O'REILLY®

Hands-On Machine Learning with Scikit-Learn and PyTorch

Concepts, Tools, and Techniques to Build Intelligent Systems

Aurélien Géron

"This book is an excellent starting point for beginners looking to understand the essential history and foundational concepts of machine learning. With well-structured code sections and practical examples, it takes readers from the basics to cutting-edge machine learning and deep learning techniques, leveraging PyTorch and Scikit-Learn for hands-on implementation."

Louis-François Bouchard, educator and cofounder and CTO at Towards AI

"Géron strikes the sweet spot: practical Scikit-Learn and PyTorch implementations that teach concepts, balanced with theory that clarifies rather than overwhelms. This is the book I recommend for getting started in ML."

Ulf Bissbort, cofounder and CTO at ZefHub

Hands-On Machine Learning with Scikit-Learn and PyTorch

The potential of machine learning today is extraordinary, yet many aspiring developers and tech professionals find themselves daunted by its complexity. Whether you're looking to enhance your skill set and apply machine learning to real-world projects or are simply curious about how AI systems function, this book is your jumping-off place.

With an approachable yet deeply informative style, author Aurélien Géron delivers the ultimate introductory guide to machine learning and deep learning. With a focus on clear explanations and real-world examples, the book takes you through cutting-edge tools like Scikit-Learn, PyTorch, and Hugging Face libraries—from basic regression techniques to advanced neural networks. Whether you're a student, professional, or hobbyist, you'll gain the skills to build intelligent systems.

- **Understand ML fundamentals, including concepts like overfitting and hyperparameter tuning**
- **Complete an end-to-end ML project using Scikit-Learn, covering everything from data exploration to model evaluation**
- **Learn techniques for unsupervised learning, such as clustering and anomaly detection**
- **Build advanced architectures like transformer-based chatbots and diffusion models with PyTorch**
- **Harness the power of pretrained models—including LLMs—and learn to fine-tune and accelerate them**
- **Train autonomous agents using reinforcement learning**

Aurélien Géron is a machine learning consultant and former YouTube video classification lead at Google. He cofounded tech firms Wifirst and Polyconseil, authored technical books, and has a diverse background in finance, defense, and healthcare.

DATA

US $89.99 CAN $112.99
ISBN: 979-8-341-60798-9

9 798341 607989 58999

O'REILLY®

Praise for *Hands-On Machine Learning with Scikit-Learn and PyTorch*

This book is an excellent starting point for beginners looking to understand the essential history and foundational concepts of machine learning. With well-structured code sections and practical examples, it takes readers from the basics to cutting-edge machine learning and deep learning techniques, leveraging PyTorch and Scikit-Learn for hands-on implementation.

—Louis-François Bouchard, educator and cofounder and CTO at Towards AI

Géron strikes the sweet spot: practical Scikit-Learn and PyTorch implementations that teach concepts, balanced with theory that clarifies rather than overwhelms. From first principles to state-of-the-art methods, this hands-on approach gets you productive quickly. This is the book I recommend for getting started in ML.

—Ulf Bissbort, cofounder and CTO at ZefHub

This book is your ultimate map for navigating the uncharted world of machine learning. Keep it within reach.

—Haesun Park, Microsoft AI MVP, Google Cloud Champion Innovator

This book launched a generation of ML practitioners. Brilliantly updated to cover PyTorch, it is once again the definitive hands-on guide to the field.

—Tarun Narayanan, machine learning engineer, Amazon AGI

A true bible for beginners in machine learning, this book not only provides clear explanations and hands-on examples but also uses thoughtfully designedfigures to simplify complex concepts, making it an indispensable resource for building a strong foundation.

—Meetu Malhotra,
Harrisburg University of Science and Technology, PA USA

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

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

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*.

Acquisitions Editor: Nicole Butterfield
Develeopment Editor: Michele Cronin
Production Editor: Beth Kelly
Copyeditor: Sonia Saruba
Proofreader: Kim Cofer
Indexer: Potomac Indexing LLC
Cover Designer: Susan Brown
Cover Illustrator: José Marzan Jr.
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

October 2025: First Edition

Revision History for the First Edition
2025-10-22: First Release

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

979-8-341-60798-9

[LSI]

Table of Contents

Preface

In 2006, Geoffrey Hinton et al. published a paper[1] 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,[2] 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 stomped 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.[3]

1 Geoffrey E. Hinton et al., "A Fast Learning Algorithm for Deep Belief Nets", *Neural Computation* 18 (2006): 1527–1554.

2 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.

3 Geoffrey Hinton was awarded the 2018 Turing Award (with Yann LeCun and Yoshua Bengio) and the 2024 Nobel Prize in Physics (with John Hopfield) for early work on neural networks back in the 1980s. DeepMind's

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 Python—the leading language for data science and machine learning—as well as 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.

---

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.

- 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 these open source machine learning libraries along the way:

- XGBoost in Chapter 6 to implement a powerful technique called *gradient boosting*.
- Hugging Face libraries in Chapters 13 and 15 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 Chapter 19 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.

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-mlp*, 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.

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*.

We appreciate, but generally 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 Geron. Copyright 2026 Aurélien Geron, 979-8-341-60798-9."

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 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", 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; the second part uses mostly PyTorch.

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.

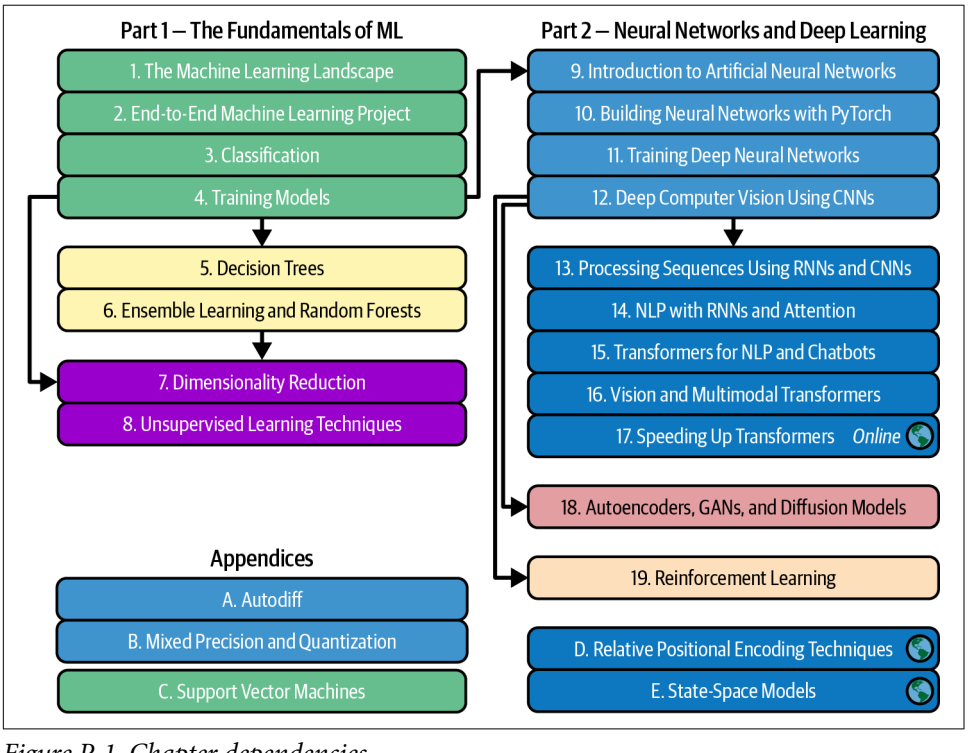

Figure P-1. Chapter dependencies

Changes Between the TensorFlow and PyTorch Versions

I wrote three TensorFlow (TF) 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 toward 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 chose to use PyTorch this time around! O'Reilly and I decided to make it the first edition of a new PyTorch series rather than the fourth edition of the original series. This leaves the door open for a JAX series or perhaps a new edition for the TF series (time will tell if either are needed).

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 and 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.
- Chapter 10 now introduces PyTorch.
- I also added three new chapters on transformers:
  - Chapter 15 covers transformers for natural language processing, including how to build a chatbot.
  - Chapter 16 presents vision transformers and multimodal transformers.
  - Chapter 17, available online at *https://homl.info/*, discusses several advanced techniques to speed up and scale up transformers. This includes FlashAttention, mixture of experts (MoE), low-rank adaptation (LoRA), and many more.
- There are also three new appendices: Appendix B explains how to shrink models so they can run faster and fit on smaller devices, "Relative Positional Encoding" discusses advanced positional encoding techniques for transformers, and "State-Space Models (SSMs)" presents state-space models.

- To make room for the newer content, the chapter on support vector machines (SVMs) was moved online and renamed "Support Vector Machines"; the last two appendices are also online at the same URL, and the deployment chapter was partially merged into Chapter 10.

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 three times in a row.

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.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

O'Reilly Online Learning

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.
141 Stony Circle, Suite 195
Santa Rosa, CA 95401
800-889-8969 (in the United States or Canada)
707-827-7019 (international or local)
707-829-0104 (fax)
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://oreil.ly/hands-on-machine-learning*.

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 generous people who offered their time and expertise to review this book, correcting errors and making countless suggestions. They made this book so much better: Jeremy Howard, Haesun Park, Omar Sanseviero, Lewis Tunstall, Leandro Von Werra, and Sam Witteveen reviewed the table of contents and helped me refine the scope of the book. Hesam Hassani, Ashu Jha, Meetu Malhotra, and Ammar Mohanna reviewed the first part, while Ulf Bissbort, Louis-Francois Bouchard, Luba Elliot, Thomas Lacombe, Tarun Narayanan, Marco Tabor, and my dear brother Sylvain reviewed the second part. Special thanks to Haesun Park, who reviewed every single chapter. You are all amazing!

Of course, this book would not exist without the fantastic staff at O'Reilly. I am especially indebted to Michele Cronin, who reviewed every chapter and supported me weekly for a whole year. I am also deeply grateful to Nicole Butterfield for leading this project and helping refine the book's scope, and to the production team—particularly Beth Kelly and Kristen Brown—who did a remarkable job. I want to acknowledge Sonia Saruba for her countless careful copyedits, Kate Dullea for making my diagrams much prettier, and Susan Thompson for the beautiful orangutan on the cover.

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 so 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 own thoughts and directly improved many parts of it. Plus, they kept bringing me cookies and coffee. Who could ask for more?

PART I

The Fundamentals of Machine Learning

CHAPTER 1
The Machine Learning Landscape

Not so long ago, if you had picked up your phone and asked it to tell you 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!

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.

