actual_norm` tensor, except that all values greater than `max_norm` are replaced by `max_norm` (this corresponds to  $r$  in the previous equation). Lastly, we rescale the weight matrix so that each column ends up with the target norm. Note that the smoothing term `epsilon` is used to avoid division by zero in case some columns have a norm equal to zero.

Next, all you need to do is call `apply_max_norm(model)` in the training loop, right after calling the optimizer's `step()` method. And of course you probably want to fine-tune the `max_norm` hyperparameter.

### TIP

When using max-norm with layers other than `nn.Linear`, you may need to tweak the `dim` argument. For example, when using convolutional layers (see [Link to Come]), you generally want to set `dim=[1, 2, 3]` to limit the norm of each convolutional kernel.

## Summary and Practical Guidelines

In this chapter we have covered a wide range of techniques, and you may be wondering which ones you should use. This depends on the task, and there is no clear consensus yet, but I have found the configuration in [Table 11-3](#) to work fine in most cases, without requiring much hyperparameter tuning. That said, please do not consider these defaults as hard rules!

*Table 11-3. Default DNN configuration*

<b>Hyperparameter</b>	<b>Default value</b>
Kernel initializer	He initialization
Activation function	ReLU if shallow; Swish if deep
Normalization	None if shallow; batch-norm or layer-norm if deep
Regularization	Early stopping; weight decay if needed
Optimizer	Nesterov accelerated gradients or AdamW
Learning rate schedule	Performance scheduling or 1cycle

If the network is a simple stack of dense layers, then it can self-normalize, and you should use the configuration in [Table 11-4](#) instead (don't forget to normalize the input features!).

*Table 11-4. DNN configuration for a self-normalizing net*

Hyperparameter	Default value
Kernel initializer	LeCun initialization
Activation function	SELU
Normalization	None (self-normalization)
Regularization	Alpha dropout if needed
Optimizer	Nesterov accelerated gradients
Learning rate schedule	Performance scheduling or 1cycle

You should also try to reuse parts of a pretrained neural network if you can find one that solves a similar problem, or use unsupervised pretraining if you have a lot of unlabeled data, or use pretraining on an auxiliary task if you have a lot of labeled data for a similar task.

While the previous guidelines should cover most cases, there are some exceptions:

- If you need a sparse model, you can use  $\ell_1$  regularization. You can also try zeroing out the smallest weights after training (e.g., using the `torch.nn.prune.l1_unstructured()` function). This will break self-normalization, so you should use the default configuration in this case.
- If you need a low-latency model (one that performs lightning-fast predictions), you may need to use fewer layers, use a fast activation function such as `nn.ReLU`, `nn.LeakyReLU`, or `nn.Hardswish`, and fold the batch-norm and layer-norm layers into the previous layers after training. Having a sparse model will also help. Finally, you may want to reduce the float precision from 32 bits to 16 or even 8 bits (see [Link to Come]).
- If you are building a risk-sensitive application, or inference latency is not very important in your application, you can use MC dropout to boost performance

and get more reliable probability estimates, along with uncertainty estimates.

Over the last three chapters, we have learned what artificial neural nets are, how to build and train them using Scikit-Learn and PyTorch, and a variety of techniques that make it possible to train deep and complex nets. In the next chapter, all of this will come together as we dive into one of the most important applications of deep learning: computer vision.

## Exercises

1. What is the problem that Glorot initialization and He initialization aim to fix?
2. Is it OK to initialize all the weights to the same value as long as that value is selected randomly using He initialization?
3. Is it OK to initialize the bias terms to 0?
4. In which cases would you want to use each of the activation functions we discussed in this chapter?
5. What may happen if you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using an SGD optimizer?
6. Name three ways you can produce a sparse model.
7. Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)? What about MC dropout?
8. Practice training a deep neural network on the CIFAR10 image dataset:
  - a. Load CIFAR10 just like you loaded the FashionMNIST dataset in [Chapter 10](#), but using `torchvision.datasets.CIFAR10` instead of `FashionMNIST`. The dataset is composed of 60,000  $32 \times 32$ -pixel color images (50,000 for training, 10,000 for testing) with 10 classes.
  - b. Build a DNN with 20 hidden layers of 100 neurons each (that's too many, but it's the point of this exercise). Use He initialization and the Swish activation function (using `nn.SiLU`). Since this is a classification task, you will need an output layer with one neuron per class.

