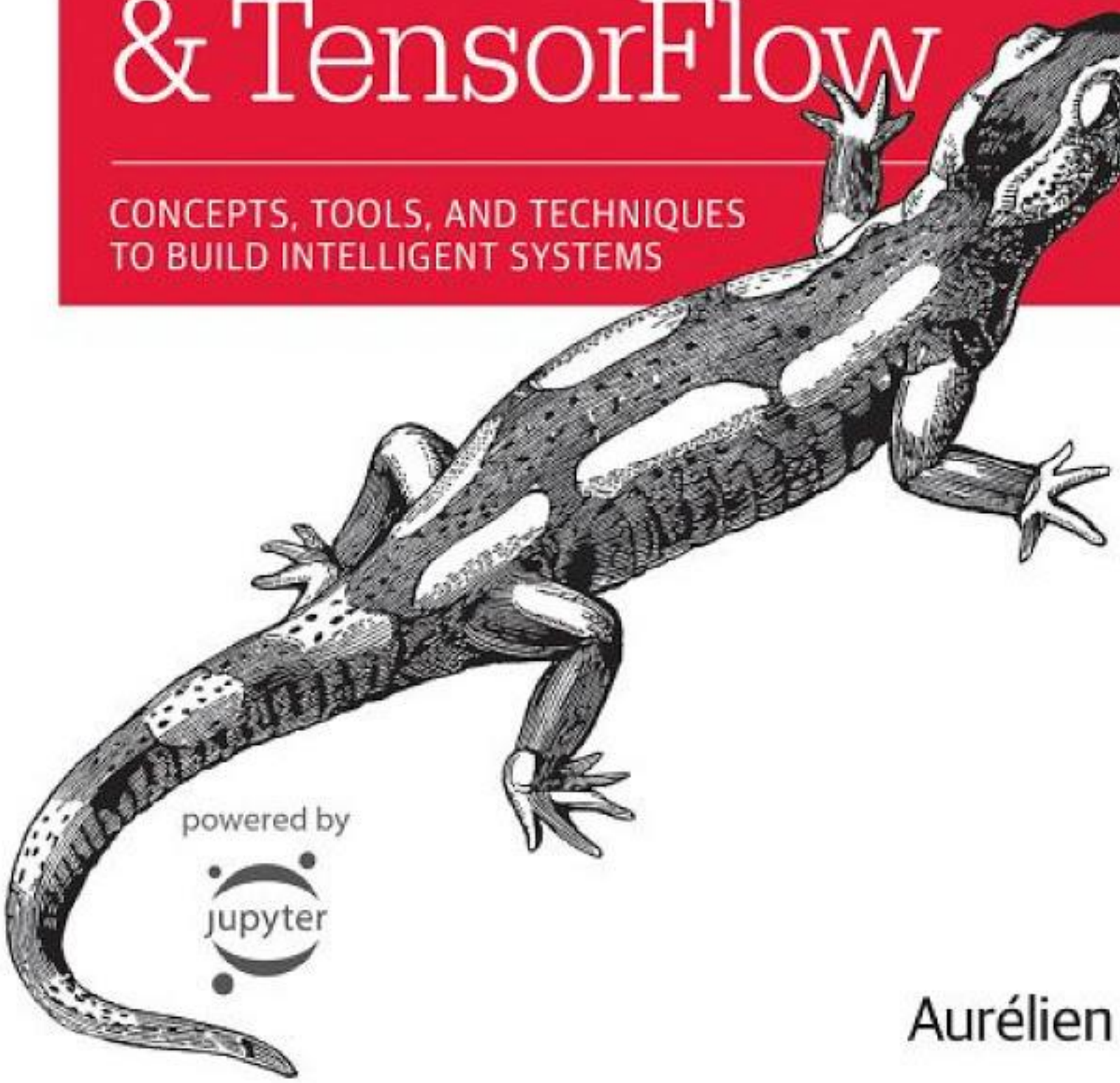
O'REILLY®

Hands-On Machine Learning with Scikit-Learn & TensorFlow

CONCEPTS, TOOLS, AND TECHNIQUES TO BUILD INTELLIGENT SYSTEMS



powered by jupyter

Aurélien Géron

Hands-On Machine Learning with Scikit-Learn and TensorFlow

by Aurélien Géron

Printed in the United States of America.

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

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

Editor: Nicole Tache

Production Editor: Nicholas Adams

Copyeditor: Rachel Monaghan

Proofreader: Charles Roumeliotis

Indexer: Wendy Catalano

Interior Designer: David Futato

Cover Designer: Randy Comer

Illustrator: Rebecca Demarest

March 2017: First Edition

Revision History for the First Edition

- 2017-03-10: First Release
- 2017-06-09: Second Release
- 2017-08-18: Third Release

See http://oreilly.com/catalog/errata.csp?isbn=9781491962299 for release details.

978-1-491-96229-9

[M]

Preface
The Machine Learning Tsunami

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." Training a deep neural net was widely considered impossible at the time,[2] and most researchers had abandoned the idea since the 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.

Fast-forward 10 years and Machine Learning has conquered the industry: it is now at the heart of much of the magic in today's high-tech products, ranking your web search results, powering your smartphone's speech recognition, and recommending videos, beating the world champion at the game of Go. Before you know it, it will be driving your car.

Machine Learning in Your Projects

So naturally you are excited about Machine Learning and you 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; for example:

- 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
- Predict next year's revenue
- And more

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, the intuitions, and the tools you need to actually 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.

Rather than implementing our own toy versions of each algorithm, we will be using actual 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 learn Machine Learning.
- TensorFlow is a more complex library for distributed numerical computation using data flow graphs. It makes it possible to train and run very large neural networks efficiently by distributing the computations across potentially thousands of multi-GPU servers. TensorFlow was created at Google and supports many of their large-scale Machine Learning applications. It was open-sourced in November 2015.

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, we highly recommend you experiment with the code examples available online as Jupyter notebooks at *https://github.com/ageron/handson-ml*.

Prerequisites

This book assumes that you have some Python programming experience and that you are familiar with Python's main scientific libraries, in particular NumPy, Pandas, and Matplotlib.

Also, if you care about what's under the hood you should have a reasonable understanding of college-level math as well (calculus, linear algebra, probabilities, and statistics).

If you don't know Python yet, *http://learnpython.org/* is a great place to start. The official tutorial on python.org is also quite good.

If you have never used Jupyter, Chapter 2 will guide you through installation and the basics: it is a great tool to have in your toolbox.

If you are not familiar with Python's scientific libraries, the provided Jupyter notebooks include a few tutorials. There is also a quick math tutorial for linear algebra.

Roadmap

This book is organized in two parts. Part I, *The Fundamentals of Machine Learning*, covers the following topics:

- What is Machine Learning? What problems does it try to solve? What are the main categories and fundamental concepts of Machine Learning systems?
- The main steps in a typical Machine Learning project.
- Learning by fitting a model to data.
- Optimizing a cost function.

- Handling, cleaning, and preparing data.
- Selecting and engineering features.
- Selecting a model and tuning hyperparameters using cross-validation.
- The main challenges of Machine Learning, in particular underfitting and overfitting (the bias/variance tradeoff).
- Reducing the dimensionality of the training data to fight the curse of dimensionality.
- The most common learning algorithms: Linear and Polynomial Regression, Logistic Regression, k-Nearest Neighbors, Support Vector Machines, Decision Trees, Random Forests, and Ensemble methods.

Part II, *Neural Networks and Deep Learning*, covers the following topics:

- What are neural nets? What are they good for?
- Building and training neural nets using TensorFlow.
- The most important neural net architectures: feedforward neural nets, convolutional nets, recurrent nets, long short-term memory (LSTM) nets, and autoencoders.
- Techniques for training deep neural nets.
- Scaling neural networks for huge datasets.
- Reinforcement learning.

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

Caution

Don’t jump into deep waters too hastily: while Deep Learning is no doubt one of the most exciting areas in Machine Learning, you should master the

fundamentals first. Moreover, most 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, provided you have enough data, computing power, and patience.

Other Resources

Many resources are available to learn about Machine Learning. Andrew Ng's ML course on Coursera and Geoffrey Hinton's course on neural networks and Deep Learning are amazing, although they both require a significant time investment (think months).

There are also many interesting websites about Machine Learning, including of course Scikit-Learn's exceptional User Guide. You may also enjoy Dataquest, which provides very nice interactive tutorials, and ML blogs such as those listed on Quora. Finally, the Deep Learning website has a good list of resources to learn more.

Of course there are also many other introductory books about Machine Learning, in particular:

- Joel Grus, *Data Science from Scratch* (O'Reilly). This book presents the fundamentals of Machine Learning, and implements some of the main algorithms in pure Python (from scratch, as the name suggests).
- Stephen Marsland, *Machine Learning: An Algorithmic Perspective* (Chapman and Hall). This book 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, *Python Machine Learning* (Packt Publishing). Also a great introduction to Machine Learning, this book leverages Python open source libraries (Pylearn 2 and Theano).

- Yaser S. Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin, *Learning from Data* (AMLBook). A rather theoretical approach to ML, this book provides deep insights, in particular on the bias/variance tradeoff (see Chapter 4).
- Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach, 3rd Edition* (Pearson). This is a great (and huge) book covering an incredible amount of topics, including Machine Learning. It helps put ML into perspective.

Finally, a great way to learn is to join ML competition websites such as Kaggle.com this 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.

Tip

This element signifies a tip or suggestion.

Note

This element signifies a general note.

Warning

This element indicates a warning or caution.

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/ageron/handson-ml*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Hands-On Machine Learning with Scikit-Learn and TensorFlow* by Aurélien Géron (O'Reilly). Copyright 2017 Aurélien Géron, 978-1-491-96229-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

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

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/hands-on-machine-learning-with-scikit-learn-and-tensorflow*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I would like to thank my Google colleagues, in particular the YouTube video classification team, for teaching me so much about Machine Learning. I could never have started this project without them. Special thanks to my personal ML gurus: Clément Courbet, Julien Dubois, Mathias Kende, Daniel Kitachewsky, James Pack, Alexander Pak, Anosh Raj, Vitor Sessak, Wiktor Tomczak, Ingrid von Glehn, Rich Washington, and everyone at YouTube Paris.

I am incredibly grateful to all the amazing people who took time out of their busy lives to review my book in so much detail. Thanks to Pete Warden for answering all my TensorFlow questions, reviewing Part II, providing many interesting insights, and of course for being part of the core TensorFlow team. You should definitely check out his blog! Many thanks to Lukas Biewald for his very thorough review of Part II: he left no stone unturned, tested all the code (and caught a few errors), made many great suggestions, and his enthusiasm was contagious. You should check out his blog and his cool robots! Thanks to Justin Francis, who also reviewed Part II very thoroughly, catching errors and providing great insights, in particular in Chapter 16. Check out his posts on TensorFlow!

Huge thanks as well to David Andrzejewski, who reviewed Part I and provided incredibly useful feedback, identifying unclear sections and

suggesting how to improve them. Check out his website! Thanks to Grégoire Mesnil, who reviewed Part II and contributed very interesting practical advice on training neural networks. Thanks as well to Eddy Hung, Salim Sémaoune, Karim Matrah, Ingrid von Glehn, Iain Smears, and Vincent Guilbeau for reviewing Part I and making many useful suggestions. And I also wish to thank my father-in-law, Michel Tessier, former mathematics teacher and now a great translator of Anton Chekhov, for helping me iron out some of the mathematics and notations in this book and reviewing the linear algebra Jupyter notebook.

And of course, a gigantic “thank you” to my dear brother Sylvain, who reviewed every single chapter, tested every line of code, provided feedback on virtually every section, and encouraged me from the first line to the last. Love you, bro!

Many thanks as well to O’Reilly’s fantastic staff, in particular Nicole Tache, who gave me insightful feedback, always cheerful, encouraging, and helpful. Thanks as well to Marie Beaugureau, Ben Lorica, Mike Loukides, and Laurel Ruma for believing in this project and helping me define its scope. Thanks to Matt Hacker and all of the Atlas team for answering all my technical questions regarding formatting, asciidoc, and LaTeX, and thanks to Rachel Monaghan, Nick Adams, and all of the production team for their final review and their hundreds of corrections.

Last but not least, I am infinitely grateful to my beloved wife, Emmanuelle, and to our three wonderful kids, Alexandre, Rémi, and Gabrielle, for encouraging me to work hard on this book, asking many questions (who said you can’t teach neural networks to a seven-year-old?), and even bringing me cookies and coffee. What more can one dream of?

[1] Available on Hinton’s home page at *http://www.cs.toronto.edu/~hinton/*.

[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.

Part I. The Fundamentals of Machine Learning

Chapter 1. The Machine Learning Landscape

When most people hear “Machine Learning,” they picture a robot: a dependable butler or a deadly Terminator depending on who you ask. But Machine Learning is not just a futuristic fantasy, it’s already here. In fact, it has been around for decades in some specialized applications, such as *Optical Character Recognition* (OCR). But the first ML application that really became mainstream, improving the lives of hundreds of millions of people, took over the world back in the 1990s: it was the *spam filter*. Not exactly a self-aware Skynet, 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). It was followed by hundreds of ML applications that now quietly power hundreds of products and features that you use regularly, from better recommendations to voice search.

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 Wikipedia, has my computer really “learned” something? Is it suddenly smarter? In this chapter we 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, 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 (the only chapter without much code), all rather simple, but you should make sure everything is crystal-clear to you before continuing to the rest of the book. So grab a coffee and let’s get started!

Tip

If you already know all the 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*

For example, your spam filter is a Machine Learning program that can learn to flag spam given examples of spam emails (e.g., flagged by users) and examples of regular (nonspam, also called "ham") emails. The examples that the system uses to learn are called the *training set*. Each training example is called a *training instance* (or *sample*). 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.

If you just download a copy of Wikipedia, your computer has a lot more data, but it is not suddenly better at any task. Thus, it 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 look at 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. Perhaps you would also notice a few other patterns in the sender’s name, the email’s body, and so on.

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 are detected.

3. You would test your program, and repeat steps 1 and 2 until it is good enough.

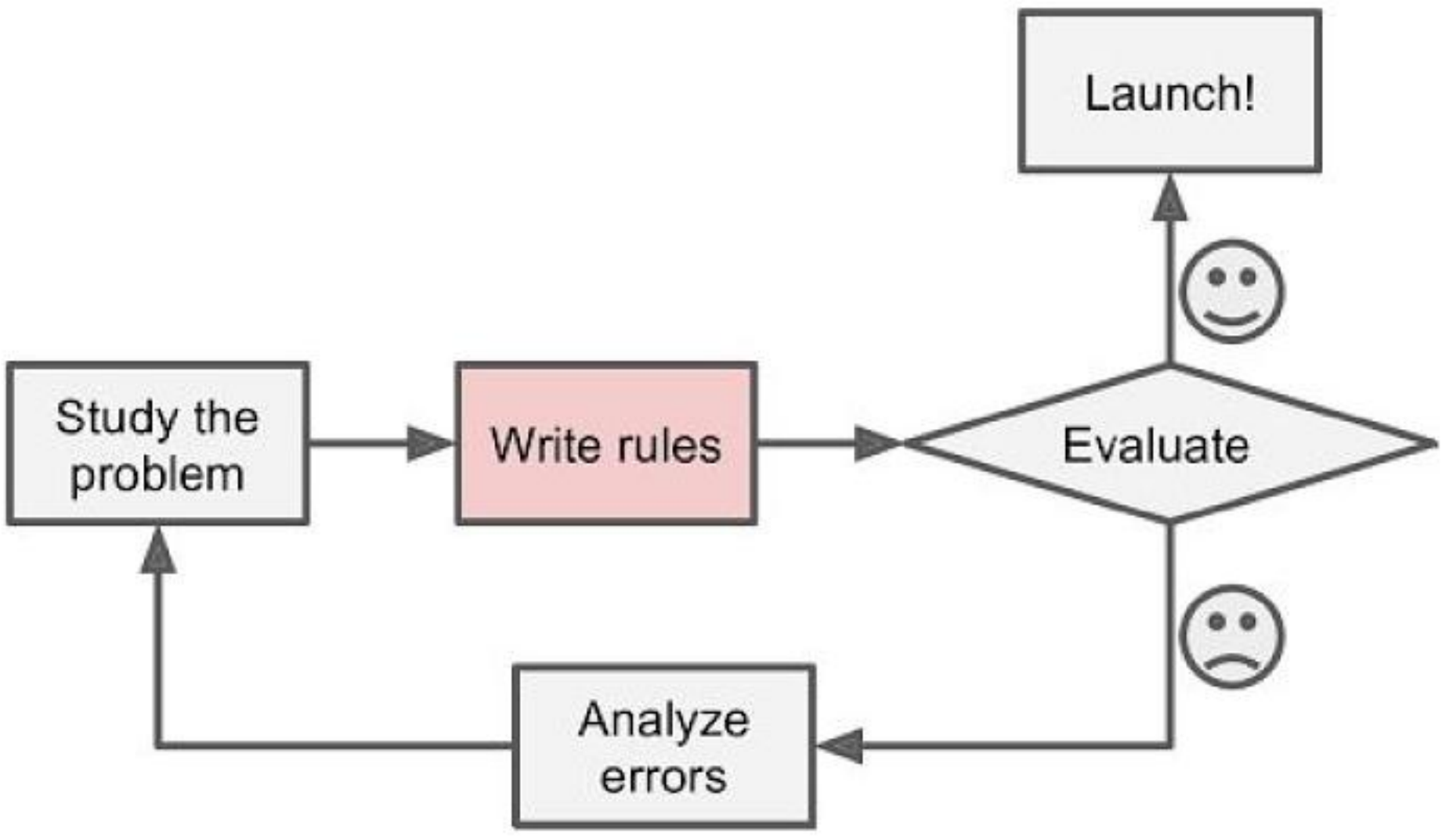


Figure 1-1. The traditional approach

Since the problem is not trivial, 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.

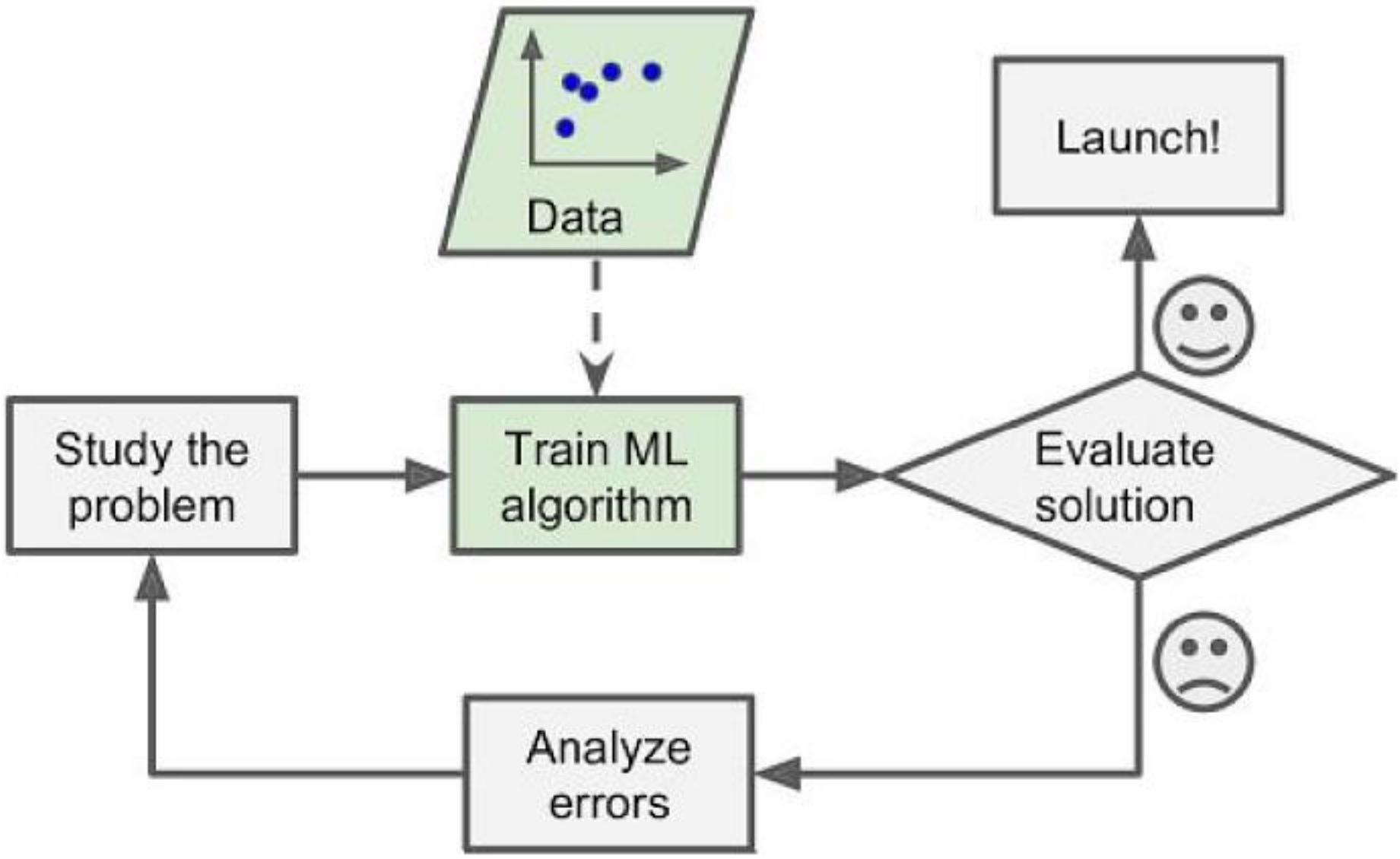


Figure 1-2. Machine Learning approach

Moreover, 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).

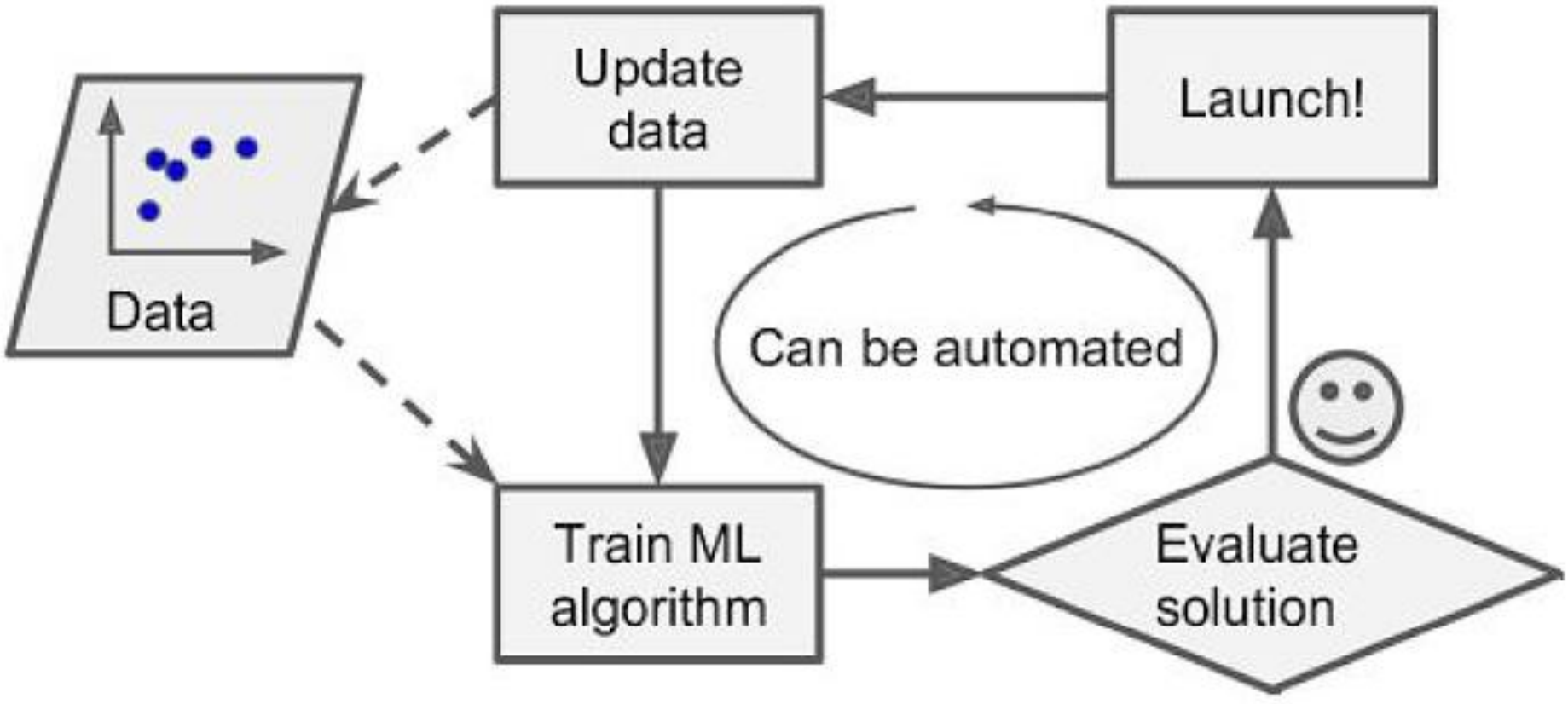


Figure 1-3. Automatically adapting to change

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. 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 algorithms can be inspected to see what they have learned (although for some algorithms this can be tricky). For instance, once the 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.

Applying ML techniques to dig into large amounts of data can help discover patterns that were not immediately apparent. This is called *data mining*.

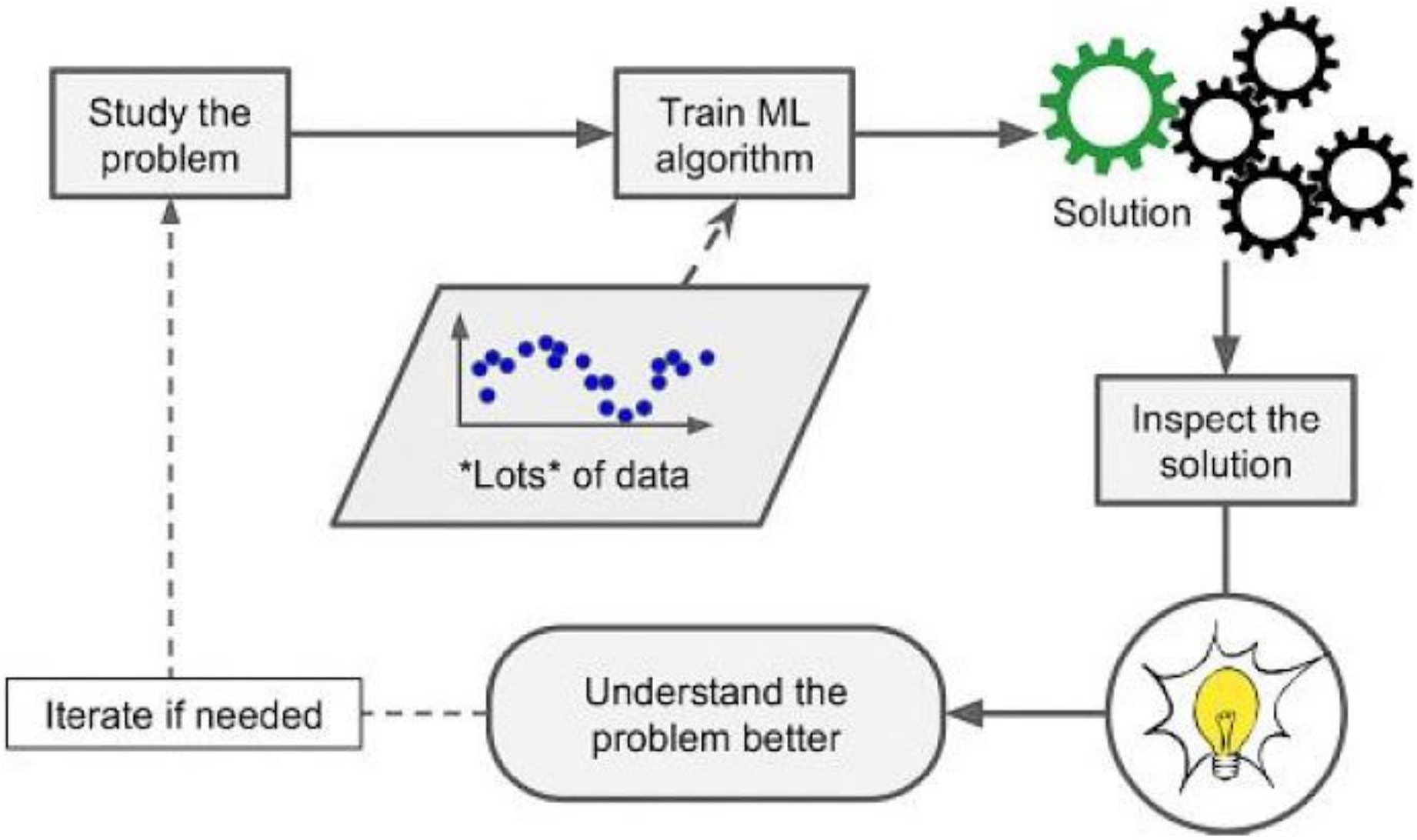


Figure 1-4. Machine Learning can help humans learn

To summarize, Machine Learning is great for:

- Problems for which existing solutions require a lot of hand-tuning or long lists of rules: one Machine Learning algorithm can often simplify code and perform better.
- Complex problems for which there is no good solution at all using a traditional approach: the best Machine Learning techniques can find a solution.
- Fluctuating environments: a Machine Learning system can adapt to new

data.

- Getting insights about complex problems and large amounts of data.

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:

- Whether or not they are trained with human supervision (supervised, unsupervised, semisupervised, and Reinforcement Learning)
- 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 detect patterns in the training data and build 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 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.

Supervised/Unsupervised Learning

Machine Learning systems can be classified according to the amount and type of supervision they get during training. There are four major categories: supervised learning, unsupervised learning, semisupervised learning, and Reinforcement Learning.

Supervised learning

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

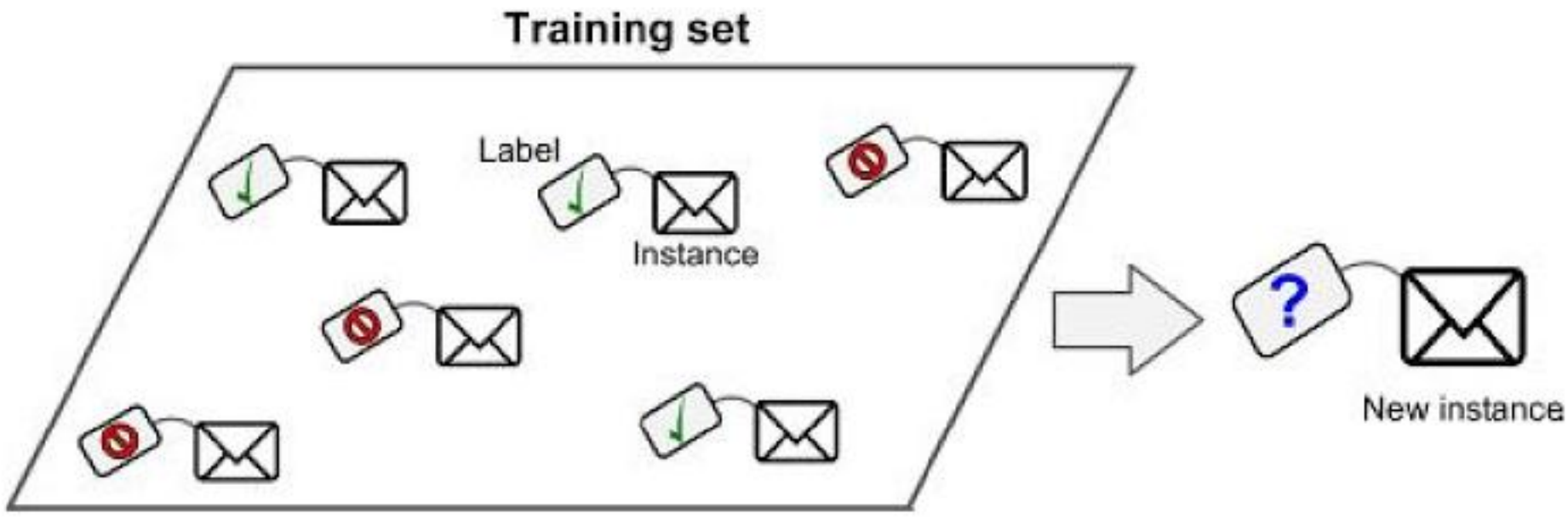


Figure 1-5. A labeled training set for supervised learning (e.g., spam classification)

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.) called *predictors*. 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 predictors and their labels (i.e., their prices).

Note

In Machine Learning an *attribute* is a data type (e.g., “Mileage”), while a *feature* has several meanings depending on the context, but generally means an attribute plus its value (e.g., “Mileage = 15,000”). Many people use the words *attribute* and *feature* interchangeably, though.

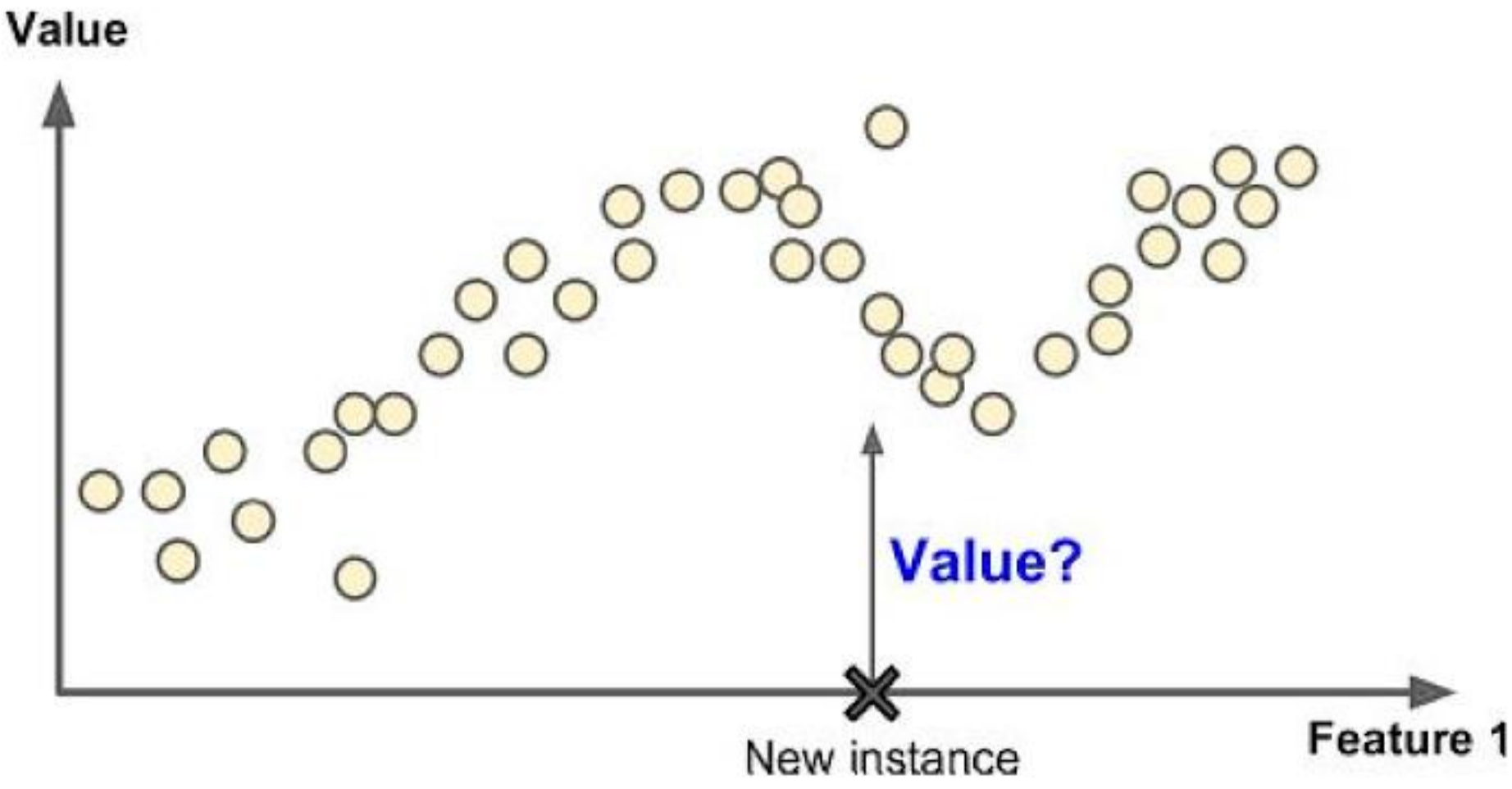


Figure 1-6. Regression

Note that some regression algorithms 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).

Here are some of the most important supervised learning algorithms (covered in this book):

- k-Nearest Neighbors
- Linear Regression
- Logistic Regression
- Support Vector Machines (SVMs)
- Decision Trees and Random Forests
- Neural networks[2]

Unsupervised learning

In *unsupervised learning*, as you might guess, the training data is unlabeled (Figure 1-7). The system tries to learn without a teacher.

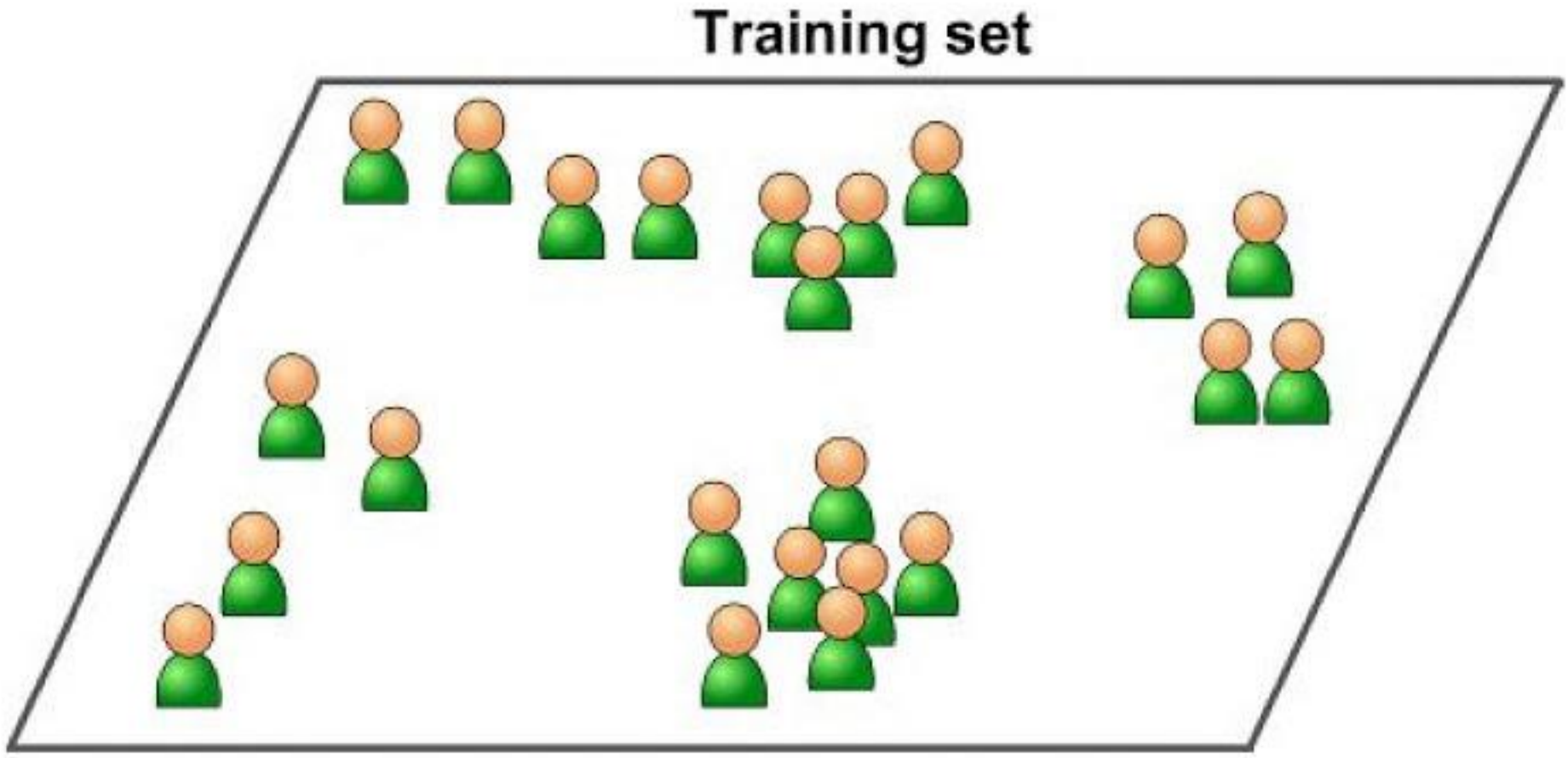


Figure 1-7. An unlabeled training set for unsupervised learning

Here are some of the most important unsupervised learning algorithms (we will cover dimensionality reduction in Chapter 8):

- Clustering
  - k-Means
  - Hierarchical Cluster Analysis (HCA)
  - Expectation Maximization
- Visualization and dimensionality reduction
  - Principal Component Analysis (PCA)
  - Kernel PCA
  - Locally-Linear Embedding (LLE)

    - t-distributed Stochastic Neighbor Embedding (t-SNE)
- Association rule learning
    - Apriori
    - Eclat

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 1-8). 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 males who love comic books and generally read your blog in the evening, while 20% are young sci-fi lovers who visit during the weekends, and so on. 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.

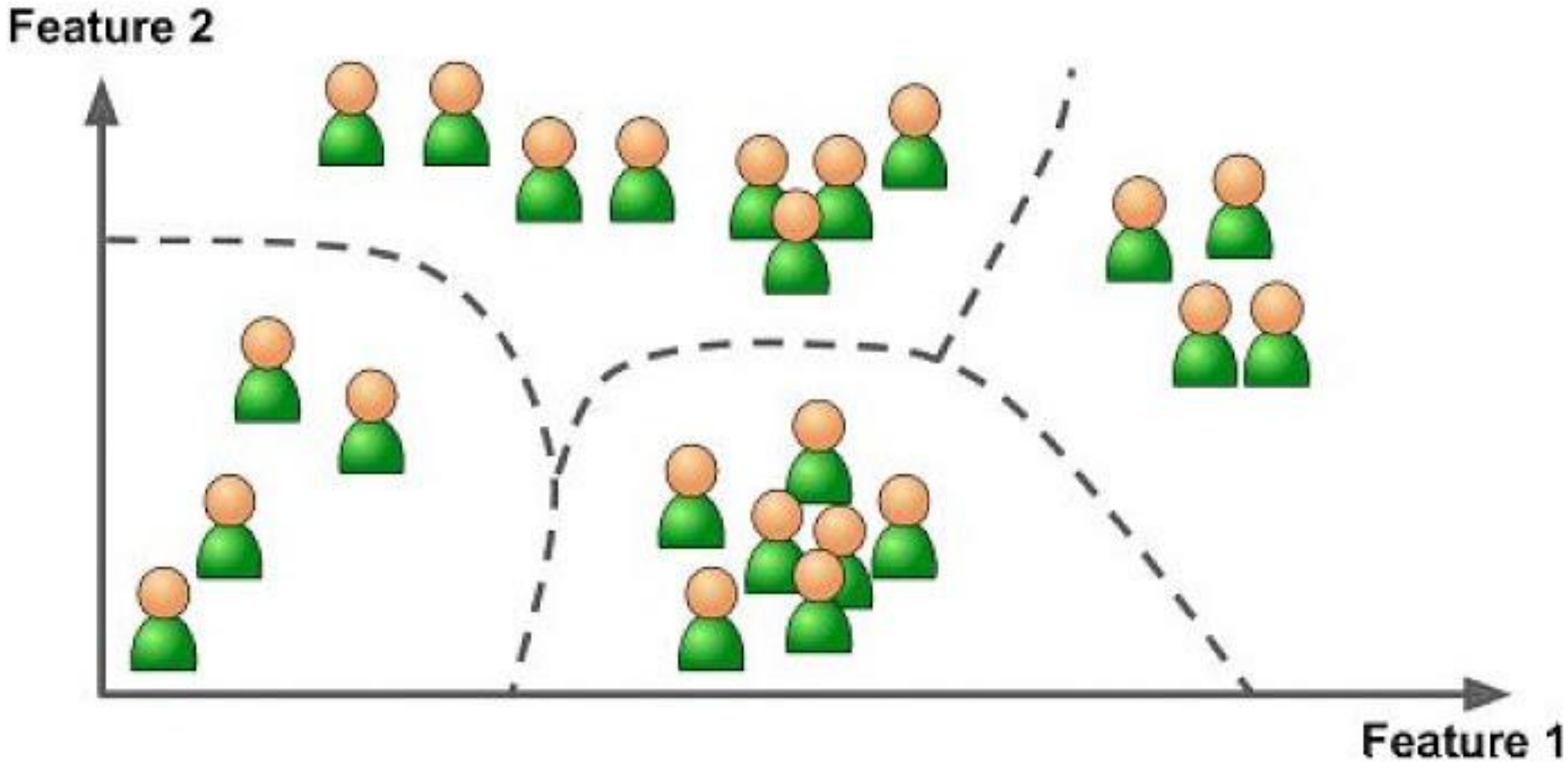


Figure 1-8. Clustering

Visualization algorithms are also good examples of unsupervised learning algorithms: 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 1-9). 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 you can understand how the data is organized and perhaps identify unsuspected patterns.