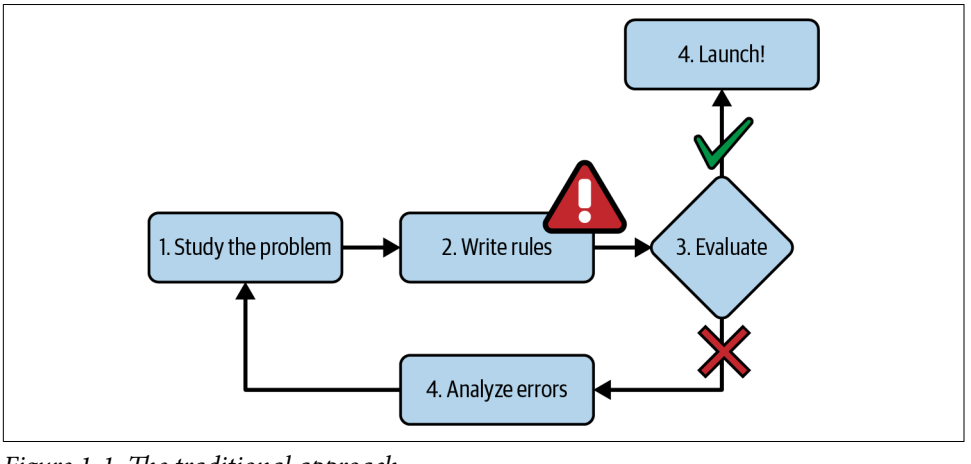

Figure 1-1. The traditional approach

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.

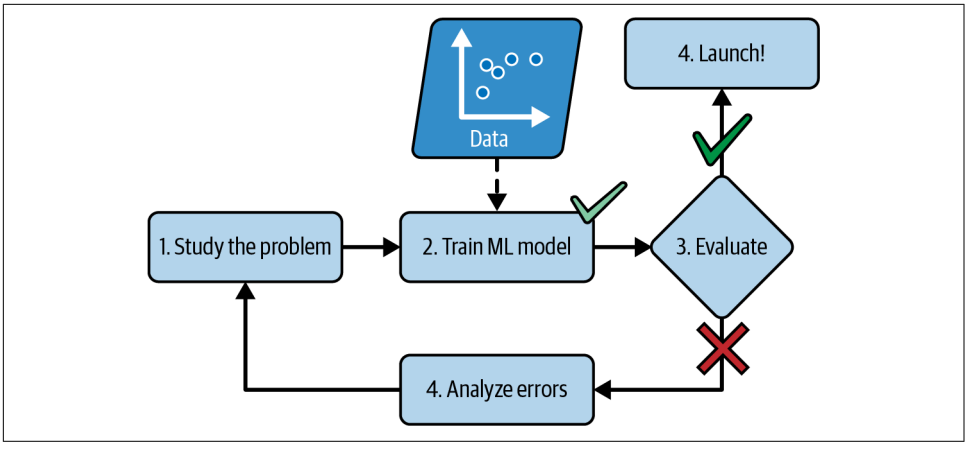

Figure 1-2. The machine learning approach

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).

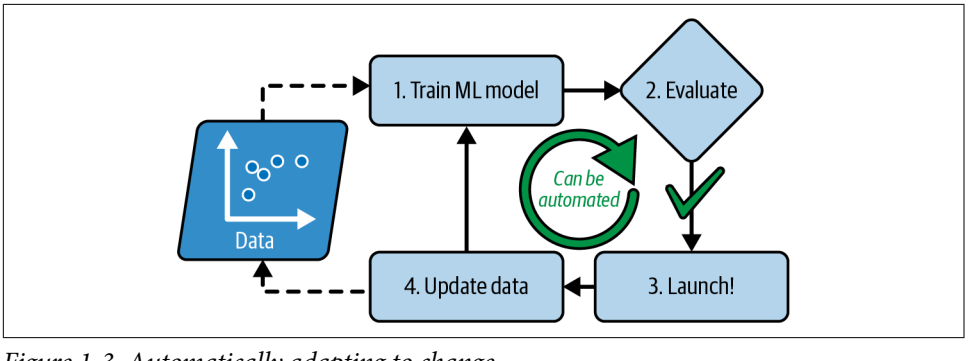

Figure 1-3. Automatically adapting to change

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.

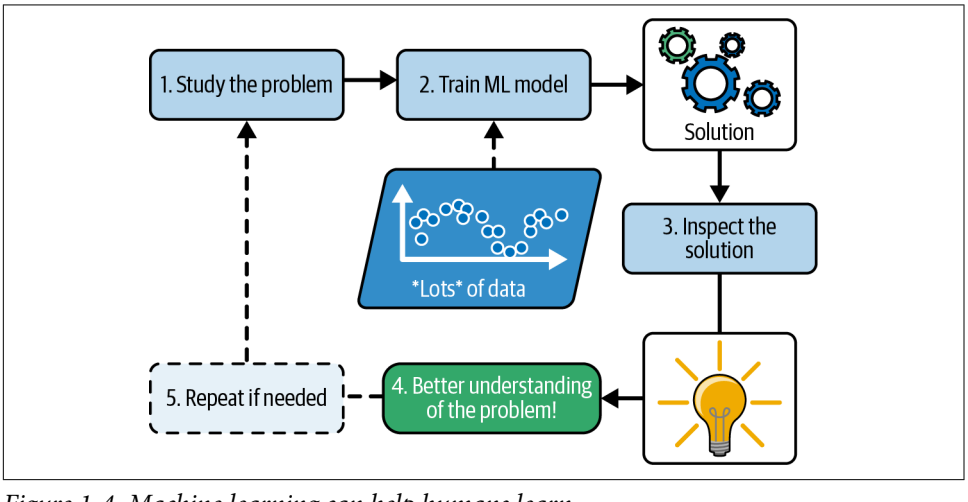

Figure 1-4. Machine learning can help humans learn

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 Chapter 12) or vision transformers (see Chapter 16).

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 Chapter 15).

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 "State-Space Models (SSMs)" at *https://homl.info*).

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 appendix on SVMs at *https://homl.info*), 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 13 to 15).

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 13 to 15).

Detecting credit card fraud
: This is anomaly detection, which can be tackled using isolation forests, Gaussian mixture models (see Chapter 8), or autoencoders (see Chapter 18).

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 Chapter 19), 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).

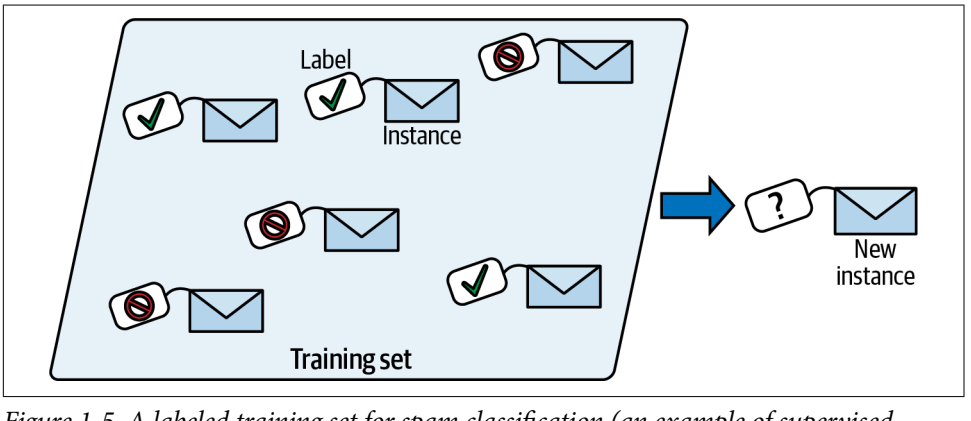

Figure 1-5. A labeled training set for spam classification (an example of supervised learning)

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).[1] 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).

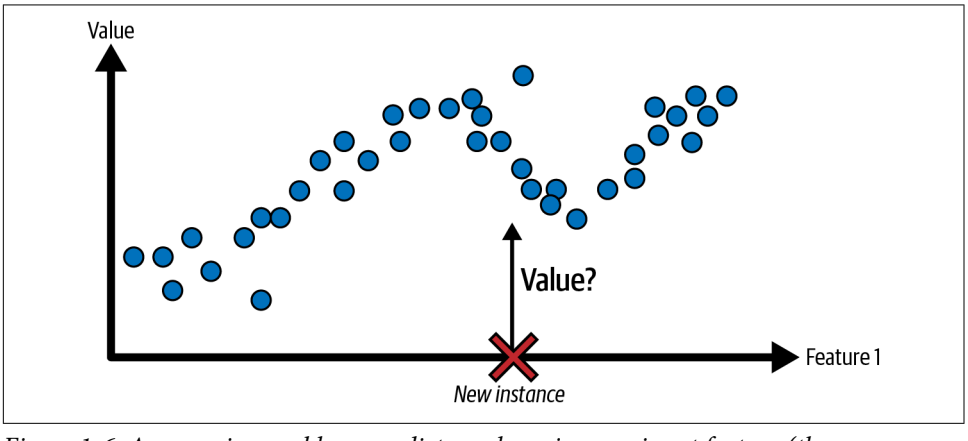

Figure 1-6. A regression problem: predict a value, given an input feature (there are usually multiple input features, and sometimes multiple output values)

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

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.

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.

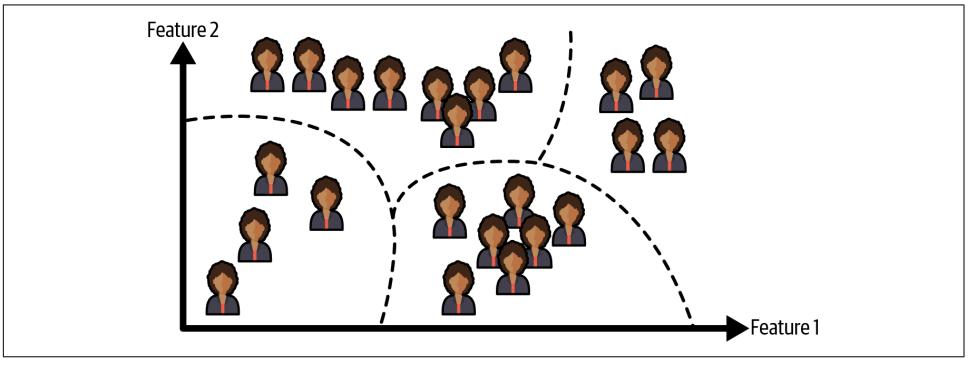

Figure 1-7. Clustering

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*.

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.

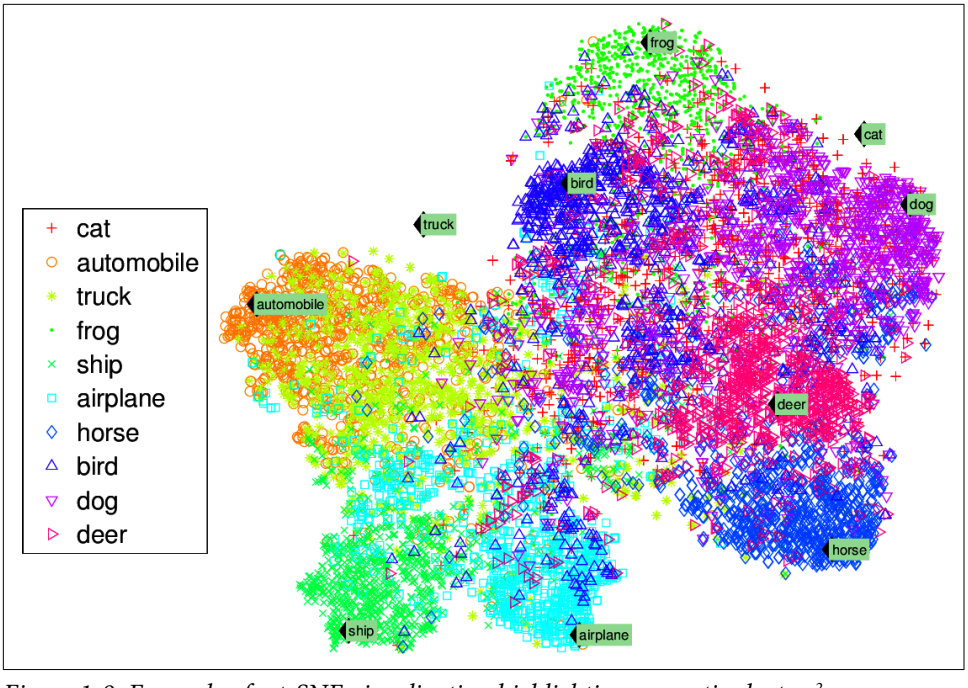

Figure 1-8. Example of a t-SNE visualization highlighting semantic clusters[2]

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

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.

from other dogs that they would likely classify them as anomalies (no offense to Chihuahuas).

