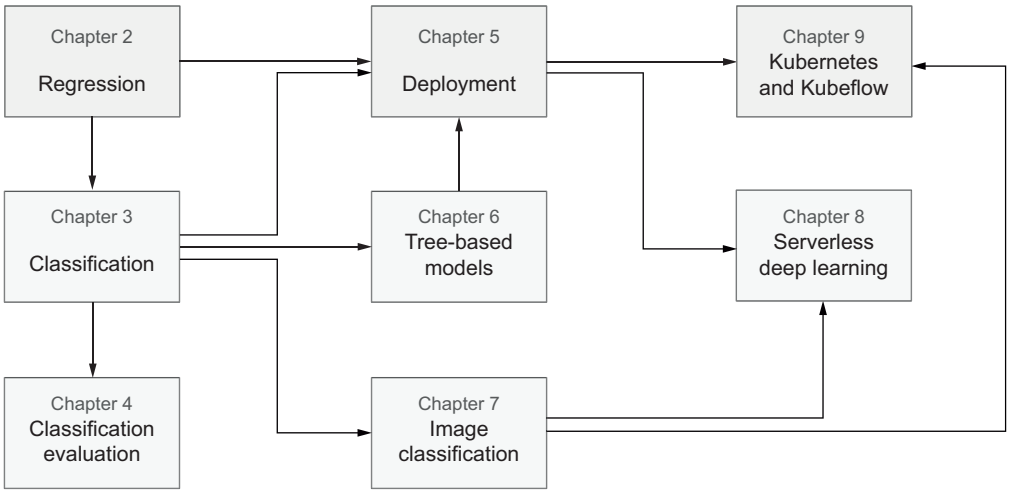
Build a portfolio of real-life projects

Machine Learning Bookcamp

Alexey Grigorev

Foreword by Luca Massaron

MANNING



Machine Learning Bookcamp

BUILD A PORTFOLIO OF REAL-LIFE PROJECTS

ALEXEY GRIGOREV

MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com


Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964

Development editor: Susan Ethridge
Technical development editor: Michael Lund
Review editor: Adriana Sabo
Production editor: Deirdre S. Hiam
Copy editor: Pamela Hunt
Proofreader: Melody Dolab
Technical proofreader: Al Krinker
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781617296819
Printed in the United States of America

brief contents

contents

foreword

I've known Alexey for more than six years. We almost worked together at the same data science team in a tech company in Berlin: Alexey started a few months after I left. Despite that, we still managed to get to know each other through Kaggle, the data science competition platform, and a common friend. We participated on the same team in a Kaggle competition on natural language processing, an interesting project that required carefully using pretrained word embeddings and cleverly mixing them. At the same time, Alexey was writing a book, and he asked me to be a technical reviewer. The book was about Java and data science, and, while reading it, I was particularly impressed by how carefully Alexey planned and orchestrated interesting examples. This led soon to a new collaboration: we coauthored a project-based book about TensorFlow, working on different projects from reinforcement learning to recommender systems that aimed to be an inspiration and example for the readers.

When working with Alexey, I noticed that he prefers to learn things by doing and by coding, like many others who transitioned to data science from software engineering.

Therefore, I wasn't very surprised when I heard that he had started another project-based book. Invited to provide feedback on Alexey's work, I read the book from its early stages and found the reading fascinating. This book is a practical introduction to machine learning with a focus on hands-on experience. It's written for people with the same background that Alexey has — for developers interested in data science and needing to quickly build up useful and reusable experience with data and data problems.

As an author of more than a dozen books on data science and AI, I know there are already a lot of books and courses on this topic. However, this book is quite different.

In *Machine Learning Bookcamp*, you won't find the same déjà vu data problems that other books offer. It doesn't have the same pedantic, repetitive flow of topics, like a route already traced on maps that always leads to places that you already know and have seen.

Everything in the book revolves around practical and nearly real-world examples. You will learn how to predict the price of a car, determine whether or not a customer is going to churn, and assess the risk of not repaying a loan. After that, you will classify clothing photos into T-shirts, dresses, pants, and other categories. This project is especially curious and interesting because Alexey personally curated this dataset, and you can enrich it with the clothes from your own wardrobe.

By reading this book, of course, you are expected to apply machine learning to solve common problems, and you will use the simplest and most efficient solutions to achieve the best results. The first chapters begin by examining basic algorithms such as linear regression and logistic regression. The reader then gradually moves to gradient boosting and neural networks. Nevertheless, the strong point of the book is that, while teaching machine learning through practice, it also prepares you for the real world. You will deal with unbalanced classes and long-tail distributions, and discover how to handle dirty data. You will evaluate your models and deploy them with AWS Lambda and Kubernetes. And these are just a few of the new techniques you learn by working through the pages.

Thinking with the mind-set of an engineer, you can say that this book is arranged so that you'll get the core 20% knowledge that covers 80% of being a great data scientist. More importantly, I'll add that you'll be also reading and practicing under Alexey's guidance, which is distilled by his work and Kaggle experience. Given such premises, I wish you a great journey through the pages and the projects of this book. I am sure that it will help you find the best way to approach data science and its problems, tools, and solutions.

— Luca Massaron

preface

I started my career working as a Java developer. Around 2012–2013, I became interested in data science and machine learning. First, I watched online courses, and then I enrolled in a master's program and spent two years studying different aspects of business intelligence and data science. Eventually, I graduated in 2015, and started working as a data scientist.

At work, my colleague showed me Kaggle — a platform for data science competitions. I thought, "With all the skills I got from courses and my master's degree, I'll be able to win any competition easily." But when I tried competing, I failed miserably. All the theoretical knowledge I had was useless on Kaggle. My models were awful, and I ended up on the bottom of the leaderboard.

I spent the next nine months taking part in data science competitions. I didn't do exceptionally well, but this was when I actually learned machine learning.

I realized that for me, the best way to learn is to do projects. When I focus on the problem, when I implement something, when I experiment, then I really learn. But if I focus on courses and theory, I invest too much time in learning things that aren't important and useful in practice.

And I'm not alone. When telling this story, I've heard "Me, too!" many times. That's why the focus of *Machine Learning Bookcamp* is on learning by doing projects. I believe that software engineers — people with the same background as me — learn best by doing.

We start this book with a car-price prediction project and learn linear regression. Then, we determine if customers want to stop using the services of our company. For

this, we learn logistic regression. To learn decision trees, we score the clients of a bank to determine if they can pay back a loan. Finally, we use deep learning to classify pictures of clothes into different classes like T-shirts, pants, shoes, outerwear, and so on.

Each project in the book starts with the problem description. We then solve this problem using different tools and frameworks. By focusing on the problem, we cover only the parts that are important for solving this problem. There is theory as well, but I keep it to a minimum and focus on the practical part.

Sometimes, however, I had to include formulas in some chapters. It's not possible to avoid formulas in a book about machine learning. I know that formulas are terrifying for some of us. I've been there, too. That's why I explain all the formulas with code as well. When you see a formula, don't let it scare you. Try to understand the code first and then get back to the formula to see how the code translates to the formula. Then the formula won't be intimidating anymore!

You won't find all possible topics in this book. I focused on the most fundamental things — things you will use with 100% certainty when you start working with machine learning. There are other important topics that I didn't cover: time series analysis, clustering, natural language processing. After reading this book, you will have enough background knowledge to learn these topics yourself.

Three chapters in this book focus on model deployment. These are very important chapters — maybe the most important ones. Being able to deploy a model makes the difference between a successful project and a failed one. Even the best model is useless if others can't use it. That's why it's worth investing your time in learning how to make it accessible for others. And that's the reason I cover it quite early in the book, right after we learn about logistic regression.

The last chapter is about deploying models with Kubernetes. It's not a simple chapter, but nowadays Kubernetes is the most commonly used container management system. It's likely that you'll need to work with it, and that's why it's included in the book.

Finally, each chapter of the book includes exercises. It might be tempting to skip them, but I don't recommend doing so. If you only follow the book, you will learn many new things. But if you don't apply this knowledge in practice, you will forget most of it quite soon. The exercises help you apply these new skills in practice — and you'll remember what you learned much better.

Enjoy your journey through the book, and feel free to get in touch with me at any time!

— Alexey Grigorev

acknowledgments

Working on this book took a lot of my free time. I spent countless evenings and sleepless nights working on it. That's why, first and foremost, I would like to thank my wife for her patience and support.

Next, I would like to thank my editor, Susan Ethridge, for her patience as well. The book's first early access version was released in January 2020. Shortly after that, the world around us went crazy, and everyone was locked down at home. Working on the book was extremely challenging for me. I don't know how many deadlines I missed (a lot!), but Susan wasn't pushing me and let me work at my own pace.

The first person who had to read all the chapters (after Susan) was Michael Lund. I would like to thank Michael for the invaluable feedback he provided and for all the comments he left on my drafts. One of the reviewers wrote that "the attention to detail across the book is marvelous," and the main reason for that is Michael's input.

Finding the motivation to work on the book during the lockdown was difficult. At times, I didn't feel any energy at all. But the feedback from the reviewers and the MEAP readers was very encouraging. It helped me to finish the book despite all the difficulties. So, I would like to thank you all for reviewing the drafts, for giving me the feedback and — most importantly — for your kind words, as well as your support!

I especially want to thank a few readers who shared their feedback with me: Martin Tschendel, Agnieszka Kamińska, and Alexey Shvets. Also, I'd like to thank everyone who left feedback in the LiveBook comments section or in the #ml-bookcamp channel of the DataTalks.Club Slack group.

In chapter 7, I use a dataset with clothes for the image classification project. This dataset was created and curated specifically for this book. I would like to thank everyone who contributed the images of their clothes, especially Kenes Shangerey and Tagias, who contributed 60% of the entire dataset.

In the last chapter, I covered model deployment with Kubernetes and Kubeflow. Kubeflow is a relatively new technology, and some things are not documented well enough yet. That's why I would like to thank my colleagues, Theofilos Papapanagiotou and Antonio Bernardino, for their help with Kubeflow.

Machine Learning Bookcamp would not have reached most of the readers without the help of Manning's marketing department. I specifically would like to thank Lana Klasic and Radmila Ercegovac for their help with arranging events for promoting the book and for running social media campaigns to attract more readers. I would also like to thank my project editor, Deirdre Hiam; my reviewing editor, Adriana Sabo; my copyeditor, Pamela Hunt; and my proofreader, Melody Dolab.

To all the reviewers: Adam Gladstone, Amaresh Rajasekharan, Andrew Courter, Ben McNamara, Billy O'Callaghan, Chad Davis, Christopher Kottmyer, Clark Dorman, Dan Sheikh, George Thomas, Gustavo Filipe Ramos Gomes, Joseph Perenia, Krishna Chaitanya Anipindi, Ksenia Legostay, Lurdu Matha Reddy Kunireddy, Mike Cuddy, Monica Guimaraes, Naga Pavan Kumar T, Nathan Delboux, Nour Taweel, Oliver Korten, Paul Silisteanu, Rami Madian, Sebastian Mohan, Shawn Lam, Vishwesh Ravi Shrimali, William Pompei, your suggestions help to make this a better book.

Last but not least, I would like to thank Luca Massaron for inspiring me to write books. I will never be such a prolific book writer like you, Luca, but thank you for being a great motivation for me!

about this book

Who should read this book

This book is written for people who can program and can grasp the basics of Python quickly. You don't need to have any prior experience with machine learning.

The ideal reader is a software engineer who would like to start working with machine learning. However, a motivated college student who needs to code for studies and side projects will succeed as well.

Additionally, people who already work with machine learning but want to learn more will also find the book useful. Many people who already work as data scientists and data analysts said that it was helpful for them, especially the chapters about deployment.

How this book is organized: a roadmap

This book contains nine chapters, and we work on four different projects throughout the book.

- In chapter 1, we introduce the topic — we discuss the difference between traditional software engineering and machine learning. We cover the process of organizing machine learning projects, from the initial step of understanding the business requirements to the last step of deploying the model. We cover the modeling step in the process in more detail and talk about how we should evaluate our models and select the best one. To illustrate the concepts in this chapter, we use the spam-detection problem.

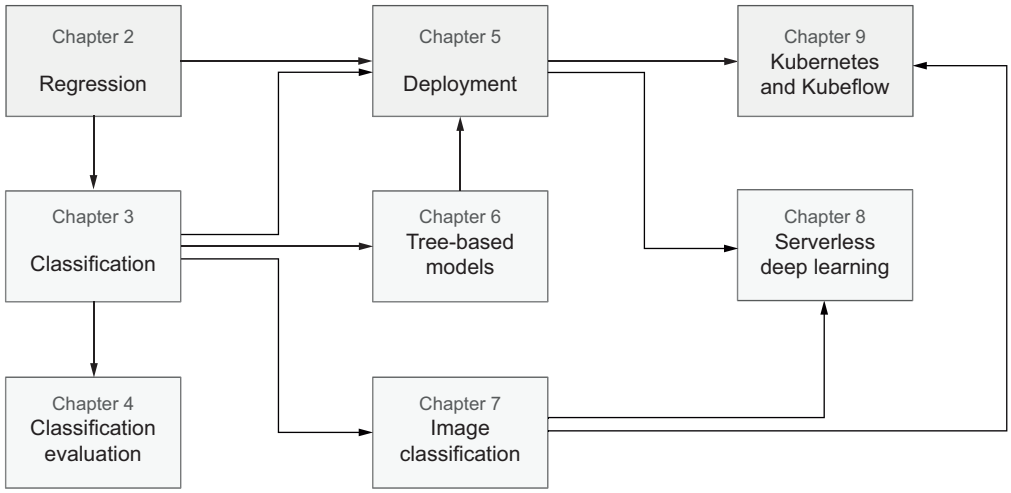
- In chapter 2, we start with our first project — we predict the price of a car. We learn how to use linear regression for that. We first prepare a dataset and do a bit of data cleaning. Next, we perform exploratory data analysis to understand the data better. Then we implement a linear regression model ourselves with NumPy to understand how machine learning models work under the hood. Finally, we discuss topics like regularization and evaluating the quality of the model.
- In chapter 3, we tackle the churn-detection problem. We work in a telecom company and want to determine which customer might stop using our services soon. It's a classification problem that we solve with logistic regression. We start by performing feature importance analysis to understand which factors are the most important ones for this problem. Then we discuss one-hot encoding as a way to handle categorical variables (factors like gender, type of contract, and so on). Finally, we train a logistic regression model with Scikit-learn to understand which customers are going to churn soon.
- In chapter 4, we take the model we developed in chapter 3 and evaluate its performance. We cover the most important classification evaluation metrics: accuracy, precision, and recall. We discuss the confusion table and then go into the details of ROC analysis and calculate AUC. We wrap up this chapter with discussing K-fold cross-validation.
- In chapter 5, we take the churn-prediction model and deploy it as a web service. This is an important step in the process, because if we don't make our model available, it's not useful for anyone. We start with Flask, a Python framework for creating web services. Then we cover Pipenv and Docker for dependency management and finish with deploying our service on AWS.
- In chapter 6, we start a project on risk scoring. We want to understand if a customer of a bank will have problems paying back a loan. For that, we learn how decision trees work and train a simple model with Scikit-learn. Then we move to more complex tree-based models like random forest and gradient boosting.
- In chapter 7, we build an image classification project. We will train a model for classifying images of clothes into 10 categories like T-shirts, dresses, pants, and so on. We use TensorFlow and Keras for training our model, and we cover things like transfer learning for being able to train a model with a relatively small dataset.
- In chapter 8, we take the clothes classification model we trained in chapter 7 and deploy it with TensorFlow Lite and AWS Lambda.
- In chapter 9, we deploy the clothes classification model, but we use Kubernetes and TensorFlow Serving in the first part, and Kubeflow and Kubeflow Serving in the second.

To help you get started with the book as well as Python and libraries around it, we prepared five appendix chapters:

- Appendix A explains how to set up the environment for the book. We show how to install Python with Anaconda, how to run Jupyter Notebook, how to install Docker, and how to create an AWS account
- Appendix B covers the basics of Python.
- Appendix C covers the basics of NumPy and gives a short introduction to the most important linear algebra concepts that we need for machine learning: matrix multiplication and matrix inversion.
- Appendix D covers Pandas.
- Appendix E explains how to get a Jupyter Notebook with a GPU on AWS SageMaker.

These appendices are optional, but they are helpful, especially if you haven't used Python or AWS before.

You don't have to read the book from cover to cover. To help you navigate, you can use this map:



Chapters 2 and 3 are the most important ones. All the other chapters depend on them. After reading them, you jump to chapter 5 to deploy the model, chapter 6 to learn about tree-based models, or chapter 7 to learn about image classifications. Chapter 4, about evaluation metrics, depends on chapter 3: we evaluate the quality of the churn-prediction model from chapter 3. In chapters 8 and 9, we deploy the image classification model, so it's helpful to read chapter 7 before moving on to chapter 8 or 9.

Each chapter contains exercises. It's important to do these exercises — they will help you remember the material a lot better.

About the code

This book contains many examples of source code, both in numbered listings and in line with normal text. In both cases, source code is formatted in a `fixed-width font like this` to separate it from ordinary text. Sometimes code is also **`in bold`** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (➥). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

The code for this book is available on GitHub at https://github.com/alexeygrigorev/mlbookcamp-code. This repository also contains a lot of useful links that will be helpful for you in your machine learning journey.

liveBook discussion forum

Purchase of Machine Learning Bookcamp includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to https://livebook.manning.com/book/machine-learning-bookcamp/welcome/v-11. You can also learn more about Manning's forums and the rules of conduct at https://livebook.manning.com/#!/discussion.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

Other online resources

- The book's website: https://mlbookcamp.com/. It contains useful articles and courses based on the book.
- Community of data enthusiasts: https://datatalks.club. You can ask any question about data or machine learning there.
- There's also a channel for discussing book-related questions: #ml-bookcamp.

about the author

ALEXEY GRIGOREV lives in Berlin with his wife and son. He's an experienced software engineer who focuses on machine learning. He works at OLX Group as a principal data scientist, where he helps his colleagues bring machine learning to production.

After work, Alexey runs DataTalks.Club, a community of people who like data science and machine learning. He's the author of two other books: *Mastering Java for Data Science* and *TensorFlow Deep Learning Projects*.

about the cover illustration

The figure on the cover of Machine Learning Bookcamp is captioned "Femme de Brabant," or a woman from Brabant. The illustration is taken from a collection of dress costumes from various countries by Jacques Grasset de Saint-Sauveur (1757–1810), titled Costumes de Différents Pays, published in France in 1797. Each illustration is finely drawn and colored by hand. The rich variety of Grasset de Saint-Sauveur's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or in the countryside, it was easy to identify where they lived and what their trade or station in life was just by their dress.

The way we dress has changed since then and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns, regions, or countries. Perhaps we have traded cultural diversity for a more varied personal life — certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Grasset de Saint-Sauveur's pictures.

1 Introduction to machine learning

This chapter covers

- Understanding machine learning and the problems it can solve
- Organizing a successful machine learning project
- Training and selecting machine learning models
- Performing model validation

In this chapter, we introduce machine learning and describe the cases in which it's most helpful. We show how machine learning projects are different from traditional software engineering (rule-based solutions) and illustrate the differences by using a spam-detection system as an example.

To use machine learning to solve real-life problems, we need a way to organize machine learning projects. In this chapter, we talk about CRISP-DM: a step-by-step methodology for implementing successful machine learning projects.

Finally, we take a closer look at one of the steps of CRISP-DM — the modeling step. In this step, we train different models and select the one that solves our problem best.

1.1 Machine learning

Machine learning is part of applied mathematics and computer science. It uses tools from mathematical disciplines such as probability, statistics, and optimization theory to extract patterns from data.

The main idea behind machine learning is learning from examples: we prepare a dataset with examples, and a machine learning system "learns" from this dataset. In other words, we give the system the input and the desired output, and the system tries to figure out how to do the conversion automatically, without asking a human.

We can collect a dataset with descriptions of cars and their prices, for example. Then we provide a machine learning model with this dataset and "teach" it by showing it cars and their prices. This process is called *training* or sometimes *fitting* (figure 1.1).

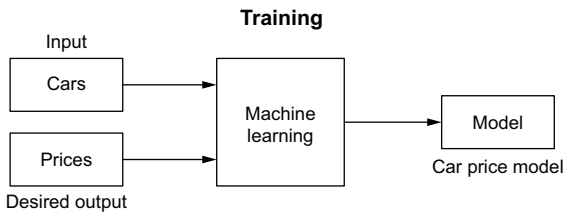


Figure 1.1 A machine learning algorithm takes in input data (descriptions of cars) and desired output (the cars' prices). Based on that data, it produces a model.

When training is done, we can use the model by asking it to predict car prices that we don't know yet (figure 1.2).

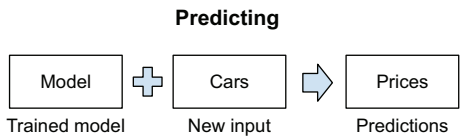


Figure 1.2 When training is done, we have a model that can be applied to new input data (cars without prices) to produce the output (predictions of prices).

All we need for machine learning is a dataset in which for each input item (a car) we have the desired output (the price).

This process is quite different from traditional software engineering. Without machine learning, analysts and developers look at the data they have and try to find patterns manually. After that, they come up with some logic: a set of rules for converting the input data to the desired output. Then they explicitly encode these rules using a

programming language such as Java or Python, and the result is called software. So, in contrast with machine learning, a human does all the difficult work (figure 1.3).

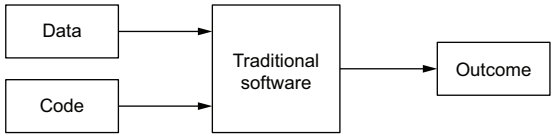
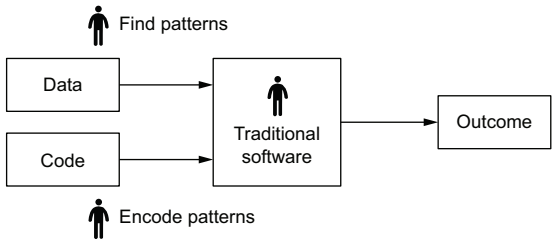
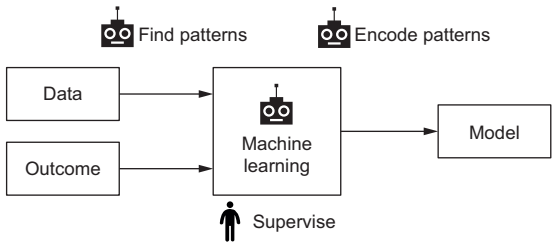


Figure 1.3 In traditional software, patterns are discovered manually and then encoded with a programming language. A human does all the work.

In summary, the difference between a traditional software system and a system based on machine learning is shown in figure 1.4. In machine learning, we give the system the input and output data, and the result is a model (code) that can transform the input into the output. The difficult work is done by the machine; we need only supervise the training process to make sure that the model is good (figure 1.4B). In contrast, in traditional systems, we first find the patterns in the data ourselves and then write code that converts the data to the desired outcome, using the manually discovered patterns (figure 1.4A).



(A) In traditional software we discover patterns manually and encode them using a programming language.



(B) A machine learning system discovers patterns automatically by learning from examples. After training, it produces a model that "knows" these patterns, but we still need to supervise it to make sure the model is correct.

Figure 1.4 The difference between a traditional software system and a machine learning system. In traditional software engineering, we do all the work, whereas in machine learning, we delegate pattern discovery to a machine.

1.1.1 Machine learning vs. rule-based systems

To illustrate the difference between these two approaches and to show why machine learning is helpful, let's consider a concrete case. In this section, we talk about a spam-detection system to show this difference.

Suppose we are running an email service, and the users start complaining about unsolicited emails with advertisements. To solve this problem, we want to create a system that marks the unwanted messages as spam and forwards them to the spam folder.

The obvious way to solve the problem is to look at these emails ourselves to see whether they have any pattern. For example, we can check the sender and the content.

If we find that there's indeed a pattern in the spam messages, we write down the discovered patterns and come up with following two simple rules to catch these messages:

- If sender = promotions@online.com, then "spam"
- If title contains "buy now 50% off" and sender domain is "online.com," then "spam"
- Otherwise, "good email"

We write these rules in Python and create a spam-detection service, which we successfully deploy. At the beginning, the system works well and catches all the spam, but after a while, new spam messages start to slip through. The rules we have are no longer successful at marking these messages as spam.

To solve the problem, we analyze the content of the new messages and find that most of them contain the word *deposit*. So we add a new rule:

- If sender = "promotions@online.com" then "spam"
- If title contains "buy now 50% off" and sender domain is "online.com," then "spam"
- If body contains a word "deposit," then "spam"
- Otherwise, "good email"

After discovering this rule, we deploy the fix to our Python service and start catching more spam, making the users of our mail system happy.

Some time later, however, users start complaining again: some people use the word *deposit* with good intentions, but our system fails to recognize that fact and marks the messages as spam. To solve the problem, we look at the good messages and try to understand how they are different from spam messages. After a while, we discover a few patterns and modify the rules again:

- If sender = "promotions@online.com," then "spam"
- If title contains "buy now 50% off" and sender domain is "online.com," then "spam"

- If body contains “deposit,” then
  - If the sender's domain is “test.com,” then spam
  - If description length is >= 100 words, then spam
- Otherwise, “good email”

In this example, we looked at the input data manually and analyzed it in an attempt to extract patterns from it. As a result of the analysis, we got a set of rules that transforms the input data (emails) to one of the two possible outcomes: spam or not spam.

Now imagine that we repeat this process a few hundred times. As a result, we end up with code that is quite difficult to maintain and understand. At some point, it becomes impossible to include new patterns in the code without breaking the existing logic. So, in the long run, it’s quite difficult to maintain and adjust existing rules such that the spam-detection system still performs well and minimizes spam complaints.

This is exactly the kind of situation in which machine learning can help. In machine learning, we typically don’t attempt to extract these patterns manually. Instead, we delegate this task to statistical methods, by giving the system a dataset with emails marked as spam or not spam and describing each object (email) with a set of its characteristics (features). Based on this information, the system tries to find patterns in the data with no human help. In the end, it learns how to combine the features in such a way that spam messages are marked as spam and good messages aren’t.

With machine learning, the problem of maintaining a hand-crafted set of rules goes away. When a new pattern emerges — for example, there’s a new type of spam — we, instead of manually adjusting the existing set of rules, simply provide a machine learning algorithm with the new data. As a result, the algorithm picks up the new important patterns from the new data without damaging the old existing patterns — provided that these old patterns are still important and present in the new data.

Let’s see how we can use machine learning to solve the spam-classification problem. For that, we first need to represent each email with a set of features. At the beginning we may choose to start with the following features:

- Length of title > 10? true/false
- Length of body > 10? true/false
- Sender “promotions@online.com”? true/false
- Sender “hpYOSKmL@test.com”? true/false
- Sender domain “test.com”? true/false
- Description contains “deposit”? true/false

In this particular case, we describe all emails with a set of six features. Coincidentally, these features are derived from the preceding rules.

With this set of features, we can encode any email as a feature vector: a sequence of numbers that contains all the feature values for a particular email.

Subject: Waiting for your reply
From: prince1@test.com

We are delighted to inform you that you won 1.000.000 (one million) US Dollars. To claim the prize, you need to pay a small processing fee. Please transfer $10 to our PayPal account at prince@test.com. Once we receive the money, we will start the transfer.

Congratulations again!

Spam: True

Figure 1.5 An email that a user marked as spam

Now imagine that we have an email that users marked as spam (figure 1.5). We can express this email as a vector [1, 1, 0, 0, 1, 1], and for each of the six features, we encode the value as 1 for true or 0 for false (figure 1.6). Because our users marked the message as spam, the target variable is 1 (true).

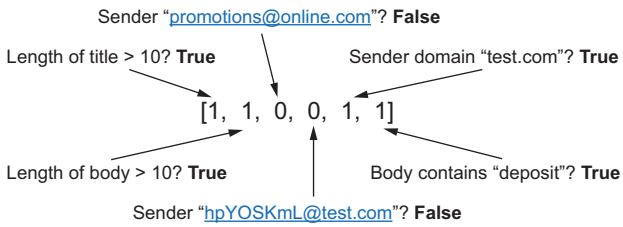


Figure 1.6 The six-dimensional feature vector for a spam email. Each of the six features is represented by a number. In this case, we use 1 if the feature is true and 0 if the feature is false.

This way, we can create feature vectors for all the emails in our database and attach a label to each one. These vectors will be the input to a model. Then the model takes all these numbers and combines the features in such a way that the prediction for spam messages is close to 1 (spam) and is 0 (not spam) for normal messages (figure 1.7).

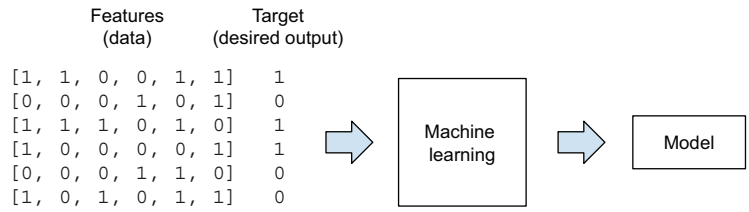


Figure 1.7 The input to a machine learning algorithm consists of multiple feature vectors and the target variable for each vector.

As a result, we have a tool that is more flexible than a set of hardcoded rules. If something changes in the future, we don't have to revisit all the rules manually and try to reorganize them. Instead, we use only the most recent data and replace the old model with the fresh one.

This example is just one way that machine learning can make our lives easier. Other applications of machine learning include

- Suggesting the price of a car.
- Predicting whether a customer will stop using the services of a company.
- Ordering documents by relevance with respect to a query.
- Showing users the ads they are more likely to click instead of irrelevant content.
- Classifying harmful and incorrect edits on Wikipedia. A system like this one can help Wikipedia's moderators prioritize their efforts when validating the suggested edits.
- Recommending items that customers may buy.
- Classifying images in different categories.

Applications of machine learning aren't limited to these examples, of course. We can use literally anything that we can express as (input data, desired output) to train a machine learning model.

1.1.2 When machine learning isn't helpful

Although machine learning is helpful and can solve many problems, it's not really needed in some cases.

For some simple tasks, rules and heuristics often work well, so it's better to start with them and then consider using machine learning. In our spam example, we started by creating a set of rules, but after maintaining this set became difficult, we switched to machine learning. We used some of the rules as features, however, and simply fed them to a model.

In some cases, it's simply not possible to use machine learning. To use machine learning, we need to have data. If no data is available, machine learning is not possible.

1.1.3 Supervised machine learning

The email-classification problem we just looked at is an example of supervised learning: we provide the model with features and the target variable, and it figures out how to use these features to arrive at the target. This type of learning is called *supervised* because we supervise or teach the model by showing it examples, exactly as we would teach children by showing them pictures of different objects and then telling them the names of those objects.

A bit more formally, we can express a supervised machine learning model mathematically as

$$y \approx g(X)$$

where

- g is the function that we want to learn with machine learning.
- X is the feature matrix in which rows are feature vectors.
- y is the target variable: a vector.

The goal of machine learning is to learn this function g in such a way that when it gets the matrix X, the output is close to the vector y. In other words, the function g must be able to take in X and produce y. The process of learning g is usually called *training* or *fitting*. We "fit" g to dataset X in such a way that it produces y (figure 1.8).

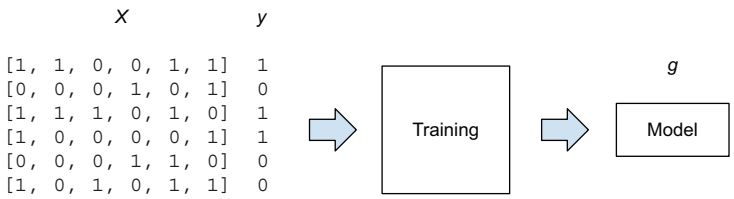


Figure 1.8 When we train a model, an algorithm takes in a matrix X in which feature vectors are rows and the desired output is the vector y, with all the values we want to predict. The result of training is g, the model. After training, g should produce y when applied to X — or, in short, $g(X) \approx y$.

There are different types of supervised learning problems, and the type depends on the target variable y. The main types are

- Regression: the target variable y is numeric, such as a car price or the temperature tomorrow. We cover regression models in chapter 2.
- Classification: the target variable y is categorical, such as spam, not spam, or car make. We can further split classification into two subcategories: (1) *binary classification*, which has only two possible outcomes, such as spam or not spam, and (2) *multiclass classification*, which has more than two possible outcomes, such as a car make (Toyota, Ford, Volkswagen, and so on). Classification, especially binary classification, is the most common application of machine learning. We cover it in multiple chapters throughout the book, starting with chapter 3. In that chapter, we'll build a model for predicting whether a customer is going to churn — stop using the services of our company.
- Ranking: the target variable y is an ordering of elements within a group, such as the order of pages in a search-result page. The problem of ranking often happens in areas like search and recommendations, but it's out of the scope of this book and we won't cover it in detail.

Each supervised learning problem can be solved with different algorithms. Many types of models are available. These models define how exactly function g learns to predict y from X. These models include

- Linear regression for solving the regression problem, covered in chapter 2
- Logistic regression for solving the classification problem, covered in chapter 3
- Tree-based models for solving both regression and classification, covered in chapter 6
- Neural networks for solving both regression and classification, covered in chapter 7

Deep learning and neural networks have received a lot of attention recently, mostly because of breakthroughs in computer vision methods. These networks solve tasks such as image classification a lot better than earlier methods did. *Deep learning* is a subfield of machine learning in which the function g is a neural network with many layers. We will learn more about neural networks and deep learning starting in chapter 7, where we train a deep learning model for image classification.

1.2 Machine learning process

Creating a machine learning system involves more than just selecting a model, training it, and applying it to new data. The model-training part of the process is only a small step in the process.

Many other steps are involved, such as identifying the problem that machine learning can solve and using the predictions of the model to affect the end users. What is more, this process is iterative. When we train a model and apply it to a new dataset, we often identify cases in which the model doesn't perform well. We use these cases to retrain the model in such a way that the new version handles such situations better.

Certain techniques and frameworks help us organize a machine learning project in such a way that it doesn't get out of control. One such framework is CRISP-DM, which stands for *Cross-Industry Standard Process for Data Mining*. It was invented quite long ago, in 1996, but in spite of its age, it's still applicable to today's problems.

According to CRISP-DM (figure 1.9), the machine learning process has six steps:

1. Business understanding
2. Data understanding
3. Data preparation
4. Modeling
5. Evaluation
6. Deployment

Each phase covers typical tasks:

- In the business understanding step, we try to identify the problem, to understand how we can solve it, and to decide whether machine learning will be a useful tool for solving it.
- In the data understanding step, we analyze available datasets and decide whether we need to collect more data.

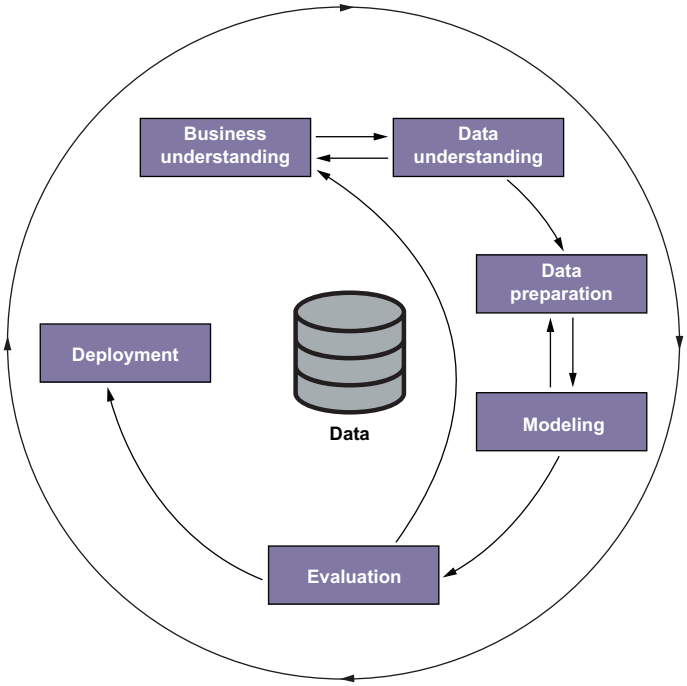


Figure 1.9 The CRISP-DM process. A machine learning project starts with understanding the problem and then moves into data preparation, training the model, and evaluating the results. Finally, the model goes to deployment. This process is iterative, and at each step, it's possible to go back to the previous one.

- In the data preparation step, we transform the data into a tabular form that we can use as input for a machine learning model.
- When the data is prepared, we move to the modeling step, in which we train a model.
- After the best model is identified, there's the evaluation step, where we evaluate the model to see if it solves the original business problem and measure its success at doing that.
- Finally, in the deployment step, we deploy the model to the production environment.

1.2.1 Business understanding

Let's consider the spam-detection example for an email service provider. We see more spam messages than ever before, and our current system cannot deal with it easily. This problem is addressed in the business understanding step: we analyze the problem and the existing solution and try to determine if adding machine learning to that system will help us stop spam messages. We also define the goal and how to measure it.

The goal could be "Reduce the amount of reported spam messages" or "Reduce the amount of complaints about spam that customer support receives per day," for example. In this step, we may also decide that machine learning is not going to help and propose a simpler way to solve the problem.

1.2.2 Data understanding

The next step is data understanding. Here, we try to identify the data sources we can use to solve the problem. If our site has a Report Spam button, for example, we can get data generated by users who marked their incoming emails as spam. Then we look at the data and analyze it to decide whether it's good enough to solve our problem.

This data may not be good enough, however, for a wide range of reasons. One reason could be that the dataset is too small for us to learn any useful patterns. Another reason could be that the data is too noisy. The users may not use the button correctly, so it will be useless for training a machine learning model, or the data-collection process could be broken, gathering only a small fraction of the data we want.

If we conclude that the data we currently have is not sufficient, we need to find a way to get better data, whether we acquire it from external sources or improve the way we collect it internally. It's also possible that discoveries we make in this step will influence the goal we set in the business understanding step, so we may need to go back to that step and adjust the goal according to our findings.

When we have reliable data sources, we go to the data preparation step.

1.2.3 Data preparation

In this step, we clean the data, transforming it in such a way that it can be used as input for a machine learning model. For the spam example, we transform the dataset into a set of features that we feed into a model later.

After the data is prepared, we go to the modeling step.

1.2.4 Modeling

In this step, we decide which machine learning model to use and how to make sure that we get the best out of it. For example, we may decide to try logistic regression and a deep neural network to solve the spam problem.

We need to know how we will measure the performance of the models to select the best one. For the spam model, we can look at how well the model predicts spam messages and choose the one that does it best. For this purpose, setting a proper validation framework is important, which is why we cover it in more detail in the next section.

It's very likely that in this step, we need to go back and adjust the way we prepare the data. Perhaps we came up with a great feature, so we go back to the data preparation step to write some code to compute that feature. When the code is done, we train the model again to check whether this feature is good. We might add a feature "length of the subject," retrain the model, and check whether this change improves the model's performance, for example.

After we select the best possible model, we go to the evaluation step.

1.2.5 Evaluation

In this step, we check whether the model lives up to expectations. When we set the goal in the business understanding step, we also define the way of establishing whether the goal is achieved. Typically, we do this by looking at an important business metric and making sure that the model moves the metric in the right direction. In the case of spam detection, the metric could be the number of people who click the Report Spam button or the number of complaints about the issue we're solving that customer support receives. In both cases, we hope that using the model reduces the number.

Nowadays, this step is tightly connected to the next step: deployment.

1.2.6 Deployment

The best way to evaluate a model is to battle-test it: roll it out to a fraction of users and then check whether our business metric changes for these users. If we want our model to reduce the number of reported spam messages, for example, we expect to see fewer reports from this group compared with the rest of the users.

After the model is deployed, we use everything we learned in all the steps and go back to the first step to reflect on what we achieved (or didn't achieve). We may realize that our initial goal was wrong and that what we actually want to do is *not* reduce the number of reports but increase customer engagement by decreasing the amount of spam. So we go all the way back to the business understanding step to redefine our goal. Then, when we evaluate the model again, we use a different business metric to measure its success.

1.2.7 Iterate

As we can see, CRISP-DM emphasizes the iterative nature of machine learning processes: after the last step, we are always expected to go back to the first step, refine the original problem, and change it based on the learned information. We never stop at the last step; instead, we rethink the problem and see what we can do better in the next iteration.

It's a common misconception that machine learning engineers and data scientists spend their entire day training machine learning models. In reality, this idea is incorrect, as we can see in the CRISP-DM diagram (figure 1.9). A lot of steps come before and after the modeling step, and all these steps are important for a successful machine learning project.

1.3 Modeling and model validation

As we saw previously, training models (the modeling step) is only one step in the whole process. But it's an important step because it's where we actually use machine learning to train models.

After we collect all the required data and determine that it's good, we find a way to process the data and then proceed to training a machine learning model. In our spam

example, this happens after we get all the spam reports, process the emails, and have a matrix ready to be put to a model.

At this point, we may ask ourselves what to use: logistic regression or a neural network. If we decide to go with a neural network because we've heard it's the best model, how can we make sure that it's indeed better than any other model?

The goal at this step is to produce a model in such a way that it achieves the best predictive performance. To do this, we need to have a way to reliably measure the performance of each possible model candidate and then choose the best one.

One possible approach is to train a model, let it run on a live system, and observe what happens. In the spam example, we decide to use a neural network for detecting spam, so we train it and deploy it to our production system. Then we observe how the model behaves on new messages and record the cases in which the system is incorrect.

This approach, however, is not ideal for our case: we cannot possibly do it for every model candidate we have. What's worse, we can accidentally deploy a really bad model and see that it's bad only after it has been run on live users of our system.

> **NOTE** Testing a model on a live system is called online testing, and it's important for evaluating the quality of a model on real data. This approach, however, belongs to the evaluation and deployment steps of the process, not to the modeling step.

A better approach for selecting the best model before deploying it is emulating the scenario of going live. We get our complete dataset, take a part out of it, and train the model on the remaining part of the data. When the training is done, we pretend that the held-out dataset is the new, unseen data, and we use it to measure the performance of our models. This part of data is often called the *validation set*, and the process of keeping part of a dataset away and using it to evaluate performance is called *validation* (see figure 1.10).

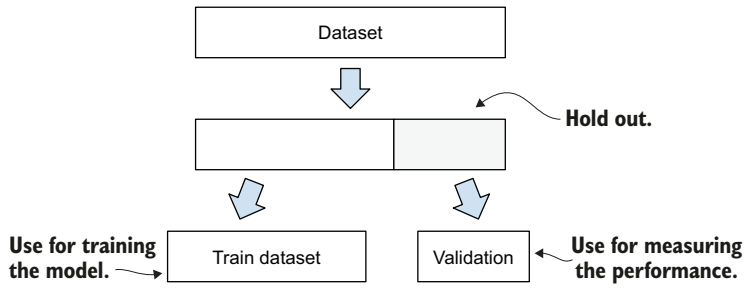


Figure 1.10 To evaluate the performance of a model, we set some data aside and use it only for validation purposes.

In the spam dataset, we can take out every tenth message. This way, we hold out 10% of the data, which we use only for validating the models, and use the remaining 90%

for training. Next, we train both logistic regression and a neural network on the training data. When the models are trained, we apply them to the validation dataset and check which one is more accurate at predicting spam.

If, after applying the models to validation, we see that logistic regression is correct in predicting the spam in only 90% of cases, whereas a neural network is correct in 93% of cases, we conclude that the neural network model is a better choice than logistic regression (figure 1.11).

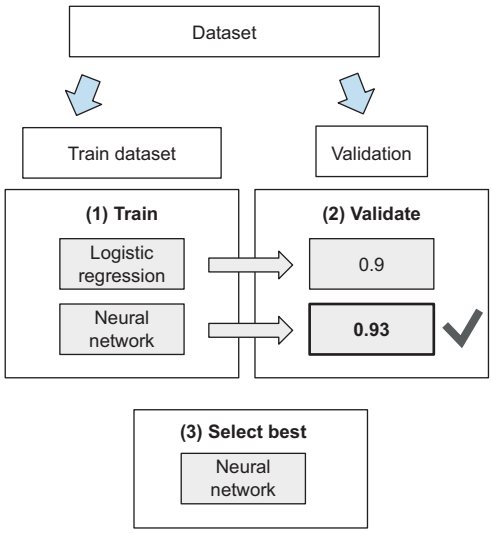


Figure 1.11 The validation process. We split the dataset into two parts, train the models on the training part, and evaluate the performance on the validation part. Using the evaluation results, we can choose the best model.