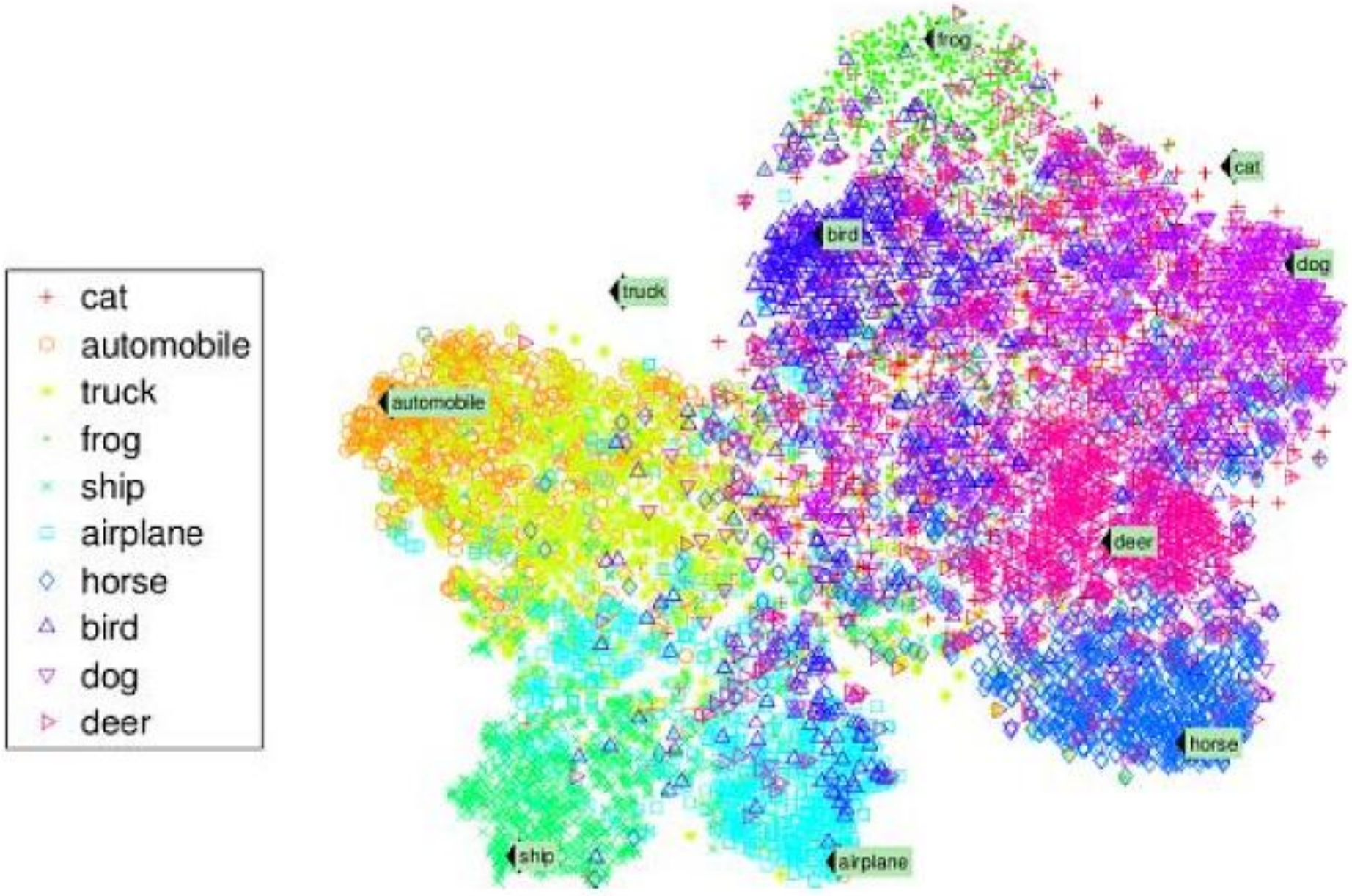


Figure 1-9. Example of a t-SNE visualization highlighting semantic clusters[3]

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 very 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*.

Tip

It is often a good idea to try to reduce the dimension of your training data

using a dimensionality reduction algorithm before you feed it to another Machine Learning algorithm (such as a supervised learning algorithm). It will run much faster, the data will take up less disk and memory space, and in some cases it may also perform better.

Yet another important unsupervised task is *anomaly detection*—for example, detecting unusual credit card transactions to prevent fraud, catching manufacturing defects, or automatically removing outliers from a dataset before feeding it to another learning algorithm. The system is trained with normal instances, and 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 1-10).

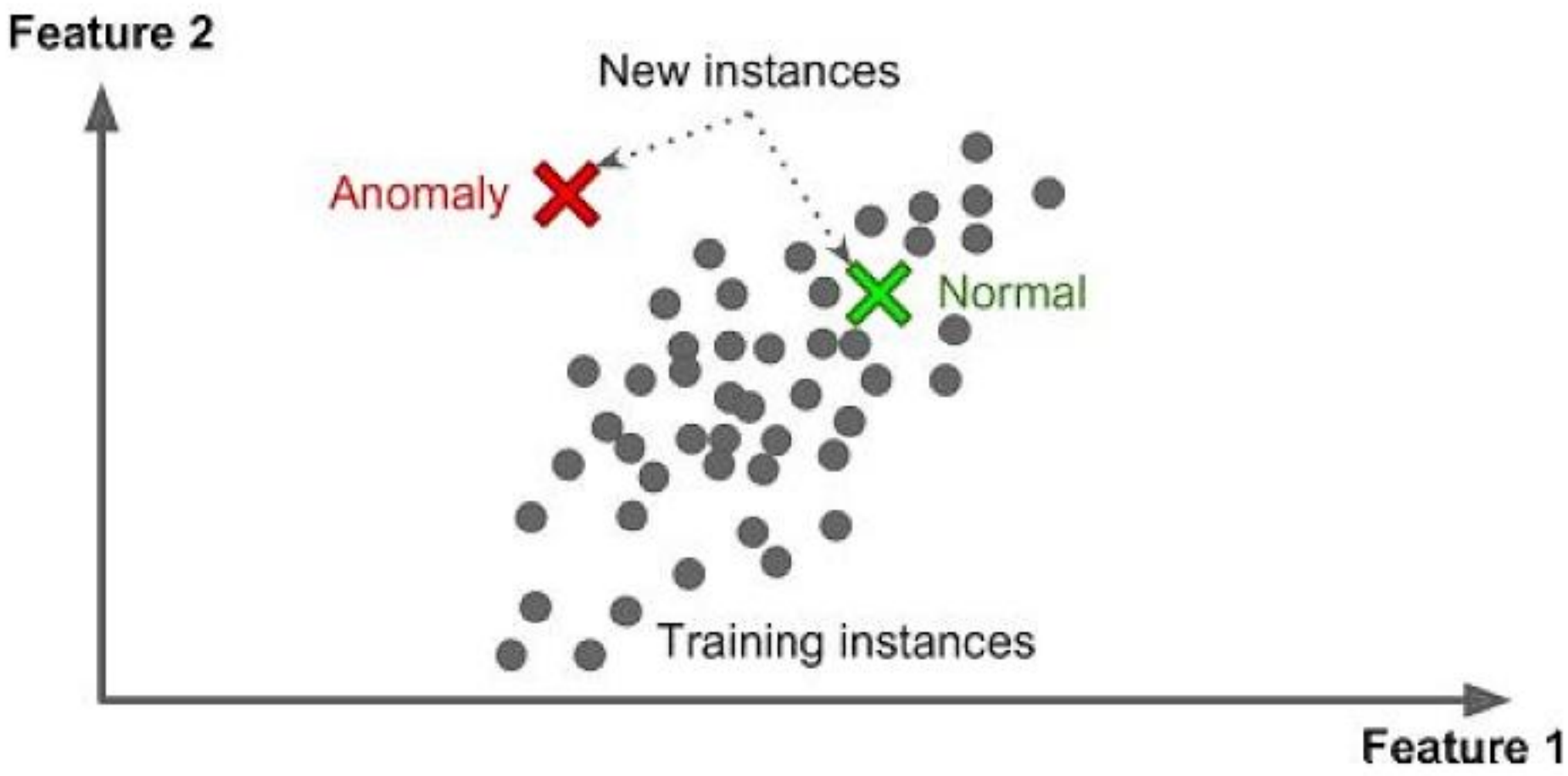


Figure 1-10. Anomaly detection

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 each other.

Semisupervised learning

Some algorithms can deal with partially labeled training data, usually a lot of unlabeled data and a little bit of labeled data. This is called *semisupervised learning* (Figure 1-11).

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 one label per person,[4] and it is able to name everyone in every photo, which is useful for searching photos.

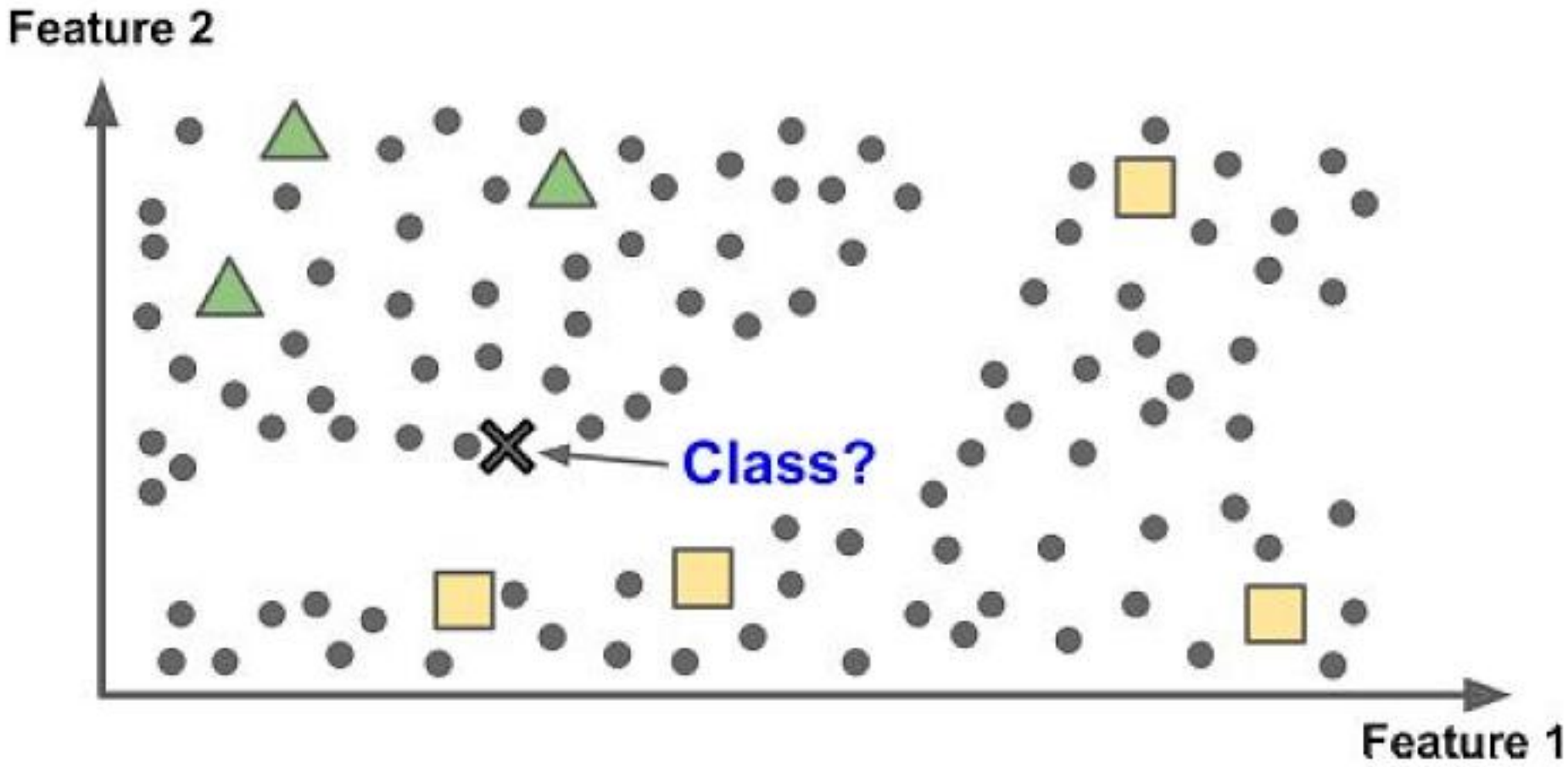


Figure 1-11. Semisupervised learning

Most semisupervised learning algorithms are combinations of unsupervised and supervised algorithms. For example, *deep belief networks* (DBNs) are based on unsupervised components called *restricted Boltzmann machines* (RBMs) stacked on top of one another. RBMs are trained sequentially in an unsupervised manner, and then the whole system is fine-tuned using supervised learning techniques.

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 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.

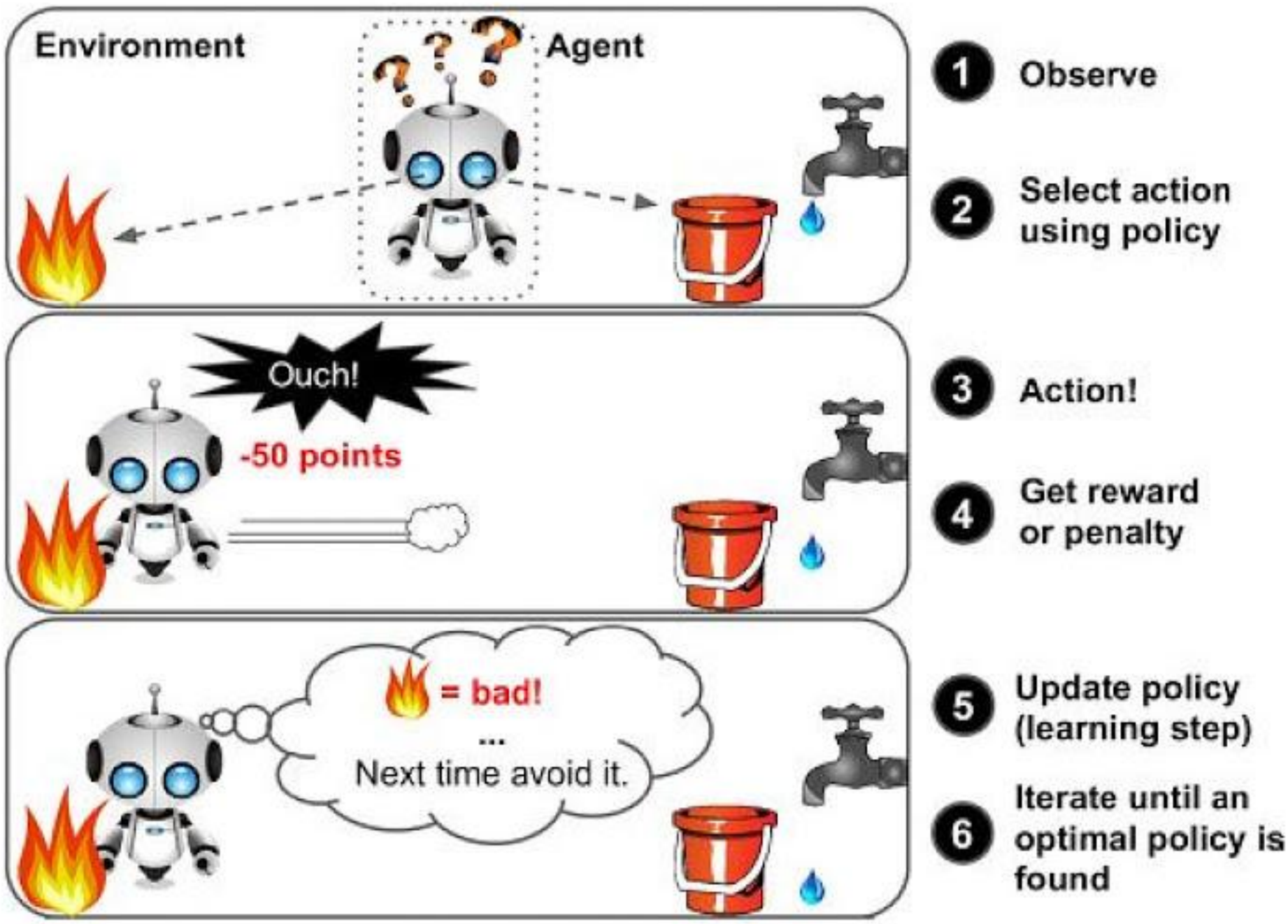


Figure 1-12. Reinforcement Learning

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 the world champion Ke Jie 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.

Batch and Online Learning

Another criterion used to classify Machine Learning systems is whether or not the system can learn incrementally from a stream of incoming data.

Batch learning

In *batch learning*, the system is incapable of learning incrementally: it 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*.

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 stop the old system and replace it with the new one.

Fortunately, the whole process of training, evaluating, and launching a Machine Learning system can be automated fairly easily (as shown 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, it

may even be impossible to use a batch learning algorithm.

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.

Fortunately, a better option in all these cases 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 by 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).

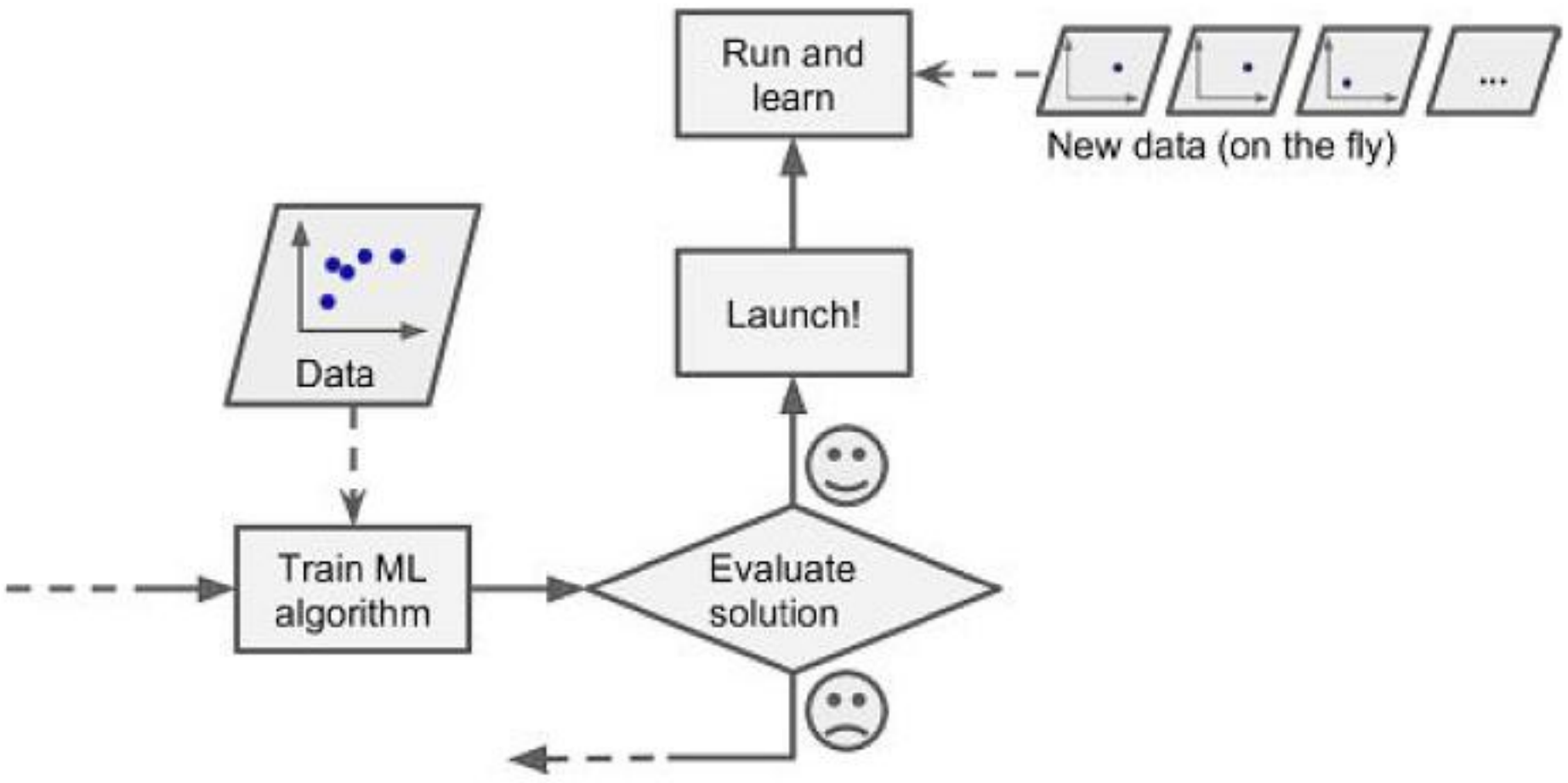


Figure 1-13. Online learning

Online learning is great for systems that receive data as a continuous flow (e.g., stock prices) and need to adapt to change rapidly or autonomously. It is also a good option if you have limited computing resources: once an online learning system has learned about new data instances, it does not

need them anymore, so you can discard them (unless you want to be able to roll back to a previous state and “replay” the data). This can save a huge amount of space.

Online learning algorithms can also be used to train systems on huge datasets that cannot fit in one machine’s main 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).

Warning

This whole process 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*.

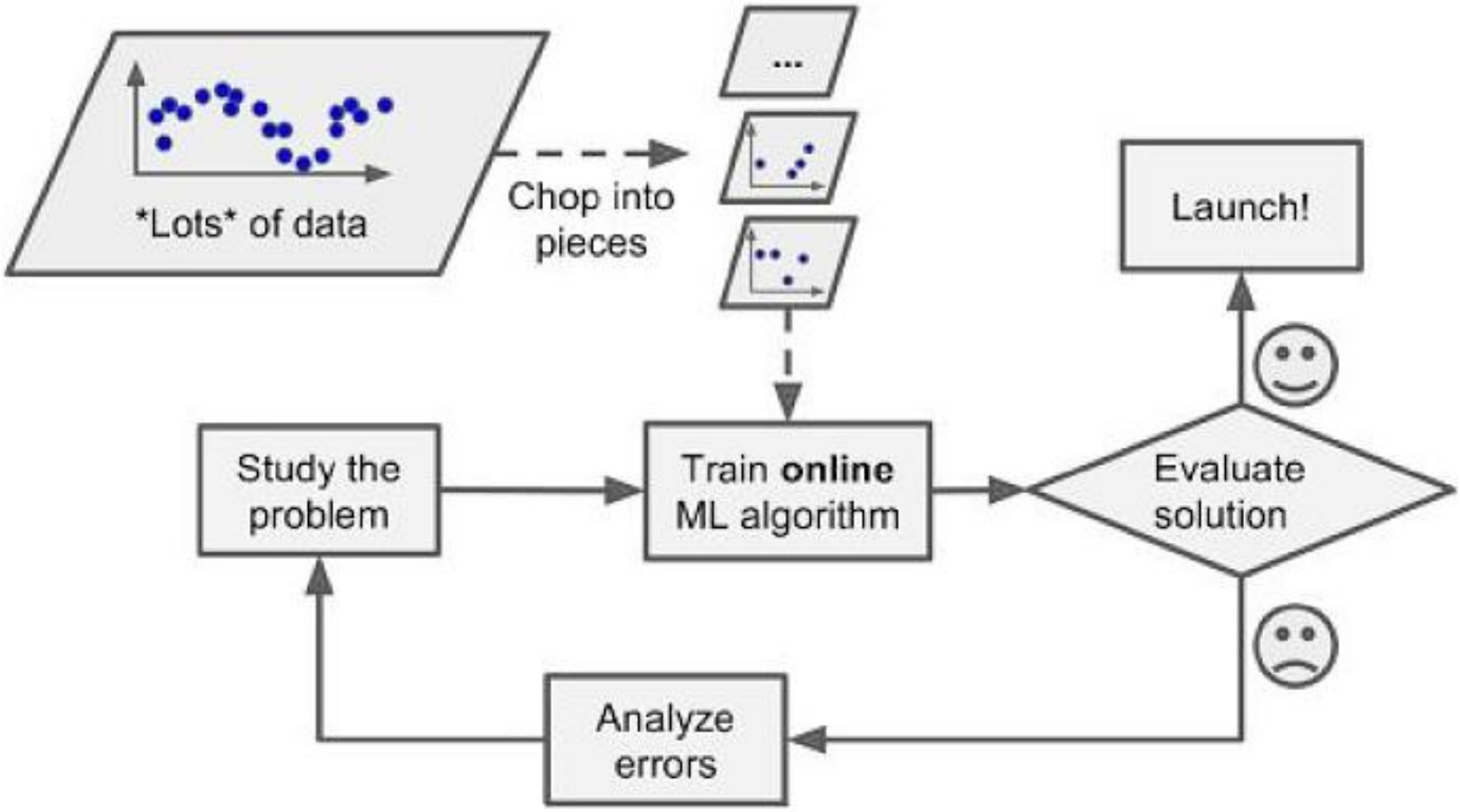


Figure 1-14. Using online learning to handle huge datasets

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 (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.

A big challenge with online learning is that if bad data is fed to the system, the system's performance will gradually decline. If we are talking about a live system, your clients will notice. For example, bad data could come from a malfunctioning sensor on a robot, or from someone 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 (e.g., using an anomaly detection algorithm).

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 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 using a similarity measure (Figure 1-15).

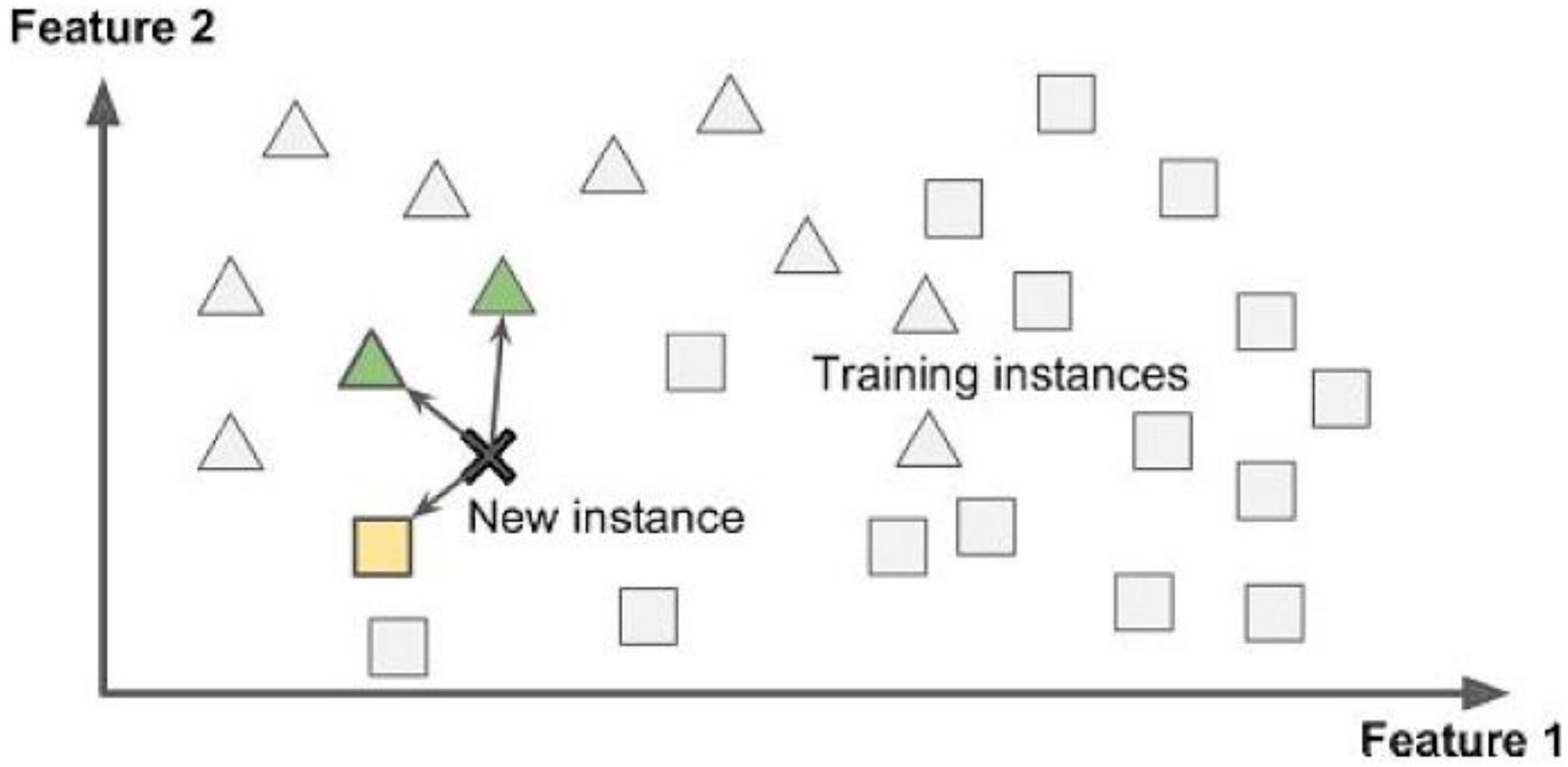


Figure 1-15. Instance-based learning

Model-based learning

Another way to generalize from a set of examples is to build a model of these examples, then use that model to make *predictions*. This is called *model-based learning* (Figure 1-16).

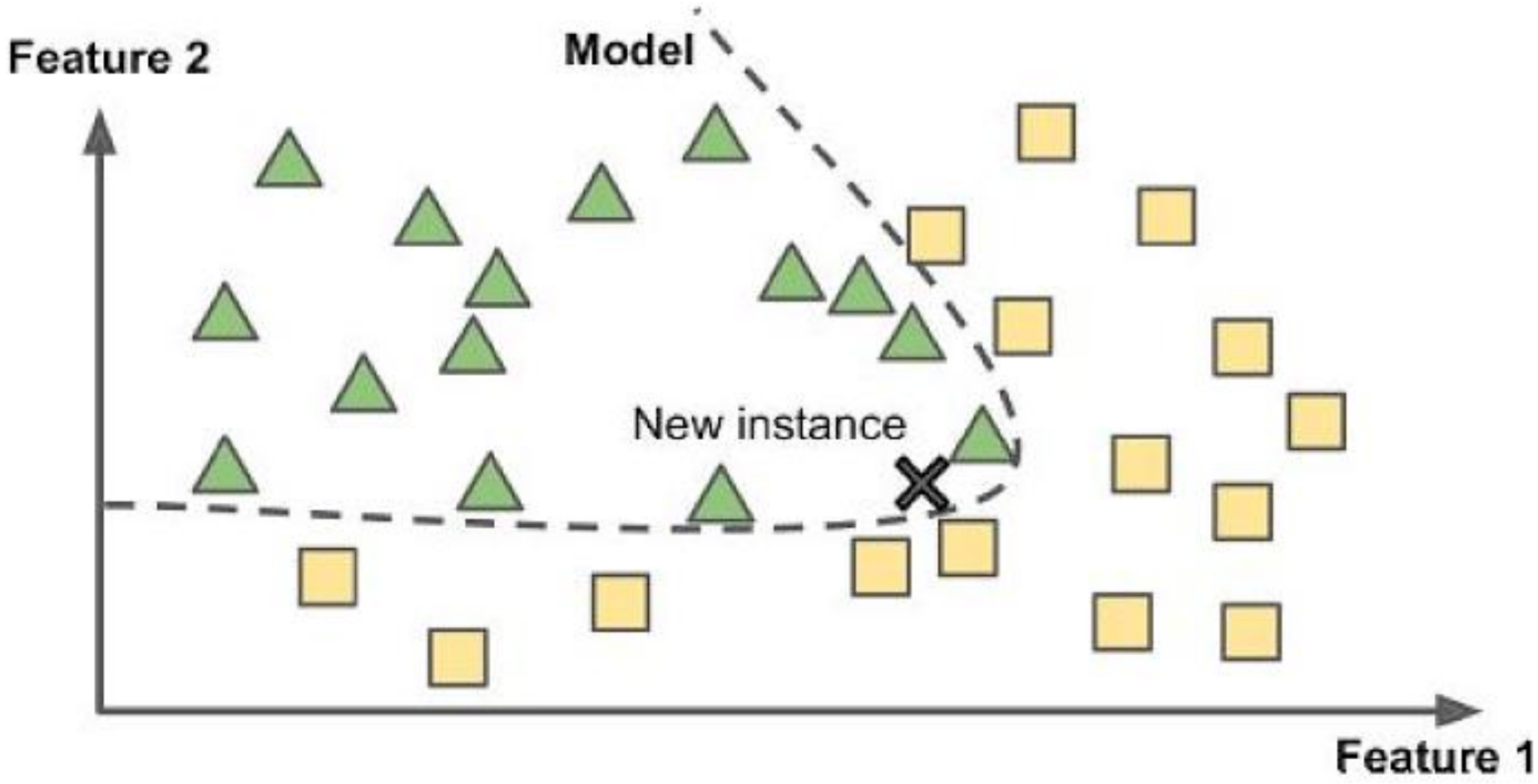


Figure 1-16. Model-based learning

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 as well as stats about GDP per capita from the IMF's website. 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
Hungary	12,240	4.9
Korea	27,195	5.8
France	37,675	6.5
Australia	50,962	7.3
United States	55,805	7.2

Let’s plot the data for a few random countries (Figure 1-17).

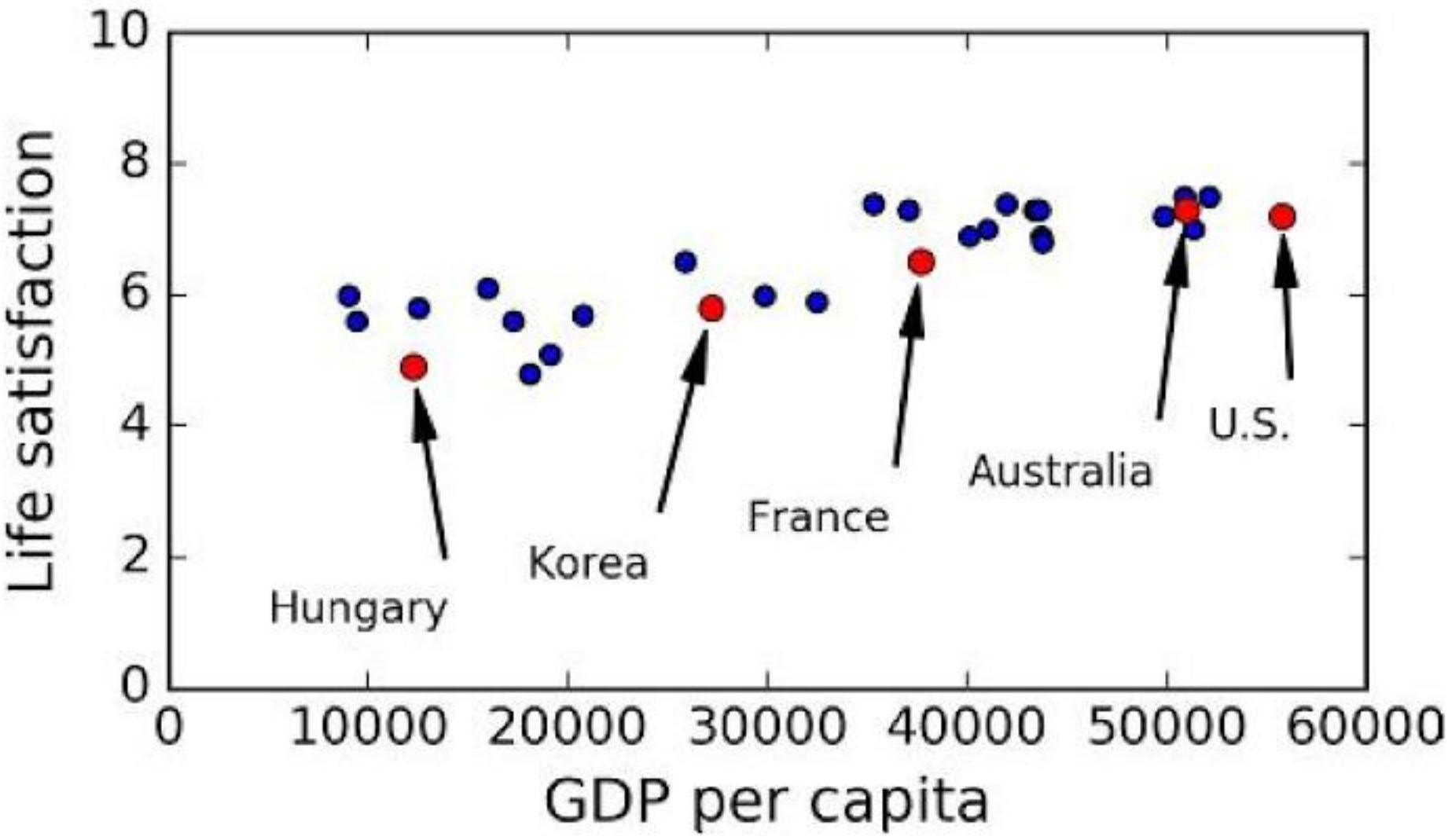


Figure 1-17. Do you see a trend here?

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

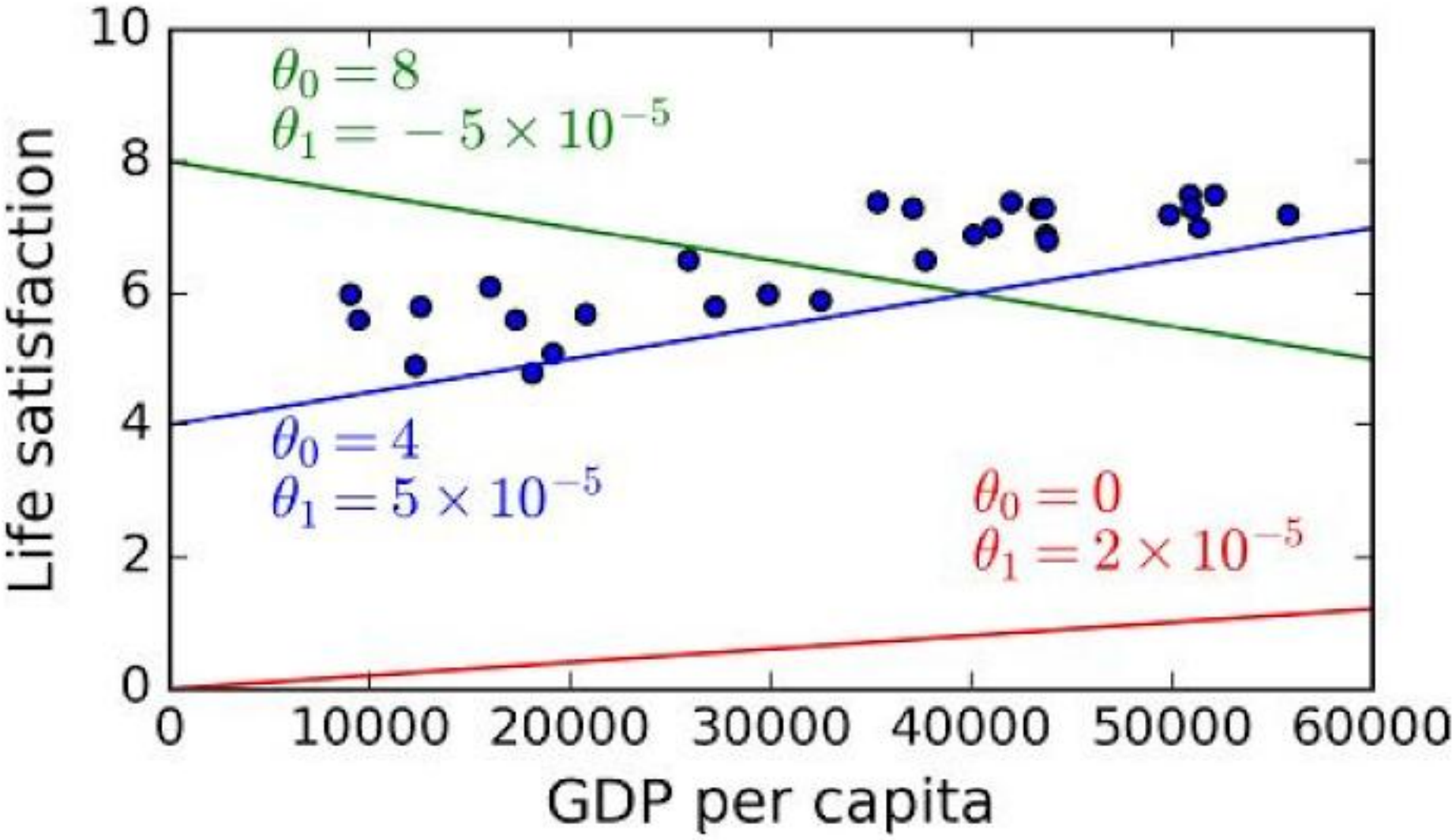


Figure 1-18. A few possible linear models

Before you can use your model, you need to define the parameter values 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* 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 $\theta_0 = 4.85$ and $\theta_1 = 4.91 \times 10^{-5}$.

Now the model fits the training data as closely as possible (for a linear

model), as you can see in Figure 1-19.

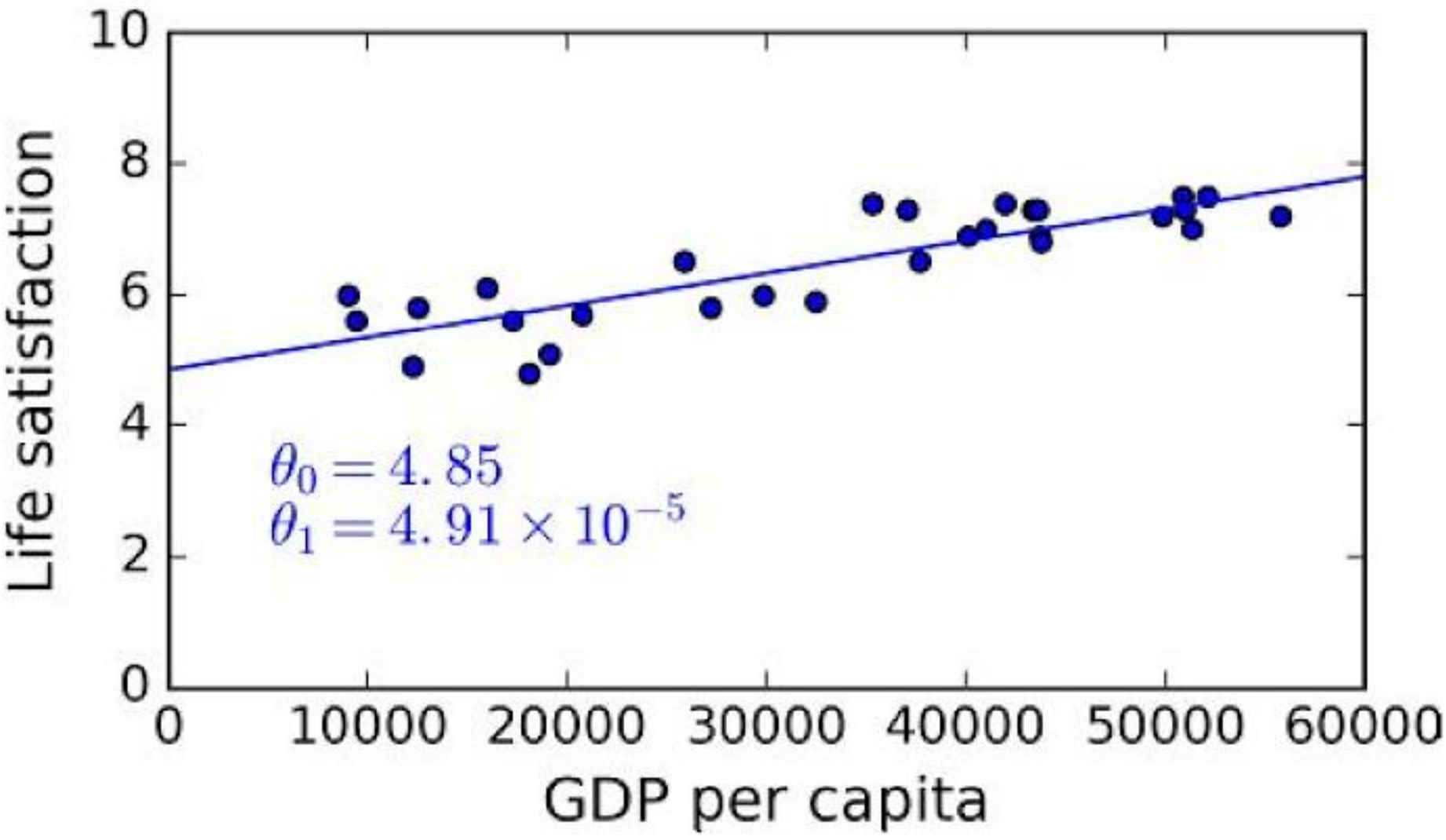


Figure 1-19. The linear model that fits the training data best

You are finally ready to run the model to make predictions. For example, say you want to know how happy Cypriots are, and the OECD data does not have the answer. Fortunately, you can use your model to make a good prediction: you look up Cyprus's GDP per capita, find $22,587, and then apply your model and find that life satisfaction is likely to be somewhere around $4.85 + 22{,}587 \times 4.91 \times 10^{-5} = 5.96$.