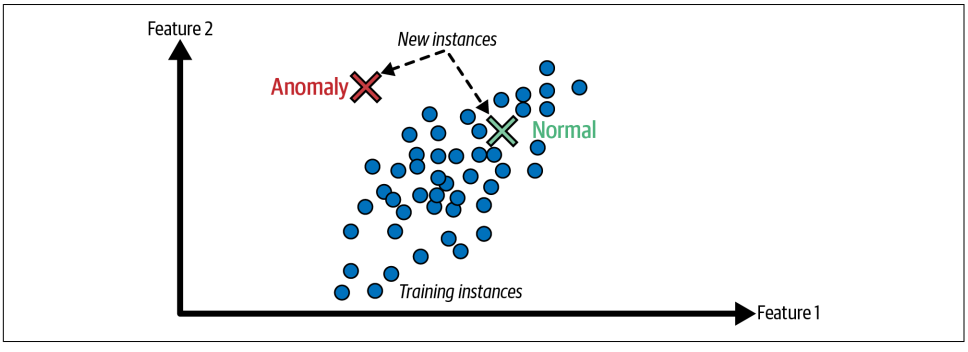

Figure 1-9. Anomaly detection

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).

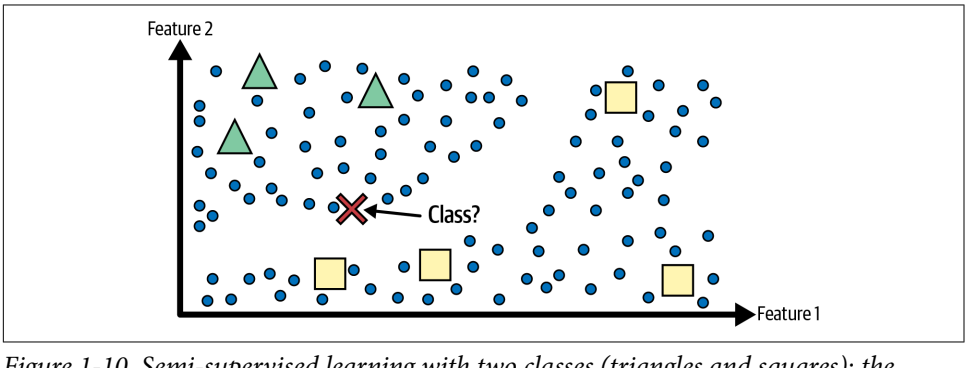

Figure 1-10. Semi-supervised learning with two classes (triangles and squares): the unlabeled examples (circles) help classify a new instance (the cross) into the triangle class rather than the square class, even though it is closer to the labeled squares

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[3] 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.

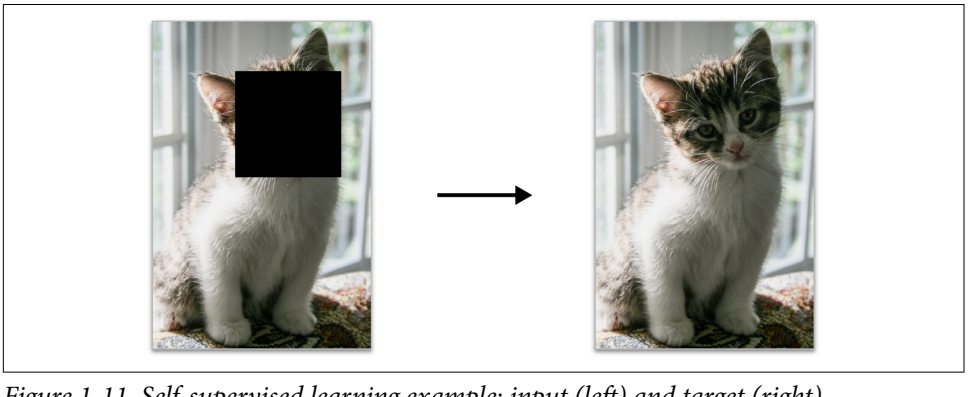

Figure 1-11. Self-supervised learning example: input (left) and target (right)

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.

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.

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 Chapter 15, 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.

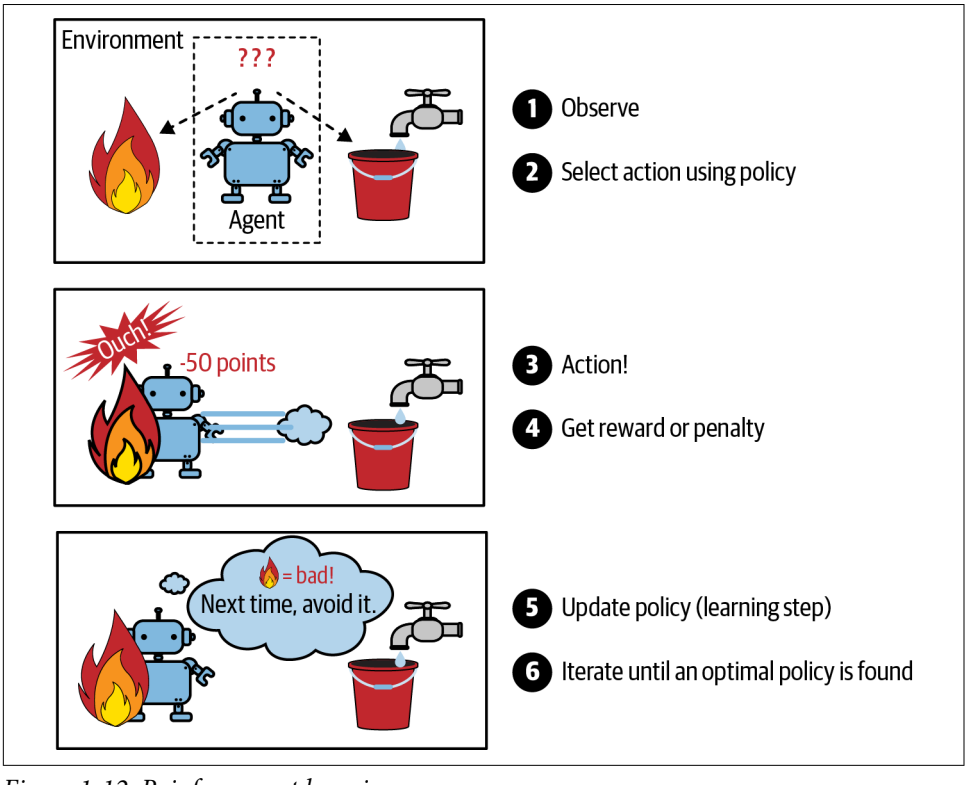

Figure 1-12. Reinforcement learning

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 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.

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.

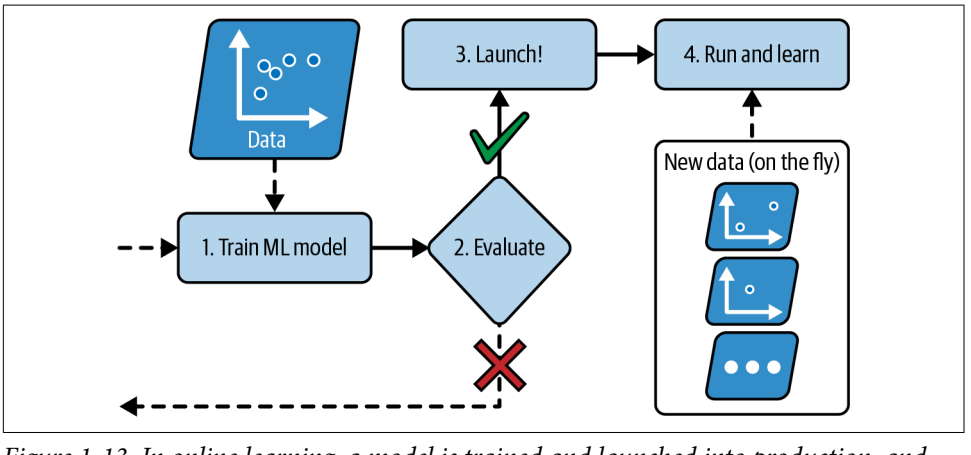

Figure 1-13. In online learning, a model is trained and launched into production, and then it keeps learning as new data comes in

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).

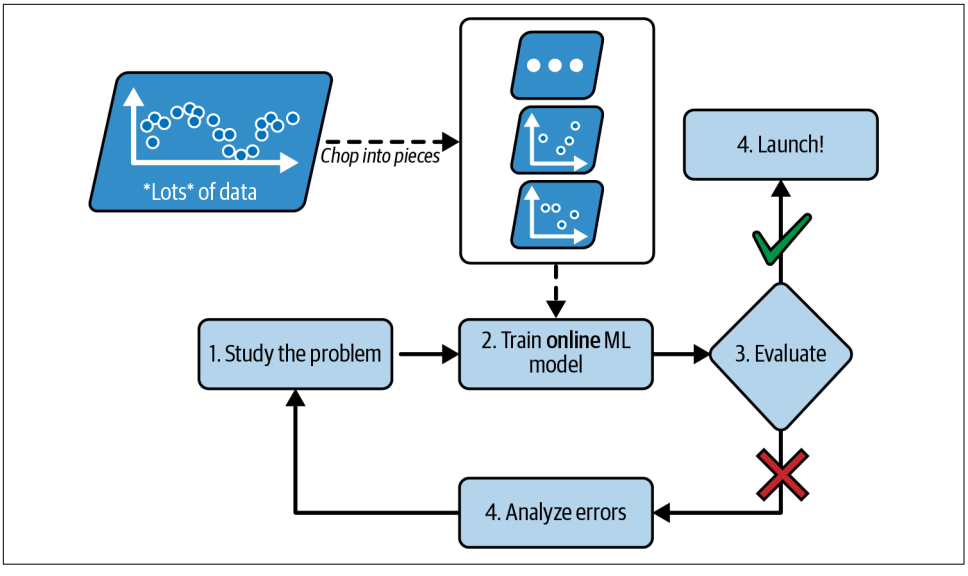

Figure 1-14. Using online learning to handle huge datasets

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).

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.

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.