- c. Using NAdam optimization and early stopping, train the network on the CIFAR10 dataset. Remember to search for the right learning rate each time you change the model’s architecture or hyperparameters.
- d. Now try adding batch norm and compare the learning curves: is it converging faster than before? Does it produce a better model? How does it affect training speed?
- e. Try replacing batch norm with SELU, and make the necessary adjustments to ensure the network self-normalizes (i.e., standardize the input features, use LeCun normal initialization, make sure the DNN contains only a sequence of dense layers, etc.).
- f. Try regularizing the model with alpha dropout. Then, without retraining your model, see if you can achieve better accuracy using MC dropout.
- g. Retrain your model using 1cycle scheduling and see if it improves training speed and model accuracy.

Solutions to these exercises are available at the end of this chapter’s notebook, at <https://homl.info/colab-p>.

---

<sup>1</sup> Xavier Glorot and Yoshua Bengio, “Understanding the Difficulty of Training Deep Feedforward Neural Networks”, *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics* (2010): 249–256.

<sup>2</sup> Here’s an analogy: if you set a microphone amplifier’s volume knob too close to zero, people won’t hear your voice, but if you set it too close to the max, your voice will be saturated and people won’t understand what you are saying. Now imagine a chain of such amplifiers: they all need to be set properly in order for your voice to come out loud and clear at the end of the chain. Your voice has to come out of each amplifier at the same amplitude as it came in.

<sup>3</sup> Kaiming He et al., “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” *Proceedings of the 2015 IEEE International Conference on Computer Vision* (2015): 1026–1034.

<sup>4</sup> There is a PyTorch issue (#18182) open since 2019 to update the weight initialization to use the current best practices.

<sup>5</sup> Andrew Saxe et al., “Exact solutions to the nonlinear dynamics of learning in deep linear neural networks,” ICLR (2014).

<sup>6</sup> A dead neuron may come back to life if its inputs evolve over time and eventually return within a range where the ReLU activation function gets a positive input again. For example, this may happen if gradient descent

tweaks the neurons in the layers below the dead neuron.

- 7 Bing Xu et al., “Empirical Evaluation of Rectified Activations in Convolutional Network,” arXiv preprint arXiv:1505.00853 (2015).
- 8 Djork-Arné Clevert et al., “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” *Proceedings of the International Conference on Learning Representations*, arXiv preprint (2015).
- 9 Günter Klambauer et al., “Self-Normalizing Neural Networks”, *Proceedings of the 31st International Conference on Neural Information Processing Systems* (2017): 972–981.
- 10 Dan Hendrycks and Kevin Gimpel, “Gaussian Error Linear Units (GELUs)”, arXiv preprint arXiv:1606.08415 (2016).
- 11 A function is convex if the line segment between any two points on the curve never lies below the curve. A monotonic function only increases, or only decreases.
- 12 Prajit Ramachandran et al., “Searching for Activation Functions”, arXiv preprint arXiv:1710.05941 (2017).
- 13 Diganta Misra, “Mish: A Self Regularized Non-Monotonic Activation Function”, arXiv preprint arXiv:1908.08681 (2019).
- 14 Sergey Ioffe and Christian Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”, *Proceedings of the 32nd International Conference on Machine Learning* (2015): 448–456.
- 15 Jimmy Lei Ba et al., “Layer Normalization”, arXiv preprint arXiv:1607.06450 (2016).
- 16 Razvan Pascanu et al., “On the Difficulty of Training Recurrent Neural Networks”, *Proceedings of the 30th International Conference on Machine Learning* (2013): 1310–1318.
- 17 Boris T. Polyak, “Some Methods of Speeding Up the Convergence of Iteration Methods”, *USSR Computational Mathematics and Mathematical Physics* 4, no. 5 (1964): 1–17.
- 18 Yurii Nesterov, “A Method for Unconstrained Convex Minimization Problem with the Rate of Convergence  $O(1/k^2)$ ,” *Doklady AN USSR* 269 (1983): 543–547.
- 19 John Duchi et al., “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, *Journal of Machine Learning Research* 12 (2011): 2121–2159.
- 20 This algorithm was created by Geoffrey Hinton and Tijmen Tieleman in 2012 and presented by Geoffrey Hinton in his Coursera class on neural networks (slides: <https://homl.info/57>; video: <https://homl.info/58>). Amusingly, since the authors did not write a paper to describe the algorithm, researchers often cite “slide 29 in lecture 6e” in their papers.
- 21 Diederik P. Kingma and Jimmy Ba, “Adam: A Method for Stochastic Optimization”, arXiv preprint arXiv:1412.6980 (2014).
- 22 Timothy Dozat, “Incorporating Nesterov Momentum into Adam” (2016).
- 23 Ilya Loshchilov, and Frank Hutter, “Decoupled Weight Decay Regularization”, arXiv preprint arXiv:1711.05101 (2017).