To whet your appetite, Example 1-1 shows the Python code that loads the data, prepares it,[6] creates a scatterplot for visualization, and then trains a linear model and makes a prediction.[7]

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

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

```
import pandas as pd
import sklearn


oecd_bli = pd.read_csv("oecd_bli_2015.csv", thousands=',')
gdp_per_capita =
pd.read_csv("gdp_per_capita.csv",thousands=',',delimiter='\t',
                                encoding='latin1', na_values="n/a")


country_stats = prepare_country_stats(oecd_bli, gdp_per_capita)
X = np.c_[country_stats["GDP per capita"]]
y = np.c_[country_stats["Life satisfaction"]]


country_stats.plot(kind='scatter', x="GDP per capita", y='Life
satisfaction')
plt.show()


model = sklearn.linear_model.LinearRegression()


model.fit(X, y)


X_new = [[22587]]
print(model.predict(X_new))
```

Note

If you had used an instance-based learning algorithm instead, you would have found that Slovenia has the closest GDP per capita to that of Cyprus ($20,732), and since the OECD data tells us that Slovenians' life satisfaction is 5.7, you would have predicted a life satisfaction of 5.7 for Cyprus. If you zoom out a bit and look at the two next closest countries, you will find Portugal and Spain with life satisfactions of 5.1 and 6.5, respectively. Averaging these three values, you get 5.77, which is pretty close to 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 simple as replacing this line:

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

with this one:

```
model = sklearn.neighbors.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 first-hand by going through an end-to-end project.

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 learning algorithm and train it on some data, the two things that can go wrong are "bad algorithm" 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).

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[8] once they were given enough data (as you can see in Figure 1-20).

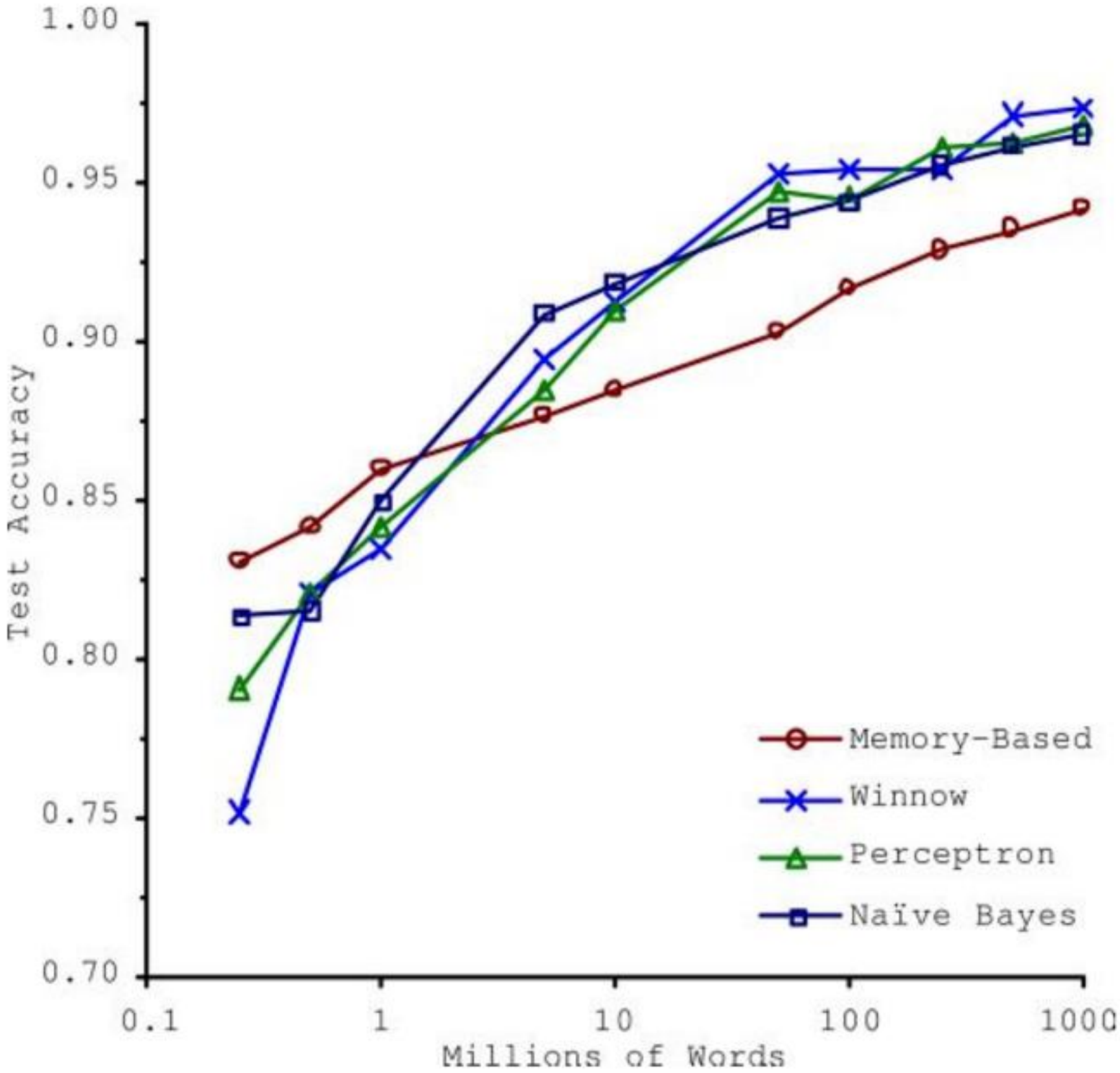


Figure 1-20. The importance of data versus algorithms[9]

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.[10] 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 we used earlier for training the linear model was not perfectly representative; a few countries were missing. Figure 1-21 shows what the data looks like when you add the missing countries.

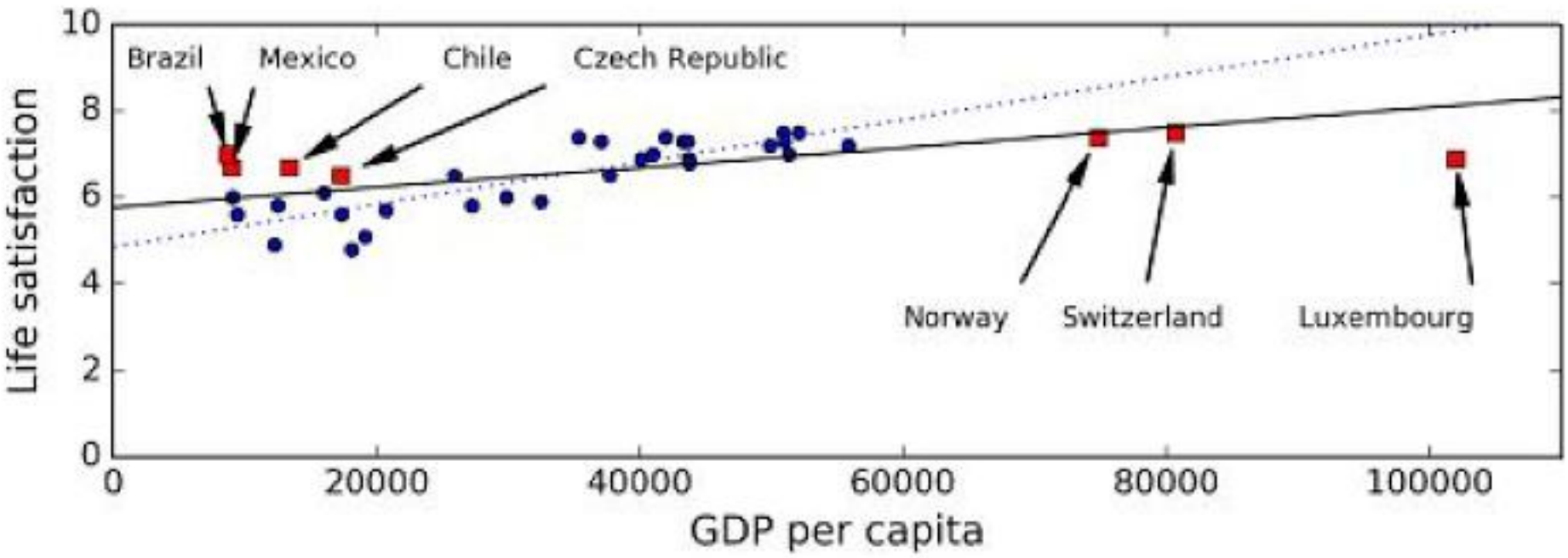


Figure 1-21. A more representative training sample

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 unhappier), and conversely some poor countries seem happier than many rich countries.

By using a nonrepresentative training set, we 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*.

A Famous Example 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 tend to favor wealthier people, who are more likely to vote Republican (hence Landon).
- Second, less than 25% of the people who received the poll answered. Again, this introduces a sampling bias, by ruling out people who don't care much about politics, people who don'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 "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. For example:

- 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, and so on.

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:

- *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 of 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 1-22 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?

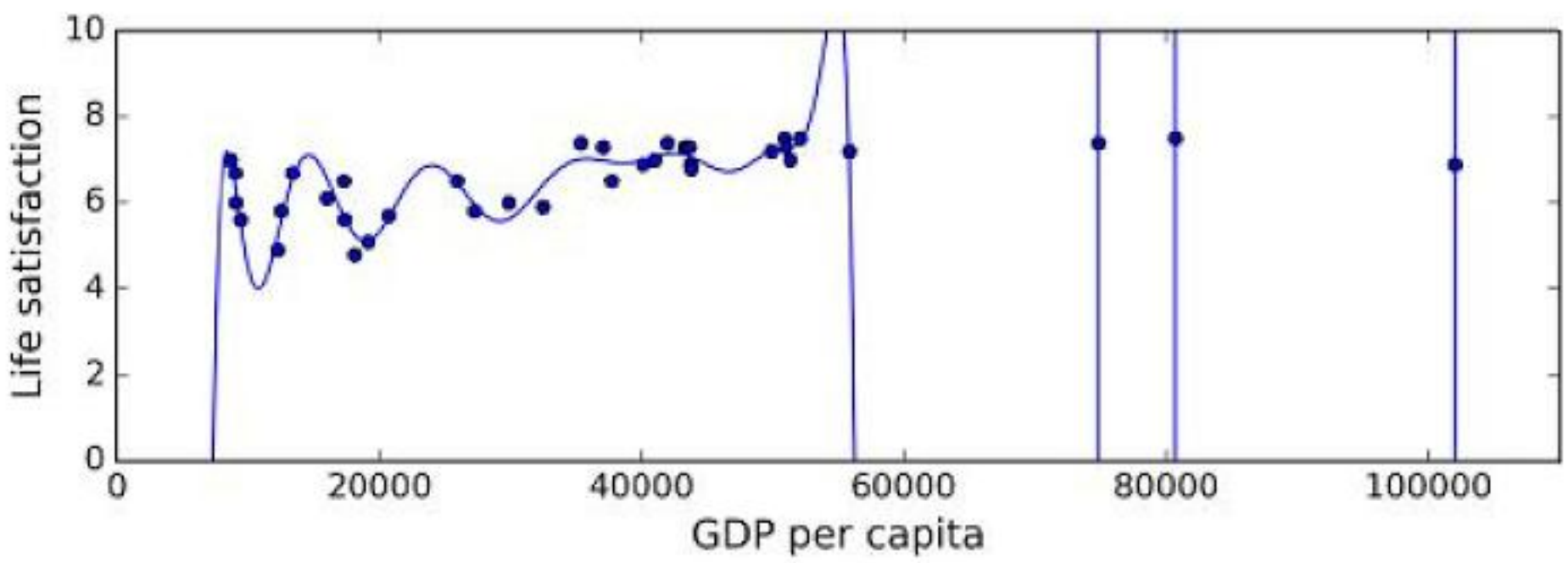


Figure 1-22. Overfitting the training data

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. 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 (7.4), Sweden (7.2), and Switzerland (7.5). How confident are you that the W-satisfaction rule generalizes to Rwanda or Zimbabwe? Obviously this pattern occurred in the training data by pure chance, but the model has no way to tell whether a pattern is real or simply the result of noise in the data.

Warning

Overfitting happens when the model is too complex relative to the amount and noisiness of the training data. The possible solutions are:

- To 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
- To gather more training data
- To 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 simpler model than with two degrees of freedom, but more complex than with just one. You want to find the right balance between fitting the data perfectly and keeping the model simple enough to ensure

that it will generalize well.

Figure 1-23 shows three models: the dotted line represents the original model that was trained with a few countries missing, the dashed line is our second model trained with all countries, and the solid line is a linear 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, which fits a bit less the training data that the model was trained on, but actually allows it to generalize better to new examples.

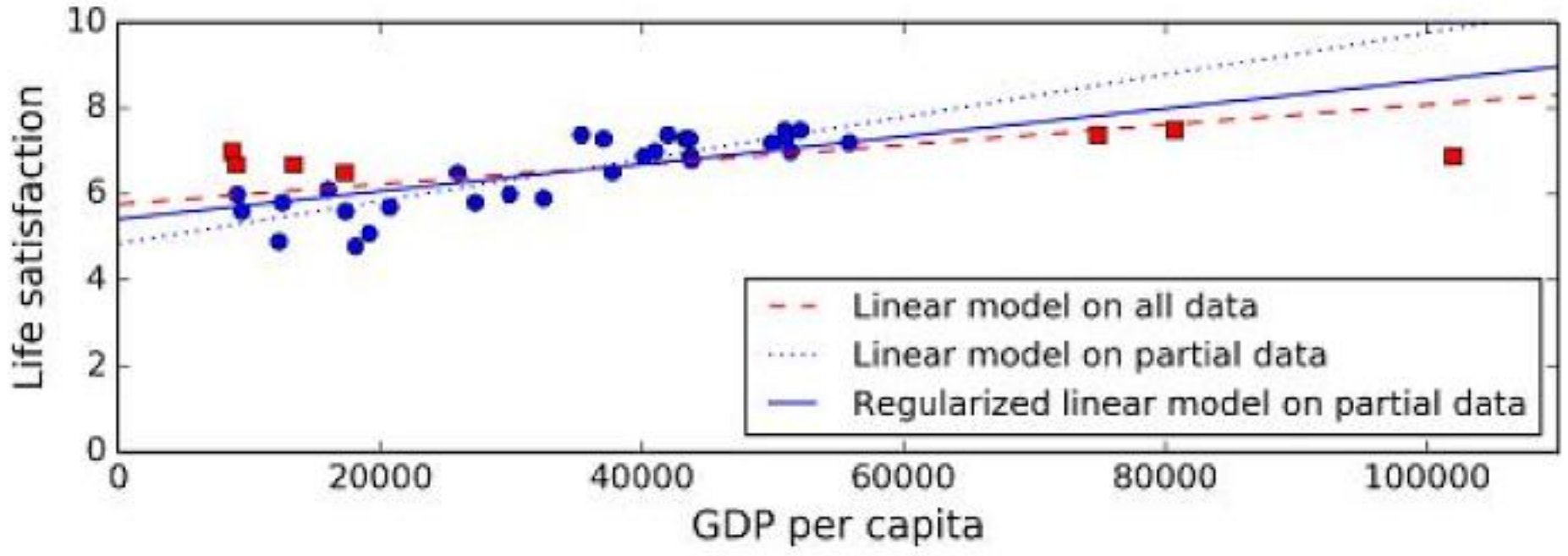


Figure 1-23. Regularization reduces the risk of overfitting

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.

The main options to fix this problem are:

- Selecting a more powerful model, with more parameters
- Feeding better features to the learning algorithm (feature engineering)
- Reducing the constraints on the model (e.g., reducing the regularization hyperparameter)

Stepping Back

By now you already 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, and so on.
- In a 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 uses a similarity measure to generalize to new instances.
- The system will not perform well if your training set is too small, or if the data is not representative, noisy, or polluted with irrelevant features

(garbage in, garbage out). Lastly, your model needs to be neither too simple (in which case it will underfit) nor too complex (in which case it will overfit).

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.

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

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

Tip

It is common to use 80% of the data for training and *hold out* 20% for testing.

So evaluating a model is simple enough: just use a test set. Now suppose you are hesitating between two models (say a linear model and a polynomial model): how can you decide? 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.

So 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 set*. This means that the model is unlikely to perform as well on new data.

A common solution to this problem is to have a second holdout set called the *validation set*. You train multiple models with various hyperparameters using the training set, you select the model and hyperparameters that perform best on the validation set, and when you're happy with your model you run a single final test against the test set to get an estimate of the generalization error.

To avoid "wasting" too much training data in validation sets, a common technique is to use *cross-validation*: the training set is split into complementary subsets, and each model is trained against a different combination of these subsets and validated against the remaining parts. Once the model type and hyperparameters have been selected, a final model is trained using these hyperparameters on the full training set, and the generalized error is measured on the test set.

No Free Lunch Theorem

A model is a simplified version of the observations. The simplifications are meant to discard the superfluous details that are unlikely to generalize to new instances. However, to decide what data to discard and what data to keep, you must make *assumptions*. For example, a linear model makes the assumption 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,[11] 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 you evaluate only a few reasonable models. For example, for simple tasks you may evaluate linear models with various levels of regularization, and for a complex problem you may evaluate various neural networks.

Exercises

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

1. How would you define Machine Learning?
2. Can you name four types of problems 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 Machine Learning 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?
0. What is out-of-core learning?
1. What type of learning algorithm relies on a similarity measure to make predictions?
2. What is the difference between a model parameter and a learning algorithm's hyperparameter?
3. What do model-based learning algorithms search for? What is the most common strategy they use to succeed? How do they make predictions?
4. Can you name four of the main challenges in Machine Learning?
5. If your model performs great on the training data but generalizes poorly to new instances, what is happening? Can you name three possible solutions?
6. What is a test set and why would you want to use it?
7. What is the purpose of a validation set?
8. What can go wrong if you tune hyperparameters using the test set?

9. What is cross-validation and why would you prefer it to a validation set?

Solutions to these exercises are available in Appendix A.

[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 children were shorter, he called this *regression to the mean*. This name was then applied to the methods he used to analyze correlations between variables.

[2] Some neural network architectures can be unsupervised, such as autoencoders and restricted Boltzmann machines. They can also be semisupervised, such as in deep belief networks and unsupervised pretraining.

[3] Notice how animals are rather well separated from vehicles, how horses are close to deer but far from birds, and so on. Figure reproduced with permission from Socher, Ganjoo, Manning, and Ng (2013), "T-SNE visualization of the semantic word space."

[4] 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 need to provide a few labels per person and manually clean up some clusters.

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

[6] The code assumes that `prepare_country_stats()` is already defined: it merges the GDP and life satisfaction data into a single Pandas dataframe.

[7] It's okay if you don't understand all the code yet; we will present Scikit-Learn in the following chapters.

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

[9] Figure reproduced with permission from Banko and Brill (2001), "Learning Curves for Confusion Set Disambiguation."

[10] "The Unreasonable Effectiveness of Data," Peter Norvig et al. (2009).

[11] "The Lack of A Priori Distinctions Between Learning Algorithms," D. Wolperts (1996).

Chapter 2. End-to-End Machine Learning Project

In this chapter, you will go through an example project end to end, pretending to be a recently hired data scientist in a real estate company.[1] Here are the main steps you will go through:

1. Look at the big picture.
2. Get the data.
3. Discover 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 actually experiment with real-world data, not just artificial datasets. Fortunately, there are thousands of open datasets to choose from, ranging across all sorts of domains. Here are a few places you can look to get data:

- Popular open data repositories:
  - UC Irvine Machine Learning Repository
  - Kaggle datasets
  - Amazon's AWS datasets
- Meta portals (they list open data repositories):

    - *http://dataportals.org/*
    - *http://opendatamonitor.eu/*
    - *http://quandl.com/*
- Other pages listing many popular open data repositories:
    - Wikipedia's list of Machine Learning datasets
    - Quora.com question
    - Datasets subreddit

In this chapter we chose the California Housing Prices dataset from the StatLib repository[2] (see Figure 2-1). This dataset was based on data from the 1990 California census. It is not exactly recent (you could still afford a nice house in the Bay Area at the time), but it has many qualities for learning, so we will pretend it is recent data. We also added a categorical attribute and removed a few features for teaching purposes.

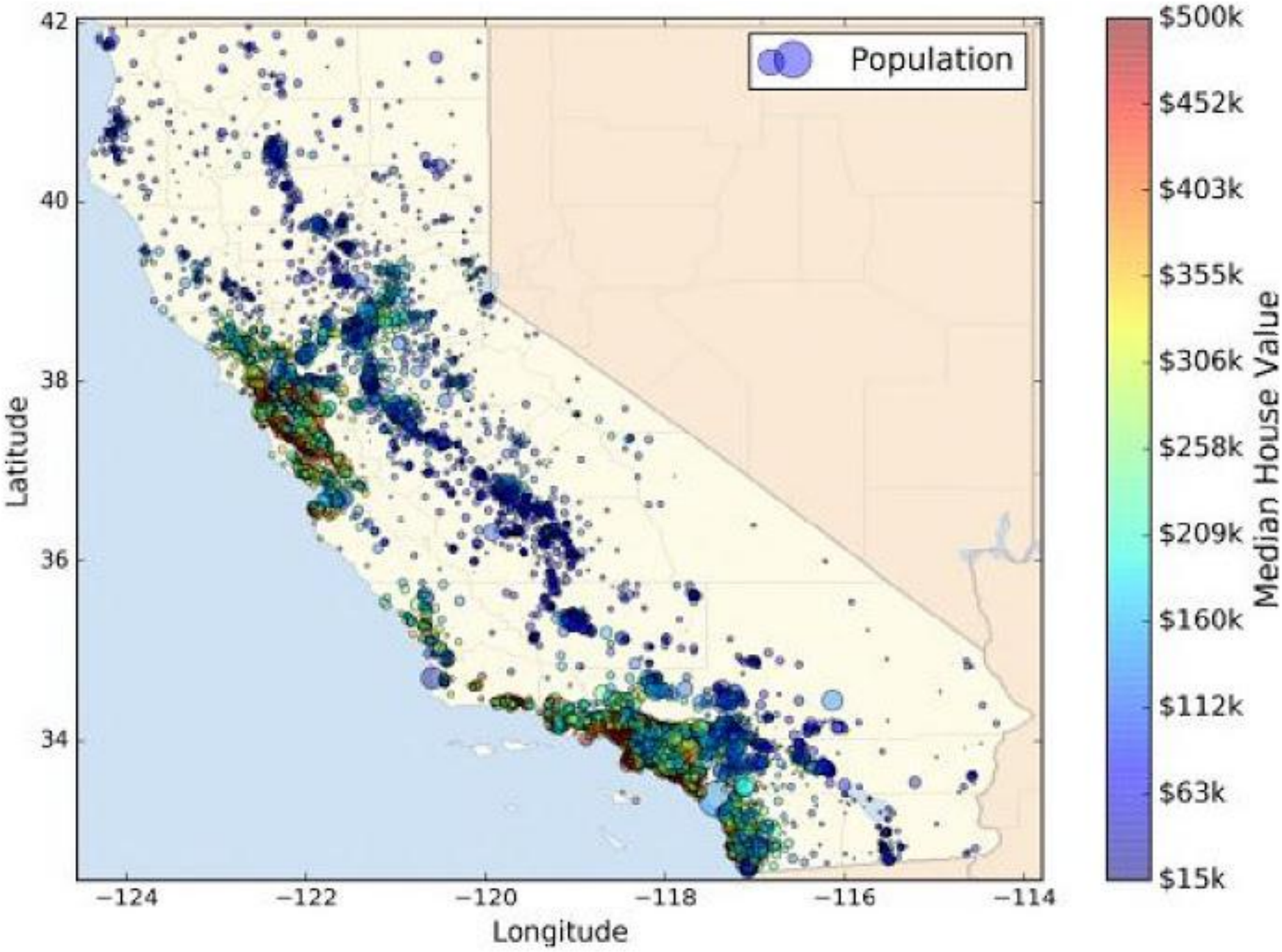


Figure 2-1. California housing prices

Look at the Big Picture

Welcome to Machine Learning Housing Corporation! The first task you are asked to perform is to build a model of housing prices in California using the California census data. This data has metrics such as the population, median income, median housing price, and so on 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). We will just call them “districts” for short.

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

Tip

Since you are a well-organized data scientist, the first thing you do is to pull out your Machine Learning project checklist. You can start with the one in Appendix B; 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 is the business objective; building a model is probably not the end goal. How does the company expect to use and benefit from this model? This is important because it will determine how you frame the problem, what algorithms you will select, what performance measure you will use to evaluate your model, and how much effort you should spend tweaking it.

Your boss answers that your model's output (a prediction of a district's median housing price) will be fed to another Machine Learning system (see Figure 2-2), along with many other *signals*.[3] This downstream system will determine whether it is worth investing in a given area or not. Getting this right is critical, as it directly affects revenue.

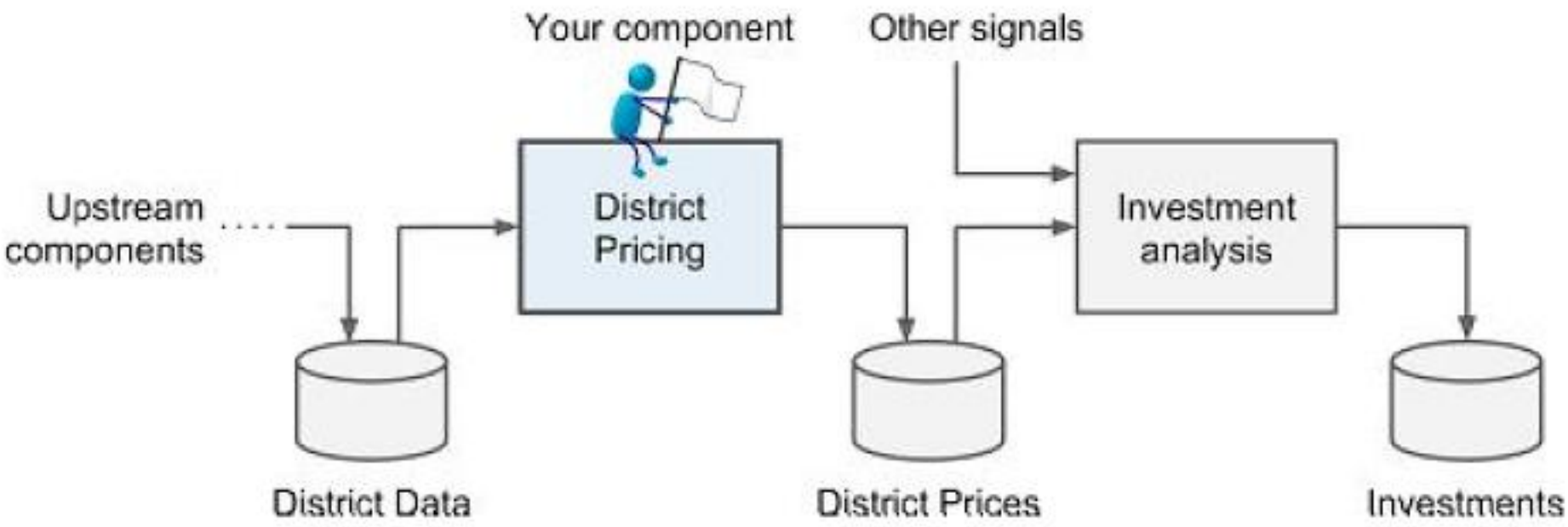


Figure 2-2. A Machine Learning pipeline for real estate investments

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, and then some time later the next component in the pipeline pulls this data and spits out its own output, and so on. Each component is fairly self-contained: the interface between components is simply the data store. This makes the system quite 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.

The next question to ask is what the current solution looks like (if any). It will often give you a reference performance, as well as insights on how to solve the problem. Your boss answers that the district housing prices are currently estimated manually by experts: a team gathers up-to-date information about a district, and when they cannot get the median housing price, they estimate it using complex rules.

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 10%. 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.

Okay, with all this information you are now ready to start designing your system. First, you need to frame the problem: is it supervised, unsupervised, or Reinforcement Learning? 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: it is clearly a typical supervised learning task since you are given *labeled* training examples (each instance comes with the expected output, i.e., the district's median housing price). Moreover, it is also a typical regression task, since you are asked to predict a value. More specifically, this is a *multivariate regression* problem since the system will use multiple features to make a prediction (it will use the district's population, the median income, etc.). In the first chapter, you predicted life satisfaction based on just one feature, the GDP per capita, so it was a *univariate regression* problem. Finally, there is no continuous flow of data coming in the system, there is no particular need to adjust to changing data rapidly, and the data is small enough to fit in memory, so plain batch learning should do just fine.

Tip

If the data was huge, you could either split your batch learning work across multiple servers (using the *MapReduce* technique, as we will see later), or you could use an online learning technique instead.

Select a Performance Measure

Your next step is to select a performance measure. A typical performance measure for regression problems is the Root Mean Square Error (RMSE). It gives an idea of how much error the system typically makes in its

predictions, with a higher weight for large errors. Equation 2-1 shows the mathematical formula to compute the RMSE.

Equation 2-1. Root Mean Square Error (RMSE)

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

Notations

This equation introduces several very common Machine Learning notations that we 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).
    - 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 the other features for now), then:

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

and:

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

- **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 $\mathbf{x}^{(i)}$, noted $(\mathbf{x}^{(i)})^T$.[4]
  - For example, if the first district is as just described, then the matrix **X** looks like this:

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

- *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 $^{(i)} = h(\mathbf{x}^{(i)})$ for that instance (is pronounced "y-hat").
    - For example, if your system predicts that the median housing price in the first district is $158,400, then $^{(1)} = h(\mathbf{x}^{(1)}) =$ 158,400. The prediction error for this district is $^{(1)} - y^{(1)} =$ 2,000.
- RMSE(**X**,*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 **X**).

Even though the RMSE is generally the preferred performance measure for regression tasks, in some contexts you may prefer to use another function. For example, suppose that there are many outlier districts. In that case, you may consider using the *Mean Absolute Error* (also called the Average Absolute Deviation; see Equation 2-2):

Equation 2-2. Mean Absolute Error

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

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

Euclidian norm: it is the notion of distance you are familiar with. It is also called the ℓ_2 *norm*, noted $\| \cdot \|_2$ (or just $\| \cdot \|$).

- Computing the sum of absolutes (MAE) corresponds to the ℓ_1 *norm*, noted $\| \cdot \|_1$. It 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 **v** containing *n* elements is defined as $\| \mathbf{v} \|_k = (| v_0 |^k + | v_1 |^k + \cdots + | v_n |^k)^{\frac{1}{k}}$. ℓ_0 just gives the number of non-zero 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 were made so far (by you or others); this can 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 we assume that these prices are going to be used as such. But what if the downstream system actually 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 following code examples in a Jupyter notebook. The full Jupyter notebook is available at *https://github.com/ageron/handson-ml*.

Create the Workspace

First you will need to have Python installed. It is probably already installed on your system. If not, you can get it at *https://www.python.org/*.[5]

Next you need to create a workspace directory for your Machine Learning code and datasets. Open a terminal and type the following commands (after the `$` prompts):

```
$ export ML_PATH="$HOME/ml"       # You can change the path if you
prefer
$ mkdir -p $ML_PATH
```

You will need a number of Python modules: Jupyter, NumPy, Pandas, Matplotlib, and Scikit-Learn. If you already have Jupyter running with all these modules installed, you can safely skip to "Download the Data". If you don't have them yet, there are many ways to install them (and their dependencies). You can use your system's packaging system (e.g., apt-get on Ubuntu, or MacPorts or HomeBrew on macOS), install a Scientific Python distribution such as Anaconda and use its packaging system, or just use Python's own packaging system, pip, which is included by default with the Python binary installers (since Python 2.7.9).[6] You can check to see if pip is installed by typing the following command:

```
$ pip3 --version
pip 9.0.1 from [...]/lib/python3.5/site-packages (python 3.5)
```

You should make sure you have a recent version of pip installed, at the very least >1.4 to support binary module installation (a.k.a. wheels). To upgrade the pip module, type:[7]

```
$ pip3 install --upgrade pip
Collecting pip
[...]
Successfully installed pip-9.0.1
```

Creating an Isolated Environment

If you would like to work in an isolated environment (which is strongly recommended so you can work on different projects without having conflicting library versions), install virtualenv by running the following pip command:

```
$ pip3 install --user --upgrade virtualenv
Collecting virtualenv
[...]
Successfully installed virtualenv
```

Now you can create an isolated Python environment by typing:

```
$ cd $ML_PATH
$ virtualenv env
Using base prefix '[...]'
New python executable in [...]/ml/env/bin/python3.5
Also creating executable in [...]/ml/env/bin/python
Installing setuptools, pip, wheel...done.
```

Now every time you want to activate this environment, just open a terminal and type:

```
$ cd $ML_PATH
$ source env/bin/activate
```

While the environment is active, any package you install using pip will be installed in this isolated environment, and Python will only have access to these packages (if you also want access to the system's site packages, you should create the environment using virtualenv's `--system-site-packages` option). Check out virtualenv's documentation for more information.

Now you can install all the required modules and their dependencies using this simple pip command:

```
$ pip3 install --upgrade jupyter matplotlib numpy pandas scipy
scikit-learn
Collecting jupyter
  Downloading jupyter-1.0.0-py2.py3-none-any.whl
Collecting matplotlib
  [...]
```

To check your installation, try to import every module like this:

```
$ python3 -c "import jupyter, matplotlib, numpy, pandas, scipy,
sklearn"
```

There should be no output and no error. Now you can fire up Jupyter by typing:

```
$ jupyter notebook
[I 15:24 NotebookApp] Serving notebooks from local directory:
[...]/ml
[I 15:24 NotebookApp] 0 active kernels
[I 15:24 NotebookApp] The Jupyter Notebook is running at:
http://localhost:8888/
[I 15:24 NotebookApp] Use Control-C to stop this server and shut
down all
kernels (twice to skip confirmation).
```

A Jupyter server is now running in your terminal, listening to port 8888. You can visit this server by opening your web browser to *http://localhost:8888/* (this usually happens automatically when the server starts). You should see your empty workspace directory (containing only the *env* directory if you followed the preceding virtualenv instructions).

Now create a new Python notebook by clicking on the New button and

selecting the appropriate Python version[8] (see Figure 2-3).

This does three things: first, it creates a new notebook file called *Untitled.ipynb* in your workspace; second, it starts a Jupyter Python kernel to run this notebook; and third, it opens this notebook in a new tab. You should start by renaming this notebook to "Housing" (this will automatically rename the file to *Housing.ipynb*) by clicking Untitled and typing the new name.

Figure 2-3. Your workspace in Jupyter

A notebook contains a list of cells. Each cell can contain executable code or formatted text. Right now the notebook contains only one empty code cell, labeled "In [1]:". Try typing `print("Hello world!")` in the cell, and click on the play button (see Figure 2-4) or press Shift-Enter. This sends the current cell to this notebook's Python kernel, which runs it and returns the output. The result is displayed below the cell, and since we reached the end of the notebook, a new cell is automatically created. Go through the User Interface Tour from Jupyter's Help menu to learn the basics.

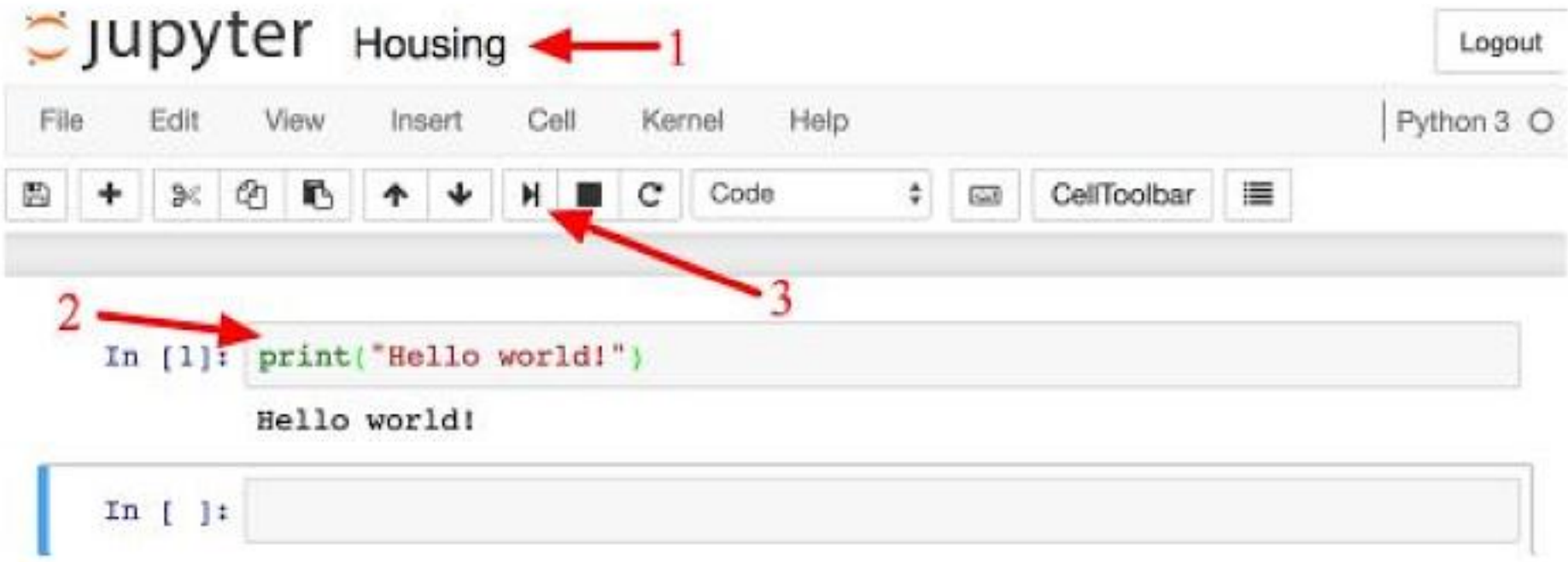


Figure 2-4. Hello world Python notebook

Download the Data

In typical environments your data would be available in a relational database (or some other common datastore) and spread across multiple tables/documents/files. To access it, you would first need to get your credentials and access authorizations,[9] and familiarize yourself with the data schema. In this project, however, things are much simpler: you will just download a single compressed file, *housing.tgz*, which contains a comma-separated value (CSV) file called *housing.csv* with all the data.

You could use your web browser to download it, and run `tar xzf housing.tgz` to decompress the file and extract the CSV file, but it is preferable to create a small function to do that. It is useful in particular if data changes regularly, as it allows you to write a small script that you can run whenever you need 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 the data:[10]

```
import os
import tarfile
```

```
from six.moves import urllib

DOWNLOAD_ROOT = "https://raw.githubusercontent.com/ageron/handson-
ml/master/"
HOUSING_PATH = os.path.join("datasets", "housing")
HOUSING_URL = DOWNLOAD_ROOT + "datasets/housing/housing.tgz"

def fetch_housing_data(housing_url=HOUSING_URL,
housing_path=HOUSING_PATH):
    if not os.path.isdir(housing_path):
        os.makedirs(housing_path)
    tgz_path = os.path.join(housing_path, "housing.tgz")
    urllib.request.urlretrieve(housing_url, tgz_path)
    housing_tgz = tarfile.open(tgz_path)
    housing_tgz.extractall(path=housing_path)
    housing_tgz.close()
```

Now when you call `fetch_housing_data()`, it creates a *datasets/housing* directory in your workspace, downloads the *housing.tgz* file, and extracts the *housing.csv* from it in this directory.

Now let's load the data using Pandas. Once again you should write a small function to load the data:

```
import pandas as pd

def load_housing_data(housing_path=HOUSING_PATH):
    csv_path = os.path.join(housing_path, "housing.csv")
    return pd.read_csv(csv_path)
```

This function returns a Pandas DataFrame object containing all the data.

Take a Quick Look at the Data Structure

Let's take a look at the top five rows using the DataFrame's `head()` method (see Figure 2-5).

```
In [5]: housing = load_housing_data()
        housing.head()
```

Out[5]:

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	populatio
0	-122.23	37.88	41.0	880.0	129.0	322.0
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0
2	-122.24	37.85	52.0	1467.0	190.0	496.0
3	-122.25	37.85	52.0	1274.0	235.0	558.0
4	-122.25	37.85	52.0	1627.0	280.0	565.0

Figure 2-5. Top five rows in the dataset

Each row represents one district. There are 10 attributes (you can see the first 6 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, and each attribute's type and number of non-null values (see Figure 2-6).

```
In [6]: housing.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 10 columns):
longitude             20640 non-null float64
latitude              20640 non-null float64
housing_median_age    20640 non-null float64
total_rooms           20640 non-null float64
total_bedrooms        20433 non-null float64
population            20640 non-null float64
households            20640 non-null float64
median_income         20640 non-null float64
median_house_value    20640 non-null float64
ocean_proximity       20640 non-null object
dtypes: float64(9), object(1)
memory usage: 1.6+ MB
```

Figure 2-6. Housing info

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. Notice that the `total_bedrooms` attribute has only 20,433 non-null values, meaning that 207 districts are missing this feature. We will need to take care of this later.

All attributes are numerical, except the `ocean_proximity` field. 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["ocean_proximity"].value_counts()
<1H OCEAN     9136
INLAND        6551
NEAR OCEAN    2658
NEAR BAY      2290
ISLAND           5
Name: ocean_proximity, dtype: int64
```

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

```
In [8]: housing.describe()
```

Out[8]:

	longitude	latitude	housing_median_age	total_rooms	total_bedr
count	20640.000000	20640.000000	20640.000000	20640.000000	20433.000(
mean	-119.569704	35.631861	28.639486	2635.763081	537.870553
std	2.003532	2.135952	12.585558	2181.615252	421.38507(
min	-124.350000	32.540000	1.000000	2.000000	1.000000
25%	-121.800000	33.930000	18.000000	1447.750000	296.00000(
50%	-118.490000	34.260000	29.000000	2127.000000	435.00000(
75%	-118.010000	37.710000	37.000000	3148.000000	647.00000(
max	-114.310000	41.950000	52.000000	39320.000000	6445.0000(

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, `count` of `total_bedrooms` is 20,433, not 20,640). The `std` row shows the *standard deviation*, which measures how dispersed the values are.[11] 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 falls. 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 1st *quartile*), the median, and the 75th percentile (or 3rd 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, and it will plot a histogram for each numerical attribute (see Figure 2-8). For example, you can see that slightly over 800 districts have a `median_house_value` equal to about $100,000.

```
%matplotlib inline
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

Note

The `hist()` method relies on Matplotlib, which in turn relies on a user-specified graphical backend to draw on your screen. So before you can plot anything, you need to specify which backend Matplotlib should use. The simplest option is to use Jupyter's magic command `%matplotlib inline`. This tells Jupyter to set up Matplotlib so it uses Jupyter's own backend. Plots are then rendered within the notebook itself. Note that calling `show()` is optional in a Jupyter notebook, as Jupyter will automatically display plots when a cell is executed.

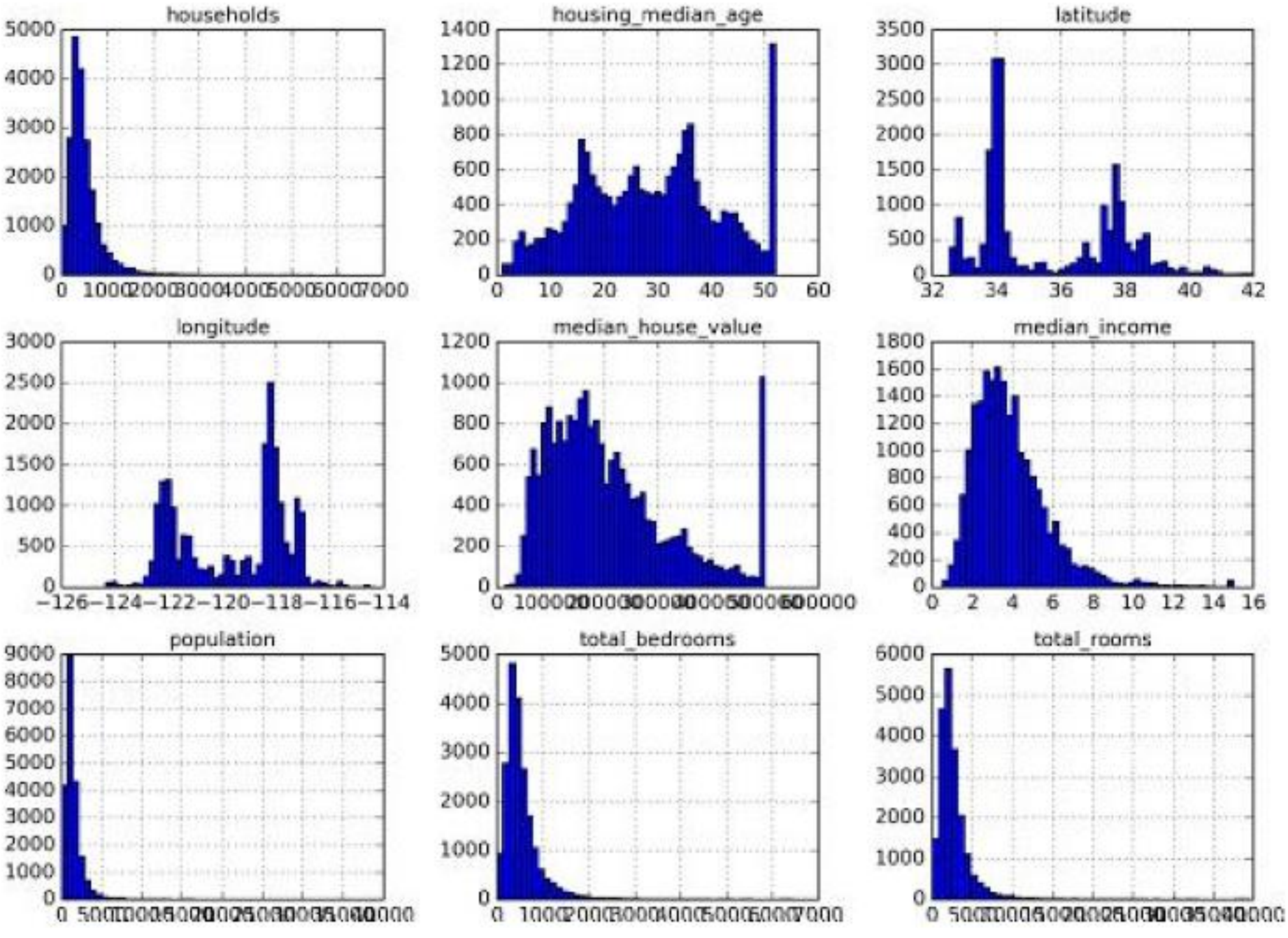


Figure 2-8. A histogram for each numerical attribute

Notice a few things in these histograms:

1. 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. 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.

2. 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 mainly two options:

   a. Collect proper labels for the districts whose labels were capped.

   b. 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).

3. These attributes have very different scales. We will discuss this later in this chapter when we explore feature scaling.

4. Finally, many histograms are *tail heavy*: 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. We will try transforming these attributes later on to have more bell-shaped distributions.

Hopefully you now have a better understanding of the kind of data you are dealing with.
Warning

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

Create a Test Set

It may sound 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 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 quite simple: just pick some instances randomly, typically 20% of the dataset, and set them aside:

```
import numpy as np

def split_train_test(data, test_ratio):
    shuffled_indices = np.random.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:

```
>>> train_set, test_set = split_train_test(housing, 0.2)
>>> print(len(train_set), "train +", len(test_set), "test")
16512 train + 4128 test
```

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., `np.random.seed(42)`)[12] before calling `np.random.permutation()`, so that it always generates the same shuffled indices.

But both these solutions will break next time you fetch an updated dataset. A common solution is to use each instance's identifier to decide whether or not it should go in the test set (assuming instances have a unique and

immutable identifier). For example, you could compute a hash of each instance's identifier, keep only the last byte of the hash, and put the instance in the test set if this value is lower or equal to 51 (~20% of 256). This ensures that the test set will remain consistent across multiple runs, even if you refresh the dataset. The new test set will contain 20% of the new instances, but it will not contain any instance that was previously in the training set. Here is a possible implementation:

```
import hashlib

def test_set_check(identifier, test_ratio, hash):
    return hash(np.int64(identifier)).digest()[-1] < 256 *
test_ratio

def split_train_test_by_id(data, test_ratio, id_column,
hash=hashlib.md5):
    ids = data[id_column]
    in_test_set = ids.apply(lambda id_: test_set_check(id_,
test_ratio, hash))
    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.reset_index()
train_set, test_set = split_train_test_by_id(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 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:[13]

```
housing_with_id["id"] = housing["longitude"] * 1000 +
housing["latitude"]
train_set, test_set = split_train_test_by_id(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 function `split_train_test` defined earlier, with a couple of additional features. First there is a `random_state` parameter that allows you to set the random generator seed as explained previously, and 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, 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 a survey company decides to call 1,000 people to ask them a few questions, they don't just pick 1,000 people randomly in a phone booth. They try to ensure that these 1,000 people are representative of the whole population. For example, the US population is composed of 51.3% female and 48.7% male, so a well-conducted survey in the US would try to maintain this ratio in the sample: 513 female and 487 male. This is called *stratified sampling*: the population is divided into homogeneous subgroups called *strata*, and the right number of instances is sampled from each stratum to guarantee that the test set is representative of the overall population. If they used purely random sampling, there would be about 12% chance of sampling a skewed test set with either less than

49% female or more than 54% female. Either way, the survey results would be significantly biased.