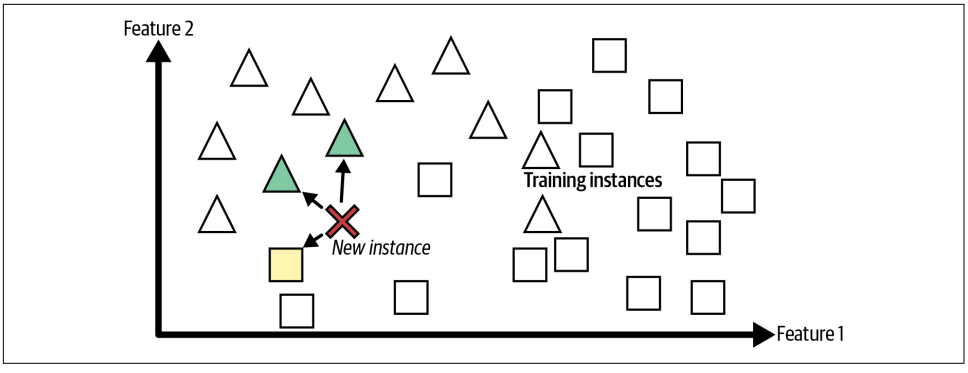

Figure 1-15. Instance-based learning: in this example we consider the class of the three nearest neighbors in the training set

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).

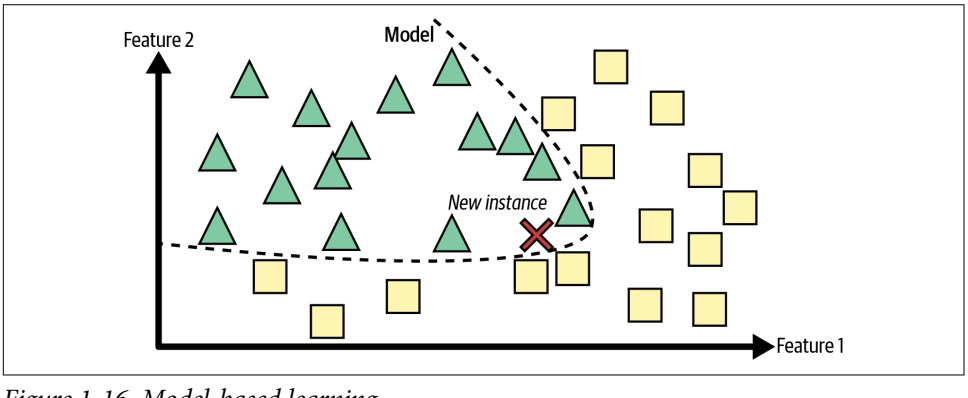

Figure 1-16. Model-based learning

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?

Country	GDP per capita (USD)	Life satisfaction
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).

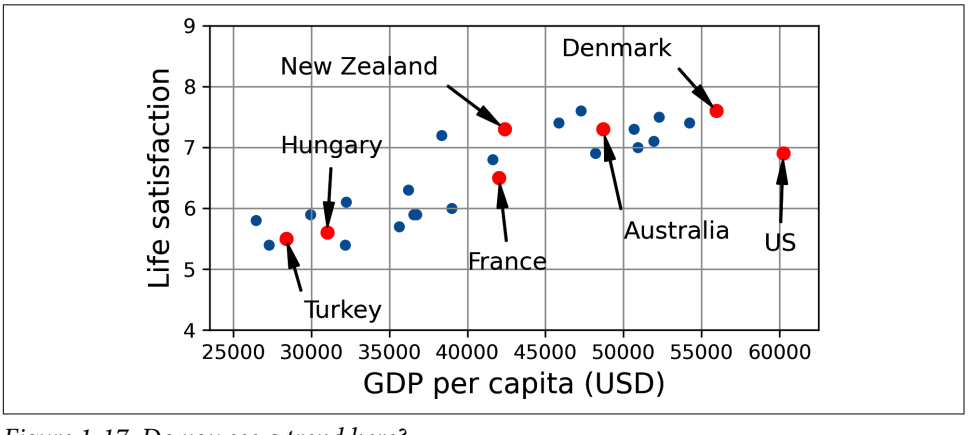

Figure 1-17. Do you see a trend here?

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*, θ_0 and θ_1.[4] By tweaking these parameters, you can make your model represent any linear function, as shown in Figure 1-18.

4 By convention, the Greek letter θ (theta) is frequently used to represent model parameters.

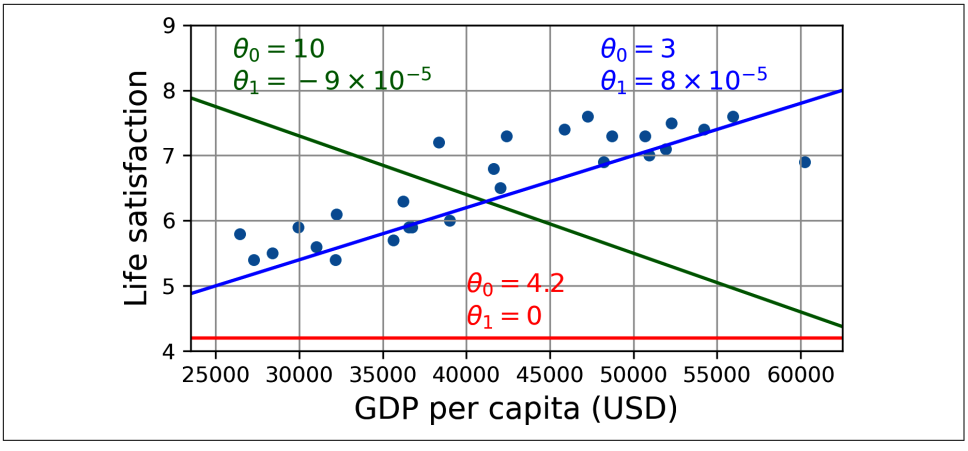

Figure 1-18. A few possible linear models

Before you can use your model, you need to define the parameter values θ_0 and θ_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 $\theta_1 = 6.78 \times 10^{-5}$.

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.

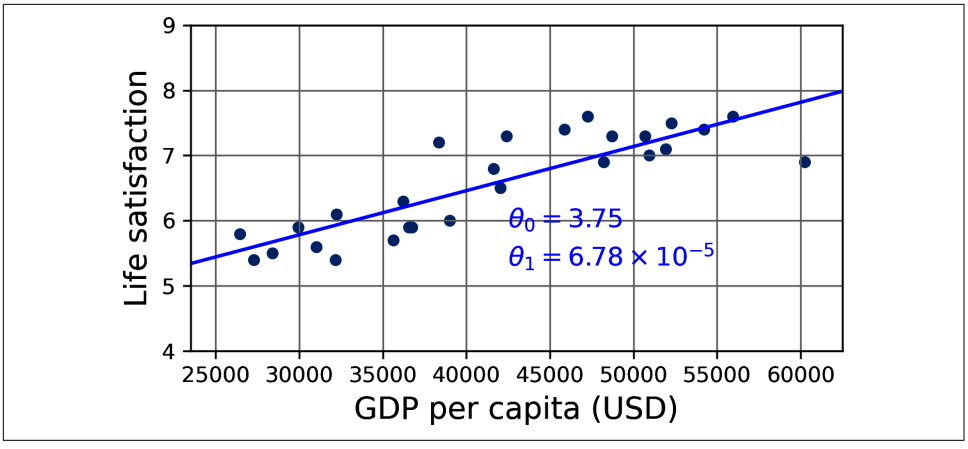

Figure 1-19. The linear model that fits the training data best

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.[5]

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()
```

5 It's OK if you don't understand all the code yet; I will present Scikit-Learn in the following chapters.