Often, we don't have just two models to try but a lot more. Logistic regression, for example, has a parameter, C, and depending on the value we set, the results can vary dramatically. Likewise, a neural network has many parameters, and each may have a great effect on the predictive performance of the final model. On top of that, we have other models, each with its own set of parameters. How do we select the best model with the best parameters?

To do so, we use the same evaluation scheme. We train the models with different parameters on the training data, apply them to the validation data, and then select the model and its parameters based on the best validation results (figure 1.12).

This approach has a subtle problem, however. If we repeat the process of model evaluation over and over again and use the same validation dataset for that purpose, the good numbers we observe in the validation dataset may appear just by chance. In other words, the "best" model may simply get lucky in predicting the outcomes for this particular dataset.

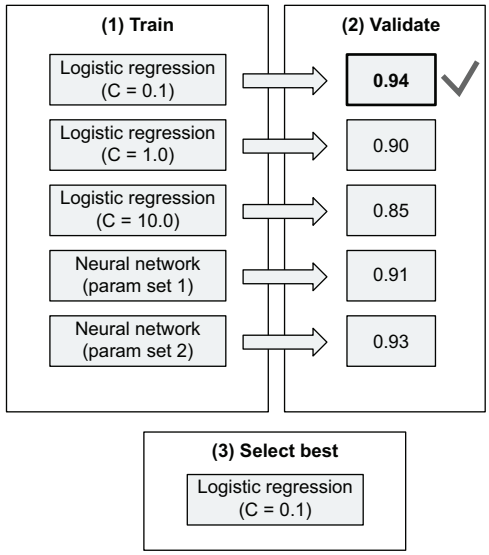


Figure 1.12 Using the validation dataset to select the best model with the best parameters

NOTE In statistics and other fields, this problem is known as the multiple-comparisons problem or multiple-tests problem. The more times we make predictions on the same dataset, the more likely we are to see good performance by chance.

To guard against this problem, we use the same idea: we hold out part of the data again. We call this part of data the *test* dataset. We use it rarely, only for testing the model that we selected as the best (figure 1.13).

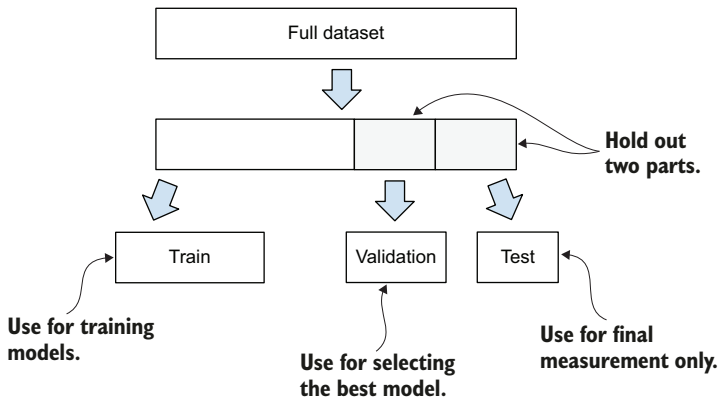


Figure 1.13 Splitting the data into training, testing, and validation parts

To apply this to the spam example, we first hold out 10% of the data as the test dataset and then hold out 10% of the data as the validation. We try multiple models on the validation dataset, select the best one, and apply it to the test dataset. If we see that the difference in performance between validation and test is not big, we confirm that this model is indeed the best one (figure 1.14).

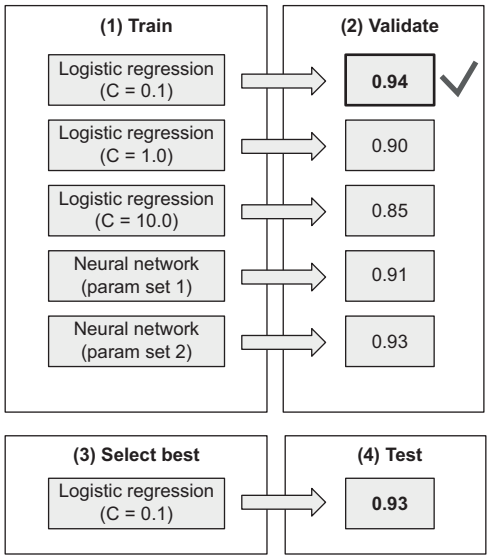


Figure 1.14 We use the test dataset to confirm that the performance of the best model on the validation set is good.

IMPORTANT Setting the validation process is the most important step in machine learning. Without it, there's no reliable way to know whether the model we've just trained is good, useless, or even harmful.

The process of selecting the best model and the best parameters for the model is called *model selection*. We can summarize model selection as follows (figure 1.15):

1. We split the data into training, validation, and testing parts.
2. We train each model first on the training part and then evaluate it on validation.
3. Each time we train a different model, we record the evaluation results using the validation part.
4. At the end, we determine which model is the best and test it on the test dataset.

It's important to use the model selection process and to validate and test the models in offline settings first to make sure that the models we train are good. If the model behaves well offline, we can decide to move to the next step and deploy the model to evaluate its performance with real users.

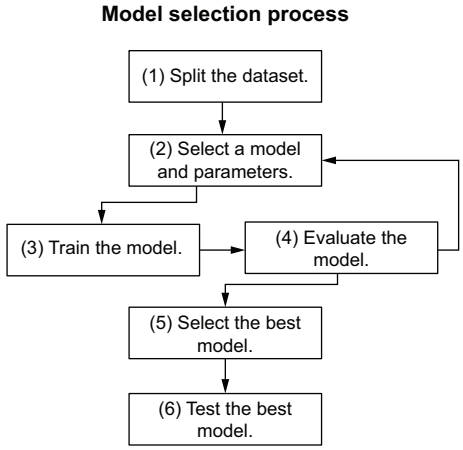


Figure 1.15 The model selection process. First, we split the dataset, select a model, and train it only on the training part of the data. Then we evaluate the model on the validation part. We repeat the process many times until we find the best model.

Summary

- Unlike traditional rule-based software engineering systems, in which rules are extracted and coded manually, machine learning systems can be taught to extract meaningful patterns from data automatically. This gives us a lot more flexibility and makes it easier to adapt to changes.
- Successfully implementing a machine learning project requires a structure and a set of guidelines. CRISP-DM is a framework for organizing a machine learning project that breaks down the process into six steps, from business understanding to deployment. The framework highlights the iterative nature of machine learning and helps us stay organized.
- Modeling is an important step in a machine learning project: the part where we actually use machine learning to train a model. During this step, we create models that achieve the best predictive performance.
- Model selection is the process of choosing the best model to solve a problem. We split all the available data into three parts: training, validation, and testing. We train models on the training set and select the best model by using the validation set. When the best model is selected, we use the test step as a final check to ensure that the best model behaves well. This process helps us create useful models that work well with no surprises.

2 Machine learning for regression

This chapter covers

- Creating a car-price prediction project with a linear regression model
- Doing an initial exploratory data analysis with Jupyter notebooks
- Setting up a validation framework
- Implementing the linear regression model from scratch
- Performing simple feature engineering for the model
- Keeping the model under control with regularization
- Using the model to predict car prices

In chapter 1, we talked about supervised machine learning, in which we teach machine learning models how to identify patterns in data by giving them examples.

Suppose that we have a dataset with descriptions of cars, like make, model, and age, and we would like to use machine learning to predict their prices. These characteristics of cars are called *features*, and the price is the *target variable* — something

we want to predict. Then the model gets the features and combines them to output the price.

This is an example of supervised learning: we have some information about the price of some cars, and we can use it to predict the price of others. In chapter 1, we also talked about different types of supervised learning: regression and classification. When the target variable is numerical, we have a regression problem, and when the target variable is categorical, we have a classification problem.

In this chapter, we create a regression model, starting with the simplest one: linear regression. We implement the algorithms ourselves, which is simple enough to do in a few lines of code. At the same time, it's very illustrative, and it will teach you how to deal with NumPy arrays and perform basic matrix operations such as matrix multiplication and matrix inversion. We also come across problems of numerical instability when inverting a matrix and see how regularization helps solve them.

2.1 Car-price prediction project

The problem we solve in this chapter is predicting the price of a car. Suppose that we have a website where people can sell and buy used cars. When posting an ad on our website, sellers often struggle to come up with a meaningful price. We want to help our users with automatic price recommendations. We ask the sellers to specify the model, make, year, mileage, and other important characteristics of a car, and based on that information, we want to suggest the best price.

One of the product managers in the company accidentally came across an open dataset with car prices and asked us to have a look at it. We checked the data and saw that it contained all the important features as well as the recommended price — exactly what we needed for our use case. Thus, we decided to use this dataset for building the price-recommendation algorithm.

The plan for the project is the following:

1. First, we download the dataset.
2. Next, we do some preliminary analysis of the data.
3. After that, we set up a validation strategy to make sure our model produces correct predictions.
4. Then we implement a linear regression model in Python and NumPy.
5. Next, we cover feature engineering to extract important features from the data to improve the model.
6. Finally, we see how to make our model stable with regularization and use it to predict car prices.

2.1.1 Downloading the dataset

The first thing we do for this project is install all the required libraries: Python, NumPy, Pandas, and Jupyter Notebook. The easiest way to do it is to use a Python distribution called Anaconda (https://www.anaconda.com). Please refer to appendix A for installation guidelines.

After the libraries are installed, we need to download the dataset. We have multiple options for doing this. You can download it manually through the Kaggle web interface, available at https://www.kaggle.com/CooperUnion/cardataset. (You can read more about the dataset and the way it was collected at https://www.kaggle.com/jshih7/car-price-prediction.) Go there, open it, and click the download link. The other option is using the Kaggle command-line interface (CLI), which is a tool for programmatic access to all datasets available via Kaggle. For this chapter, we will use the second option. We describe how to configure the Kaggle CLI in appendix A.

> **NOTE** Kaggle is an online community for people who are interested in machine learning. It is mostly known for hosting machine learning competitions, but it is also a data-sharing platform where anyone can share a dataset. More than 16,000 datasets are available for anyone to use. It is a great source of project ideas and very useful for machine learning projects.

In this chapter, as well as throughout the book, we will actively use NumPy. We cover all necessary NumPy operations as we go along, but please refer to appendix C for a more in-depth introduction.

The source code for this project is available in the book's repository in GitHub at https://github.com/alexeygrigorev/mlbookcamp-code in chapter-02-car-price.

As the first step, we will create a folder for this project. We can give it any name, such as chapter-02-car-price:

```
mkdir chapter-02-car-price
cd chapter-02-car-price
```

Then we download the dataset:

```
kaggle datasets download -d CooperUnion/cardataset
```

This command downloads the cardataset.zip file, which is a zip archive. Let's unpack it:

```
unzip cardataset.zip
```

Inside, there's one file: data.csv.

When we have the dataset, let's move on to the next step: understanding it.

2.2 Exploratory data analysis

Understanding data is an important step in the machine learning process. Before we can train any model, we need to know what kind of data we have and whether it is useful. We do this with exploratory data analysis (EDA).

We look at the dataset to learn

- The distribution of the target variable
- The features in this dataset
- The distribution of values in these features

- The quality of the data
- The number of missing values

2.2.1 Exploratory data analysis toolbox

The main tools for this analysis are Jupyter Notebook, Matplotlib, and Pandas:

- Jupyter Notebook is a tool for interactive execution of Python code. It allows us to execute a piece of code and immediately see the outcome. In addition, we can display charts and add notes with comments in free text. It also supports other languages such as R or Julia (hence the name: Jupyter stands for Julia, Python, R), but we will use it only for Python.
- Matplotlib is a library for plotting. It is very powerful and allows you to create different types of visualizations, such as line charts, bar charts, and histograms.
- Pandas is a library for working with tabular data. It can read data from any source, be it a CSV file, a JSON file, or a database.

We will also use Seaborn, another tool for plotting that is built on top of Matplotlib and makes it easier to draw charts.

Let's start a Jupyter Notebook by executing the following command:

```
jupyter notebook
```

This command starts a Jupyter Notebook server in the current directory and opens it in the default web browser (figure 2.1).

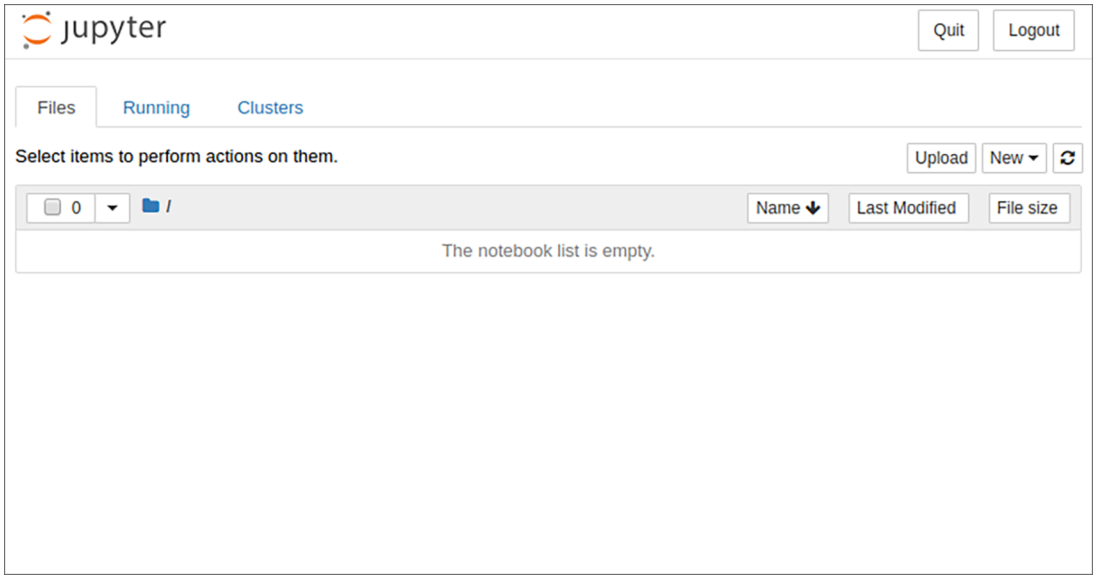
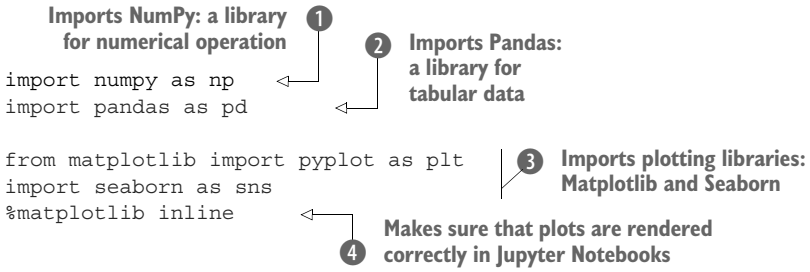


Figure 2.1 The starting screen of the Jupyter Notebook service

If Jupyter is running on a remote server, it requires additional configuration. Please refer to appendix A for details on the setup.

Now let's create a notebook for this project. Click New, then select Python 3 in the Notebooks section. We can call it chapter-02-car-price-project — click the current title (Untitled), and replace it with the new one.

First, we need to import all the libraries required for this project. Write the following in the first cell:



The first two lines, ❶ and ❷, are imports for required libraries: NumPy for numeric operations and Pandas for tabular data. The convention is to import these libraries using shorter aliases (such as `pd` in `import pandas as pd`). This convention is common in the Python machine learning community, and everybody follows it.

The next two lines, ❸, are imports for plotting libraries. The first one, Matplotlib, is a library for creating good-quality visualizations. It's not always easy to use this library as is. Some libraries make using Matplotlib simpler, and Seaborn is one of them.

Finally, `%matplotlib inline` in line ❹ tells Jupyter to expect plots in the notebook, so it will be able to render them when we need them.

Press Shift+Enter or click Run to execute the content of the selected cell.

We will not get into more detail about Jupyter Notebooks. Check the official website (https://jupyter.org) to learn more about it. The site has plenty of documentation and examples that will help you master it.

2.2.2 Reading and preparing data

Now let's read our dataset. We can use the `read_csv` function from Pandas for that purpose. Put the following code in the next cell and again press Shift+Enter:

```
df = pd.read_csv('data.csv')
```

This line of code reads the CSV file and writes the results to a variable named `df`, which is short for *DataFrame*. Now we can check how many rows there are. Let's use the `len` function:

```
len(df)
```

The function prints 11914, which means that there are almost 12,000 cars in this dataset (figure 2.2).

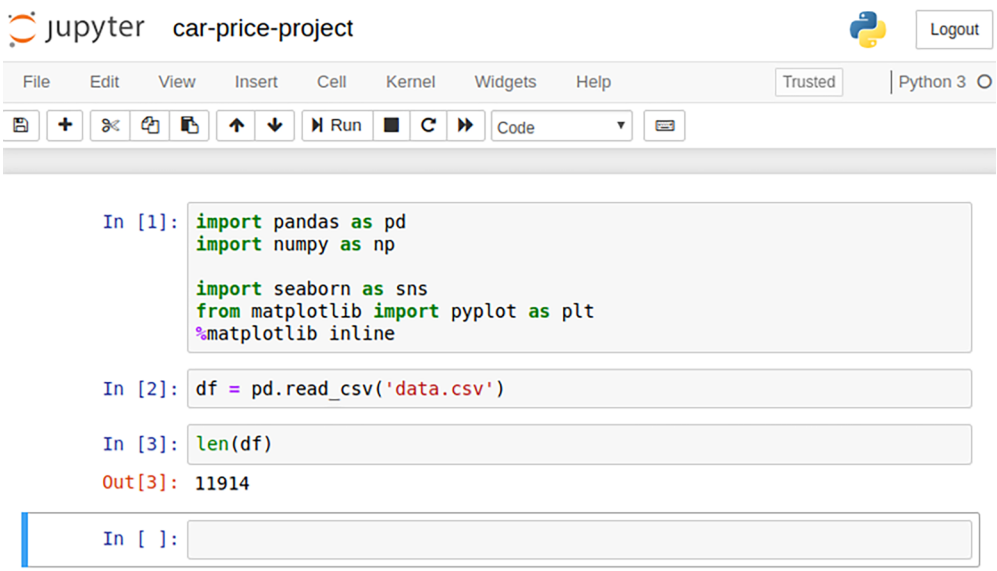


Figure 2.2 Jupyter Notebooks are interactive. We can type some code in a cell, execute it, and see the results immediately, which is ideal for exploratory data analysis.

Now let's use `df.head()` to look at the first five rows of our DataFrame (figure 2.3).

```
In [4]: df.head()
```

Out[4]:

	Make	Model	Year	Engine Fuel Type	Engine HP	Engine Cylinders	Transmission Type	Driven_Wheels	Number of Doors	
0	BMW	1 Series M	2011	premium unleaded (required)	335.0	6.0	MANUAL	rear wheel drive	2.0	T
1	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	Lux
2	BMW	1 Series	2011	premium unleaded (required)	300.0	6.0	MANUAL	rear wheel drive	2.0	
3	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	Lu
4	BMW	1 Series	2011	premium unleaded (required)	230.0	6.0	MANUAL	rear wheel drive	2.0	

Figure 2.3 The output of the `head()` function of a Pandas DataFrame: it shows the first five rows of the dataset. This output allows us to understand what the data looks like.

This gives us an idea of what the data looks like. We can already see that there are some inconsistencies in this dataset: the column names sometimes have spaces, and sometimes have underscores (_). The same is true for feature values: sometimes they're capitalized, and sometimes they are short strings with spaces. This is inconvenient and confusing, but we can solve this by normalizing them — replacing all spaces with underscores and lowercase all letters:

```
df.columns = df.columns.str.lower().str.replace(' ', '_')

string_columns = list(df.dtypes[df.dtypes == 'object'].index)

for col in string_columns:
    df[col] = df[col].str.lower().str.replace(' ', '_')
```

❶ Lowercases all the column names, and replaces spaces with underscores

❷ Selects only columns with string values

❸ Lowercases and replaces spaces with underscores for values in all string columns of the DataFrame

In ❶ and ❸, we use the special `str` attribute. Using it, we can apply string operations to the entire column at that same time without writing any `for` loops. We use it to lowercase the column names and the content of these columns as well as to replace spaces with underscores.

We can use this attribute only for columns with string values inside. This is exactly why we first select such columns in ❷.

> **NOTE** In this chapter and subsequent chapters, we cover relevant Pandas operations as we go along, but at a fairly high level. Please refer to appendix D for a more consistent and in-depth introduction to Pandas.

After this initial preprocessing, the DataFrame looks more uniform (figure 2.4).

```
In [6]: df.head()
```

Out[6]:

	make	model	year	engine_fuel_type	engine_hp	engine_cylinders	transmission_type	driven_wheels	n
0	bmw	1_series_m	2011	premium_unleaded_(required)	335.0	6.0	manual	rear_wheel_drive	
1	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	rear_wheel_drive	
2	bmw	1_series	2011	premium_unleaded_(required)	300.0	6.0	manual	rear_wheel_drive	
3	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	rear_wheel_drive	
4	bmw	1_series	2011	premium_unleaded_(required)	230.0	6.0	manual	rear_wheel_drive	

Figure 2.4 The result of preprocessing the data. The column names and values are normalized: they are lowercase, and the spaces are converted to underscores.

As we see, this dataset contains multiple columns:

- make: make of a car (BMW, Toyota, and so on)
- model: model of a car
- year: year when the car was manufactured
- engine_fuel_type: type of fuel the engine needs (diesel, electric, and so on)
- engine_hp: horsepower of the engine
- engine_cylinders: number of cylinders in the engine
- transmission_type: type of transmission (automatic or manual)
- driven_wheels: front, rear, all
- number_of_doors: number of doors a car has
- market_category: luxury, crossover, and so on
- vehicle_size: compact, midsize, or large
- vehicle_style: sedan or convertible
- highway_mpg: miles per gallon (mpg) on the highway
- city_mpg: miles per gallon in the city
- popularity: number of times the car was mentioned in a Twitter stream
- msrp: manufacturer's suggested retail price

For us, the most interesting column here is the last one: MSRP (manufacturer's suggested retail price, or simply the price of a car). We will use this column for predicting the prices of a car.

2.2.3 Target variable analysis

The MSRP column contains the important information — it's our target variable, the y, which is the value that we want to learn to predict.

One of the first steps of exploratory data analysis should always be to look at what the values of y look like. We typically do this by checking the distribution of y: a visual description of what the possible values of y can be and how often they occur. This type of visualization is called a *histogram*.

We will use Seaborn to plot the histogram, so type the following in the Jupyter Notebook:

```
sns.histplot(df.msrp, bins=40)
```

After plotting this graph, we immediately notice that the distribution of prices has a very long tail. There are many cars with low prices on the left side, but the number quickly drops, and there's a long tail of very few cars with high prices (see figure 2.5).

We can have a closer look by zooming in a bit and looking at values below $100,000 (figure 2.6):

```
sns.histplot(df.msrp[df.msrp < 100000])
```

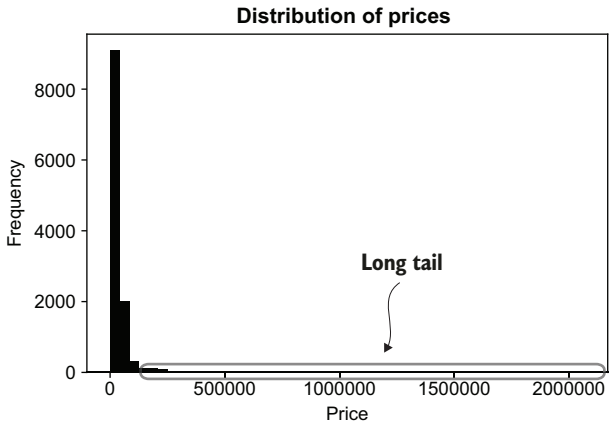


Figure 2.5 The distribution of the prices in the dataset. We see many values at the low end of the price axis and almost nothing at the high end. This is a long tail distribution, which is a typical situation for many items with low prices and very few expensive ones.

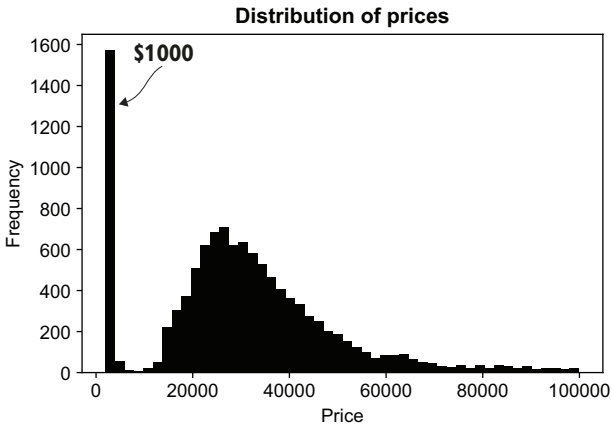


Figure 2.6 The distribution of the prices for cars below $100,000. Looking only at car prices below $100,000 allows us to see the head of the distribution better. We also notice a lot of cars that cost $1,000.

The long tail makes it quite difficult for us to see the distribution, but it has an even stronger effect on a model: such distribution can greatly confuse the model, so it won't learn well enough. One way to solve this problem is log transformation. If we apply the log function to the prices, it removes the undesired effect (figure 2.7).

$$y_{\text{new}} = \log(y + 1)$$

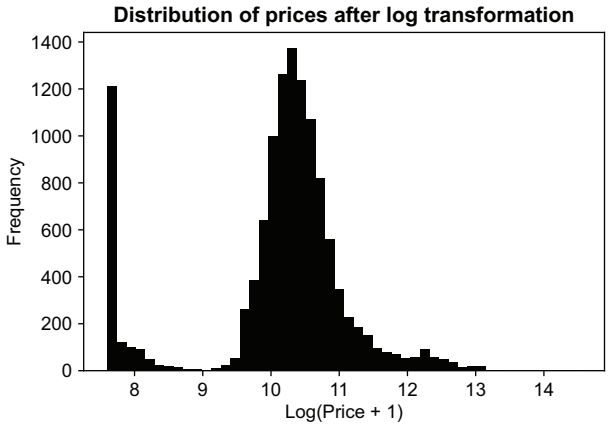


Figure 2.7 The logarithm of the price. The effect of the long tail is removed, and we can see the entire distribution in one plot.

The +1 part is important in cases that have zeros. The logarithm of zero is minus infinity, but the logarithm of one is zero. If our values are all non-negative, by adding 1, we make sure that the transformed values do not go below zero.

For our specific case, zero values are not an issue — all the prices we have start at $1,000 — but it's still a convention that we follow. NumPy has a function that performs this transformation:

```
log_price = np.log1p(df.msrp)
```

To look at the distribution of the prices after the transformation, we can use the same `histplot` function (figure 2.7):

```
sns.histplot(log_price)
```

As we see, this transformation removes the long tail, and now the distribution resembles a bell-shaped curve. This distribution is not normal, of course, because of the large peak in lower prices, but the model can deal with it more easily.

NOTE Generally, it's good when the target distribution looks like the normal distribution (figure 2.8). Under this condition, models such as linear regression perform well.

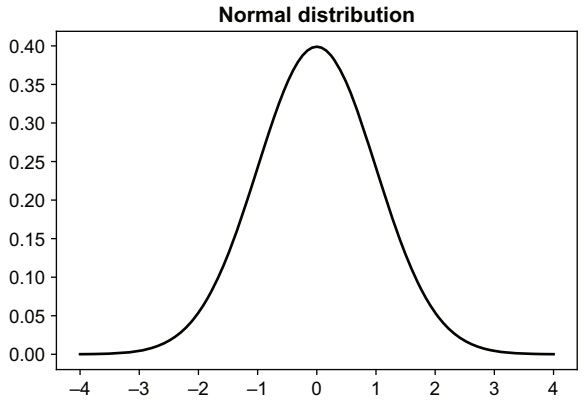


Figure 2.8 The normal distribution, also known as Gaussian, follows the bell-shaped curve, which is symmetrical and has a peak in the center.

Exercise 2.1

The head of a distribution is a range where there are many values. What is a long tail of a distribution?

a A big peak around 1,000 USD
b A case when many values are spread very far from the head — and these values visually appear as a "tail" on the histogram
c A lot of very similar values packed together within a short range

2.2.4 Checking for missing values

We will look more closely at other features a bit later, but one thing we should do now is check for missing values in the data. This step is important because, typically, machine learning models cannot deal with missing values automatically. We need to know whether we need to do anything special to handle those values.

Pandas has a convenient function that checks for missing values:

```
df.isnull().sum()
```

This function shows

```
make                  0
model                 0
year                  0
engine_fuel_type      3
engine_hp            69
engine_cylinders     30
transmission_type     0
driven_wheels         0
number_of_doors       6
```

```
market_category      3742
vehicle_size            0
vehicle_style           0
highway_mpg             0
city_mpg                0
popularity              0
msrp                    0
```

The first thing we see is that MSRP — our target variable — doesn't have any missing values. This result is good, because otherwise, such records won't be useful to us: we always need to know the target value of an observation to use it for training the model. Also, a few columns have missing values, especially market_category, in which we have almost 4,000 rows with missing values.

We need to deal with missing values later when we train the model, so we should keep this problem in mind. For now, we don't do anything else with these features and proceed to the next step: setting up the validation framework so that we can train and test machine learning models.

2.2.5 Validation framework

As we learned previously, it's important to set up the validation framework as early as possible to make sure that the models we train are good and can generalize — that is, that the model can be applied to new, unseen data. To do that, we put aside some data and train the model only on one part. Then we use the held-out dataset — the one we didn't use for training — to make sure that the predictions of the model make sense.

This step is important because we train the model by using optimization methods that fit the function $g(X)$ to the data X. Sometimes these optimization methods pick up spurious patterns — patterns that appear to be real patterns to the model but in reality are random fluctuations. If we have a small training dataset in which all BMW cars cost only $10,000, for example, the model will think that this is true for all BMW cars in the world.

To ensure that this doesn't happen, we use validation. Because the validation dataset is not used for training the model, the optimization method did not see this data. When we apply the model to this data, it emulates the case of applying the model to new data that we've never seen. If the validation dataset has BMW cars with prices higher than $10,000, but our model will predict $10,000 on them, we will notice that the model doesn't perform well on these examples.

As we already know, we need to split the dataset into three parts: train, validation, and test (figure 2.9).

Let's split the DataFrame such that

- 20% of data goes to validation.
- 20% goes to test.
- The remaining 60% goes to train.

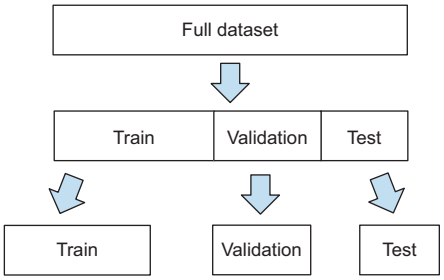


Figure 2.9 The entire dataset is split into three parts: train, validation and test.

Listing 2.1 Splitting Data into validation, test, and training sets

```
n = len(df)

n_val = int(0.2 * n)
n_test = int(0.2 * n)
n_train = n - (n_val + n_test)

np.random.seed(2)
idx = np.arange(n)
np.random.shuffle(idx)

df_shuffled = df.iloc[idx]

df_train = df_shuffled.iloc[:n_train].copy()
df_val = df_shuffled.iloc[n_train:n_train+n_val].copy()
df_test = df_shuffled.iloc[n_train+n_val:].copy()
```

1. Gets the number of rows in the DataFrame
2. Calculates how many rows should go to train, validation, and test
3. Fixes the random seed to make sure that the results are reproducible
4. Creates a NumPy array with indices from 0 to (n–1), and shuffles it
5. Uses the array with indices to get a shuffled DataFrame
6. Splits the shuffled DataFrame into train, validation, and test

Let's take a closer look at this code and clarify a few things.

In 4, we create an array and then shuffle it. Let's see what happens there. We can take a smaller array of five elements and shuffle it:

```
idx = np.arange(5)
print('before shuffle', idx)
np.random.shuffle(idx)
print('after shuffle', idx)
```

If we run it, it prints something similar to

```
before shuffle [0 1 2 3 4]
after shuffle  [2 3 0 4 1]
```

If we run it again, however, the results will be different:

```
before shuffle [0 1 2 3 4]
after shuffle  [4 3 0 2 1]
```

To make sure that every time we run it, the results are the same, in ❸ we fix the random seed:

```
np.random.seed(2)
idx = np.arange(5)
print('before shuffle', idx)
np.random.shuffle(idx)
print('after shuffle', idx)
```

The function `np.random.seed` takes in any number and uses this number as the starting seed for all the generated data inside NumPy's random package.

When we execute this code, it prints

```
before shuffle [0 1 2 3 4]
after shuffle  [2 4 1 3 0]
```

In this case the results are still random, but when we re-execute it, the result turns out to be the same as the previous run:

```
before shuffle [0 1 2 3 4]
after shuffle  [2 4 1 3 0]
```

This is good for reproducibility. If we want somebody else to run this code and get the same results, we need to make sure that everything is fixed, even the "random" component of our code.

> **NOTE** This makes the results reproducible on the same computer. With a different operating system and a different version of NumPy, the result may be different.

After we create an array with indices `idx`, we can use it to get a shuffled version of our initial DataFrame. For that purpose in ❺, we use `iloc`, which is a way to access the rows of the DataFrame by their numbers:

```
df_shuffled = df.iloc[idx]
```

If `idx` contains shuffled consequent numbers, this code will produce a shuffled DataFrame (figure 2.10).

	make	model	year	msrp
0	lotus	evora_400	2017	91900
1	aston_martin	v8_vantage	2014	136900
2	hyundai	genesis	2015	38000
3	suzuki	samurai	1993	2000
4	mitsubishi	outlander	2015	26195

`df.iloc[idx]`

`idx = [2, 4, 1, 3, 0]`

	make	model	year	msrp
2	hyundai	genesis	2015	38000
4	mitsubishi	outlander	2015	26195
1	aston_martin	v8_vantage	2014	136900
3	suzuki	samurai	1993	2000
0	lotus	evora_400	2017	91900

Figure 2.10 Using `iloc` to shuffle a DataFrame. When used with a shuffled array of indices, it creates a shuffled DataFrame.

In this example, we used `iloc` with a list of indices. We can also use ranges with the colon operator (:), and this is exactly what we do in ❻ for splitting the shuffled DataFrame into train, validation, and test:

```
df_train = df_shuffled.iloc[:n_train].copy()
df_val = df_shuffled.iloc[n_train:n_train+n_val].copy()
df_test = df_shuffled.iloc[n_train+n_val:].copy()
```

Now the DataFrame is split into three parts, and we can continue. Our initial analysis showed a long tail in the distribution of prices, and to remove its effect, we need to apply the log transformation. We can do that for each DataFrame separately:

```
y_train = np.log1p(df_train.msrp.values)
y_val = np.log1p(df_val.msrp.values)
y_test = np.log1p(df_test.msrp.values)
```

To avoid accidentally using the target variable later, let's remove it from the dataframes:

```
del df_train['msrp']
del df_val['msrp']
del df_test['msrp']
```

> **NOTE** Removing the target variable is an optional step. But it's helpful to make sure that we don't use it when training a model: if that happens, we'd use price for predicting the price, and our model would have perfect accuracy.

When the validation split is done, we can go to the next step: training a model.

2.3 Machine learning for regression

After performing the initial data analysis, we are ready to train a model. The problem we are solving is a regression problem: the goal is to predict a number — the price of a car. For this project we will use the simplest regression model: linear regression.

2.3.1 Linear regression

To predict the price of a car, we need to use a machine learning model. To do this, we will use linear regression, which we will implement ourselves. Typically, we don't do this by hand; instead, we let a framework do this for us. In this chapter, however, we want to show that there is no magic inside these frameworks: it's just code. Linear regression is a perfect model because it's relatively simple and can be implemented with just a few lines of NumPy code.

First, let's understand how linear regression works. As we know from chapter 1, a supervised machine learning model has the form

$$y \approx g(X)$$

This is a matrix form. X is a matrix where the features of observations are rows of the matrix, and y is a vector with the values we want to predict.

These matrices and vectors may sound confusing, so let's take a step back and consider what happens with a single observation x_i and the value y_i that we want to predict. The index i here means that this is an observation number i, one of m observations that we have in our training dataset.

Then, for this single observation, the previous formula looks like

$$y_i \approx g(x_i)$$

If we have n features, our vector x_i is n-dimensional, so it has n components:

$$x_i = (x_{i1}, x_{i2}, ..., x_{in})$$

Because it has n components, we can write the function g as a function with n parameters, which is the same as the previous formula:

$$y_i = g(x_i) = g(x_{i1}, x_{i2}, ..., x_{in})$$

For our case, we have 7,150 cars in the training dataset. This means that m = 7,150, and i can be any number between 0 and 7,149. For i = 10, for example, we have the following car:

```
make                                rolls-royce
model                  phantom_drophead_coupe
year                                      2015
engine_fuel_type  premium_unleaded_(required)
engine_hp                                  453
engine_cylinders                            12
transmission_type                    automatic
driven_wheels                rear_wheel_drive
number_of_doors                             2
market_category     exotic,luxury,performance
vehicle_size                            large
vehicle_style                      convertible
highway_mpg                                19
city_mpg                                   11
popularity                                 86
msrp                                   479775
```

Let's pick a few numerical features and ignore the rest for now. We can start with horsepower, MPG in the city, and popularity:

```
engine_hp     453
city_mpg       11
popularity     86
```

Then let's assign these features to x_{i1}, x_{i2}, and x_{i3}, respectively. This way, we get the feature vector x_i with three components:

$$x_i = (x_{i1}, x_{i2}, x_{i3}) = (453, 11, 86)$$

To make it easier to understand, we can translate this mathematical notation to Python. In our case, the function g has the following signature:

```
def g(xi):
    # xi is a list with n elements
    # do something with xi
    # return the result
    pass
```

In this code, the variable `xi` is our vector x_i. Depending on implementation, `xi` could be a list with n elements or a NumPy array of size n.

For the car described previously, `xi` is a list with three elements:

```
xi = [453, 11, 86]
```

When we apply the function `g` to a vector `xi`, it produces `y_pred` as the output, which is the `g`'s prediction for `xi`:

```
y_pred = g(xi)
```

We expect this prediction to be as close as possible to y_i, which is the real price of the car.

NOTE In this section, we will use Python to illustrate the ideas behind mathematical formulas. We don't need to use these code snippets for doing the project. On the other hand, taking this code, putting it into Jupyter, and trying to run it could be helpful for understanding the concepts.

There are many ways the function g could look, and the choice of a machine learning algorithm defines the way it works.

If g is the linear regression model, it has the following form:

$$g(x_i) = g(x_{i1}, x_{i2}, ..., x_{in}) = w_0 + x_{i1}w_1 + x_{i2}w_2 + \cdots + x_{in}w_n$$

The variables w_0, w_1, w_2, …, w_n are the parameters of the model:

- w_0 is the *bias* term.
- w_1, w_2, …, w_n are the *weights* for each feature x_{i1}, x_{i2}, …, x_{in}.

These parameters define exactly how the model should combine the features so that the predictions at the end are as good as possible. It's okay if the meaning behind these parameters is not clear yet, because we will cover them later in this section.

To keep the formula shorter, let's use sum notation:

$$g(x_i) = g(x_{i1}, x_{i2}, ..., x_{in}) = w_0 + \sum_{j=1}^{n} x_{ij} w_j$$

Exercise 2.2

For supervised learning, we use a machine learning model for a single observation $y_i \approx g(x_i)$. What are x_i and y_i for this project?

a x_i is a feature vector — a vector that contains a few numbers that describe the object (a car) — and y_i is the logarithm of the price of this car.

b y_i is a feature vector — a vector that contains a few numbers that describe the object (a car) — and x_i is the logarithm of the price of this car.

These weights are what the model learns when we train it. To better understand how the model uses these weights, let's consider the following values (table 2.1).

Table 2.1 An example of weights that a linear regression model learned

w_0	w_1	w_2	w_3
7.17	0.01	0.04	0.002

So if we want to translate this model to Python, it will look like this:

```
w0 = 7.17
#   [w1    w2    w3   ]
w = [0.01, 0.04, 0.002]
n = 3

def linear_regression(xi):
    result = w0
    for j in range(n):
        result = result + xi[j] * w[j]
    return result
```

We put all the feature weights inside a single list `w` — just like we did with `xi` previously. All we need to do now is loop over these weights and multiply them by the corresponding feature values. This is nothing else but the direct translation of the previous formula to Python.

This is easy to see. Have another look at the formula:

$$w_0 + \sum_{j=1}^{n} x_{ij} w_j$$

Our example has three features, so $n = 3$, and we have

$$g(x_i) = g(x_{i1}, x_{i2}, x_{i3}) = w_0 + \sum_{j=1}^{3} x_{ij} w_j = w_0 + x_{i1} w_1 + x_{i2} w_2 + x_{i3} w_3$$

This is exactly what we have in the code

```
result = w0 + xi[0] * w[0] + xi[1] * w[1] + xi[2] * w[2]
```

with the simple exception that indexing in Python starts with 0, x_{i1} becomes `xi[0]` and w_1 is `w[0]`.

Now let's see what happens when we apply the model to our observation x_i and replace the weights with their values:

$$g(x_i) = 7.17 + 453 \cdot 0.01 + 11 \cdot 0.04 + 86 \cdot 0.002 = 12.31$$

The prediction we get for this observation is 12.31. Remember that during preprocessing, we applied the logarithmic transformation to our target variable y. This is why the model we trained on this data also predicts the logarithm of the price. To undo the transformation, we need to take the exponent of the logarithm. In our case, when we do it, the prediction becomes $603,000:

$$\exp(12.31 + 1) = 603{,}000$$

The bias term (7.17) is the value we would predict if we didn't know anything about the car; it serves as a baseline.

We do know something about the car, however: horsepower, MPG in the city, and popularity. These features are the x_{i1}, x_{i2}, and x_{i3} features, each of which tells us something about the car. We use this information to adjust the baseline.

Let's consider the first feature: horsepower. The weight for this feature is 0.01, which means that for each extra unit of horsepower, we adjust the baseline by adding 0.01. Because we have 453 horses in the engine, we add 4.53 to the baseline: 453 horses · 0.01 = 4.53.

The same happens with MPG. Each additional mile per gallon increases the price by 0.04, so we add 0.44: 11 MPG · 0.04 = 0.44.

Finally, we take popularity into account. In our example, each mention in the Twitter stream results in a 0.002 increase. In total, popularity contributes 0.172 to the final prediction.

This is exactly why we get 12.31 when we combine everything (figure 2.11).

$$g(x_i) = \underset{\text{Bias}}{7.17} + \underset{\text{Horsepower}}{453} \cdot \underset{}{0.01} + \underset{\text{MPG}}{11} \cdot \underset{}{0.04} + \underset{\text{Popularity}}{86} \cdot \underset{}{0.002} = 12.31$$

$$\underset{}{} \quad \underset{}{} \quad \underset{4.53}{} \quad \underset{0.44}{} \quad \underset{0.172}{}$$

Figure 2.11 The prediction of linear regression is the baseline of 7.17 (the bias term) adjusted by information we have from the features. Horsepower contributes 4.53 to the final prediction; MPG, 0.44; and popularity, 0.172.

Now, let's remember that we are actually dealing with vectors, not individual numbers. We know that x_i is a vector with n components:

$$x_i = (x_{i1}, x_{i2}, ..., x_{in})$$

We can also put all the weights together in a single vector w:

$$w = (w_0, w_1, w_2, ..., w_n)$$

In fact, we already did that in the Python example when we put all the weights in a list, which was a vector of dimensionality 3 with weights for each individual feature. This is how the vectors look for our example:

$$x_i = (x_{i1}, x_{i2}, x_{i3}) = (453, 11, 86)$$

$$w = (0.01, 0.04, 0.002)$$

Because we now think of both features and weights as vectors x_i and w, respectively, we can replace the sum of the elements of these vectors with a dot product between them:

$$x_i^T w = \sum_{j=1}^{n} x_{ij} w_j = x_{i1} w_1 + x_{i2} w_2 + ... + x_{in} w_n$$

The dot product is a way of multiplying two vectors: we multiply corresponding elements of the vectors and then sum the results. Refer to appendix C for more details about vector-vector multiplication.