Suppose you chatted with 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 $20,000–$50,000, but some median incomes go far beyond $60,000. It is important to have a sufficient number of instances in your dataset for each stratum, or else the estimate of the 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 creates an income category attribute by dividing the median income by 1.5 (to limit the number of income categories), and rounding up using `ceil` (to have discrete categories), and then merging all the categories greater than 5 into category 5:

```
housing["income_cat"] = np.ceil(housing["median_income"] / 1.5)
housing["income_cat"].where(housing["income_cat"] < 5, 5.0,
inplace=True)
```

These income categories are represented on Figure 2-9):

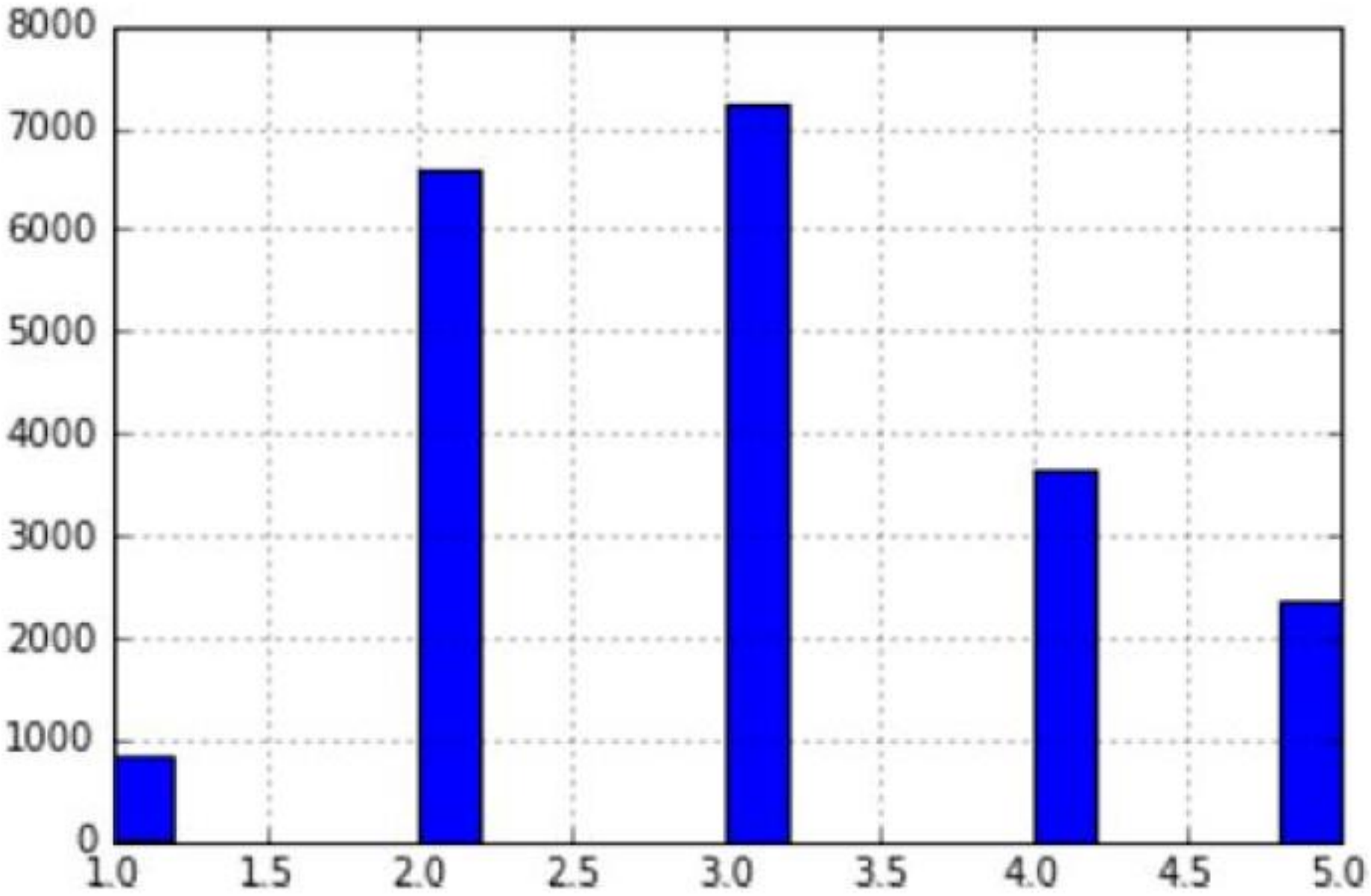


Figure 2-9. Histogram of income categories

Now you are ready to do stratified sampling based on the income category. For this you can use Scikit-Learn's `StratifiedShuffleSplit` class:

```
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2,
random_state=42)
for train_index, test_index in split.split(housing,
housing["income_cat"]):
    strat_train_set = housing.loc[train_index]
    strat_test_set = housing.loc[test_index]
```

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

```
>>> housing["income_cat"].value_counts() / len(housing)
```

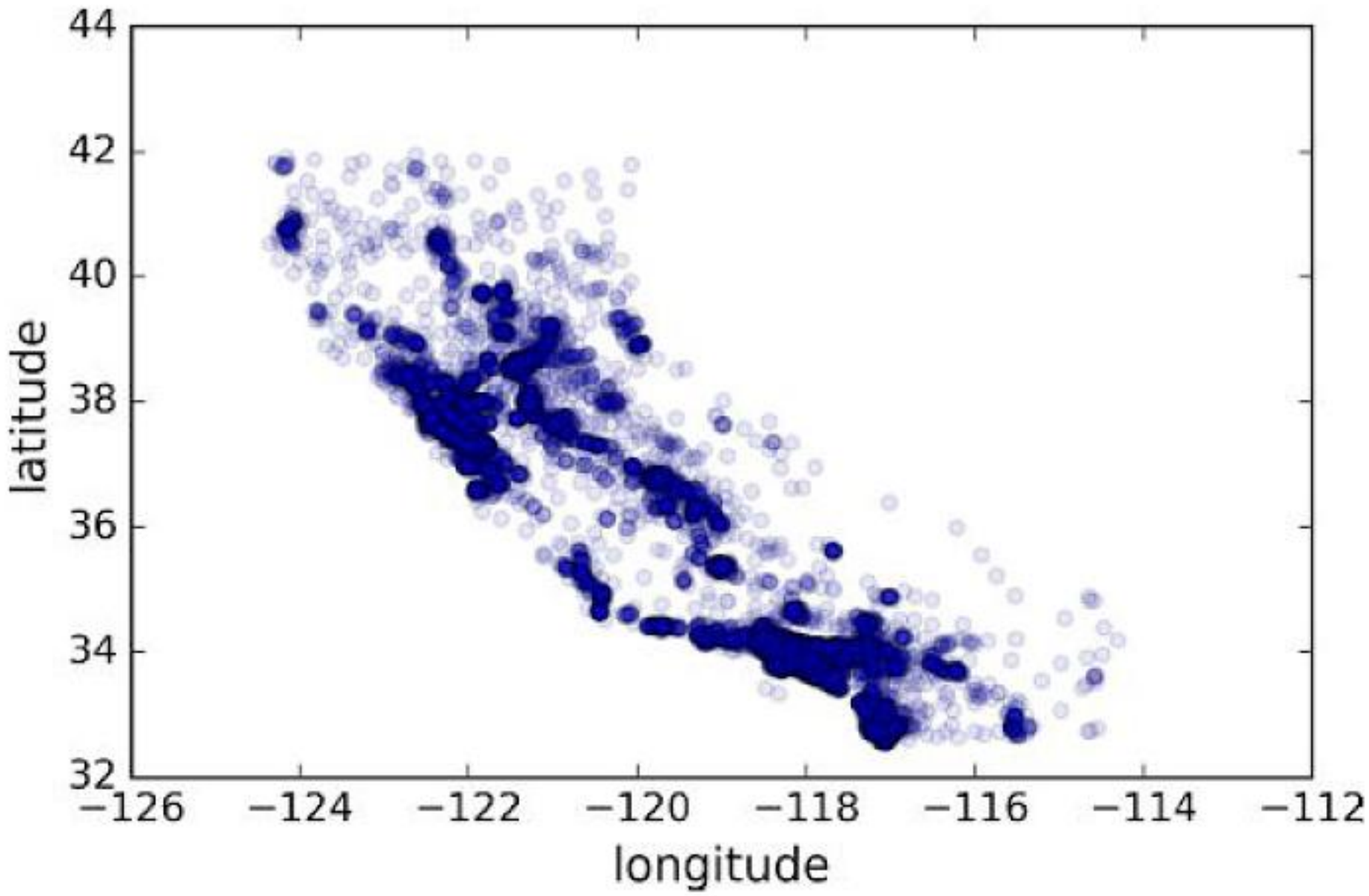


Figure 2-12. A better visualization highlighting high-density areas

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 in the Central Valley, in particular around Sacramento and Fresno.

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

Now let's 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`). We will use a predefined color map (option `cmap`) called `jet`, which ranges from blue (low values) to red (high prices):[14]

```
housing.plot(kind="scatter", x="longitude", y="latitude",
```

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

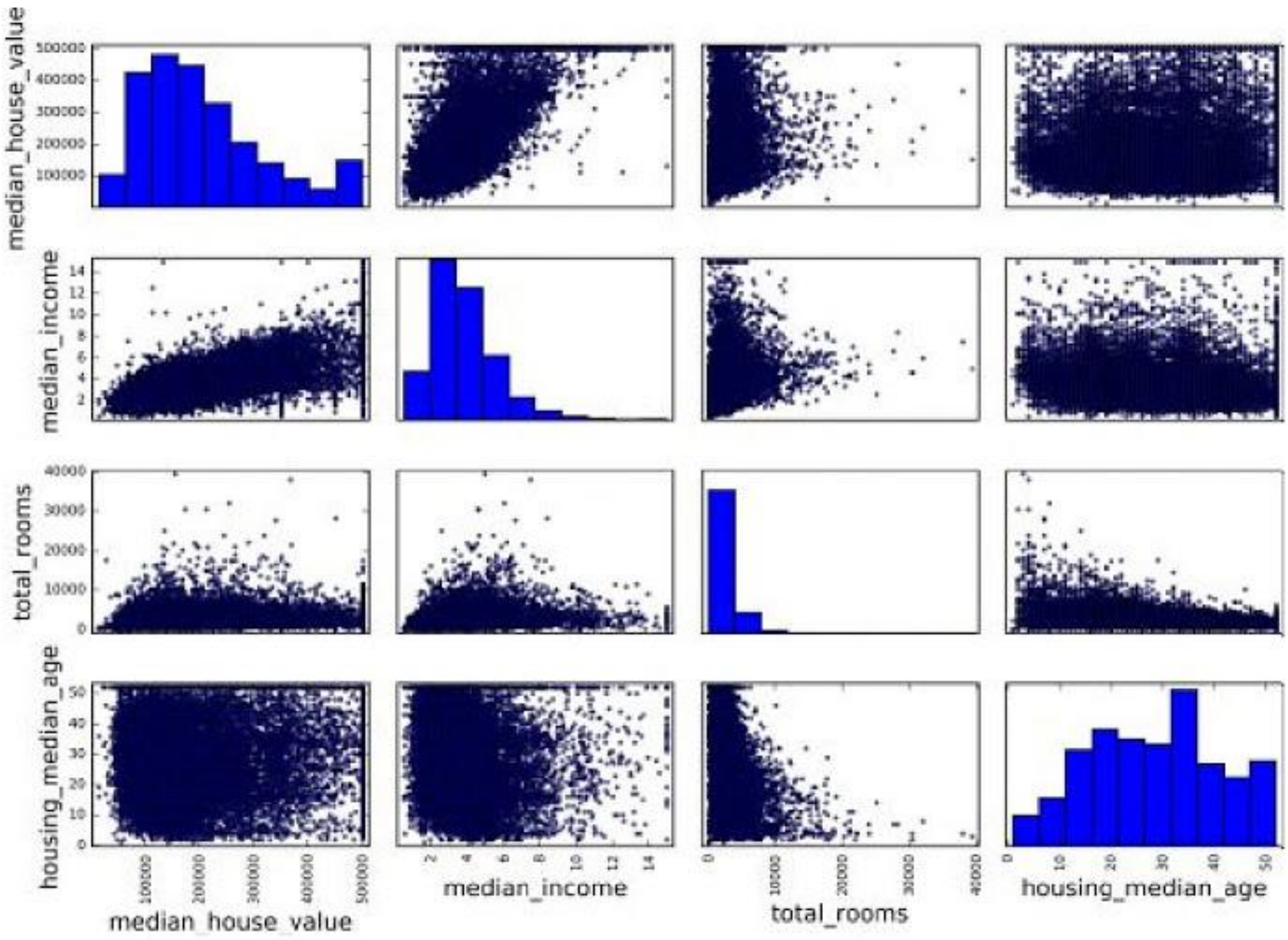


Figure 2-15. Scatter matrix

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

The most promising attribute to predict the median house value is the median income, so let's zoom in on their correlation scatterplot (Figure 2-16):

But first let's revert to a clean training set (by copying `strat_train_set` once again), and let's separate the predictors and the labels since we 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()
```

Data Cleaning

Most Machine Learning algorithms cannot work with missing features, so let's create a few functions to take care of them. You noticed earlier that the `total_bedrooms` attribute has some missing values, so let's fix this. You have three options:

- Get rid of the corresponding districts.
- Get rid of the whole attribute.
- Set the values to some value (zero, the mean, the median, etc.).

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

```
housing.dropna(subset=["total_bedrooms"])
housing.drop("total_bedrooms", axis=1)
median = housing["total_bedrooms"].median()
housing["total_bedrooms"].fillna(median, inplace=True)
```

If you choose option 3, you should compute the median value on the training set, and use it to fill the missing values in the training set, but also don't forget to save the median value that you have computed. You will need it later to replace missing values in the test set when you want to

estimator, as we will see.

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

Handling Text and Categorical Attributes

Earlier we left out the categorical attribute `ocean_proximity` because it is a text attribute so we cannot compute its median. Most Machine Learning algorithms prefer to work with numbers anyway, so let's convert these text labels to numbers.

Scikit-Learn provides a transformer for this task called `LabelEncoder`:

```
>>> from sklearn.preprocessing import LabelEncoder
>>> encoder = LabelEncoder()
>>> housing_cat = housing["ocean_proximity"]
>>> housing_cat_encoded = encoder.fit_transform(housing_cat)
>>> housing_cat_encoded
array([0, 0, 4, ..., 1, 0, 3])
```

This is better: now we can use this numerical data in any ML algorithm. You can look at the mapping that this encoder has learned using the `classes_` attribute ("<1H OCEAN" is mapped to 0, "INLAND" is mapped to 1, etc.):

```
>>> print(encoder.classes_)
['<1H OCEAN' 'INLAND' 'ISLAND' 'NEAR BAY' 'NEAR OCEAN']
```

One issue with this representation is that ML algorithms will assume that two nearby values are more similar than two distant values. Obviously this is not the case (for example, categories 0 and 4 are more similar than

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. Note that scaling the target values is generally not required.

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

Min-max scaling (many people call this *normalization*) is quite simple: values are shifted and rescaled so that they end up ranging from 0 to 1. We do this by subtracting the min value and dividing by the max minus the min. Scikit-Learn provides a transformer called `MinMaxScaler` for this. It has a `feature_range` hyperparameter that lets you change the range if you don't want 0–1 for some reason.

Standardization is quite different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the variance so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers. For example, suppose a district had a median income equal to 100 (by mistake). Min-max scaling would then crush all the other values from 0–15 down to 0–0.15, whereas standardization would not be much affected. Scikit-Learn provides a transformer called `StandardScaler` for standardization.

Warning

As with all the transformations, it is important to fit the scalers to the training data only, not to the full dataset (including the test set). Only then can you use them to transform the training set and the test set (and new

[18] See SciPy's documentation for more details.

[19] But check out Pull Request #3886, which may introduce a `ColumnTransformer` class making attribute-specific transformations easy. You could also run `pip3 install sklearn-pandas` to get a `DataFrameMapper` class with a similar objective.

Figure 3-1. A few digits from the MNIST dataset

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

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

`SGDClassifier` model using K-fold cross-validation, with three folds. Remember that K-fold cross-validation means splitting the training set into K-folds (in this case, three), then making predictions and evaluating them on each fold using a model trained on the remaining folds (see Chapter 2):

```
>>> from sklearn.model_selection import cross_val_score
>>> cross_val_score(sgd_clf, X_train, y_train_5, cv=3,
scoring="accuracy")
array([ 0.9502 ,  0.96565,  0.96495])
```

Wow! Above 95% *accuracy* (ratio of correct predictions) on all cross-validation folds? This looks amazing, doesn't it? Well, before you get too excited, let's look at a very dumb classifier that just classifies every single image in the "not-5" class:

```
from sklearn.base import BaseEstimator
class Never5Classifier(BaseEstimator):
    def fit(self, X, y=None):
        pass
    def predict(self, X):
        return np.zeros((len(X), 1), dtype=bool)
```

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

```
>>> never_5_clf = Never5Classifier()
>>> cross_val_score(never_5_clf, X_train, y_train_5, cv=3,
scoring="accuracy")
array([ 0.909  ,  0.90715,  0.9128 ])
```

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

This demonstrates why accuracy is generally not the preferred performance

```
>>> from sklearn.metrics import precision_score, recall_score
>>> precision_score(y_train_5, y_train_pred)

0.7687135020350380 8
>>> recall_score(y_train_5, y_train_pred)

0.8013281682346430 3
```

Now your 5-detector does not look as shiny as it did when you looked at its accuracy. When it claims an image represents a 5, it is correct only 77% of the time. Moreover, it only detects 80% of the 5s.

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

Equation 3-3. F_1 score

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

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

```
>>> from sklearn.metrics import f1_score
>>> f1_score(y_train_5, y_train_pred)
0.7846820809248554 7
```

The F_1 score favors classifiers that have similar precision and recall. This is not always what you want: in some contexts you mostly care about precision, and in other contexts you really care about recall. For example, if you trained a classifier to detect videos that are safe for kids, you would probably prefer a classifier that rejects many good videos (low recall) but keeps only safe ones (high precision), rather than a classifier that has a

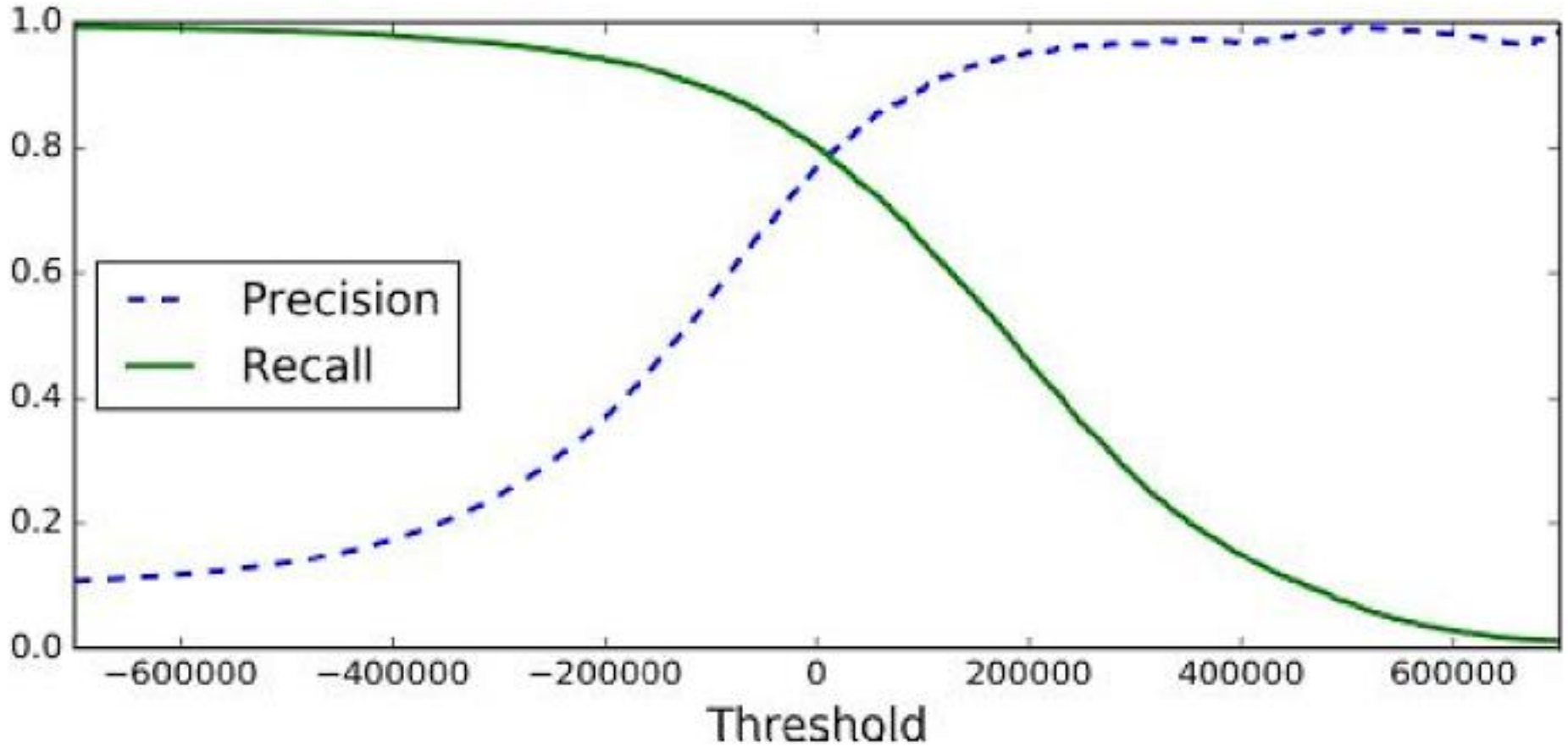


Figure 3-4. Precision and recall versus the decision threshold

Note

You may wonder why the precision curve is bumpier than the recall curve in Figure 3-4. The reason is that precision may sometimes go down when you raise the threshold (although in general it will go up). To understand why, look back at Figure 3-3 and notice what happens when you start from the central threshold and move it just one digit to the right: precision goes from 4/5 (80%) down to 3/4 (75%). On the other hand, recall can only go down when the threshold is increased, which explains why its curve looks smooth.

Now you can simply select the threshold value that gives you the best precision/recall tradeoff for your task. Another way to select a good precision/recall tradeoff is to plot precision directly against recall, as shown in Figure 3-5.

since it is faster to train many classifiers on small training sets than training few classifiers on large training sets. For most binary classification algorithms, however, OvA is preferred.

Scikit-Learn detects when you try to use a binary classification algorithm for a multiclass classification task, and it automatically runs OvA (except for SVM classifiers for which it uses OvO). Let's try this with the `SGDClassifier`:

```
>>> sgd_clf.fit(X_train, y_train)
>>> sgd_clf.predict([some_digit])
array([ 5.])
```

That was easy! This code trains the `SGDClassifier` on the training set using the original target classes from 0 to 9 (`y_train`), instead of the 5-versus-all target classes (`y_train_5`). Then it makes a prediction (a correct one in this case). Under the hood, Scikit-Learn actually trained 10 binary classifiers, got their decision scores for the image, and selected the class with the highest score.

To see that this is indeed the case, you can call the `decision_function()` method. Instead of returning just one score per instance, it now returns 10 scores, one per class:

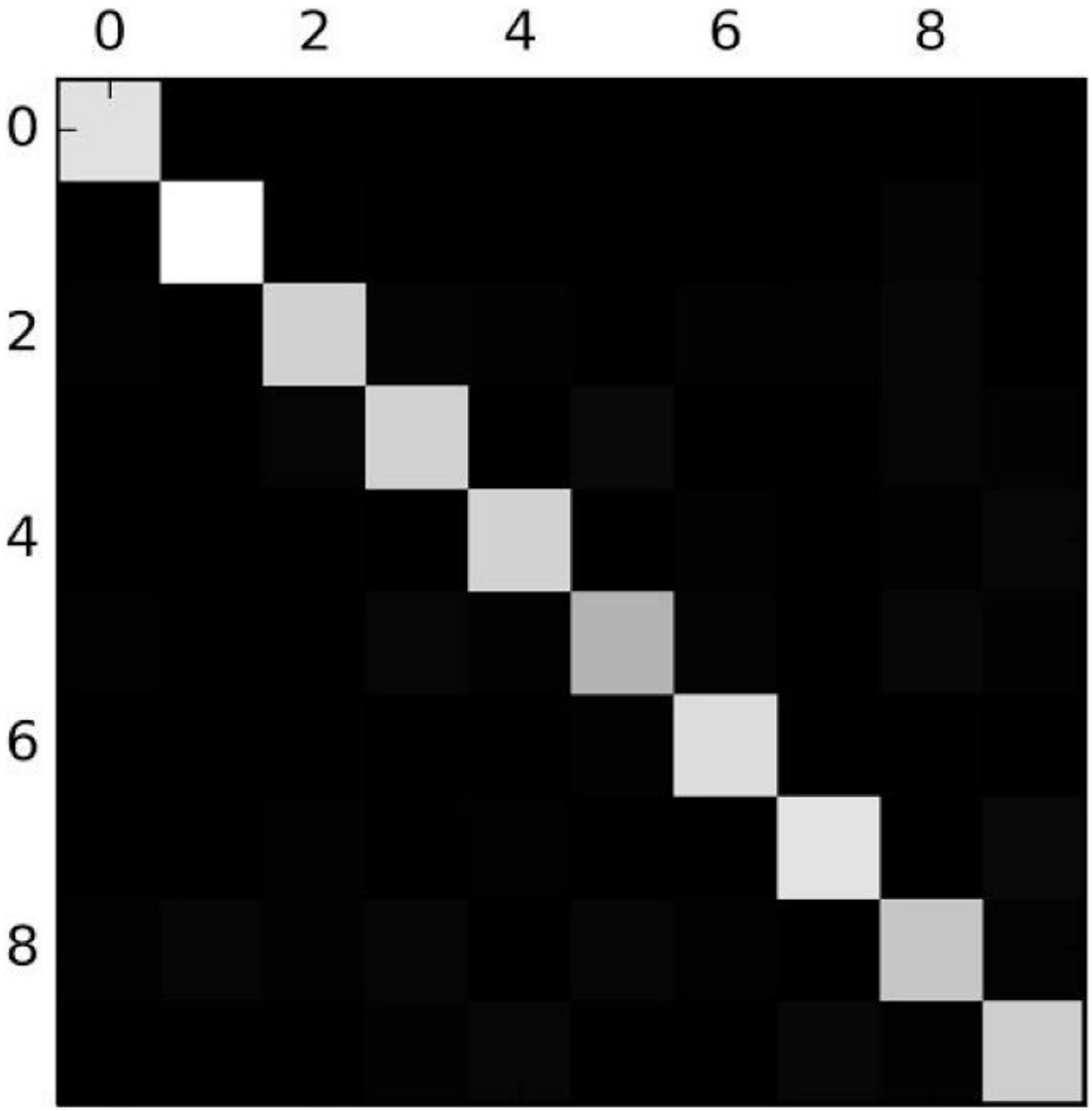
```
>>> some_digit_scores = sgd_clf.decision_function([some_digit])
>>> some_digit_scores
array([[-311402.62954431, -363517.28355739, -446449.5306454 ,
        -183226.61023518, -414337.15339485,  161855.74572176,
        -452576.39616343, -471957.14962573, -518542.33997148,
        -536774.63961222]])
```

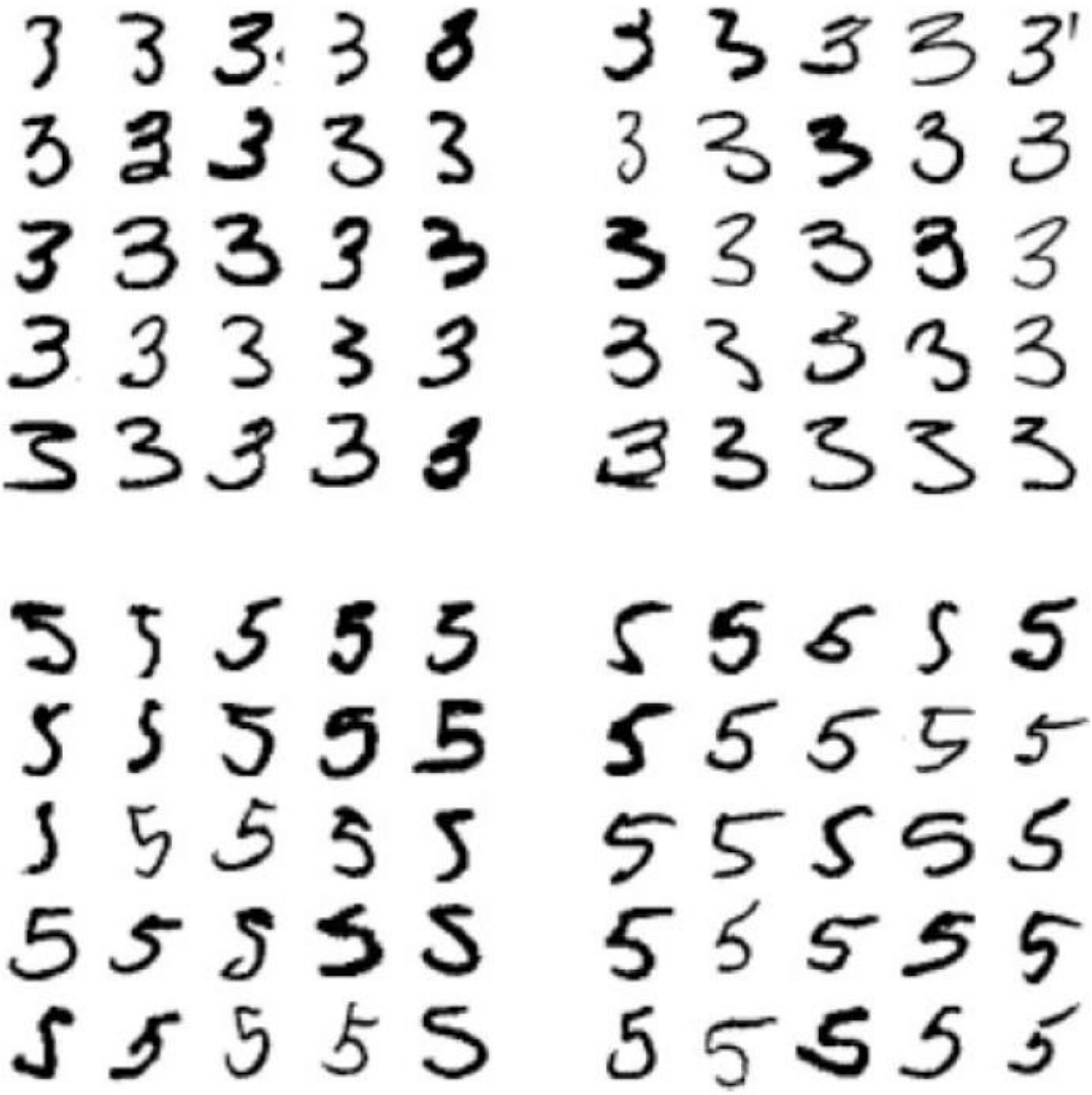
The highest score is indeed the one corresponding to class 5:

```
>>> np.argmax(some_digit_scores)
```

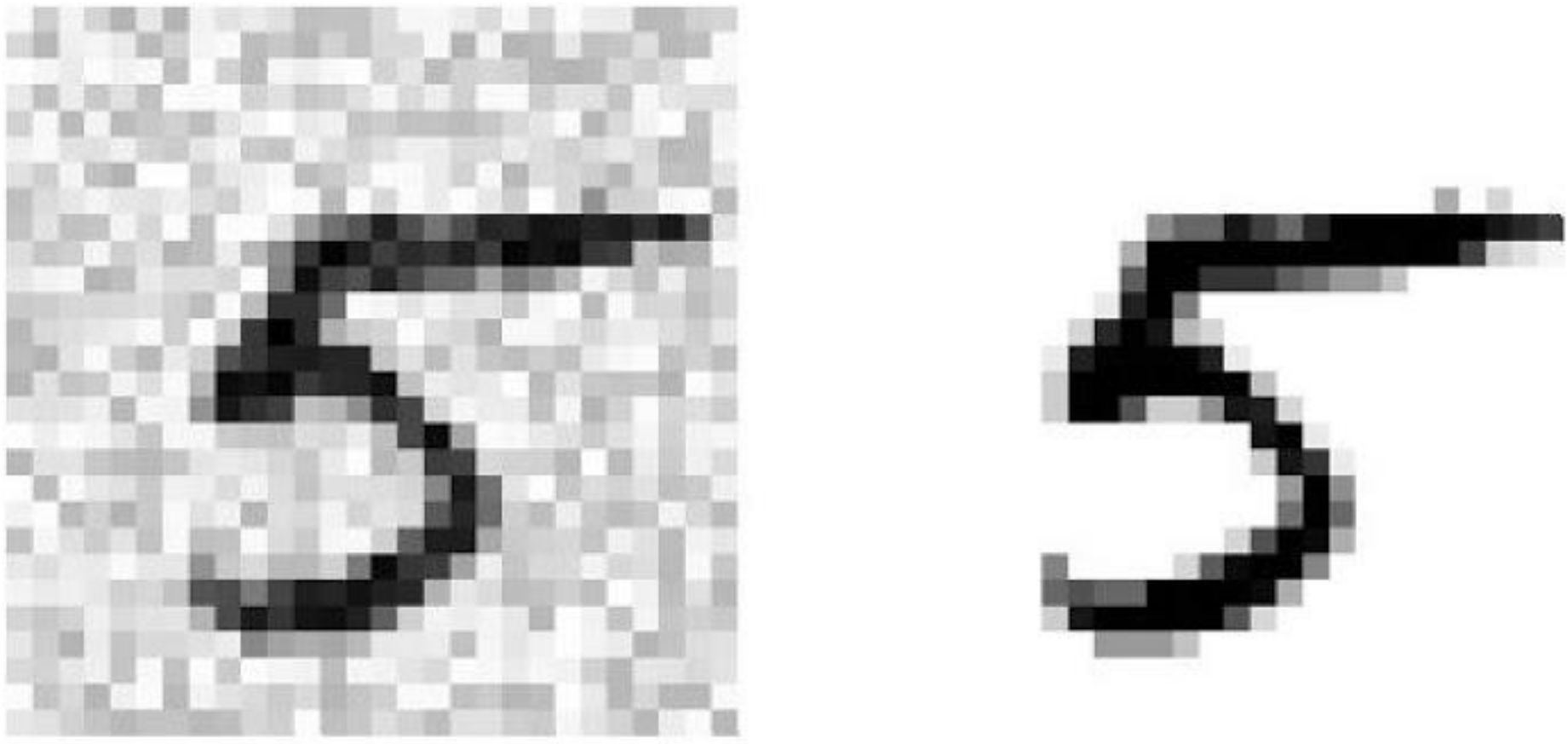
That’s a lot of numbers. It’s often more convenient to look at an image representation of the confusion matrix, using Matplotlib’s `matshow()` function:

```
plt.matshow(conf_mx, cmap=plt.cm.gray)
plt.show()
```





The two 5×5 blocks on the left show digits classified as 3s, and the two 5×5 blocks on the right show images classified as 5s. Some of the digits that the classifier gets wrong (i.e., in the bottom-left and top-right blocks) are so badly written that even a human would have trouble classifying them (e.g., the 5 on the 8th row and 1st column truly looks like a 3). However, most misclassified images seem like obvious errors to us, and it's hard to understand why the classifier made the mistakes it did.[3] The reason is that



On the left is the noisy input image, and on the right is the clean target image. Now let's train the classifier and make it clean this image:

```
knn_clf.fit(X_train_mod, y_train_mod)
clean_digit = knn_clf.predict([X_test_mod[some_index]])
plot_digit(clean_digit)
```

Chapter 4. Training Models

So far we have treated Machine Learning models and their training algorithms mostly like black boxes. If you went through some of the exercises in the previous chapters, you may have been surprised by how much you can get done without knowing anything about what's under the hood: you optimized a regression system, you improved a digit image classifier, and you even built a spam classifier from scratch—all this without knowing how they actually work. Indeed, in many situations you don't really need to know the implementation details.

However, having a good understanding of how things work can help you quickly home in on the appropriate model, the right training algorithm to use, and a good set of hyperparameters for your task. Understanding what's under the hood will also help you debug issues and perform error analysis more efficiently. Lastly, most of the topics discussed in this chapter will be essential in understanding, building, and training neural networks (discussed in Part II of this book).

In this chapter, we will start by looking at the Linear Regression model, one of the simplest models there is. We will discuss two very different ways to train it:

- Using a direct "closed-form" equation that directly computes the model parameters that best fit the model to the training set (i.e., the model parameters that minimize the cost function over the training set).
- Using an iterative optimization approach, called Gradient Descent (GD), that gradually tweaks the model parameters to minimize the cost function over the training set, eventually converging to the same set of parameters as the first method. We will look at a few variants of Gradient Descent that we will use again and again when we study neural networks in Part II: Batch GD, Mini-batch GD, and Stochastic GD.

Let's generate some linear-looking data to test this equation on (Figure 4-1):

```
import numpy as np

X = 2 * np.random.rand(100, 1)
y = 4 + 3 * X + np.random.randn(100, 1)
```

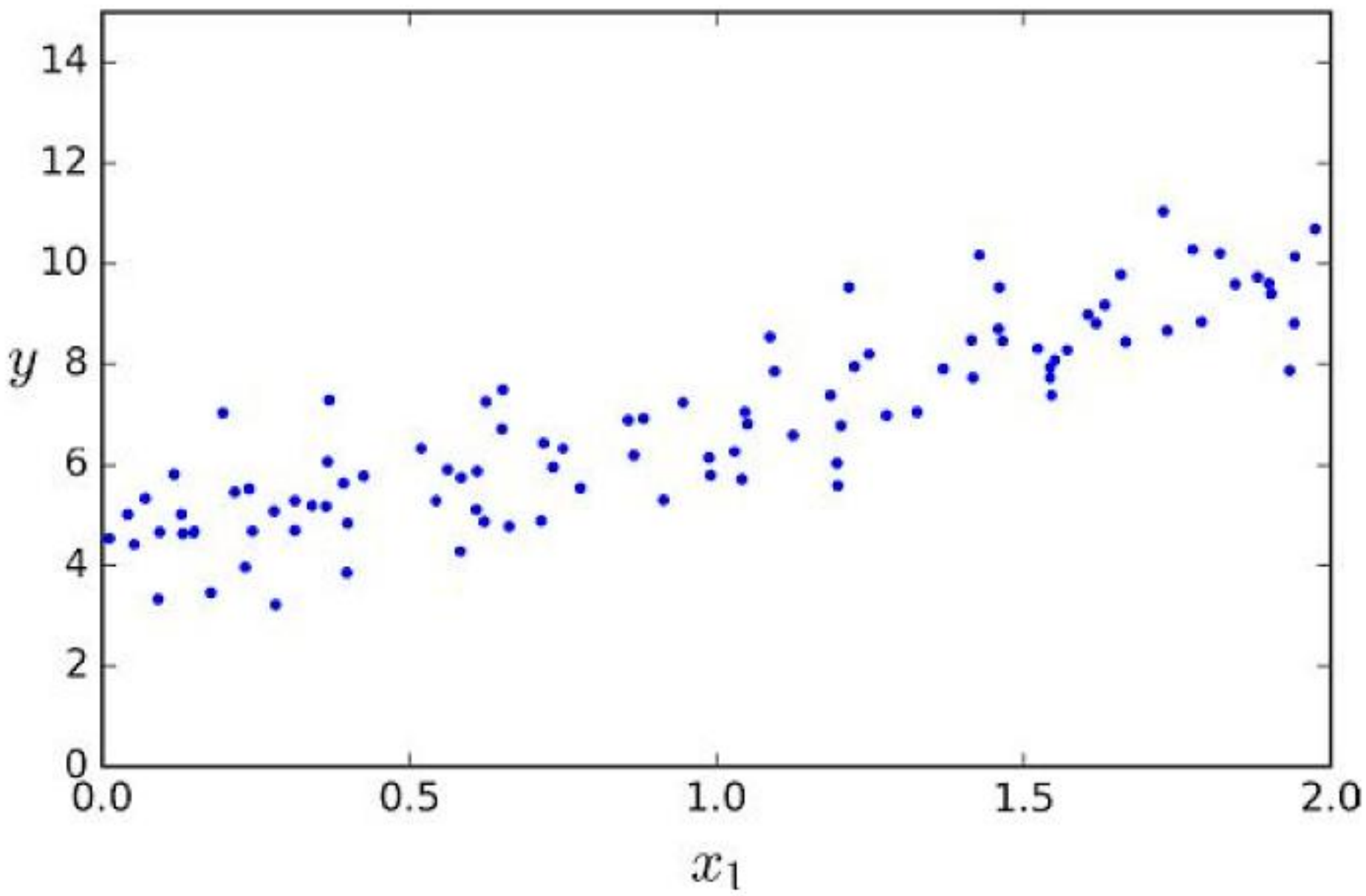


Figure 4-1. Randomly generated linear dataset

Now let's compute $\hat{\theta}$ using the Normal Equation. We will use the `inv()` function from NumPy's Linear Algebra module (`np.linalg`) to compute the inverse of a matrix, and the `dot()` method for matrix multiplication:

```
X_b = np.c_[np.ones((100, 1)), X]
theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y)
```

Concretely, you start by filling with random values (this is called *random initialization*), and then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm *converges* to a minimum (see Figure 4-3).

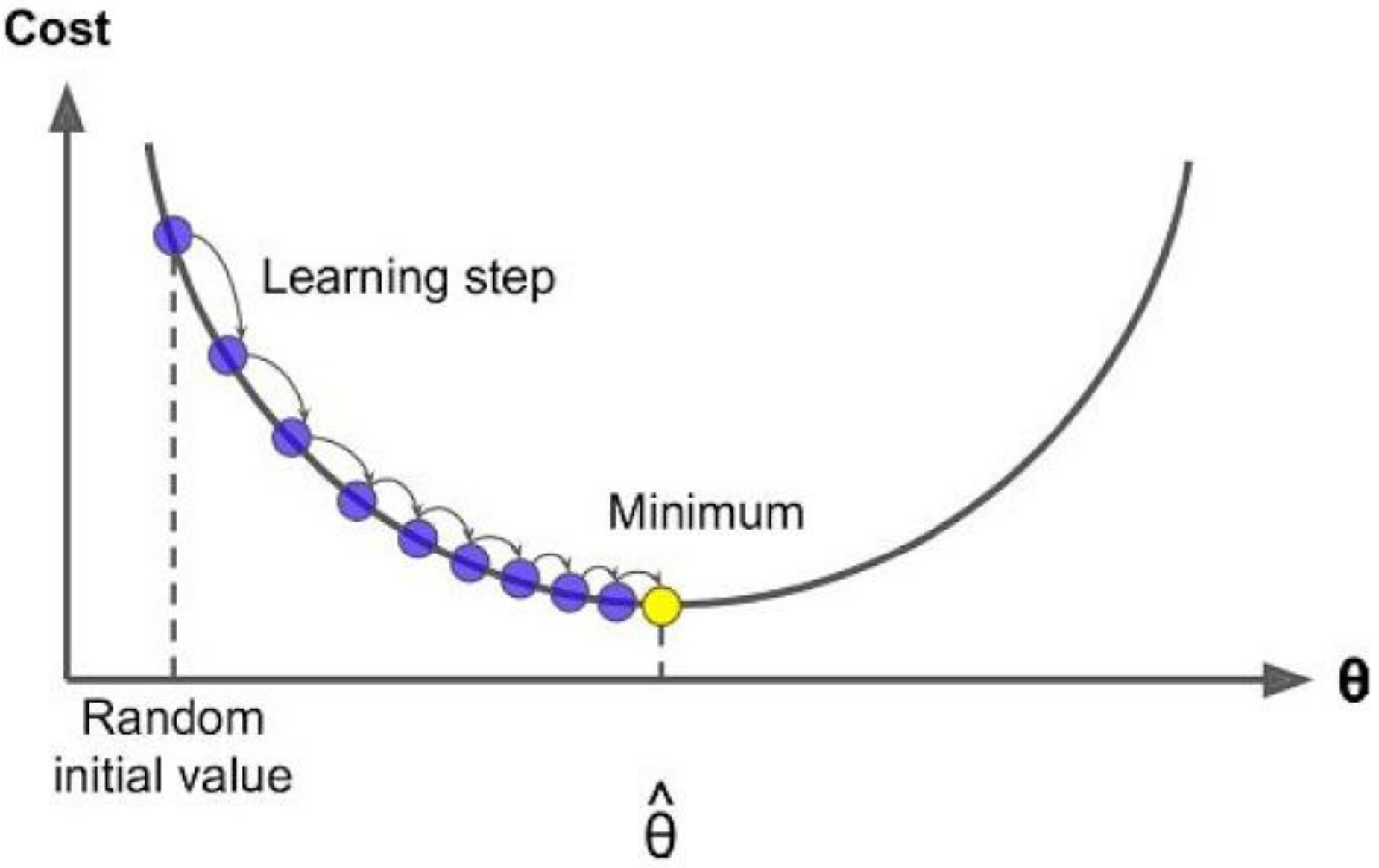


Figure 4-3. Gradient Descent

An important parameter in Gradient Descent is the size of the steps, determined by the *learning rate* hyperparameter. If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time (see Figure 4-4).

Table 4-1. Comparison of algorithms for Linear Regression

Algorithm	Large *m*	Out-of-core support	Large *n*	Hyperparams	Scaling required	Scikit-Learn
Normal Equation	Fast	No	Slow	0	No	`LinearRegression`
Batch GD	Slow	No	Fast	2	Yes	n/a
Stochastic GD	Fast	Yes	Fast	≥2	Yes	`SGDRegressor`
Mini-batch GD	Fast	Yes	Fast	≥2	Yes	n/a

Note

There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

Polynomial Regression

What if your data is actually more complex than a simple straight line? Surprisingly, you can actually use a linear model to fit nonlinear data. A simple way to do this is to add powers of each feature as new features, then train a linear model on this extended set of features. This technique is called *Polynomial Regression*.

Let's look at an example. First, let's generate some nonlinear data, based on a simple *quadratic equation*[9] (plus some noise; see Figure 4-12):

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 + X + 2 + np.random.randn(m, 1)
```

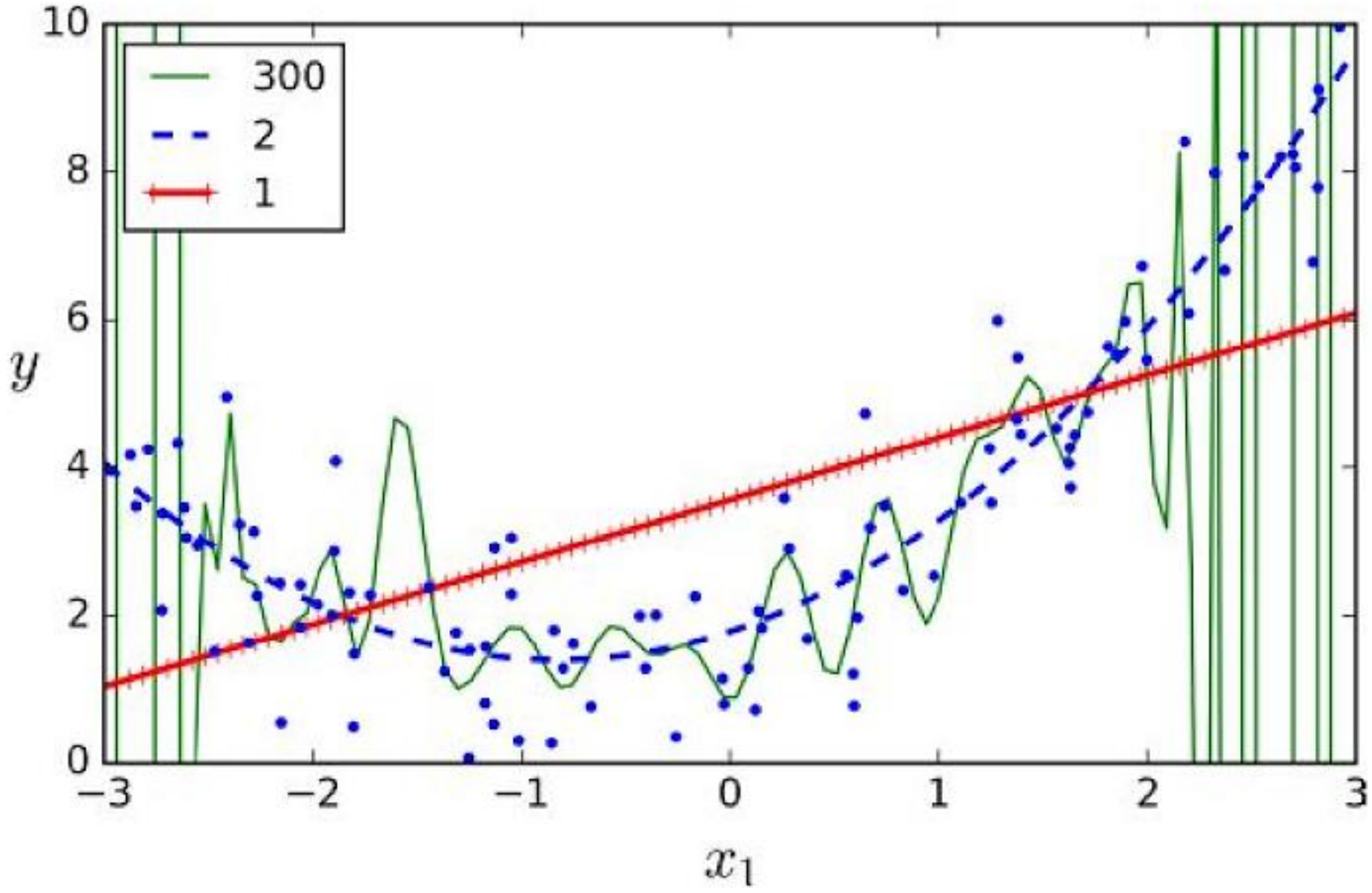


Figure 4-14. High-degree Polynomial Regression

Of course, this high-degree Polynomial Regression model is severely overfitting the training data, while the linear model is underfitting it. The model that will generalize best in this case is the quadratic model. It makes sense since the data was generated using a quadratic model, but in general you won't know what function generated the data, so how can you decide how complex your model should be? How can you tell that your model is overfitting or underfitting the data?