```
# 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]]
```

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.

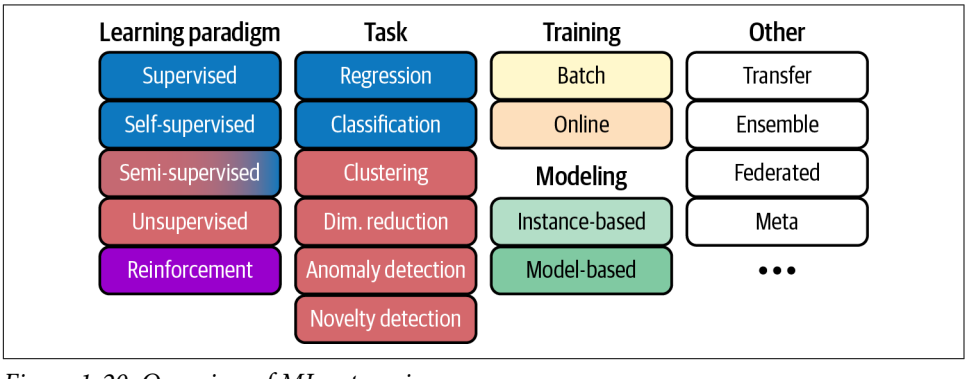

Figure 1-20. Overview of ML categories

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[6] once they were given enough data (as you can see in Figure 1-21).

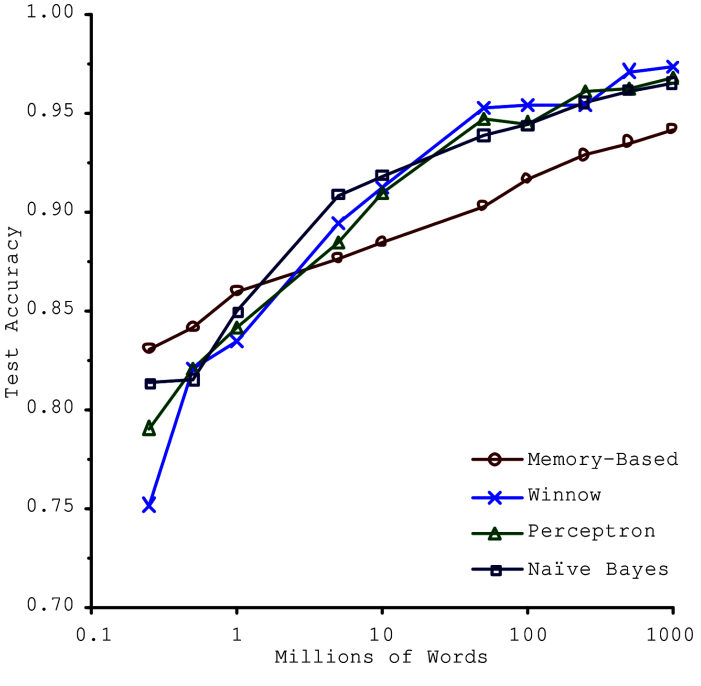

Figure 1-21. The importance of data versus algorithms[7]

6 For example, knowing whether to write "to", "two", or "too", depending on the context.

7 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.

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.[8] 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.

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*.

8 Peter Norvig et al., "The Unreasonable Effectiveness of Data", *IEEE Intelligent Systems* 24, no. 2 (2009): 8–12.

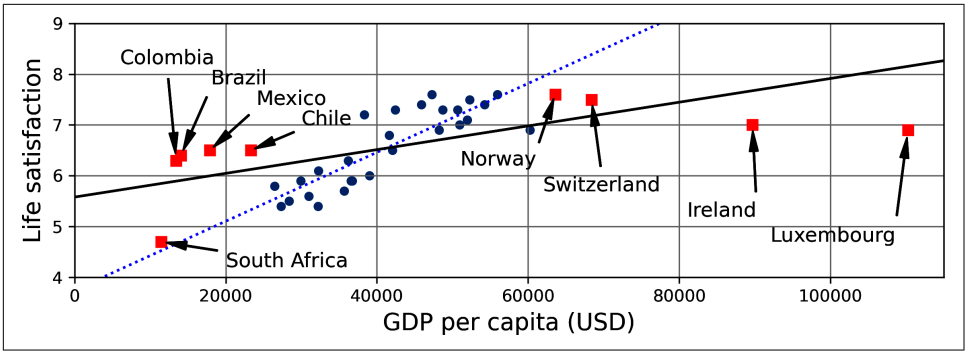

Figure 1-22. A more representative training sample

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?

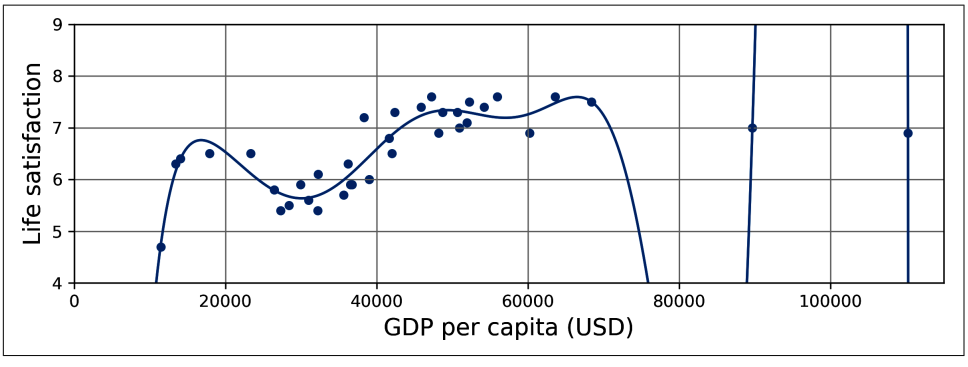

Figure 1-23. Overfitting the training data

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.

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, θ_0 and θ_1. This gives the learning algorithm two *degrees of freedom* to adapt the model to the training data: it can tweak both the height (θ_0) and the slope (θ_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 θ_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).

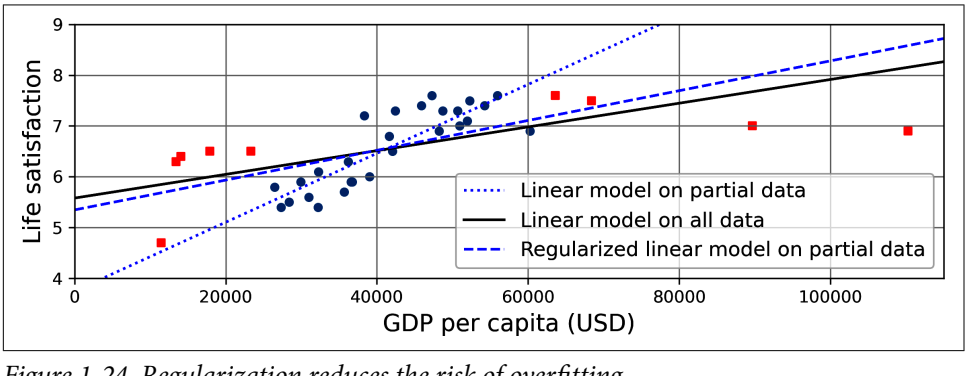

Figure 1-24. Regularization reduces the risk of overfitting

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.

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.

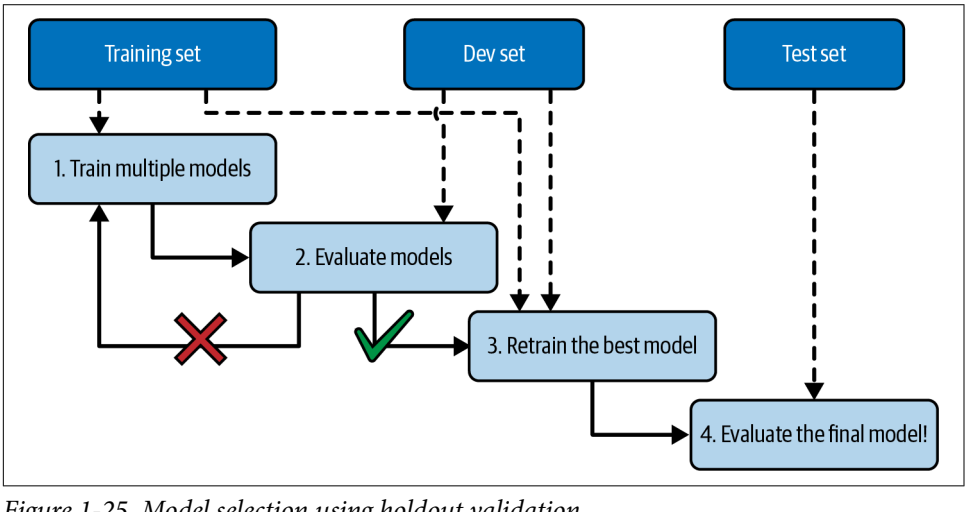

Figure 1-25. Model selection using holdout validation

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.

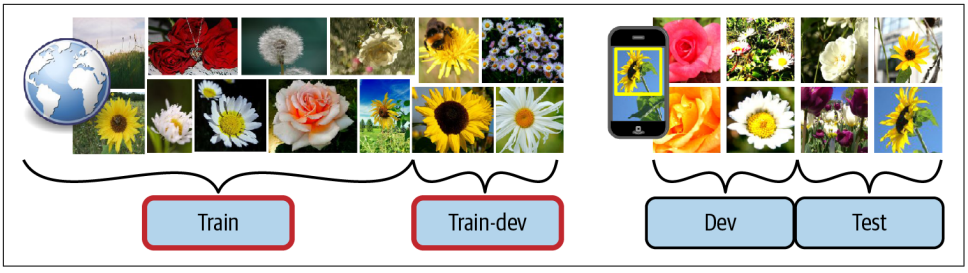

Figure 1-26. When real data is scarce (right), you may use similar abundant data (left) for training and hold out some of it in a train-dev set to evaluate overfitting; the real data is then used to evaluate data mismatch (dev set) and to evaluate the final model's performance (test set)

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,[9] 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.

9 David Wolpert, "The Lack of A Priori Distinctions Between Learning Algorithms", *Neural Computation* 8, no. 7 (1996): 1341–1390.

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*.

CHAPTER 2

End-to-End Machine Learning Project

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 popular open data repositories you can use to get data:

- Google Datasets Search
- Hugging Face Datasets
- OpenML.org

- Kaggle.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[1] (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.

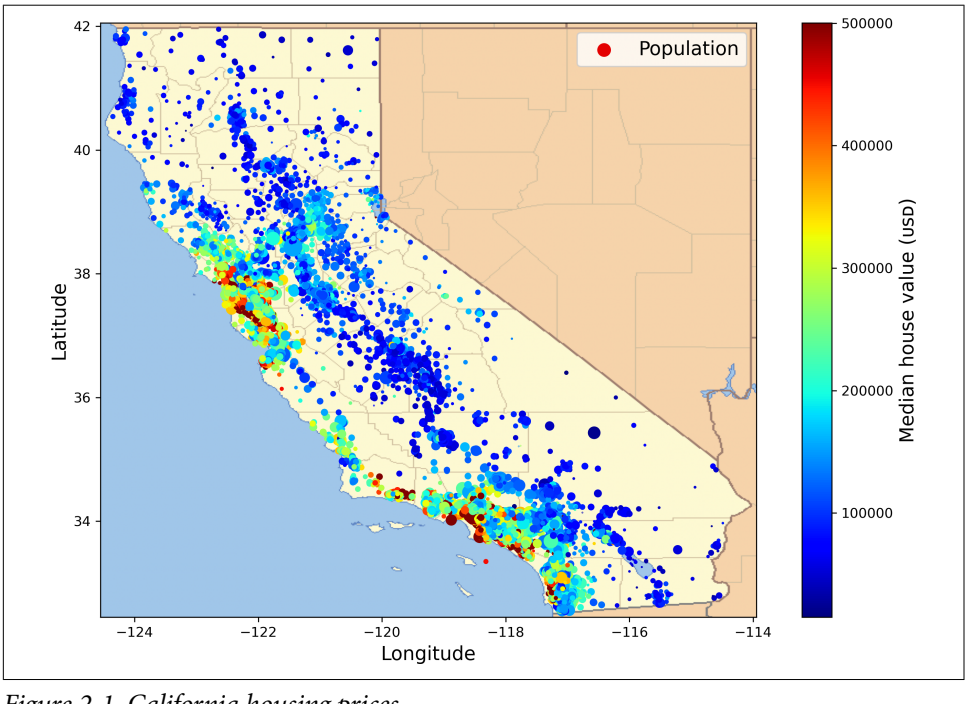

Figure 2-1. California housing prices

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

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.

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 at *https://homl.info/checklist*; 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 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.[2] 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

2 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.

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

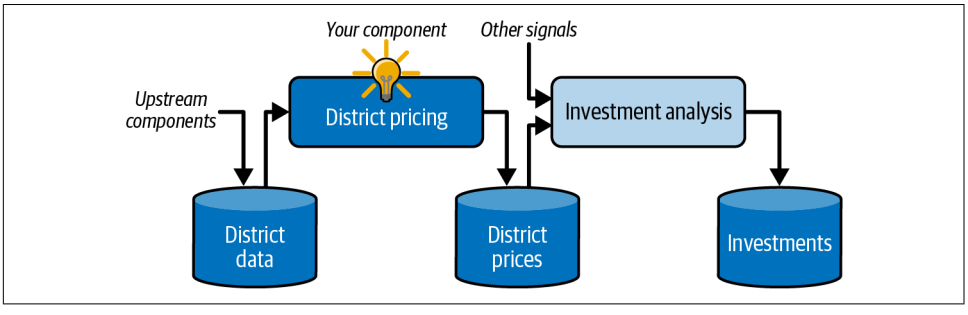

Figure 2-2. A machine learning pipeline for real estate investments

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.

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)

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

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^{th} instance in the dataset, and $y^{(i)}$ is its label (the desired output value for that instance). $\mathbf{y}$ is a vector containing the labels of all the instances in the dataset.
  - For example, if the first district in the dataset is located at longitude –118.29°, latitude 33.91°, 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^{th} row is equal to the transpose of $\mathbf{x}^{(i)}$, denoted $(\mathbf{x}^{(i)})^{\intercal}$.[3]
  - For example, if the first district is as just described, then the matrix $\mathbf{X}$ looks like this:

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

- 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 $\hat{y}^{(1)} - y^{(1)} = 2{,}000$.

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

- RMSE(**X**,*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)

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

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 ℓ_2 *norm*, denoted $\|\cdot\|_2$ (or just $\|\cdot\|$).
- Computing the sum of absolutes (MAE) corresponds to the ℓ_1 *norm*, denoted $\|\cdot\|_1$. 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 ℓ_k *norm* of a vector $\mathbf{v}$ containing n elements is defined as $\|\mathbf{v}\|_k = (|v_1|^k + |v_2|^k + \ldots + |v_n|^k)^{1/k}$. ℓ_0 gives the number of nonzero elements in the vector, and ℓ_∞ 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.

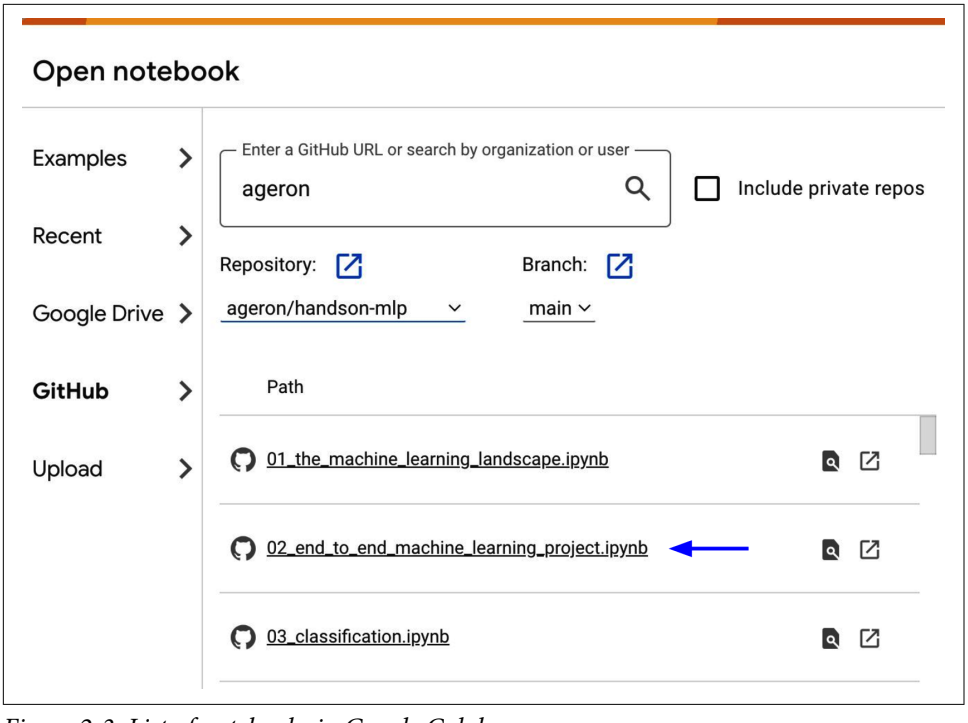

Figure 2-3. List of notebooks in Google Colab

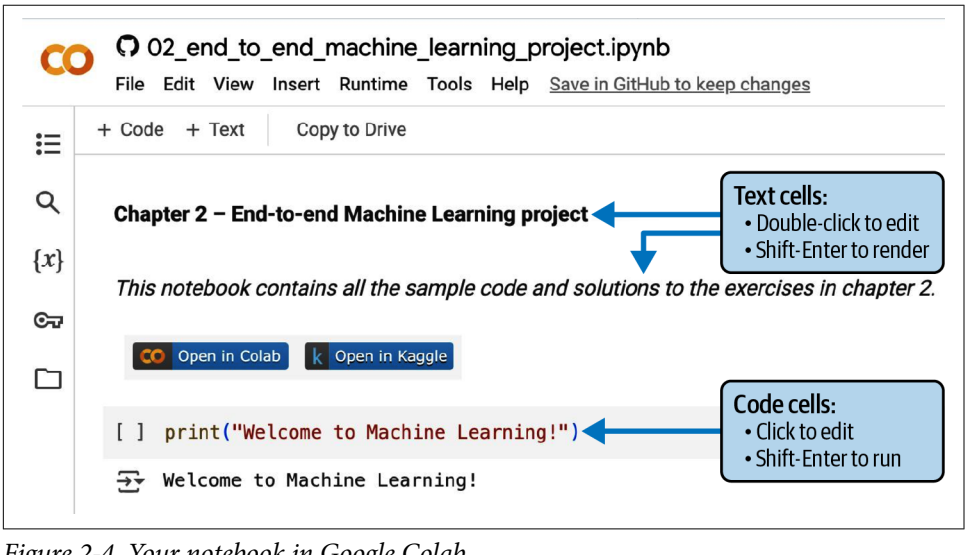

Figure 2-4. Your notebook in Google Colab

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!

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).

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.

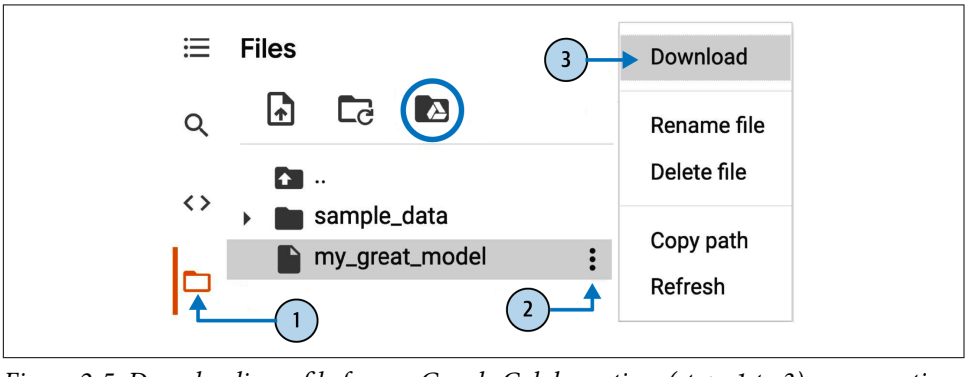

Figure 2-5. Downloading a file from a Google Colab runtime (steps 1 to 3), or mounting your Google Drive (circled icon)

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 [.keep-together]#/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.

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

and familiarize yourself with the data schema.[4] 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", filter="data")
    return pd.read_csv(Path("datasets/housing/housing.csv"))

housing_full = load_housing_data()
```