The translation of the formula for dot product to the code is straightforward:

```
def dot(xi, w):
    n = len(w)
    result = 0.0
    for j in range(n):
```

```
        result = result + xi[j] * w[j]
    return result
```

Using the new notation, we can rewrite the entire equation for linear regression as

$$g(x_i) = w_0 + x_i^T w$$

where

- w_0 is the bias term.
- w is the n-dimensional vector of weights.

Now we can use the new `dot` function, so the linear regression function in Python becomes very short:

```
def linear_regression(xi):
    return w0 + dot(xi, w)
```

Alternatively, if `xi` and `w` are NumPy arrays, we can use the built-in `dot` method for multiplication:

```
def linear_regression(xi):
    return w0 + xi.dot(w)
```

To make it even shorter, we can combine w_0 and w into one $(n+1)$-dimensional vector by prepending w_0 to w right in front of w_1:

$$w = (w_0, w_1, w_2, ..., w_n)$$

Here, we have a new weights vector w that consists of the bias term w_0 followed by the weights w_1, w_2, ... from the original weights vector w.

In Python, this is very easy to do. If we already have the old weights in a list `w`, all we need to do is the following:

```
w = [w0] + w
```

Remember that the plus operator in Python concatenates lists, so `[1] + [2, 3, 4]` will create a new list with four elements: `[1, 2, 3, 4]`. In our case, `w` is already a list, so we create a new `w` with one extra element at the beginning: `w0`.

Because now w becomes a $(n+1)$-dimensional vector, we also need to adjust the feature vector x_i so that the dot product between them still works. We can do this easily by adding a dummy feature x_{i0}, which always takes the value 1. Then we prepend this new dummy feature to x_i right before x_{i1}:

$$x_i = (x_{i0}, x_{i1}, x_{i2}, ..., x_{in}) = (1, x_{i1}, x_{i2}, ..., x_{in})$$

Or, in code:

```
xi = [1] + xi
```

We create a new list `xi` with 1 as the first element followed by all the elements from the old list `xi`.

With these modifications, we can express the model as the dot product between the new x_i and the new w:

$$g(x_i) = x_i^T w$$

The translation to the code is simple:

```
w0 = 7.17
w = [0.01, 0.04, 0.002]
w = [w0] + w

def linear_regression(xi):
    xi = [1] + xi
    return dot(xi, w)
```

These formulas for linear regressions are equivalent because the first feature of the new x_i is 1, so when we multiply the first component of x_i by the first component of w, we get the bias term, because $w_0 \cdot 1 = w_0$.

We are ready to consider the bigger picture again and talk about the matrix form. There are many observations and x_i is one of them. Thus, we have m feature vectors x_1, x_2, …, x_i, …, x_m, and each of these vectors consists of n+1 features:

$$\begin{gathered} x_1 = (1, x_{11}, x_{12}, ..., x_{1n}) \\ x_2 = (1, x_{21}, x_{22}, ..., x_{2n}) \\ \cdots \\ x_i = (1, x_{i1}, x_{i2}, ..., x_{in}) \\ \cdots \\ x_m = (1, x_{m1}, x_{m2}, ..., x_{mn}) \end{gathered}$$

We can put these vectors together as rows of a matrix. Let's call this matrix X (figure 2.12).

$$X = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1n} \\ 1 & x_{21} & x_{22} & \cdots & x_{2n} \\ 1 & x_{31} & x_{32} & \cdots & x_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 1 & x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix} = \begin{bmatrix} \text{vector } x_1 \\ \text{vector } x_2 \\ \text{vector } x_3 \\ \cdots \\ \text{vector } x_m \end{bmatrix}$$

Figure 2.12 Matrix X, in which observations x_1, x_2, …, x_m are rows

Let's see how it looks in code. We can take a few rows from the training dataset, such as the first, second, and tenth:

```
x1  = [1, 148, 24, 1385]
x2  = [1, 132, 25, 2031]
x10 = [1, 453, 11, 86]
```

Now let's put the rows together in another list:

```
X = [x1, x2, x10]
```

List `X` now contains three lists. We can think of it as a 3x4 matrix — a matrix with three rows and four columns:

```
X = [[1, 148, 24, 1385],
     [1, 132, 25, 2031],
     [1, 453, 11, 86]]
```

Each column of this matrix is a feature:

1. The first column is a dummy feature with "1."
2. The second column is the engine horsepower.
3. The third — MPG in the city.
4. And the last one — popularity, or the number of mentions in a Twitter stream.

We already learned that to make a prediction for a single feature vector, we need to calculate the dot product between this feature vector and the weights vector. Now we have a matrix `X`, which in Python is a list of feature vectors. To make predictions for all the rows of the matrix, we can simply iterate over all rows of `X` and compute the dot product:

```
predictions = []

for xi in X:
    pred = dot(xi, w)
    predictions.append(pred)
```

In linear algebra, this is the matrix-vector multiplication: we multiply the matrix X by the vector w. The formula for linear regression becomes

$$g(X) = w_0 + Xw$$

The result is an array with predictions for each row of X. Refer to appendix C for more details about matrix-vector multiplication.

With this matrix formulation, the code for applying linear regression to make predictions becomes very simple. The translation to NumPy becomes straightforward:

```
predictions = X.dot(w)
```

Exercise 2.3

When we multiply the matrix X by the weights vector w, we get

a A vector y with the actual price
b A vector y with price predictions
c A single number y with price predictions

2.3.2 Training linear regression model

So far, we've only covered making predictions. To be able to do that, we need to know the weights w. How do we get them?

We learn the weights from data: we use the target variable y to find such w that combines the features of X in the best possible way. "Best possible" in the case of linear regression means that it minimizes the error between the predictions $g(X)$ and the actual target y.

We have multiple ways to do that. We will use normal equation, which is the simplest method to implement. The weight vector w can be computed with the following formula:

$$w = (X^T X)^{-1} X^T y$$

NOTE Covering the derivation of the normal equation is out of scope for this book. We give a bit of intuition of how it works in appendix C, but you should consult a machine learning textbook for a more in-depth introduction. *The Elements of Statistical Learning*, 2nd edition by Hastie, Tibshirani, and Friedman is a good start.

This piece of math may appear scary or confusing, but it's quite easy to translate to NumPy:

- X^T is the transpose of X. In NumPy, it's `X.T`.
- $X^T X$ is a matrix–matrix multiplication, which we can do with the `dot` method from NumPy: `X.T.dot(X)`.
- X^{-1} is the inverse of X. We can use `np.linalg.inv` function to calculate the inverse.

So the formula above translates directly to

```
inv(X.T.dot(X)).dot(X.T).dot(y)
```

Please refer to appendix C for more details about this equation.

To implement the normal equation, we need to do the following:

1 Create a function that takes in a matrix X with features and a vector y with the target.

2 Add a dummy column (the feature that is always set to 1) to the matrix X.
3 Train the model: compute the weights w by using the normal equation.
4 Split this w into the bias w_0 and the rest of the weights, and return them.

The last step — splitting w into the bias term and the rest — is optional and mostly for convenience; otherwise, we need to add the dummy column every time we want to make predictions instead of doing it once during training.

Let's implement it.

Listing 2.2 Linear regression implemented with NumPy

```
def train_linear_regression(X, y):
    # adding the dummy column
    ones = np.ones(X.shape[0])
    X = np.column_stack([ones, X])

    # normal equation formula
    XTX = X.T.dot(X)
    XTX_inv = np.linalg.inv(XTX)
    w = XTX_inv.dot(X.T).dot(y)

    return w[0], w[1:]
```

1 Creates an array that contains only ones
2 Adds the array of Is as the first column of X
3 Computes X^TX
4 Computes the inverse of X^TX
5 Computes the rest of the normal equation
6 Splits the weights vector into the bias and the rest of the weights

With six lines of code, we have implemented our first machine learning algorithm. In 1, we create a vector containing only ones, which we append to the matrix X as the first column; this is the dummy feature in 2. Next, we compute X^TX in 3 and its inverse in 4, and we put them together to calculate w in 5. Finally, we split the weights into the bias w_0 and the remaining weights w in 6.

The `column_stack` function from NumPy that we used for adding a column of ones might be confusing at first, so let's have a closer look at it:

```
np.column_stack([ones, X])
```

It takes in a list of NumPy arrays, which in our case contains `ones` and `X` and stacks them (figure 2.13).

If weights are split into the bias term and the rest, the linear regression formula for making predictions changes slightly:

$$g(X) = w_0 + Xw$$

This is still very easy to translate to NumPy:

```
y_pred = w0 + X.dot(w)
```

Let's use it for our project!

```
ones = np.array([1, 1])
ones
```

```
array([1, 1])
```

```
X = np.array([[2, 3], [4, 5]])
X
```

```
array([[2, 3],
       [4, 5]])
```

```
np.column_stack([ones, X])
```

```
array([[1, 2, 3],
       [1, 4, 5]])
```

Ones X

Figure 2.13 The function `column_stack` takes a list of NumPy arrays and stacks them in columns. In our case, the function appends the array with ones as the first column of the matrix.

2.4 Predicting the price

We've covered a great deal of theory, so let's come back to our project: predicting the price of a car. We now have a function for training a linear regression model at our disposal, so let's use it to build a simple baseline solution.

2.4.1 Baseline solution

To be able to use it, however, we need to have some data: a matrix X and a vector with the target variable y. We have already prepared the y, but we still don't have the X: what we have right now is a data frame, not a matrix. So we need to extract some features from our dataset to create this matrix X.

We will start with a very naive way of creating features: select a few numerical features, and form the matrix X from them. In the previous example, we used only three features. This time, we include a couple more features and use the following columns:

- engine_hp
- engine_cylinders
- highway_mpg
- city_mpg
- popularity

Let's select the features from the data frame and write them to a new variable, `df_num`:

```
base = ['engine_hp', 'engine_cylinders', 'highway_mpg', 'city_mpg',
        'popularity']
df_num = df_train[base]
```

As discussed in the section on exploratory data analysis, the dataset has missing values. We need to do something because the linear regression model cannot deal with missing values automatically.

One option is to drop all the rows that contain at least one missing value. This approach, however, has some disadvantages. Most important, we will lose the information that we have in the other columns. Even though we may not know the number of doors of a car, we still know other things about the car, such as make, model, age, and other things that we don't want to throw away.

The other option is filling the missing values with some other value. This way, we don't lose the information in other columns and still can make predictions, even if the row has missing values. The simplest possible approach is to fill the missing values with zeros. We can use the `fillna` method from Pandas:

```
df_num = df_num.fillna(0)
```

This method may not be the best way to deal with missing values, but often, it's good enough. If we set the missing feature value to zero, the respective feature is simply ignored.

> **NOTE** An alternative option is to replace the missing values with the average values. For some variables, for example, the number of cylinders, the value of zero doesn't make much sense: a car cannot have zero cylinders. However, this will make our code more complex and won't have a significant impact on the results. That's why we follow a simpler approach and replace the missing values with zeros.

It's not difficult to see why setting a feature to zero is the same as ignoring it. Let's recall the formula for linear regression. In our case, we have five features, so the formula is

$$g(x_i) = w_0 + x_{i1}w_1 + x_{i2}w_2 + x_{i3}w_3 + x_{i4}w_4 + x_{i5}w_5$$

If feature three is missing, and we fill it with zero, x_{i3} becomes zero:

$$g(x_i) = w_0 + x_{i1}w_1 + x_{i2}w_2 + 0\ w_3 + x_{i4}w_4 + x_{i5}w_5$$

In this case, regardless of the weight w_3 for this feature, the product $x_{i3}w_3$ will always be zero. In other words, this feature will have no contribution to the final prediction, and we will base our prediction only on features that aren't missing:

$$g(x_i) = w_0 + x_{i1}w_1 + x_{i2}w_2 + x_{i4}w_4 + x_{i5}w_5$$

Now we need to convert this DataFrame to a NumPy array. The easiest way to do it is to use its `values` property:

```
X_train = df_num.values
```

`X_train` is a matrix — a two-dimensional NumPy array. It's something we can use as input to our `linear_regresson` function. Let's call it

```
w_0, w = train_linear_regression(X_train, y_train)
```

We have just trained the first model! Now we can apply it to the training data to see how well it predicts:

```
y_pred = w_0 + X_train.dot(w)
```

To see how good the predictions are, we can use `histplot` — a function from Seaborn for plotting histograms that we used previously — to plot the predicted values and compare them with the actual prices:

```
sns.histplot(y_pred, label='prediction')
sns.histplot(y_train, label='target')
plt.legend()
```

We can see from the plot (figure 2.14) that the distribution of values we predicted looks quite different from the actual values. This result may indicate that the model is not powerful enough to capture the distribution of the target variable. This shouldn't be a surprise to us: the model we used is quite basic and includes only five very simple features.

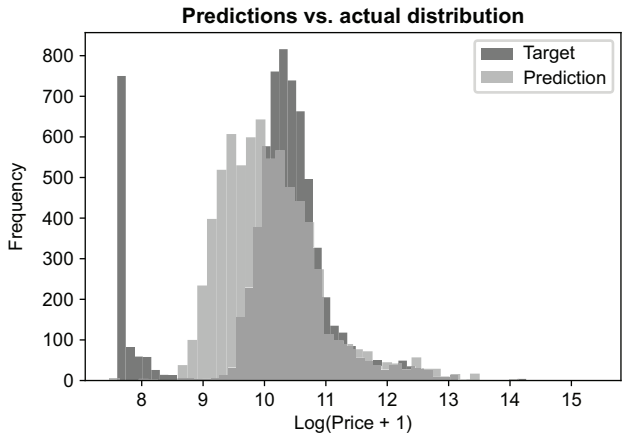


Figure 2.14 The distribution of the predicted values (light gray) and the actual values (dark gray). We see that our predictions aren't very good; they are very different from the actual distribution.

2.4.2 RMSE: Evaluating model quality

Looking at plots and comparing the distributions of the actual target variable with the predictions is a good way to evaluate quality, but we cannot do this every time we change something in the model. Instead, we need to use a metric that quantifies the quality of the model. We can use many metrics to evaluate how well a regression model behaves. The most commonly used one is *root mean squared error* — RMSE for short.

RMSE tells us how large the errors are that our model makes. It's computed with the following formula:

$$\text{RMSE} = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\Big(g(x_i) - y_i\Big)^2}$$

Let's try to understand what's going on here. First, let's look inside the sum. We have

$$\Big(g(x_i) - y_i\Big)^2$$

This is the difference between the prediction we make for the observation and the actual target value for that observation (figure 2.15).

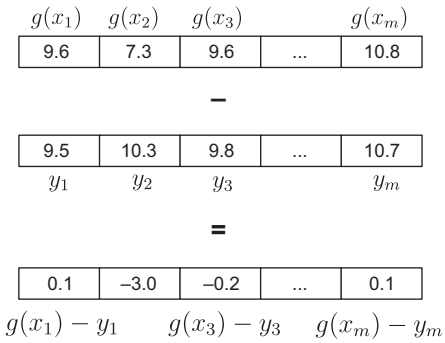


Figure 2.15 The difference between the predictions $g(x_i)$ and the actual values y_i

Then we use the square of the difference, which gives a lot more weight to larger differences. If we predict 9.5, for example, and the actual value is 9.6, the difference is 0.1, so its square is 0.01, which is quite small. But if we predict 7.3, and the actual value is 10.3, the difference is 3, and the square of the difference is 9 (figure 2.16).

This is the SE part (*squared error*) of RMSE.

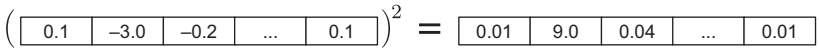


Figure 2.16 The square of the difference between the predictions and the actual values. For large differences, the square is quite big.

Next, we have a sum:

$$\sum_{i=1}^{m}\Big(g(x_i)-y_i\Big)^2$$

This summation goes over all m observations and puts all the squared errors together (figure 2.17) into a single number.

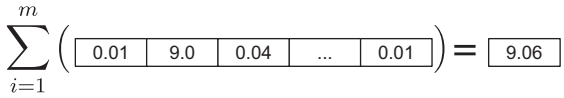


Figure 2.17 The result of the summation of all the square differences is a single number.

If we divide this sum by m, we get the mean squared error:

$$\frac{1}{m}\sum_{i=1}^{m}\Big(g(x_i)-y_i\Big)^2$$

This is the squared error that our model makes on average — the M part (*mean*) of RMSE, or *mean squared error* (MSE). MSE is also a good metric on its own (figure 2.18).

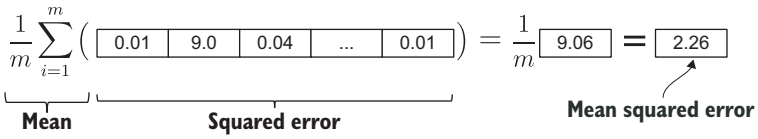


Figure 2.18 MSE is computed by calculating the mean of the squared errors.

Finally, we take the square root of that:

$$\mathrm{RMSE}=\sqrt{\frac{1}{m}\sum_{i=1}^{m}\Big(g(x_i)-y_i\Big)^2}$$

This is the R part (*root*) of RMSE (figure 2.19).

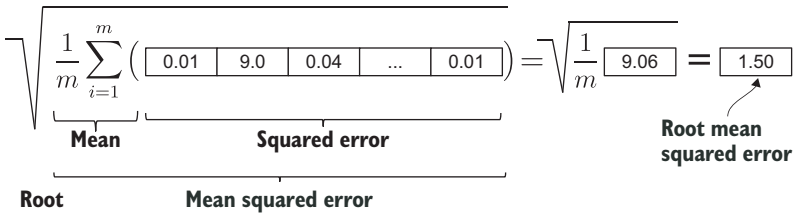


Figure 2.19 RMSE: we first compute MSE and then calculate its square root.

When using NumPy to implement RMSE, we can take advantage of *vectorization*: the process of applying the same operation to all elements of one or more NumPy arrays. We get multiple benefits from using vectorization. First, the code is more concise: we don't have to write any loops to apply the same operation to each element of the array. Second, vectorized operations are a lot faster than simple Python for loops.

Consider the following implementation.

Listing 2.3 The implementation of root mean squared error

```
def rmse(y, y_pred):
    error = y_pred - y
    mse = (error ** 2).mean()
    return np.sqrt(mse)
```

1 Computes the difference between the prediction and the

2 Computes MSE: first computes the squared error, and then calculates its mean

3 Takes the square root to get RMSE

In 1, we compute element-wise difference between the vector with predictions and the vector with the target variable. The result is a new NumPy array `error` that contains the differences. In 2, we do two operations in one line: compute the square of each element of the `error` array and then get the mean value of the result, which gives us MSE. In 3, we compute the square root to get RMSE.

Element-wise operations in NumPy and Pandas are quite convenient. We can apply an operation to an entire NumPy array (or a Pandas series) without writing loops.

In the first line of our `rmse` function, for example, we compute the difference between the predictions and the actual prices:

```
error = y_pred - y
```

What happens here is that for each element of `y_pred`, we subtract the corresponding element of `y` and then put the result to the new array `error` (figure 2.20).

Next, we compute the square of each element of the `error` array and then calculate its mean to get the mean squared error of our model (figure 2.21).

To see exactly what happens, we need to know that the power operator (**) is also applied element-wise, so the result is another array in which all elements of the

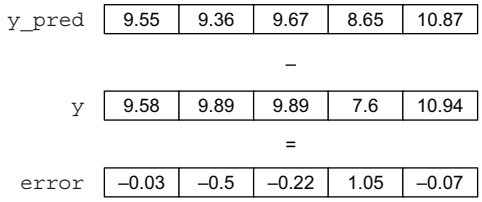


Figure 2.20 The element-wise difference between `y_pred` and `y`. The result is written to the `error` array.

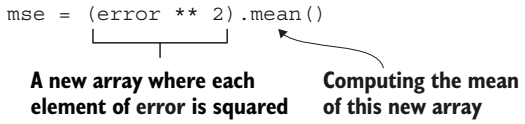


Figure 2.21 To calculate MSE, we first compute the square of each element in the error array and then compute the mean value of the result.

original array are squared. When we have this new array with squared elements, we simply compute its mean by using the `mean()` method (figure 2.22).

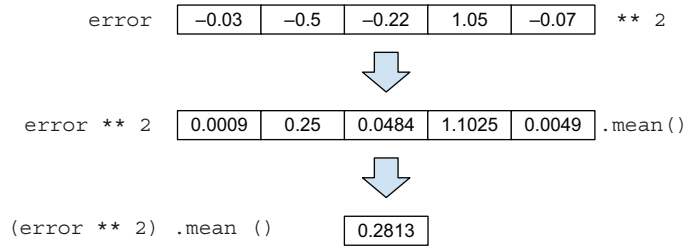


Figure 2.22 The power operator (`**`) applied element-wise to the error array. The result is another array in which each element is squared. Then we compute the mean of the array with the squared error to compute MSE.

Finally, we compute the square root of the mean value to get RMSE:

```
np.sqrt(mse)
```

Now we can use RMSE to evaluate the quality of the model:

```
rmse(y_train, y_pred)
```

The code prints 0.75. This number tells us that on average, the model's predictions are off by 0.75. This result alone may not be very useful, but we can use it to compare this model with other models. If one model has a better (lower) RMSE than the other, it indicates that model is better.

2.4.3 Validating the model

In the example from the previous section we computed RMSE on the training set. The result is useful to know but doesn't reflect the way the model will be used later. The model will be used to predict the price of cars that it didn't see before. For that purpose, we set aside a validation dataset. We intentionally don't use it for training and keep it for validating the model.

We have already split our data into multiple parts: `df_train`, `df_val`, and `df_test`. We have also created a matrix `X_train` from `df_train` and used `X_train` and `y_train` to train the model. Now we need to do the same steps to get `X_val` — a matrix with features computed from the validation dataset. Then we can apply the model to `X_val` to get predictions and compare them with `y_val`.

First, we create the `X_val` matrix, following the same steps as for `X_train`:

```
df_num = df_val[base]
df_num = df_num.fillna(0)
X_val = df_num.values
```

We're ready to apply the model to `X_val` to get predictions:

```
y_pred = w_0 + X_val.dot(w)
```

The `y_pred` array contains the predictions for the validation dataset. Now we use `y_pred` and compare it with the actual prices from `y_val`, using the RMSE function that we implemented previously:

```
rmse(y_val, y_pred)
```

The value this code prints is 0.76, which is the number we should use for comparing models.

In the previous code we already see some duplication: training and validation tests require the same preprocessing, and we wrote the same code twice. Thus, it makes sense to move this logic to a separate function and avoid duplicating the code.

We can call this function `prepare_X` because it creates a matrix `X` from a DataFrame.

Listing 2.4 The `prepare_X` function for converting a DataFrame into a matrix

```
def prepare_X(df):
    df_num = df[base]
    df_num = df_num.fillna(0)
    X = df_num.values
    return X
```

Now the whole training and evaluation becomes simpler and looks like this:

```
X_train = prepare_X(df_train)
w_0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))
```

Trains the model

Applies the model to the validation dataset

Computes RMSE on the validation data

This gives us a way to check whether any model adjustments lead to improvements in the predictive quality of the model. As the next step, let's add more features and check whether it gets lower RMSE scores.

2.4.4 Simple feature engineering

We already have a simple baseline model with simple features. To improve our model further, we can add more features to the model: we create others and add them to the existing features. As we already know, this process is called *feature engineering*.

Because we have already set up the validation framework, we can easily verify whether adding new features improves the quality of the model. Our aim is to improve the RMSE calculated on the validation data.

First, we create a new feature, "age," from the feature "year." The age of a car should be very helpful when predicting its price: intuitively, the newer the car, the more expensive it should be.

Because the dataset was created in 2017 (which we can verify by checking `df_train.year.max()`), we can calculate the age by subtracting the year when the car was made from 2017:

```
df_train['age'] = 2017 - df_train.year
```

This operation is an element-wise operation. We calculate the difference between 2017 and each element of the year series. The result is a new Pandas series containing the differences, which we write back to the dataframe as the age column.

We already know that we will need to apply the same preprocessing twice: to the training and validation sets. Because we don't want to repeat the feature extraction code multiple times, let's put this logic into the `prepare_X` function.

Listing 2.5 Creating the age feature in the `prepare_X` function

```
def prepare_X(df):
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age')

    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values
    return X
```

1. Creates a copy of the input parameter to prevent side effects
2. Creates a copy of the base list with the basic features
3. Computes the age feature
4. Appends age to the list of feature names we use for the model

The way we implement the function this time is slightly different from the previous version. Let's look at these differences. First, in ❶, we create a copy of the DataFrame `df` that we pass in the function. Later in the code, we modify `df` by adding extra rows in ❸. This kind of behavior is known as a *side effect*: the caller of the function may not expect the function to change the DataFrame. To prevent the unpleasant surprise, we instead modify the copy of the original DataFrame. In ❷, we create a copy for the list with the base features for the same reason. Later, we extend this list with new features ❹, but we don't want to change the original list. The rest of the code is the same as previously.

Let's test if adding the feature "age" leads to any improvements:

```
X_train = prepare_X(df_train)
w_0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))
```

The code prints

```
validation: 0.517
```

The validation error is 0.517, which is a good improvement from 0.76 — the value we had in the baseline solution. Thus, we conclude that adding "age" is indeed helpful when making predictions.

We can also look at the distribution of the predicted values:

```
sns.histplot(y_pred, label='prediction')
sns.histplot(y_val, label='target')
plt.legend()
```

We see (figure 2.23) that the distribution of the predictions follows the target distribution a lot more closely than previously. Indeed, the validation RMSE score confirms it.

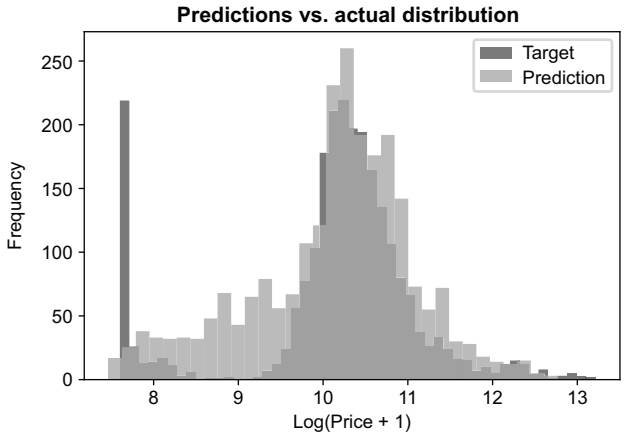


Figure 2.23 The distribution of predicted (light gray) versus actual (dark gray). With the new features, the model follows the original distribution closer than previously.

2.4.5 Handling categorical variables

We see that adding "age" is quite helpful for the model. Let's continue adding more features. One of the columns we can use next is the number of doors. This variable appears to be numeric and can take three values: 2, 3, and 4 doors. Even though it's tempting to put the variable to the model as is, it's not really a numeric variable: we cannot say that by adding one more door, the price of a car grows (or drops) by a certain amount of money. Rather, the variable is categorical.

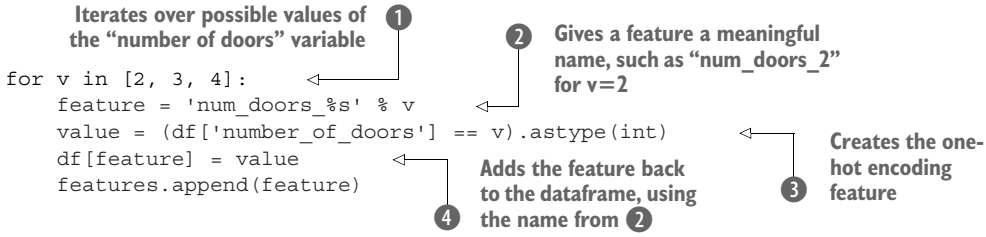
Categorical variables describe characteristics of objects and can take one of a few possible values. The make of a car is a categorical variable; for example, it can be Toyota, BWM, Ford, or any other make. It's easy to recognize a categorical variable by its values, which typically are strings and not numbers. That's not always the case, however. The number of doors, for example, is categorical: it can take only one of the three possible values (2, 3, and 4).

We can use categorical variables in a machine learning model in multiple ways. One of the simplest ways is to encode such variables by a set of binary features, with a separate feature for each distinct value.

In our case, we will create three binary features: `num_doors_2`, `num_doors_3`, and `num_doors_4`. If the car has two doors, `num_doors_2` will be set to 1, and the rest will be 0. If the car has three doors, `num_doors_3` will get the value 1, and the same goes for `num_doors_4`.

This method of encoding categorical variables is called *one-hot encoding*. We will learn more about this way of encoding categorical variables in chapter 3. For now, let's choose the simplest way to do this encoding: looping over the possible values (2, 3, and 4) and, for each value, checking whether the value of the observation matches it.

Let's add these lines to the `prepare_X` function:



This code may be difficult to understand, so let's take a closer look at what's going on here. The most difficult line is 3:

```
(df['number_of_doors'] == v).astype(int)
```

Two things happen here. The first one is the expression inside the parentheses, where we use the equals (==) operator. This operation is also an element-wise operation, like the ones we used previously when computing RMSE. In this case, the operation creates a new Pandas series. If elements of the original series equal `v`, the corresponding elements in the result is True; otherwise, the elements are False. The operation creates a

series of True/False values. Because v has three values (2, 3, and 4), and we apply this operation to every value of v, we create three series (figure 2.24).

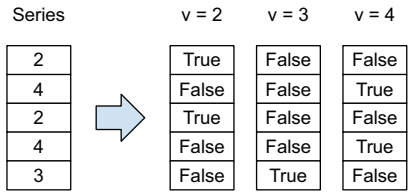


Figure 2.24 We use the == operator to create the new series from the original one: one for two doors, one for three doors, and one for four doors.

Next, we convert the Boolean series to integers in such a way that True becomes 1 and False becomes 0, which is easy to do with the `astype(int)` method (figure 2.25). Now we can use the results as features and put them into linear regression.

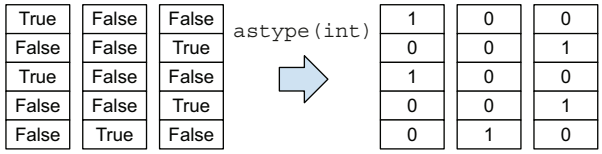


Figure 2.25 Using `astype(int)` to convert series with Boolean values to integers

The number of doors, as we discussed, is a categorical variable that appears to be numerical because the values are integers (2, 3 and 4). All the remaining categorical variables we have in the dataset are strings.

We can use the same approach to encode other categorical variables. Let's start with `make`. For our purposes, it should be enough to get and use only the most frequently occurring values. Let's find out what the five most frequent values are:

```
df['make'].value_counts().head(5)
```

The code prints

```
chevrolet     1123
ford           881
volkswagen     809
toyota         746
dodge          626
```

We take these values and use them to encode `make` in the same way that we encoded the number of doors.

Next, we create five new variables called `is_make_chevrolet`, `is_make_ford`, `is_make_volkswagen`, `is_make_toyota`, and `is_make_dodge`:

```
for v in ['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge']:
    feature = 'is_make_%s' % v
    df[feature] = (df['make'] == v).astype(int)
    features.append(feature)
```

Now the whole `prepare_X` should look like the following.

Listing 2.6 Handling categorical variables `number of doors` and `make`

```
def prepare_X(df):
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age')

    for v in [2, 3, 4]:
        feature = 'num_doors_%s' % v
        df[feature] = (df['number_of_doors'] == v).astype(int)
        features.append(feature)

    for v in ['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge']:
        feature = 'is_make_%s' % v
        df[feature] = (df['make'] == v).astype(int)
        features.append(feature)

    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values
    return X
```

Encodes the number of doors variable

Encodes the make variable

Let's check whether this code improves the RMSE of the model:

```
X_train = prepare_X(df_train)
w_0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))
```

The code prints

```
validation: 0.507
```

The previous value was 0.517, so we managed to improve the RMSE score further.

We can use a few more variables: `engine_fuel_type`, `transmission_type`, `driven_wheels`, `market_category`, `vehicle_size`, and `vehicle_style`. Let's do the same thing for them. After the modifications, the `prepare_X` starts looking a bit more complex.

Listing 2.7 Handling more categorical variables in the `prepare_X` function

```
def prepare_X(df):
    df = df.copy()
    features = base.copy()

    df['age'] = 2017 - df.year
    features.append('age')

    for v in [2, 3, 4]:
        feature = 'num_doors_%s' % v
        df[feature] = (df['number_of_doors'] == v).astype(int)
        features.append(feature)

    for v in ['chevrolet', 'ford', 'volkswagen', 'toyota', 'dodge']:
        feature = 'is_make_%s' % v
        df[feature] = (df['make'] == v).astype(int)
        features.append(feature)

    for v in ['regular_unleaded', 'premium_unleaded_(required)',
              'premium_unleaded_(recommended)',
              'flex-fuel_(unleaded/e85)']:
        feature = 'is_type_%s' % v
        df[feature] = (df['engine_fuel_type'] == v).astype(int)
        features.append(feature)

    for v in ['automatic', 'manual', 'automated_manual']:
        feature = 'is_transmission_%s' % v
        df[feature] = (df['transmission_type'] == v).astype(int)
        features.append(feature)

    for v in ['front_wheel_drive', 'rear_wheel_drive',
              'all_wheel_drive', 'four_wheel_drive']:
        feature = 'is_driven_wheels_%s' % v
        df[feature] = (df['driven_wheels'] == v).astype(int)
        features.append(feature)

    for v in ['crossover', 'flex_fuel', 'luxury',
              'luxury,performance', 'hatchback']:
        feature = 'is_mc_%s' % v
        df[feature] = (df['market_category'] == v).astype(int)
        features.append(feature)

    for v in ['compact', 'midsize', 'large']:
        feature = 'is_size_%s' % v
        df[feature] = (df['vehicle_size'] == v).astype(int)
        features.append(feature)

    for v in ['sedan', '4dr_suv', 'coupe', 'convertible',
              '4dr_hatchback']:
        feature = 'is_style_%s' % v
        df[feature] = (df['vehicle_style'] == v).astype(int)
        features.append(feature)
```

Encodes the type variable

Encodes the transmission variable

Encodes the number of driven wheels

Encodes the market category

Encodes the size

Encodes the style

```
    df_num = df[features]
    df_num = df_num.fillna(0)
    X = df_num.values
    return X
```

Let's test it:

```
X_train = prepare_X(df_train)
w_0, w = train_linear_regression(X_train, y_train)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))
```

The number we see is significantly worse than before. We get 34.2, which is a lot more than the 0.5 we had before.

> **NOTE** The number you get may be different, depending on the Python version, the NumPy version, the versions of NumPy dependencies, the OS, and other factors. But the jump in the validation metric from 0.5 to something significantly bigger should always alert us.

Instead of helping, the new features made the score a lot worse. Luckily, we have validation to help us spot this problem. In the next section, we will see why it happens and how to deal with it.

2.4.6 Regularization

We saw that adding new features does not always help, and in our case, it made things a lot worse. The reason for this behavior is numerical instability. Recall the formula of the normal equation:

$$w = (X^T X)^{-1} X^T y$$

One of the terms in the equation is the inverse of the $X^T X$ matrix:

$$(X^T X)^{-1}$$

The inversion is the issue in our case. Sometimes, when adding new columns to X, we can accidentally add a column that is a combination of other columns. For example, if we already have the MPG in the city feature and decide to add kilometers per liter in the city, the second feature is the same as the first one but multiplied by a constant.

When this happens, $X^T X$ becomes *undetermined* or *singular*, which means that it's not possible to find an inverse for this matrix. If we try to invert a singular matrix, NumPy will tell us about that by raising a `LinAlgError`:

```
LinAlgError: Singular matrix
```

Our code didn't raise any exceptions, however. It happened because we don't typically have columns that are perfect linear combinations of other columns. The real data is often noisy, with measurement errors (such as recording 1.3 instead of 13 for MPG), rounding errors (such as storing 0.0999999 instead of 0.1), and many other errors. Technically, such matrices are not singular, so NumPy doesn't complain.

For this reason, however, some of the values in the weights become extremely large — a lot larger than they are supposed to be.

If we look at the values of our w_0 and w, we see that this is indeed the case. The bias term w_0 has the value 5788519290303866.0, for example (the value may vary depending on the machine, OS, and version of NumPy), and a few components of w have extremely large negative values as well.

In numerical linear algebra, such issues are called *numerical instability issues*, and they are typically solved with regularization techniques. The aim of *regularization* is to make sure that the inverse exists by forcing the matrix to be invertible. Regularization is an important concept in machine learning: it means "controlling" — controlling the weights of the model so that they behave correctly and don't grow too large, as in our case.

One way to do regularization is to add a small number to each diagonal element of the matrix. Then we get the following formula for linear regression:

$$w = (X^T X + \alpha I)^{-1} X^T y$$

NOTE Regularized linear regression is often called *ridge regression*. Many libraries, including Scikit-learn, use *ridge* to refer to regularized linear regression and *linear regression* to refer to the unregularized model.

Let's look at the part that changed: the matrix that we need to invert. This is how it looks:

$$X^T X + \alpha I$$

This formula says that we need I — an *identity matrix*, which is a matrix with ones on the main diagonal and zeros everywhere else. We multiply this identity matrix by a number α. This way, all the ones on the diagonal of I become α. Then we sum αI and $X^T X$, which adds α to all the diagonal elements of $X^T X$.

This formula can directly translate to NumPy code:

```
XTX = X_train.T.dot(X_train)
XTX = XTX + 0.01 * np.eye(XTX.shape[0])
```

The `np.eye` function creates a two-dimensional NumPy array that is also an identity matrix. When we multiply by 0.01, the ones on the diagonal become 0.01, so when we add this matrix to `XTX`, we add only 0.01 to its main diagonal (figure 2.26).

```
np.eye(4)
```

```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

(A) The eye function from NumPy creates an identity matrix.

```
np.eye(4) * 0.01
```

```
array([[0.01, 0.  , 0.  , 0.  ],
       [0.  , 0.01, 0.  , 0.  ],
       [0.  , 0.  , 0.01, 0.  ],
       [0.  , 0.  , 0.  , 0.01]])
```

(B) When we multiply the identity matrix by a number, this number goes to the main diagonal of the result.