In Chapter 2 you used cross-validation to get an estimate of a model's generalization performance. If a model performs well on the training data but generalizes poorly according to the cross-validation metrics, then your model is overfitting. If it performs poorly on both, then it is underfitting. This is one way to tell when a model is too simple or too complex.

Another way is to look at the *learning curves*: these are plots of the model's

performance on the training set and the validation set as a function of the training set size. To generate the plots, simply train the model several times on different sized subsets of the training set. The following code defines a function that plots the learning curves of a model given some training data:

```
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split

def plot_learning_curves(model, X, y):
    X_train, X_val, y_train, y_val = train_test_split(X, y,
test_size=0.2)
    train_errors, val_errors = [], []
    for m in range(1, len(X_train)):
        model.fit(X_train[:m], y_train[:m])
        y_train_predict = model.predict(X_train[:m])
        y_val_predict = model.predict(X_val)
        train_errors.append(mean_squared_error(y_train_predict,
y_train[:m]))
        val_errors.append(mean_squared_error(y_val_predict,
y_val))
    plt.plot(np.sqrt(train_errors), "r-+", linewidth=2,
label="train")
    plt.plot(np.sqrt(val_errors), "b-", linewidth=3, label="val")
```

Let’s look at the learning curves of the plain Linear Regression model (a straight line; Figure 4-15):

```
lin_reg = LinearRegression()
plot_learning_curves(lin_reg, X, y)
```

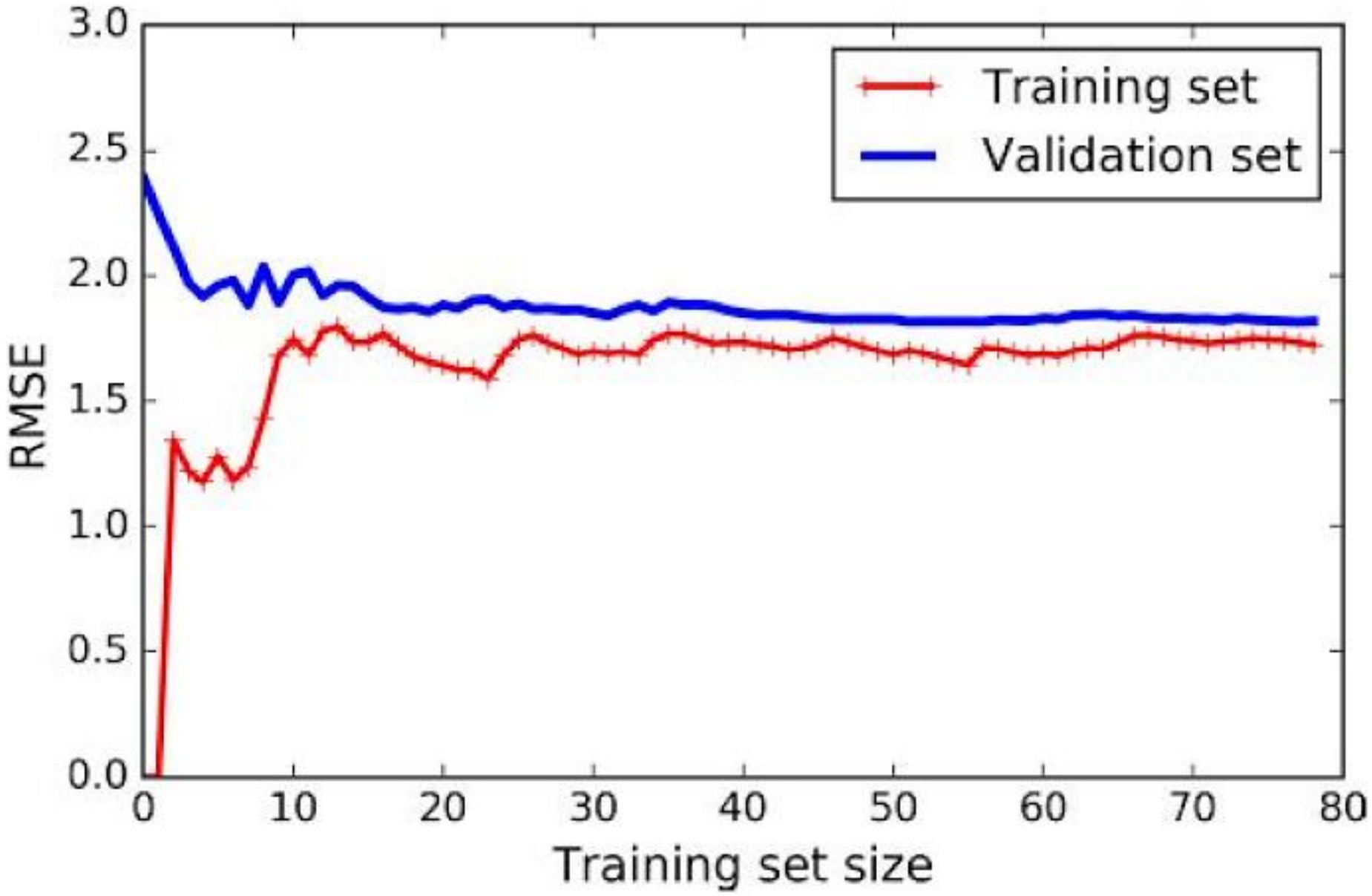


Figure 4-15. Learning curves

This deserves a bit of explanation. First, let's look at the performance on the training data: when there are just one or two instances in the training set, the model can fit them perfectly, which is why the curve starts at zero. But as new instances are added to the training set, it becomes impossible for the model to fit the training data perfectly, both because the data is noisy and because it is not linear at all. So the error on the training data goes up until it reaches a plateau, at which point adding new instances to the training set doesn't make the average error much better or worse. Now let's look at the performance of the model on the validation data. When the model is trained on very few training instances, it is incapable of generalizing properly, which is why the validation error is initially quite big. Then as the model is shown more training examples, it learns and thus the validation error slowly goes down. However, once again a straight line cannot do a good job modeling the data, so the error ends up at a plateau,

very close to the other curve.

These learning curves are typical of an underfitting model. Both curves have reached a plateau; they are close and fairly high.

Tip

If your model is underfitting the training data, adding more training examples will not help. You need to use a more complex model or come up with better features.

Now let's look at the learning curves of a 10^{th}-degree polynomial model on the same data (Figure 4-16):

```
from sklearn.pipeline import Pipeline

polynomial_regression = Pipeline((
        ("poly_features", PolynomialFeatures(degree=10,
include_bias=False)),
        ("lin_reg", LinearRegression()),
    ))

plot_learning_curves(polynomial_regression, X, y)
```

These learning curves look a bit like the previous ones, but there are two very important differences:

- The error on the training data is much lower than with the Linear Regression model.
- There is a gap between the curves. This means that the model performs significantly better on the training data than on the validation data, which is the hallmark of an overfitting model. However, if you used a much larger training set, the two curves would continue to get closer.

The Bias/Variance Tradeoff

An important theoretical result of statistics and Machine Learning is the fact that a model's generalization error can be expressed as the sum of three very different errors:

Bias

> This part of the generalization error is due to wrong assumptions, such as assuming that the data is linear when it is actually quadratic. A high-bias model is most likely to underfit the training data.[10]

Variance

> This part is due to the model's excessive sensitivity to small variations in the training data. A model with many degrees of freedom (such as a high-degree polynomial model) is likely to have high variance, and thus to overfit the training data.

Irreducible error

> This part is due to the noisiness of the data itself. The only way to reduce this part of the error is to clean up the data (e.g., fix the data sources, such as broken sensors, or detect and remove outliers).

Increasing a model's complexity will typically increase its variance and reduce its bias. Conversely, reducing a model's complexity increases its bias and reduces its variance. This is why it is called a tradeoff.

Regularized Linear Models

As we saw in Chapters 1 and 2, a good way to reduce overfitting is to regularize the model (i.e., to constrain it): the fewer degrees of freedom it has, the harder it will be for it to overfit the data. For example, a simple

way to regularize a polynomial model is to reduce the number of polynomial degrees.

For a linear model, regularization is typically achieved by constraining the weights of the model. We will now look at Ridge Regression, Lasso Regression, and Elastic Net, which implement three different ways to constrain the weights.

Ridge Regression

Ridge Regression (also called *Tikhonov regularization*) is a regularized version of Linear Regression: a *regularization term* equal to $\alpha \sum_{i=1}^{n} \theta_i^2$ is added to the cost function. This forces the learning algorithm to not only fit the data but also keep the model weights as small as possible. Note that the regularization term should only be added to the cost function during training. Once the model is trained, you want to evaluate the model's performance using the unregularized performance measure.

Note

It is quite common for the cost function used during training to be different from the performance measure used for testing. Apart from regularization, another reason why they might be different is that a good training cost function should have optimization-friendly derivatives, while the performance measure used for testing should be as close as possible to the final objective. A good example of this is a classifier trained using a cost function such as the log loss (discussed in a moment) but evaluated using precision/recall.

The hyperparameter controls how much you want to regularize the model. If = 0 then Ridge Regression is just Linear Regression. If is very large, then all weights end up very close to zero and the result is a flat line going through the data's mean. Equation 4-8 presents the Ridge Regression cost function.[11]

Equation 4-8. Ridge Regression cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha\frac{1}{2}\sum_{i=1}^{n}\theta_i^2$$

Note that the bias term θ_0 is not regularized (the sum starts at $i = 1$, not 0). If we define $\mathbf{w}$ as the vector of feature weights (θ_1 to θ_n), then the regularization term is simply equal to $\frac{1}{2}(\|\mathbf{w}\|_2)^2$, where $\|\cdot\|_2$ represents the ℓ_2 norm of the weight vector.[12] For Gradient Descent, just add $\mathbf{w}$ to the MSE gradient vector (Equation 4-6).

Warning

It is important to scale the data (e.g., using a `StandardScaler`) before performing Ridge Regression, as it is sensitive to the scale of the input features. This is true of most regularized models.

Figure 4-17 shows several Ridge models trained on some linear data using different α value. On the left, plain Ridge models are used, leading to linear predictions. On the right, the data is first expanded using `PolynomialFeatures(degree=10)`, then it is scaled using a `StandardScaler`, and finally the Ridge models are applied to the resulting features: this is Polynomial Regression with Ridge regularization. Note how increasing α leads to flatter (i.e., less extreme, more reasonable) predictions; this reduces the model's variance but increases its bias.

As with Linear Regression, we can perform Ridge Regression either by computing a closed-form equation or by performing Gradient Descent. The pros and cons are the same. Equation 4-9 shows the closed-form solution (where $\mathbf{A}$ is the $n \times n$ *identity matrix*[13] except with a 0 in the top-left cell, corresponding to the bias term).

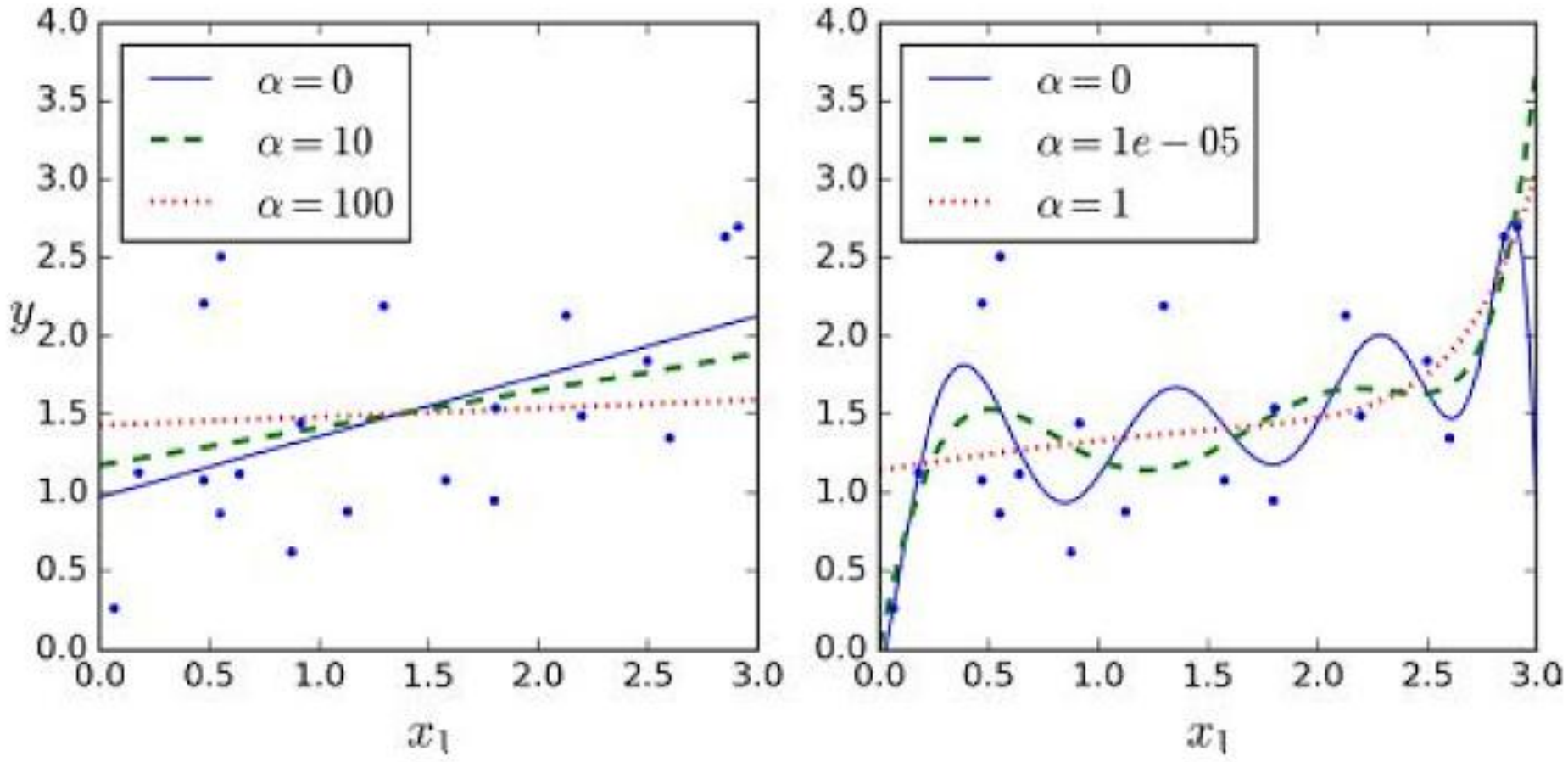


Figure 4-17. Ridge Regression

Equation 4-9. Ridge Regression closed-form solution

$$\hat{\theta} = (\mathbf{X}^T \cdot \mathbf{X} + \alpha \mathbf{A})^{-1} \cdot \mathbf{X}^T \cdot \mathbf{y}$$

Here is how to perform Ridge Regression with Scikit-Learn using a closed-form solution (a variant of Equation 4-9 using a matrix factorization technique by André-Louis Cholesky):

```
>>> from sklearn.linear_model import Ridge
>>> ridge_reg = Ridge(alpha=1, solver="cholesky")
>>> ridge_reg.fit(X, y)
>>> ridge_reg.predict([[1.5]])
array([[ 1.55071465]])
```

And using Stochastic Gradient Descent:[14]

```
>>> sgd_reg = SGDRegressor(penalty="l2")
>>> sgd_reg.fit(X, y.ravel())
>>> sgd_reg.predict([[1.5]])
array([ 1.13500145])
```

The `penalty` hyperparameter sets the type of regularization term to use. Specifying `"l2"` indicates that you want SGD to add a regularization term to the cost function equal to half the square of the ℓ_2 norm of the weight vector: this is simply Ridge Regression.

Lasso Regression

Least Absolute Shrinkage and Selection Operator Regression (simply called *Lasso Regression*) is another regularized version of Linear Regression: just like Ridge Regression, it adds a regularization term to the cost function, but it uses the ℓ_1 norm of the weight vector instead of half the square of the ℓ_2 norm (see Equation 4-10).

Equation 4-10. Lasso Regression cost function

$$J(\theta) = \text{MSE}(\theta) + \alpha \sum_{i=1}^{n} |\theta_i|$$

Figure 4-18 shows the same thing as Figure 4-17 but replaces Ridge models with Lasso models and uses smaller values.

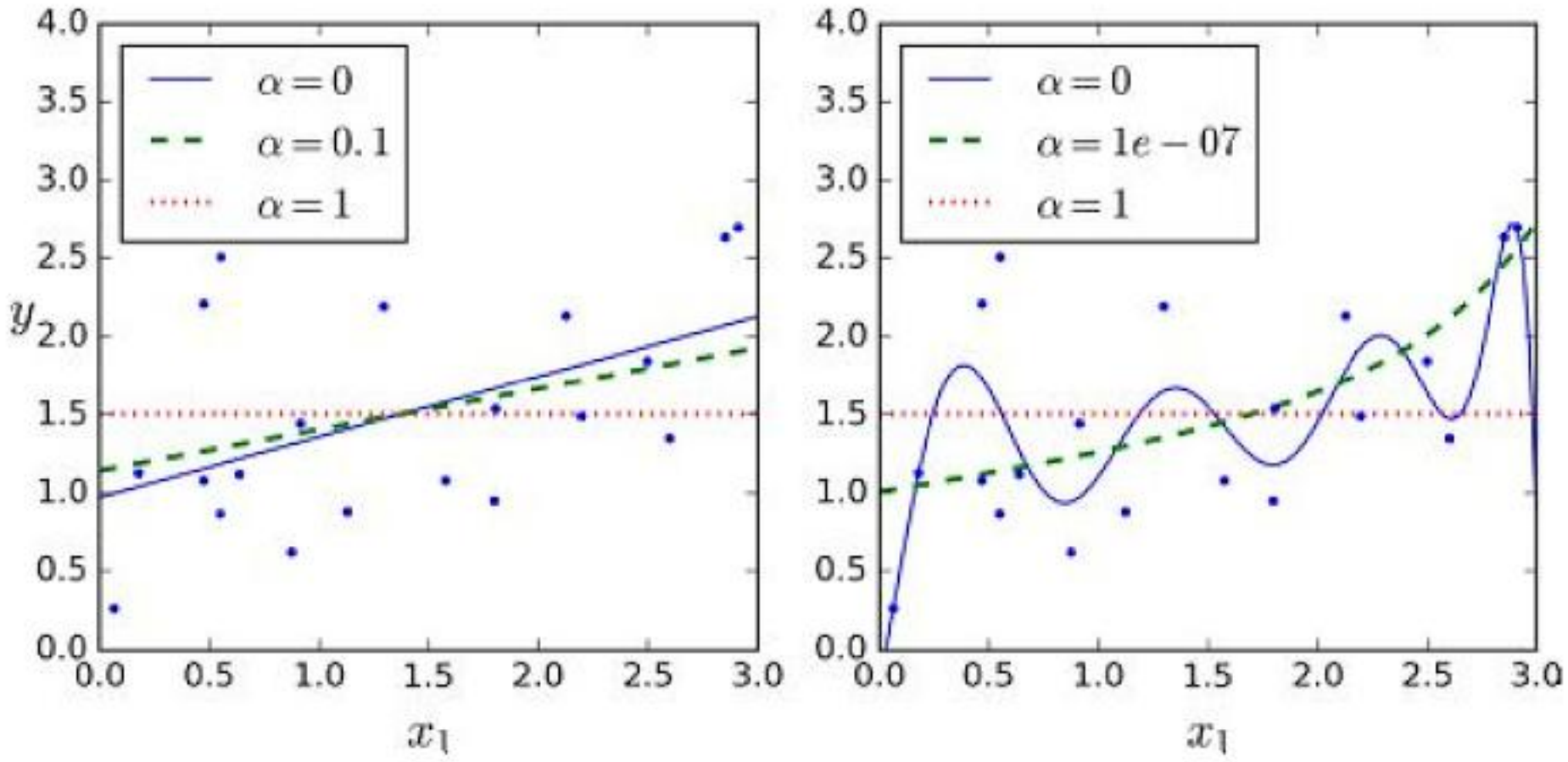


Figure 4-18. Lasso Regression

An important characteristic of Lasso Regression is that it tends to completely eliminate the weights of the least important features (i.e., set them to zero). For example, the dashed line in the right plot on Figure 4-18 (with $\alpha = 10^{-7}$) looks quadratic, almost linear: all the weights for the high-degree polynomial features are equal to zero. In other words, Lasso Regression automatically performs feature selection and outputs a *sparse model* (i.e., with few nonzero feature weights).

You can get a sense of why this is the case by looking at Figure 4-19: on the top-left plot, the background contours (ellipses) represent an unregularized MSE cost function ($\alpha = 0$), and the white circles show the Batch Gradient Descent path with that cost function. The foreground contours (diamonds) represent the ℓ_1 penalty, and the triangles show the BGD path for this penalty only ($\alpha \to \infty$). Notice how the path first reaches $\theta_1 = 0$, then rolls down a gutter until it reaches $\theta_2 = 0$. On the top-right plot, the contours represent the same cost function plus an ℓ_1 penalty with $\alpha = 0.5$. The global minimum is on the $\theta_2 = 0$ axis. BGD first reaches $\theta_2 = 0$, then rolls down the gutter until it reaches the global minimum. The two bottom plots show the same thing but uses an ℓ_2 penalty

instead. The regularized minimum is closer to = 0 than the unregularized minimum, but the weights do not get fully eliminated.

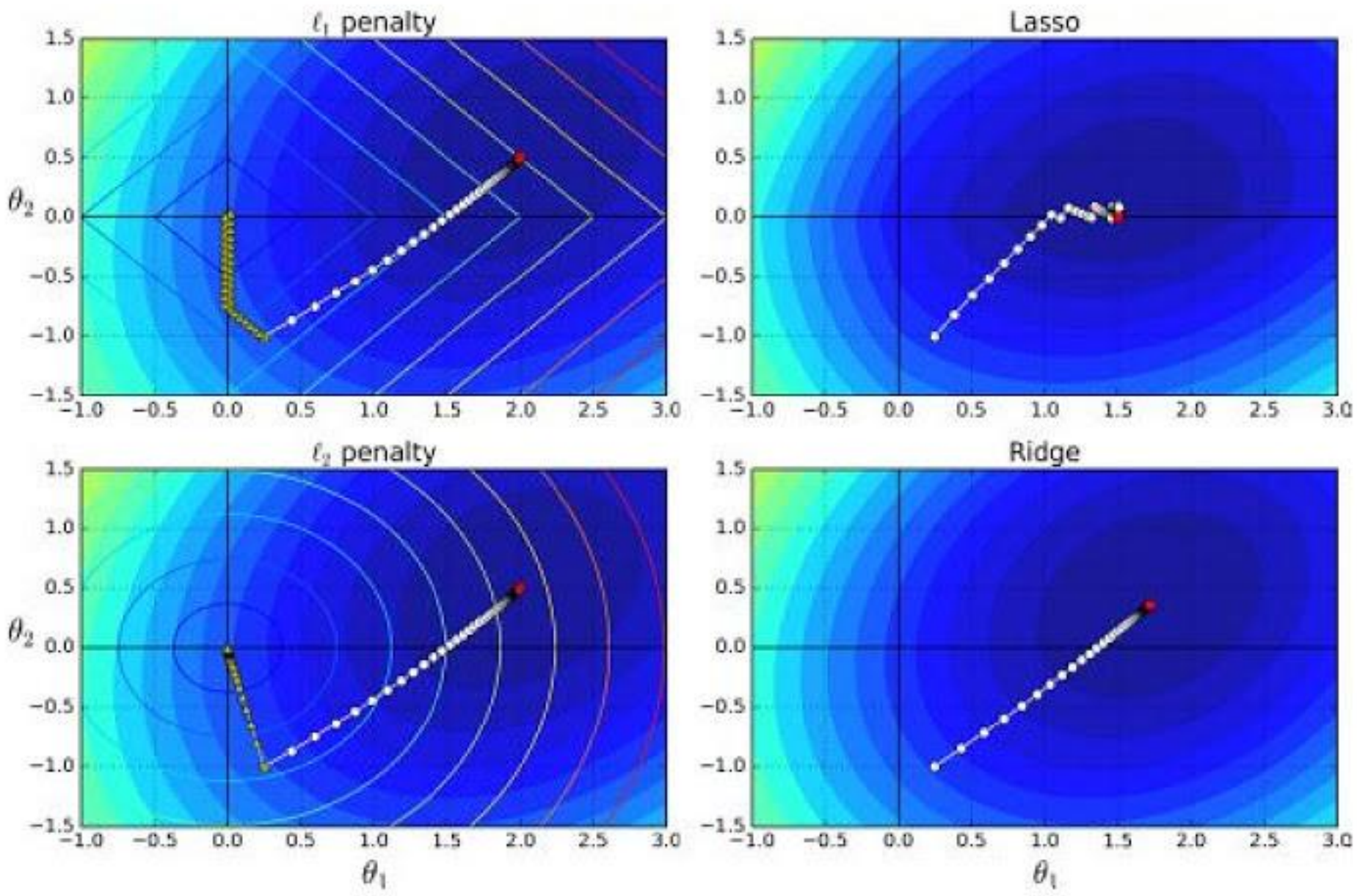


Figure 4-19. Lasso versus Ridge regularization

Tip

On the Lasso cost function, the BGD path tends to bounce across the gutter toward the end. This is because the slope changes abruptly at $_2 = 0$. You need to gradually reduce the learning rate in order to actually converge to the global minimum.

The Lasso cost function is not differentiable at $_i = 0$ (for $i = 1, 2, \cdots, n$), but Gradient Descent still works fine if you use a *subgradient vector* $\mathbf{g}$[15] instead when any $_i = 0$. Equation 4-11 shows a subgradient vector equation you can use for Gradient Descent with the Lasso cost function.

Equation 4-11. Lasso Regression subgradient vector

$$g(\theta, J) = \nabla_\theta \text{MSE}(\theta) + \alpha \begin{pmatrix} \text{sign}(\theta_1) \\ \text{sign}(\theta_2) \\ \vdots \\ \text{sign}(\theta_n) \end{pmatrix} \quad \text{where sign}(\theta_i) = \begin{cases} -1 & \text{if } \theta_i < 0 \\ 0 & \text{if } \theta_i = 0 \\ +1 & \text{if } \theta_i > 0 \end{cases}$$

Here is a small Scikit-Learn example using the `Lasso` class. Note that you could instead use an `SGDRegressor(penalty="l1")`.

```
>>> from sklearn.linear_model import Lasso
>>> lasso_reg = Lasso(alpha=0.1)
>>> lasso_reg.fit(X, y)
>>> lasso_reg.predict([[1.5]])
array([ 1.53788174])
```

Elastic Net

Elastic Net is a middle ground between Ridge Regression and Lasso Regression. The regularization term is a simple mix of both Ridge and Lasso's regularization terms, and you can control the mix ratio *r*. When *r* = 0, Elastic Net is equivalent to Ridge Regression, and when *r* = 1, it is equivalent to Lasso Regression (see Equation 4-12).

Equation 4-12. Elastic Net cost function

$$J(\theta) = \text{MSE}(\theta) + r\alpha \sum_{i=1}^{n} |\theta_i| + \frac{1-r}{2}\alpha \sum_{i=1}^{n} \theta_i^2$$

So when should you use plain Linear Regression (i.e., without any regularization), Ridge, Lasso, or Elastic Net? It is almost always preferable to have at least a little bit of regularization, so generally you should avoid plain Linear Regression. Ridge is a good default, but if you suspect that only a few features are actually useful, you should prefer Lasso or Elastic Net since they tend to reduce the useless features' weights down to zero as we have discussed. In general, Elastic Net is preferred over Lasso since Lasso may behave erratically when the number of features is greater than

the number of training instances or when several features are strongly correlated.

Here is a short example using Scikit-Learn's `ElasticNet` (`l1_ratio` corresponds to the mix ratio *r*):

```
>>> from sklearn.linear_model import ElasticNet
>>> elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
>>> elastic_net.fit(X, y)
>>> elastic_net.predict([[1.5]])
array([ 1.54333232])
```

Early Stopping

A very different way to regularize iterative learning algorithms such as Gradient Descent is to stop training as soon as the validation error reaches a minimum. This is called *early stopping*. Figure 4-20 shows a complex model (in this case a high-degree Polynomial Regression model) being trained using Batch Gradient Descent. As the epochs go by, the algorithm learns and its prediction error (RMSE) on the training set naturally goes down, and so does its prediction error on the validation set. However, after a while the validation error stops decreasing and actually starts to go back up. This indicates that the model has started to overfit the training data. With early stopping you just stop training as soon as the validation error reaches the minimum. It is such a simple and efficient regularization technique that Geoffrey Hinton called it a "beautiful free lunch."

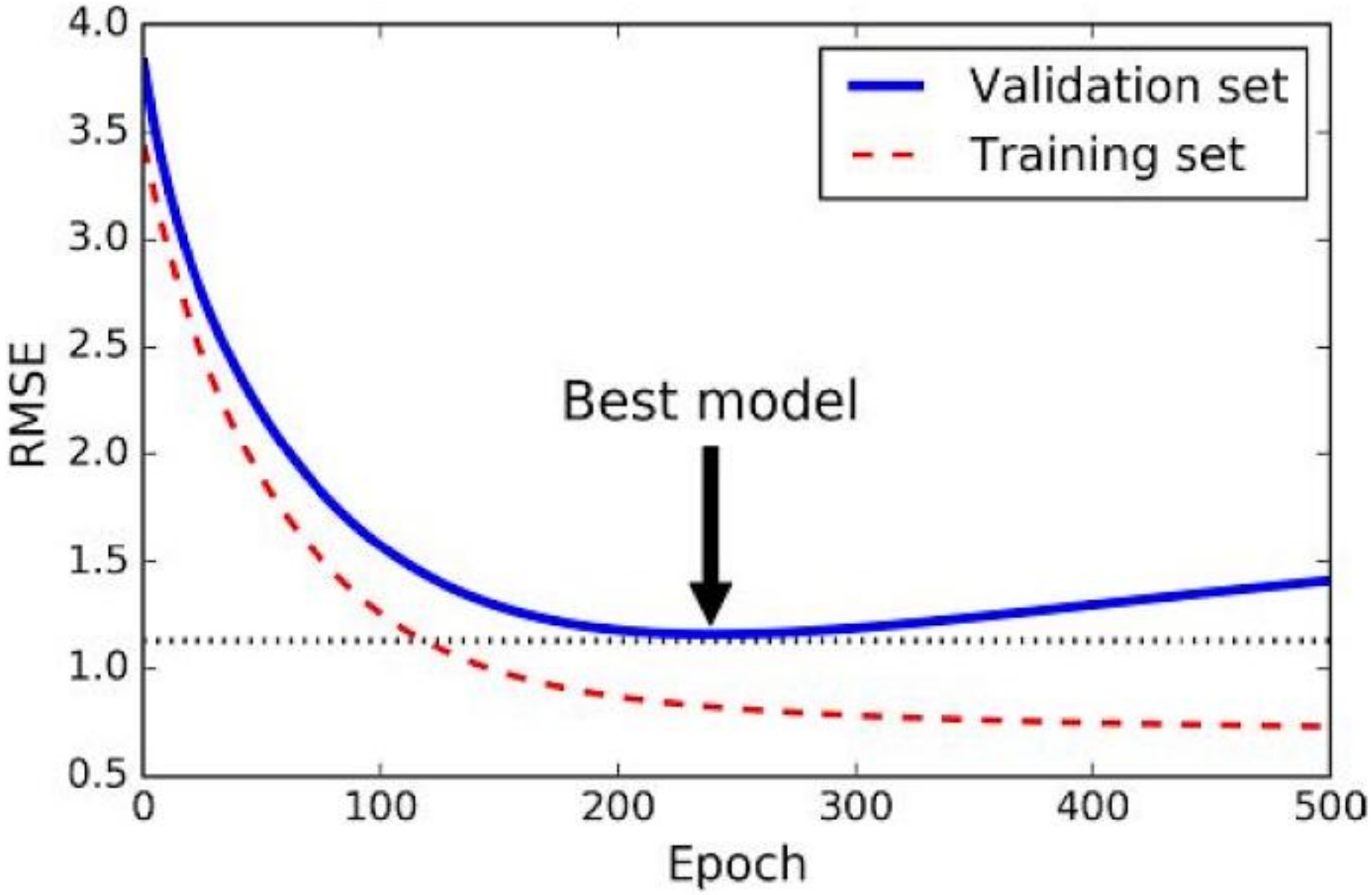


Figure 4-20. Early stopping regularization

Tip

With Stochastic and Mini-batch Gradient Descent, the curves are not so smooth, and it may be hard to know whether you have reached the minimum or not. One solution is to stop only after the validation error has been above the minimum for some time (when you are confident that the model will not do any better), then roll back the model parameters to the point where the validation error was at a minimum.

Here is a basic implementation of early stopping:

```
from sklearn.base import clone

sgd_reg = SGDRegressor(n_iter=1, warm_start=True, penalty=None,
                       learning_rate="constant", eta0=0.0005)

minimum_val_error = float("inf")
```

```
best_epoch = None
best_model = None
for epoch in range(1000):
    sgd_reg.fit(X_train_poly_scaled, y_train)

    y_val_predict = sgd_reg.predict(X_val_poly_scaled)
    val_error = mean_squared_error(y_val_predict, y_val)
    if val_error < minimum_val_error:
        minimum_val_error = val_error
        best_epoch = epoch
        best_model = clone(sgd_reg)
```

Note that with `warm_start=True`, when the `fit()` method is called, it just continues training where it left off instead of restarting from scratch.

Logistic Regression

As we discussed in Chapter 1, some regression algorithms can be used for classification as well (and vice versa). *Logistic Regression* (also called *Logit Regression*) is commonly used to estimate the probability that an instance belongs to a particular class (e.g., what is the probability that this email is spam?). If the estimated probability is greater than 50%, then the model predicts that the instance belongs to that class (called the positive class, labeled "1"), or else it predicts that it does not (i.e., it belongs to the negative class, labeled "0"). This makes it a binary classifier.

Estimating Probabilities

So how does it work? Just like a Linear Regression model, a Logistic Regression model computes a weighted sum of the input features (plus a bias term), but instead of outputting the result directly like the Linear Regression model does, it outputs the *logistic* of this result (see Equation 4-13).

Equation 4-13. Logistic Regression model estimated probability (vectorized form)

$$\hat{p} = h_{\theta}(\mathbf{x}) = \sigma(\theta^{T} \cdot \mathbf{x})$$

The logistic—also called the *logit*, noted $\sigma(\cdot)$—is a *sigmoid function* (i.e., *S*-shaped) that outputs a number between 0 and 1. It is defined as shown in Equation 4-14 and Figure 4-21.

Equation 4-14. Logistic function

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

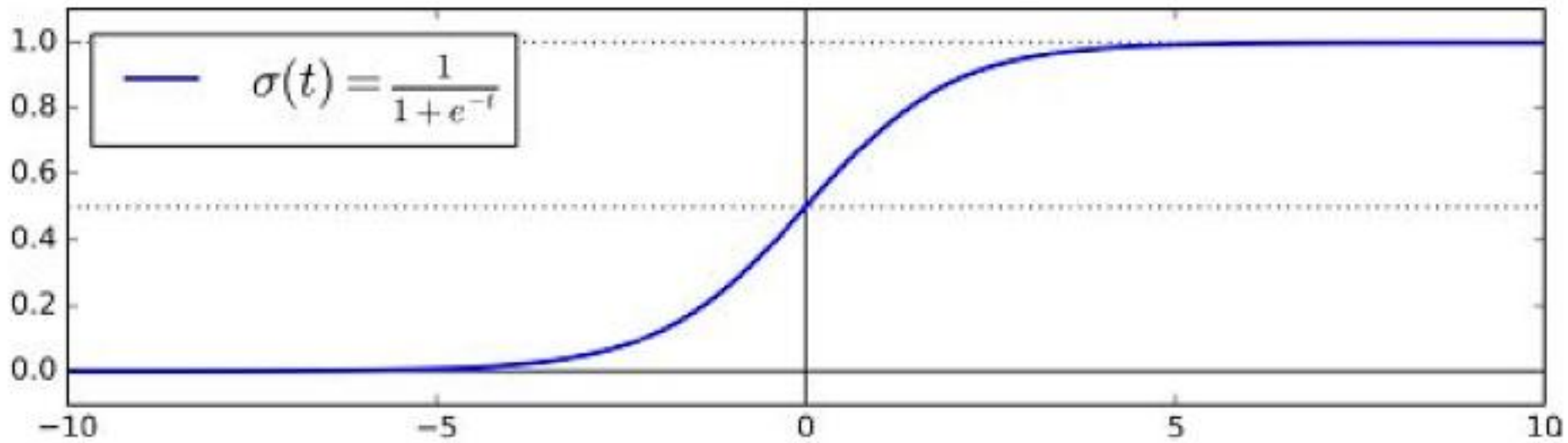


Figure 4-21. Logistic function

Once the Logistic Regression model has estimated the probability $\hat{p} = h(\mathbf{x})$ that an instance $\mathbf{x}$ belongs to the positive class, it can make its prediction easily (see Equation 4-15).

Equation 4-15. Logistic Regression model prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \hat{p} < 0.5, \\ 1 & \text{if } \hat{p} \geq 0.5. \end{cases}$$

Notice that (*t*) < 0.5 when *t* < 0, and (*t*) ≥ 0.5 when *t* ≥ 0, so a Logistic Regression model predicts 1 if ^{T} · **x** is positive, and 0 if it is negative.

Training and Cost Function

Good, now you know how a Logistic Regression model estimates probabilities and makes predictions. But how is it trained? The objective of training is to set the parameter vector so that the model estimates high probabilities for positive instances (*y* = 1) and low probabilities for negative instances (*y* = 0). This idea is captured by the cost function shown in Equation 4-16 for a single training instance **x**.

Equation 4-16. Cost function of a single training instance

$$c(\theta) = \begin{cases} -\log(\hat{p}) & \text{if } y = 1, \\ -\log(1-\hat{p}) & \text{if } y = 0. \end{cases}$$

This cost function makes sense because – log(*t*) grows very large when *t* approaches 0, so the cost will be large if the model estimates a probability close to 0 for a positive instance, and it will also be very large if the model estimates a probability close to 1 for a negative instance. On the other hand, – log(*t*) is close to 0 when *t* is close to 1, so the cost will be close to 0 if the estimated probability is close to 0 for a negative instance or close to 1 for a positive instance, which is precisely what we want.

The cost function over the whole training set is simply the average cost over all training instances. It can be written in a single expression (as you can verify easily), called the *log loss*, shown in Equation 4-17.

Equation 4-17. Logistic Regression cost function (log loss)

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(\hat{p}^{(i)}\right) + \left(1-y^{(i)}\right)\log\left(1-\hat{p}^{(i)}\right)\right]$$

The bad news is that there is no known closed-form equation to compute

the value of that minimizes this cost function (there is no equivalent of the Normal Equation). But the good news is that this cost function is convex, so Gradient Descent (or any other optimization algorithm) is guaranteed to find the global minimum (if the learning rate is not too large and you wait long enough). The partial derivatives of the cost function with regards to the jth model parameter $_j$ is given by Equation 4-18.

Equation 4-18. Logistic cost function partial derivatives

$$\frac{\partial}{\partial \theta_j} \mathrm{J}(\theta) = \frac{1}{m} \sum_{i=1}^{m} (\sigma(\theta^T \cdot \mathbf{x}^{(i)}) - y^{(i)})\, x_j^{(i)}$$

This equation looks very much like Equation 4-5: for each instance it computes the prediction error and multiplies it by the jth feature value, and then it computes the average over all training instances. Once you have the gradient vector containing all the partial derivatives you can use it in the Batch Gradient Descent algorithm. That's it: you now know how to train a Logistic Regression model. For Stochastic GD you would of course just take one instance at a time, and for Mini-batch GD you would use a mini-batch at a time.

Decision Boundaries

Let's use the iris dataset to illustrate Logistic Regression. This is a famous dataset that contains the sepal and petal length and width of 150 iris flowers of three different species: Iris-Setosa, Iris-Versicolor, and Iris-Virginica (see Figure 4-22).