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*.

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.

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

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
```

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.[5] 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).

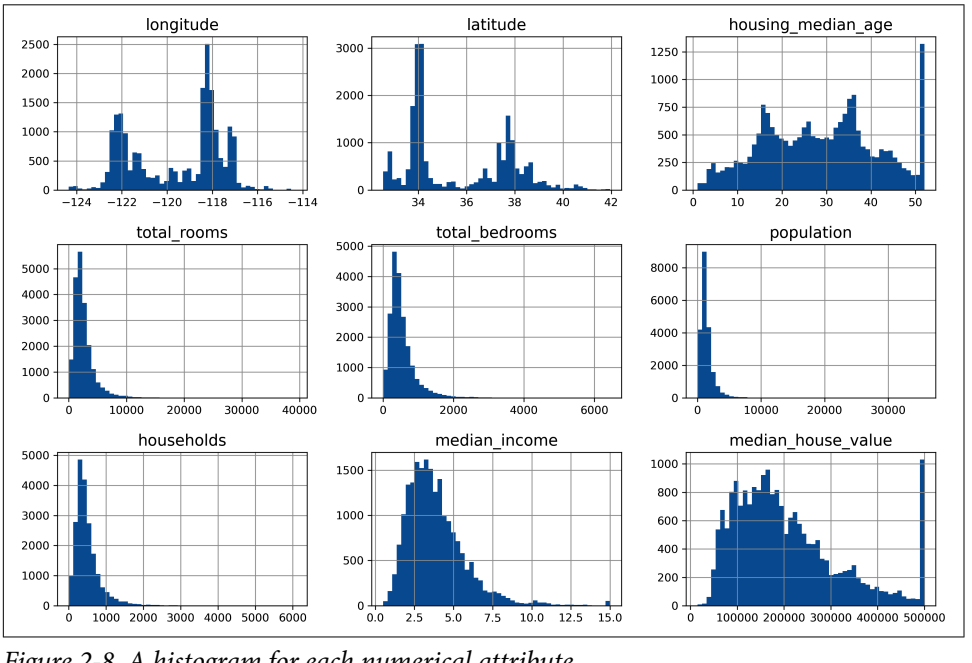

Figure 2-8. A histogram for each numerical attribute

5 The standard deviation is generally denoted σ (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 1σ of the mean, 95% within 2σ, and 99.7% within 3σ.

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()
```

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.

Create a Test Set

Before you look at the data any further, you need to create a test set, put it aside, and never look at it. 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)[6] 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 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.

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.

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:[7]

```
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

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.

a survey company decide 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, according to the US Census Bureau, 51.6% of citizens of voting age are female, while 48.4% are male, so a well-conducted survey in the US would try to maintain this ratio in the sample: 516 females and 484 males (at least if it seems likely 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 over 10% chance of sampling a skewed test set with less than 49% female or more than 54% 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.

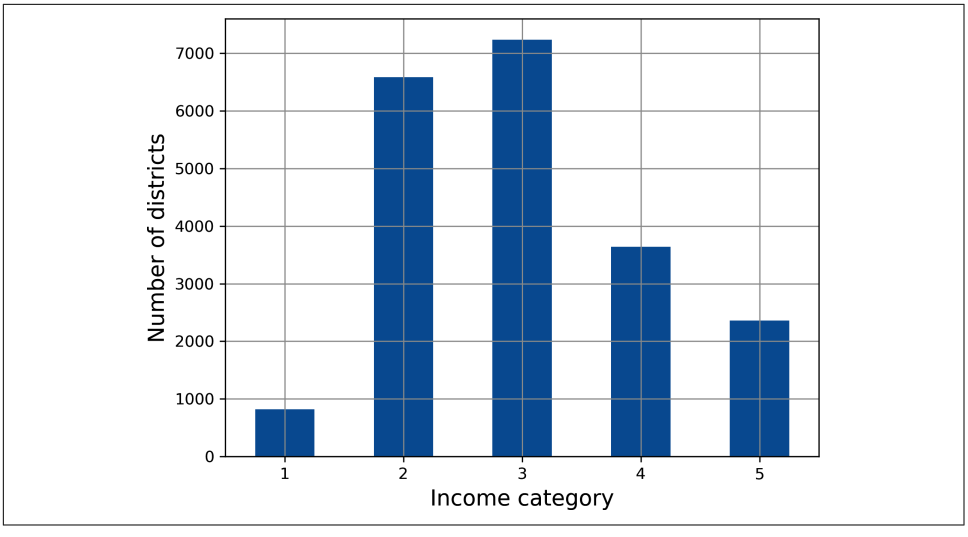

Figure 2-9. Histogram of income categories

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()
```

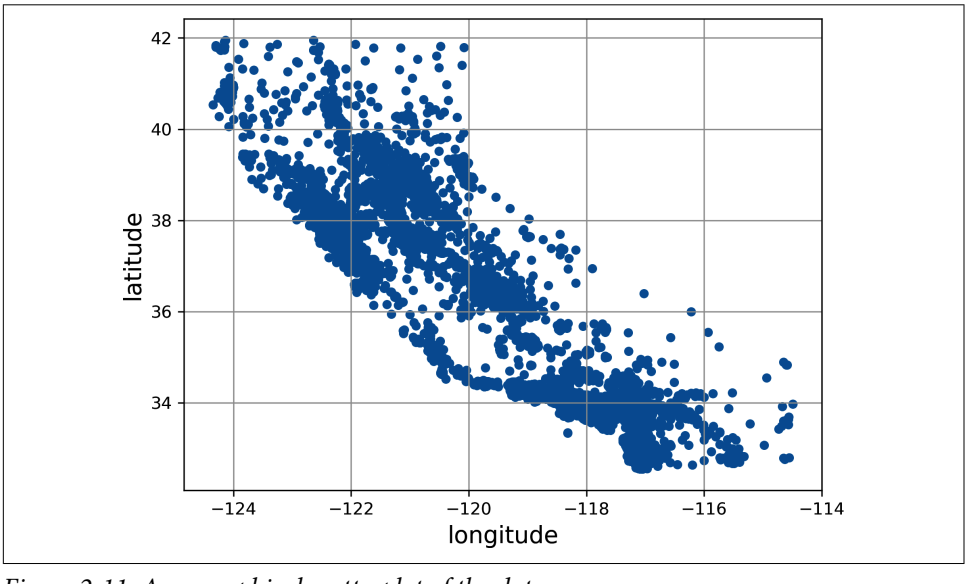

Figure 2-11. A geographical scatterplot of the data

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.