- <sup>24</sup> Ashia C. Wilson et al., “The Marginal Value of Adaptive Gradient Methods in Machine Learning”, *Advances in Neural Information Processing Systems* 30 (2017): 4148–4158.
- <sup>25</sup> The *Jacobian matrix* contains all the first-order partial derivatives of a function with multiple parameters and multiple outputs: one column per parameter, and one row per output. When training a neural net with gradient descent, there’s a single output—the loss—so the matrix contains a single row, and there’s one column per model parameter, so it’s a  $1 \times n$  matrix. The *Hessian matrix* contains all the second-order derivatives of a single-output function with multiple parameters: for each model parameter it contains one row and one column, so it’s an  $n \times n$  matrix. The informal names *Jacobians* and *Hessians* refer to the elements of these matrices.
- <sup>26</sup> V. Gupta et al., “Shampoo: Preconditioned Stochastic Tensor Optimization”, arXiv preprint arXiv:1802.09568 (2018).
- <sup>27</sup> Ilya Loshchilov and Frank Hutter, “SGDR: Stochastic Gradient Descent With Warm Restarts”, arXiv preprint arXiv:1608.03983 (2016).
- <sup>28</sup> Leslie N. Smith, “A Disciplined Approach to Neural Network Hyper-Parameters: Part 1—Learning Rate, Batch Size, Momentum, and Weight Decay”, arXiv preprint arXiv:1803.09820 (2018).
- <sup>29</sup> Geoffrey E. Hinton et al., “Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors”, arXiv preprint arXiv:1207.0580 (2012).
- <sup>30</sup> Nitish Srivastava et al., “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, *Journal of Machine Learning Research* 15 (2014): 1929–1958.
- <sup>31</sup> Yarin Gal and Zoubin Ghahramani, “Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning”, *Proceedings of the 33rd International Conference on Machine Learning* (2016): 1050–1059.
- <sup>32</sup> Specifically, they show that training a dropout network is mathematically equivalent to approximate Bayesian inference in a specific type of probabilistic model called a *deep Gaussian process*.
- <sup>33</sup> We must move the data to the CPU using the `cpu()` method before converting it to a NumPy array. We need a NumPy array to call the `round()` method with a number of decimals, since the tensor’s `round()` method sadly does not have this feature.

## About the Author

**Aurélien Géron** is a machine learning consultant and lecturer. A former Googler, he led YouTube's video classification team. He's been a founder of and CTO at a few different companies: Wifirst, a leading wireless ISP in France; Polyconseil, a consulting firm focused on telecoms, media, and strategy; and Geron AI, a consulting firm focused on machine learning.

Before all that Aurélien worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He also published a few technical books (on C++, WiFi, and internet architectures) and lectured in several universities.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1,023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.