```
XTX = np.array([
    [0, 1, 2, 3],
    [0, 1, 2, 3],
    [0, 1, 2, 3],
    [0, 1, 2, 3],
])

XTX + 0.01 * np.eye(4)
```

```
array([[0.01, 1.  , 2.  , 3.  ],
       [0.  , 1.01, 2.  , 3.  ],
       [0.  , 1.  , 2.01, 3.  ],
       [0.  , 1.  , 2.  , 3.01]])
```

(C) The effect of adding an identity matrix multiplied by 0.01 to another matrix is the same as adding 0.01 to the main diagonal of that matrix.

Figure 2.26 Using an identity matrix to add 0.01 to the main diagonal of a square matrix

Let's create a new function that uses this idea and implements linear regression with regularization.

Listing 2.8 Linear regression with regularization

```
def train_linear_regression_reg(X, y, r=0.0):
    ones = np.ones(X.shape[0])
    X = np.column_stack([ones, X])

    XTX = X.T.dot(X)
    reg = r * np.eye(XTX.shape[0])
    XTX = XTX + reg

    XTX_inv = np.linalg.inv(XTX)
    w = XTX_inv.dot(X.T).dot(y)

    return w[0], w[1:]
```

Controls the amount of regularization by using the parameter *r*

Adds *r* to the main diagonal of XTX

The function is very similar to linear regression, but a few lines are different. First, there's an extra parameter `r` that controls the amount of regularization — this corresponds to the number α in the formula that we add to the main diagonal of X^TX.

Regularization affects the final solution by making the components of w smaller. We can see that the more regularization we add, the smaller the weights become.

Let's check what happens with our weights for different values of `r`:

```
for r in [0, 0.001, 0.01, 0.1, 1, 10]:
    w_0, w = train_linear_regression_reg(X_train, y_train, r=r)
    print('%5s, %.2f, %.2f, %.2f' % (r, w_0, w[13], w[21]))
```

The code prints

```
    0, 5788519290303866.00, -9.26, -5788519290303548.00
0.001, 7.20, -0.10, 1.81
 0.01, 7.18, -0.10, 1.81
  0.1, 7.05, -0.10, 1.78
    1, 6.22, -0.10, 1.56
   10, 4.39, -0.09, 1.08
```

We start with 0, which is an unregularized solution, and get very large numbers. Then we try 0.001 and increase it 10 times on each step: 0.01, 0.1, 1, and 10. We see that the values that we selected become smaller as `r` grows.

Now let's check whether regularization helps with our problem and what RMSE we get after that. Let's run it with r=0.001:

```
X_train = prepare_X(df_train)
w_0, w = train_linear_regression_reg(X_train, y_train, r=0.001)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))
```

The code prints

```
Validation: 0.460
```

This result is an improvement over the previous score: 0.507.

> **NOTE** Sometimes, when adding a new feature causes performance degradation, simply removing this feature may be enough to solve the problem. Having a validation dataset is important to decide whether to add regularization, remove the feature, or do both: we use the score on the validation data to choose the best option. In our particular case, we see that adding regularization helps: it improves the score we had previously.

We tried using `r`=0.001, but we should try other values as well. Let's try a couple of different ones to select the best parameter `r`:

```
X_train = prepare_X(df_train)
X_val = prepare_X(df_val)

for r in [0.000001, 0.0001, 0.001, 0.01, 0.1, 1, 5, 10]:
    w_0, w = train_linear_regression_reg(X_train, y_train, r=r)
    y_pred = w_0 + X_val.dot(w)
    print('%6s' %r, rmse(y_val, y_pred))
```

We see that the best performance is achieved with a smaller r:

```
 1e-06 0.460225
0.0001 0.460225
 0.001 0.460226
  0.01 0.460239
   0.1 0.460370
     1 0.461829
     5 0.468407
    10 0.475724
```

We also notice that the performance for values below 0.1 don't change much except in the sixth digit, which we shouldn't consider to be significant.

Let's take the model with r=0.01 as the final model. Now we can check it against the test dataset to verify if the model works:

```
X_train = prepare_X(df_train)
w_0, w = train_linear_regression_reg(X_train, y_train, r=0.01)

X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))

X_test = prepare_X(df_test)
y_pred = w_0 + X_test.dot(w)
print('test:', rmse(y_test, y_pred))
```

The code prints

```
validation: 0.460
test: 0.457
```

Because these two numbers are pretty close, we conclude that the model can generalize well to the new unseen data.

Exercise 2.4

Regularization is needed because

- a It can control the weights of the model and not let them grow too large.
- b Real-world data is noisy.
- c We often have numerical instability problems.

Multiple answers are possible.

2.4.7 Using the model

Because we now have a model, we can start using it for predicting the price of a car.

Suppose that a user posts the following ad on our website:

```
ad = {
    'city_mpg': 18,
```

```
    'driven_wheels': 'all_wheel_drive',
    'engine_cylinders': 6.0,
    'engine_fuel_type': 'regular_unleaded',
    'engine_hp': 268.0,
    'highway_mpg': 25,
    'make': 'toyota',
    'market_category': 'crossover,performance',
    'model': 'venza',
    'number_of_doors': 4.0,
    'popularity': 2031,
    'transmission_type': 'automatic',
    'vehicle_size': 'midsize',
    'vehicle_style': 'wagon',
    'year': 2013
}
```

We'd like to suggest the price for this car. For that, we use our model:

```
df_test = pd.DataFrame([ad])
X_test = prepare_X(df_test)
```

First, we create a small DataFrame with one row. This row contains all the values of the `ad` dictionary we created earlier. Next, we convert this DataFrame to a matrix.

Now we can apply our model to the matrix to predict the price of this car:

```
y_pred = w_0 + X_test.dot(w)
```

This prediction is not the final price, however; it's the logarithm of the price. To get the actual price, we need to undo the logarithm and apply the exponent function:

```
suggestion = np.expm1(y_pred)
suggestion
```

The output is 28,294.13. The real price of this car is $31,120, so our model is not far from the actual price.

2.5 Next steps

2.5.1 Exercises

You can try the following things to make the model better:

- *Write a function for binary encoding.* In this chapter we implemented the category encoding manually: we looked at the top five values, wrote them in a list, and then looped over the list to create binary features. Doing it this way is cumbersome, which is why it's a good idea to write a function that will do this automatically. It should have multiple arguments: the dataframe, the name of the categorical variable, and the number of most frequent values it should consider. This function should also help us do the previous exercise.

- *Try more feature engineering.* When implementing category encoding, we included only the top five values for each categorical variable. Including more values during the encoding process might improve the model. Try doing that, and reevaluate the quality of the model in terms of RMSE.

2.5.2 Other projects

There are other projects you can do now:

- Predict the price of a house. You can take the New York City Airbnb Open Data dataset from https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data or the California housing dataset from https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_california_housing.html.
- Check other datasets, such as https://archive.ics.uci.edu/ml/datasets.php?task=reg, that have numerical target values. For example, we can use the data from the student performance dataset (http://archive.ics.uci.edu/ml/datasets/Student+Performance) to train a model for determining the performance of students.

Summary

- Doing simple initial exploratory analysis is important. Among other things, it helps us find out whether the data has missing values. It's not possible to train a linear regression model when there are missing values, so it's important to check our data and fill in the missing values if necessary.
- As a part of exploratory data analysis, we need to check the distribution of the target variable. If the target distribution has a long tail, we need to apply the log transformation. Without it, we may get inaccurate and misleading predictions from the linear regression model.
- The train/validation/test split is the best way to check our models. It gives us a way to measure the performance of the model reliably, and things like numerical instability issues won't go unnoticed.
- The linear regression model is based on a simple mathematical formula, and understanding this formula is the key to successful application of the model. Knowing these details helps us learn how the model works before coding it.
- It's not difficult to implement linear regression from scratch using Python and NumPy. Doing so helps us understand that there's no magic behind machine learning: it's simple math translated to code.
- RMSE gives us a way to measure the predictive performance of our model on the validation set. It lets us confirm that the model is good and helps us compare multiple models to find the best one.
- Feature engineering is the process of creating new features. Adding new features is important for improving the performance of a model. While adding new features, we always need to use the validation set to make sure that our

model indeed improves. Without constant monitoring, we risk getting mediocre or very bad performance.

- Sometimes, we face numerical instability issues that we can solve with regularization. Having a good way to validate models is crucial for spotting a problem before it's too late.
- After the model is trained and validated, we can use it to make predictions, such as applying it to cars with unknown prices to estimate how much they may cost.

In chapter 3, we will learn how to do classification with machine learning, using logistic regression to predict customer churn.

Answers to exercises

- Exercise 2.1 B) Values spread far from the head.
- Exercise 2.2 A) x_i is a feature vector and y_i is the logarithm of the price.
- Exercise 2.3 B) A vector y with price predictions.
- Exercise 2.4 A), B), and C) All three answers are correct.

3 Machine learning for classification

This chapter covers

- Performing exploratory data analysis for identifying important features
- Encoding categorical variables to use them in machine learning models
- Using logistic regression for classification

In this chapter, we are going to use machine learning to predict churn.

Churn is when customers stop using the services of a company. Thus, churn prediction is about identifying customers who are likely to cancel their contracts soon. If the company can do that, it can offer discounts on these services in an effort to keep the users.

Naturally, we can use machine learning for that: we can use past data about customers who churned and, based on that, create a model for identifying present customers who are about to leave. This is a binary classification problem. The target variable that we want to predict is categorical and has only two possible outcomes: churn or not churn.

In chapter 1, we learned that many supervised machine learning models exist, and we specifically mentioned ones that can be used for binary classification,

including logistic regression, decision trees, and neural networks. In this chapter, we start with the simplest one: logistic regression. Even though it's indeed the simplest, it's still powerful and has many advantages over other models: it's fast and easy to understand, and its results are easy to interpret. It's a workhorse of machine learning and the most widely used model in the industry.

3.1 Churn prediction project

The project we prepared for this chapter is churn prediction for a telecom company. We will use logistic regression and Scikit-learn for that.

Imagine that we are working at a telecom company that offers phone and internet services, and we have a problem: some of our customers are churning. They no longer are using our services and are going to a different provider. We would like to prevent that from happening, so we develop a system for identifying these customers and offer them an incentive to stay. We want to target them with promotional messages and give them a discount. We also would like to understand why the model thinks our customers churn, and for that, we need to be able to interpret the model's predictions.

We have collected a dataset where we've recorded some information about our customers: what type of services they used, how much they paid, and how long they stayed with us. We also know who canceled their contracts and stopped using our services (churned). We will use this information as the target variable in the machine learning model and predict it using all other available information.

The plan for the project follows:

1. First, we download the dataset and do some initial preparation: rename columns and change values inside columns to be consistent throughout the entire dataset.
2. Then we split the data into train, validation, and test so we can validate our models.
3. As part of the initial data analysis, we look at feature importance to identify which features are important in our data.
4. We transform categorical variables into numeric variables so we can use them in the model.
5. Finally, we train a logistic regression model.

In the previous chapter, we implemented everything ourselves, using Python and NumPy. In this project, however, we will start using Scikit-learn, a Python library for machine learning. Namely, we will use it for

- Splitting the dataset into train and test
- Encoding categorical variables
- Training logistic regression

3.1.1 Telco churn dataset

As in the previous chapter, we will use Kaggle datasets for data. This time we will use data from https://www.kaggle.com/blastchar/telco-customer-churn.

According to the description, this dataset has the following information:

- Services of the customers: phone; multiple lines; internet; tech support and extra services such as online security, backup, device protection, and TV streaming
- Account information: how long they have been clients, type of contract, type of payment method
- Charges: how much the client was charged in the past month and in total
- Demographic information: gender, age, and whether they have dependents or a partner
- Churn: yes/no, whether the customer left the company within the past month

First, we download the dataset. To keep things organized, we first create a folder, chapter-03-churn-prediction. Then we go to that directory and use Kaggle CLI for downloading the data:

```
kaggle datasets download -d blastchar/telco-customer-churn
```

After downloading it, we unzip the archive to get the CSV file from there:

```
unzip telco-customer-churn.zip
```

We are ready to start now.

3.1.2 Initial data preparation

The first step is creating a new notebook in Jupyter. If it's not running, start it:

```
jupyter notebook
```

We name the notebook chapter-03-churn-project (or any other name that we like).

As previously, we begin with adding the usual imports:

```
import pandas as pd
import numpy as np

import seaborn as sns
from matplotlib import pyplot as plt
%matplotlib inline
```

And now we can read the dataset:

```
df = pd.read_csv('WA_Fn-UseC_-Telco-Customer-Churn.csv')
```

We use the `read_csv` function to read the data and then write the results to a dataframe named `df`. To see how many rows it contains, let's use the `len` function:

```
len(df)
```

It prints 7043, so there are 7,043 rows in this dataset. The dataset is not large but should be enough to train a decent model.

Next, let's look at the first couple of rows using `df.head()` (figure 3.1). By default, it shows the first five rows of the dataframe.

```
df.head()
```

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines	InternetService	OnlineSecurity	...	DeviceProtection	Tech
0	7590-VHVEG	Female	0	Yes	No	1	No	No phone service	DSL	No	...	No	
1	5575-GNVDE	Male	0	No	No	34	Yes	No	DSL	Yes	...	Yes	
2	3668-QPYBK	Male	0	No	No	2	Yes	No	DSL	Yes	...	No	
3	7795-CFOCW	Male	0	No	No	45	No	No phone service	DSL	Yes	...	Yes	
4	9237-HQITU	Female	0	No	No	2	Yes	No	Fiber optic	No	...	No	

Figure 3.1 The output of the `df.head()` command showing the first five rows of the telco churn dataset

This dataframe has quite a few columns, so they all don't fit on the screen. Instead, we can transpose the dataframe using the `T` function, switching columns and rows so the columns (customerID, gender, and so on) become rows. This way we can see a lot more data (figure 3.2):

```
df.head().T
```

We see that the dataset has a few columns:

- CustomerID: the ID of the customer
- Gender: male/female
- SeniorCitizen: whether the customer is a senior citizen (0/1)
- Partner: whether they live with a partner (yes/no)
- Dependents: whether they have dependents (yes/no)
- Tenure: number of months since the start of the contract
- PhoneService: whether they have phone service (yes/no)
- MultipleLines: whether they have multiple phone lines (yes/no/no phone service)
- InternetService: the type of internet service (no/fiber/optic)
- OnlineSecurity: if online security is enabled (yes/no/no internet)
- OnlineBackup: if online backup service is enabled (yes/no/no internet)
- DeviceProtection: if the device protection service is enabled (yes/no/no internet)
- TechSupport: if the customer has tech support (yes/no/no internet)
- StreamingTV: if the TV streaming service is enabled (yes/no/no internet)

```
df.head().T
```

	0	1	2
customerID	7590-VHVEG	5575-GNVDE	3668-QPYBK
gender	Female	Male	Male
SeniorCitizen	0	0	0
Partner	Yes	No	No
Dependents	No	No	No
tenure	1	34	2
PhoneService	No	Yes	Yes
MultipleLines	No phone service	No	No
InternetService	DSL	DSL	DSL
OnlineSecurity	No	Yes	Yes
OnlineBackup	Yes	No	Yes
DeviceProtection	No	Yes	No
TechSupport	No	No	No
StreamingTV	No	No	No
StreamingMovies	No	No	No
Contract	Month-to-month	One year	Month-to-month
PaperlessBilling	Yes	No	Yes
PaymentMethod	Electronic check	Mailed check	Mailed check
MonthlyCharges	29.85	56.95	53.85
TotalCharges	29.85	1889.5	108.15
Churn	No	No	Yes

Figure 3.2 The output of the `df.head().T` command showing the first three rows of the telco churn dataset. The original rows are shown as columns: this way, it's possible to see more data without having to use the slider.

- StreamingMovies: if the movie streaming service is enabled (yes/no/no internet)
- Contract: the type of contract (monthly/yearly/two years)
- PaperlessBilling: if the billing is paperless (yes/no)
- PaymentMethod: payment method (electronic check, mailed check, bank transfer, credit card)
- MonthlyCharges: the amount charged monthly (numeric)
- TotalCharges: the total amount charged (numeric)
- Churn: if the client has canceled the contract (yes/no)

The most interesting one for us is Churn. As the target variable for our model, this is what we want to learn to predict. It takes two values: yes if the customer churned and no if the customer didn't.

When reading a CSV file, Pandas tries to automatically determine the proper type of each column. However, sometimes it's difficult to do it correctly, and the inferred types aren't what we expect them to be. This is why it's important to check whether the actual types are correct. Let's have a look at them by using `df.dtypes`:

```
df.dtypes
```

We see (figure 3.3) that most of the types are inferred correctly. Recall that object means a string value, which is what we expect for most of the columns. However, we may notice two things. First, SeniorCitizen is detected as int64, so it has a type of integer, not object. The reason for this is that instead of the values yes and no, as we have in other columns, there are 1 and 0 values, so Pandas interprets this as a column with integers. It's not really a problem for us, so we don't need to do any additional preprocessing for this column.

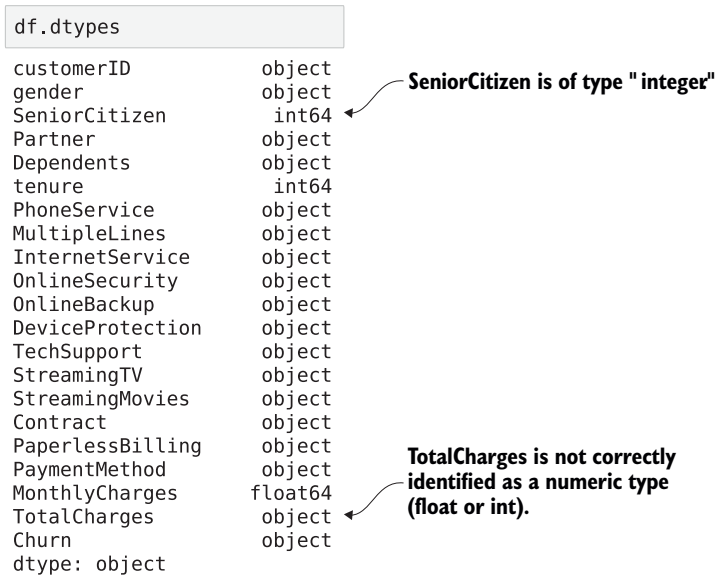


Figure 3.3 Automatically inferred types for all the columns of the dataframe. Object means a string. TotalCharges is incorrectly identified as "object," but it should be "float."

The other thing to note is the type for TotalCharges. We would expect this column to be numeric: it contains the total amount of money the client was charged, so it should be a number, not a string. Yet Pandas infers the type as "object." The reason is that in some cases this column contains a space (" ") to represent a missing value. When coming across nonnumeric characters, Pandas has no other option but to declare the column "object."

IMPORTANT Watch out for cases when you expect a column to be numeric, but Pandas says it's not: most likely the column contains special encoding for missing values that require additional preprocessing.

We can force this column to be numeric by converting it to numbers using a special function in Pandas: `to_numeric`. By default, this function raises an exception when it sees nonnumeric data (such as spaces), but we can make it skip these cases by specifying the `errors='coerce'` option. This way Pandas will replace all nonnumeric values with a `NaN` (not a number):

```
total_charges = pd.to_numeric(df.TotalCharges, errors='coerce')
```

To confirm that data indeed contains nonnumeric characters, we can now use the `isnull()` function of `total_charges` to refer to all the rows where Pandas couldn't parse the original string:

```
df[total_charges.isnull()][['customerID', 'TotalCharges']]
```

We see that indeed there are spaces in the TotalCharges column (figure 3.4).

```
total_charges = pd.to_numeric(df.TotalCharges, errors='coerce')
df[total_charges.isnull()][['customerID', 'TotalCharges']]
```

	customerID	TotalCharges
488	4472-LVYGI	
753	3115-CZMZD	
936	5709-LVOEQ	
1082	4367-NUYAO	
1340	1371-DWPAZ	
3331	7644-OMVMY	
3826	3213-VVOLG	
4380	2520-SGTTA	
5218	2923-ARZLG	
6670	4075-WKNIU	
6754	2775-SEFEE	

Figure 3.4 We can spot nonnumeric data in a column by parsing the content as numeric and see at which rows the parsing fails.

Now it's up to us to decide what to do with these missing values. Although we could do many things with them, we are going to do the same thing we did in the previous chapter — set the missing values to zero:

```
df.TotalCharges = pd.to_numeric(df.TotalCharges, errors='coerce')
df.TotalCharges = df.TotalCharges.fillna(0)
```

In addition, we notice that the column names don't follow the same naming convention. Some of them start with a lower letter, whereas others start with a capital letter, and there are also spaces in the values.

Let's make it uniform by lowercasing everything and replacing spaces with underscores. This way we remove all the inconsistencies in the data. We use the exact same code we used in the previous chapter:

```
df.columns = df.columns.str.lower().str.replace(' ', '_')

string_columns = list(df.dtypes[df.dtypes == 'object'].index)

for col in string_columns:
    df[col] = df[col].str.lower().str.replace(' ', '_')
```

Next, let's look at our target variable: `churn`. Currently, it's categorical, with two values, "yes" and "no" (figure 3.5A). For binary classification, all models typically expect a number: 0 for "no" and 1 for "yes." Let's convert it to numbers:

```
df.churn = (df.churn == 'yes').astype(int)
```

When we use `df.churn == 'yes'`, we create a Pandas series of type boolean. A position in the series is equal to `True` if it's "yes" in the original series and `False` otherwise. Because the only other value it can take is "no," this converts "yes" to `True` and "no" to `False` (figure 3.5B). When we perform casting by using the `astype(int)` function, we

```
df.churn.head()
```

```
0     no
1     no
2    yes
3     no
4    yes
Name: churn, dtype: object
```

(A) The original Churn column: it's a Pandas series that contains only "yes" and "no" values.

```
(df.churn == 'yes').head()
```

```
0    False
1    False
2     True
3    False
4     True
Name: churn, dtype: bool
```

(B) The result of the == operator: it's a Boolean series with `True` when the elements of the original series are "yes" and `False` otherwise.

```
(df.churn == 'yes').astype(int).head()
```

```
0    0
1    0
2    1
3    0
4    1
Name: churn, dtype: int64
```

(C) The result of converting the Boolean series to integer: `True` is converted to 1 and `False` is converted to 0.

Figure 3.5 The expression `(df.churn == 'yes').astype(int)` broken down by individual steps

convert `True` to 1 and `False` to 0 (figure. 3.5C). This is exactly the same idea that we used in the previous chapter when we implemented category encoding.

We've done a bit of preprocessing already, so let's put aside some data for testing. In the previous chapter, we implemented the code for doing it ourselves. This is great for understanding how it works, but typically we don't write such things from scratch every time we need them. Instead, we use existing implementations from libraries. In this chapter we use Scikit-learn, and it has a module called `model_selection` that can handle data splitting. Let's use it.

The function we need to import from `model_selection` is called `train_test_split`:

```
from sklearn.model_selection import train_test_split
```

After importing, it's ready to be used:

```
df_train_full, df_test = train_test_split(df, test_size=0.2, random_state=1)
```

The function `train_test_split` takes a dataframe `df` and creates two new dataframes: `df_train_full` and `df_test`. It does this by shuffling the original dataset and then splitting it in such a way that the test set contains 20% of the data and the train set contains the remaining 80% (figure 3.6). Internally, it's implemented similarly to what we did ourselves in the previous chapter.

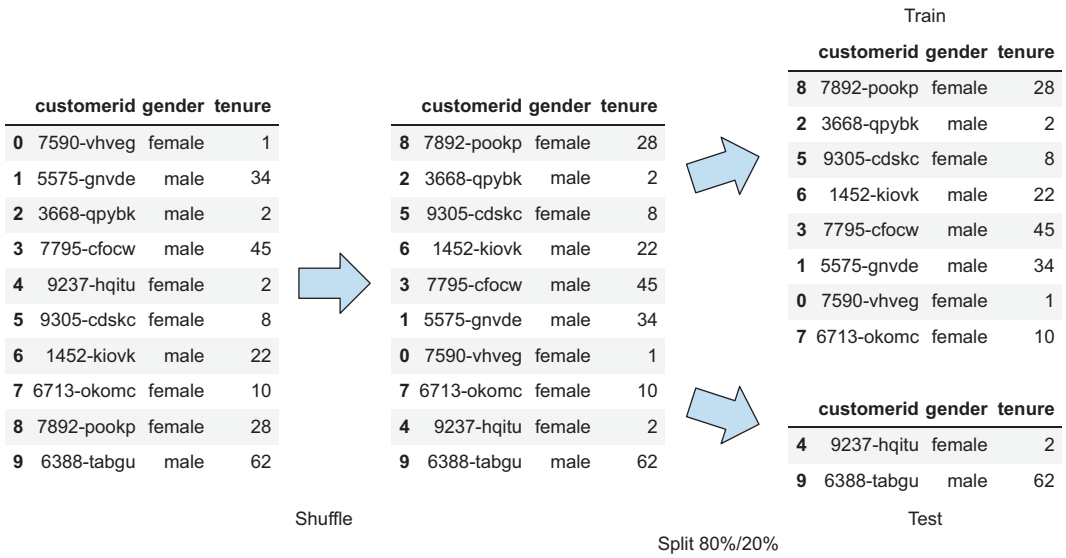


Figure 3.6 When using `train_test_split`, the original dataset is shuffled and then split such that 80% of the data goes to the train set and the remaining 20% goes to the test set.

This function contains a few parameters:

1. The first parameter that we pass is the dataframe that we want to split: `df`.
2. The second parameter is `test_size`, which specifies the size of the dataset we want to set aside for testing — 20% for our case.
3. The third parameter we pass is `random_state`. It's needed for ensuring that every time we run this code, the dataframe is split in the exact same way.

Shuffling of data is done using a random-number generator; it's important to fix the random seed to ensure that every time we shuffle the data, the final arrangement of rows will be the same.

```
df_train_full.head()
```

	customerid	gender	seniorcitizen	partner	dependents	tenure	phoneservice
1814	5442-pptjy	male	0	yes	yes	12	yes
5946	6261-rcvns	female	0	no	no	42	yes
3881	2176-osjuv	male	0	yes	no	71	yes
2389	6161-erdgd	male	0	yes	yes	71	yes
3676	2364-ufrom	male	0	no	no	30	yes

Figure 3.7 The side effect of `train_test_split`: the indices (the first column) are shuffled in the new dataframes, so instead of consecutive numbers like 0, 1, 2, …, they look random.

We do see a side effect from shuffling: if we look at the dataframes after splitting by using the `head()` method, for example, we notice that the indices appear to be randomly ordered (figure 3.7).

In the previous chapter, we split the data into three parts: train, validation, and test. However, the `train_test_split` function splits the data into only two parts: train and test. In spite of that, we can still split the original dataset into three parts; we just take one part and split it again (figure 3.8).

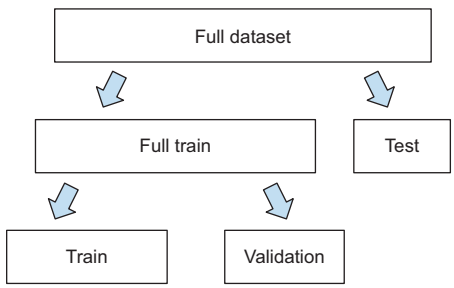


Figure 3.8 Because `train_test_split` splits a dataset into only two parts, we perform the split two times because we need three parts. First, we split the entire dataset into full train and test, and then we split full train into train and validation.

Let's take the `df_train_full` dataframe and split it one more time into train and validation:

```
df_train, df_val = train_test_split(df_train_full, test_size=0.33,
     random_state=11)

y_train = df_train.churn.values
y_val = df_val.churn.values

del df_train['churn']
del df_val['churn']
```

Sets the random seed when doing the split to make sure that every time we run the code, the result is the same

Takes the column with the target variable, churn, and saves it outside the dataframe

Deletes the churn columns from both dataframes to make sure we don't accidentally use the churn variable as a feature during training

Now the dataframes are prepared, and we are ready to use the training dataset for performing initial exploratory data analysis.

3.1.3 Exploratory data analysis

Looking at the data before training a model is important. The more we know about the data and the problems inside, the better the model we can build afterward.

We should always check for any missing values in the dataset because many machine learning models cannot easily deal with missing data. We have already found a problem with the TotalCharges column and replaced the missing values with zeros. Now let's see if we need to perform any additional null handling:

```
df_train_full.isnull().sum()
```

It prints all zeros (figure 3.9), so we have no missing values in the dataset and don't need to do anything extra.

```
df_train_full.isnull().sum()
```

```
customerid          0
gender              0
seniorcitizen       0
partner             0
dependents          0
tenure              0
phoneservice        0
multiplelines       0
internetservice     0
onlinesecurity      0
onlinebackup        0
deviceprotection    0
techsupport         0
streamingtv         0
streamingmovies     0
contract            0
paperlessbilling    0
paymentmethod       0
monthlycharges      0
totalcharges        0
churn               0
dtype: int64
```

Figure 3.9 We don't have to handle missing values in the dataset: all the values in all the columns are present.

Another thing we should do is check the distribution of values in the target variable. Let's take a look at it using the `value_counts()` method:

```
df_train_full.churn.value_counts()
```

It prints

```
0    4113
1    1521
```

The first column is the value of the target variable, and the second is the count. As we see, the majority of the customers didn't churn.

We know the absolute numbers, but let's also check the proportion of churned users among all customers. For that, we need to divide the number of customers who churned by the total number of customers. We know that 1,521 of 5,634 churned, so the proportion is

$$1521 \ / \ 5634 = 0.27$$

This gives us the proportion of churned users, or the probability that a customer will churn. As we see in the training dataset, approximately 27% of the customers stopped using our services, and the rest remained as customers.

The proportion of churned users, or the probability of churning, has a special name: churn rate.

There's another way to calculate the churn rate: the `mean()` method. It's more convenient to use than manually calculating the rate:

```
global_mean = df_train_full.churn.mean()
```

Using this method, we also get 0.27 (figure 3.10).

```
global_mean = df_train_full.churn.mean()
round(global_mean, 3)
```

```
0.27
```

Figure 3.10 Calculating the global churn rate in the training dataset

The reason it produces the same result is the way we calculate the mean value. If you don't remember, the formula for that is

$$\frac{1}{n}\sum_{i=1}^{n} y_i$$

where n is the number of items in the dataset.

Because y_i can take only zeros and ones, when we sum all of them, we get the number of ones, or the number of people who churned. Then we divide it by the total number of customers, which is exactly the same as the formula we used for calculating the churn rate previously.

Our churn dataset is an example of a so-called *imbalanced* dataset. There were three times as many people who didn't churn in our dataset as those who did churn, and we say that the nonchurn class dominates the churn class. We can clearly see that: the churn rate in our data is 0.27, which is a strong indicator of class imbalance. The opposite of *imbalanced* is the *balanced* case, when positive and negative classes are equally distributed among all observations.

Exercise 3.1

The mean of a Boolean array is

a. The percentage of `False` elements in the array: the number of `False` elements divided by the length of the array
b. The percentage of `True` elements in the array: the number of `True` elements divided by the length of the array
c. The length of an array

Both the categorical and numerical variables in our dataset are important, but they are also different and need different treatment. For that, we want to look at them separately.

We will create two lists:

- `categorical`, which will contain the names of categorical variables
- `numerical`, which, likewise, will have the names of numerical variables

Let's create them:

```
categorical = ['gender', 'seniorcitizen', 'partner', 'dependents',
               'phoneservice', 'multiplelines', 'internetservice',
               'onlinesecurity', 'onlinebackup', 'deviceprotection',
               'techsupport', 'streamingtv', 'streamingmovies',
               'contract', 'paperlessbilling', 'paymentmethod']
numerical = ['tenure', 'monthlycharges', 'totalcharges']
```

First, we can see how many unique values each variable has. We already know we should have just a few for each column, but let's verify it:

```
df_train_full[categorical].nunique()
```

Indeed, we see that most of the columns have two or three values and one (payment-method) has four (figure 3.11). This is good. We don't need to spend extra time preparing and cleaning the data; everything is already good to go.

Now we come to another important part of exploratory data analysis: understanding which features may be important for our model.

```
df_train_full[categorical].nunique()
```

```
gender              2
seniorcitizen       2
partner             2
dependents          2
phoneservice        2
multiplelines       3
internetservice     3
onlinesecurity      3
onlinebackup        3
deviceprotection    3
techsupport         3
streamingtv         3
streamingmovies     3
contract            3
paperlessbilling    2
paymentmethod       4
dtype: int64
```

Figure 3.11 The number of distinct values for each categorical variable. We see that all the variables have very few unique values.

3.1.4 Feature importance

Knowing how other variables affect the target variable, churn, is the key to understanding the data and building a good model. This process is called *feature importance analysis*, and it's often done as a part of exploratory data analysis to figure out which variables will be useful for the model. It also gives us additional insights about the dataset and helps answer questions like "What makes customers churn?" and "What are the characteristics of people who churn?"

We have two different kinds of features: categorical and numerical. Each kind has different ways of measuring feature importance, so we will look at each separately.

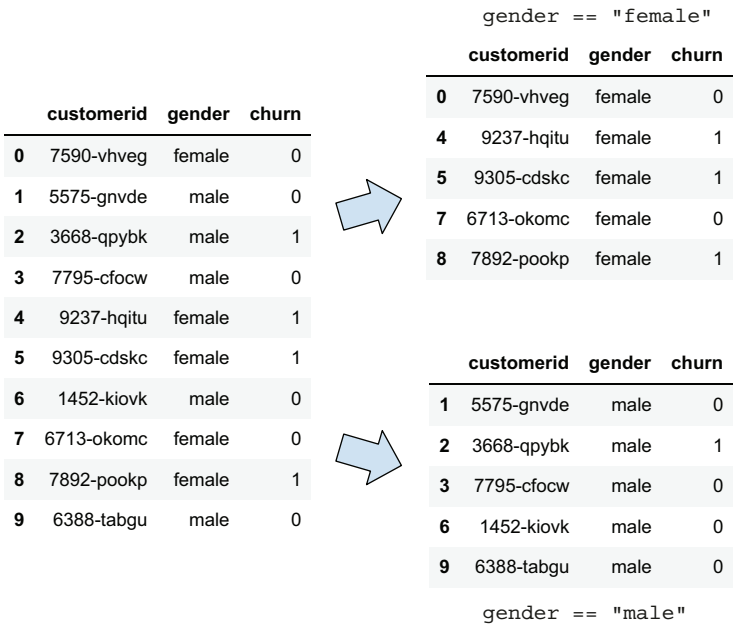
CHURN RATE

Let's start by looking at categorical variables. The first thing we can do is look at the churn rate for each variable. We know that a categorical variable has a set of values it can take, and each value defines a group inside the dataset.

We can look at all the distinct values of a variable. Then, for each variable, there's a group of customers: all the customers who have this value. For each such group, we can compute the churn rate, which is the group churn rate. When we have it, we can compare it with the global churn rate — the churn rate calculated for all the observations at once.

If the difference between the rates is small, the value is not important when predicting churn because this group of customers is not really different from the rest of the customers. On the other hand, if the difference is not small, something inside that group sets it apart from the rest. A machine learning algorithm should be able to pick this up and use it when making predictions.

Let's check first for the `gender` variable. This `gender` variable can take two values, female and male. There are two groups of customers: ones that have `gender == 'female'` and ones that have `gender == 'male'` (figure 3.12). To compute the churn rate for all



	customerid	gender	churn
0	7590-vhveg	female	0
1	5575-gnvde	male	0
2	3668-qpybk	male	1
3	7795-cfocw	male	0
4	9237-hqitu	female	1
5	9305-cdskc	female	1
6	1452-kiovk	male	0
7	6713-okomc	female	0
8	7892-pookp	female	1
9	6388-tabgu	male	0

	customerid	gender	churn
0	7590-vhveg	female	0
4	9237-hqitu	female	1
5	9305-cdskc	female	1
7	6713-okomc	female	0
8	7892-pookp	female	1

	customerid	gender	churn
1	5575-gnvde	male	0
2	3668-qpybk	male	1
3	7795-cfocw	male	0
6	1452-kiovk	male	0
9	6388-tabgu	male	0

Figure 3.12 The dataframe is split by the values of the `gender` variable into two groups: a group with `gender == "female"` and a group with `gender == "male"`.

female customers, we first select only rows that correspond to `gender == 'female'` and then compute the churn rate for them:

```
female_mean = df_train_full[df_train_full.gender == 'female'].churn.mean()
```

We then do the same for all male customers:

```
male_mean = df_train_full[df_train_full.gender == 'male'].churn.mean()
```

When we execute this code and check the results, we see that the churn rate of female customers is 27.7% and that of male customers is 26.3%, whereas the global churn rate is 27% (figure 3.13). The difference between the group rates for both females and males is quite small, which indicates that knowing the gender of the customer doesn't help us identify whether they will churn.

Now let's take a look at another variable: `partner`. It takes values of yes and no, so there are two groups of customers: the ones for which `partner == 'yes'` and the ones for which `partner == 'no'`.

We can check the group churn rates using the same code as we used previously. All we need to change is the filter conditions:

```
partner_yes = df_train_full[df_train_full.partner == 'yes'].churn.mean()
partner_no = df_train_full[df_train_full.partner == 'no'].churn.mean()
```

```
global_mean = df_train_full.churn.mean()
round(global_mean, 3)
```

```
0.27
```

```
female_mean = df_train_full[df_train_full.gender == 'female'].churn.mean()
print('gender == female:', round(female_mean, 3))

male_mean = df_train_full[df_train_full.gender == 'male'].churn.mean()
print('gender == male:  ', round(male_mean, 3))
```

```
gender == female: 0.277
gender == male:   0.263
```

Figure 3.13 The global churn rate compared with churn rates among males and females. The numbers are quite close, which means that `gender` is not a useful variable when predicting churn.

As we see, the rates for those who have a partner are quite different from rates for those who don't: 20% and 33%, respectively. It means that clients with no partner are more likely to churn than the ones with a partner (figure 3.14).

```
partner_yes = df_train_full[df_train_full.partner == 'yes'].churn.mean()
print('partner == yes:', round(partner_yes, 3))

partner_no = df_train_full[df_train_full.partner == 'no'].churn.mean()
print('partner == no :', round(partner_no, 3))
```

```
partner == yes: 0.205
partner == no : 0.33
```

Figure 3.14 The churn rate for people with a partner is significantly less than the rate for the ones without a partner — 20.5% versus 33% — which indicates that the `partner` variable is useful for predicting churn.

RISK RATIO

In addition to looking at the difference between the group rate and the global rate, it's interesting to look at the ratio between them. In statistics, the ratio between probabilities in different groups is called the *risk ratio*, where *risk* refers to the risk of having the effect. In our case, the effect is churn, so it's the risk of churning:

$$\text{risk} = \text{group rate} \;/\; \text{global rate}$$

For `gender == female`, for example, the risk of churning is 1.02:

$$\text{risk} = 27.7\% \;/\; 27\% = 1.02$$

Risk is a number between zero and infinity. It has a nice interpretation that tells you how likely the elements of the group are to have the effect (churn) compared with the entire population.

If the difference between the group rate and the global rate is small, the risk is close to 1: this group has the same level of risk as the rest of the population. Customers in the group are as likely to churn as anyone else. In other words, a group with a risk close to 1 is not risky at all (figure 3.15, group A).

If the risk is lower than 1, the group has lower risks: the churn rate in this group is smaller than the global churn. For example, the value 0.5 means that the clients in this group are two times less likely to churn than clients in general (figure 3.15, group B).

On the other hand, if the value is higher than 1, the group is risky: there's more churn in the group than in the population. So a risk of 2 means that customers from the group are two times more likely to churn (figure 3.15, group C).

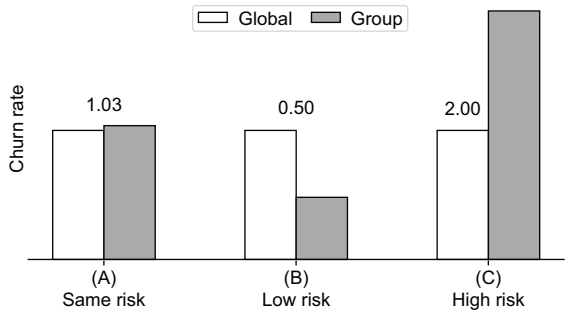


Figure 3.15 Churn rate of different groups compared with the global churn rate. In group (A), the rates are approximately the same, so the risk of churn is around 1. In group (B), the group churn rate is smaller than the global rate, so the risk is around 0.5. Finally, in group (C), the group churn rate is higher than the global rate, so the risk is close to 2.

The term *risk* originally comes from controlled trials, in which one group of patients is given a treatment (a medicine) and the other group isn't (only a placebo). Then we compare how effective the medicine is by calculating the rate of negative outcomes in each group and then calculating the ratio between the rates:

$$\text{risk} = \text{negative outcome rate in group 1} \;/\; \text{negative outcome rate in group 2}$$

If medicine turns out to be effective, it's said to reduce the risk of having the negative outcome, and the value of the risk is less than 1.

Let's calculate the risks for `gender` and `partner`. For the `gender` variable, the risks for both males and females is around 1 because the rates in both groups aren't significantly different from the global rate. Not surprisingly, it's different for the `partner` variable; having no partner is more risky (table 3.1).

We did this from only two variables. Let's now do this for all the categorical variables. To do that, we need a piece of code that checks all the values a variable has and computes churn rate for each of these values.

Table 3.1 Churn rates and risks for the `gender` and `partner` variables. The churn rates for females and males are not significantly different from the global churn rates, so the risks for them to churn are low: both have risks values around 1. On the other hand, the churn rate for people with no partner is significantly higher than average, making them risky, with the risk value of 1.22. People with partners tend to churn less, so for them, the risk is only 0.75.

Variable	Value	Churn rate	Risk
gender	Female	27.7%	1.02
	Male	26.3%	0.97
partner	Yes	20.5%	0.75
	No	33%	1.22

If we used SQL, that would be straightforward to do. For gender, we'd need to do something like this:

```
SELECT
    gender, AVG(churn),
    AVG(churn) - global_churn,
    AVG(churn) / global_churn
FROM
    data
GROUP BY
    gender
```

This is a rough translation to Pandas:

```
global_mean = df_train_full.churn.mean()

df_group = df_train_full.groupby(by='gender').churn.agg(['mean'])
df_group['diff'] = df_group['mean'] - global_mean
df_group['risk'] = df_group['mean'] / global_mean

df_group
```

Computes the AVG(churn) ❶

Calculates the difference between group churn rate and global rate ❷

Calculates the risk of churning ❸

In ❶ we calculate the `AVG(churn)` part. For that, we use the `agg` function to indicate that we need to aggregate data into one value per group: the mean value. In ❷ we create another column, diff, where we will keep the difference between the group mean and the global mean. Likewise, in ❸ we create the column risk, where we calculate the fraction between the group mean and the global mean.

We can see the results in figure 3.16.

gender	mean	diff	risk
female	0.276824	0.006856	1.025396
male	0.263214	-0.006755	0.974980

Figure 3.16 The churn rate for the `gender` variable. We see that for both values, the difference between the group churn rate and the global churn rate is not very large.

Let's now do that for all categorical variables. We can iterate through them and apply the same code for each:

```
from IPython.display import display

for col in categorical:
    df_group = df_train_full.groupby(by=col).churn.agg(['mean'])
    df_group['diff'] = df_group['mean'] - global_mean
    df_group['rate'] = df_group['mean'] / global_mean
    display(df_group)
```

Loops over all categorical variables

Performs groupby for each categorical variable

Displays the resulting dataframe

Two things are different in this code. First, instead of manually specifying the column name, we iterate over all categorical variables.

The second difference is more subtle: we need to call the `display` function to render a dataframe inside the loop. The way we typically display a dataframe is to leave it as the last line in a Jupyter Notebook cell and then execute the cell. If we do it that way, the dataframe is displayed as the cell output. This is exactly how we managed to see the content of the dataframe at the beginning of the chapter (figure 3.1). However, we cannot do this inside a loop. To still be able to see the content of the dataframe, we call the `display` function explicitly.

From the results (figure 3.17) we learn that

- For gender, there is not much difference between females and males. Both means are approximately the same, and for both groups the risks are close to 1.
- Senior citizens tend to churn more than nonseniors: the risk of churning is 1.53 for seniors and 0.89 for nonseniors.
- People with a partner churn less than people with no partner. The risks are 0.75 and 1.22, respectively.
- People who use phone service are not at risk of churning: the risk is close to 1, and there's almost no difference with the global churn rate. People who don't use phone service are even less likely to churn: the risk is below 1, and the difference with the global churn rate is negative.

gender	mean	diff	risk
female	0.276824	0.006856	1.025396
male	0.263214	-0.006755	0.974980

(A) Churn ratio and risk: `gender`

seniorcitizen	mean	diff	risk
0	0.242270	-0.027698	0.897403
1	0.413377	0.143409	1.531208

(B) Churn ratio and risk: `seniorcitizen`

partner	mean	diff	risk
no	0.329809	0.059841	1.221659
yes	0.205033	-0.064935	0.759472

(C) Churn ratio and risk: `partner`

phoneservice	mean	diff	risk
no	0.241316	-0.028652	0.893870
yes	0.273049	0.003081	1.011412

(D) Churn ratio and risk: `phoneservice`

Figure 3.17 Churn rate difference and risk for four categorical variables: `gender`, `seniorcitizen`, `partner`, and `phoneservice`

Some of the variables have quite significant differences (figure 3.18):

- Clients with no tech support tend to churn more than those who do.
- People with monthly contracts cancel the contract a lot more often than others, and people with two-year contacts churn very rarely.

techsupport	mean	diff	risk
no	0.418914	0.148946	1.551717
no_internet_service	0.077805	-0.192163	0.288201
yes	0.159926	-0.110042	0.592390

(A) Churn ratio and risk: `techsupport`

contract	mean	diff	risk
month-to-month	0.431701	0.161733	1.599082
one_year	0.120573	-0.149395	0.446621
two_year	0.028274	-0.241694	0.104730

(B) Churn ratio and risk: `contract`

Figure 3.18 Difference between the group churn rate and the global churn rate for `techsupport` and `contract`. People with no tech support and month-to-month contracts tend to churn a lot more than clients from other groups, whereas people with tech support and two-year contracts are very low-risk clients.

This way, just by looking at the differences and the risks, we can identify the most discriminative features: the features that are helpful for detecting churn. Thus, we expect that these features will be useful for our future models.

MUTUAL INFORMATION

The kinds of differences we just explored are useful for our analysis and important for understanding the data, but it's hard to use them to say what the most important feature is and whether tech support is more useful than the type of contract.

Luckily, the metrics of importance can help us: we can measure the degree of dependency between a categorical variable and the target variable. If two variables are dependent, knowing the value of one variable gives us some information about another. On the other hand, if a variable is completely independent of the target variable, it's not useful and can be safely removed from the dataset.

In our case, knowing that the customer has a month-to-month contract may indicate that this customer is more likely to churn than not.

> **IMPORTANT** Customers with month-to-month contracts tend to churn a lot more than customers with other kinds of contracts. This is exactly the kind of relationship we want to find in our data. Without such relationships in data, machine learning models will not work — they will not be able to make predictions. The higher the degree of dependency, the more useful a feature is.

For categorical variables, one such metric is mutual information, which tells how much information we learn about one variable if we learn the value of the other variable. It's a concept from information theory, and in machine learning, we often use it to measure the mutual dependency between two variables.

Higher values of mutual information mean a higher degree of dependence: if the mutual information between a categorical variable and the target is high, this categorical variable will be quite useful for predicting the target. On the other hand, if the mutual information is low, the categorical variable and the target are independent, and thus the variable will not be useful for predicting the target.

Mutual information is already implemented in Scikit-learn in the `mutual_info_score` function from the `metrics` package, so we can just use it:

```
from sklearn.metrics import mutual_info_score

def calculate_mi(series):
    return mutual_info_score(series, df_train_full.churn)

df_mi = df_train_full[categorical].apply(calculate_mi)
df_mi = df_mi.sort_values(ascending=False).to_frame(name='MI')
df_mi
```

1 Creates a stand-alone function for calculating mutual information

2 Uses the mutual_info_score function from Scikit-learn

3 Applies the function from 1 to each categorical column of the dataset

4 Sorts the values of the result

In 3, we use the `apply` method to apply the `calculate_mi` function we defined in 1 to each column of the `df_train_full` dataframe. Because we include an additional step of selecting only categorical variables, it's applied only to them. The function we define in 1 takes only one parameter: `series`. This is a column from the dataframe

on which we invoked the `apply()` method. In ❷, we compute the mutual information score between the series and the target variable `churn`. The output is a single number, so the output of the `apply()` method is a Pandas series. Finally, we sort the elements of the series by the mutual information score and convert the series to a dataframe. This way, the result is rendered nicely in Jupyter.

As we see, `contract`, `onlinesecurity`, and `techsupport` are among the most important features (figure 3.19). Indeed, we've already noted that `contract` and `techsupport` are quite informative. It's also not surprising that `gender` is among the least important features, so we shouldn't expect it to be useful for the model.

	MI
contract	0.098320
onlinesecurity	0.063085
techsupport	0.061032
internetservice	0.055868
onlinebackup	0.046923

(A) The most useful features according to the mutual information score.

	MI
partner	0.009968
seniorcitizen	0.009410
multiplelines	0.000857
phoneservice	0.000229
gender	0.000117

(B) The least useful features according to the mutual information score.

Figure 3.19 Mutual information between categorical variables and the target variable. Higher values are better. According to it, `contract` is the most useful variable, whereas `gender` is the least useful.

CORRELATION COEFFICIENT

Mutual information is a way to quantify the degree of dependency between two categorical variables, but it doesn't work when one of the features is numerical, so we cannot apply it to the three numerical variables that we have.

We can, however, measure the dependency between a binary target variable and a numerical variable. We can pretend that the binary variable is numerical (containing only the numbers zero and one) and then use the classical methods from statistics to check for any dependency between these variables.

One such method is the *correlation* coefficient (sometimes referred as *Pearson's correlation coefficient*). It is a value from –1 to 1:

- Positive correlation means that when one variable goes up, the other variable tends to go up as well. In the case of a binary target, when the values of the variable are high, we see ones more often than zeros. But when the values of the variable are low, zeros become more frequent than ones.
- Zero correlation means no relationship between two variables: they are completely independent.

- Negative correlation occurs when one variable goes up and the other goes down. In the binary case, if the values are high, we see more zeros than ones in the target variable. When the values are low, we see more ones.

It's very easy to calculate the correlation coefficient in Pandas:

```
df_train_full[numerical].corrwith(df_train_full.churn)
```

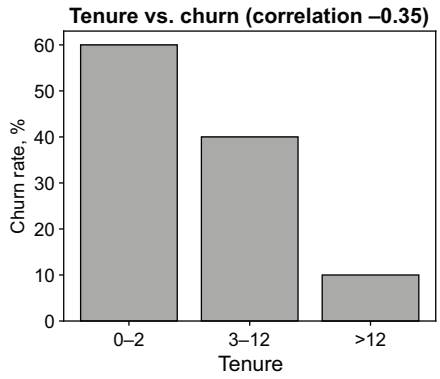
We see the results in figure 3.20:

- The correlation between `tenure` and churn is –0.35: it has a negative sign, so the longer customers stay, the less often they tend to churn. For customers staying with the company for two months or less, the churn rate is 60%; for customers with tenure between 3 and 12 months, the churn rate is 40%; and for customers staying longer than a year, the churn rate is 17%. So the higher the value of tenure, the smaller the churn rate (figure 3.21A).
- `monthlycharges` has a positive coefficient of 0.19, which means that customers who pay more tend to leave more often. Only 8% of those who pay less than $20 monthly churned; customers paying between $21 and $50 churn more frequently with a churn rate of 18%; and 32% of people paying more than $50 churned (figure 3.21B).
- `totalcharges` has a negative correlation, which makes sense: the longer people stay with the company, the more they have paid in total, so it's less likely that they will leave. In this case, we expect a pattern similar to `tenure`. For small values, the churn rate is high; for larger values, it's lower.

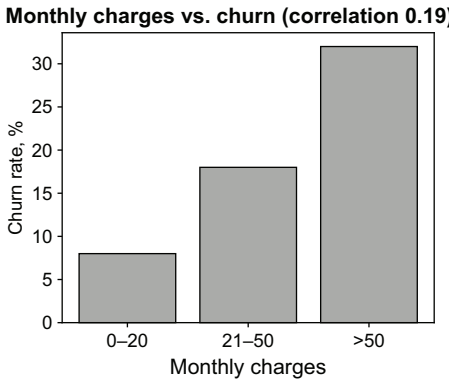
	correlation
tenure	-0.351885
monthlycharges	0.196805
totalcharges	-0.196353

Figure 3.20 Correlation between numerical variables and churn. `tenure` has a high negative correlation: as tenure grows, churn rate goes down. `monthlycharges` has positive correlation: the more customers pay, the more likely they are to churn.

After doing initial exploratory data analysis, identifying important features, and getting some insights into the problem, we are ready to do the next step: feature engineering and model training.



(A) Churn rate for different values of `tenure`. The correlation coefficient is negative, so the trend is downward: for higher values of `tenure`, the churn rate is smaller.



(B) Churn rate for different values of `monthlycharges`. The correlation coefficient is positive, so the trend is upward: for higher values of `monthlycharges`, the churn rate is higher.

Figure 3.21 Churn rate for `tenure` (negative correlation of –0.35) and `monthlycharges` (positive correlation of 0.19)

3.2 Feature engineering

We had an initial look at the data and identified what could be useful for the model. After doing that, we have a clear understanding how other variables affect churn — our target.

Before we proceed to training, however, we need to perform the feature engineering step: transforming all categorical variables to numeric features. We'll do that in the next section, and after that, we'll be ready to train the logistic regression model.

3.2.1 One-hot encoding for categorical variables

As we already know from the first chapter, we cannot just take a categorical variable and put it into a machine learning model. The models can deal only with numbers in matrices. So, we need to convert our categorical data into a matrix form, or encode.

One such encoding technique is *one-hot encoding*. We already saw this encoding technique in the previous chapter, when creating features for the make of a car and other categorical variables. There, we mentioned it only briefly and used it in a very simple way. In this chapter, we will spend more time understanding and using it.

If a variable `contract` has possible values (monthly, yearly, and two-year), we can represent a customer with the yearly contract as (0, 1, 0). In this case, the yearly value is active, or *hot*, so it gets 1, whereas the remaining values are not active, or *cold*, so they are 0.

To understand this better, let's consider a case with two categorical variables and see how we create a matrix from them. These variables are

- `gender`, with values female and male
- `contract`, with values monthly, yearly, and two-year

Because the `gender` variable has only two possible values, we create two columns in the resulting matrix. The `contract` variable has three columns, and in total, our new matrix will have five columns:

- gender=female
- gender=male
- contract=monthly
- contract=yearly
- contract=two-year

Let's consider two customers (figure 3.22):

- A female customer with a yearly contract
- A male customer with a monthly contract

For the first customer, the `gender` variable is encoded by putting 1 in the gender =female column and 0 in the gender=male column. Likewise, contract=yearly gets 1, whereas the remaining contract columns, contract=monthly and contract=two-year, get 0.

As for the second customer, gender=male and contract=monthly get ones, and the rest of the columns get zeros (figure 3.22).

gender	contract
male	monthly
female	yearly

gender		contract		
female	male	monthly	yearly	two-year
0	**1**	**1**	0	0
1	0	0	**1**	0

Figure 3.22 The original dataset with categorical variables is on the left and the one-hot encoded representation on the right. For the first customer, gender=male and contract=monthly are the hot columns, so they get 1. For the second customer, the hot columns are gender=female and contract=yearly.

The way we implemented it previously was simple but quite limited. We first looked at the top five values of the variable and then looped over each value and manually created a column in the dataframe. When the number of features grows, however, this process becomes tedious.

Luckily, we don't need to implement this by hand: we can use Scikit-learn. We can perform one-hot encoding in multiple ways in Scikit-learn, but we will use `DictVectorizer`.

As the name suggests, `DictVectorizer` takes in a dictionary and *vectorizes* it — that is, it creates vectors from it. Then the vectors are put together as rows of one matrix. This matrix is used as input to a machine learning algorithm (figure 3.23).

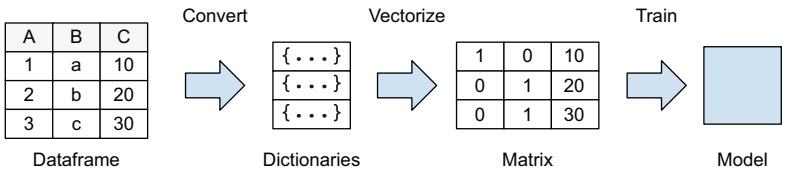


Figure 3.23 The process of creating a model. First, we convert a dataframe to a list of dictionaries, then we vectorize the list to a matrix, and finally, we use the matrix to train a model.

To use this method, we need to convert our dataframe to a list of dictionaries, which is simple to do in Pandas using the `to_dict` method with the `orient="records"` parameter:

```
train_dict = df_train[categorical + numerical].to_dict(orient='records')
```

If we take a look at the first element of this new list, we see

```
{'gender': 'male',
 'seniorcitizen': 0,
 'partner': 'yes',
 'dependents': 'yes',
 'phoneservice': 'yes',
 'multiplelines': 'no',
 'internetservice': 'no',
 'onlinesecurity': 'no_internet_service',
 'onlinebackup': 'no_internet_service',
 'deviceprotection': 'no_internet_service',
 'techsupport': 'no_internet_service',
 'streamingtv': 'no_internet_service',
 'streamingmovies': 'no_internet_service',
 'contract': 'two_year',
 'paperlessbilling': 'no',
 'paymentmethod': 'mailed_check',
 'tenure': 12,
 'monthlycharges': 19.7,
 'totalcharges': 258.35}
```

Each column from the dataframe is the key in this dictionary, with values coming from the actual dataframe row values.

Now we can use `DictVectorizer`. We create it and then fit it to the list of dictionaries we created previously:

```
from sklearn.feature_extraction import DictVectorizer

dv = DictVectorizer(sparse=False)
dv.fit(train_dict)
```

In this code we create a `DictVectorizer` instance, which we call `dv`, and "train" it by invoking the `fit` method. The `fit` method looks at the content of these dictionaries and figures out the possible values for each variable and how to map them to the columns in the output matrix. If a feature is categorical, it applies the one-hot encoding scheme, but if a feature is numerical, it's left intact.

The `DictVectorizer` class can take in a set of parameters. We specify one of them: `sparse=False`. This parameter means that the created matrix will not be sparse and instead will create a simple NumPy array. If you don't know about sparse matrices, don't worry: we don't need them in this chapter.

After we fit the vectorizer, we can use it for converting the dictionaries to a matrix by using the `transform` method:

```
X_train = dv.transform(train_dict)
```

This operation creates a matrix with 45 columns. Let's have a look at the first row, which corresponds to the customer we looked at previously:

```
X_train[0]
```

When we put this code into a Jupyter Notebook cell and execute it, we get the following output:

```
array([   0. ,    0. ,    1. ,    1. ,    0. ,    0. ,    0. ,    1. ,
          0. ,    1. ,    1. ,    0. ,    0. ,   86.1,    1. ,    0. ,
          0. ,    0. ,    0. ,    1. ,    0. ,    0. ,    1. ,    0. ,
          1. ,    0. ,    1. ,    1. ,    0. ,    0. ,    0. ,    0. ,
          1. ,    0. ,    0. ,    0. ,    1. ,    0. ,    0. ,    1. ,
          0. ,    0. ,    1. ,   71. , 6045.9])
```

As we see, most of the elements are ones and zeros — they're one-hot encoded categorical variables. Not all of them are ones and zeros, however. We see that three of them are other numbers. These are our numeric variables: `monthlycharges`, `tenure`, and `totalcharges`.

We can learn the names of all these columns by using the `get_feature_names` method:

```
dv.get_feature_names()
```

It prints

```
['contract=month-to-month',
 'contract=one_year',
 'contract=two_year',
 'dependents=no',
```

```
 'dependents=yes',
 # some rows omitted
 'tenure',
 'totalcharges']
```

As we see, for each categorical feature it creates multiple columns for each of its distinct values. For `contract`, we have `contract=month-to-month`, `contract=one_year`, and `contract=two_year`, and for `dependents`, we have `dependents=no` and `dependents=yes`. Features such as `tenure` and `totalcharges` keep the original names because they are numerical; therefore, `DictVectorizer` doesn't change them.

Now our features are encoded as a matrix, so we can move to the next step: using a model to predict churn.

Exercise 3.2

How would `DictVectorizer` encode the following list of dictionaries?

```
records = [
    {'total_charges': 10, 'paperless_billing': 'yes'},
    {'total_charges': 30, 'paperless_billing': 'no'},
    {'total_charges': 20, 'paperless_billing': 'no'}
]
```

a Columns: `['total_charges', 'paperless_billing=yes', 'paperless_billing=no']`

Values: `[10, 1, 0], [30, 0, 1], [20, 0, 1]`

b Columns: `['total_charges=10', 'total_charges=20', 'total_charges=30', 'paperless_billing=yes', 'paperless_billing=no']`

Values: `[1, 0, 0, 1, 0], [0, 0, 1, 0, 1], [0, 1, 0, 0, 1]`

3.3 Machine learning for classification

We have learned how to use Scikit-learn to perform one-hot encoding for categorical variables, and now we can transform them into a set of numerical features and put everything together into a matrix.

When we have a matrix, we are ready to do the model training part. In this section we learn how to train the logistic regression model and interpret its results.

3.3.1 Logistic regression

In this chapter, we use logistic regression as a classification model, and now we train it to distinguish churned and not-churned users.

Logistic regression has a lot in common with linear regression, the model we learned in the previous chapter. If you remember, the linear regression model is a regression model that can predict a number. It has the form

$$g(x_i) = w_0 + x_i^T w$$

where

- x_i is the feature vector corresponding to the ith observation.
- w_0 is the bias term.
- w is a vector with the weights of the model.

We apply this model and get $g(x_i)$ — the prediction of what we think the value for x_i should be. Linear regression is trained to predict the target variable y_i — the actual value of the observation i. In the previous chapter, this was the price of a car.

Linear regression is a linear model. It's called *linear* because it combines the weights of the model with the feature vector *linearly*, using the dot product. Linear models are simple to implement, train, and use. Because of their simplicity, they are also fast.

Logistic regression is also a linear model, but unlike linear regression, it's a classification model, not regression, even though the name might suggest that. It's a binary classification model, so the target variable y_i is binary; the only values it can have are zero and one. Observations with $y_i = 1$ are typically called *positive examples*: examples in which the effect we want to predict is present. Likewise, examples with $y_i = 0$ are called *negative examples*: the effect we want to predict is absent. For our project, $y_i = 1$ means that the customer churned, and $y_i = 0$ means the opposite: the customer stayed with us.

The output of logistic regression is probability — the probability that the observation x_i is positive, or, in other words, the probability that $y_i = 1$. For our case, it's the probability that the customer i will churn.

To be able to treat the output as a probability, we need to make sure that the predictions of the model always stay between zero and one. We use a special mathematical function for this purpose called *sigmoid*, and the full formula for the logistic regression model is

$$g(x_i) = \text{sigmoid}(w_0 + x_i^T w)$$

If we compare it with the linear regression formula, the only difference is this sigmoid function: in case of linear regression, we have only $w_0 + x_i^T w$. This is why both of these models are linear; they are both based on the dot product operation.

The sigmoid function maps any value to a number between zero and one (figure 3.24). It's defined this way:

$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$

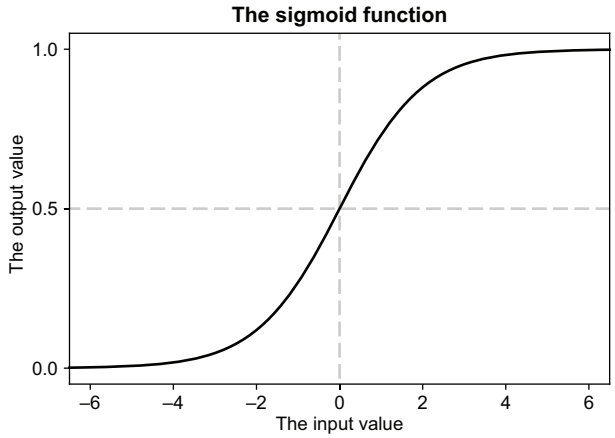


Figure 3.24 The sigmoid function outputs values that are always between 0 and 1. When the input is 0, the result of sigmoid is 0.5; for negative values, the results are below 0.5 and start approaching 0 for input values less than –6. When the input is positive, the result of sigmoid is above 0.5 and approaches 1 for input values starting from 6.

We know from chapter 2 that if the feature vector x_i is n-dimensional, the dot product $x_i^T w$ can be unwrapped as a sum, and we can write $g(x_i)$ as

$$g(x_i) = \text{sigmoid}(w_0 + x_{i1}w_1 + x_{i2}w_2 + ... + x_{in}w_n)$$

Or, using sum notation, as

$$g(x_i) = \text{sigmoid}(w_0 + \sum_{j=1}^{n} x_{ij}w_j)$$

Previously, we translated the formulas to Python for illustration. Let's do the same here.

The linear regression model has the following formula:

$$g(x_i) = w_0 + \sum_{j=1}^{n} x_{ij}w_j$$

If you remember from the previous chapter, this formula translates to the following Python code:

```
def linear_regression(xi):
    result = bias
```

```
    for j in range(n):
        result = result + xi[j] * w[j]
    return result
```

The translation of the logistic regression formula to Python is almost identical to the linear regression case, except that at the end, we apply the sigmoid function:

```
def logistic_regression(xi):
    score = bias
    for j in range(n):
        score = score + xi[j] * w[j]
    prob = sigmoid(score)
    return prob
```

Of course, we also need to define the sigmoid function:

```
import math

def sigmoid(score):
    return 1 / (1 + math.exp(-score))
```

We use *score* to mean the intermediate result before applying the sigmoid function. The score can take any real value. The *probability* is the result of applying the sigmoid function to the score; this is the final output, and it can take only the values between zero and one.

The parameters of the logistic regression model are the same as for linear regression:

- w_0 is the bias term.
- $w = (w_1, w_2, \ldots, w_n)$ is the weights vector.

To learn the weights, we need to train the model, which we will do now using Scikit-learn.

Exercise 3.3

Why do we need sigmoid for logistic regression?

a. Sigmoid converts the output to values between –6 and 6, which is easier to deal with.
b. Sigmoid makes sure the output is between zero and one, which can be interpreted as probability.

3.3.2 Training logistic regression

To get started, we first import the model:

```
from sklearn.linear_model import LogisticRegression
```

Then we train it by calling the `fit` method:

```
model = LogisticRegression(solver='liblinear', random_state=1)
model.fit(X_train, y_train)
```

The class `LogisticRegression` from Scikit-learn encapsulates the training logic behind this model. It's configurable, and we can change quite a few parameters. In fact, we already specify two of them: `solver` and `random_state`. Both are needed for reproducibility:

- `random_state`. The seed number for the random-number generator. It shuffles the data when training the model; to make sure the shuffle is the same every time, we fix the seed.
- `solver`. The underlying optimization library. In the current version (at the moment of writing, v0.20.3), the default value for this parameter is `liblinear`, but according to the documentation (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html), it will change to a different one in version v0.22. To make sure our results are reproducible in the later versions, we also set this parameter.

Other useful parameters for the model include `C`, which controls the regularization level. We talk about it in the next chapter when we cover parameter tuning. Specifying `C` is optional; by default, it gets the value 1.0.

The training takes a few seconds, and when it's done, the model is ready to make predictions. Let's see how well the model performs. We can apply it to our validation data to obtain the probability of churn for each customer in the validation dataset.

To do that, we need to apply the one-hot encoding scheme to all the categorical variables. First, we convert the dataframe to a list of dictionaries and then feed it to the `DictVectorizer` we fit previously:

```
val_dict = df_val[categorical + numerical].to_dict(orient='records')
X_val = dv.transform(val_dict)
```

We perform one-hot encoding in exactly the same way as during training.

Instead of fit and then transform, we use transform, which we fit previously.

As a result, we get `X_val`, a matrix with features from the validation dataset. Now we are ready to put this matrix to the model. To get the probabilities, we use the `predict_proba` method of the model:

```
y_pred = model.predict_proba(X_val)
```

The result of `predict_proba` is a two-dimensional NumPy array, or a two-column matrix. The first column of the array contains the probability that the target is negative (no churn), and the second column contains the probability that the target is positive (churn) (figure 3.25).

These columns convey the same information. We know the probability of churn — it's p — and the probability of not churning is always $1 - p$, so we don't need both columns.

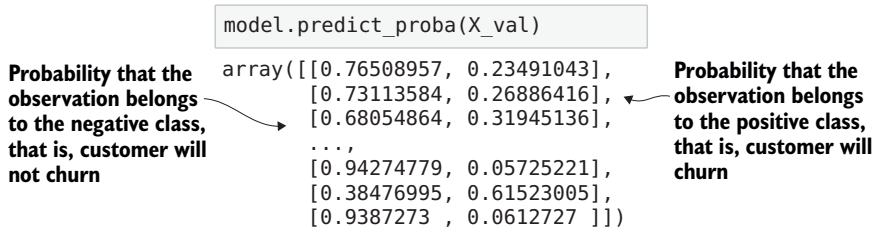


Figure 3.25 The predictions of the model: a two-column matrix. The first column contains the probability that the target is zero (the client won't churn). The second column contains the opposite probability (the target is one, and the client will churn).

Thus, it's enough to take only the second column of the prediction. To select only one column from a two-dimensional array in NumPy, we can use the slicing operation `[:, 1]`:

```
y_pred = model.predict_proba(X_val)[:, 1]
```

This syntax might be confusing, so let's break it down. Two positions are inside the brackets, the first one for rows and the second one for columns.

When we use `[:, 1]`, NumPy interprets it this way:

- `:` means select all the rows.
- `1` means select only the column at index 1, and because the indexing starts at 0, it's the second column.

As a result, we get a one-dimensional NumPy array that contains the values from the second column only.

This output (probabilities) is often called *soft* predictions. These tell us the probability of churning as a number between zero and one. It's up to us to decide how to interpret this number and how to use it.

Remember how we wanted to use this model: we wanted to retain customers by identifying those who are about to cancel their contract with the company and send them promotional messages, offering discounts and other benefits. We do this in the hope that after receiving the benefit, they will stay with the company. On the other hand, we don't want to give promotions to all our customers, because it will hurt us financially: we will make less profit, if any.

To make the actual decision about whether to send a promotional letter to our customers, using the probability alone is not enough. We need *hard* predictions — binary values of `True` (churn, so send the mail) or `False` (not churn, so don't send the mail).

To get the binary predictions, we take the probabilities and cut them above a certain threshold. If the probability for a customer is higher than this threshold, we predict churn, otherwise, not churn. If we select 0.5 to be this threshold, making the binary predictions is easy. We just use the "`>=`" operator:

```
y_pred >= 0.5
```

The comparison operators in NumPy are applied element-wise, and the result is a new array that contains only Boolean values: `True` and `False`. Under the hood, it performs the comparison for each element of the `y_pred` array. If the element is greater than 0.5 or equal to 0.5, the corresponding element in the output array is `True`, and otherwise, it's `False` (figure 3.26).

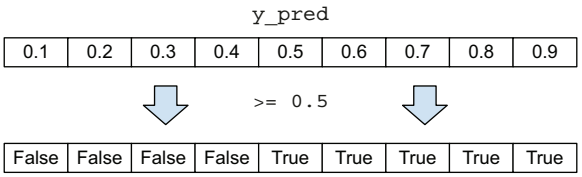


Figure 3.26 The >= operator is applied element-wise in NumPy. For every element, it performs the comparison, and the result is another array with `True` or `False` values, depending on the result of the comparison.

Let's write the results to the `churn` array:

```
churn = y_pred >= 0.5
```

When we have these hard predictions made by our model, we would like to understand how good they are, so we are ready to move to the next step: evaluating the quality of these predictions. In the next chapter, we will spend a lot more time learning about different evaluation techniques for binary classification, but for now, let's do a simple check to make sure our model learned something useful.

The simplest thing to check is to take each prediction and compare it with the actual value. If we predict churn and the actual value is churn, or we predict non-churn and the actual value is non-churn, our model made the correct prediction. If the predictions don't match, they aren't good. If we calculate the number of times our predictions match the actual value, we can use it for measuring the quality of our model.

This quality measure is called *accuracy*. It's very easy to calculate accuracy with NumPy:

```
(y_val == churn).mean()
```

Even though it's easy to calculate, it might be difficult to understand what this expression does when you see it for the first time. Let's try to break it down into individual steps.

First, we apply the `==` operator to compare two NumPy arrays: `y_val` and `churn`. If you remember, the first array, `y_val`, contains only numbers: zeros and ones. This is our target variable: one if the customer churned and zero otherwise. The second array contains Boolean predictions: `True` and `False` values. In this case `True` means

we predict the customer will churn, and `False` means the customer will not churn (figure 3.27).

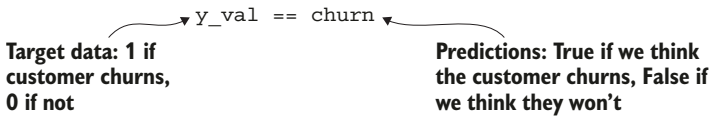


Figure 3.27 Applying the == operator to compare the target data with our predictions

Even though these two arrays have different types inside (integer and Boolean), it's still possible to compare them. The Boolean array is cast to integer such that `True` values are turned to "1" and `False` values are turned to "0." Then it's possible for NumPy to perform the actual comparison (figure 3.28).

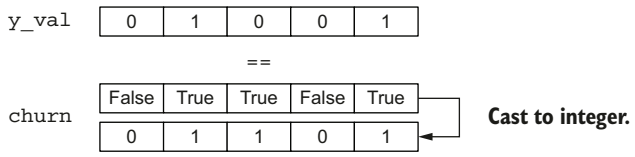


Figure 3.28 To compare the prediction with the target data, the array with predictions is cast to integer.

Like the `>=` operator, the `==` operator is applied element-wise. In this case, however, we have two arrays to compare, and here, we compare each element of one array with the respective element of the other array. The result is again a Boolean array with `True` or `False` values, depending on the outcome of the comparison (figure 3.29).

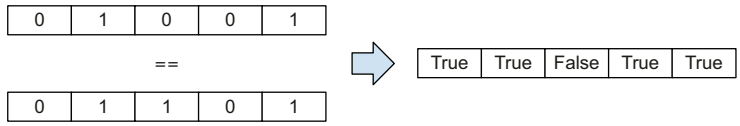


Figure 3.29 The == operator from NumPy is applied element-wise for two NumPy arrays.

In our case, if the true value in `y_pred` matches our prediction in churn, the label is `True`, and if it doesn't, the label is `False`. In other words, we have `True` if our prediction is correct and `False` if it's not.

Finally, we take the results of comparison — the Boolean array — and compute its mean using the `mean()` method. This method, however, is applied to numbers, not

Boolean values, so before calculating the mean, the values are cast to integers: `True` values to "1" and `False` values to "0" (figure 3.30).

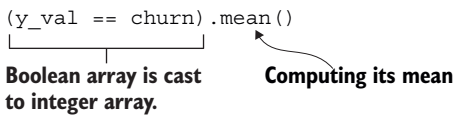


Figure 3.30 When computing the mean of a Boolean array, NumPy first casts it to integers and then computes the mean.

Finally, as we already know, if we compute the mean of an array that contains only ones and zeros, the result is the fraction of ones in that array, which we already used for calculating the churn rate. Because "1" (`True`) in this case is a correct prediction and "0" (`False`) is an incorrect prediction, the resulting number tells us the percentage of correct predictions.

After executing this line of code, we see 0.8 in output. This means that the model predictions matched the actual value 80% of the time, or the model makes correct predictions in 80% of cases. This is what we call the accuracy of the model.

Now we know how to train a model and evaluate its accuracy, but it's still useful to understand how it makes the predictions. In the next section, we try to look inside the models and see how we can interpret the coefficients it learned.

3.3.3 Model interpretation

We know that the logistic regression model has two parameters that it learns from data:

- w_0 is the bias term.
- $w = (w_1, w_2, \ldots, w_n)$ is the weights vector.

We can get the bias term from `model.intercept_[0]`. When we train our model on all features, the bias term is –0.12.

The rest of the weights are stored in `model.coef_[0]`. If we look inside, it's just an array of numbers, which is hard to understand on its own.

To see which feature is associated with each weight, let's use the `get_feature_names` method of the `DictVectorizer`. We can zip the feature names together with the coefficients before looking at them:

```
dict(zip(dv.get_feature_names(), model.coef_[0].round(3)))
```

This prints

```
{'contract=month-to-month': 0.563,
 'contract=one_year': -0.086,
 'contract=two_year': -0.599,
 'dependents=no': -0.03,
 'dependents=yes': -0.092,
```

```
 ... # the rest of the weights is omitted
 'tenure': -0.069,
 'totalcharges': 0.0}
```

To understand how the model works, let's consider what happens when we apply this model. To build the intuition, let's train a simpler and smaller model that uses only three variables: `contract`, `tenure`, and `totalcharges`.

The variables `tenure` and `totalcharges` are numeric so we don't need to do any additional preprocessing; we can take them as is. On the other hand, `contract` is a categorical variable, so to be able to use it, we need to apply one-hot encoding.

Let's redo the same steps we did for training, this time using a smaller set of features:

```
small_subset = ['contract', 'tenure', 'totalcharges']
train_dict_small = df_train[small_subset].to_dict(orient='records')
dv_small = DictVectorizer(sparse=False)
dv_small.fit(train_dict_small)

X_small_train = dv_small.transform(train_dict_small)
```

So as not to confuse it with the previous model, we add `small` to all the names. This way, it's clear that we use a smaller model, and it saves us from accidentally overwriting the results we already have. Additionally, we will use it to compare the quality of the small model with the full one.

Let's see which features the small model will use. For that, as previously, we use the `get_feature_names` method from `DictVectorizer`:

```
dv_small.get_feature_names()
```

It outputs the feature names:

```
['contract=month-to-month',
 'contract=one_year',
 'contract=two_year',
 'tenure',
 'totalcharges']
```

There are five features. As expected, we have `tenure` and `totalcharges`, and because they are numeric, their names are not changed.

As for the `contract` variable, it's categorical, so `DictVectorizer` applies the one-hot encoding scheme to convert it to numbers. `contract` has three distinct values: month-to-month, one year, and two years. Thus, one-hot encoding scheme creates three new features: `contract=month-to-month`, `contract=one_year`, and `contract=two_years`.

Let's train the small model on this set of features:

```
model_small = LogisticRegression(solver='liblinear', random_state=1)
model_small.fit(X_small_train, y_train)
```

The model is ready after a few seconds, and we can look inside the weights it learned. Let's first check the bias term:

```
model_small.intercept_[0]
```

It outputs –0.638. Then we can check the other weights, using the same code as previously:

```
dict(zip(dv_small.get_feature_names(), model_small.coef_[0].round(3)))
```

This line of code shows the weight for each feature:

```
{'contract=month-to-month': 0.91,
 'contract=one_year': -0.144,
 'contract=two_year': -1.404,
 'tenure': -0.097,
 'totalcharges': 0.000}
```

Let's put all these weights together in one table and call them w_1, w_2, w_3, w_4, and w_5 (table 3.2).

Table 3.2 The weights of a logistic regression model

Bias	contract			tenure	charges
	month	year	2-year		
w_0	w_1	w_2	w_3	w_4	w_5
—0.639	0.91	—0.144	—1.404	—0.097	0.0

Now let's take a look at these weights and try to understand what they mean and how we can interpret them.

First, let's think about the bias term and what it means. Recall that in the case of linear regression, it's the baseline prediction: the prediction we would make without knowing anything else about the observation. In the car price prediction project, it would be the price of a car on average. This is not the final prediction; later, this baseline is corrected with other weights.

In the case of logistic regression, it's similar: it's the baseline prediction — or the score we would make on average. Likewise, we later correct this score with the other weights. However, for logistic regression, interpretation is a bit trickier because we also need to apply the sigmoid function before we get the final output. Let's consider an example to help us understand that.

In our case, the bias term has the value of –0.639. This value is negative. If we look at the sigmoid function, we can see that for negative values, the output is lower than 0.5 (figure 3.31). For –0.639, the resulting probability of churning is 34%. This means that on average, a customer is more likely to stay with us than churn.

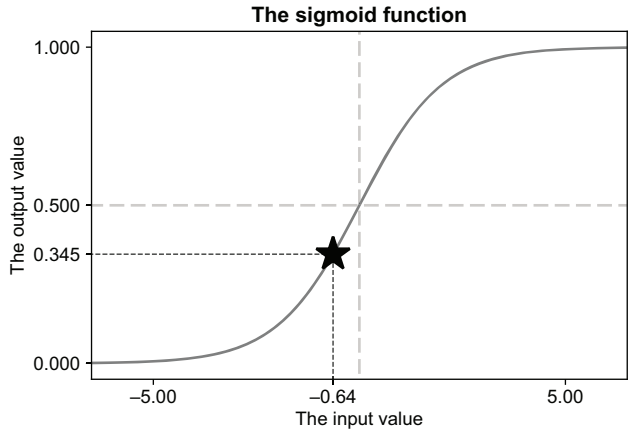


Figure 3.31 The bias term –0.639 on the sigmoid curve. The resulting probability is less than 0.5, so the average customer is more likely to not churn.

The reason why the sign before the bias term is negative is the class imbalance. There are a lot fewer churned users in the training data than non-churned ones, meaning the probability of churn on average is low, so this value for the bias term makes sense.

The next three weights are the weights for the contract variable. Because we use one-hot encoding, we have three `contract` features and three weights, one for each feature:

```
'contract=month-to-month': 0.91,
'contract=one_year': -0.144,
'contract=two_year': -1.404.
```

To build our intuition on how one-hot encoded weights can be understood and interpreted, let's think of a client with a month-to-month contract. The `contract` variable has the following one-hot encoding representation: the first position corresponds to the month-to-month value and is hot, so it's set to "1." The remaining positions correspond to one_year and two_years, so they are cold and set to "0" (figure 3.32).

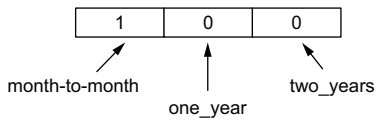


Figure 3.32 The one-hot encoding representation for a customer with a month-to-month contract

We also know the weights w_1, w_2, and w_3 that correspond to `contract=month-to-month`, `contract=one_year`, and `contract=two_years` (figure 3.33).

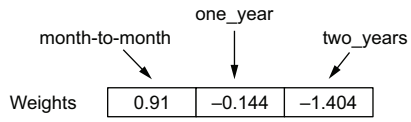


Figure 3.33 The weights of the contract=month-to-month, contract=one_year, and contract=two_years features

To make a prediction, we perform the dot product between the feature vector and the weights, which is multiplication of the values in each position and then summation. The result of the multiplication is 0.91, which turns out to be the same as the weight of the `contract=month-to-month` feature (figure 3.34).

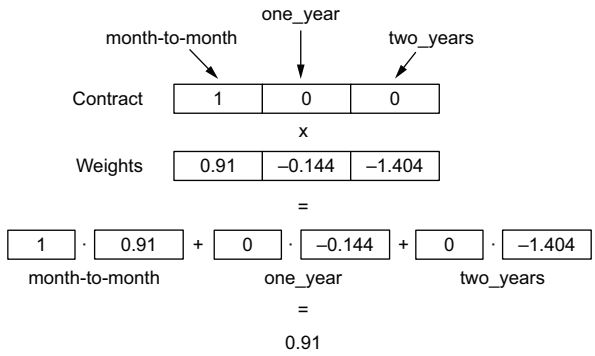


Figure 3.34 The dot product between the one-hot encoding representation of the contract variable and the corresponding weights. The result is 0.91, which is the weight of the hot feature.

Let's consider another example: a client with a two-year contract. In this case, the `contract=two_year` feature is hot and has a value of "1," and the rest are cold. When we multiply the vector with the one-hot encoding representation of the variable by the weight vector, we get –1.404 (figure 3.35).

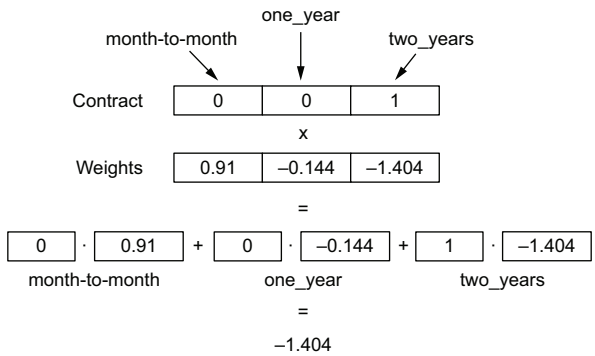


Figure 3.35 For a customer with a two-year contract, the result of the dot product is –1.404.

As we see, during the prediction, only the weight of the hot feature is taken into account, and the rest of the weights are not considered in calculating the score. This makes sense: the cold features have values of zero, and when we multiply by zero, we get zero again (figure 3.36).

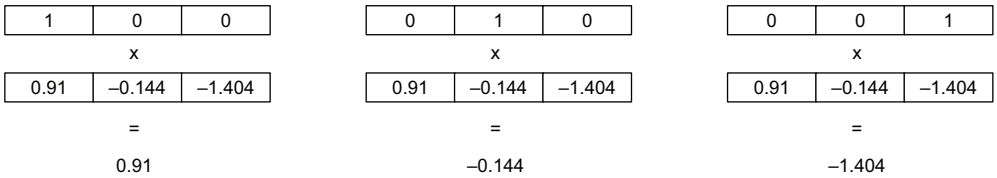


Figure 3.36 When we multiply the one-hot encoding representation of a variable by the weight vector from the model, the result is the weight corresponding to the hot feature.

The interpretation of the signs of the weights for one-hot encoded features follows the same intuition as the bias term. If a weight is positive, the respective feature is an indicator of churn, and vice versa. If it's negative, it's more likely to belong to a non-churning customer.

Let's look again at the weights of the `contract` variable. The first weight for `contract=month-to-month` is positive, so customers with this type of contract are more likely to churn than not. The other two features, `contract=one_year` and `contract=two_years`, have negative signs, so such clients are more likely to remain loyal to the company (figure 3.37).

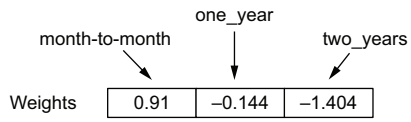


Figure 3.37 The sign of the weight matters. If it's positive, it's a good indicator of churn; if it's negative, it indicates a loyal customer.

The magnitude of the weights also matters. For `two_year`, the weight is –1.404, which is greater in magnitude than –0.144 — the weight for `one_year`. So, a two-year contract is a stronger indicator of not churning than a one-year one. It confirms the feature importance analysis we did previously. The risk ratios (the risk of churning) for this set of features are 1.55 for monthly, 0.44 for one-year, and 0.10 for two-year (figure 3.38).

Now let's have a look at the numerical features. We have two of them: `tenure` and `totalcharges`. The weight of the `tenure` feature is –0.097, which has a negative sign. This means the same thing: the feature is an indicator of no churn. We already know

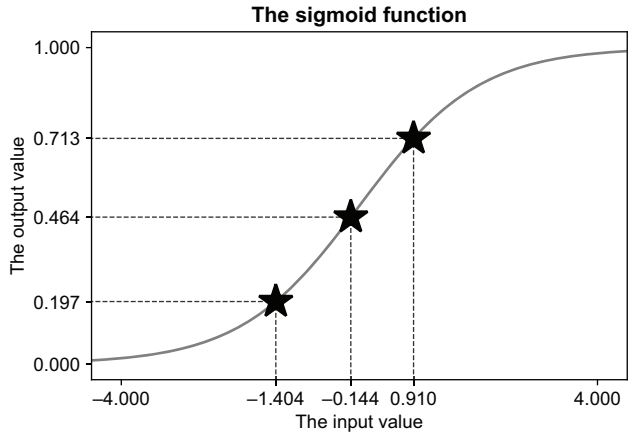


Figure 3.38 The weights for the contract features and their translation to probabilities. For `contract=two_year`, the weight is –1.404, which translates to very low probability of churn. For `contract=one_year`, the weight is –0.144, so the probability is moderate. And for `contract=month-to-month`, the weight is 0.910, and the probability is quite high.

from the feature importance analysis that the longer clients stay with us, the less likely they are to churn. The correlation between `tenure` and churn is –0.35, which is also a negative number. The weight of this feature confirms it: for every month that the client spends with us, the total score gets lower by 0.097.

The other numerical feature, `totalchanges`, has weight of zero. Because it's zero, no matter what the value of this feature is, the model will never consider it, so this feature is not really important for making the predictions.

To understand it better, let's consider a couple of examples. For the first example, let's imagine we have a user with a month-to-month contract, who spent a year with us and paid $1,000 (figure 3.39).

$$\underset{\text{Bias}}{-0.639} + \underset{\substack{\text{Monthly}\\\text{contract}}}{0.91} - \underset{\substack{\text{12 months}\\\text{of tenure}}}{12 \cdot 0.097} + \underset{\substack{\text{Total charges}\\\text{don't matter.}}}{0 \cdot 1000} = \underset{\substack{\text{Negative, so low}\\\text{likelihood of churn}}}{-0.893}$$

Figure 3.39 The score the model calculates for a customer with a month-to-month contract and 12 months of tenure

This is the prediction we make for this customer:

- We start with the baseline score. It's the bias term with the value of –0.639.
- Because it's a month-to-month contract, we add 0.91 to this value and get 0.271. Now the score becomes positive, so it may mean that the client is going to churn. We know that a monthly contract is a strong indicator of churning.

- Next, we consider the `tenure` variable. For each month that the customer stayed with us, we subtract 0.097 from the score so far. Thus, we get $0.271 - 12 \cdot 0.097 = -0.893$. Now the score is negative again, so the likelihood of churn decreases.
- Now we add the amount of money the customer paid us (`totalcharges`) multiplied by the weight of this feature, but because it's zero, we don't do anything. The result stays –0.893.
- The final score is a negative number, so we believe that the customer is not very likely to churn soon.
- To see the actual probability of churn, we compute the sigmoid of the score, and it's approximately 0.29. We can treat this as the probability that this customer will churn.

If we have another client with a yearly contract who stayed 24 months with us and spent $2,000, the score is –2.823 (figure 3.40).

$$-0.639 + 0.144 - 24 \cdot 0.097 + 0 \cdot 2000 = -2.823$$

Bias	Yearly contract	24 months of tenure	Total charges don't matter.	Negative, very low likelihood of churn

Figure 3.40 The score that the model calculates for a customer with a yearly contract and 24 months of tenure

After taking sigmoid, the score of –2.823 becomes 0.056, so the probability of churn for this customer is even lower (figure 3.41).

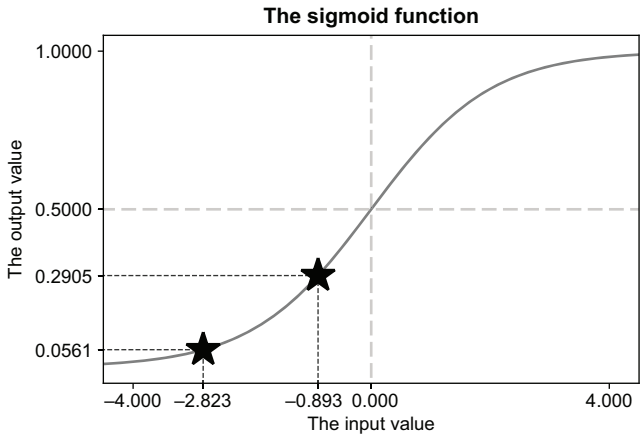


Figure 3.41 The scores of –2.823 and –0.893 translated to probability: 0.05 and 0.29, respectively

3.3.4 Using the model

Now we know a lot better how logistic regression, and we can also interpret what our model learned and understand how it makes the predictions.

Additionally, we applied the model to the validation set, computed the probabilities of churning for every customer there, and concluded that the model is 80% accurate. In the next chapter we will evaluate whether this number is satisfactory, but for now, let's try to use the model we trained. Now we can apply the model to customers for scoring them. It's quite easy.

First, we take a customer we want to score and put all the variable values in a dictionary:

```
customer = {
    'customerid': '8879-zkjof',
    'gender': 'female',
    'seniorcitizen': 0,
    'partner': 'no',
    'dependents': 'no',
    'tenure': 41,
    'phoneservice': 'yes',
    'multiplelines': 'no',
    'internetservice': 'dsl',
    'onlinesecurity': 'yes',
    'onlinebackup': 'no',
    'deviceprotection': 'yes',
    'techsupport': 'yes',
    'streamingtv': 'yes',
    'streamingmovies': 'yes',
    'contract': 'one_year',
    'paperlessbilling': 'yes',
    'paymentmethod': 'bank_transfer_(automatic)',
    'monthlycharges': 79.85,
    'totalcharges': 3320.75,
}
```

> **NOTE** When we prepare items for prediction, they should undergo the same preprocessing steps we did for training the model. If we don't do it in exactly the same way, the model might not get things it expects to see, and, in this case, the predictions could get really off. This is why in the previous example, in the `customer` dictionary, the field names and string values are lowercased and spaces are replaced with underscores.

Now we can use our model to see whether this customer is going to churn. Let's do it.

First, we convert this dictionary to a matrix by using the `DictVectorizer`:

```
X_test = dv.transform([customer])
```

The input to the vectorizer is a list with one item: we want to score only one customer. The output is a matrix with features, and this matrix contains only one row — the features for this one customer:

```
[[   0.  ,    1.  ,    0.  ,    1.  ,    0.  ,    0.  ,    0.  ,
     1.  ,    1.  ,    0.  ,    1.  ,    0.  ,    0.  ,   79.85,
     1.  ,    0.  ,    0.  ,    1.  ,    0.  ,    0.  ,    0.  ,
     0.  ,    1.  ,    0.  ,    1.  ,    1.  ,    0.  ,    1.  ,
     0.  ,    0.  ,    0.  ,    0.  ,    1.  ,    0.  ,    0.  ,
     0.  ,    1.  ,    0.  ,    0.  ,    1.  ,    0.  ,    0.  ,
     1.  ,   41.  , 3320.75]]
```

We see a bunch of one-hot encoding features (ones and zeros) as well as some numeric ones (`monthlycharges`, `tenure`, and `totalcharges`).

Now we take this matrix and put it into the trained model:

```
model.predict_proba(X_test)
```

The output is a matrix with predictions. For each customer, it outputs two numbers, which are the probability of staying with the company and the probability of churn. Because there's only one customer, we get a tiny NumPy array with one row and two columns:

```
[[0.93, 0.07]]
```

All we need from the matrix is the number at the first row and second column: the probability of churning for this customer. To select this number from the array, we use the brackets operator:

```
model.predict_proba(X_test)[0, 1]
```

We used this operator to select the second column from the array. However, this time there's only one row, so we can explicitly ask NumPy to return the value from that row. Because indexes start from 0 in NumPy, `[0, 1]` means first row, second column.

When we execute this line, we see that the output is 0.073, so that the probability that this customer will churn is only 7%. It's less than 50%, so we will not send this customer a promotional mail.

We can try to score another client:

```
customer = {
    'gender': 'female',
    'seniorcitizen': 1,
    'partner': 'no',
    'dependents': 'no',
    'phoneservice': 'yes',
    'multiplelines': 'yes',
    'internetservice': 'fiber_optic',
    'onlinesecurity': 'no',
    'onlinebackup': 'no',
    'deviceprotection': 'no',
    'techsupport': 'no',
    'streamingtv': 'yes',
    'streamingmovies': 'no',
    'contract': 'month-to-month',
```

```
    'paperlessbilling': 'yes',
    'paymentmethod': 'electronic_check',
    'tenure': 1,
    'monthlycharges': 85.7,
    'totalcharges': 85.7
}
```

Let's make a prediction:

```
X_test = dv.transform([customer])
model.predict_proba(X_test)[0, 1]
```

The output of the model is 83% likelihood of churn, so we should send this client a promotional mail in the hope of retaining them.

So far, we've built intuition on how logistic regression works, how to train it with Scikit-learn, and how to apply it to new data. We haven't covered the evaluation of the results yet; this is what we will do in the next chapter.

3.4 Next steps

3.4.1 Exercises

You can try a couple of things to learn the topic better:

- In the previous chapter, we implemented many things ourselves, including linear regression and dataset splitting. In this chapter we learned how to use Scikit-learn for that. Try to redo the project from the previous chapter using Scikit-learn. To use linear regression, you need `LinearRegression` from the `sklearn.linear_model` package. To use regularized regression, you need to import `Ridge` from the same package `sklearn.linear_model`.
- We looked at feature importance metrics to get some insights into the dataset but did not really use this information for other purposes. One way to use this information could be removing features that aren't useful from the dataset to make the model simpler, faster, and potentially better. Try to exclude the two least useful features (`gender` and `phoneservices`) from the training data matrix, and see what happens to validation accuracy. What if we remove the most useful feature (`contract`)?

3.4.2 Other projects

We can use classification in numerous ways to solve real-life problems, and now, after learning the materials of this chapter, you should have enough knowledge to apply logistic regression to solve similar problems. In particular, we suggest these:

- Classification models are often used for marketing purposes, and one of the problems it solves is *lead scoring*. A *lead* is a potential customer who may convert (become an actual customer) or not. In this case, the conversion is the target

we want to predict. You can take a dataset from https://www.kaggle.com/ashydv/leads-dataset and build a model for that. You may notice that the lead-scoring problem is similar to churn prediction, but in one case, we want to get a new client to sign a contract with us, and in another case, we want a client not to cancel the contract.

- Another popular application of classification is default prediction, which is estimating the risk of a customer's not paying back a loan. In this case, the variable we want to predict is default, and it also has two outcomes: whether the customer managed to pay back the loan in time (good customer) or not (default). You can find many datasets online for training a model, such as https://archive.ics.uci.edu/ml/datasets/default+of+credit+card+clients (or the same one available via Kaggle: https://www.kaggle.com/pratjain/credit-card-default).

Summary

- The *risk* of a categorical feature tells us if a group that has the feature will have the condition we model. For churn, values lower than 1.0 indicate low risk of churning, whereas values higher than 1.0 indicate high risk of churning. It tells us which features are important for predicting the target variable and helps us better understand the problem we're solving.
- Mutual information measures the degree of (in)dependence between a categorical variable and the target. It's a good way of determining important features: the higher the mutual information is, the more important the feature.
- Correlation measures the dependence between two numerical variables, and it can be used for determining if a numerical feature is useful for predicting the target variable.
- One-hot encoding gives us a way to represent categorical variables as numbers. Without it, it wouldn't be possible to easily use these variables in a model. Machine learning models typically expect all input variables to be numeric, so having an encoding scheme is crucial if we want to use categorical features in modeling.
- We can implement one-hot encoding by using `DictVectorizer` from Scikit-learn. It automatically detects categorical variables and applies the one-hot encoding scheme to them while leaving numerical variables intact. It's very convenient to use and doesn't require a lot of coding on our side.
- Logistic regression is a linear model, just like linear regression. The difference is that logistic regression has an extra step at the end: it applies the sigmoid function to convert the scores to probabilities (a number between zero and one). That allows us to use it for classification. The output is the probability of belonging to a positive class (churn, in our case).
- When the data is prepared, training logistic regression is very simple: we use the `LogisticRegression` class from Scikit-learn and invoke the `fit` function.

- The model outputs probabilities, not hard predictions. To binarize the output, we cut the predictions at a certain threshold. If the probability is greater than or equal to 0.5, we predict `True` (churn), and `False` (no churn) otherwise. This allows us to use the model for solving our problem: predicting customers who churn.
- The weights of the logistic regression model are easy to interpret and explain, especially when it comes to the categorical variables encoded using the one-hot encoding scheme. It helps us understand the behavior of the model better and explain to others what it's doing and how it's working.

In the next chapter we will continue with this project on churn prediction. We will look at ways of evaluating binary classifiers and then use this information for tuning the model's performance.

Answers to exercises

- Exercise 3.1 B) The percentage of `True` elements
- Exercise 3.2 A) It will keep a numeric variable as is and encode only the categorical variable.
- Exercise 3.3 B) Sigmoid converts the output to a value between zero and one.

4 Evaluation metrics for classification

This chapter covers

- Accuracy as a way of evaluating binary classification models and its limitations
- Determining where our model makes mistakes using a confusion table
- Deriving other metrics like precision and recall from the confusion table
- Using ROC (receiver operating characteristics) and AUC (area under the ROC curve) to further understand the performance of a binary classification model
- Cross-validating a model to make sure it behaves optimally
- Tuning the parameters of a model to achieve the best predictive performance

In this chapter, we continue with the project we started in the previous chapter: churn prediction. We have already downloaded the dataset, performed the initial preprocessing and exploratory data analysis, and trained the model that predicts

whether customers will churn. We have also evaluated this model on the validation dataset and concluded that it has 80% accuracy.

The question we postponed until now was whether 80% accuracy is good and what it actually means in terms of the quality of our model. We answer this question in this chapter and discuss other ways of evaluating a binary classification model: the confusion table, precision and recall, the ROC curve, and AUC.

This chapter provides a lot of complex information, but the evaluation metrics we cover here are essential for doing practical machine learning. Don't worry if you don't immediately understand all the details of the different evaluation metrics: it requires time and practice. Feel free to come back to this chapter to revisit the finer points.

4.1 Evaluation metrics

We have already built a binary classification model for predicting churning customers. Now we need to be able to determine how good it is.

For this, we use a *metric* — a function that looks at the predictions the model makes and compares them with the actual values. Then, based on the comparison, it calculates how good the model is. This is quite useful: we can use it to compare different models and select the one with the best metric value.

There are different kinds of metrics. In chapter 2, we used RMSE (root mean squared error) to evaluate regression models. However, this metric can be used only for regression models and doesn't work for classification.

For evaluating classification models, we have other more suitable metrics. In this section, we cover the most common evaluation metrics for binary classification, starting with accuracy, which we already saw in chapter 3.

4.1.1 Classification accuracy

As you probably remember, the accuracy of a binary classification model is the percentage of correct predictions it makes (figure 4.1).

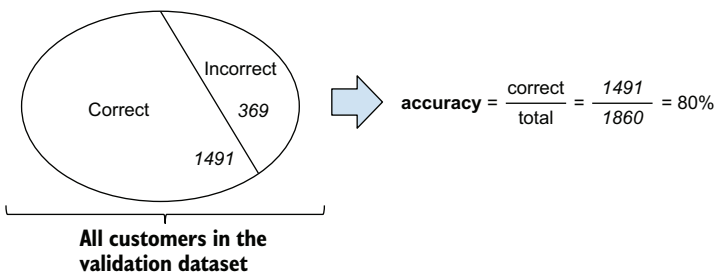


Figure 4.1 The accuracy of a model is the fraction of predictions that turned out to be correct.

This accuracy is the simplest way to evaluate a classifier: by counting the number of cases in which our model turned out to be right, we can learn a lot about the model's behavior and quality.

Computing accuracy on the validation dataset is easy — we simply calculate the fraction of correct predictions:

```
y_pred = model.predict_proba(X_val)[:, 1]
churn = y_pred >= 0.5
(churn == y_val).mean()
```

❶ Gets the predictions from the model

❷ Makes "hard" predictions

❸ Computes the accuracy

We first apply the model to the validation set to get the predictions in ❶. These predictions are probabilities, so we cut them at 0.5 in ❷. Finally, we calculate the fraction of predictions that matched reality in ❷.

The result is 0.8016, which means that our model is 80% accurate.

The first thing we should ask ourselves is why we chose 0.5 as the threshold and not any other number. That was an arbitrary choice, but it's actually not difficult to check other thresholds as well: we can just loop over all possible threshold candidates and compute the accuracy for each. Then we can choose the one with the best accuracy score.

Even though it's easy to implement accuracy ourselves, we can use existing implementations as well. The Scikit-learn library offers a variety of metrics, including accuracy and many others that we will use later. You can find these metrics in the metrics package.

We'll continue working on the same notebook that we started in chapter 3. Let's open it and add the `import` statement to import accuracy from Scikit-learn's metrics package:

```
from sklearn.metrics import accuracy_score
```

Now we can loop over different thresholds and check which one gives the best accuracy:

```
thresholds = np.linspace(0, 1, 11)

for t in thresholds:
    churn = y_pred >= t
    acc = accuracy_score(y_val, churn)
    print('%0.2f %0.3f' % (t, acc))
```

Creates an array with different thresholds: 0.0, 0.1, 0.2, and so on

Uses the accuracy_score function from Scikit-learn for computing accuracy

Loops over each threshold value

Prints the thresholds and the accuracy values to standard output

In this code, we first create an array with thresholds. We use the `linspace` function from NumPy for that: it takes two numbers (0 and 1, in our case) and the number of elements the array should have (11). As a result, we get an array with the numbers 0.0,

0.1, 0.2, …, 1.0. You can learn more about `linspace` and other NumPy functions in appendix C.

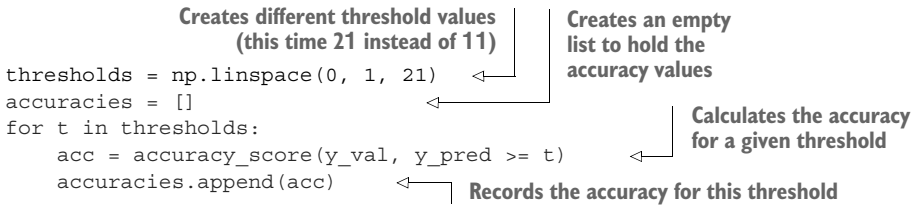
We use these numbers as thresholds: we loop over them, and for each one, we calculate the accuracy. Finally, we print the thresholds and the accuracy scores so we can decide which threshold is the best.

When we execute the code, it prints the following:

```
0.00 0.261
0.10 0.595
0.20 0.690
0.30 0.755
0.40 0.782
0.50 0.802
0.60 0.790
0.70 0.774
0.80 0.742
0.90 0.739
1.00 0.739
```

As we see, using the threshold of 0.5 gives us the best accuracy. Typically, 0.5 is a good threshold value to start with, but we should always try other threshold values to make sure 0.5 is the best choice.

To make it more visual, we can use Matplotlib to create a plot that shows how accuracy changes depending on the threshold. We repeat the same process as previously, but instead of just printing the accuracy scores, we first put the values to a list:



And then we plot these values using Matplotlib:

```
plt.plot(thresholds, accuracies)
```

After executing this line, we should see a plot that shows the relationship between the threshold and the accuracy (figure 4.2). As we already know, the 0.5 threshold is the best in terms of accuracy.

So, the best threshold is 0.5, and the best accuracy for this model that we can achieve is 80%.

In the previous chapter, we trained a simpler model: we called it model_small. It was based on only three variables: `contract`, `tenure`, and `totalcharges`.

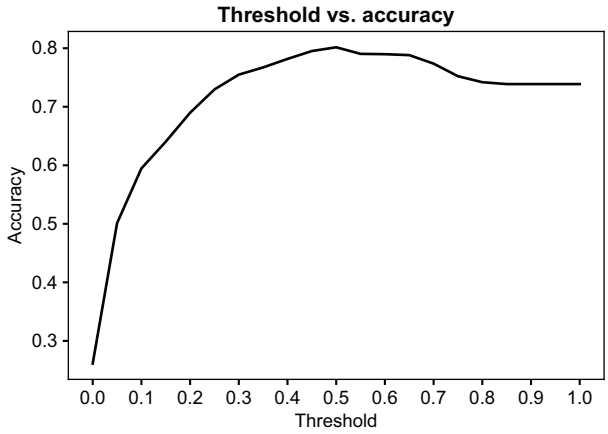


Figure 4.2 Accuracy of our model evaluated at different thresholds. The best accuracy is achieved when cutting the predictions at the 0.5 threshold: if a prediction is higher than 0.5, we predict "churn," and otherwise, we predict "no churn."

Let's also check its accuracy. For that, we first make predictions on the validation dataset and then compute the accuracy score:

```
val_dict_small = df_val[small_subset].to_dict(orient='records')

X_small_val = dv_small.transform(val_dict_small)
y_pred_small = model_small.predict_proba(X_small_val)[:, 1]

churn_small = y_pred_small >= 0.5
accuracy_score(y_val, churn_small)
```

Applies one-hot encoding to the validation data

Predicts churn using the small model

Calculates the accuracy of the predictions

When we run this code, we see that the accuracy of the small model is 76%. So, the large model is actually 4% more accurate than the small model.

However, this still doesn't tell us whether 80% (or 76%) is a good accuracy score.

4.1.2 Dummy baseline

Although it seems like a decent number, to understand whether 80% is actually good, we need to relate it to something — for example, a simple baseline that's easy to understand. One such baseline could be a dummy model that always predicts the same value.

In our example, the dataset is imbalanced, and we don't have many churned users. So, the dummy model can always predict the majority class — "no churn." In other words, this model will always output False, regardless of the features. This is not a super useful model, but we can use it as a baseline and compare it with the other two models.

Let's create this baseline prediction:

```
size_val = len(y_val)
baseline = np.repeat(False, size_val)
```

Gets the number of customers in the validation set

Creates an array with only False elements

To create an array with the baseline predictions, we first need to determine how many elements are in the validation set.

Next, we create an array of dummy predictions — all the elements of this array are False values. We do this using the `repeat` function from NumPy: it takes in an element and repeats it as many times as we ask. For more details about the `repeat` function and other NumPy functions, please refer to appendix C.

Now we can check the accuracy of this baseline prediction using the same code as we used previously:

```
accuracy_score(baseline, y_val)
```

When we run this code, it shows 0.738. This means that the accuracy of the baseline model is around 74% (figure 4.3).

```
size_val = len(y_val)
baseline = np.repeat(False, size_val)
baseline
```

```
array([False, False, False, ..., False, False, False])
```

```
accuracy_score(baseline, y_val)
```

```
0.7387096774193549
```

Figure 4.3 The baseline is a "model" that always predicts the same value for all the customers. The accuracy of this baseline is 74%.

As we see, the small model is only 2% better than the naive baseline, and the large one is 6% better. If we think about all the trouble we have gone through to train this large model, 6% doesn't seem like a significant improvement over the dummy baseline.

Churn prediction is a complex problem, and maybe this improvement is great. However, that's not evident from the accuracy score alone. According to accuracy, our model is only slightly better than a dummy model that treats all the customers as non-churning and doesn't attempt to keep any of them.

Thus, we need other metrics — other ways of measuring the quality of our model. These metrics are based on the confusion table, the concept that we cover in the next section.

4.2 Confusion table

Even though accuracy is easy to understand, it's not always the best metric. In fact, it sometimes can be misleading. We've already seen this occur: the accuracy of our model is 80%, and although that seems like a good number, it's just 6% better than the accuracy of a dummy model that always outputs the same prediction of "no churn."

This situation typically happens when we have a class imbalance (more instances of one class than another). We know that this is definitely the case for our problem: 74% of customers did not churn, and only 26% did churn.

For such cases, we need a different way of measuring the quality of our models. We have a few options, and most of them are based on the confusion table: a table that concisely represents every possible outcome for our model's predictions.

4.2.1 Introduction to the confusion table

We know that for a binary classification model, we can have only two possible predictions: True and False. In our case, we can predict that a customer is either going to churn (True) or not (False).

When we apply the model to the entire validation dataset with customers, we split it into two parts (figure 4.4):

- Customers for whom the model predicts "churn"
- Customers for whom the model predicts "no churn"

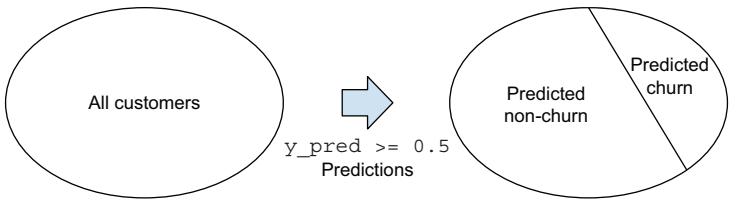


Figure 4.4 Our model splits all the customers in the validation dataset into two groups: customers who we think will churn and customers who will not.

Only two possible correct outcomes can occur: again, True or False. A customer has either actually churned (True) or not (False).

This means that by using the ground truth information — the information about the target variable — we can again split the dataset into two parts (figure 4.5):

- The customers who churned
- The customers who didn't churn

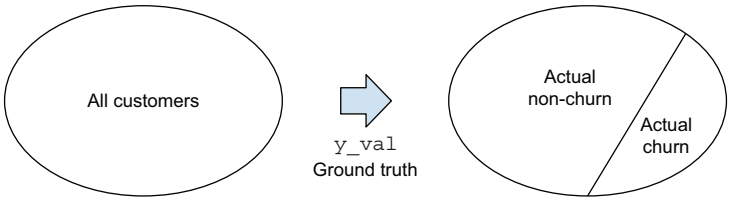


Figure 4.5 Using the ground truth data, we can split the validation dataset into two groups: customers who actually churned and customers who didn't.

When we make a prediction, it will either turn out to be correct or not:

- If we predict "churn," the customer may indeed churn, or they may not.
- If we predict "no churn," it's possible that the customer indeed doesn't churn, but it's also possible that they do churn.

This gives us four possible outcomes (figure 4.6):

- We predict False, and the answer is False.
- We predict False, and the answer is True.
- We predict True, and the answer is False.
- We predict True, and the answer is True.

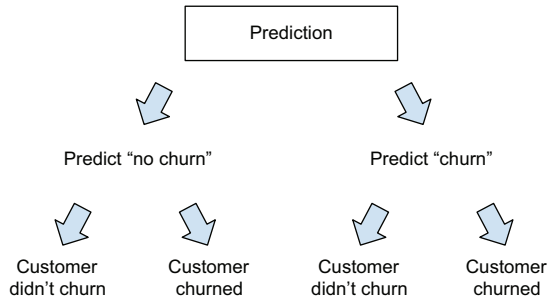


Figure 4.6 There are four possible outcomes: we predict "churn," and the customers either churn or do not, and we predict "no churn," and the customers again either churn or do not.

Two of these situations — the first and last ones — are good: the prediction matched the actual value. The two remaining ones are bad: we didn't make a correct prediction.

Each of these four situations has its own name (figure 4.7):

- True negative (TN): we predict False ("no churn"), and the actual label is also False ("no churn").
- True positive (TP): we predict True ("churn"), and the actual label is True ("churn").

- False negative (FN): we predict False ("no churn"), but it's actually True (the customer churned).
- False positive (FP): we predict True ("churn"), but it's actually False (the customer stayed with us).

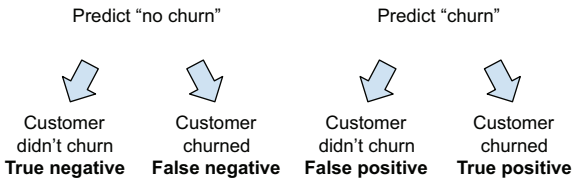


Figure 4.7 Each of the four possible outcomes has its own name: true negative, false negative, false positive, and true positive.

It's visually helpful to arrange these outcomes in a table. We can put the predicted classes (False and True) in the columns and the actual classes (False and True) in the rows (figure 4.8).

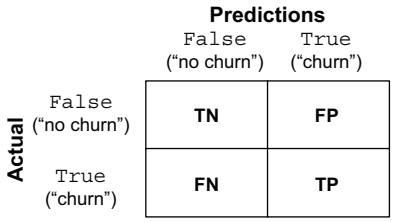


Figure 4.8 We can organize the outcomes in a table — the predicted values as columns and the actual values as rows. This way, we break down all prediction scenarios into four distinct groups: TN (true negative), TP (true positive), FN (false negative), and FP (false positive).

When we substitute the number of times each outcome happens, we get the confusion table for our model (figure 4.9).

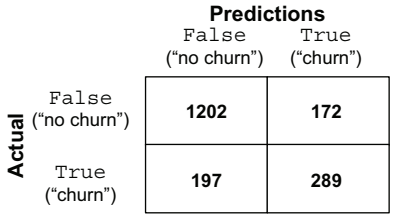


Figure 4.9 In the confusion table, each cell contains the number of times each outcome happens.

Calculating the values in the cells of the confusion matrix is quite easy with NumPy. Next, we see how to do it.

4.2.2 Calculating the confusion table with NumPy

To help us understand our confusion table better, we can visually depict what it does to the validation dataset (figure 4.10).

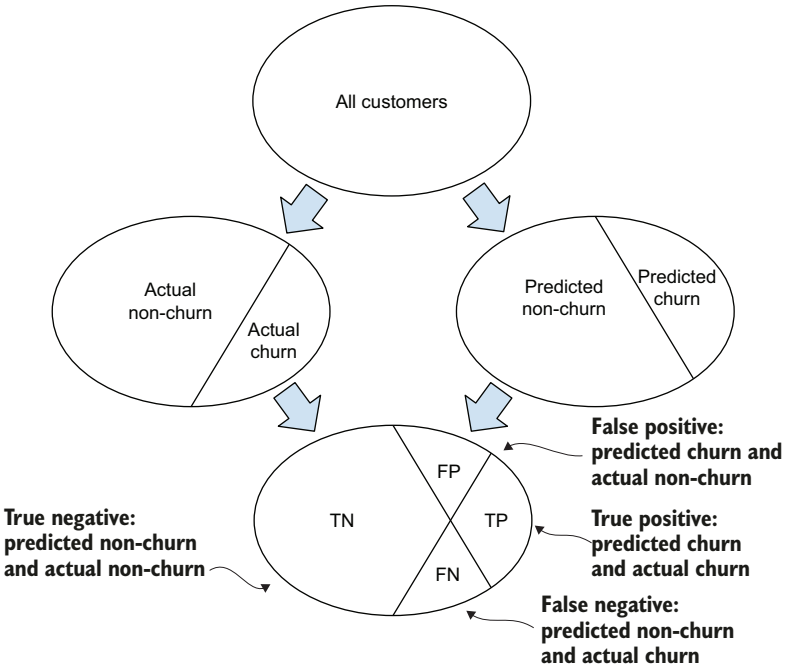


Figure 4.10 When we apply the model to the validation dataset, we get four different outcomes (TN, FP, TP, and FN).

To calculate the confusion table, we need to do these steps:

- First, the predictions split the dataset into two parts: the part for which we predict True ("churn") and the part for which we predict False ("no churn").
- At the same time, the target variable splits this dataset into two different parts: the customers who actually churned ("1" in `y_val`) and the customers who didn't ("0" in `y_val`).
- When we combine these splits, we get four groups of customers, which are exactly the four different outcomes from the confusion table.

Translating these steps to NumPy is straightforward:

```
t = 0.5
predict_churn = (y_pred >= t)
predict_no_churn = (y_pred < t)

actual_churn = (y_val == 1)
actual_no_churn = (y_val == 0)

true_positive = (predict_churn & actual_churn).sum()
false_positive = (predict_churn & actual_no_churn).sum()

false_negative = (predict_no_churn & actual_churn).sum()
true_negative = (predict_no_churn & actual_no_churn).sum()
```

1. Makes predictions at threshold 0.5
2. Gets the actual target values
3. Calculates true positives (cases when we predicted churn correctly)
4. Calculates false positives (cases when we predicted churn, but the customers didn't churn)
5. Calculates true negatives (cases when we predicted no churn correctly)
6. Calculates false negatives (cases when we predicted no churn, but the customers churned)

We begin by making predictions at the threshold of 0.5.

The results are two NumPy arrays:

- In the first array (`predict_churn`), an element is True if the model thinks the respective customer is going to churn and False otherwise.
- Likewise, in the second array (`predict_no_churn`), True means that the model thinks the customer isn't going to churn.

The second array, `predict_no_churn`, is the exact opposite of `predict_churn`: if an element is True in `predict_churn`, it's False in `predict_no_churn` and vice versa (figure 4.11). This is the first split of the validation dataset into two parts — the one that's based on the predictions.

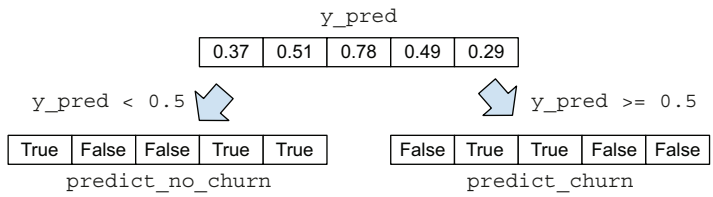


Figure 4.11 Splitting the predictions into two Boolean NumPy arrays: `predict_churn` if the probability is higher than 0.5, and `predict_no_churn` if it's lower

Next, we record the actual values of the target variable in 2. The results are two NumPy arrays as well (figure 4.12):

- If the customer churned (value "1"), then the respective element of `actual_churn` is True, and it's False otherwise.
- For `actual_no_churn` it's exactly the opposite: it's True when the customer didn't churn.

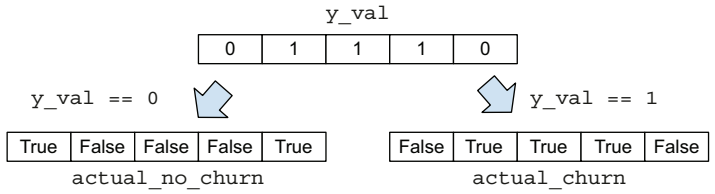


Figure 4.12 Splitting the array with actual values into two Boolean NumPy arrays: `actual_no_churn` if the customer didn't churn (`y_val == 0`) and `actual_churn` if the customer churned (`y_val == 1`)

That's the second split of the dataset — the one that's based on the target variable.

Now we combine these two splits — or, to be exact, these four NumPy arrays.

To calculate the number of true positive outcomes in ❸, we use the logical "and" operator of NumPy (`&`) and the `sum` method:

```
true_positive = (predict_churn & actual_churn).sum()
```

The logical "and" operator evaluates to True only if both values are True. If at least one is False or both are False, it's False. In case of `true_positive`, it will be True only if we predict "churn" and the customer actually churned (figure 4.13).

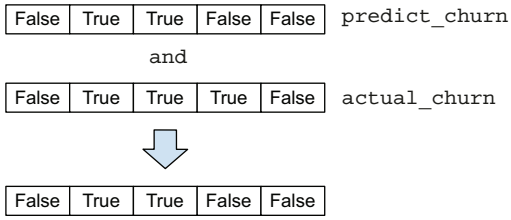


Figure 4.13 Applying the element-wise and operator (`&`) to two NumPy arrays, `predict_churn` and `actual_churn`; this creates another array with True in any position where both arrays contained True and False in all others.

Then we use the `sum` method from NumPy, which simply counts how many `True` values are in the array. It does that by first casting the Boolean array to integers and then summing it (figure 4.14). We already saw similar behavior in the previous chapter when we used the `mean` method.

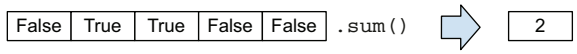


Figure 4.14 Invoking the `sum` method on a Boolean array: we get the number of elements in this array that are True.

As a result, we have the number of true positive cases. The other values are computed similarly in lines ❹, ❺, and ❻.

Now we just need to put all these values together in a NumPy array:

```
confusion_table = np.array(
 [[true_negative, false_positive],
 [false_negative, true_positive]])
```

When we print it, we get the following numbers:

```
[[1202, 172],
 [ 197, 289]]
```

The absolute numbers may be difficult to understand, so we can turn them into fractions by dividing each value by the total number of items:

```
confusion_table / confusion_table.sum()
```

This prints the following numbers:

```
[[0.646, 0.092],
[0.105, 0.155]]
```

We can summarize the results in a table (table 4.1). We see that the model predicts negative values quite well: 65% of the predictions are true negatives. However, it makes quite a few mistakes of both types: the number of false positives and false negatives is roughly equal (9% and 11%, respectively).

Table 4.1 The confusion table for the churn classifier at the threshold of 0.5. We see that it's easy for the model to correctly predict non-churning users, but it's more difficult for it to identify churning users.

Full model with all features			
		Predicted	
		False	True
Actual	False	1202 (65%)	172 (9%)
	True	197 (11%)	289 (15%)

This table gives us a better understanding of the performance of the model — it's now possible to break down the performance into different components and understand where the model makes mistakes. We actually see that the performance of the model is not great: it makes quite a few errors when trying to identify users that will churn. This is something we couldn't see with the accuracy score alone.

We can repeat the same process for the small model using exactly the same code (table 4.2).

Table 4.2 The confusion table for the small model

Small model with three features			
		Predicted	
		False	True
Actual	False	1189 (63%)	185 (10%)
	True	248 (12%)	238 (13%)

When we compare the smaller model with the full model, we see that it's 2% worse at correctly identifying non-churning users (63% versus 65% for true negatives) and 2% worse at correctly identifying churning users (13% versus 15% for true positives), which together accounts for the 4% difference between the accuracies of these two models (76% versus 80%).

The values from the confusion table serve as the basis for many other evaluation metrics. For example, we can calculate accuracy by taking all the correct predictions — TN and TP together — and dividing that number by the total number of observations in all four cells of the table:

$$\text{accuracy} = (TN + TP) \;/\; (TN + TP + FN + FP)$$

Apart from accuracy, we can calculate other metrics based on the values from the confusion table. The most useful ones are precision and recall, which we will cover next.

Exercise 4.1

What is a false positive?

a A customer for whom we predicted "not churn," but they stopped using our services
b A customer for whom we predicted "churn," but they didn't churn
c A customer for whom we predicted "churn," and they churned

4.2.3 Precision and recall

As already mentioned, accuracy can be misleading when dealing with imbalanced datasets such as ours. Other metrics are helpful to use for such cases: precision and recall.

Both precision and recall are calculated from the values of the confusion table. They both help us understand the quality of the model in cases of class imbalance.

Let's start with precision. The precision of a model tells us how many of the positive predictions turned out to be correct. It's the fraction of correctly predicted

positive examples. In our case, it's the number of customers who actually churned (TP) out of all the customers we thought would churn (TP + FP) (figure 4.15):

$$P = TP / (TP + FP)$$

For our model, the precision is 62%:

$$P = 289 / (289 + 172) = 172 / 461 = 0.62$$

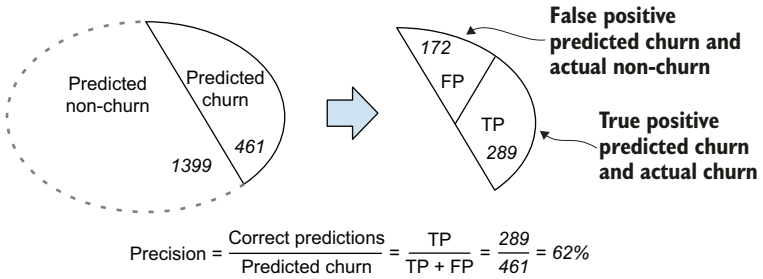


Figure 4.15 The precision of a model is the fraction of correct predictions (TP) among all positive predictions (TP + FP).

Recall is the fraction of correctly classified positive examples among all positive examples. In our case, to calculate recall we first look at all the customers who churned and see how many of them we managed to identify correctly.

The formula for calculating recall is

$$R = TP / (TP + FN)$$

Like in the formula for precision, the numerator is the number of true positives, but the denominator is different: it's the number of all positive examples (`y_val == 1`) in our validation dataset (figure 4.16).

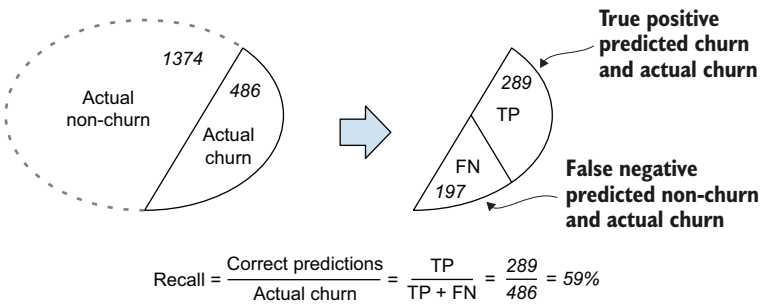


Figure 4.16 The recall of a model is the fraction of correctly predicted churning customers (TP) among all customers who churned (TP + FN).

For our model, the recall is 59%:

$$R = 286 / (289 + 197) = 289 / 486 = 0.59$$

The difference between precision and recall may seem subtle at first. In both cases, we look at the number of correct predictions, but the difference is in the denominators (figure 4.17):

- Precision: what's the percent of correct predictions (TP) among customers predicted as churning (TP + FP)?
- Recall: what's the percentage correctly predicted as churning (TP) among all churned customers (TP + FN)?

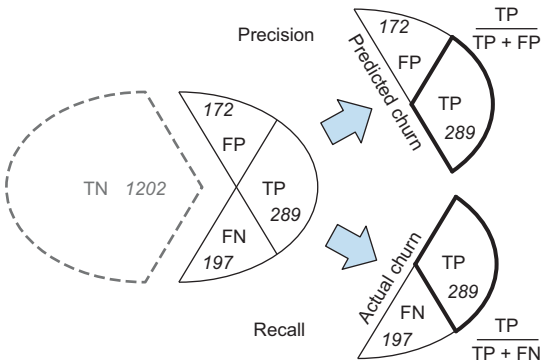


Figure 4.17 Both precision and recall look at the correct predictions (TP), but the denominators are different. For precision, it's the number of customers predicted as churning, whereas for recall, it's the number of customers who churned.

We can also see that both precision and recall don't take true negatives into account (figure 4.17). This is exactly why they are good evaluation metrics for imbalanced datasets. For situations with class imbalance, true negatives typically outnumber everything else — but at the same time, they are also often not really interesting for us. Let's see why.

The goal of our project is to identify customers who are likely to churn. Once we do, we can send them promotional messages in the hopes that they'll change their mind.

When doing this, we make two types of mistakes:

- We accidentally send messages to people who weren't going to churn — these people are the false positives of the model.
- We also sometimes fail to identify people who are actually going to churn. We don't send messages to these people — they are our false negatives.

Precision and recall help us quantify these errors.

Precision helps us understand how many people received a promotional message by mistake. The better the precision, the fewer false positives we have. The precision

of 62% means that 62% of the reached customers indeed were going to churn (our true positives), whereas the remaining 38% were not (false positives).

Recall helps us understand how many of the churning customers we failed to find. The better the recall, the fewer false negatives we have. The recall of 59% means that we reach only 59% of all churning users (true positives) and fail to identify the remaining 41% (false negatives).

As we can see, in both cases, we don't really need to know the number of true negatives: even though we can correctly identify them as not churning, we aren't going to do anything with them.

Although the accuracy of 80% might suggest that the model is great, looking at its precision and recall tells us that it actually makes quite a few errors. This is typically not a deal-breaker: with machine learning it's inevitable that models make mistakes, and at least now we have a better and more realistic understanding of the performance of our churn-prediction model.

Precision and recall are useful metrics, but they describe the performance of a classifier only at a certain threshold. Often it's useful to have a metric that summarizes the performance of a classifier for all possible threshold choices. We look at such metrics in the next section.

Exercise 4.2

What is precision?

a The percent of correctly identified churned customers in the validation dataset
b The percent of customers who actually churned among the customers who we predicted as churning

Exercise 4.3

What is recall?

a The percent of correctly identified churned customers among all churned customers
b The percent of correctly classified customers among customers we predicted as churning

4.3 ROC curve and AUC score

The metrics we have covered so far work only with binary predictions — when we have only True and False values in the output. However, we do have ways to evaluate the performance of a model across all possible threshold choices. ROC curves is one of these options.

ROC stands for "receiver operating characteristic," and it was initially designed for evaluating the strength of radar detectors during World War II. It was used to

assess how well a detector could separate two signals: whether an airplane was there or not. Nowadays it's used for a similar purpose: it shows how well a model can separate two classes, positive and negative. In our case, these classes are "churn" and "no churn."

We need two metrics for ROC curves: TPR and FPR, or true positive rate and false positive rate. Let's take a look at these metrics.

4.3.1 True positive rate and false positive rate

The ROC curve is based on two quantities, FPR and TPR:

- False positive rate (FPR): the fraction of false positives among all negative examples
- True positive rate (TPR): the fraction of true positives among all positive examples

Like precision and recall, these values are based on the confusion matrix. We can calculate them using the following formulas:

$$FPR = FP / (FP + TN)$$

$$TPR = TP / (TP + FN)$$

FPR and TPR involve two separate parts of the confusion table (figure 4.18):

- For FPR, we look at the first row of the table: the fraction of false positives among all negatives.
- For TPR, we look at the second row of the table: the fraction of true positives among all positives.

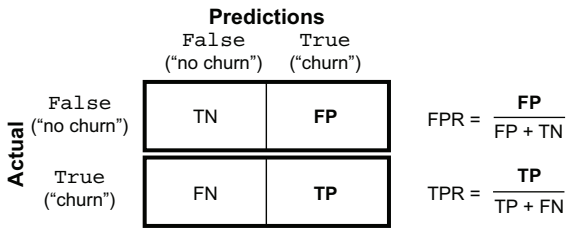


Figure 4.18 For calculating FPR, we look at the first row of the confusion table, and for calculating TPR, we look at the second row.

Let's calculate these values for our model (figure 4.19):

$$FPR = 172 / 1374 = 12.5\%$$

FPR is the fraction of users we predicted as churning among everybody who didn't churn. A small value for FPR tells us that a model is good — it has few false positives:

$$\text{TPR} = 289 / 486 = 59\%$$

TPR is the fraction of users who we predicted as churning among everybody who actually did churn. Note that TPR is the same as recall, so the higher the TPR is, the better.

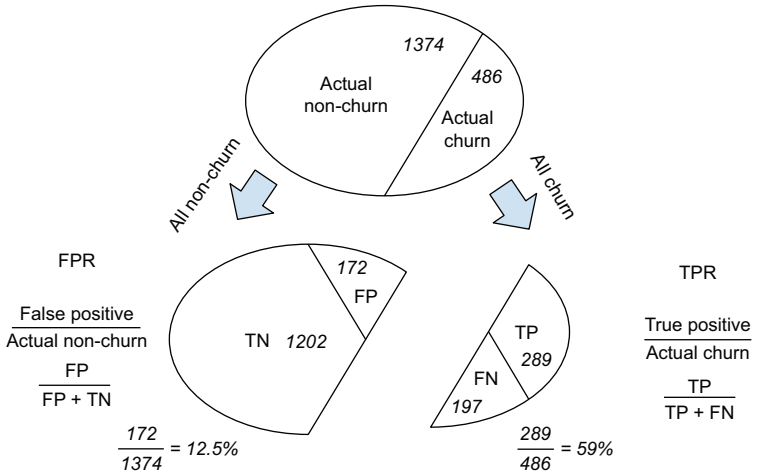


Figure 4.19 FPR is the fraction of false positives among all non-churning customers: the smaller the FPR, the better. TPR is the fraction of true positives among all churning customers: the larger the TPR, the better.

However, we still consider FPR and TPR metrics at only one threshold value — in our case, 0.5. To be able to use them for ROC curves, we need to calculate these metrics for many different threshold values.

4.3.2 Evaluating a model at multiple thresholds

Binary classification models, such as logistic regression, typically output a probability — a score between zero and one. To make actual predictions, we binarize the output by setting some threshold to get only True and False values.

Instead of evaluating the model at one particular threshold, we can do it for a range of them — in the same way we did it for accuracy earlier in this chapter.

For that, we first iterate over different threshold values and compute the values of the confusion table for each.

Listing 4.1 Computing the confusion table for different thresholds

```
scores = []

thresholds = np.linspace(0, 1, 101)

for t in thresholds:
    tp = ((y_pred >= t) & (y_val == 1)).sum()
    fp = ((y_pred >= t) & (y_val == 0)).sum()
    fn = ((y_pred < t) & (y_val == 1)).sum()
    tn = ((y_pred < t) & (y_val == 0)).sum()
    scores.append((t, tp, fp, fn, tn))
```

Creates a list where we'll keep the results

Creates an array with different threshold values, and loops over them

Computes the confusion table for predictions at each threshold

Appends the results to the scores list

The idea is similar to what we previously did with accuracy, but instead of recording just one value, we record all the four outcomes for the confusion table.

It's not easy to deal with a list of tuples, so let's convert it to a Pandas dataframe:

```
df_scores = pd.DataFrame(scores)
df_scores.columns = ['threshold', 'tp', 'fp', 'fn', 'tn']
```

Turns the list into a Pandas dataframe

Assigns names to the columns of the dataframe

This gives us a dataframe with five columns (figure 4.20).

```
df_scores[::10]
```

	threshold	tp	fp	fn	tn
0	0.0	486	1374	0	0
10	0.1	458	726	28	648
20	0.2	421	512	65	862
30	0.3	380	350	106	1024
40	0.4	337	257	149	1117
50	0.5	289	172	197	1202
60	0.6	200	105	286	1269
70	0.7	99	34	387	1340
80	0.8	7	1	479	1373
90	0.9	0	0	486	1374
100	1.0	0	0	486	1374

Figure 4.20 The dataframe with the elements of the confusion matrix evaluated at different threshold levels. The `[::10]` expression selects every 10th record of the dataframe.

Now we can compute the TPR and FPR scores. Because the data is now in a dataframe, we can do it for all the values at once:

```
df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)
df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.tn)
```

After running this code, we have two new columns in the dataframe: tpr and fpr (figure 4.21).