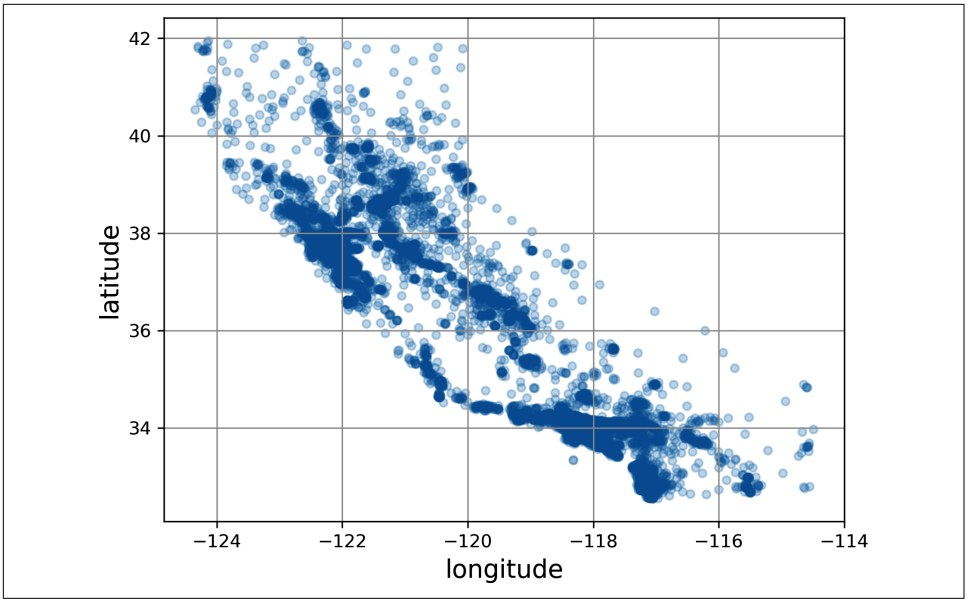

Figure 2-12. A better visualization that highlights high-density areas

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):[8]

```
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.

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.

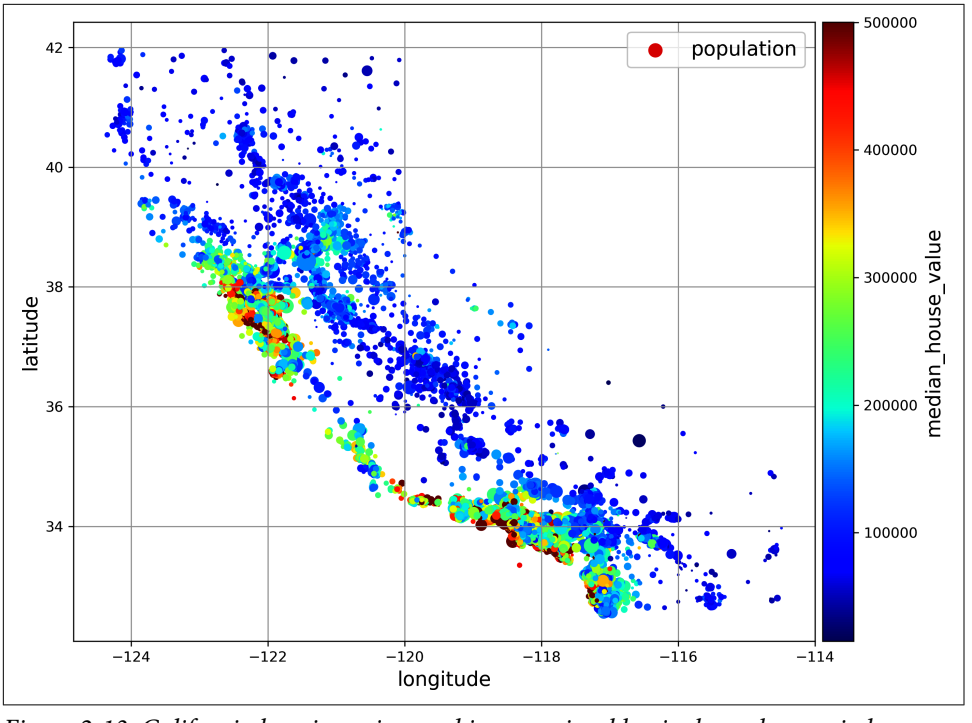

Figure 2-13. California housing prices: red is expensive, blue is cheap, larger circles indicate areas with a larger population

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()
```

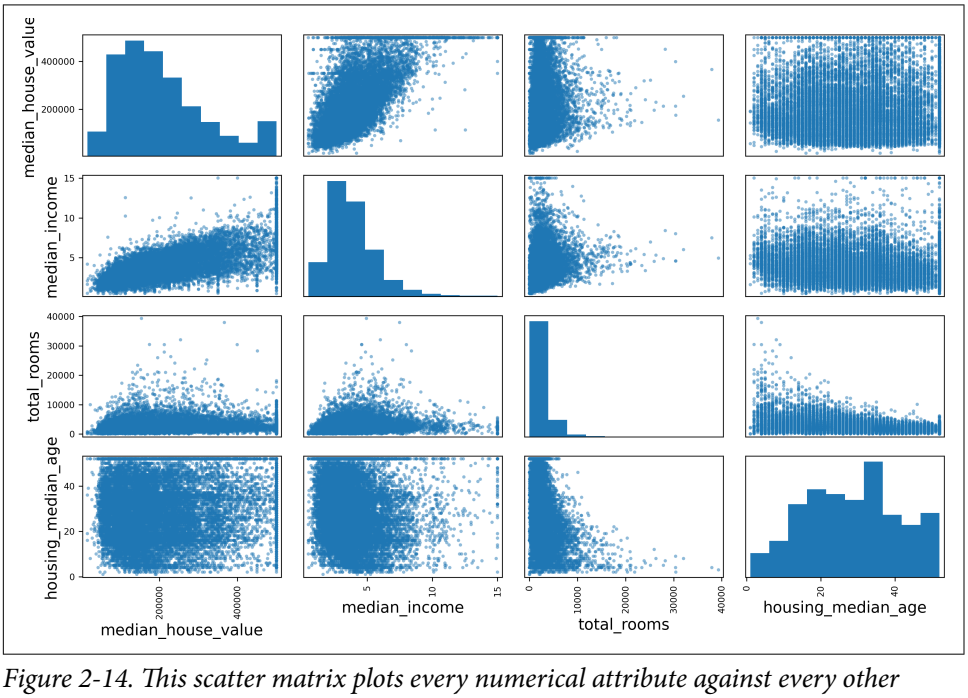

Figure 2-14. This scatter matrix plots every numerical attribute against every other numerical attribute, plus a histogram of each numerical attribute's values on the main diagonal (top left to bottom right)

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 that scatterplot (Figure 2-15):

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

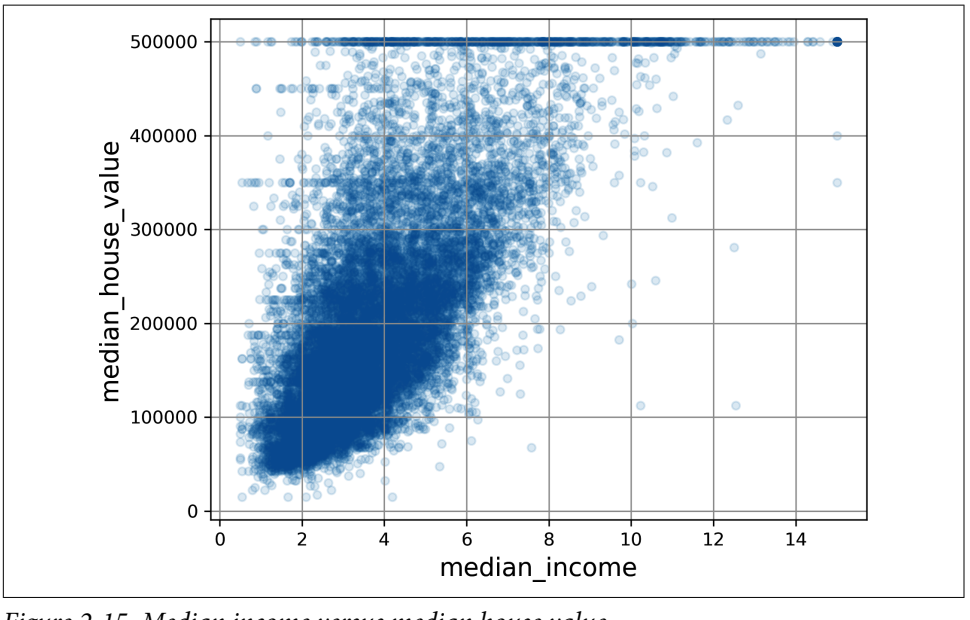

Figure 2-15. Median income versus median house value

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.

Note that 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.

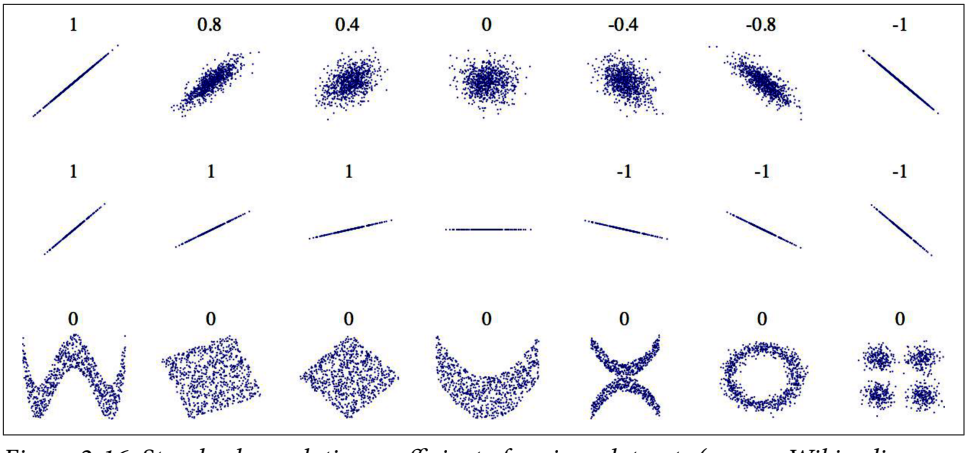

Figure 2-16. Standard correlation coefficient of various datasets (source: Wikipedia; public domain image)

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 total 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.