Figure 4-22. Flowers of three iris plant species[16]

Let’s try to build a classifier to detect the Iris-Virginica type based only on the petal width feature. First let’s load the data:

```
>>> from sklearn import datasets
>>> iris = datasets.load_iris()
>>> list(iris.keys())
['data', 'target_names', 'feature_names', 'target', 'DESCR']
>>> X = iris["data"][:, 3:]
>>> y = (iris["target"] == 2).astype(np.int)
```

Now let’s train a Logistic Regression model:

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression()
log_reg.fit(X, y)
```

Let’s look at the model’s estimated probabilities for flowers with petal

```
>>> log_reg.predict([[1.7], [1.5]])
array([1, 0])
```

Figure 4-24 shows the same dataset but this time displaying two features: petal width and length. Once trained, the Logistic Regression classifier can estimate the probability that a new flower is an Iris-Virginica based on these two features. The dashed line represents the points where the model estimates a 50% probability: this is the model's decision boundary. Note that it is a linear boundary.[17] Each parallel line represents the points where the model outputs a specific probability, from 15% (bottom left) to 90% (top right). All the flowers beyond the top-right line have an over 90% chance of being Iris-Virginica according to the model.

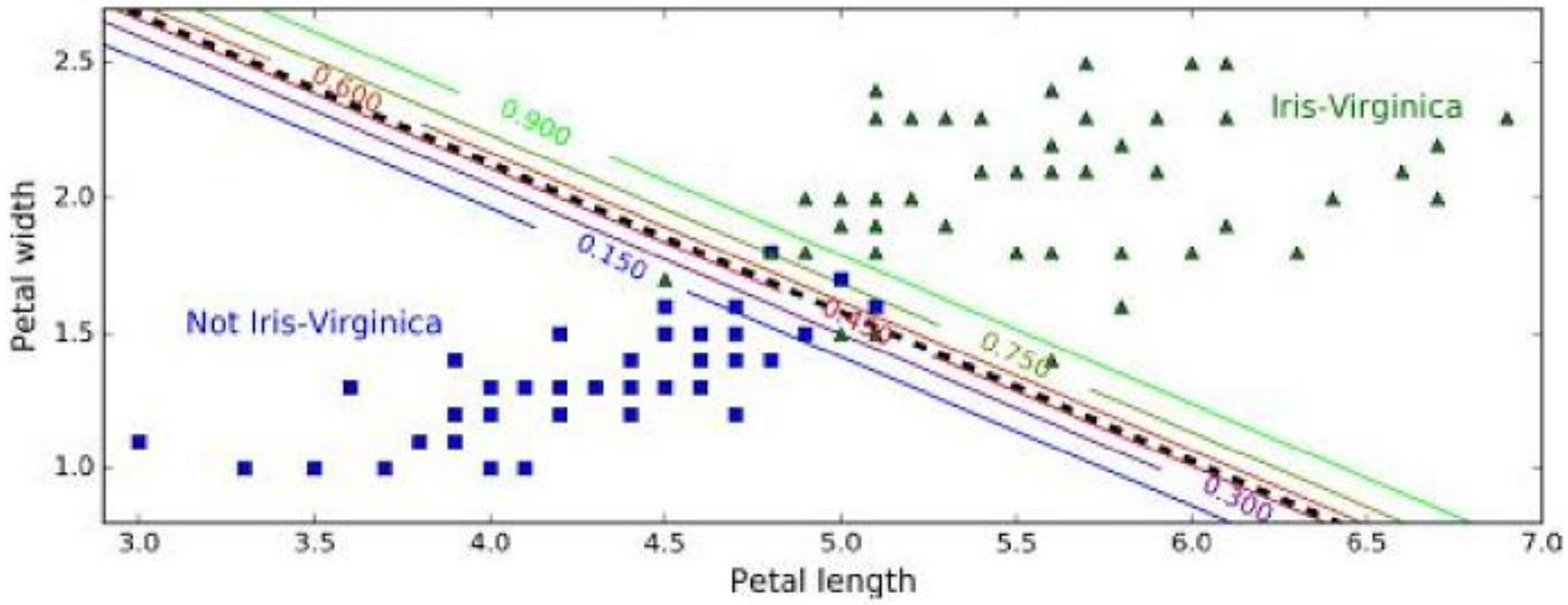


Figure 4-24. Linear decision boundary

Just like the other linear models, Logistic Regression models can be regularized using ℓ_1 or ℓ_2 penalties. Scitkit-Learn actually adds an ℓ_2 penalty by default.

Note

The hyperparameter controlling the regularization strength of a Scikit-Learn `LogisticRegression` model is not `alpha` (as in other linear models), but its inverse: `C`. The higher the value of `C`, the *less* the model is regularized.

- K is the number of classes.
- $\mathbf{s}(\mathbf{x})$ is a vector containing the scores of each class for the instance $\mathbf{x}$.
- $(\mathbf{s}(\mathbf{x}))_k$ is the estimated probability that the instance $\mathbf{x}$ belongs to class k given the scores of each class for that instance.

Just like the Logistic Regression classifier, the Softmax Regression classifier predicts the class with the highest estimated probability (which is simply the class with the highest score), as shown in Equation 4-21.

Equation 4-21. Softmax Regression classifier prediction

$$\hat{y} = \underset{k}{\operatorname{argmax}}\ \sigma(\mathbf{s}(\mathbf{x}))_k = \underset{k}{\operatorname{argmax}}\ s_k(\mathbf{x}) = \underset{k}{\operatorname{argmax}}\ \left((\theta^{(k)})^T \cdot \mathbf{x}\right)$$

- The *argmax* operator returns the value of a variable that maximizes a function. In this equation, it returns the value of k that maximizes the estimated probability $(\mathbf{s}(\mathbf{x}))_k$.

Tip

The Softmax Regression classifier predicts only one class at a time (i.e., it is multiclass, not multioutput) so it should be used only with mutually exclusive classes such as different types of plants. You cannot use it to recognize multiple people in one picture.

Now that you know how the model estimates probabilities and makes predictions, let's take a look at training. The objective is to have a model that estimates a high probability for the target class (and consequently a low probability for the other classes). Minimizing the cost function shown in Equation 4-22, called the *cross entropy*, should lead to this objective because it penalizes the model when it estimates a low probability for a target class. Cross entropy is frequently used to measure how well a set of estimated class probabilities match the target classes (we will use it again several times in the following chapters).

Equation 4-22. Cross entropy cost function

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_k^{(i)} \log\left(\hat{p}_k^{(i)}\right)$$

- $y_k^{(i)}$ is equal to 1 if the target class for the ith instance is *k*; otherwise, it is equal to 0.

Notice that when there are just two classes ($K = 2$), this cost function is equivalent to the Logistic Regression's cost function (log loss; see Equation 4-17).

Cross Entropy

Cross entropy originated from information theory. Suppose you want to efficiently transmit information about the weather every day. If there are eight options (sunny, rainy, etc.), you could encode each option using 3 bits since $2^3 = 8$. However, if you think it will be sunny almost every day, it would be much more efficient to code "sunny" on just one bit (0) and the other seven options on 4 bits (starting with a 1). Cross entropy measures the average number of bits you actually send per option. If your assumption about the weather is perfect, cross entropy will just be equal to the entropy of the weather itself (i.e., its intrinsic unpredictability). But if your assumptions are wrong (e.g., if it rains often), cross entropy will be greater by an amount called the *Kullback–Leibler divergence*.

The cross entropy between two probability distributions *p* and *q* is defined as $H(p, q) = -\sum_x p(x) \log q(x)$ (at least when the distributions are discrete).

The gradient vector of this cost function with regards to $^{(k)}$ is given by Equation 4-23:

Equation 4-23. Cross entropy gradient vector for class k

$$\nabla_{\theta^{(k)}} J(\Theta) = \frac{1}{m} \sum_{i=1}^{m} \left(\hat{p}_k^{(i)} - y_k^{(i)} \right) \mathbf{x}^{(i)}$$

Now you can compute the gradient vector for every class, then use Gradient Descent (or any other optimization algorithm) to find the parameter matrix that minimizes the cost function.

Let's use Softmax Regression to classify the iris flowers into all three classes. Scikit-Learn's `LogisticRegression` uses one-versus-all by default when you train it on more than two classes, but you can set the `multi_class` hyperparameter to `"multinomial"` to switch it to Softmax Regression instead. You must also specify a solver that supports Softmax Regression, such as the `"lbfgs"` solver (see Scikit-Learn's documentation for more details). It also applies ℓ_2 regularization by default, which you can control using the hyperparameter `C`.

```
X = iris["data"][:, (2, 3)]
y = iris["target"]

softmax_reg =
LogisticRegression(multi_class="multinomial",solver="lbfgs", C=10)
softmax_reg.fit(X, y)
```

So the next time you find an iris with 5 cm long and 2 cm wide petals, you can ask your model to tell you what type of iris it is, and it will answer Iris-Virginica (class 2) with 94.2% probability (or Iris-Versicolor with 5.8% probability):

```
>>> softmax_reg.predict([[5, 2]])
array([2])
>>> softmax_reg.predict_proba([[5, 2]])
array([[  6.33134078e-07,   5.75276067e-02,   9.42471760e-01]])
```

Figure 4-25 shows the resulting decision boundaries, represented by the background colors. Notice that the decision boundaries between any two classes are linear. The figure also shows the probabilities for the Iris-Versicolor class, represented by the curved lines (e.g., the line labeled with 0.450 represents the 45% probability boundary). Notice that the model can predict a class that has an estimated probability below 50%. For example, at the point where all decision boundaries meet, all classes have an equal estimated probability of 33%.

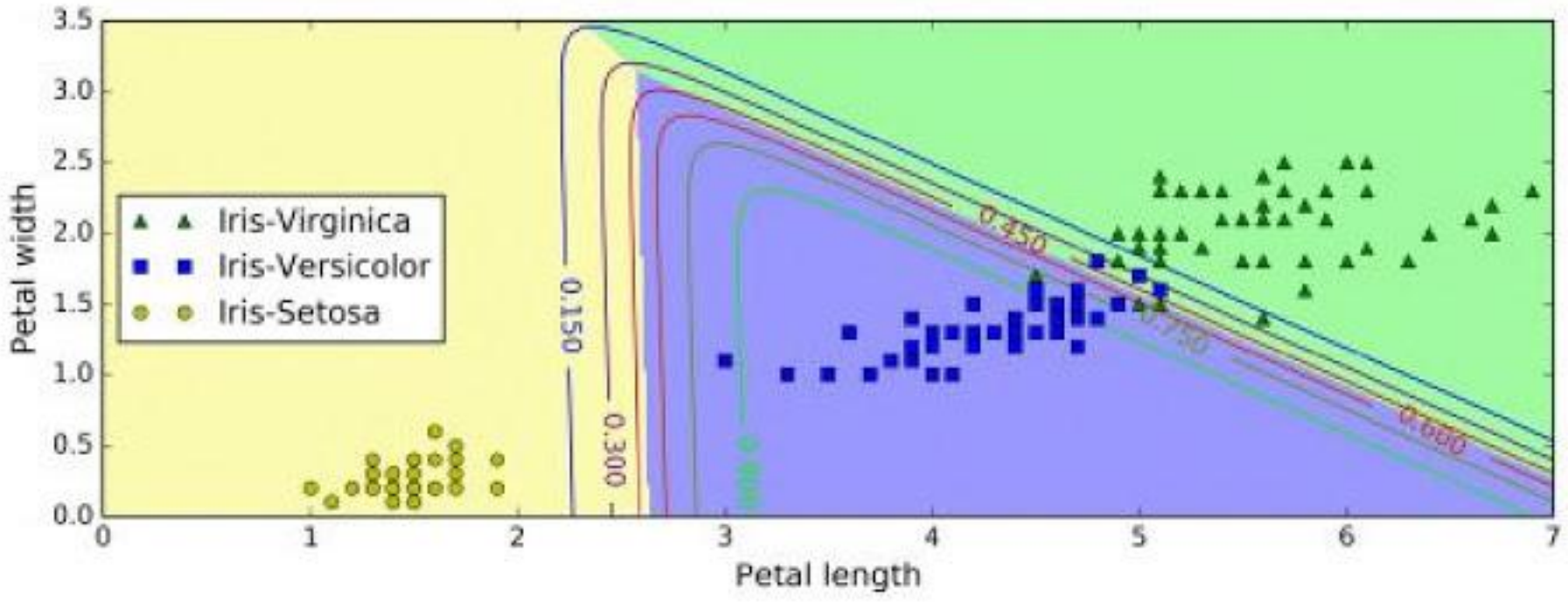


Figure 4-25. Softmax Regression decision boundaries

Exercises

1. What Linear Regression training algorithm can you use if you have a training set with millions of features?

2. Suppose the features in your training set have very different scales. What algorithms might suffer from this, and how? What can you do about it?

3. Can Gradient Descent get stuck in a local minimum when training a Logistic Regression model?

4. Do all Gradient Descent algorithms lead to the same model provided you let them run long enough?

5. Suppose you use Batch Gradient Descent and you plot the validation error at every epoch. If you notice that the validation error consistently goes up, what is likely going on? How can you fix this?

6. Is it a good idea to stop Mini-batch Gradient Descent immediately when the validation error goes up?

7. Which Gradient Descent algorithm (among those we discussed) will reach the vicinity of the optimal solution the fastest? Which will actually converge? How can you make the others converge as well?

8. Suppose you are using Polynomial Regression. You plot the learning curves and you notice that there is a large gap between the training error and the validation error. What is happening? What are three ways to solve this?

9. Suppose you are using Ridge Regression and you notice that the training error and the validation error are almost equal and fairly high. Would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter or reduce it?

0. Why would you want to use:

   - Ridge Regression instead of plain Linear Regression (i.e., without any regularization)?
   - Lasso instead of Ridge Regression?
   - Elastic Net instead of Lasso?

1. Suppose you want to classify pictures as outdoor/indoor and

daytime/nighttime. Should you implement two Logistic Regression classifiers or one Softmax Regression classifier?

2. Implement Batch Gradient Descent with early stopping for Softmax Regression (without using Scikit-Learn).

Solutions to these exercises are available in Appendix A.

[1] It is often the case that a learning algorithm will try to optimize a different function than the performance measure used to evaluate the final model. This is generally because that function is easier to compute, because it has useful differentiation properties that the performance measure lacks, or because we want to constrain the model during training, as we will see when we discuss regularization.

[2] The demonstration that this returns the value of that minimizes the cost function is outside the scope of this book.

[3] Note that Scikit-Learn separates the bias term (`intercept_`) from the feature weights (`coef_`).

[4] Technically speaking, its derivative is *Lipschitz continuous*.

[5] Since feature 1 is smaller, it takes a larger change in $_1$ to affect the cost function, which is why the bowl is elongated along the $_1$ axis.

[6] Eta ( ) is the 7th letter of the Greek alphabet.

[7] Out-of-core algorithms are discussed in Chapter 1.

[8] While the Normal Equation can only perform Linear Regression, the Gradient Descent algorithms can be used to train many other models, as we will see.

[9] A quadratic equation is of the form $y = ax^2 + bx + c$.

[10] This notion of bias is not to be confused with the bias term of linear

models.

[11] It is common to use the notation $J(\quad)$ for cost functions that don't have a short name; we will often use this notation throughout the rest of this book. The context will make it clear which cost function is being discussed.

[12] Norms are discussed in Chapter 2.

[13] A square matrix full of 0s except for 1s on the main diagonal (top-left to bottom-right).

[14] Alternatively you can use the `Ridge` class with the `"sag"` solver. Stochastic Average GD is a variant of SGD. For more details, see the presentation "Minimizing Finite Sums with the Stochastic Average Gradient Algorithm" by Mark Schmidt et al. from the University of British Columbia.

[15] You can think of a subgradient vector at a nondifferentiable point as an intermediate vector between the gradient vectors around that point.

[16] Photos reproduced from the corresponding Wikipedia pages. Iris-Virginica photo by Frank Mayfield (Creative Commons BY-SA 2.0), Iris-Versicolor photo by D. Gordon E. Robertson (Creative Commons BY-SA 3.0), and Iris-Setosa photo is public domain.

[17] It is the the set of points $\mathbf{x}$ such that $\quad_0 + \quad_1 x_1 + \quad_2 x_2 = 0$, which defines a straight line.

Chapter 5. Support Vector Machines

A *Support Vector Machine* (SVM) is a very powerful and versatile Machine Learning model, capable of performing linear or nonlinear classification, regression, and even outlier detection. It is one of the most popular models in Machine Learning, and anyone interested in Machine Learning should have it in their toolbox. SVMs are particularly well suited for classification of complex but small- or medium-sized datasets.

This chapter will explain the core concepts of SVMs, how to use them, and how they work.

Linear SVM Classification

The fundamental idea behind SVMs is best explained with some pictures. Figure 5-1 shows part of the iris dataset that was introduced at the end of Chapter 4. The two classes can clearly be separated easily with a straight line (they are *linearly separable*). The left plot shows the decision boundaries of three possible linear classifiers. The model whose decision boundary is represented by the dashed line is so bad that it does not even separate the classes properly. The other two models work perfectly on this training set, but their decision boundaries come so close to the instances that these models will probably not perform as well on new instances. In contrast, the solid line in the plot on the right represents the decision boundary of an SVM classifier; this line not only separates the two classes but also stays as far away from the closest training instances as possible. You can think of an SVM classifier as fitting the widest possible street (represented by the parallel dashed lines) between the classes. This is called *large margin classification*.

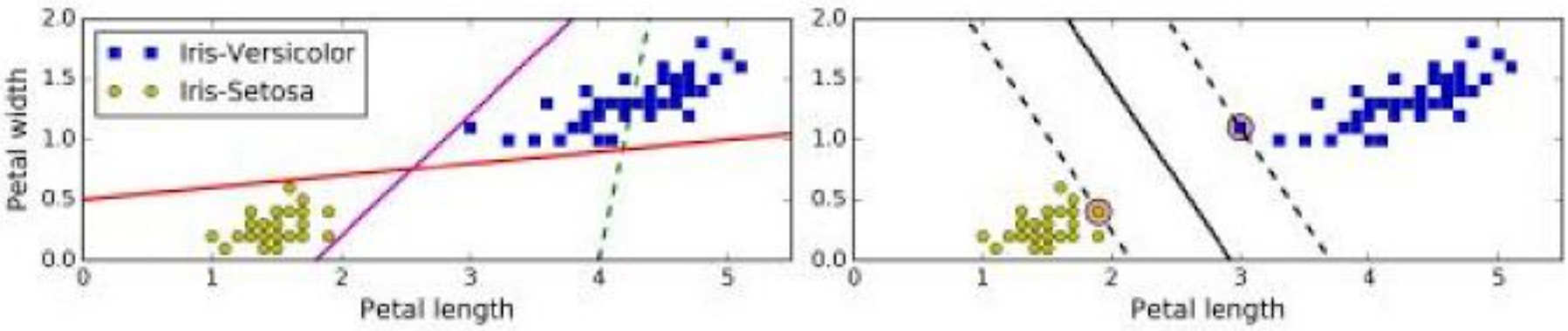


Figure 5-1. Large margin classification

Notice that adding more training instances "off the street" will not affect the decision boundary at all: it is fully determined (or "supported") by the instances located on the edge of the street. These instances are called the *support vectors* (they are circled in Figure 5-1).

Warning

SVMs are sensitive to the feature scales, as you can see in Figure 5-2: on the left plot, the vertical scale is much larger than the horizontal scale, so the widest possible street is close to horizontal. After feature scaling (e.g., using Scikit-Learn's `StandardScaler`), the decision boundary looks much better (on the right plot).

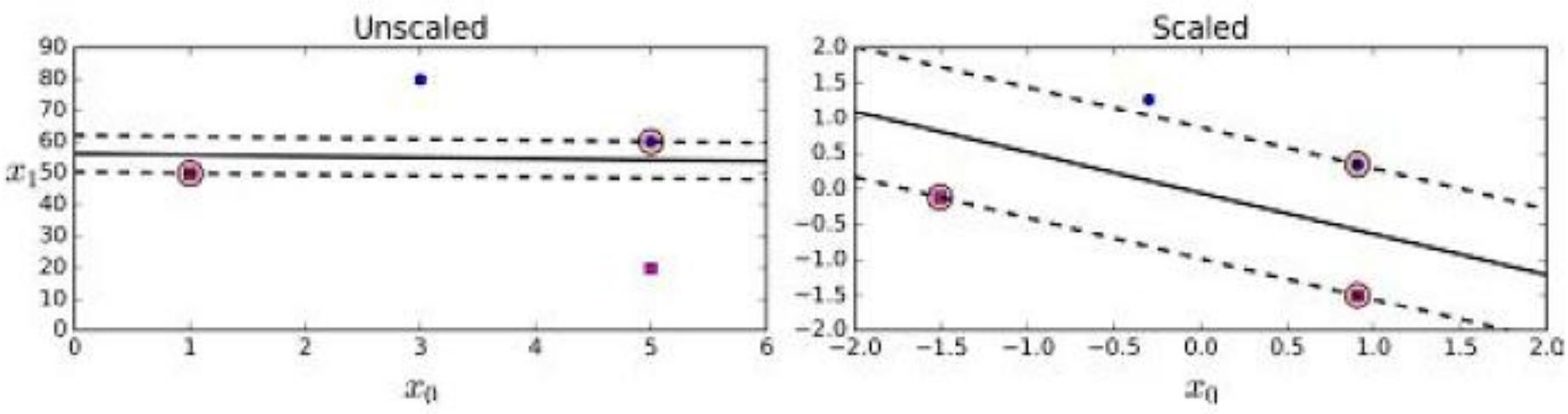


Figure 5-2. Sensitivity to feature scales

Soft Margin Classification

If we strictly impose that all instances be off the street and on the right side, this is called *hard margin classification*. There are two main issues with hard margin classification. First, it only works if the data is linearly separable, and second it is quite sensitive to outliers. Figure 5-3 shows the

iris dataset with just one additional outlier: on the left, it is impossible to find a hard margin, and on the right the decision boundary ends up very different from the one we saw in Figure 5-1 without the outlier, and it will probably not generalize as well.

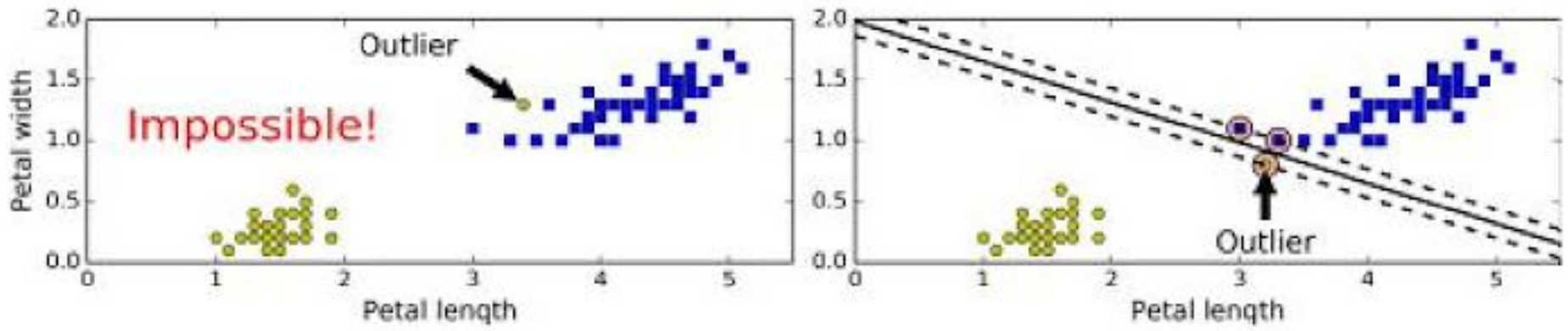


Figure 5-3. Hard margin sensitivity to outliers

To avoid these issues it is preferable to use a more flexible model. The objective is to find a good balance between keeping the street as large as possible and limiting the *margin violations* (i.e., instances that end up in the middle of the street or even on the wrong side). This is called *soft margin classification*.

In Scikit-Learn's SVM classes, you can control this balance using the `C` hyperparameter: a smaller `C` value leads to a wider street but more margin violations. Figure 5-4 shows the decision boundaries and margins of two soft margin SVM classifiers on a nonlinearly separable dataset. On the left, using a high `C` value the classifier makes fewer margin violations but ends up with a smaller margin. On the right, using a low `C` value the margin is much larger, but many instances end up on the street. However, it seems likely that the second classifier will generalize better: in fact even on this training set it makes fewer prediction errors, since most of the margin violations are actually on the correct side of the decision boundary.

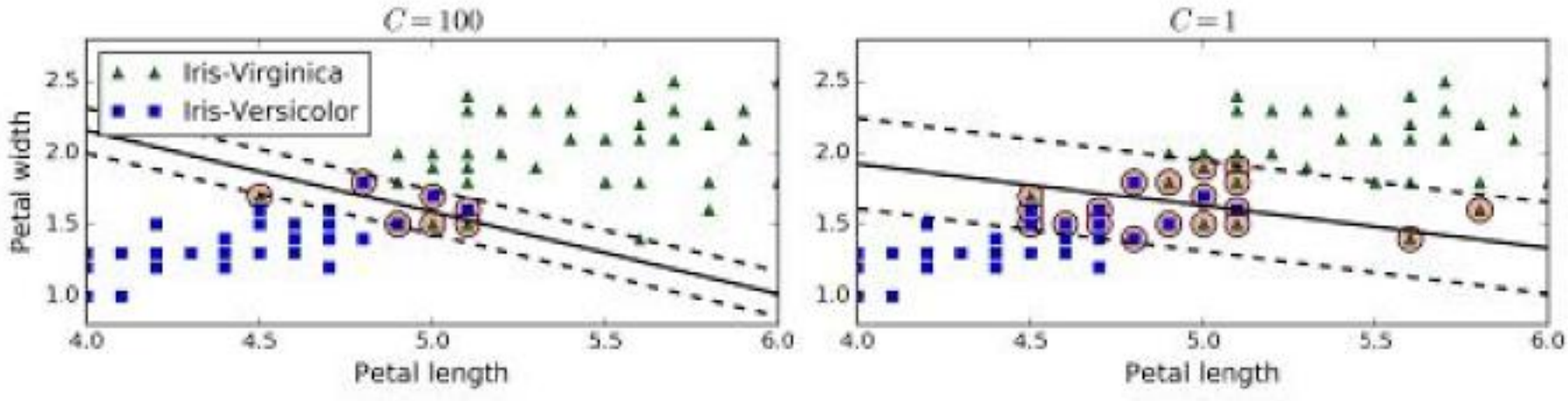


Figure 5-4. Fewer margin violations versus large margin

Tip

If your SVM model is overfitting, you can try regularizing it by reducing `C`.

The following Scikit-Learn code loads the iris dataset, scales the features, and then trains a linear SVM model (using the `LinearSVC` class with $C = 1$ and the *hinge loss* function, described shortly) to detect Iris-Virginica flowers. The resulting model is represented on the right of Figure 5-4.

```
import numpy as np
from sklearn import datasets
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC

iris = datasets.load_iris()
X = iris["data"][:, (2, 3)]
y = (iris["target"] == 2).astype(np.float64)

svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("linear_svc", LinearSVC(C=1, loss="hinge")),
    ))

svm_clf.fit(X, y)
```

Then, as usual, you can use the model to make predictions:

```
>>> svm_clf.predict([[5.5, 1.7]])
array([ 1.])
```

Note

Unlike Logistic Regression classifiers, SVM classifiers do not output probabilities for each class.

Alternatively, you could use the `SVC` class, using `SVC(kernel="linear", C=1)`, but it is much slower, especially with large training sets, so it is not recommended. Another option is to use the `SGDClassifier` class, with `SGDClassifier(loss="hinge", alpha=1/(m*C))`. This applies regular Stochastic Gradient Descent (see Chapter 4) to train a linear SVM classifier. It does not converge as fast as the `LinearSVC` class, but it can be useful to handle huge datasets that do not fit in memory (out-of-core training), or to handle online classification tasks.

Tip

The `LinearSVC` class regularizes the bias term, so you should center the training set first by subtracting its mean. This is automatic if you scale the data using the `StandardScaler`. Moreover, make sure you set the `loss` hyperparameter to `"hinge"`, as it is not the default value. Finally, for better performance you should set the `dual` hyperparameter to `False`, unless there are more features than training instances (we will discuss duality later in the chapter).

Nonlinear SVM Classification

Although linear SVM classifiers are efficient and work surprisingly well in many cases, many datasets are not even close to being linearly separable. One approach to handling nonlinear datasets is to add more features, such as polynomial features (as you did in Chapter 4); in some cases this can result in a linearly separable dataset. Consider the left plot in Figure 5-5: it represents a simple dataset with just one feature x_1. This dataset is not linearly separable, as you can see. But if you add a second feature x_2 =

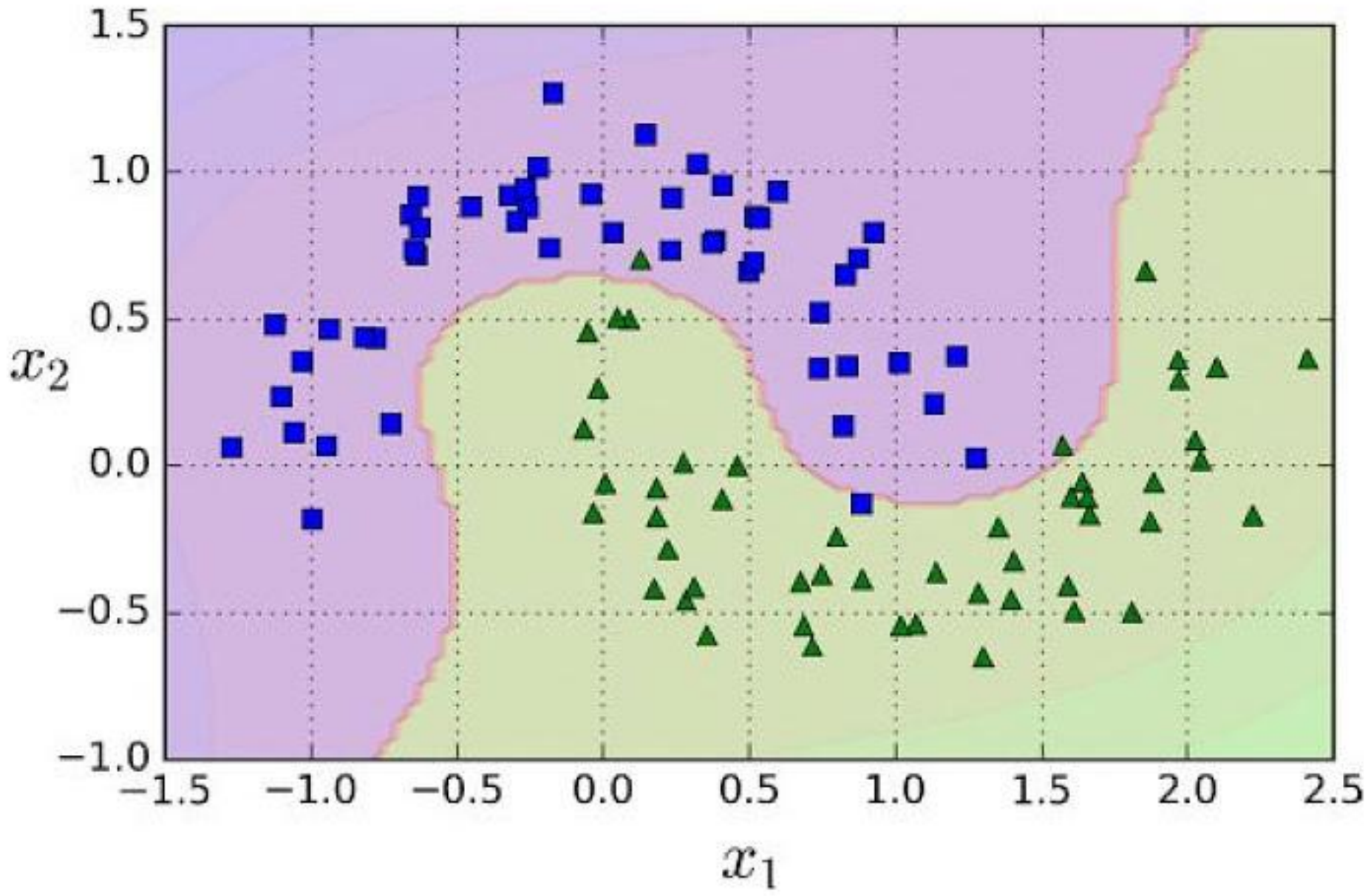


Figure 5-6. Linear SVM classifier using polynomial features

Polynomial Kernel

Adding polynomial features is simple to implement and can work great with all sorts of Machine Learning algorithms (not just SVMs), but at a low polynomial degree it cannot deal with very complex datasets, and with a high polynomial degree it creates a huge number of features, making the model too slow.

Fortunately, when using SVMs you can apply an almost miraculous mathematical technique called the *kernel trick* (it is explained in a moment). It makes it possible to get the same result as if you added many polynomial features, even with very high-degree polynomials, without actually having to add them. So there is no combinatorial explosion of the number of features since you don't actually add any features. This trick is implemented by the `SVC` class. Let's test it on the moons dataset:

```
from sklearn.svm import SVC
poly_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="poly", degree=3, coef0=1, C=5))
    ))
poly_kernel_svm_clf.fit(X, y)
```

This code trains an SVM classifier using a 3rd-degree polynomial kernel. It is represented on the left of Figure 5-7. On the right is another SVM classifier using a 10th-degree polynomial kernel. Obviously, if your model is overfitting, you might want to reduce the polynomial degree. Conversely, if it is underfitting, you can try increasing it. The hyperparameter `coef0` controls how much the model is influenced by high-degree polynomials versus low-degree polynomials.

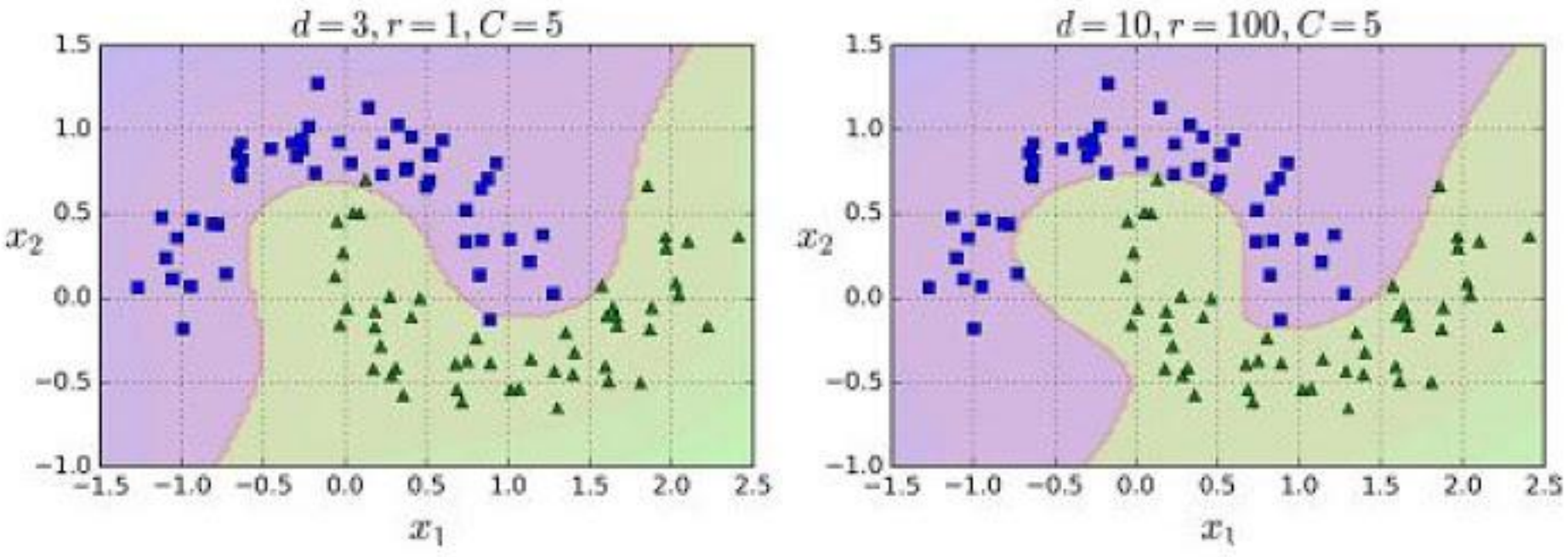


Figure 5-7. SVM classifiers with a polynomial kernel

Tip

A common approach to find the right hyperparameter values is to use grid search (see Chapter 2). It is often faster to first do a very coarse grid search, then a finer grid search around the best values found. Having a good sense of what each hyperparameter actually does can also help you search in the right part of the hyperparameter space.

Adding Similarity Features

Another technique to tackle nonlinear problems is to add features computed using a *similarity function* that measures how much each instance resembles a particular *landmark*. For example, let's take the one-dimensional dataset discussed earlier and add two landmarks to it at $x_1 = -2$ and $x_1 = 1$ (see the left plot in Figure 5-8). Next, let's define the similarity function to be the Gaussian *Radial Basis Function* (*RBF*) with $\gamma = 0.3$ (see Equation 5-1).

Equation 5-1. Gaussian RBF

$$\phi_\gamma(\mathbf{x}, \boldsymbol{\ell}) = \exp\left(-\gamma \| \mathbf{x} - \boldsymbol{\ell} \|^2\right)$$

It is a bell-shaped function varying from 0 (very far away from the landmark) to 1 (at the landmark). Now we are ready to compute the new features. For example, let's look at the instance $x_1 = -1$: it is located at a distance of 1 from the first landmark, and 2 from the second landmark. Therefore its new features are $x_2 = \exp(-0.3 \times 1^2) \approx 0.74$ and $x_3 = \exp(-0.3 \times 2^2) \approx 0.30$. The plot on the right of Figure 5-8 shows the transformed dataset (dropping the original features). As you can see, it is now linearly separable.

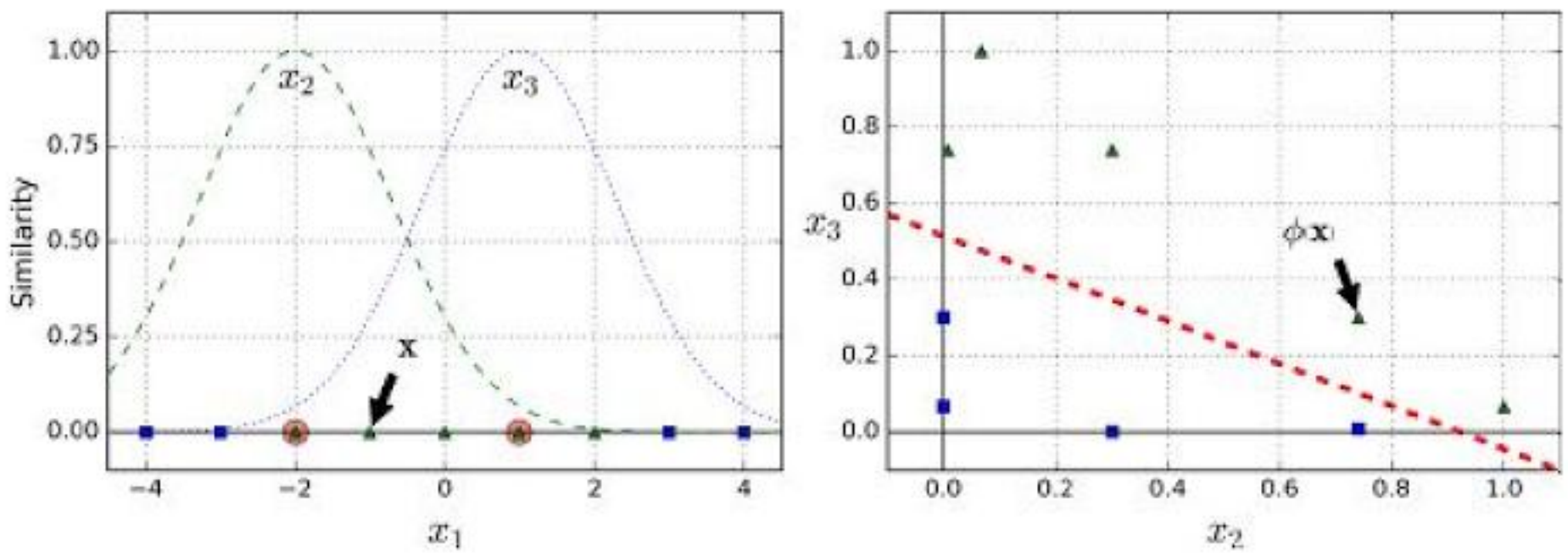


Figure 5-8. Similarity features using the Gaussian RBF

You may wonder how to select the landmarks. The simplest approach is to create a landmark at the location of each and every instance in the dataset.

This creates many dimensions and thus increases the chances that the transformed training set will be linearly separable. The downside is that a training set with *m* instances and *n* features gets transformed into a training set with *m* instances and *m* features (assuming you drop the original features). If your training set is very large, you end up with an equally large number of features.

Gaussian RBF Kernel

Just like the polynomial features method, the similarity features method can be useful with any Machine Learning algorithm, but it may be computationally expensive to compute all the additional features, especially on large training sets. However, once again the kernel trick does its SVM magic: it makes it possible to obtain a similar result as if you had added many similarity features, without actually having to add them. Let's try the Gaussian RBF kernel using the `SVC` class:

```
rbf_kernel_svm_clf = Pipeline((
        ("scaler", StandardScaler()),
        ("svm_clf", SVC(kernel="rbf", gamma=5, C=0.001))
    ))
rbf_kernel_svm_clf.fit(X, y)
```

This model is represented on the bottom left of Figure 5-9. The other plots show models trained with different values of hyperparameters `gamma` ( ) and *C*. Increasing `gamma` makes the bell-shape curve narrower (see the left plot of Figure 5-8), and as a result each instance's range of influence is smaller: the decision boundary ends up being more irregular, wiggling around individual instances. Conversely, a small `gamma` value makes the bell-shaped curve wider, so instances have a larger range of influence, and the decision boundary ends up smoother. So acts like a regularization hyperparameter: if your model is overfitting, you should reduce it, and if it is underfitting, you should increase it (similar to the `C` hyperparameter).

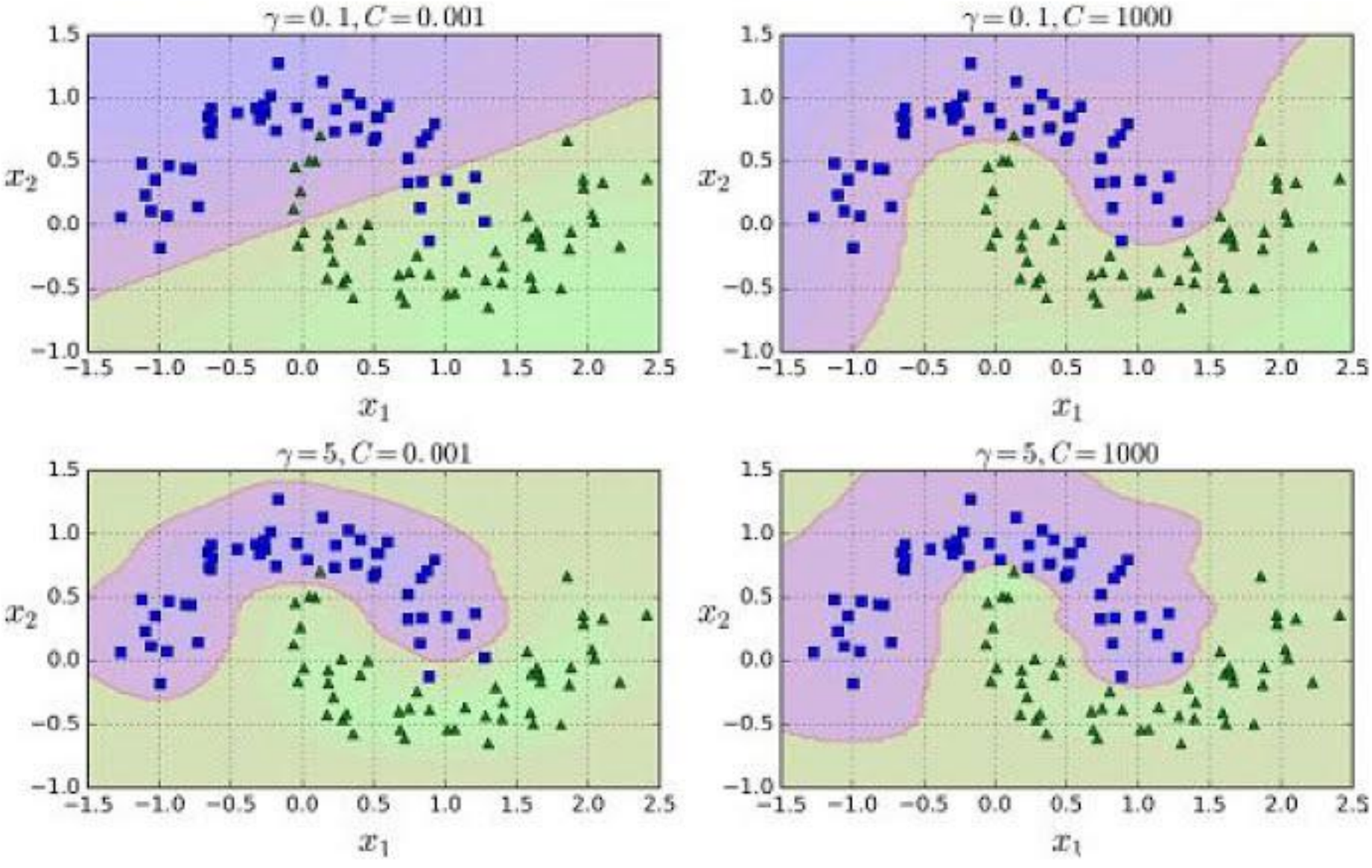


Figure 5-9. SVM classifiers using an RBF kernel

Other kernels exist but are used much more rarely. For example, some kernels are specialized for specific data structures. *String kernels* are sometimes used when classifying text documents or DNA sequences (e.g., using the *string subsequence kernel* or kernels based on the *Levenshtein distance*).

Tip

With so many kernels to choose from, how can you decide which one to use? As a rule of thumb, you should always try the linear kernel first (remember that `LinearSVC` is much faster than `SVC(kernel="linear")`), especially if the training set is very large or if it has plenty of features. If the training set is not too large, you should try the Gaussian RBF kernel as well; it works well in most cases. Then if you have spare time and computing power, you can also experiment with a few other kernels using cross-validation and grid search, especially if there are kernels specialized for your training set's data structure.

Computational Complexity

The `LinearSVC` class is based on the *liblinear* library, which implements an optimized algorithm for linear SVMs.[1] It does not support the kernel trick, but it scales almost linearly with the number of training instances and the number of features: its training time complexity is roughly $O(m \times n)$.

The algorithm takes longer if you require a very high precision. This is controlled by the tolerance hyperparameter (called `tol` in Scikit-Learn). In most classification tasks, the default tolerance is fine.

The `SVC` class is based on the *libsvm* library, which implements an algorithm that supports the kernel trick.[2] The training time complexity is usually between $O(m^2 \times n)$ and $O(m^3 \times n)$. Unfortunately, this means that it gets dreadfully slow when the number of training instances gets large (e.g., hundreds of thousands of instances). This algorithm is perfect for complex but small or medium training sets. However, it scales well with the number of features, especially with *sparse features* (i.e., when each instance has few nonzero features). In this case, the algorithm scales roughly with the average number of nonzero features per instance. Table 5-1 compares Scikit-Learn's SVM classification classes.

Table 5-1. Comparison of Scikit-Learn classes for SVM classification

Class	Time complexity	Out-of-core support	Scaling required	Kernel trick
`LinearSVC`	$O(m \times n)$	No	Yes	No
`SGDClassifier`	$O(m \times n)$	Yes	Yes	No
`SVC`	$O(m^2 \times n)$ to $O(m^3 \times n)$	No	Yes	Yes

SVM Regression

As we mentioned earlier, the SVM algorithm is quite versatile: not only does it support linear and nonlinear classification, but it also supports linear and nonlinear regression. The trick is to reverse the objective: instead of trying to fit the largest possible street between two classes while limiting margin violations, SVM Regression tries to fit as many instances as possible *on* the street while limiting margin violations (i.e., instances *off* the street). The width of the street is controlled by a hyperparameter .

Figure 5-10 shows two linear SVM Regression models trained on some random linear data, one with a large margin (= 1.5) and the other with a small margin (= 0.5).

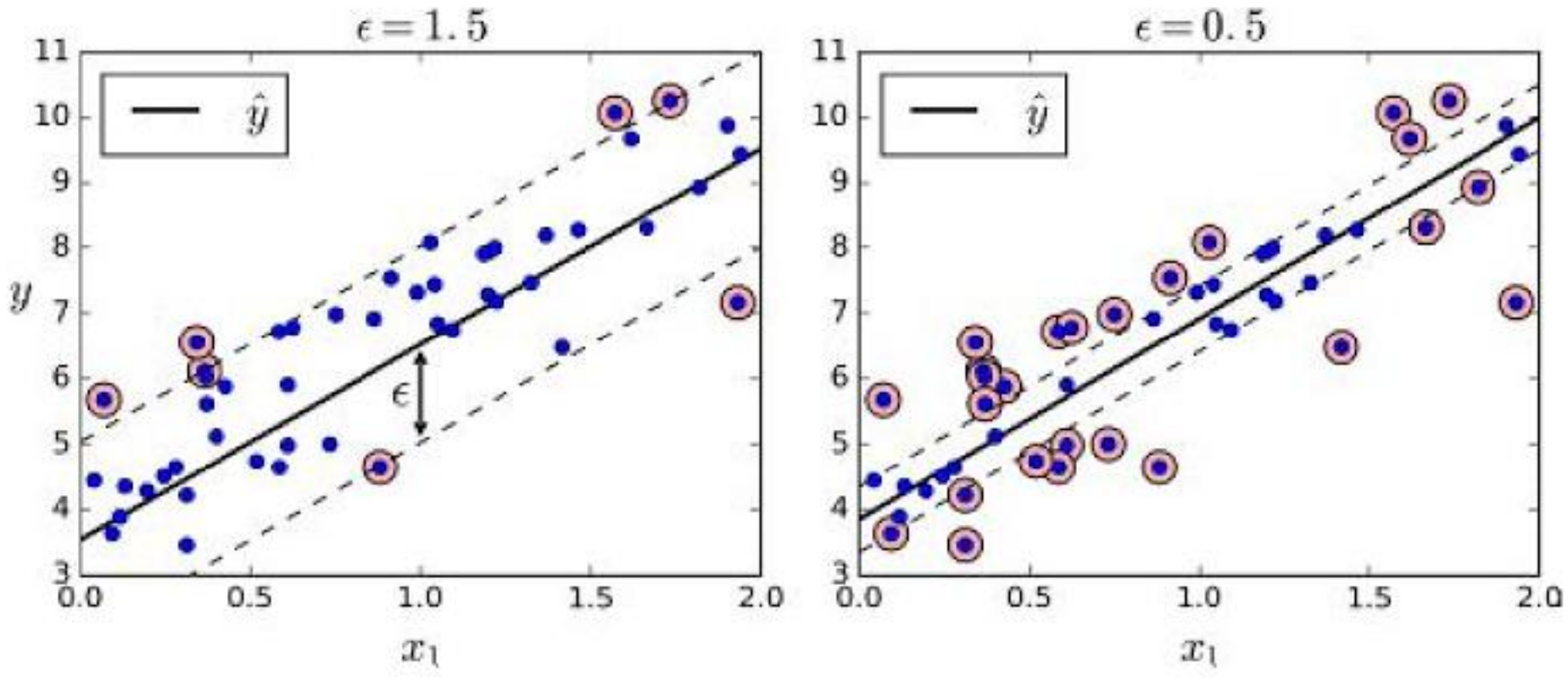


Figure 5-10. SVM Regression

Adding more training instances within the margin does not affect the model's predictions; thus, the model is said to be *-insensitive*.

You can use Scikit-Learn's `LinearSVR` class to perform linear SVM Regression. The following code produces the model represented on the left of Figure 5-10 (the training data should be scaled and centered first):

```
from sklearn.svm import LinearSVR
```

```
svm_reg = LinearSVR(epsilon=1.5)
svm_reg.fit(X, y)
```

To tackle nonlinear regression tasks, you can use a kernelized SVM model. For example, Figure 5-11 shows SVM Regression on a random quadratic training set, using a 2nd-degree polynomial kernel. There is little regularization on the left plot (i.e., a large C value), and much more regularization on the right plot (i.e., a small C value).

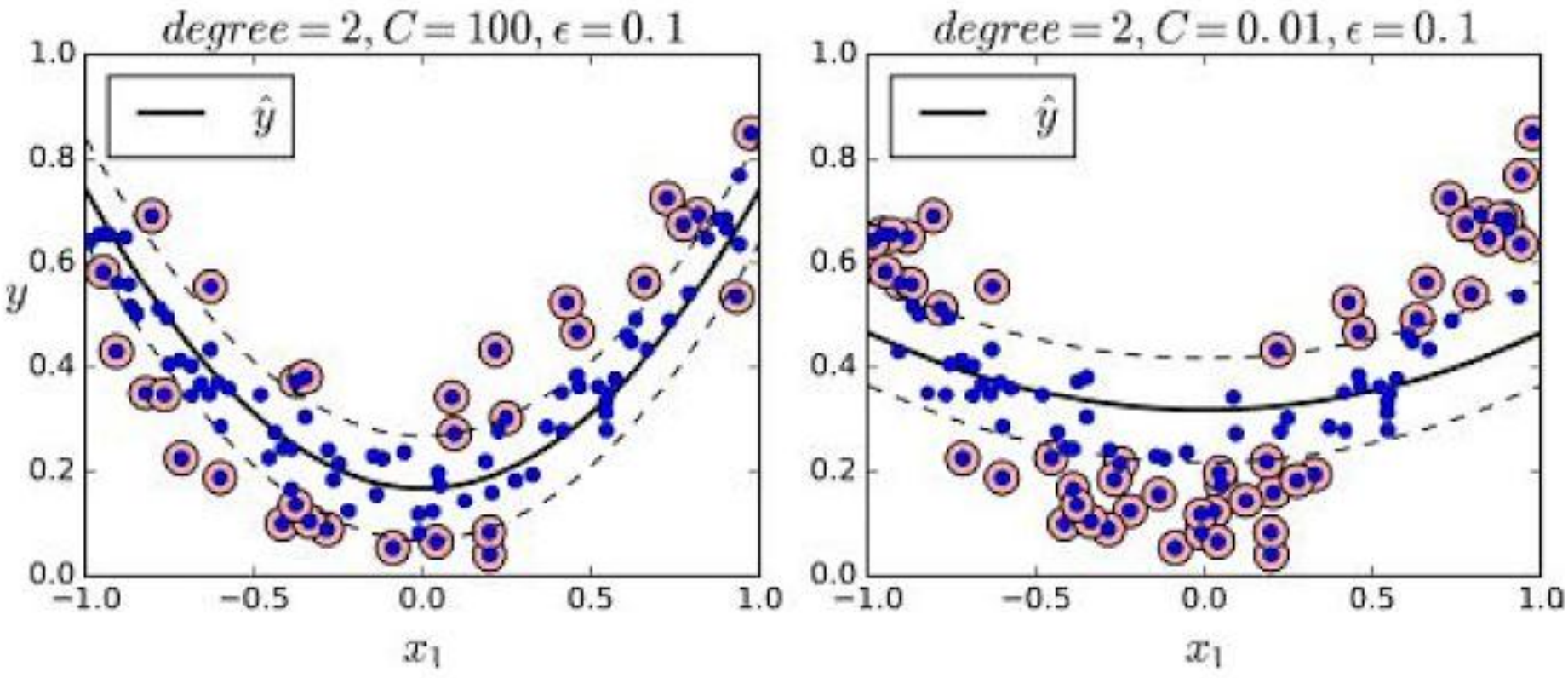


Figure 5-11. SVM regression using a 2nd-degree polynomial kernel

The following code produces the model represented on the left of Figure 5-11 using Scikit-Learn's SVR class (which supports the kernel trick). The SVR class is the regression equivalent of the SVC class, and the LinearSVR class is the regression equivalent of the LinearSVC class. The LinearSVR class scales linearly with the size of the training set (just like the LinearSVC class), while the SVR class gets much too slow when the training set grows large (just like the SVC class).