```
df_scores[::10]
```

	threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000	1.000000
10	0.1	458	726	28	648	0.942387	0.528384
20	0.2	421	512	65	862	0.866255	0.372635
30	0.3	380	350	106	1024	0.781893	0.254731
40	0.4	337	257	149	1117	0.693416	0.187045
50	0.5	289	172	197	1202	0.594650	0.125182
60	0.6	200	105	286	1269	0.411523	0.076419
70	0.7	99	34	387	1340	0.203704	0.024745
80	0.8	7	1	479	1373	0.014403	0.000728
90	0.9	0	0	486	1374	0.000000	0.000000
100	1.0	0	0	486	1374	0.000000	0.000000

Figure 4.21 The dataframe with the values of the confusion matrix as well as TPR and FPR evaluated at different thresholds

Let's plot them (figure 4.22):

```
plt.plot(df_scores.threshold, df_scores.tpr, label='TPR')
plt.plot(df_scores.threshold, df_scores.fpr, label='FPR')
plt.legend()
```

Both TPR and FPR start at 100% — at the threshold of 0.0, we predict "churn" for everyone:

- FPR is 100% because we have only false positives in the prediction. There are no true negatives: nobody is predicted as non-churning.
- TPR is 100% because we have only true positives and no false negatives.

As the threshold grows, both metrics decline but at different rates.

Ideally, FPR should go down very quickly. A small FPR indicates that the model makes very few mistakes predicting negative examples (false positives).

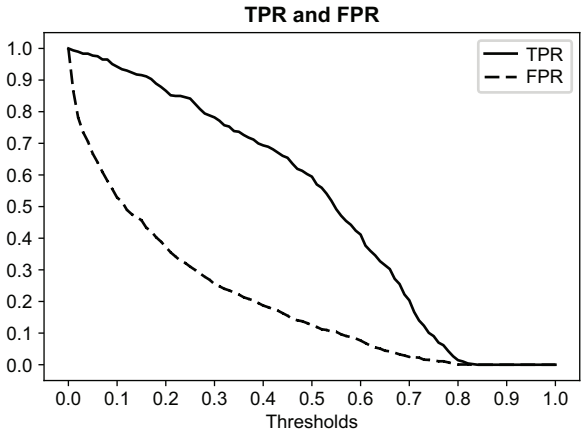


Figure 4.22 The TPR and FPR for our model, evaluated at different thresholds

On the other hand, TPR should go down slowly, ideally staying near 100% all the time: that will mean that the model predicts true positives well.

To better understand what these TPR and FPR mean, let's compare it with two baseline models: a random model and the ideal model. We will start with a random model.

4.3.3 Random baseline model

A random model outputs a random score between 0 and 1, regardless of the input. It's easy to implement — we simply generate an array with uniform random numbers:

```
np.random.seed(1)
y_rand = np.random.uniform(0, 1, size=len(y_val))
```

Fixes the random seed for reproducibility

Generates an array with random numbers between 0 and 1

Now we can simply pretend that `y_rand` contains the predictions of our "model."

Let's calculate FPR and TPR for our random model. To make it simpler, we'll reuse the code we wrote previously and put it into a function.

Listing 4.2 Function for calculating TPR and FPR at different thresholds

```
def tpr_fpr_dataframe(y_val, y_pred):
    scores = []

    thresholds = np.linspace(0, 1, 101)

    for t in thresholds:
        tp = ((y_pred >= t) & (y_val == 1)).sum()
        fp = ((y_pred >= t) & (y_val == 0)).sum()
        fn = ((y_pred < t) & (y_val == 1)).sum()
        tn = ((y_pred < t) & (y_val == 0)).sum()
        scores.append((t, tp, fp, fn, tn))
```

Defines a function that takes in actual and predicted values

Calculates the confusion table for different thresholds

```
    df_scores = pd.DataFrame(scores)
    df_scores.columns = ['threshold', 'tp', 'fp', 'fn', 'tn']

    df_scores['tpr'] = df_scores.tp / (df_scores.tp + df_scores.fn)
    df_scores['fpr'] = df_scores.fp / (df_scores.fp + df_scores.fn)

    return df_scores
```

Converts the confusion table numbers to a dataframe

Calculates TPR and FPR using the confusion table numbers

Returns the resulting dataframe

Now let's use this function to calculate the TPR and FPR for the random model:

```
df_rand = tpr_fpr_dataframe(y_val, y_rand)
```

This creates a dataframe with TPR and FPR values at different thresholds (figure 4.23).

```
np.random.seed(1)
y_rand = np.random.uniform(0, 1, size=len(y_val))
df_rand = tpr_fpr_dataframe(y_val, y_rand)
df_rand[::10]
```

	threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000	1.000000
10	0.1	440	1236	46	138	0.905350	0.899563
20	0.2	392	1101	94	273	0.806584	0.801310
30	0.3	339	972	147	402	0.697531	0.707424
40	0.4	288	849	198	525	0.592593	0.617904
50	0.5	239	723	247	651	0.491770	0.526201
60	0.6	193	579	293	795	0.397119	0.421397
70	0.7	152	422	334	952	0.312757	0.307132
80	0.8	98	302	388	1072	0.201646	0.219796
90	0.9	57	147	429	1227	0.117284	0.106987
100	1.0	0	0	486	1374	0.000000	0.000000

Figure 4.23 The TPR and FPR values of a random model

Let's plot them:

```
plt.plot(df_rand.threshold, df_rand.tpr, label='TPR')
plt.plot(df_rand.threshold, df_rand.fpr, label='FPR')
plt.legend()
```

We see that both TPR and FPR curves go from 100% to 0%, almost following the straight line (figure 4.24).

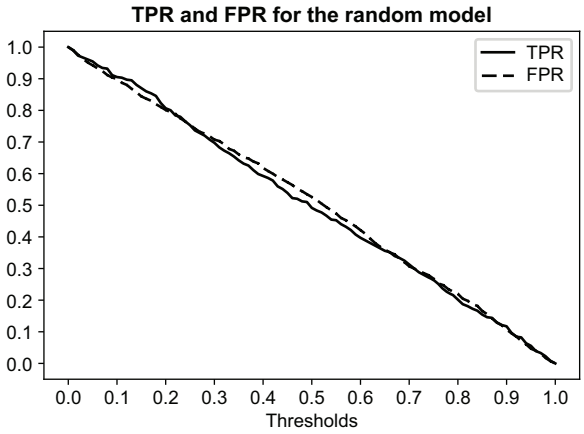


Figure 4.24 Both TPR and FPR of a random classifier decrease from 100% to 0% as a straight line.

At the threshold of 0.0, we treat everybody as churning. Both TPR and FPR are 100%:

- FPR is 100% because we have only false positives: all non-churning customers are identified as churning.
- TPR is 100% because we have only true positives: we can correctly classify all churning customers as churning.

As we increase the threshold, both TPR and FPR decrease.

At the threshold of 0.4, the model with a probability of 40% predicts "non-churn," and with a probability of 60% predicts "churn." Both TPR and FPR are 60%:

- FPR is 60% because we incorrectly classify 60% of non-churning customers as churning.
- TPR is 60% because we correctly classify 60% of churning customers as churning.

Finally, at 1.0, both TPR and FPR are 0%. At this threshold, we predict everybody as non-churning:

- FPR is 0% because we have no false positives: we can correctly classify all non-churning customers as non-churning.
- TPR is 0% because we have no true positives: all churning customers are identified as non-churning.

Let's now move on to the next baseline and see how TPR and FPR look for the ideal model.

4.3.4 The ideal model

The ideal model always makes correct decisions. We'll take it a step further and consider the ideal ranking model. This model outputs scores in such a way that churning

customers always have higher scores than non-churning ones. In other words, the predicted probability for all churned ones should be higher than the predicted probability for non-churned ones.

So, if we apply the model to all the customers in our validation set and then sort them by the predicted probability, we first will have all the non-churning customers, followed by the churning ones (figure 4.25).

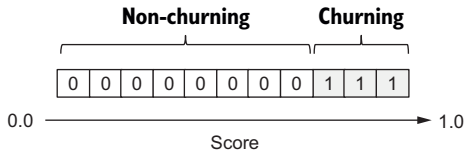


Figure 4.25 The ideal model orders customers such that first we have non-churning customers and then churning ones.

Of course, we cannot have such a model in real life. It's still useful, however: we can use it for comparing our TPR and FPR to the TPR and FPR of the ideal model.

Let's generate the ideal predictions. To make it easier, we generate an array with fake target variables that are already ordered: first it contains only 0s and then only 1s (figure 4.25). As for "predictions," we simply can create an array with numbers that grow from 0 in the first cell to 1 in the last cell using the `np.linspace` function.

Let's do it:

```
num_neg = (y_val == 0).sum()
num_pos = (y_val == 1).sum()
```

Calculates the number of negative and positive examples in the dataset

```
y_ideal = np.repeat([0, 1], [num_neg, num_pos])
y_pred_ideal = np.linspace(0, 1, num_neg + num_pos)

df_ideal = tpr_fpr_dataframe(y_ideal, y_pred_ideal)
```

Generates an array that first repeats 0s num_neg number of times, followed by 1s repeated num_pos number of times

Generates the predictions of the "model": numbers that grow from 0 in the first cell to 1 in the last

Computes the TPR and FPR curves for the classifier

As a result, we get a dataframe with the TPR and FPR values of the ideal model (figure 4.26). You can read more about `np.linspace` and `np.repeat` functions in appendix C.

Now we can plot it (figure 4.27):

```
plt.plot(df_ideal.threshold, df_ideal.tpr, label='TPR')
plt.plot(df_ideal.threshold, df_ideal.fpr, label='FPR')
plt.legend()
```

	threshold	tp	fp	fn	tn	tpr	fpr
0	0.0	486	1374	0	0	1.000000	1.000000
10	0.1	486	1188	0	186	1.000000	0.864629
20	0.2	486	1002	0	372	1.000000	0.729258
30	0.3	486	816	0	558	1.000000	0.593886
40	0.4	486	630	0	744	1.000000	0.458515
50	0.5	486	444	0	930	1.000000	0.323144
60	0.6	486	258	0	1116	1.000000	0.187773
70	0.7	486	72	0	1302	1.000000	0.052402
80	0.8	372	0	114	1374	0.765432	0.000000
90	0.9	186	0	300	1374	0.382716	0.000000
100	1.0	1	0	485	1374	0.002058	0.000000

Figure 4.26 The TPR and FPR values for the ideal model

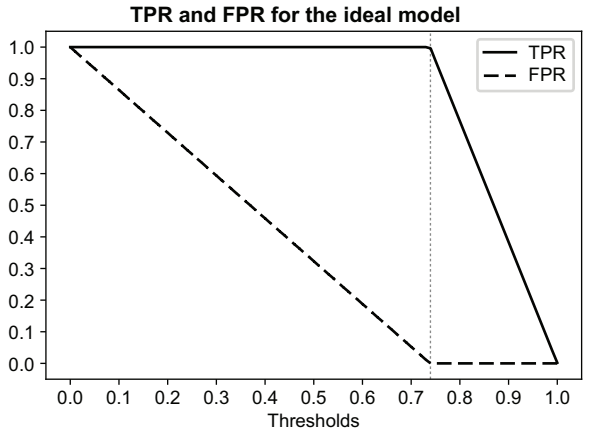


Figure 4.27 The TPR and FPR curves for the ideal model

From the plot, we can see that

- Both TPR and FPR start at 100% and end at 0%.
- For thresholds lower than 0.74, we always correctly classify all churning customers as churning; that's why TRP stays at 100%. On the other hand, we incorrectly classify some non-churning ones as churning — those are our false positives. As we increase the threshold, fewer and fewer non-churning customers are classified as churning, so FPR goes down. At 0.6, we misclassify 258 non-churning customers as churning (figure 4.28, A).

- The threshold of 0.74 is the ideal situation: all churning customers are classified as churning, and all non-churning are classified as non-churning; that's why TPR is 100% and FPR is 0% (figure 4.28, B).
- Between 0.74 and 1.0, we always correctly classify all non-churning customers, so FPR stays at 0%. However, as we increase the threshold, we start incorrectly classifying more and more churning customers as non-churning, so TPR goes down. At 0.8, 114 out of 446 churning customers are incorrectly classified as non-churning. Only 372 predictictions are correct, so TPR is 76% (figure 4.28, C).

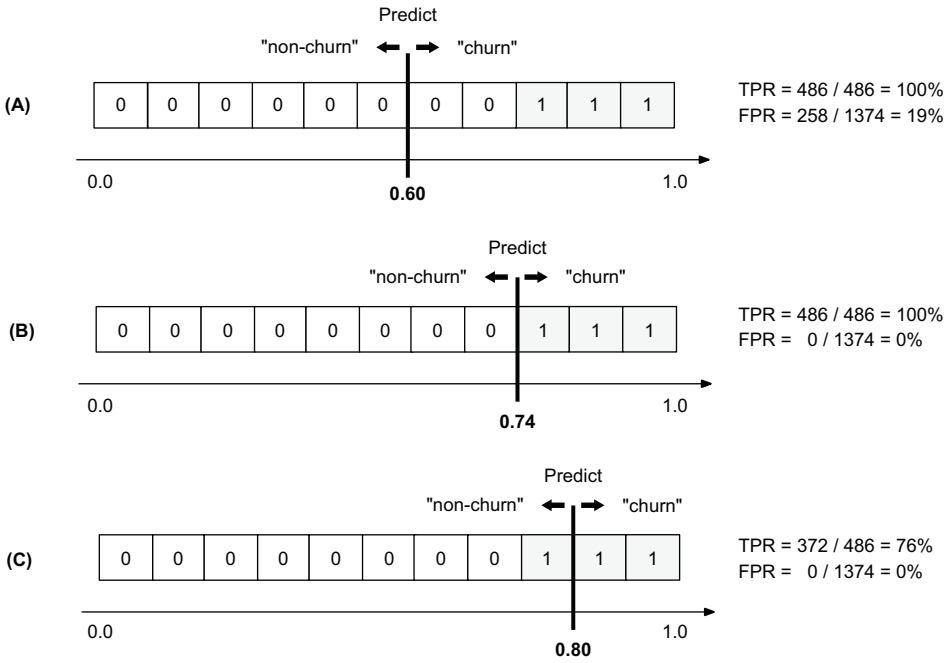


Figure 4.28 TPR and FPR of the ideal ranking model evaluated at different thresholds

Now we're ready to build the ROC curve.

Exercise 4.4

What does the ideal ranking model do?

a When applied to the validation data, it scores the customers such that for non-churning customers, the score is always lower than for churning ones.
b It scores non-churning customers higher than churning ones.

4.3.5 ROC Curve

To create an ROC curve, instead of plotting FPR and TPR against different threshold values, we plot them against each other. For comparison, we also add the ideal and random models to the plot:

```
plt.figure(figsize=(5, 5))      <-- Makes the plot square

plt.plot(df_scores.fpr, df_scores.tpr, label='Model')
plt.plot(df_rand.fpr, df_rand.tpr, label='Random')
plt.plot(df_ideal.fpr, df_ideal.tpr, label='Ideal')
                                    Plots the ROC curve for the model and baselines

plt.legend()
```

As a result, we get an ROC curve (figure 4.29). When we plot it, we can see that the ROC curve of the random classifier is an approximately straight line from bottom left to top right. For the ideal model, however, the curve first goes up until it reaches 100% TPR, and from there it goes right until it reaches 100% FPR.

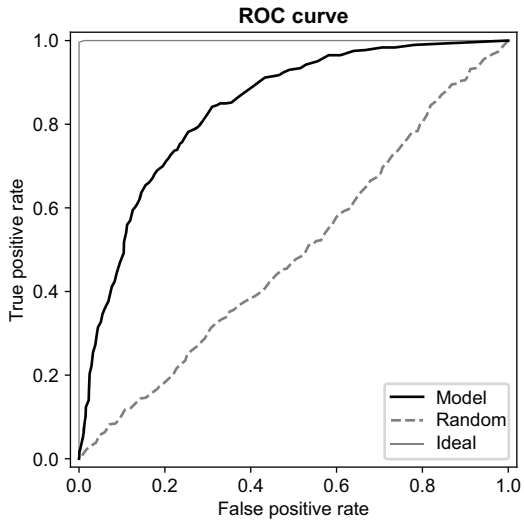


Figure 4.29 The ROC curve shows the relationship between the FPR and TPR of a model.

Our models should always be somewhere between these two curves. We want our model to be as close to the ideal curve as possible and as far as possible from the random curve.

The ROC curve of a random model serves as a good visual baseline — when we add it to the plot, it helps us to judge how far our model is from this baseline — so it's a good idea to always include this line in the plot.

However, we don't really need to generate a random model each time we want to have an ROC curve: we know what it looks like, so we can simply include a straight line from (0, 0) to (1, 1) in the plot.

As for the ideal model, we know that it always goes up to (0, 1) and then goes right to (1, 1). The top-left corner is called the "ideal spot": it's the point when the ideal model gets 100% TPR and 0% FPR. We want our models to get as close to the ideal spot as possible.

With this information, we can reduce the code for plotting the curve to the following:

```
plt.figure(figsize=(5, 5))
plt.plot(df_scores.fpr, df_scores.tpr)
plt.plot([0, 1], [0, 1])
```

This produces the result in figure 4.30.

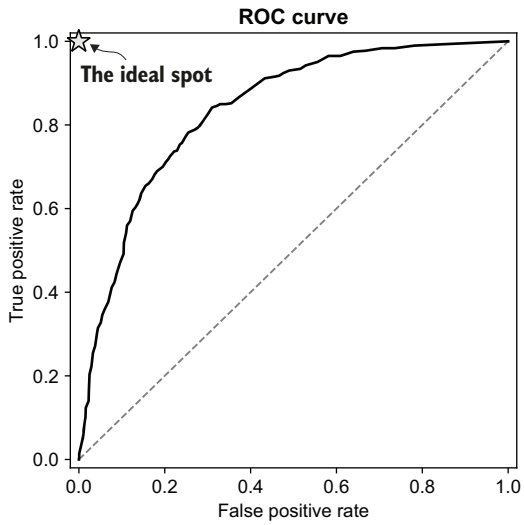


Figure 4.30 The ROC curve. The baseline makes it easier to see how far the ROC curve of our model is from that of a random model. The top-left corner (0, 1) is the "ideal spot": the closer our models get to it, the better.

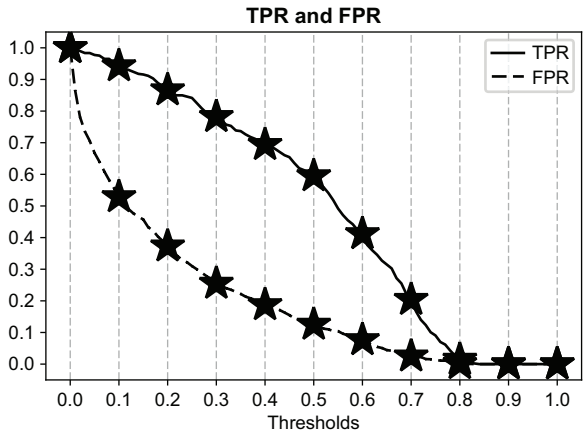
Although computing all the FPR and TPR values across many thresholds is a good exercise, we don't need to do it ourselves every time we want to plot an ROC curve. We simply can use the `roc_curve` function from the `metrics` package of Scikit-learn:

```
from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_val, y_pred)

plt.figure(figsize=(5, 5))
plt.plot(fpr, tpr)
plt.plot([0, 1], [0, 1])
```

As a result, we get a plot identical to the previous one (figure 4.30).

Now let's try to make more sense of the curve and understand what it can actually tell us. To do this, we visually map the TPR and FPR values to their thresholds on the ROC curve (figure 4.31).

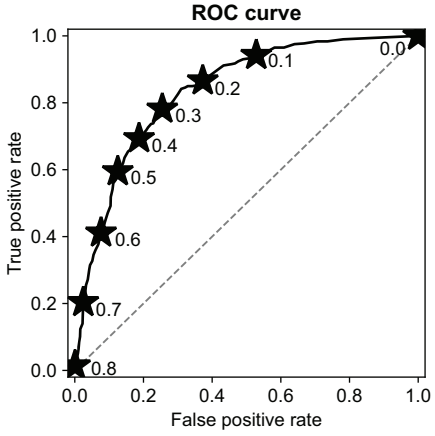
In the ROC plot, we start from the (0, 0) point — this is the point at the bottom left. It corresponds to 0% FPR and 0% TPR, which happens at high thresholds like 1.0, when



(A) TPR and FPR at different thresholds

	threshold	fpr	tpr
100	1.0	0.000000	0.000000
90	0.9	0.000000	0.000000
80	0.8	0.000728	0.014403
70	0.7	0.024745	0.203704
60	0.6	0.076419	0.411523
50	0.5	0.125182	0.594650
40	0.4	0.187045	0.693416
30	0.3	0.254731	0.781893
20	0.2	0.372635	0.866255
10	0.1	0.528384	0.942387
0	0.0	1.000000	1.000000

(B) FPR and TPR values of the model for different thresholds



(C) FPR and TPR values for selected thresholds

Figure 4.31 Translation of the TPR and FPR plots against different threshold values (A and B) to the ROC curve (C). In the ROC plot, we start from the bottom left with high threshold values, where most of the customers are predicted as non-churning, and gradually go to the top right with low thresholds, where most of the customers are predicted as churning.

no customers are above that score. For these cases we simply end up predicting "no churn" for everyone. That's why our TPR is 0%: we are never correctly predicting churned customers. FPR, on the other hand, is 0% because this dummy model can correctly predict all non-churning customers as non-churning, so there are no false positives.

As we go up the curve, we consider FPR and TPR values evaluated at smaller thresholds. At 0.7, FPR changes only slightly, from 0% to 2%, but the TPR increases from 0% to 20% (figure 4.31, B and C).

As we follow the line, we keep decreasing the threshold and evaluating the model at smaller values, predicting more and more customers as churning. At some point, we cover most of the positives (churning customers). For example, at the threshold of 0.2, we predict most of the users as churning, which means that many of these predictions are false positives. FPR then starts to grow faster than TPR; at the threshold of 0.2, it's already at almost 40%.

Eventually, we reach the 0.0 threshold and predict that everyone is churning, thus reaching the top-right corner of the ROC plot.

When we start at high threshold values, all models are equal: any model at high threshold values degrades to the constant "model" that predicts False all the time. As we decrease the threshold, we start predicting some of the customers as churning. The better the model, the more customers are correctly classified as churning, resulting in a better TPR. Likewise, good models have a smaller FPR because they have fewer false positives.

Thus, the ROC curve of a good model first goes up as high as it can and only then starts turning right. Poor models, on the other hand, from the start have higher FPRs and lower TPRs, so their curves tend to go to the right earlier (figure 4.32).

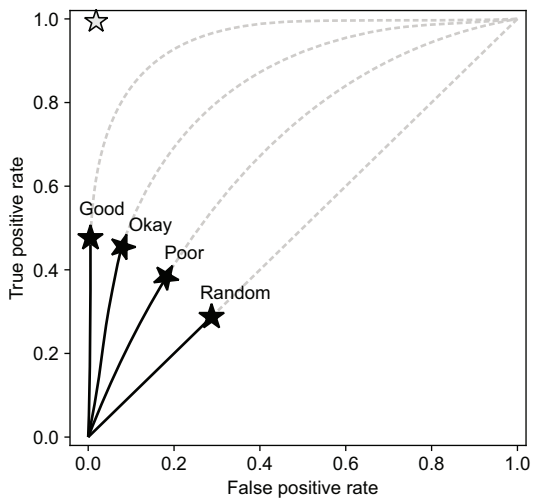


Figure 4.32 ROC curves of good models go up as much as they can before turning right. Poor models, on the other hand, tend to have more false positives from the beginning, so they tend to go right earlier.

We can use this for comparing multiple models: we can simply plot them on the same graph and see which of them is closer to the ideal point of (0, 1). For example, let's take a look at the ROC curves of the large and small models and plot them on the same graph:

```
fpr_large, tpr_large, _ = roc_curve(y_val, y_pred)
fpr_small, tpr_small, _ = roc_curve(y_val, y_pred_small)

plt.figure(figsize=(5, 5))

plt.plot(fpr_large, tpr_large, color='black', label='Large')
plt.plot(fpr_small, tpr_small, color='black', label='Small')
plt.plot([0, 1], [0, 1])
plt.legend()
```

This way we can get two ROC curves on the same plot (figure 4.33). We can see that the large model is better than the small model: it's closer to the ideal point for all the thresholds.

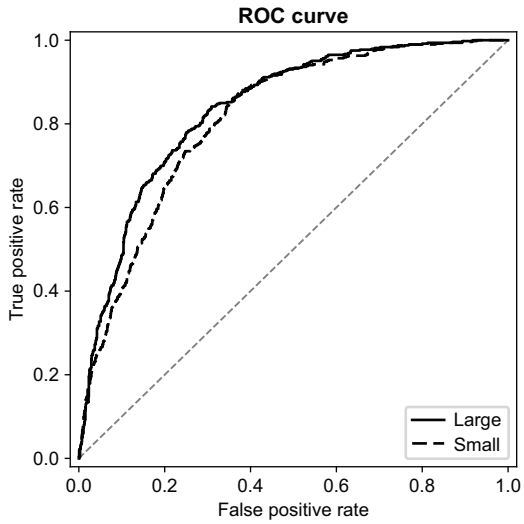


Figure 4.33 Plotting multiple ROC curves on the same graph helps us visually identify which model performs better.

ROC curves are quite useful on their own, but we also have another metric that's based on it: AUC, or the area under the ROC curve.

4.3.6 Area under the ROC curve (AUC)

When evaluating our models using the ROC curve, we want them to be as close to the ideal spot and as far from the random baseline as possible.

We can quantify this "closeness" by measuring the area under the ROC curve. We can use this metric — abbreviated as AU ROC, or often simply AUC — as a metric for evaluating the performance of a binary classification model.

The ideal model forms a 1x1 square, so the area under its ROC curve is 1, or 100%. The random model takes only half of that, so its AUC is 0.5, or 50%. The AUCs of our two models — the large one and the small one — will be somewhere between the random baseline of 50% and the ideal curve of 100%.

IMPORTANT An AUC of 0.9 is indicative of a reasonably good model; 0.8 is okay, 0.7 is not very performant, and 0.6 indicates quite poor performance.

To calculate the AUC for our models we can use `auc`, a function from the `metrics` package of Scikit-learn:

```
from sklearn.metrics import auc
auc(df_scores.fpr, df_scores.tpr)
```

For the large model, the result is 0.84; for the small model, it's 0.81 (figure 4.34). Churn prediction is a complex problem, so an AUC of 80% is quite good.

```
from sklearn.metrics import auc
auc(df_scores.fpr, df_scores.tpr)
```

```
0.8359001084215382
```

```
auc(df_scores_small.fpr, df_scores_small.tpr)
```

```
0.8125475467380692
```

Figure 4.34 The AUC for our models: 84% for the large model and 81% for the small model

If all we need is the AUC, we don't need to compute the ROC curve first. We can take a shortcut and use the `roc_auc_score` function from Scikit-learn, which takes care of everything and simply returns the AUC of our model:

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_val, y_pred)
```

We get approximately the same results as previously (figure 4.35).

NOTE The values from `roc_auc_score` may be slightly different from AUC computed from the dataframes where we calculated TPR and FPR ourselves: Scikit-learn internally uses a more precise method for creating ROC curves.

ROC curves and AUC scores tell us how well the model separates positive and negative examples. What is more, AUC has a nice probabilistic interpretation: it tells us what

```
from sklearn.metrics import roc_auc_score
roc_auc_score(y_val, y_pred)
```

```
0.8363366398907399
```

```
roc_auc_score(y_val, y_pred_small)
```

```
0.8129354083179088
```

Figure 4.35 Calculating AUC using Scikit-learn's `roc_auc_score` function.

the probability is that a randomly selected positive example will have a score higher than a randomly selected negative example.

Suppose we randomly pick a customer that we know churned and a customer who didn't and then apply the model to these customers and see what the score is for each. We want the model to score the churning customer higher than the non-churning one. AUC tells us the probability of that happening: it's the probability that the score of a randomly selected churning customer is higher than the score of a randomly selected non-churning one.

We can verify this. If we do this experiment 10,000 times and then count how many times the score of the positive example was higher than the score of the negative one, the percentage of cases when it's true should roughly correspond to the AUC:

Selects the score for all non-churning customers

Selects the score for all churning customers

Randomly selects 10,000 scores of negative examples (non-churning customers)

Fixes the seed to make sure the results are reproducible

```
neg = y_pred[y_val == 0]
pos = y_pred[y_val == 1]

np.random.seed(1)
neg_choice = np.random.randint(low=0, high=len(neg), size=10000)
pos_choice = np.random.randint(low=0, high=len(pos), size=10000)
(pos[pos_choice] > neg[neg_choice]).mean()
```

Randomly selects 10,000 scores of positive examples (churning customers)

For each of the positive examples, checks if the score is higher than the respective negative example

This prints 0.8356, which is indeed pretty close to the AUC value of our classifier.

This interpretation of AUC gives us additional insight into the quality of our models. The ideal model orders all the customers such that we first have non-churning customers and then churning customers. With this order, the AUC is always 1.0: the score of a randomly chosen churning customer is always higher than the score of a non-churning customer. On the other hand, the random model just shuffles the customers, so the score of a churning customer has only a 50% chance of being higher than the score of a non-churning one.

AUC thus not only gives us a way of evaluating the models at all possible thresholds but also describes how well the model separates two classes: in our case, churning and

non-churning. If the separation is good, then we can order the customers such that most of the churning users come first. Such a model will have a good AUC score.

> **NOTE** You should keep this interpretation in mind: it provides an easy way to explain the meaning behind AUC to people without a machine learning background, such as managers and other decision makers.

This makes AUC the default classification metric in most situations, and it's often the metric we use when finding the best parameter set for our models.

The process of finding the best parameters is called "parameter tuning," and in the next section we will see how to do this.

4.4 Parameter tuning

In the previous chapter, we used a simple hold-out validation scheme for testing our models. In this scheme, we take part of the data out and keep it for validation purposes only. This practice is good but doesn't always give us the whole picture. It tells us how well the model will perform on these specific data points. However, it doesn't necessarily mean the model will perform equally well on other data points. So, how do we check if the model indeed works well in a consistent and predictable manner?

4.4.1 K-fold cross-validation

It's possible to use all the available data to assess the quality of models and get more reliable validation results. We can simply perform validation multiple times.

First, we split the entire dataset into a certain number of parts (say, three). Then we train a model on two parts and validate on the remaining one. We repeat this process three times and at the end get three different scores. This is exactly the idea behind K-fold cross-validation (figure 4.36).

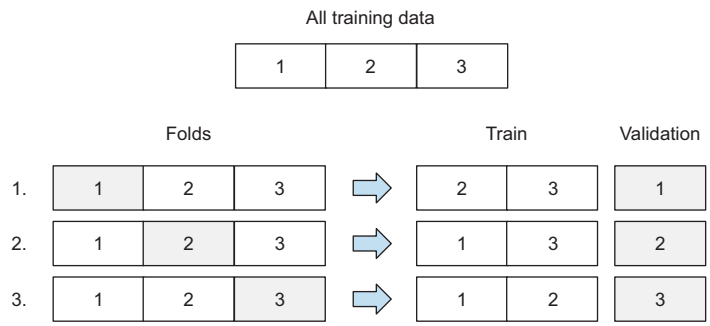


Figure 4.36 K-fold cross-validation (K=3). We split the entire dataset into three equal parts, or folds. Then, for each fold, we take it as the validation dataset and use the remaining K – 1 folds as the training data. After training the model, we evaluate it on the validation fold, and at the end we get k metric values.

Before we implement it, we need to make the training process simpler, so it's easy to run this process multiple times. For that, we'll put all the code for training into a `train` function, which first converts the data into a one-hot encoding representation and then trains the model.

Listing 4.3 Training the model

```
def train(df, y):
    cat = df[categorical + numerical].to_dict(orient='records')

    dv = DictVectorizer(sparse=False)
    dv.fit(cat)

    X = dv.transform(cat)

    model = LogisticRegression(solver='liblinear')
    model.fit(X, y)

    return dv, model
```

Applies one-hot encoding

Trains the model

Likewise, we also put the prediction logic into a `predict` function. This function takes in a dataframe with customers, the vectorizer we "trained" previously — for doing one-hot encoding — and the model. Then we apply the vectorizer to the dataframe, get a matrix, and finally apply the model to the matrix to get predictions.

Listing 4.4 Applying the model to new data

```
def predict(df, dv, model):
    cat = df[categorical + numerical].to_dict(orient='records')

    X = dv.transform(cat)
    y_pred = model.predict_proba(X)[:, 1]

    return y_pred
```

Applies the same one-hot encoding scheme as in training

Uses the model to make predictions

Now we can use these functions for implementing K-fold cross-validation.

We don't need to implement cross-validation ourselves: in Scikit-learn there's a class for doing that. It's called `KFold`, and it lives in the `model_selection` package.

Listing 4.5 K-fold cross-validation

```
from sklearn.model_selection import KFold

kfold = KFold(n_splits=10, shuffle=True, random_state=1)

aucs = []

for train_idx, val_idx in kfold.split(df_train_full):
    df_train = df_train_full.iloc[train_idx]
    df_val = df_train_full.iloc[val_idx]

    y_train = df_train.churn.values
    y_val = df_val.churn.values
```

1 Imports the KFold class

2 Uses it to split the data into 10 parts

3 Creates a list for storing the results

4 Iterates over the 10 different splits of the data

5 Splits the data into train and validation sets

```
dv, model = train(df_train, y_train)
y_pred = predict(df_val, dv, model)

auc = roc_auc_score(y_val, y_pred)
aucs.append(auc)
```

6 Trains the model and makes predictions

7 Evaluates the quality of the train model on the validation data using AUC

8 Saves the AUC to the list with the results

Note that when defining the splitting in the `KFold` class in 2, we set three parameters:

- `n_splits = 10`: That's K, which specifies the number of splits.
- `shuffle = True`: We ask it to shuffle the data before splitting it.
- `random_state = 1`: Because there's randomization in the process (shuffling data), we want the results to be reproducible, so we fix the seed for the random-number generator.

Here we used K-fold cross-validation with K =10. Thus, when we run it, at the end we get 10 different numbers — 10 AUC scores evaluated on 10 different validation folds:

```
0.849, 0.841, 0.859, 0.833, 0.824, 0.841, 0.844, 0.822, 0.845, 0.861
```

It's not a single number anymore, and we can think of it as a distribution of AUC scores for our model. We can get some statistics from this distribution, such as the mean and standard deviation:

```
print('auc = %0.3f ± %0.3f' % (np.mean(aucs), np.std(aucs)))
```

This prints "0.842 ± 0.012".

Now, not only do we know the average performance, but we also have an idea of how volatile that performance is, or how far it may deviate from the average.

A good model should be quite stable across different folds: this way, we make sure we don't get a lot of surprises when the model goes live. The standard deviation tells us about that: the smaller it is, the more stable the model is.

Now we can use K-fold cross-validation for parameter tuning: selecting the best parameters.

4.4.2 Finding best parameters

We learned how we can use K-fold cross-validation for evaluating the performance of our model. The model we trained previously was using the default value for the parameter `C`, which controls the amount of regularization.

Let's select our cross-validation procedure for selecting the best parameter `C`. For that, we first adjust the `train` function to take in an additional parameter.

Listing 4.6 Function for training the model with parameter C for controlling regularization

```
def train(df, y, C):
    cat = df[categorical + numerical].to_dict(orient='records')
```

Adds an extra parameter to the train function

```
    dv = DictVectorizer(sparse=False)
    dv.fit(cat)

    X = dv.transform(cat)

    model = LogisticRegression(solver='liblinear', C=C)
    model.fit(X, y)

    return dv, model
```

Uses this parameter during training

Now let's find the best parameter `C`. The idea is simple:

- Loop over different values of `C`.
- For each `C`, run cross-validation and record the mean AUC across all folds as well as the standard deviation.

Listing 4.7 Tuning the model: selecting the best parameter C using cross-validation

```
nfolds = 5
kfold = KFold(n_splits=nfolds, shuffle=True, random_state=1)

for C in [0.001, 0.01, 0.1, 0.5, 1, 10]:
    aucs = []

    for train_idx, val_idx in kfold.split(df_train_full):
        df_train = df_train_full.iloc[train_idx]
        df_val = df_train_full.iloc[val_idx]

        y_train = df_train.churn.values
        y_val = df_val.churn.values

        dv, model = train(df_train, y_train, C=C)
        y_pred = predict(df_val, dv, model)

        auc = roc_auc_score(y_val, y_pred)
        aucs.append(auc)

    print('C=%s, auc = %0.3f ± %0.3f' % (C, np.mean(aucs), np.std(aucs)))
```

When we run it, it prints

```
C=0.001, auc = 0.825 ± 0.013
C=0.01, auc = 0.839 ± 0.009
C=0.1, auc = 0.841 ± 0.008
C=0.5, auc = 0.841 ± 0.007
C=1, auc = 0.841 ± 0.007
C=10, auc = 0.841 ± 0.007
```

What we see is that after `C` = 0.1, the average AUC is the same and doesn't grow anymore.

However, the standard deviation is smaller for `C` = 0.5 than for `C` = 0.1, so we should use that. The reason we prefer `C` = 0.5 to `C` = 1 and `C` = 10 is simple: when the

C parameter is small, the model is more regularized. The weights of this model are more restricted, so in general, they are smaller. Small weights in the model give us additional assurance that the model will behave well when we use it on real data. So we select C = 0.5.

Now we need to do the last step: train the model on the entire train and validation datasets and apply it to the test dataset to verify it indeed works well.

Let's use our `train` and `predict` functions for that:

```
y_train = df_train_full.churn.values
y_test = df_test.churn.values

dv, model = train(df_train_full, y_train, C=0.5)
y_pred = predict(df_test, dv, model)

auc = roc_auc_score(y_test, y_pred)
print('auc = %.3f' % auc)
```

Trains the model on the full training dataset

Applies it to the test dataset

Evaluates the predictions on the test data

When we execute the code, we see that the performance of the model (AUC) on the held-out test set is 0.858.

That's a little higher than what we had on the validation set, but that's not an issue; it could happen just by chance. What's important is that the score is not significantly different from the validation score.

Now we can use this model for scoring real customers and think about our marketing campaign for preventing churn. In the next chapter, we will see how to deploy this model in a production environment.

4.5 Next steps

4.5.1 Exercises

Try the following exercises to further explore the topics of model evaluation and model selection:

- In this chapter, we plotted TPR and FPR for different threshold values, and it helped us understand what these metrics mean and also how the performance of our model changes when we choose a different threshold. It's helpful to do a similar exercise for precision and recall, so try to repeat this experiment, this time using precision and recall instead of TPR and FPR.
- When plotting precision and recall for different threshold values, we can see that a conflict exists between precision and recall: when one goes up, the other goes down, and the other way around. This is called the "precision-recall tradeoff": we cannot select a threshold that makes both precision and recall good. However, we do have strategies for selecting the threshold, even though precision and recall are conflicting. One of them is plotting precision and recall curves and seeing where they intersect, and using this threshold for binarizing the predictions. Try implementing this idea.

- Another idea for working around the precision-recall trade-off is the F1 score — a score that combines both precision and recall into one value. Then, to select the best threshold, we can simply choose the one that maximizes the F1 score. The formula for computing the F1 score is $F1 = 2 \cdot P \cdot R / (P + R)$, where P is precision and R is recall. Implement this idea, and select the best threshold based on the F1 metric.
- We've seen that precision and recall are better metrics for evaluating classification models than accuracy because they don't rely on false positives, the amount of which could be high in imbalanced datasets. Yet, we saw later that AUC does actually use false positives in FPR. For very highly imbalanced cases (say, 1,000 negatives to 1 positive), AUC may become problematic as well. Another metric works better in such cases: area under the precision-recall curve, or AU PR. The precision-recall curve is similar to ROC, but instead of plotting FPR versus TPR, we plot recall on the x-axis and precision on the y-axis. Like for the ROC curve, we can also calculate the area under the PR curve and use it as a metric for evaluating different models. Try plotting the PR curves for our models, calculating the AU PR scores, and comparing them with those of the random model as well as the ideal model.
- We covered K-fold cross-validation, and we used it to understand what the distribution of AUC scores could look like on a test dataset. When K = 10, we get 10 observations, which under some circumstances might not be enough. However, the idea can be extended to repeated K-fold cross-validation steps. The process is simple: we repeat the K-fold cross-validation process multiple times, each time shuffling the dataset differently by selecting a different random seed at each iteration. Implement repeated cross-validation and perform 10-fold cross-validation 10 times to see what the distribution of scores looks like.

4.5.2 Other projects

You can also continue with the other self-study projects from the previous chapter: the lead scoring project and the default prediction project. Try the following:

- Calculate all the metrics that we covered in this chapter: the confusion table, precision and recall, and AUC. Also try to calculate the scores from the exercises: the F1 score as well as AU PR (the area under the precision-recall curve).
- Use K-fold cross-validation to select the best parameter `C` for the model.

Summary

- A metric is a single number that can be used for evaluating the performance of a machine learning model. Once we choose a metric, we can use it to compare multiple machine learning models with each other and select the best one.
- Accuracy is the simplest binary classification metric: it tells us the percentage of correctly classified observations in the validation set. It's easy to understand and compute, but it can be misleading when a dataset is imbalanced.

- When a binary classification model makes a prediction, we have only four possible outcomes: true positive and true negative (correct answers) and false positive and false negative (incorrect answers). The confusion table arranges these outcomes visually so it's easy to understand them. It gives us the foundation for many other binary classification metrics.
- Precision is the fraction of correct answers among observations for which our prediction is True. If we use the churn model to send promotional messages, precision tells us the percentage of customers who really were going to churn among everybody who received the message. The higher the precision, the fewer non-churning users we incorrectly classify as churning.
- Recall is the fraction of correct answers among all positive observations. It tells us the percentage of churning customers who we correctly identified as churning. The higher the recall, the fewer churning customers we fail to identify.
- The ROC curve analyzes binary classification models at all the thresholds at once. The area under the ROC curve (AUC) tells us how well a model separates positive observations from negative ones. Because of its interpretability and wide applicability, AUC has become the default metric for evaluating binary classification models.
- K-fold cross-validation gives us a way to use all the training data for model validation: we split the data into K folds and use each fold in turn as a validation set, and the remaining K – 1 folds are used for training. As a result, instead of a single number, we have K values, one for each fold. We can use these numbers to understand the performance of a model on average as well as to estimate how volatile it is across different folds.
- K-fold cross-validation is the best way of tuning parameters and selecting the best model: it gives us a reliable estimate of the metric across multiple folds.

In the next chapter we look into deploying our model into a production environment.

Answers to exercises

- Exercise 4.1 B) A customer for whom we predicted "churn," but they didn't churn.
- Exercise 4.2 B) The percent of customers who actually churned among the customers who we predicted as churning.
- Exercise 4.3 A) The percent of correctly identified churned customers among all churned customers.
- Exercise 4.4 A) The ideal ranking model always scores churning customers higher than non-churning ones.

5 Deploying machine learning models

This chapter covers

- Saving models with Pickle
- Serving models with Flask
- Managing dependencies with Pipenv
- Making the service self-contained with Docker
- Deploying it to the cloud using AWS Elastic Beanstalk

As we continue to work with machine learning techniques, we'll keep using the project we already started: churn prediction. In chapter 3, we used Scikit-learn to build a model for identifying churning customers. After that, in chapter 4, we evaluated the quality of this model and selected the best parameter `C` using cross-validation.

We already have a model that lives in our Jupyter Notebook. Now we need to put this model into production, so other services can use the model to make decisions based on the output of our model.

In this chapter, we cover *model deployment*: the process of putting models to use. In particular, we see how to package a model inside a web service, so other services can use it. We also see how to deploy the web service to a production-ready environment.

5.1 Churn-prediction model

To get started with deployment we use the model we trained previously. First, in this section, we review how we can use the model for making predictions, and then we see how to save it with Pickle.

5.1.1 Using the model

To make it easier, we can continue the same Jupyter Notebook we used for chapters 3 and 4.

Let's use this model to calculate the probability of churning for the following customer:

```
customer = {
    'customerid': '8879-zkjof',
    'gender': 'female',
    'seniorcitizen': 0,
    'partner': 'no',
    'dependents': 'no',
    'tenure': 41,
    'phoneservice': 'yes',
    'multiplelines': 'no',
    'internetservice': 'dsl',
    'onlinesecurity': 'yes',
    'onlinebackup': 'no',
    'deviceprotection': 'yes',
    'techsupport': 'yes',
    'streamingtv': 'yes',
    'streamingmovies': 'yes',
    'contract': 'one_year',
    'paperlessbilling': 'yes',
    'paymentmethod': 'bank_transfer_(automatic)',
    'monthlycharges': 79.85,
    'totalcharges': 3320.75,
}
```

To predict whether this customer is going to churn, we can use the `predict` function we wrote in the previous chapter:

```
df = pd.DataFrame([customer])
y_pred = predict(df, dv, model)
y_pred[0]
```

This function needs a dataframe, so first we create a dataframe with one row — our customer. Next, we put it into the `predict` function. The result is a NumPy array with a single element — the predicted probability of churn for this customer:

```
0.059605
```

This means that this customer has a 6% probability of churning.