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.

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:[9]

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).[10]

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_`).

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.

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.[11] 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
```

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.

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,[12] 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
```

12 See SciPy's documentation for more details.

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.]])
```

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 Chapter 14). This is an example of *representation learning* (we will see more examples in Chapter 18).

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*.

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)
```

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).

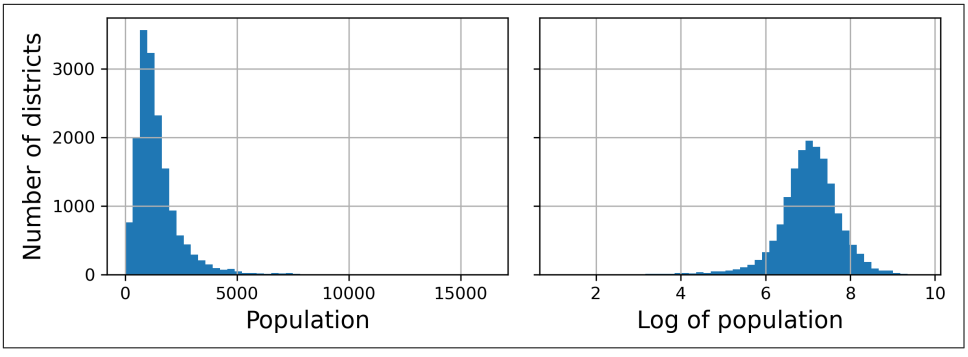

Figure 2-17. Transforming a feature to make it closer to a Gaussian distribution

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 γ (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.

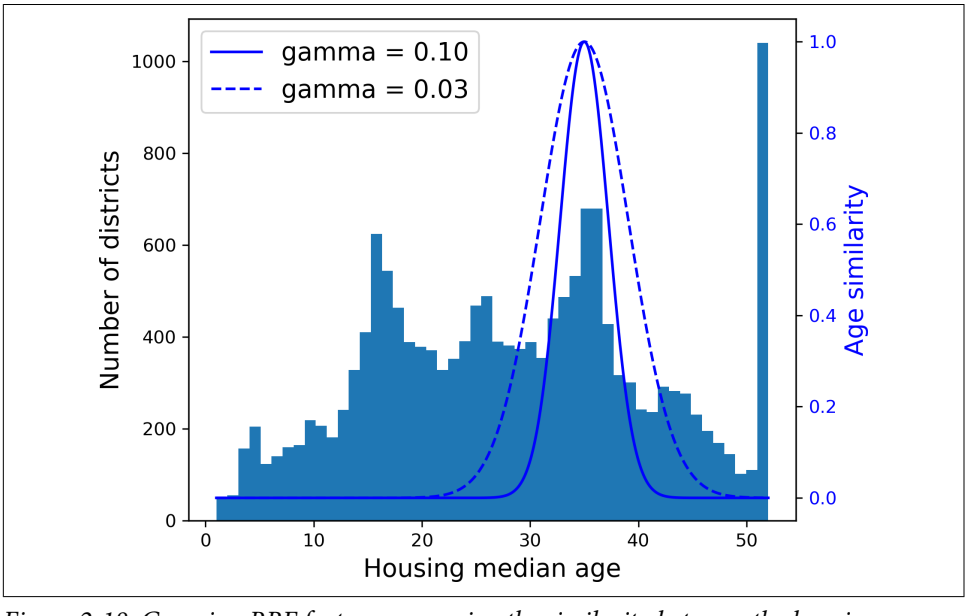

Figure 2-18. Gaussian RBF feature measuring the similarity between the housing median age and 35

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.

The rest of this section shows how to define custom transformer classes. In particular, it defines a custom transformer that groups districts into 10 geographical clusters, then measures the distance between each district and the center of each cluster, 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,[13] 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.

13 With duck typing, an object's methods and behavior are what matters, not its type: "if it looks like a duck and quacks like a duck, it must be a duck".

- 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)]
```

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 could pass it the median income if we wanted the clusters to be biased toward wealthy 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"]])
```

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 (you can try weighting each district by its median income to see how that affects the clusters). 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([[0.46, 0.  , 0.08, 0.  , 0.  , 0.  , 0.  , 0.98, 0.  , 0.  ],
       [0.  , 0.96, 0.  , 0.03, 0.04, 0.  , 0.  , 0.  , 0.11, 0.35],
       [0.34, 0.  , 0.45, 0.  , 0.  , 0.  , 0.01, 0.73, 0.  , 0.  ]])
```

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 areas.

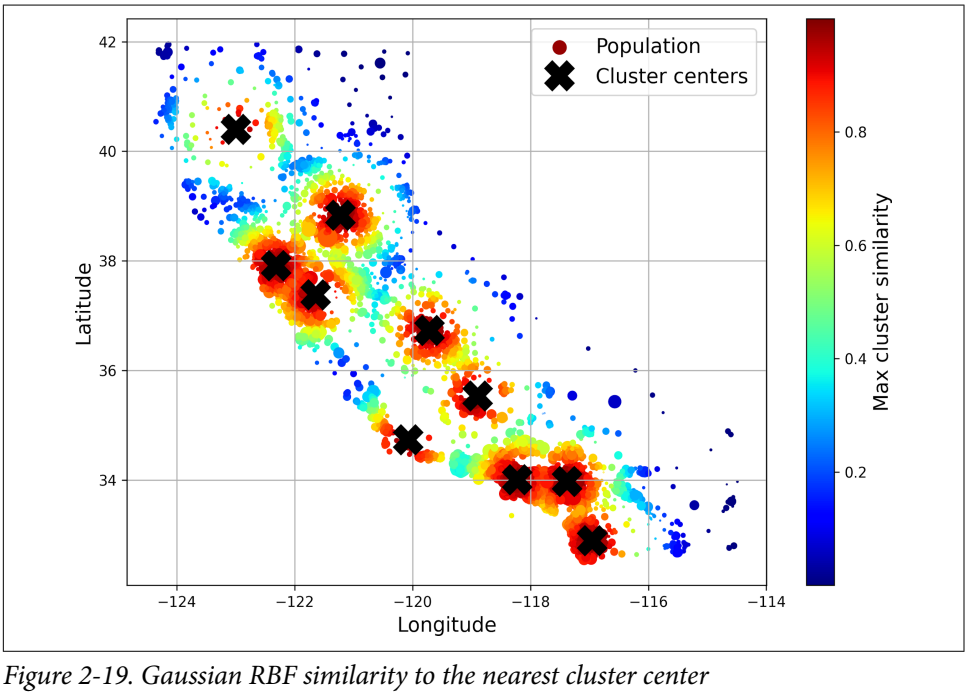

Figure 2-19. Gaussian RBF similarity to the nearest cluster center

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.

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.

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.

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
68972.88910758484
```

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).

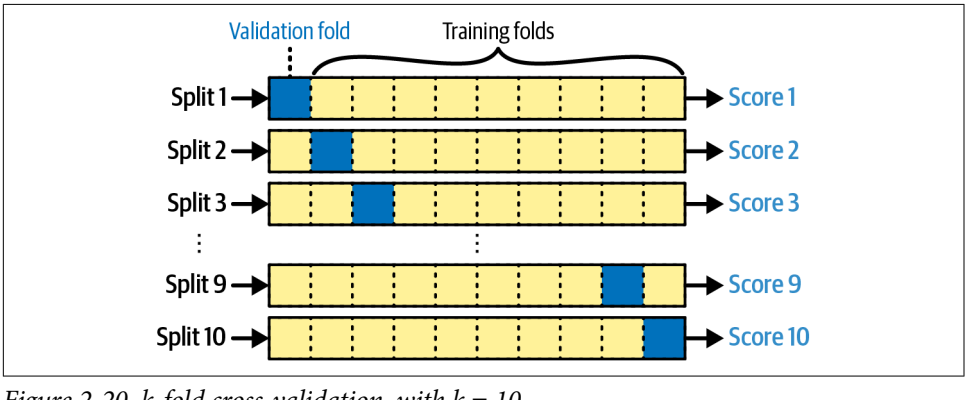

Figure 2-20. k-fold cross-validation, with k = 10

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)
```

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 66,574, 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).

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.

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   43708   44335           43590
13          15             8   43486   43820   44900           44069
6           10             4   43798   44036   44961           44265
9           10             6   43710   44163   44967           44280
7           10             6   43710   44163   44967           44280
```

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_distribs = {'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_distribs, 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(0.18599734460509476), 'log__median_income'),
 (np.float64(0.07338850855844489), 'cat__ocean_proximity_INLAND'),
 (np.float64(0.06556941990883976), 'bedrooms__ratio'),
 (np.float64(0.053648710076725316), 'rooms_per_house__ratio'),
 (np.float64(0.04598870861894749), 'people_per_house__ratio'),
 (np.float64(0.04175269214442519), 'geo__Cluster 30 similarity'),
 (np.float64(0.025976797232869678), 'geo__Cluster 25 similarity'),
 (np.float64(0.023595895886342255), 'geo__Cluster 36 similarity'),
 [...]
 (np.float64(0.0004325970342247361), 'cat__ocean_proximity_NEAR BAY'),
 (np.float64(3.0190221102670295e-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).

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 41445.533268606625
```

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 39,521 to 43,702, and your previous point estimate of 41,445 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")
```

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[14] (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.

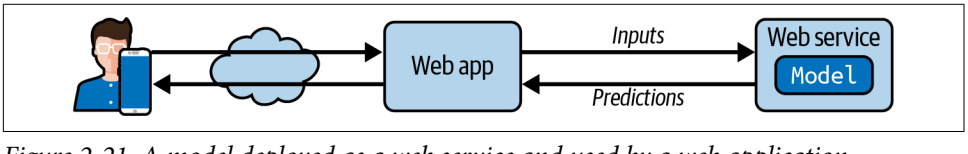

Figure 2-21. A model deployed as a web service and used by a web application

Another popular strategy is to deploy your model to the cloud, for example on Google's Vertex AI (formerly Google Cloud AI Platform and Google Cloud ML Engine): just save your model using `joblib` and upload it to Google Cloud Storage (GCS), then go 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

14 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.

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 nonrecommended 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.[15]

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:

15 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.

- 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/*. 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. The KNN regressor should use only the latitude and longitude as input and predict the median income. Next, add this new transformer to the preprocessing pipeline. This will add a feature representing the smoothed median income over the nearby 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.

7. Tackle a regression task of your choice by following the process you learned in this chapter. For example, you can try tackling the Vehicle dataset, where the goal is to predict the selling price of a used car, based on its age, the number of kilometers it has driven, its make and model, and more. Another good dataset to try is the Bike Sharing dataset: the objective is to predict the number of bikes rented within a period of time (column `cnt`), based on the day of the week, the time, and the weather conditions.

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