```
from sklearn.svm import SVR
```

```
svm_poly_reg = SVR(kernel="poly", degree=2, C=100, epsilon=0.1)
svm_poly_reg.fit(X, y)
```

Note

SVMs can also be used for outlier detection; see Scikit-Learn's documentation for more details.

Under the Hood

This section explains how SVMs make predictions and how their training algorithms work, starting with linear SVM classifiers. You can safely skip it and go straight to the exercises at the end of this chapter if you are just getting started with Machine Learning, and come back later when you want to get a deeper understanding of SVMs.

First, a word about notations: in Chapter 4 we used the convention of putting all the model parameters in one vector , including the bias term $_0$ and the input feature weights $_1$ to $_n$, and adding a bias input $x_0 = 1$ to all instances. In this chapter, we will use a different convention, which is more convenient (and more common) when you are dealing with SVMs: the bias term will be called b and the feature weights vector will be called $\mathbf{w}$. No bias feature will be added to the input feature vectors.

Decision Function and Predictions

The linear SVM classifier model predicts the class of a new instance $\mathbf{x}$ by simply computing the decision function $\mathbf{w}^T \cdot \mathbf{x} + b = w_1 x_1 + \cdots + w_n x_n + b$: if the result is positive, the predicted class is the positive class (1), or else it is the negative class (0); see Equation 5-2.

Equation 5-2. Linear SVM classifier prediction

$$\hat{y} = \begin{cases} 0 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b < 0, \\ 1 & \text{if } \mathbf{w}^T \cdot \mathbf{x} + b \geq 0 \end{cases}$$

Figure 5-12 shows the decision function that corresponds to the model on the right of Figure 5-4: it is a two-dimensional plane since this dataset has two features (petal width and petal length). The decision boundary is the set of points where the decision function is equal to 0: it is the intersection of two planes, which is a straight line (represented by the thick solid line).[3]

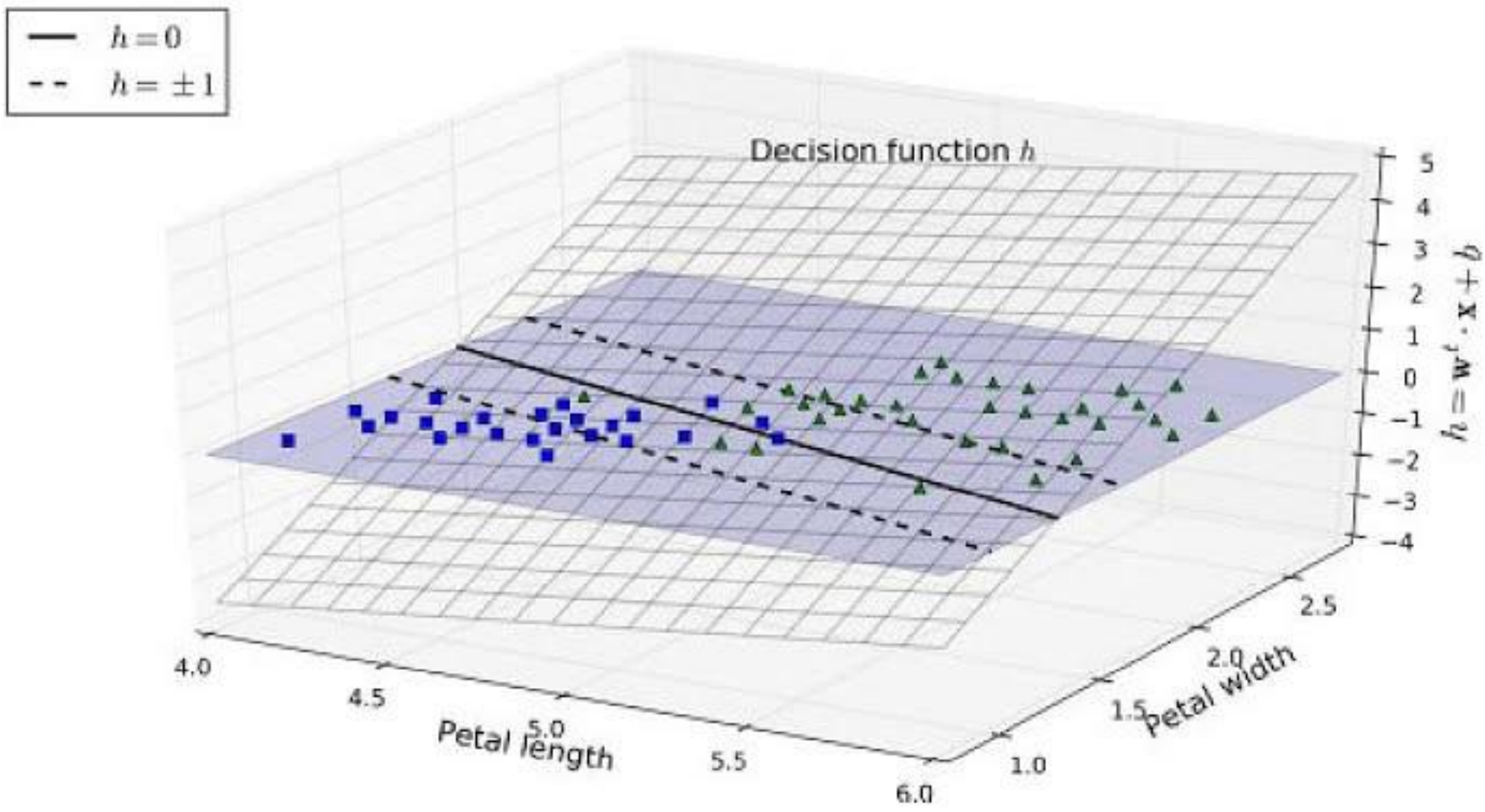


Figure 5-12. Decision function for the iris dataset

The dashed lines represent the points where the decision function is equal to 1 or –1: they are parallel and at equal distance to the decision boundary, forming a margin around it. Training a linear SVM classifier means finding the value of **w** and *b* that make this margin as wide as possible while avoiding margin violations (hard margin) or limiting them (soft margin).

Training Objective

Consider the slope of the decision function: it is equal to the norm of the weight vector, $\| \mathbf{w} \|$. If we divide this slope by 2, the points where the decision function is equal to ±1 are going to be twice as far away from the decision boundary. In other words, dividing the slope by 2 will multiply the margin by 2. Perhaps this is easier to visualize in 2D in Figure 5-13. The

smaller the weight vector **w**, the larger the margin.

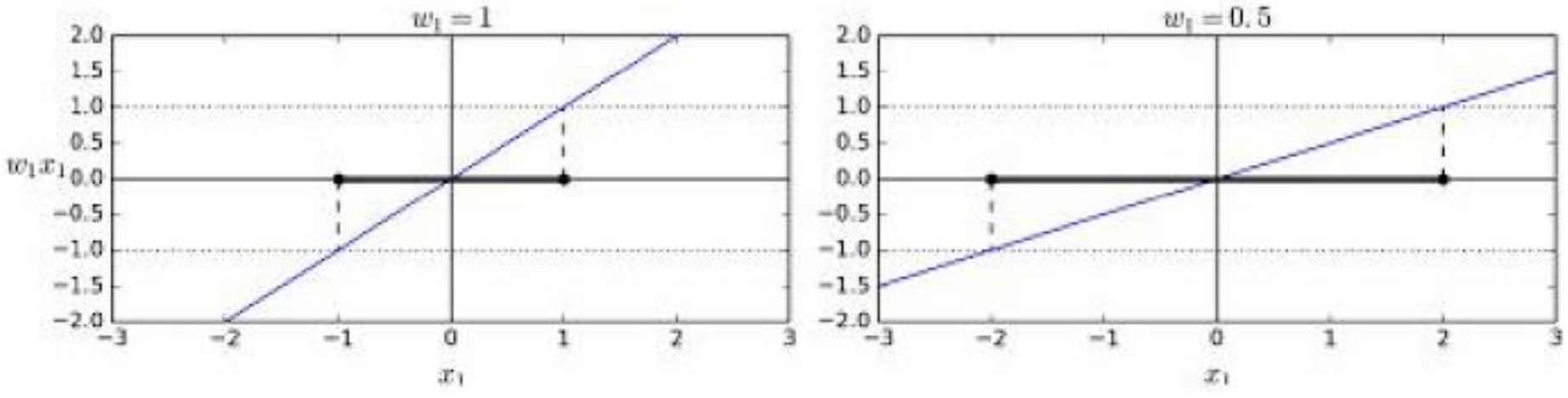


Figure 5-13. A smaller weight vector results in a larger margin

So we want to minimize $\| \mathbf{w} \|$ to get a large margin. However, if we also want to avoid any margin violation (hard margin), then we need the decision function to be greater than 1 for all positive training instances, and lower than –1 for negative training instances. If we define $t^{(i)} = -1$ for negative instances (if $y^{(i)} = 0$) and $t^{(i)} = 1$ for positive instances (if $y^{(i)} = 1$), then we can express this constraint as $t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1$ for all instances.

We can therefore express the hard margin linear SVM classifier objective as the *constrained optimization* problem in Equation 5-3.

Equation 5-3. Hard margin linear SVM classifier objective

$$\underset{\mathbf{w},b}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$$

$$\text{subject to} \quad t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 \quad \text{for } i = 1, 2, \cdots, m$$

Note

We are minimizing $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$, which is equal to $\frac{1}{2}\| \mathbf{w} \|^2$, rather than minimizing $\| \mathbf{w} \|$. This is because it will give the same result (since the values of **w** and *b* that minimize a value also minimize half of its square), but $\frac{1}{2}\| \mathbf{w} \|^2$ has a nice and simple derivative (it is just **w**) while $\| \mathbf{w} \|$ is not differentiable at $\mathbf{w} = \mathbf{0}$. Optimization algorithms work much better on differentiable functions.

To get the soft margin objective, we need to introduce a *slack variable* $\zeta^{(i)} \geq 0$ for each instance:[4] $\zeta^{(i)}$ measures how much the ith instance is allowed to violate the margin. We now have two conflicting objectives: making the slack variables as small as possible to reduce the margin violations, and making $\frac{1}{2}\mathbf{w}^T \cdot \mathbf{w}$ as small as possible to increase the margin. This is where the C hyperparameter comes in: it allows us to define the tradeoff between these two objectives. This gives us the constrained optimization problem in Equation 5-4.

Equation 5-4. Soft margin linear SVM classifier objective

$$\underset{\mathbf{w},b,\zeta}{\text{minimize}} \quad \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} + C\sum_{i=1}^{m}\zeta^{(i)}$$

$$\text{subject to} \quad t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) \geq 1 - \zeta^{(i)} \quad \text{and} \quad \zeta^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

Quadratic Programming

The hard margin and soft margin problems are both convex quadratic optimization problems with linear constraints. Such problems are known as *Quadratic Programming* (QP) problems. Many off-the-shelf solvers are available to solve QP problems using a variety of techniques that are outside the scope of this book.[5] The general problem formulation is given by Equation 5-5.

Equation 5-5. Quadratic Programming problem

$$\underset{\mathbf{p}}{\text{Minimize}} \quad \frac{1}{2}\mathbf{p}^T \cdot \mathbf{H} \cdot \mathbf{p} \quad + \quad \mathbf{f}^T \cdot \mathbf{p}$$

$$\text{subject to} \quad \mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$$

$$\text{where} \begin{cases} \mathbf{p} & \text{is an } n_p\text{-dimensional vector } (n_p = \text{number of parameters}), \\ \mathbf{H} & \text{is an } n_p \times n_p \text{ matrix}, \\ \mathbf{f} & \text{is an } n_p\text{-dimensional vector}, \\ \mathbf{A} & \text{is an } n_c \times n_p \text{ matrix } (n_c = \text{number of constraints}), \\ \mathbf{b} & \text{is an } n_c\text{-dimensional vector}. \end{cases}$$

Note that the expression $\mathbf{A} \cdot \mathbf{p} \leq \mathbf{b}$ actually defines n_c constraints: $\mathbf{p}^T \cdot \mathbf{a}^{(i)} \leq b^{(i)}$ for $i = 1, 2, \cdots, n_c$, where $\mathbf{a}^{(i)}$ is the vector containing the elements of the

i^{th} row of **A** and $b^{(i)}$ is the i^{th} element of **b**.

You can easily verify that if you set the QP parameters in the following way, you get the hard margin linear SVM classifier objective:

- $n_p = n + 1$, where n is the number of features (the +1 is for the bias term).
- $n_c = m$, where m is the number of training instances.
- **H** is the $n_p \times n_p$ identity matrix, except with a zero in the top-left cell (to ignore the bias term).
- $\mathbf{f} = \mathbf{0}$, an n_p-dimensional vector full of 0s.
- $\mathbf{b} = \mathbf{1}$, an n_c-dimensional vector full of 1s.
- $\mathbf{a}^{(i)} = -t^{(i)} \dot{\mathbf{X}}^{(i)}$, where $\dot{\mathbf{X}}^{(i)}$ is equal to $\mathbf{x}^{(i)}$ with an extra bias feature $\dot{\mathbf{X}}_0 = 1$.

So one way to train a hard margin linear SVM classifier is just to use an off-the-shelf QP solver by passing it the preceding parameters. The resulting vector **p** will contain the bias term $b = p_0$ and the feature weights $w_i = p_i$ for $i = 1, 2, \cdots, m$. Similarly, you can use a QP solver to solve the soft margin problem (see the exercises at the end of the chapter).

However, to use the kernel trick we are going to look at a different constrained optimization problem.

The Dual Problem

Given a constrained optimization problem, known as the *primal problem*, it is possible to express a different but closely related problem, called its *dual problem*. The solution to the dual problem typically gives a lower bound to the solution of the primal problem, but under some conditions it can even have the same solutions as the primal problem. Luckily, the SVM problem happens to meet these conditions,[6] so you can choose to solve the primal

$$\text{Linear:} \quad K(\mathbf{a}, \mathbf{b}) = \mathbf{a}^T \cdot \mathbf{b}$$

$$\text{Polynomial:} \quad K(\mathbf{a}, \mathbf{b}) = (\gamma \mathbf{a}^T \cdot \mathbf{b} + r)^d$$

$$\text{Gaussian RBF:} \quad K(\mathbf{a}, \mathbf{b}) = \exp\left(-\gamma \| \mathbf{a} - \mathbf{b} \|^2\right)$$

$$\text{Sigmoid:} \quad K(\mathbf{a}, \mathbf{b}) = \tanh\left(\gamma \mathbf{a}^T \cdot \mathbf{b} + r\right)$$

Mercer's Theorem

According to *Mercer's theorem*, if a function $K(\mathbf{a}, \mathbf{b})$ respects a few mathematical conditions called *Mercer's conditions* (K must be continuous, symmetric in its arguments so $K(\mathbf{a}, \mathbf{b}) = K(\mathbf{b}, \mathbf{a})$, etc.), then there exists a function that maps **a** and **b** into another space (possibly with much higher dimensions) such that $K(\mathbf{a}, \mathbf{b}) = (\mathbf{a})^T \cdot (\mathbf{b})$. So you can use K as a kernel since you know exists, even if you don't know what is. In the case of the Gaussian RBF kernel, it can be shown that actually maps each training instance to an infinite-dimensional space, so it's a good thing you don't need to actually perform the mapping!

Note that some frequently used kernels (such as the Sigmoid kernel) don't respect all of Mercer's conditions, yet they generally work well in practice.

There is still one loose end we must tie. Equation 5-7 shows how to go from the dual solution to the primal solution in the case of a linear SVM classifier, but if you apply the kernel trick you end up with equations that include $(x^{(i)})$. In fact, $\hat{\mathbf{w}}$ must have the same number of dimensions as $(x^{(i)})$, which may be huge or even infinite, so you can't compute it. But how can you make predictions without knowing $\hat{\mathbf{w}}$? Well, the good news

Before concluding this chapter, let's take a quick look at online SVM classifiers (recall that online learning means learning incrementally, typically as new instances arrive).

For linear SVM classifiers, one method is to use Gradient Descent (e.g., using `SGDClassifier`) to minimize the cost function in Equation 5-13, which is derived from the primal problem. Unfortunately it converges much more slowly than the methods based on QP.

Equation 5-13. Linear SVM classifier cost function

$$J(\mathbf{w}, b) = \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} \quad + \quad C\sum_{i=1}^{m} max(0,\ 1 - t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b))$$

The first sum in the cost function will push the model to have a small weight vector **w**, leading to a larger margin. The second sum computes the total of all margin violations. An instance's margin violation is equal to 0 if it is located off the street and on the correct side, or else it is proportional to the distance to the correct side of the street. Minimizing this term ensures that the model makes the margin violations as small and as few as possible

Chapter 6. Decision Trees

Like SVMs, *Decision Trees* are versatile Machine Learning algorithms that can perform both classification and regression tasks, and even multioutput tasks. They are very powerful algorithms, capable of fitting complex datasets. For example, in Chapter 2 you trained a `DecisionTreeRegressor` model on the California housing dataset, fitting it perfectly (actually overfitting it).

Decision Trees are also the fundamental components of Random Forests (see Chapter 7), which are among the most powerful Machine Learning algorithms available today.

In this chapter we will start by discussing how to train, visualize, and make predictions with Decision Trees. Then we will go through the CART training algorithm used by Scikit-Learn, and we will discuss how to regularize trees and use them for regression tasks. Finally, we will discuss some of the limitations of Decision Trees.

Training and Visualizing a Decision Tree

To understand Decision Trees, let's just build one and take a look at how it makes predictions. The following code trains a `DecisionTreeClassifier` on the iris dataset (see Chapter 4):

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier

iris = load_iris()
X = iris.data[:, 2:] # petal length and width
y = iris.target

tree_clf = DecisionTreeClassifier(max_depth=2)
tree_clf.fit(X, y)
```

You can visualize the trained Decision Tree by first using the `export_graphviz()` method to output a graph definition file called

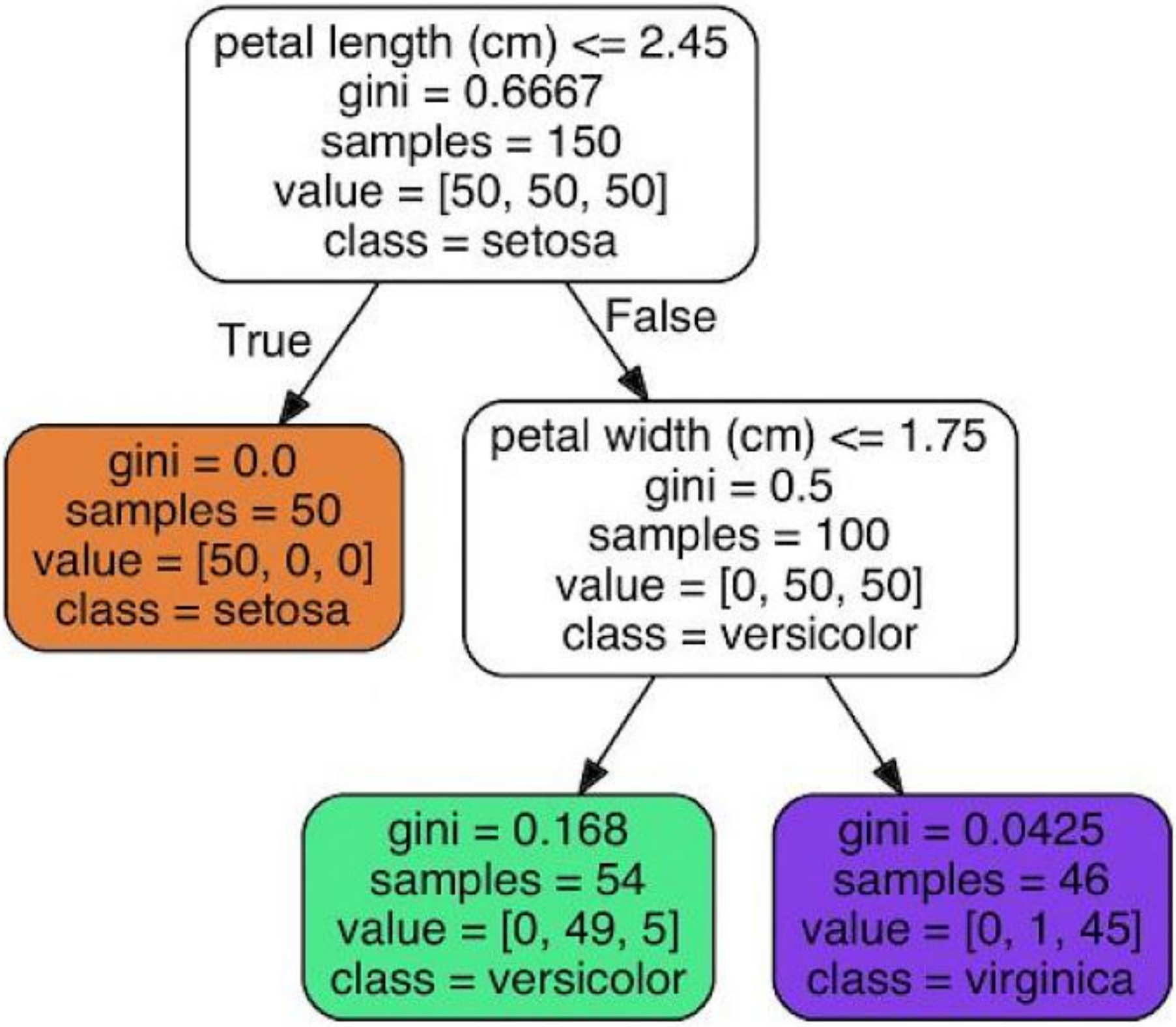


Figure 6-1. Iris Decision Tree

Making Predictions

Let's see how the tree represented in Figure 6-1 makes predictions. Suppose you find an iris flower and you want to classify it. You start at the *root node* (depth 0, at the top): this node asks whether the flower's petal length is smaller than 2.45 cm. If it is, then you move down to the root's left child node (depth 1, left). In this case, it is a *leaf node* (i.e., it does not have any children nodes), so it does not ask any questions: you can simply look at the predicted class for that node and the Decision Tree predicts that your flower is an Iris-Setosa (`class=setosa`).

Now suppose you find another flower, but this time the petal length is greater than 2.45 cm. You must move down to the root's right child node (depth 1, right), which is not a leaf node, so it asks another question: is the petal width smaller than 1.75 cm? If it is, then your flower is most likely an Iris-Versicolor (depth 2, left). If not, it is likely an Iris-Virginica (depth 2, right). It's really that simple.

Note

One of the many qualities of Decision Trees is that they require very little data preparation. In particular, they don't require feature scaling or centering at all.

A node's `samples` attribute counts how many training instances it applies to. For example, 100 training instances have a petal length greater than 2.45 cm (depth 1, right), among which 54 have a petal width smaller than 1.75 cm (depth 2, left). A node's `value` attribute tells you how many training instances of each class this node applies to: for example, the bottom-right node applies to 0 Iris-Setosa, 1 Iris-Versicolor, and 45 Iris-Virginica. Finally, a node's `gini` attribute measures its *impurity*: a node is "pure" (`gini=0`) if all training instances it applies to belong to the same class. For example, since the depth-1 left node applies only to Iris-Setosa training instances, it is pure and its `gini` score is 0. Equation 6-1 shows how the training algorithm computes the gini score G_i of the i^{th} node. For example, the depth-2 left node has a `gini` score equal to $1 - (0/54)^2 - (49/54)^2 - (5/54)^2 \approx 0.168$. Another *impurity measure* is discussed shortly.

Equation 6-1. Gini impurity

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2$$

- $p_{i,k}$ is the ratio of class k instances among the training instances in the i^{th} node.

Note

Scikit-Learn uses the CART algorithm, which produces only *binary trees*: nonleaf nodes always have two children (i.e., questions only have yes/no answers). However, other algorithms such as ID3 can produce Decision Trees with nodes that have more than two children.

Figure 6-2 shows this Decision Tree's decision boundaries. The thick vertical line represents the decision boundary of the root node (depth 0): petal length = 2.45 cm. Since the left area is pure (only Iris-Setosa), it cannot be split any further. However, the right area is impure, so the depth-1 right node splits it at petal width = 1.75 cm (represented by the dashed line). Since `max_depth` was set to 2, the Decision Tree stops right there. However, if you set `max_depth` to 3, then the two depth-2 nodes would each add another decision boundary (represented by the dotted lines).

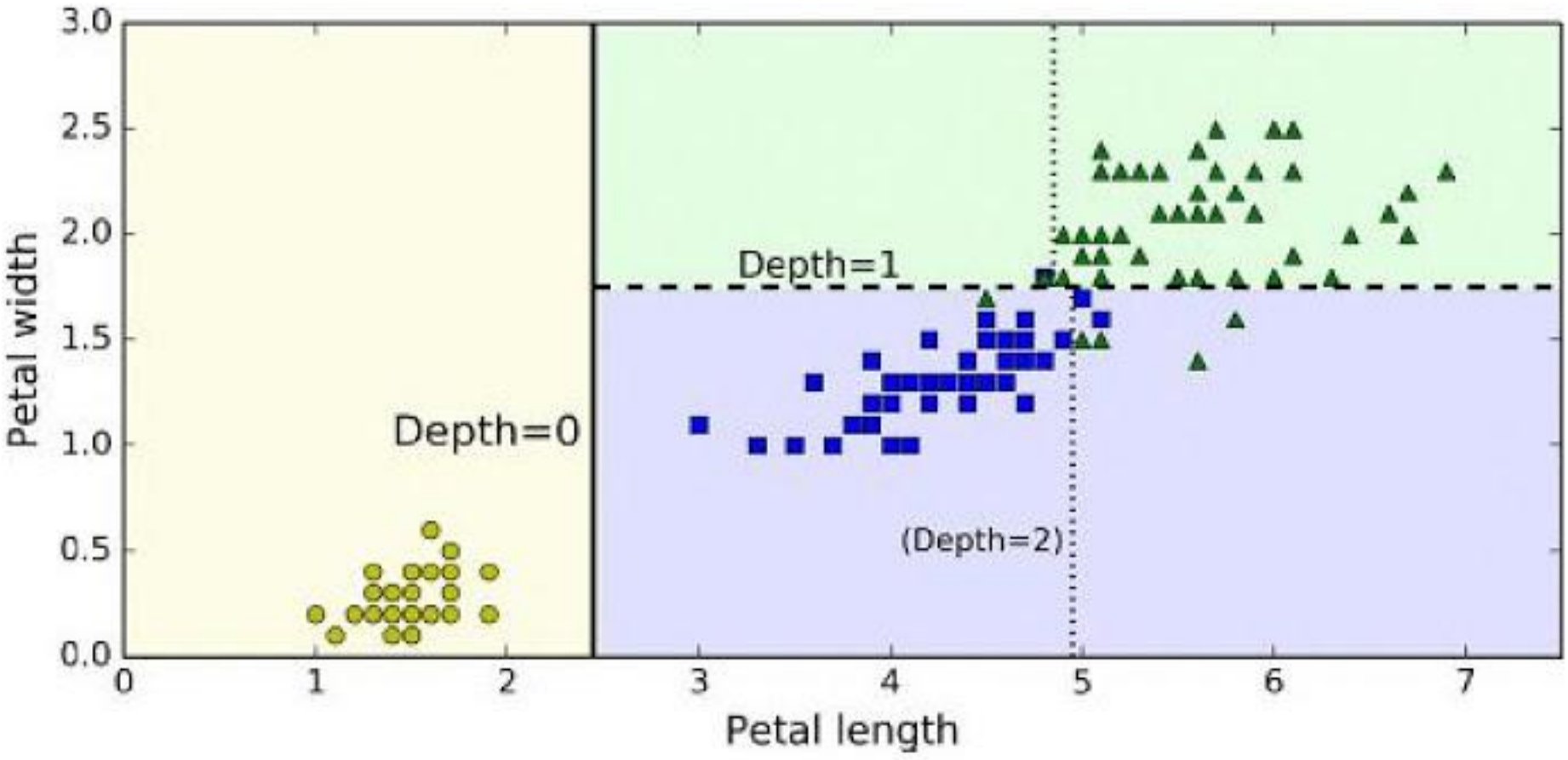


Figure 6-2. Decision Tree decision boundaries

Model Interpretation: White Box Versus Black Box

As you can see Decision Trees are fairly intuitive and their decisions are easy to interpret. Such models are often called *white box models*. In contrast, as we will see, Random Forests or neural networks are generally considered *black box models*. They make great predictions, and you can easily check the calculations that they performed to make these predictions; nevertheless, it is usually hard to explain in simple terms why the predictions were made. For example, if a neural network says that a particular person appears on a picture, it is hard to know what actually contributed to this prediction: did the model recognize that person's eyes? Her mouth? Her nose? Her shoes? Or even the couch that she was sitting on? Conversely, Decision Trees provide nice and simple classification rules that can even be applied manually if need be (e.g., for flower classification).

Estimating Class Probabilities

A Decision Tree can also estimate the probability that an instance belongs

to a particular class *k*: first it traverses the tree to find the leaf node for this instance, and then it returns the ratio of training instances of class *k* in this node. For example, suppose you have found a flower whose petals are 5 cm long and 1.5 cm wide. The corresponding leaf node is the depth-2 left node, so the Decision Tree should output the following probabilities: 0% for Iris-Setosa (0/54), 90.7% for Iris-Versicolor (49/54), and 9.3% for Iris-Virginica (5/54). And of course if you ask it to predict the class, it should output Iris-Versicolor (class 1) since it has the highest probability. Let's check this:

```
>>> tree_clf.predict_proba([[5, 1.5]])
array([[ 0. ,  0.90740741,  0.09259259]])
>>> tree_clf.predict([[5, 1.5]])
array([1])
```

Perfect! Notice that the estimated probabilities would be identical anywhere else in the bottom-right rectangle of Figure 6-2—for example, if the petals were 6 cm long and 1.5 cm wide (even though it seems obvious that it would most likely be an Iris-Virginica in this case).

The CART Training Algorithm

Scikit-Learn uses the *Classification And Regression Tree* (CART) algorithm to train Decision Trees (also called "growing" trees). The idea is really quite simple: the algorithm first splits the training set in two subsets using a single feature k and a threshold t_k (e.g., "petal length ≤ 2.45 cm"). How does it choose k and t_k? It searches for the pair (k, t_k) that produces the purest subsets (weighted by their size). The cost function that the algorithm tries to minimize is given by Equation 6-2.

Equation 6-2. CART cost function for classification

$$J(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}}$$

$$\text{where} \begin{cases} G_{\text{left/right}} \text{ measures the impurity of the left/right subset,} \\ m_{\text{left/right}} \text{ is the number of instances in the left/right subset.} \end{cases}$$

Once it has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets and so on, recursively. It stops recursing once it reaches the maximum depth (defined by the `max_depth` hyperparameter), or if it cannot find a split that will reduce impurity. A few other hyperparameters (described in a moment) control additional stopping conditions (`min_samples_split`, `min_samples_leaf`, `min_weight_fraction_leaf`, and `max_leaf_nodes`).

Warning

As you can see, the CART algorithm is a *greedy algorithm*: it greedily searches for an optimum split at the top level, then repeats the process at each level. It does not check whether or not the split will lead to the lowest possible impurity several levels down. A greedy algorithm often produces a reasonably good solution, but it is not guaranteed to be the optimal solution.

Unfortunately, finding the optimal tree is known to be an *NP-Complete* problem:[2] it requires $O(\exp(m))$ time, making the problem intractable even for fairly small training sets. This is why we must settle for a "reasonably good" solution.

Computational Complexity

Making predictions requires traversing the Decision Tree from the root to a leaf. Decision Trees are generally approximately balanced, so traversing the Decision Tree requires going through roughly $O(log_2(m))$ nodes.[3] Since each node only requires checking the value of one feature, the overall prediction complexity is just $O(log_2(m))$, independent of the number of features. So predictions are very fast, even when dealing with large training sets.

However, the training algorithm compares all features (or less if `max_features` is set) on all samples at each node. This results in a training complexity of $O(n \times m\ log(m))$. For small training sets (less than a few thousand instances), Scikit-Learn can speed up training by presorting the

data (set `presort=True`), but this slows down training considerably for larger training sets.

Gini Impurity or Entropy?

By default, the Gini impurity measure is used, but you can select the *entropy* impurity measure instead by setting the `criterion` hyperparameter to `"entropy"`. The concept of entropy originated in thermodynamics as a measure of molecular disorder: entropy approaches zero when molecules are still and well ordered. It later spread to a wide variety of domains, including Shannon's *information theory*, where it measures the average information content of a message:[4] entropy is zero when all messages are identical. In Machine Learning, it is frequently used as an impurity measure: a set's entropy is zero when it contains instances of only one class. Equation 6-3 shows the definition of the entropy of the i^{th} node. For example, the depth-2 left node in Figure 6-1 has an entropy equal to $-\frac{49}{54}\log\left(\frac{49}{54}\right) - \frac{5}{54}\log\left(\frac{5}{54}\right) \approx 0.31$.

Equation 6-3. Entropy

$$H_i = -\sum_{\substack{k=1 \\ p_{i,k}\neq 0}}^{n} p_{i,k}\log\left(p_{i,k}\right)$$

So should you use Gini impurity or entropy? The truth is, most of the time it does not make a big difference: they lead to similar trees. Gini impurity is slightly faster to compute, so it is a good default. However, when they differ, Gini impurity tends to isolate the most frequent class in its own branch of the tree, while entropy tends to produce slightly more balanced trees.[5]

Regularization Hyperparameters

Decision Trees make very few assumptions about the training data (as opposed to linear models, which obviously assume that the data is linear, for example). If left unconstrained, the tree structure will adapt itself to the training data, fitting it very closely, and most likely overfitting it. Such a model is often called a *nonparametric model*, not because it does not have any parameters (it often has a lot) but because the number of parameters is not determined prior to training, so the model structure is free to stick closely to the data. In contrast, a *parametric model* such as a linear model has a predetermined number of parameters, so its degree of freedom is limited, reducing the risk of overfitting (but increasing the risk of underfitting).

To avoid overfitting the training data, you need to restrict the Decision Tree's freedom during training. As you know by now, this is called regularization. The regularization hyperparameters depend on the algorithm used, but generally you can at least restrict the maximum depth of the Decision Tree. In Scikit-Learn, this is controlled by the `max_depth` hyperparameter (the default value is `None`, which means unlimited). Reducing `max_depth` will regularize the model and thus reduce the risk of overfitting.

The `DecisionTreeClassifier` class has a few other parameters that similarly restrict the shape of the Decision Tree: `min_samples_split` (the minimum number of samples a node must have before it can be split), `min_samples_leaf` (the minimum number of samples a leaf node must have), `min_weight_fraction_leaf` (same as `min_samples_leaf` but expressed as a fraction of the total number of weighted instances), `max_leaf_nodes` (maximum number of leaf nodes), and `max_features` (maximum number of features that are evaluated for splitting at each node). Increasing `min_*` hyperparameters or reducing `max_*` hyperparameters will regularize the model.

Note

Other algorithms work by first training the Decision Tree without restrictions, then *pruning* (deleting) unnecessary nodes. A node whose children are all leaf nodes is considered unnecessary if the purity improvement it provides is not *statistically significant*. Standard statistical tests, such as the ^{2} *test*, are used to estimate the probability that the improvement is purely the result of chance (which is called the *null hypothesis*). If this probability, called the *p-value*, is higher than a given threshold (typically 5%, controlled by a hyperparameter), then the node is considered unnecessary and its children are deleted. The pruning continues until all unnecessary nodes have been pruned.

Figure 6-3 shows two Decision Trees trained on the moons dataset (introduced in Chapter 5). On the left, the Decision Tree is trained with the default hyperparameters (i.e., no restrictions), and on the right the Decision Tree is trained with `min_samples_leaf=4`. It is quite obvious that the model on the left is overfitting, and the model on the right will probably generalize better.

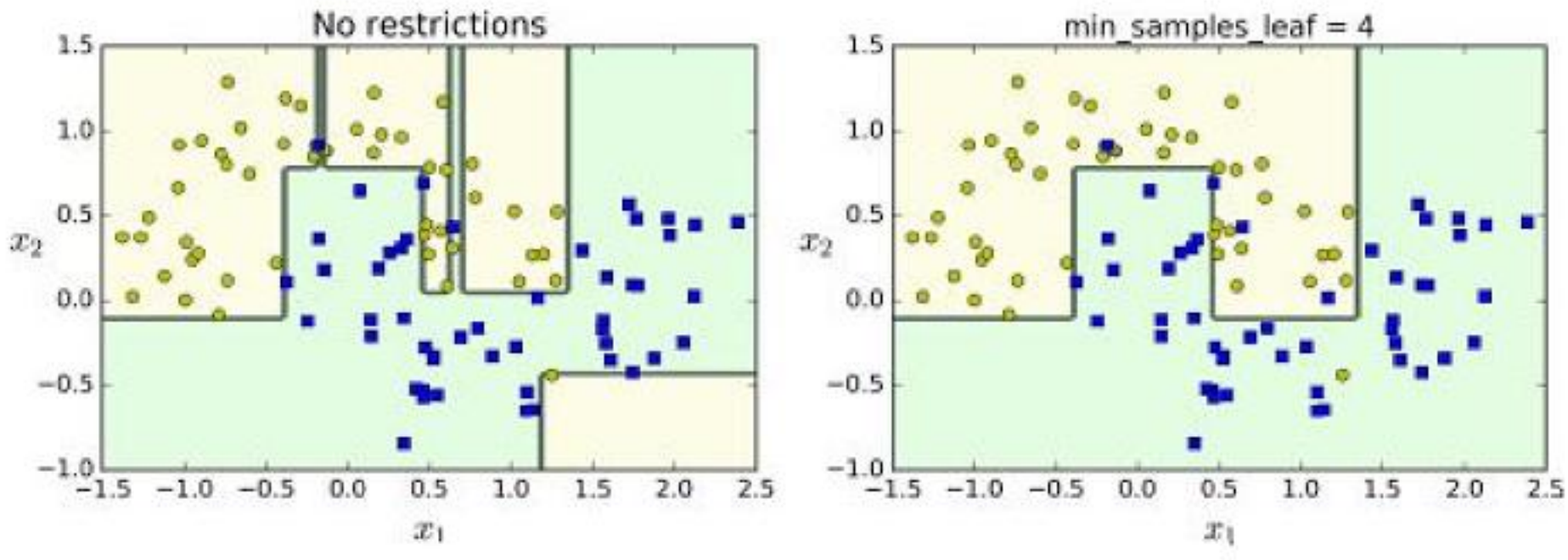


Figure 6-3. Regularization using min_samples_leaf

Regression

Decision Trees are also capable of performing regression tasks. Let's build a regression tree using Scikit-Learn's `DecisionTreeRegressor` class, training it on a noisy quadratic dataset with `max_depth=2`:

```
from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2)
tree_reg.fit(X, y)
```

The resulting tree is represented on Figure 6-4.

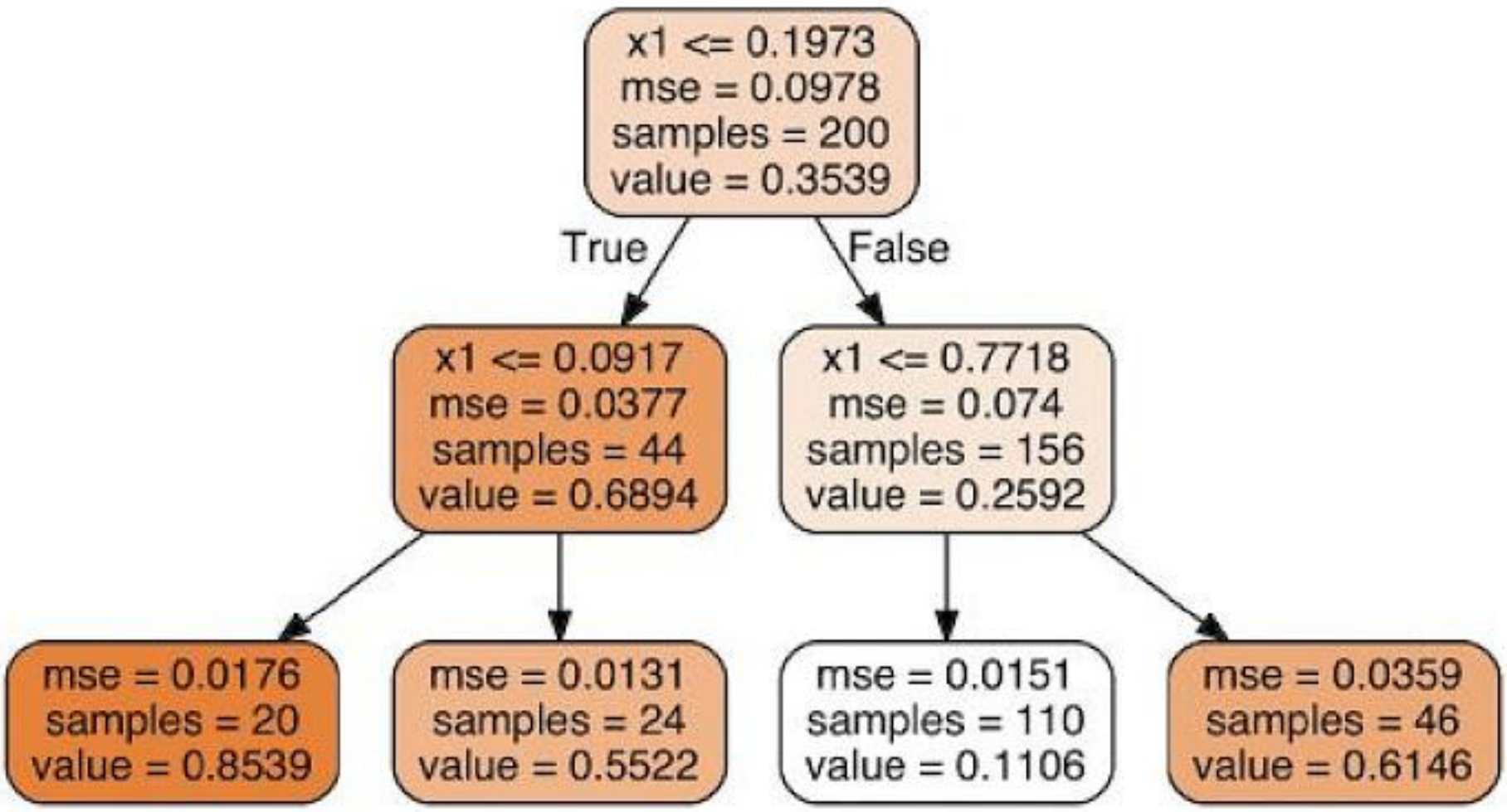


Figure 6-4. A Decision Tree for regression

This tree looks very similar to the classification tree you built earlier. The main difference is that instead of predicting a class in each node, it predicts a value. For example, suppose you want to make a prediction for a new instance with $x_1 = 0.6$. You traverse the tree starting at the root, and you eventually reach the leaf node that predicts `value=0.1106`. This prediction is simply the average target value of the 110 training instances associated to this leaf node. This prediction results in a Mean Squared Error (MSE) equal to 0.0151 over these 110 instances.

This model's predictions are represented on the left of Figure 6-5. If you set `max_depth=3`, you get the predictions represented on the right. Notice how the predicted value for each region is always the average target value of the

instances in that region. The algorithm splits each region in a way that makes most training instances as close as possible to that predicted value.

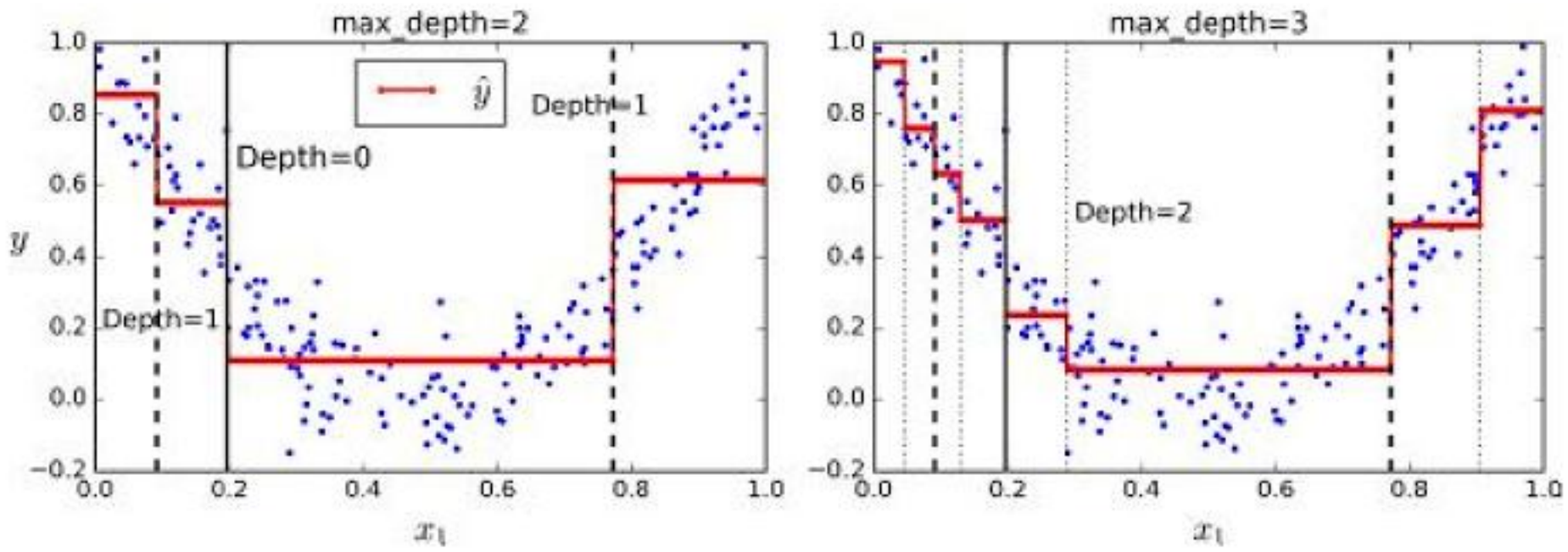


Figure 6-5. Predictions of two Decision Tree regression models

The CART algorithm works mostly the same way as earlier, except that instead of trying to split the training set in a way that minimizes impurity, it now tries to split the training set in a way that minimizes the MSE. Equation 6-4 shows the cost function that the algorithm tries to minimize.

Equation 6-4. CART cost function for regression

$$J(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}} \quad \text{where} \begin{cases} \text{MSE}_{\text{node}} = \sum_{i \in \text{node}} (\hat{y}_{\text{node}} - y^{(i)})^2 \\ \hat{y}_{\text{node}} = \frac{1}{m_{\text{node}}} \sum_{i \in \text{node}} y^{(i)} \end{cases}$$

Just like for classification tasks, Decision Trees are prone to overfitting when dealing with regression tasks. Without any regularization (i.e., using the default hyperparameters), you get the predictions on the left of Figure 6-6. It is obviously overfitting the training set very badly. Just setting `min_samples_leaf=10` results in a much more reasonable model, represented on the right of Figure 6-6.

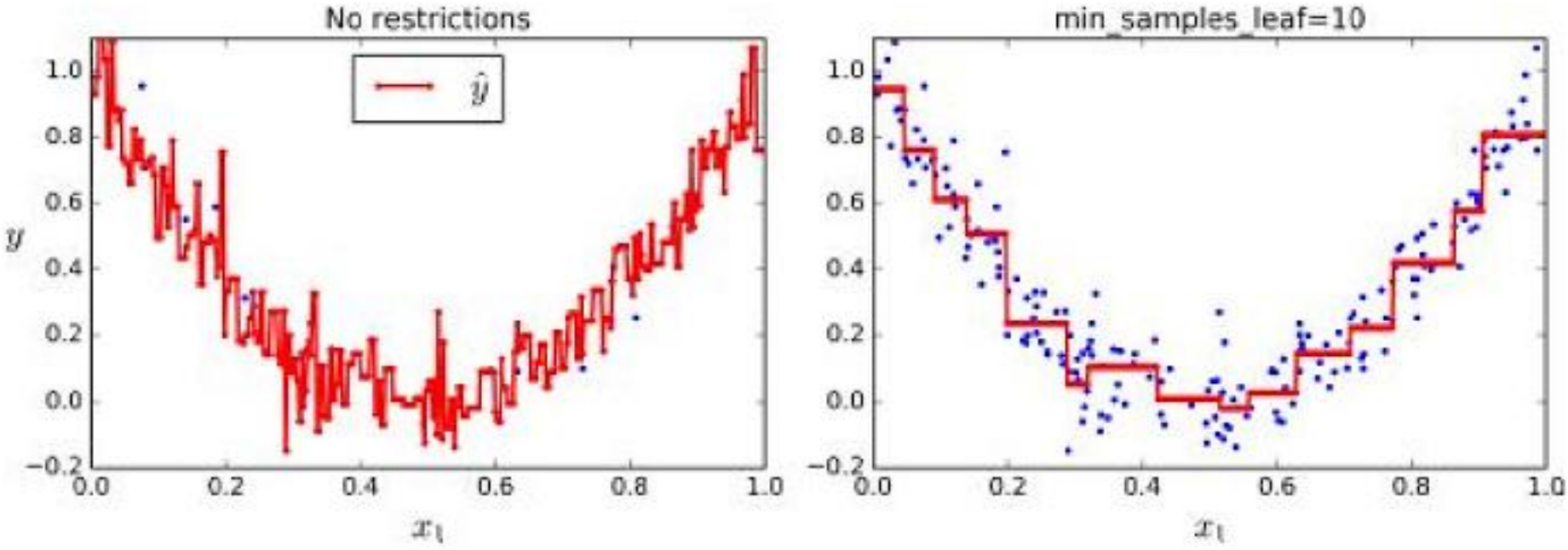


Figure 6-6. Regularizing a Decision Tree regressor

Instability

Hopefully by now you are convinced that Decision Trees have a lot going for them: they are simple to understand and interpret, easy to use, versatile, and powerful. However they do have a few limitations. First, as you may have noticed, Decision Trees love orthogonal decision boundaries (all splits are perpendicular to an axis), which makes them sensitive to training set rotation. For example, Figure 6-7 shows a simple linearly separable dataset: on the left, a Decision Tree can split it easily, while on the right, after the dataset is rotated by 45°, the decision boundary looks unnecessarily convoluted. Although both Decision Trees fit the training set perfectly, it is very likely that the model on the right will not generalize well. One way to limit this problem is to use PCA (see Chapter 8), which often results in a better orientation of the training data.

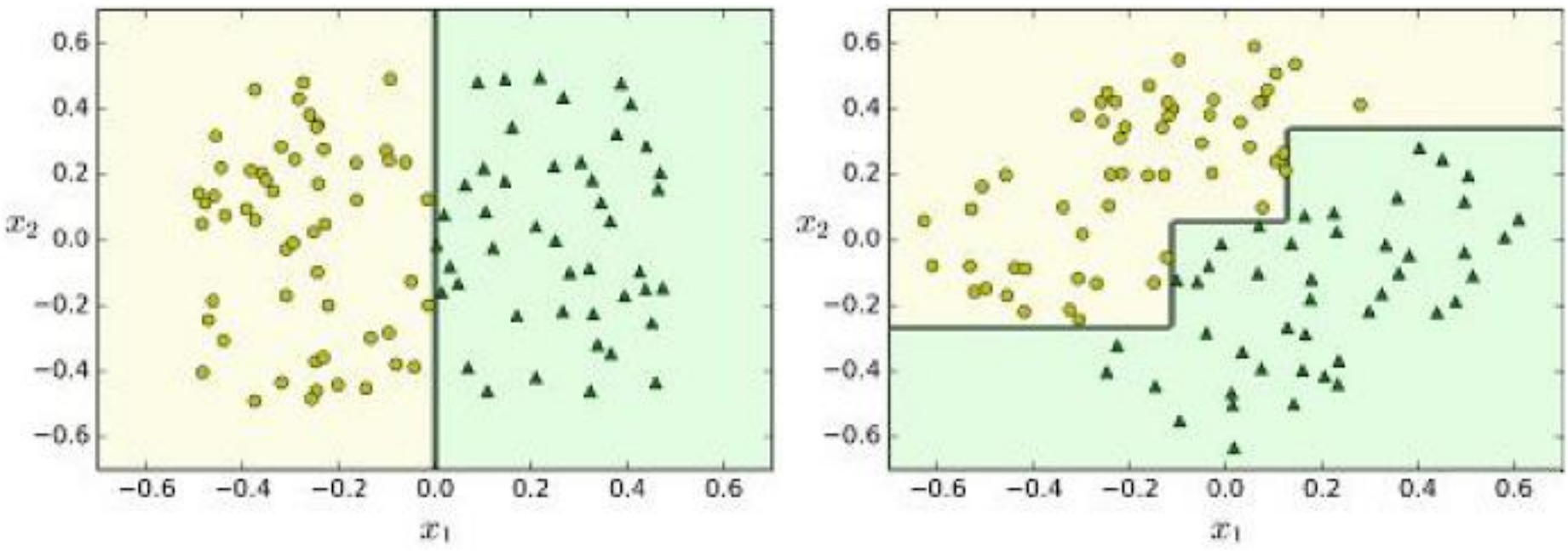


Figure 6-7. Sensitivity to training set rotation

More generally, the main issue with Decision Trees is that they are very sensitive to small variations in the training data. For example, if you just remove the widest Iris-Versicolor from the iris training set (the one with petals 4.8 cm long and 1.8 cm wide) and train a new Decision Tree, you may get the model represented in Figure 6-8. As you can see, it looks very different from the previous Decision Tree (Figure 6-2). Actually, since the training algorithm used by Scikit-Learn is stochastic[6] you may get very different models even on the same training data (unless you set the `random_state` hyperparameter).

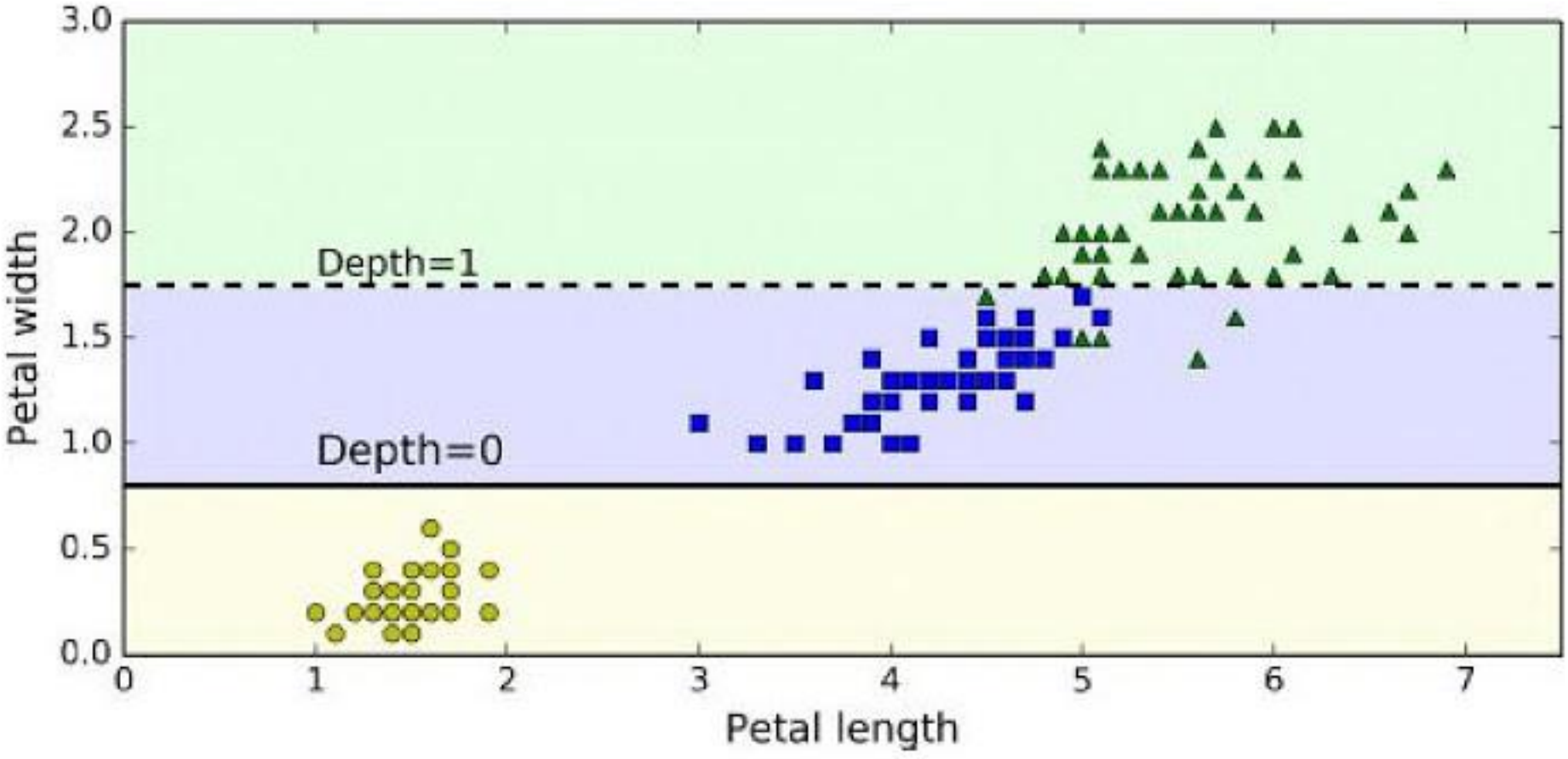


Figure 6-8. Sensitivity to training set details

Random Forests can limit this instability by averaging predictions over many trees, as we will see in the next chapter.

Exercises

1. What is the approximate depth of a Decision Tree trained (without restrictions) on a training set with 1 million instances?
2. Is a node's Gini impurity generally lower or greater than its parent's? Is it *generally* lower/greater, or *always* lower/greater?
3. If a Decision Tree is overfitting the training set, is it a good idea to try decreasing `max_depth`?
4. If a Decision Tree is underfitting the training set, is it a good idea to try scaling the input features?
5. If it takes one hour to train a Decision Tree on a training set containing 1 million instances, roughly how much time will it take to train another Decision Tree on a training set containing 10 million instances?
6. If your training set contains 100,000 instances, will setting

d. Evaluate these predictions on the test set: you should obtain a slightly higher accuracy than your first model (about 0.5 to 1.5% higher). Congratulations, you have trained a Random Forest classifier!

Solutions to these exercises are available in Appendix A.

[1] Graphviz is an open source graph visualization software package, available at *http://www.graphviz.org/*.

[2] P is the set of problems that can be solved in polynomial time. NP is the set of problems whose solutions can be verified in polynomial time. An NP-Hard problem is a problem to which any NP problem can be reduced in polynomial time. An NP-Complete problem is both NP and NP-Hard. A major open mathematical question is whether or not P = NP. If P ≠ NP (which seems likely), then no polynomial algorithm will ever be found for any NP-Complete problem (except perhaps on a quantum computer).

[3] log_2 is the binary logarithm. It is equal to $log_2(m) = log(m) / log(2)$.

[4] A reduction of entropy is often called an *information gain*.

[5] See Sebastian Raschka's interesting analysis for more details.

[6] It randomly selects the set of features to evaluate at each node.

Chapter 7. Ensemble Learning and Random Forests

Suppose you ask a complex question to thousands of random people, then aggregate their answers. In many cases you will find that this aggregated answer is better than an expert's answer. This is called the *wisdom of the crowd*. Similarly, if you aggregate the predictions of a group of predictors (such as classifiers or regressors), you will often get better predictions than with the best individual predictor. A group of predictors is called an *ensemble*; thus, this technique is called *Ensemble Learning*, and an Ensemble Learning algorithm is called an *Ensemble method*.

For example, you can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you just obtain the predictions of all individual trees, then predict the class that gets the most votes (see the last exercise in Chapter 6). Such an ensemble of Decision Trees is called a *Random Forest*, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today.

Moreover, as we discussed in Chapter 2, you will often use Ensemble methods near the end of a project, once you have already built a few good predictors, to combine them into an even better predictor. In fact, the winning solutions in Machine Learning competitions often involve several Ensemble methods (most famously in the Netflix Prize competition).

In this chapter we will discuss the most popular Ensemble methods, including *bagging*, *boosting*, *stacking*, and a few others. We will also explore Random Forests.

Voting Classifiers

Suppose you have trained a few classifiers, each one achieving about 80% accuracy. You may have a Logistic Regression classifier, an SVM classifier, a Random Forest classifier, a K-Nearest Neighbors classifier, and perhaps a few more (see Figure 7-1).

Similarly, suppose you build an ensemble containing 1,000 classifiers that are individually correct only 51% of the time (barely better than random guessing). If you predict the majority voted class, you can hope for up to 75% accuracy! However, this is only true if all classifiers are perfectly independent, making uncorrelated errors, which is clearly not the case since they are trained on the same data. They are likely to make the same types of errors, so there will be many majority votes for the wrong class, reducing the ensemble's accuracy.

Tip

Ensemble methods work best when the predictors are as independent from one another as possible. One way to get diverse classifiers is to train them using very different algorithms. This increases the chance that they will make very different types of errors, improving the ensemble's accuracy.

The following code creates and trains a voting classifier in Scikit-Learn, composed of three diverse classifiers (the training set is the moons dataset, introduced in Chapter 5):

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression()
rnd_clf = RandomForestClassifier()
svm_clf = SVC()

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc',
svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)
```

Let's look at each classifier's accuracy on the test set:

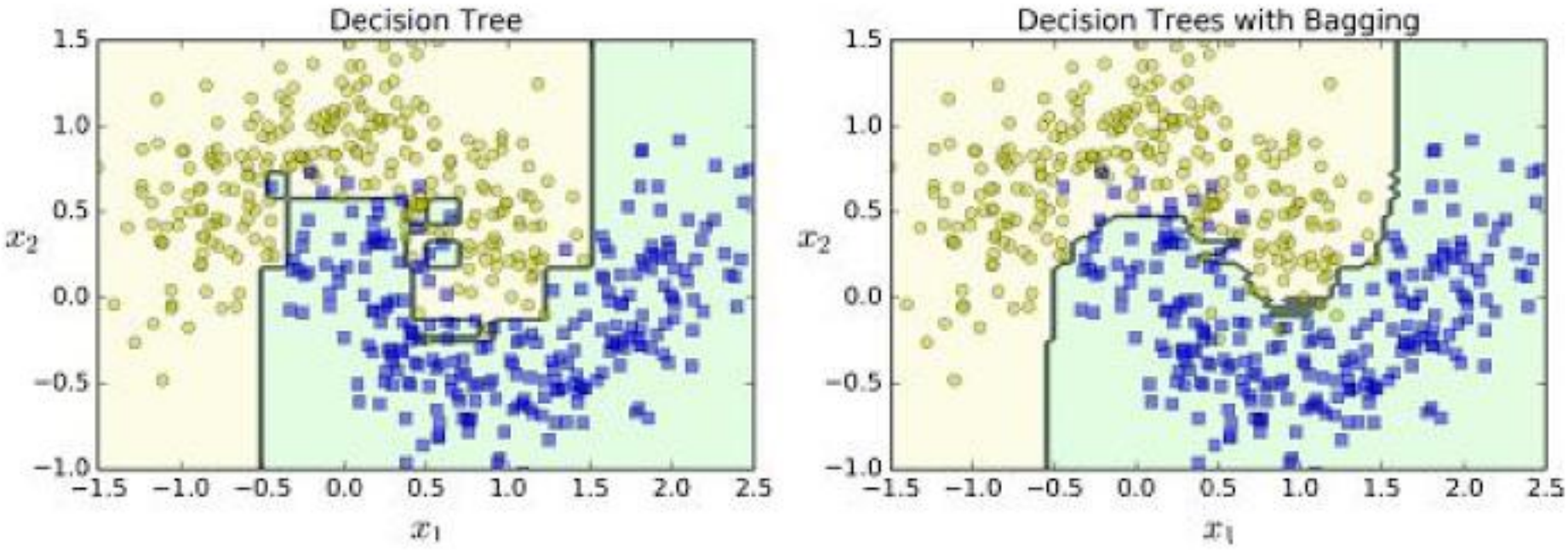


Figure 7-5. A single Decision Tree versus a bagging ensemble of 500 trees

Bootstrapping introduces a bit more diversity in the subsets that each predictor is trained on, so bagging ends up with a slightly higher bias than pasting, but this also means that predictors end up being less correlated so the ensemble's variance is reduced. Overall, bagging often results in better models, which explains why it is generally preferred. However, if you have spare time and CPU power you can use cross-validation to evaluate both bagging and pasting and select the one that works best.

Out-of-Bag Evaluation

With bagging, some instances may be sampled several times for any given predictor, while others may not be sampled at all. By default a `BaggingClassifier` samples *m* training instances with replacement (`bootstrap=True`), where *m* is the size of the training set. This means that only about 63% of the training instances are sampled on average for each predictor.[6] The remaining 37% of the training instances that are not sampled are called *out-of-bag* (oob) instances. Note that they are not the same 37% for all predictors.

Since a predictor never sees the oob instances during training, it can be evaluated on these instances, without the need for a separate validation set or cross-validation. You can evaluate the ensemble itself by averaging out the oob evaluations of each predictor.

In Scikit-Learn, you can set `oob_score=True` when creating a `BaggingClassifier` to request an automatic oob evaluation after training. The following code demonstrates this. The resulting evaluation score is available through the `oob_score_` variable:

```
>>> bag_clf = BaggingClassifier(
...     DecisionTreeClassifier(), n_estimators=500,
...     bootstrap=True, n_jobs=-1, oob_score=True)
...
>>> bag_clf.fit(X_train, y_train)
>>> bag_clf.oob_score_
0.90133333333333332
```

According to this oob evaluation, this `BaggingClassifier` is likely to achieve about 90.1% accuracy on the test set. Let's verify this:

```
>>> from sklearn.metrics import accuracy_score
>>> y_pred = bag_clf.predict(X_test)
>>> accuracy_score(y_test, y_pred)
0.91200000000000003
```

We get 91.2% accuracy on the test set—close enough!

The oob decision function for each training instance is also available through the `oob_decision_function_` variable. In this case (since the base estimator has a `predict_proba()` method) the decision function returns the class probabilities for each training instance. For example, the oob evaluation estimates that the first training instance has a 68.25% probability of belonging to the positive class (and 31.75% of belonging to the negative class):

```
>>> bag_clf.oob_decision_function_
array([[ 0.31746032,  0.68253968],
       [ 0.34117647,  0.65882353],
       [ 1.        ,  0.        ],
```

```
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> rnd_clf = RandomForestClassifier(n_estimators=500, n_jobs=-1)
>>> rnd_clf.fit(iris["data"], iris["target"])
>>> for name, score in zip(iris["feature_names"],
rnd_clf.feature_importances_):
...     print(name, score)
...
sepal length (cm) 0.112492250999
sepal width (cm) 0.0231192882825
petal length (cm) 0.441030464364
petal width (cm) 0.423357996355
```

Similarly, if you train a Random Forest classifier on the MNIST dataset (introduced in Chapter 3) and plot each pixel's importance, you get the image represented in Figure 7-6.

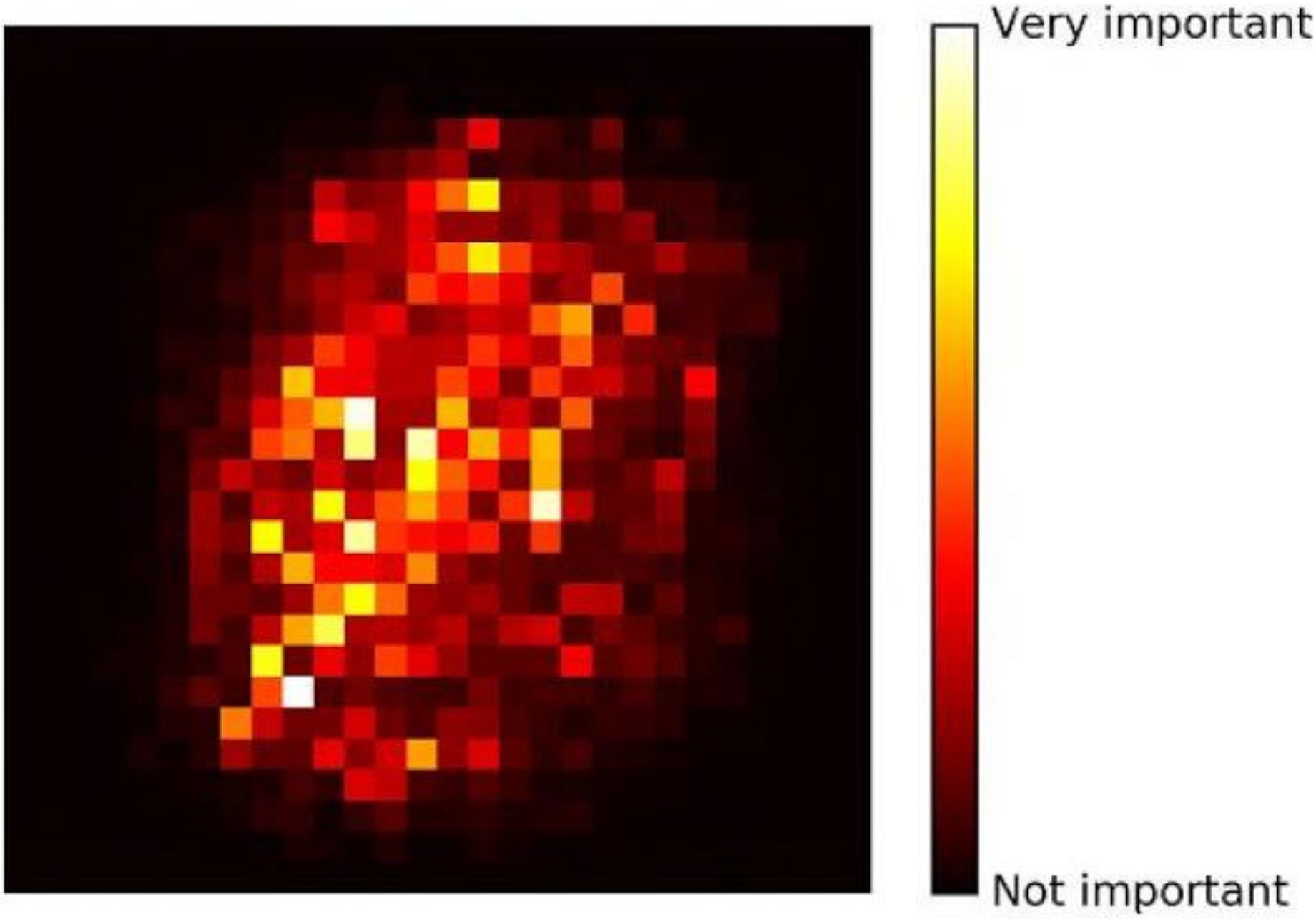


Figure 7-6. MNIST pixel importance (according to a Random Forest classifier)

class is the one that receives the majority of weighted votes (see Equation 7-4).

Equation 7-4. AdaBoost predictions

$$\hat{y}(\mathbf{x}) = \underset{k}{\operatorname{argmax}} \sum_{\substack{j=1 \\ \hat{y}_j(\mathbf{x})=k}}^{N} \alpha_j \quad \text{where } N \text{ is the number of predictors.}$$

Scikit-Learn actually uses a multiclass version of AdaBoost called *SAMME*[16] (which stands for *Stagewise Additive Modeling using a Multiclass Exponential loss function*). When there are just two classes, SAMME is equivalent to AdaBoost. Moreover, if the predictors can estimate class probabilities (i.e., if they have a `predict_proba()` method), Scikit-Learn can use a variant of SAMME called *SAMME.R* (the *R* stands for "Real"), which relies on class probabilities rather than predictions and generally performs better.

The following code trains an AdaBoost classifier based on 200 *Decision Stumps* using Scikit-Learn's `AdaBoostClassifier` class (as you might expect, there is also an `AdaBoostRegressor` class). A Decision Stump is a Decision Tree with `max_depth=1`—in other words, a tree composed of a single decision node plus two leaf nodes. This is the default base estimator for the `AdaBoostClassifier` class:

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5)
ada_clf.fit(X_train, y_train)
```

Tip

If your AdaBoost ensemble is overfitting the training set, you can try reducing the number of estimators or more strongly regularizing the base

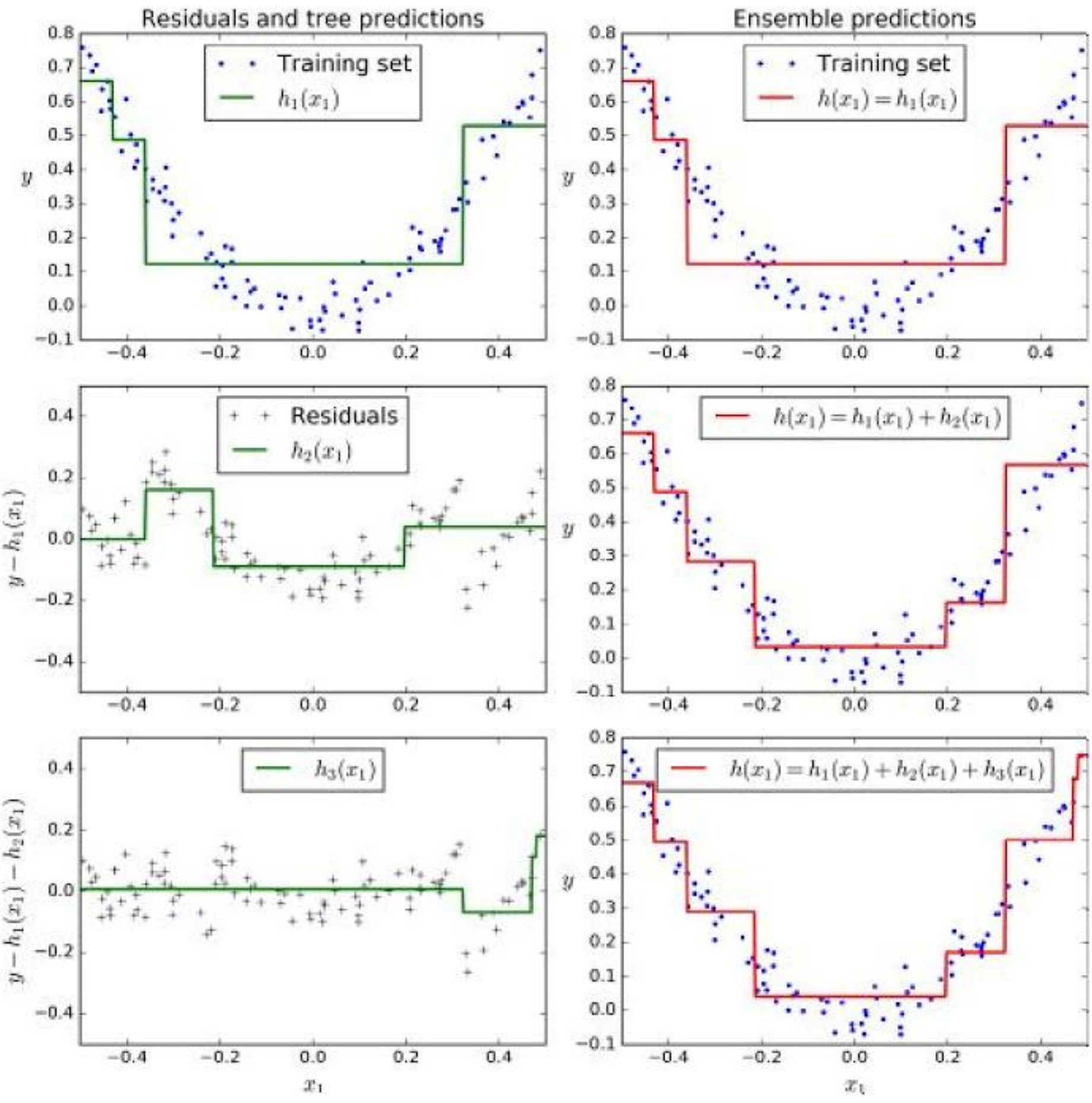


Figure 7-9. Gradient Boosting

The `learning_rate` hyperparameter scales the contribution of each tree. If you set it to a low value, such as `0.1`, you will need more trees in the ensemble to fit the training set, but the predictions will usually generalize better. This is a regularization technique called *shrinkage*. Figure 7-10 shows two GBRT ensembles trained with a low learning rate: the one on the left does not have enough trees to fit the training set, while the one on

Chapter 8. Dimensionality Reduction

Many Machine Learning problems involve thousands or even millions of features for each training instance. Not only does this make training extremely slow, it can also make it much harder to find a good solution, as we will see. This problem is often referred to as the *curse of dimensionality*.

Fortunately, in real-world problems, it is often possible to reduce the number of features considerably, turning an intractable problem into a tractable one. For example, consider the MNIST images (introduced in Chapter 3): the pixels on the image borders are almost always white, so you could completely drop these pixels from the training set without losing much information. Figure 7-6 confirms that these pixels are utterly unimportant for the classification task. Moreover, two neighboring pixels are often highly correlated: if you merge them into a single pixel (e.g., by taking the mean of the two pixel intensities), you will not lose much information.

Warning

Reducing dimensionality does lose some information (just like compressing an image to JPEG can degrade its quality), so even though it will speed up training, it may also make your system perform slightly worse. It also makes your pipelines a bit more complex and thus harder to maintain. So you should first try to train your system with the original data before considering using dimensionality reduction if training is too slow. In some cases, however, reducing the dimensionality of the training data may filter out some noise and unnecessary details and thus result in higher performance (but in general it won’t; it will just speed up training).

Apart from speeding up training, dimensionality reduction is also extremely useful for data visualization (or *DataViz*). Reducing the number of dimensions down to two (or three) makes it possible to plot a high-dimensional training set on a graph and often gain some important insights by visually detecting patterns, such as clusters.

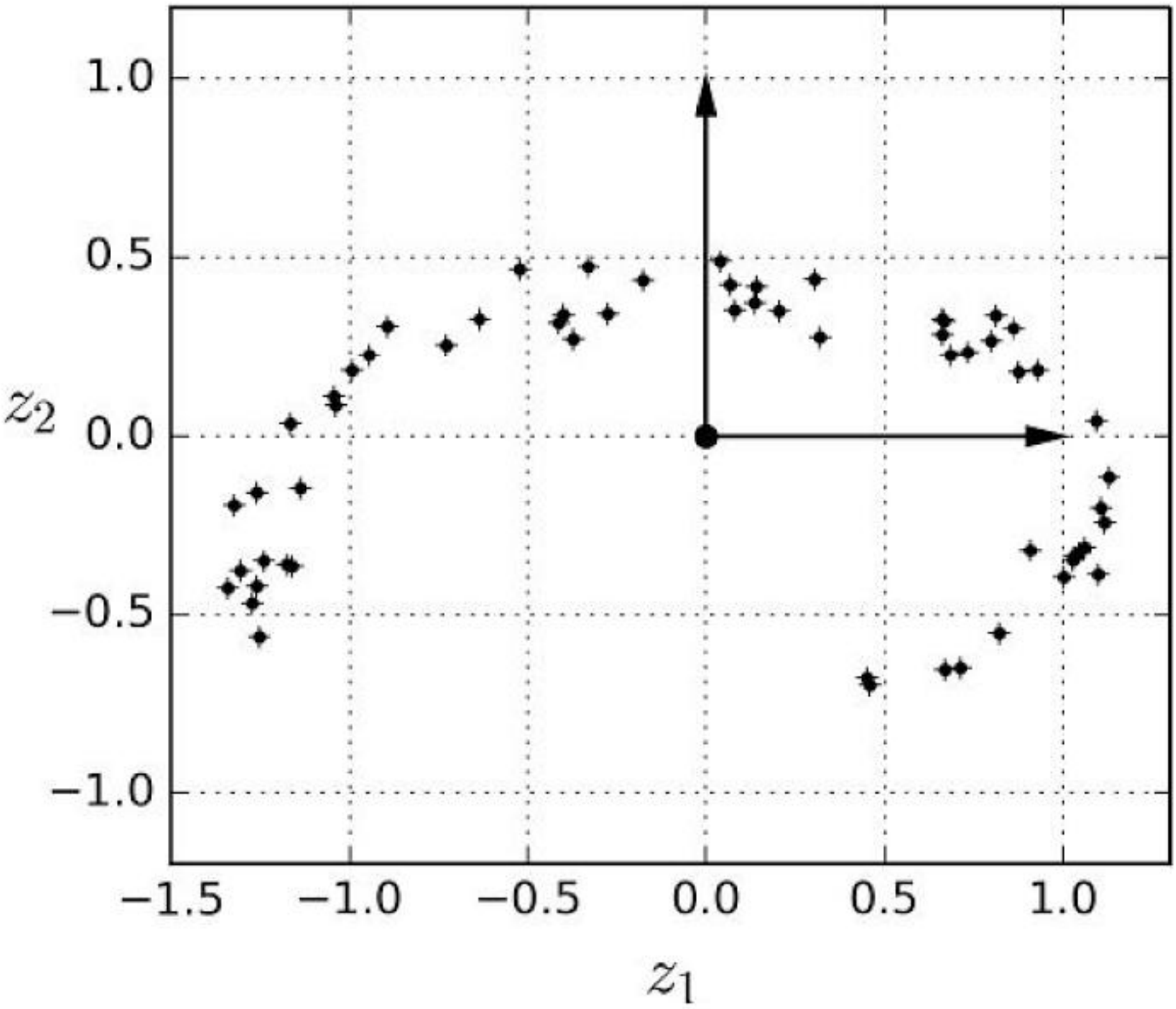


Figure 8-3. The new 2D dataset after projection

However, projection is not always the best approach to dimensionality reduction. In many cases the subspace may twist and turn, such as in the famous *Swiss roll* toy dataset represented in Figure 8-4.

freedom you would have if you were allowed to generate any image you wanted. These constraints tend to squeeze the dataset into a lower-dimensional manifold.

The manifold assumption is often accompanied by another implicit assumption: that the task at hand (e.g., classification or regression) will be simpler if expressed in the lower-dimensional space of the manifold. For example, in the top row of Figure 8-6 the Swiss roll is split into two classes: in the 3D space (on the left), the decision boundary would be fairly complex, but in the 2D unrolled manifold space (on the right), the decision boundary is a simple straight line.

However, this assumption does not always hold. For example, in the bottom row of Figure 8-6, the decision boundary is located at $x_1 = 5$. This decision boundary looks very simple in the original 3D space (a vertical plane), but it looks more complex in the unrolled manifold (a collection of four independent line segments).

In short, if you reduce the dimensionality of your training set before training a model, it will definitely speed up training, but it may not always lead to a better or simpler solution; it all depends on the dataset.

Hopefully you now have a good sense of what the curse of dimensionality is and how dimensionality reduction algorithms can fight it, especially when the manifold assumption holds. The rest of this chapter will go through some of the most popular algorithms.

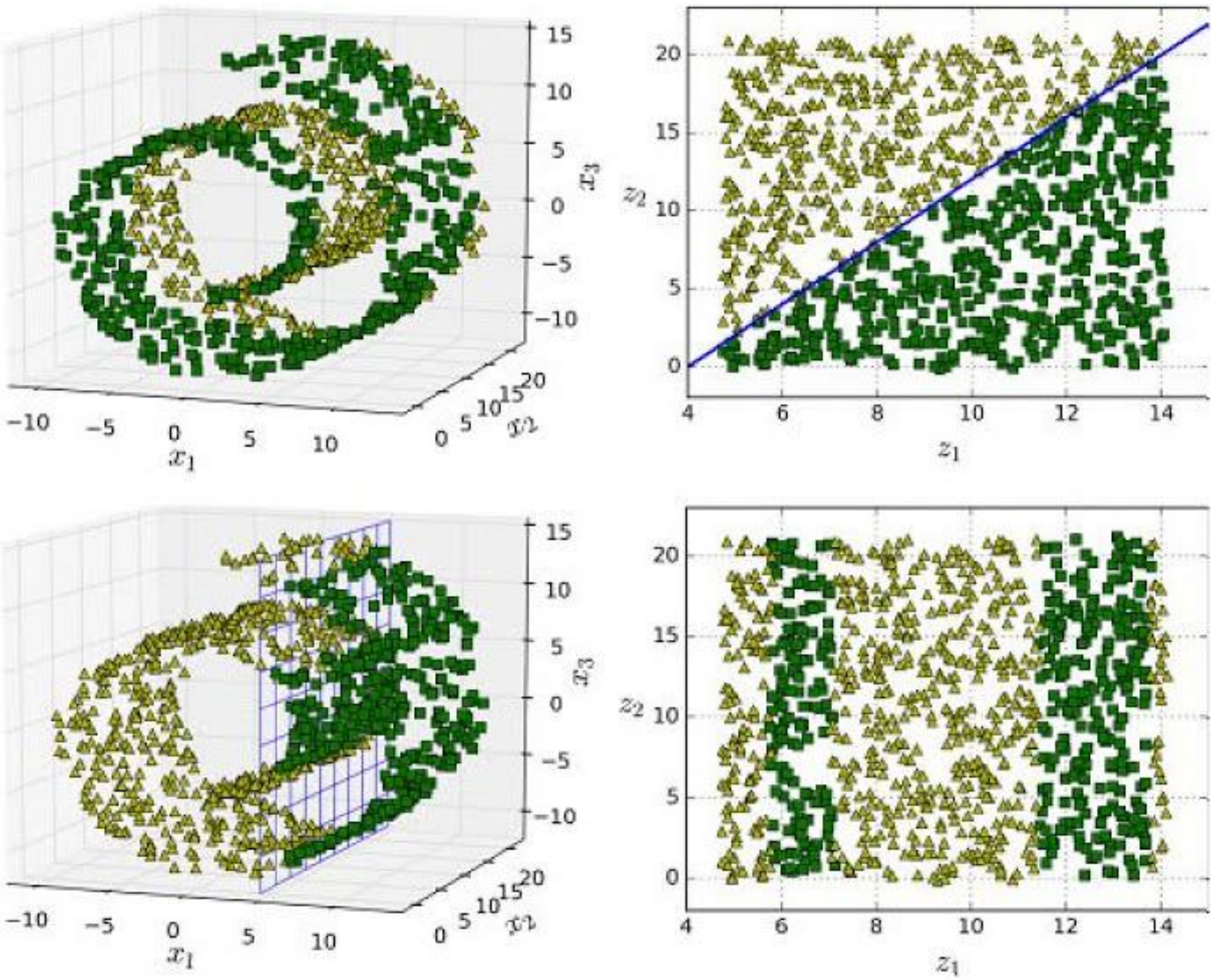


Figure 8-6. The decision boundary may not always be simpler with lower dimensions

PCA

Principal Component Analysis (PCA) is by far the most popular dimensionality reduction algorithm. First it identifies the hyperplane that lies closest to the data, and then it projects the data onto it.

Preserving the Variance

Before you can project the training set onto a lower-dimensional hyperplane, you first need to choose the right hyperplane. For example, a simple 2D dataset is represented on the left of Figure 8-7, along with three different axes (i.e., one-dimensional hyperplanes). On the right is the result of the projection of the dataset onto each of these axes. As you can see, the