Now let's take a look at the `predict` function, which we wrote previously for applying the model to the customers in the validation set:

```
def predict(df, dv, model):
    cat = df[categorical + numerical].to_dict(orient='rows')
    X = dv.transform(cat)
    y_pred = model.predict_proba(X)[:, 1]
    return y_pred
```

Using it for one customer seems inefficient and unnecessary: we create a dataframe from a single customer only to convert this dataframe back to a dictionary later inside `predict`.

To avoid doing this unnecessary conversion, we can create a separate function for predicting the probability of churn for a single customer only. Let's call this function `predict_single`:

```
def predict_single(customer, dv, model):
    X = dv.transform([customer])
    y_pred = model.predict_proba(X)[:, 1]
    return y_pred[0]
```

Instead of passing a dataframe, passes a single customer

Vectorizes the customer: creates the matrix X

Applies the model to this matrix

Because we only have one customer, we only need the first element of the result.

Using it becomes simpler — we simply invoke it with our customer (a dictionary):

```
predict_single(customer, dv, model)
```

The result is the same: this customer has a 6% probability of churning.

We trained our model inside the Jupyter Notebook we started in chapter 3. This model lives there, and once we stop the Jupyter Notebook, the trained model will disappear. This means that now we can use it only inside the notebook and nowhere else. Next, we see how to address it.

5.1.2 Using Pickle to save and load the model

To be able to use it outside of our notebook, we need to save it, and then later, another process can load and use it (figure 5.1).

Pickle is a serialization/deserialization module that's already built into Python: using it, we can save an arbitrary Python object (with a few exceptions) to a file. Once we have a file, we can load the model from there in a different process.

NOTE "Pickle" can also be used as a verb: *pickling* an object in Python means saving it using the Pickle module.

SAVING THE MODEL

To save the model, we first import the Pickle module, and then use the `dump` function:

```
import pickle

with open('churn-model.bin', 'wb') as f_out:
    pickle.dump(model, f_out)
```

Specifies the file where we want to save

Saves the model to file with Pickle

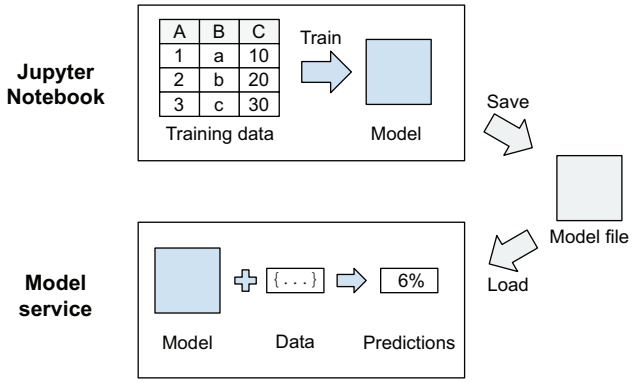


Figure 5.1 We train a model in a Jupyter Notebook. To use it, we first need to save it and then load it in a different process.

To save the model, we use the `open` function, which takes two arguments:

- The name of the file that we want to open. For us, it's churn-model.bin.
- The mode with which we open the file. For us, it's `wb`, which means we want to write to the file (`w`), and the file is binary (`b`) and not text — Pickle uses binary format for writing to files.

The `open` function returns `f_out` — the file descriptor we can use to write to the file.

Next, we use the `dump` function from Pickle. It also takes two arguments:

- The object we want to save. For us, it's `model`.
- The file descriptor, pointing to the output file, which is `f_out` for us.

Finally, we use the `with` construction in this code. When we open a file with `open`, we need to close it after we finish writing. When using `with`, it happens automatically. Without `with`, our code would look like this:

```
f_out = open('churn-model.bin', 'wb')
pickle.dump(model, f_out)
f_out.close()
```

In our case, however, saving just the model is not enough: we also have a `DictVectorizer` that we "trained" together with the model. We need to save both.

The simplest way of doing this is to put both of them in a tuple when pickling:

```
with open('churn-model.bin', 'wb') as f_out:
    pickle.dump((dv, model), f_out)
```

The object we save is a tuple with two elements.

LOADING THE MODEL

To load the model, we use the `load` function from Pickle. We can test it in the same Jupyter Notebook:

```
with open('churn-model.bin', 'rb') as f_in:
    dv, model = pickle.load(f_in)
```

Opens the file in read mode

Loads the tuple and unpacks it

We again use the `open` function, but this time, with a different mode: `rb`, which means we open it for reading (`r`), and the file is binary (`b`).

WARNING Be careful when specifying the mode. Accidentally specifying an incorrect mode may result in data loss: if you open an existing file with the `w` mode instead of `r`, it will overwrite the content.

Because we saved a tuple, we unpack it when loading, so we get both the vectorizer and the model at the same time.

WARNING Unpickling objects found on the internet is not secure: it can execute arbitrary code on your machine. Use it only for things you trust and things you saved yourself.

Let's create a simple Python script that loads the model and applies it to a customer.

We will call this file churn_serving.py. (In the book's GitHub repository, this file is called churn_serving_simple.py.) It contains

- The `predict_single` function that we wrote earlier
- The code for loading the model
- The code for applying the model to a customer

You can refer to appendix B to learn more about creating Python scripts.

First, we start with imports. For this script, we need to import Pickle and NumPy:

```
import pickle
import numpy as np
```

Next, let's put the `predict_single` function there:

```
def predict_single(customer, dv, model):
    X = dv.transform([customer])
    y_pred = model.predict_proba(X)[:, 1]
    return y_pred[0]
```

Now we can load our model:

```
with open('churn-model.bin', 'rb') as f_in:
    dv, model = pickle.load(f_in)
```

And apply it:

```
customer = {
    'customerid': '8879-zkjof',
    'gender': 'female',
```

```
    'seniorcitizen': 0,
    'partner': 'no',
    'dependents': 'no',
    'tenure': 41,
    'phoneservice': 'yes',
    'multiplelines': 'no',
    'internetservice': 'dsl',
    'onlinesecurity': 'yes',
    'onlinebackup': 'no',
    'deviceprotection': 'yes',
    'techsupport': 'yes',
    'streamingtv': 'yes',
    'streamingmovies': 'yes',
    'contract': 'one_year',
    'paperlessbilling': 'yes',
    'paymentmethod': 'bank_transfer_(automatic)',
    'monthlycharges': 79.85,
    'totalcharges': 3320.75,
}

prediction = predict_single(customer, dv, model)
```

Finally, let's display the results:

```
print('prediction: %.3f' % prediction)

if prediction >= 0.5:
    print('verdict: Churn')
else:
    print('verdict: Not churn')
```

After saving the file, we can run this script with Python:

```
python churn_serving.py
```

We should immediately see the results:

```
prediction: 0.059
verdict: Not churn
```

This way, we can load the model and apply it to the customer we specified in the script.

Of course, we aren't going to manually put the information about customers in the script. In the next section, we cover a more practical approach: putting the model into a web service.

5.2 Model serving

We already know how to load a trained model in a different process. Now we need to *serve* this model — make it available for others to use.

In practice, this usually means that a model is deployed as a web service, so other services can communicate with it, ask for predictions, and use the results to make their own decisions.

In this section, we see how to do it in Python with Flask — a Python framework for creating web services. First, we take a look at why we need to use a web service for it.

5.2.1 Web services

We already know how to use a model to make a prediction, but so far, we have simply hardcoded the features of a customer as a Python dictionary. Let's try to imagine how our model will be used in practice.

Suppose we have a service for running marketing campaigns. For each customer, it needs to determine the probability of churn, and if it's high enough, it will send a promotional email with discounts. Of course, this service needs to use our model to decide whether it should send an email.

One possible way of achieving this is to modify the code of the campaign service: load the model, and score the customers right in the service. This approach is good, but the campaign service needs to be in Python, and we need to have full control over its code.

Unfortunately, this situation is not always the case: it may be written in some other language, or a different team might be in charge of this project, which means we won't have the control we need.

The typical solution for this problem is putting a model inside a web service — a small service (a *microservice*) that only takes care of scoring customers.

So, we need to create a churn service — a service in Python that will serve the churn model. Given the features of a customer, it will respond with the probability of churn for this customer. For each customer, the campaign service will ask the churn service for the probability of churn, and if it's high enough, then we send a promotional email (figure 5.2).

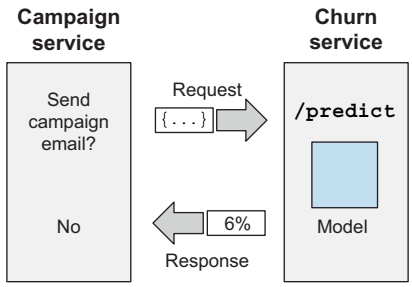


Figure 5.2 The churn service takes care of serving the churn-prediction model, making it possible for other services to use it.

This gives us another advantage: separation of concerns. If the model is created by data scientists, then they can take ownership of the service and maintain it, while the other team takes care of the campaign service.

One of the most popular frameworks for creating web services in Python is Flask, which we cover next.

5.2.2 Flask

The easiest way to implement a web service in Python is to use Flask. It's quite lightweight, requires little code to get started, and hides most of the complexity of dealing with HTTP requests and responses.

Before we put our model inside a web service, let's cover the basics of using Flask. For that, we'll create a simple function and make it available as a web service. After covering the basics, we'll take care of the model.

Suppose we have a simple Python function called `ping`:

```
def ping():
    return 'PONG'
```

It doesn't do much: when invoked, it simply responds with PONG. Let's use Flask to turn this function into a web service.

Anaconda comes with Flask preinstalled, but if you use a different Python distribution, you'll need to install it:

```
pip install flask
```

We will put this code in a Python file and will call it flask_test.py.

To be able to use Flask, we first need to import it:

```
from flask import Flask
```

Now we create a Flask app — the central object for registering functions that need to be exposed in the web service. We'll call our app test:

```
app = Flask('test')
```

Next, we need to specify how to reach the function by assigning it to an address, or a *route* in Flask terms. In our case, we want to use the `/ping` address:

```
@app.route('/ping', methods=['GET'])
def ping():
    return 'PONG'
```

Registers the /ping route, and assigns it to the ping function

This code uses decorators — an advanced Python feature that we don't cover in this book. We don't need to understand how it works in detail; it's enough to know that by putting `@app.route` on top of the function definition, we assign the `/ping` address of the web service to the `ping` function.

To run it, we only need one last bit:

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=9696)
```

The `run` method of `app` starts the service. We specify three parameters:

- `debug=True`. Restarts our application automatically when there are changes in the code.
- `host='0.0.0.0'`. Makes the web service public; otherwise, it won't be possible to reach it when it's hosted on a remote machine (e.g., in AWS).
- `port=9696`. The port that we use to access the application.

We're ready to start our service now. Let's do it:

```
python flask_test.py
```

When we run it, we should see the following:

```
* Serving Flask app "test" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production
    deployment.
  Use a production WSGI server instead.
* Debug mode: on
* Running on http://0.0.0.0:9696/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger PIN: 162-129-136
```

This means that our Flask app is now running and ready to get requests. To test it, we can use our browser: open it and type localhost:9696/ping in the address bar. If you run it on a remote server, you should replace localhost with the address of the server. (For AWS EC2, use the public DNS hostname. Make sure that the port 9696 is open in the security group of your EC2 instance: go to the security group, and add a custom TCP rule with the port 9696 and the source 0.0.0.0/0.) The browser should respond with PONG (figure 5.3).

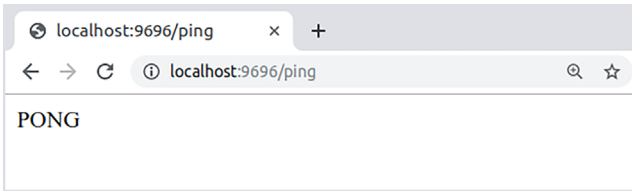


Figure 5.3 The easiest way to check if our application works is to use a web browser.

Flask logs all the requests it receives, so we should see a line indicating that there was a GET request on the `/ping` route:

```
127.0.0.1 - - [02/Apr/2020 21:59:09] "GET /ping HTTP/1.1" 200 -
```

As we can see, Flask is quite simple: with fewer than 10 lines of code, we created a web service.

Next, we'll see how to adjust our script for churn prediction and also turn it into a web service.

5.2.3 Serving churn model with Flask

We've learned a bit of Flask, so now we can come back to our script and convert it to a Flask application.

To score a customer, our model needs to get the features, which means that we need a way of transferring some data from one service (the campaign service) to another (the churn service).

As a data exchange format, web services typically use JSON (Javascript Object Notation). It's similar to the way we define dictionaries in Python:

```
{
    "customerid": "8879-zkjof",
    "gender": "female",
    "seniorcitizen": 0,
    "partner": "no",
    "dependents": "no",
    ...
}
```

To send data, we use POST requests, not GET: POST requests can include the data in the request, whereas GET cannot.

Thus, to make it possible for the campaign service to get predictions from the churn service, we need to create a `/predict` route that accepts POST requests. The churn service will parse JSON data about a customer and respond in JSON as well (figure 5.4).

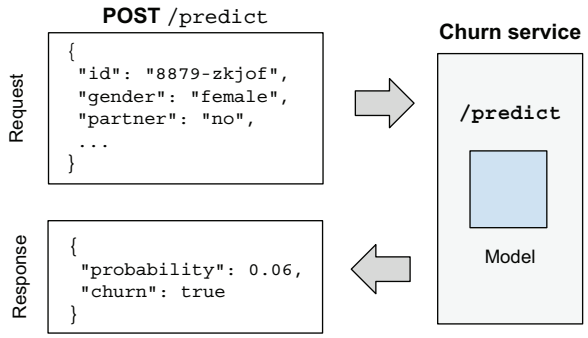


Figure 5.4 To get predictions, we POST the data about a customer in JSON to the `/predict` route and get the probability of churn in response.

Now we know what we want to do, so let's start modifying the churn_serving.py file.

First, we add a few more imports at the top of the file:

```
from flask import Flask, request, jsonify
```

Although previously we imported only `Flask`, now we need to import two more things:

- `request`: To get the content of a POST request
- `jsonsify`: To respond with JSON

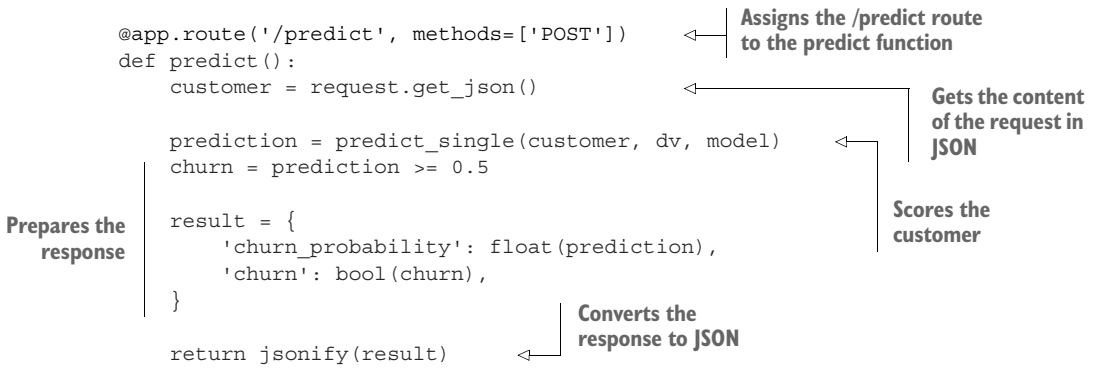
Next, create the Flask app. Let's call it churn:

```
app = Flask('churn')
```

Now we need to create a function that

- Gets the customer data in a request
- Invokes `predict_simple` to score the customer
- Responds with the probability of churn in JSON

We'll call this function `predict` and assign it to the `/predict` route:



To assign the route to the function, we use the `@app.route` decorator, where we also tell Flask to expect POST requests only.

The core content of the `predict` function is similar to what we did in the script previously: it takes a customer, passes it to `predict_single`, and does some work with the result.

Finally, let's add the last two lines for running the Flask app:

```
if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=9696)
```

We're ready to run it:

```
python churn_serving.py
```

After running it, we should see a message saying that the app started and is now waiting for incoming requests:

```
 * Serving Flask app "churn" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
     deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:9696/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
```

Testing this code is a bit more difficult than previously: this time, we need to use POST requests and include the customer we want to score in the body of the request.

The simplest way of doing this is to use the requests library in Python. It also comes preinstalled in Anaconda, but if you use a different distribution, you can install it with `pip`:

```
pip install requests
```

We can open the same Jupyter Notebook that we used previously and test the web service from there.

First, import requests:

```
import requests
```

Now, make a POST request to our service

```
url = 'http://localhost:9696/predict'
response = requests.post(url, json=customer)
result = response.json()
```

The URL where the service lives

Sends the customer (as JSON) in the POST request

Parses the response as JSON

The `results` variable contains the response from the churn service:

```
{'churn': False, 'churn_probability': 0.0596059075831639 1}
```

This is the same information we previously saw in the terminal, but now we got it as a response from a web service.

> **NOTE** Some tools, like Postman (https://www.postman.com/), make it easier to test web services. We don't cover Postman in this book, but you're free to give it a try.

If the campaign service used Python, this is exactly how it could communicate with the churn service and decide who should get promotional emails.

With just a few lines of code, we created a working web service that runs on our laptop. In the next section, we'll see how to manage dependencies in our service and prepare it for deployment.

5.3 Managing dependencies

For local development, Anaconda is a perfect tool: it has almost all the libraries we may ever need. This, however, also has a downside: it takes up 4 GB when unpacked, which is too large. For production, we prefer to have only the libraries we actually need.

Additionally, different services have different requirements. Often, these requirements conflict, so we cannot use the same environment for running multiple services at the same time.

In this section, we see how to manage dependencies of our application in an isolated way that doesn't interfere with other services. We cover two tools for this: Pipenv, for managing Python libraries, and Docker, for managing the system dependencies such as the operating system and the system libraries.

5.3.1 Pipenv

To serve the churn model, we only need a few libraries: NumPy, Scikit-learn, and Flask. So, instead of bringing in the entire Anaconda distribution with all its libraries, we can get a fresh Python installation and install only the libraries we need with `pip`:

```
pip install numpy scikit-learn flask
```

Before we do that, let's think for a moment about what happens when we use `pip` to install a library:

- We run `pip install library` to install a Python package called Library (let's suppose it exists).
- Pip goes to PyPI.org (the Python package index — a repository with Python packages), and gets and installs the latest version of this library. Let's say, it's version 1.0.0.

After installing it, we develop and test our service using this particular version. Everything works great. Later, our colleagues want to help us with the project, so they also run `pip install` to set up everything on their machine — except this time, the latest version turns out to be 1.3.1.

If we're unlucky, versions 1.0.0 and 1.3.1 might not be compatible with each other, meaning that the code we wrote for version 1.0.0 won't work for version 1.3.1.

It's possible to solve this problem by specifying the exact version of the library when installing it with `pip`:

```
pip install library==1.0.0
```

Unfortunately, a different problem may appear: what if some of our colleagues already have version 1.3.1 installed, and they already used it for some other services?

In this case, they cannot go back to using version 1.0.0: it could cause their code to stop working.

We can solve these problems by creating a *virtual environment* for each project — a separate Python distribution with nothing else but libraries required for this particular project.

Pipenv is a tool that makes managing virtual environments easier. We can install it with pip:

```
pip install pipenv
```

After that, we use `pipenv` instead of `pip` for installing dependencies:

```
pipenv install numpy scikit-learn flask
```

When running it, we'll see that first, it configures the virtual environment, and then it installs the libraries:

```
Running virtualenv with interpreter .../bin/python3
✓ Successfully created virtual environment!
Virtualenv location: ...
Creating a Pipfile for this project…
Installing numpy…
Adding numpy to Pipfile's [packages]…
✓ Installation Succeeded
Installing scikit-learn…
Adding scikit-learn to Pipfile's [packages]…
✓ Installation Succeeded
Installing flask…
Adding flask to Pipfile's [packages]…
✓ Installation Succeeded
Pipfile.lock not found, creating…
Locking [dev-packages] dependencies…
Locking [packages] dependencies…
⠙ Locking...
```

After finishing the installation, it creates two files: Pipenv and Pipenv.lock.

The Pipenv file looks pretty simple:

```
[[source]]
name = "pypi"
url = "https://pypi.org/simple"
verify_ssl = true

[dev-packages]

[packages]
numpy = "*"
scikit-learn = "*"
flask = "*"

[requires]
python_version = "3.7"
```

We see that this file contains a list of libraries as well as the version of Python we use.

The other file — Pipenv.lock — contains the specific versions of the libraries that we used for the project. The file is too large to show in its entirety here, but let's take a look at one of the entries in the file:

```
"flask": {
    "hashes": [
        "sha256:4efa1ae2d7c9865af48986de8aeb8504...",
        "sha256:8a4fdd8936eba2512e9c85df320a37e6..."
    ],
    "index": "pypi",
    "version": "==1.1.2"
}
```

As we can see, it records the exact version of the library that was used during installation. To make sure the library doesn't change, it also saves the hashes — the checksums that can be used to validate that in the future we download the exact same version of the library. This way, we "lock" the dependencies to specific versions. By doing this, we make sure that in the future we will not have surprises with two incompatible versions of the same library.

If somebody needs to work on our project, they simply need to run the `install` command:

```
pipenv install
```

This step will first create a virtual environment and then install all the required libraries from Pipenv.lock.

> **IMPORTANT** Locking the version of a library is important for reproducibility in the future and helps us avoid having unpleasant surprises with code incompatibility.

After all the libraries are installed, we need to activate the virtual environment — this way, our application will use the correct versions of the libraries. We do it by running the `shell` command:

```
pipenv shell
```

It tells us that it's running in a virtual environment:

```
Launching subshell in virtual environment…
```

Now we can run our script for serving:

```
python churn_serving.py
```

Alternatively, instead of first explicitly entering the virtual environment and then running the script, we can perform these two steps with just one command:

```
pipenv run python churn_serving.py
```

The `run` command in Pipenv simply runs the specified program in the virtual environment.

Regardless of the way we run it, we should see exactly the same output as previously:

```
 * Serving Flask app "churn" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
     deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://0.0.0.0:9696/ (Press CTRL+C to quit)
```

When we test it with requests, we see the same output:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

You most likely also noticed the following warning in the console:

```
 * Environment: production
   WARNING: This is a development server. Do not use it in a production
     deployment.
   Use a production WSGI server instead.
```

The built-in Flask web server is indeed for development only: it's very easy to use for testing our application, but it won't work reliably under load. We should use a proper WSGI server instead, as the warning suggests.

WSGI stands for *web server gateway interface*, which is a specification describing how Python applications should handle HTTP requests. The details of WSGI are not important for the purposes of this book, so we won't cover it in detail.

We will, however, address the warning by installing a production WSGI server. We have multiple possible options in Python, and we'll use Gunicorn.

> **NOTE** Gunicorn doesn't work on Windows: it relies on features specific to Linux and Unix (which includes MacOS). A good alternative that also works on Windows is Waitress. Later, we will use Docker, which will solve this problem — it runs Linux inside a container.

Let's install it with Pipenv:

```
pipenv install gunicorn
```

This command installs the library and includes it as a dependency in the project by adding it to the Pipenv and Pipenv.lock files.

Let's run our application with Gunicorn:

```
pipenv run gunicorn --bind 0.0.0.0:9696 churn_serving:app
```

If everything goes well, we should see the following messages in the terminal:

```
[2020-04-13 22:58:44 +0200] [15705] [INFO] Starting gunicorn 20.0.4
[2020-04-13 22:58:44 +0200] [15705] [INFO] Listening at: http://0.0.0.0:9696
(15705)
[2020-04-13 22:58:44 +0200] [15705] [INFO] Using worker: sync
[2020-04-13 22:58:44 +0200] [16541] [INFO] Booting worker with pid: 16541
```

Unlike the Flask built-in web server, Gunicorn is ready for production, so it will not have any problems under load when we start using it.

If we test it with the same code as previously, we see the same answer:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

Pipenv is a great tool for managing dependencies: it isolates the required libraries into a separate environment, thus helping us avoid conflicts between different versions of the same package.

In the next section, we look at Docker, which allows us to isolate our application even further and ensure it runs smoothly anywhere.

5.3.2 Docker

We have learned how to manage Python dependencies with Pipenv. However, some of the dependencies live outside of Python. Most importantly, these dependencies include the operating system (OS) as well as the system libraries.

For example, we might use Ubuntu version 16.04 for developing our service, but if some of our colleagues use Ubuntu version 20.04, they may run into trouble when trying to execute the service on their laptop.

Docker solves this "but it works on my machine" problem by also packaging the OS and the system libraries into a *Docker container* — a self-contained environment that works anywhere where Docker is installed (figure 5.5).

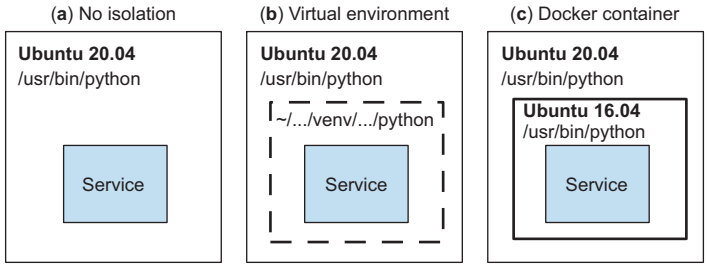


Figure 5.5 In case of no isolation (a), the service runs with system Python. In virtual environments (b), we isolate the dependencies of our service inside the environment. In Docker containers (c), we isolate the entire environment of the service, including the OS and system libraries.

Once the service is packaged into a Docker container, we can run it on the *host machine* — our laptop (regardless of the OS) or any public cloud provider.

Let's see how to use it for our project. We assume you already have Docker installed. Please refer to appendix A for details on how to install it.

First, we need to create a *Docker image* — the description of our service that includes all the settings and dependencies. Docker will later use the image to create a container. To do it, we need a Dockerfile — a file with instructions on how the image should be created (figure 5.6).

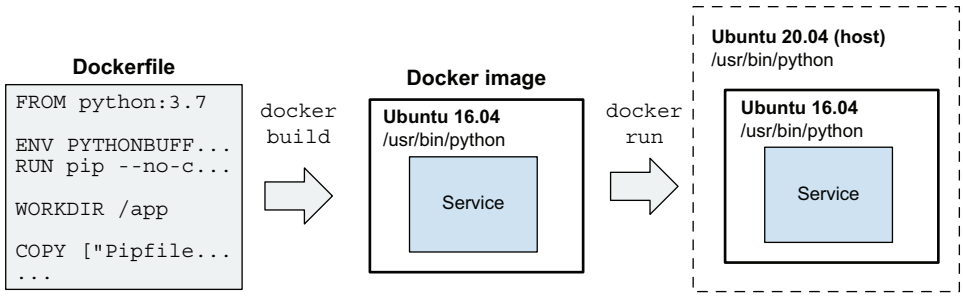


Figure 5.6 We build an image using instructions from Dockerfile. Then we can run this image on a host machine.

Let's create a file with name Dockerfile and the following content (note that the file shouldn't include the annotations):

```
FROM python:3.7.5-slim                          # Specifies the base image

ENV PYTHONUNBUFFERED=TRUE                       # Sets a special Python settings
                                                # for being able to see logs

RUN pip --no-cache-dir install pipenv           # Installs Pipenv

WORKDIR /app                                    # Sets the working directory to /app

COPY ["Pipfile", "Pipfile.lock", "./"]          # Copies the Pipenv files

RUN pipenv install --deploy --system && \       # Installs the dependencies from
    rm -rf /root/.cache                         # the Pipenv files

COPY ["*.py", "churn-model.bin", "./"]          # Copies our code as well as the model

EXPOSE 9696                                     # Opens the port that our web service uses

ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "churn_serving:app"]
                                                # Specifies how the service should be started
```

That's a lot of information to unpack, especially if you have never seen Dockerfiles previously. Let's go line by line.

First, we specify the base Docker image:

```
FROM python:3.7.5-slim
```

We use this image as the starting point and build our own image on top of that. Typically, the base image already contains the OS and the system libraries like Python itself, so we need to install only the dependencies of our project. In our case, we use `python:3.7.5-slim`, which is based on Debian 10.2 and contains Python version 3.7.5 and `pip`. You can read more about the Python base image in Docker hub (https://hub.docker.com/_/python) — the service for sharing Docker images.

All Dockerfiles should start with the `FROM` statement.

Next, we set the `PYTHONUNBUFFERED` environmental variable to `TRUE`:

```
ENV PYTHONUNBUFFERED=TRUE
```

Without this setting, we won't be able to see the logs when running Python scripts inside Docker.

Then, we use `pip` to install Pipenv:

```
RUN pip --no-cache-dir install pipenv
```

The `RUN` instruction in Docker simply runs a shell command. By default, `pip` saves the libraries to a cache, so later they can be installed faster. We don't need that in a Docker container, so we use the `--no-cache-dir` setting.

Then, we specify the working directory:

```
WORKDIR /app
```

This is roughly equivalent to the `cd` command in Linux (change directory), so everything we will run after that will be executed in the /app folder.

Then, we copy the Pipenv files to the current working directory (i.e., /app):

```
COPY ["Pipfile", "Pipfile.lock", "./"]
```

We use these files for installing the dependencies with Pipenv:

```
RUN pipenv install --deploy --system && \
    rm -rf /root/.cache
```

Previously, we simply used `pipenv install` for doing this. Here, we include two extra parameters: `--deploy` and `--system`. Inside Docker, we don't need to create a virtual environment — our Docker container is already isolated from the rest of the system. Setting these parameters allows us to skip creating a virtual environment and use the system Python for installing all the dependencies.

After installing the libraries, we clean the cache to make sure our Docker image doesn't grow too big.

Then, we copy our project files as well as the pickled model:

```
COPY ["*.py", "churn-model.bin", "./"]
```

Next, we specify which port our application will use. In our case, it's 9696:

```
EXPOSE 9696
```

Finally, we tell Docker how our application should be started:

```
ENTRYPOINT ["gunicorn", "--bind", "0.0.0.0:9696", "churn_serving:app"]
```

This is the same command we used previously when running Gunicorn locally.

Let's build the image. We do it by running the `build` command in Docker:

```
docker build -t churn-prediction .
```

The `-t` flag lets us set the tag name for the image, and the final parameter — the dot — specifies the directory with the Dockerfile. In our case, it means that we use the current directory.

When we run it, the first thing Docker does is download the base image:

```
Sending build context to Docker daemon  51.71kB
Step 1/11 : FROM python:3.7.5-slim
3.7.5-slim: Pulling from library/python
000eee12ec04: Downloading  24.84MB/27.09MB
ddc2d83f8229: Download complete
735b0bee82a3: Downloading  19.56MB/28.02MB
8c69dcedfc84: Download complete
495e1cccc7f9: Download complete
```

Then it executes each line of the Dockerfile one by one:

```
Step 2/9 : ENV PYTHONUNBUFFERED=TRUE
 ---> Running in d263b412618b
Removing intermediate container d263b412618b
 ---> 7987e3cf611f
Step 3/9 : RUN pip --no-cache-dir install pipenv
 ---> Running in e8e9d329ed07
Collecting pipenv
 ...
```

At the end, Docker tells us that it successfully built an image and tagged it as churn-prediction:latest:

```
Successfully built d9c50e4619a1
Successfully tagged churn-prediction:latest
```

We're ready to use this image to start a Docker container. Use the `run` command for that:

```
docker run -it -p 9696:9696 churn-prediction:latest
```

We specify a few parameters here:

- The `-it` flag tells Docker that we run it from our terminal and we need to see the results.
- The `-p` parameter specifies the port mapping. `9696:9696` means to map the port 9696 on the container to the port 9696 on the host machine.
- Finally, we need the image name and tag, which in our case is `churn-prediction :latest`.

Now our service is running inside a Docker container, and we can connect to it using port 9696 (figure 5.7). This is the same port we used for our application previously.

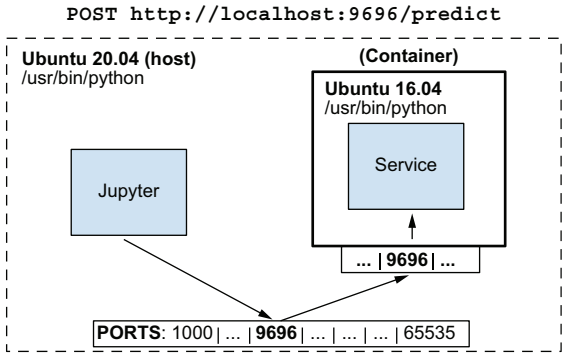


Figure 5.7 The 9696 port on the host machine is mapped to the 9696 port of the container, so when we send a request to `localhost:9696`, it's handled by our service in Docker.

Let's test it using the same code. When we run it, we'll see the same response:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

Docker makes it easy to run services in a reproducible way. With Docker, the environment inside the container always stays the same. This means that if we can run our service on a laptop, it will work anywhere else.

We already tested our application on our laptop, so now let's see how to run it on a public cloud and deploy it to AWS.

5.4 Deployment

We don't run production services on our laptops; we need special servers for that.

In this section, we'll cover one possible option for that: Amazon Web Services, or AWS. We decided to choose AWS for its popularity — we're not affiliated with Amazon or AWS.

Other popular public clouds include Google Cloud, Microsoft Azure, and Digital Ocean. We don't cover them in this book, but you should be able to find similar instructions online and deploy a model to your favourite cloud provider.

This section is optional, so you can safely skip it. To follow the instructions in this section, you need to have an AWS account and configure the AWS command-line tool (CLI). Please refer to appendix A to see how to set it up.

5.4.1 AWS Elastic Beanstalk

AWS provides a lot of services, and we have many possible ways of deploying a web service there. For example, you can rent an EC2 machine (a server in AWS) and manually set up a service on it, use a "serverless" approach with AWS Lambda, or use a range of other services.

In this section, we'll use AWS Elastic Beanstalk, which is one of the simplest ways of deploying a model to AWS. Additionally, our service is simple enough, so it's possible to stay within the free-tier limits. In other words, we can use it for free for the first year.

Elastic Beanstalk automatically takes care of many things that we typically need in production, including

- Deploying our service to EC2 instances
- Scaling up: adding more instances to handle the load during peak hours
- Scaling down: removing these instances when the load goes away
- Restarting the service if it crashes for any reason
- Balancing the load between instances

We'll also need a special utility — Elastic Beanstalk command-line interface (CLI) — to use Elastic Beanstalk. The CLI is written in Python, so we can install it with `pip`, like any other Python tool.

However, because we use Pipenv, we can add it as a development dependency. This way, we'll install it only for our project and not systemwide.

```
pipenv install awsebcli --dev
```

> **NOTE** Development dependencies are the tools and libraries that we use for developing our application. Usually, we need them only locally and don't need them in the actual package deployed to production.

After installing Elastic Beanstalk, we can enter the virtual environment of our project:

```
pipenv shell
```

Now the CLI should be available. Let's check it:

```
eb --version
```

It should print the version:

```
EB CLI 3.18.0 (Python 3.7.7)
```

Next, we run the initialization command:

```
eb init -p docker churn-serving
```

Note that we use `-p docker`: this way, we specify that this is a Docker-based project.

If everything is fine, it creates a couple of files, including a config.yml file in .elasticbeanstalk folder.

Now we can test our application locally by using `local run` command:

```
eb local run --port 9696
```

This should work in the same way as in the previous section with Docker: it'll first build an image and then run the container.

To test it, we can use the same code as previously and get the same answer:

```
{'churn': False, 'churn_probability': 0.05960590758316391}
```

After verifying that it works well locally, we're ready to deploy it to AWS. We can do that with one command:

```
eb create churn-serving-env
```

This simple command takes care of setting up everything we need, from the EC2 instances to auto-scaling rules:

```
Creating application version archive "app-200418_120347".
Uploading churn-serving/app-200418_120347.zip to S3. This may take a while.
Upload Complete.
Environment details for: churn-serving-env
  Application name: churn-serving
  Region: us-west-2
  Deployed Version: app-200418_120347
  Environment ID: e-3xkqdzdjbq
  Platform: arn:aws:elasticbeanstalk:us-west-2::platform/Docker running on
     64bit Amazon Linux 2/3.0.0
  Tier: WebServer-Standard-1.0
  CNAME: UNKNOWN
  Updated: 2020-04-18 10:03:52.276000+00:00
Printing Status:
2020-04-18 10:03:51    INFO    createEnvironment is starting.
 -- Events -- (safe to Ctrl+C)
```

It'll take a few minutes to create everything. We can monitor the process and see what it's doing in the terminal.

When it's ready, we should see the following information:

```
2020-04-18 10:06:53    INFO    Application available at churn-serving-
env.5w9pp7bkmj.us-west-2.elasticbeanstalk.com.
2020-04-18 10:06:53    INFO    Successfully launched environment: churn-
serving-env
```

The URL (churn-serving-env.5w9pp7bkmj.us-west-2.elasticbeanstalk.com) in the logs is important: this is how we reach our application. Now we can use this URL to make predictions (figure 5.8).

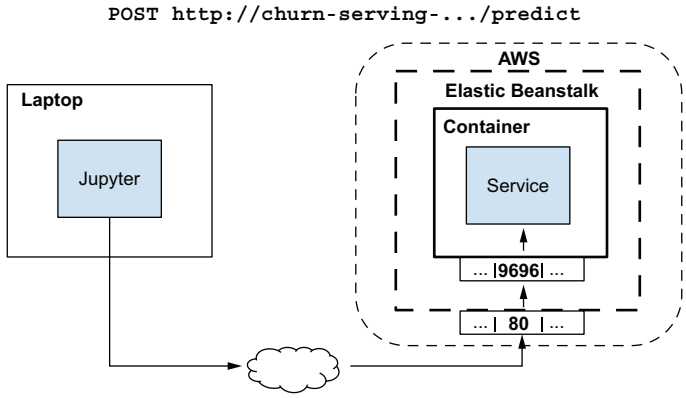


Figure 5.8 Our service is deployed inside a container on AWS Elastic Beanstalk. To reach it, we use its public URL.

Let's test it:

```
host = 'churn-serving-env.5w9pp7bkmj.us-west-2.elasticbeanstalk.com'
url = 'http://%s/predict' % host
response = requests.post(url, json=customer)
result = response.json()
result
```

As previously, we should see the same response:

```
{'churn': False, 'churn_probability': 0.05960590758316393}
```

That's all! We have a running service.

> **WARNING** This is a toy example, and the service we created is accessible by anyone in the world. If you do it inside an organization, the access should be restricted as much as possible. It's not difficult to extend this example to be secure, but it's outside the scope of this book. Consult the security department at your company before doing it at work.

We can do everything from the terminal using the CLI, but it's also possible to manage it from the AWS Console. To do so, we find Elastic Beanstalk there and select the environment we just created (figure 5.9).

To turn it off, choose Terminate deployment in the Environment actions menu using the AWS Console.

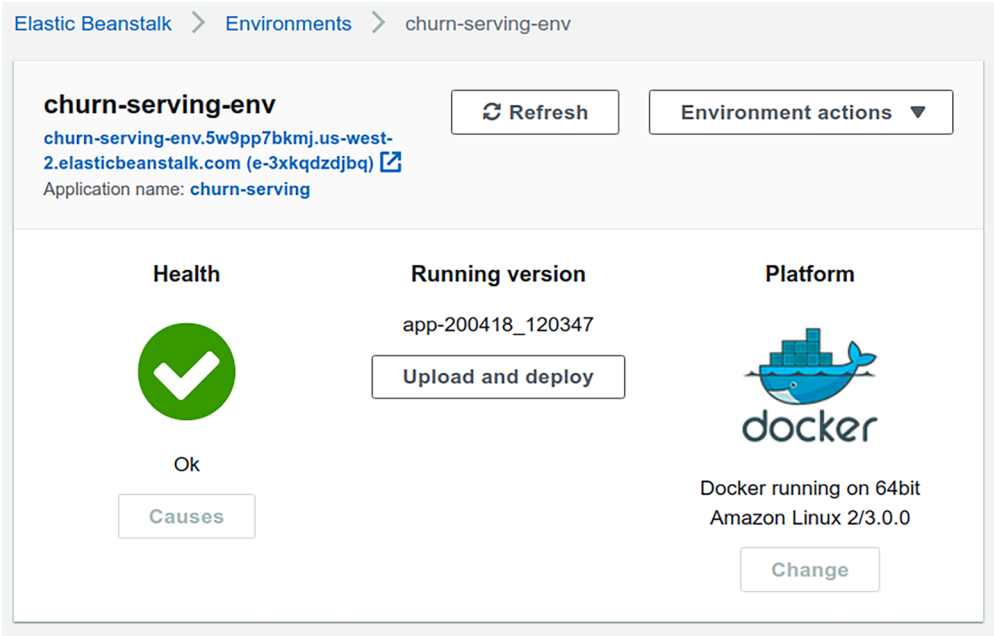


Figure 5.9 We can manage the Elastic Beanstalk environment in the AWS Console.

WARNING Even though Elastic Beanstalk is free-tier eligible, we should always be careful and turn it off as soon as we no longer need it.

Alternatively, we use the CLI to do it:

```
eb terminate churn-serving-env
```

After a few minutes, the deployment will be removed from AWS and the URL will no longer be accessible.

AWS Elastic Beanstalk is a great tool for getting started with serving machine learning models. More advanced ways of doing it involve container orchestration systems like AWS ECS or Kubernetes or "serverless" with AWS Lambda. We will come back to this topic in chapters 8 and 9 when covering the deployment of deep learning models.

5.5 Next steps

We've learned about Pipenv and Docker and deployed our model to AWS Elastic Beanstalk. Try these other things to expand your skills on your own.

5.5.1 Exercises

Try the following exercises to further explore the topics of model deployment:

- If you don't use AWS, try to repeat the steps from section 5.4 on any other cloud provider. For example, you could try Google Cloud, Microsoft Azure, Heroku, or Digital Ocean.
- Flask is not the only way of creating web services in Python. You can try alternative frameworks like FastAPI (https://fastapi.tiangolo.com/), Bottle (https://github.com/bottlepy/bottle), or Falcon (https://github.com/falconry/falcon).

5.5.2 Other projects

You can continue other projects from the previous chapters and make them available as a web service as well. For example:

- The car-price prediction model we created in chapter 2
- The self-study projects from chapter 3: the lead scoring project and the default prediction project.

Summary

- Pickle is a serialization/deserialization library that comes built into Python. We can use it to save a model we trained in Jupyter Notebook and load it in a Python script.
- The simplest way of making a model available for others is wrapping it into a Flask web service.
- Pipenv is a tool for managing Python dependencies by creating virtual environments, so dependencies of one Python project don't interfere with dependencies of another Python project.
- Docker makes it possible to isolate the service completely from other services by packaging into a Docker container not only the Python dependencies but also the system dependencies, as well as the operational system itself.
- AWS Elastic Beanstalk is a simple way to deploy a web service. It takes care of managing EC2 instances, scaling the service up and down, and restarting if something fails.

In the next chapter, we continue learning about classification but with a different type of model — decision trees.

6 Decision trees and ensemble learning

This chapter covers

- Decision trees and the decision tree learning algorithm
- Random forests: putting multiple trees together into one model
- Gradient boosting as an alternative way of combining decision trees

In chapter 3, we described the binary classification problem and used the logistic regression model to predict if a customer is going to churn.

In this chapter, we also solve a binary classification problem, but we use a different family of machine learning models: tree-based models. Decision trees, the simplest tree-based model, are nothing but a sequence of if-then-else rules put together. We can combine multiple decision trees into an ensemble to achieve better performance. We cover two tree-based ensemble models: random forest and gradient boosting.

The project we prepared for this chapter is default prediction: we predict whether or not a customer will fail to pay back a loan. We learn how to train decision trees and random forest models with Scikit-learn and explore XGBoost — a library for implementing gradient boosting models.

6.1 Credit risk scoring project

Imagine that we work at a bank. When we receive a loan application, we need to make sure that if we give the money, the customer will be able to pay it back. Every application carries a risk of *default* — the failure to return the money.

We'd like to minimize this risk: before agreeing to give a loan, we want to score the customer and assess the chances of default. If it's too high, we reject the application. This process is called "credit risk scoring."

Machine learning can be used for calculating the risk. For that, we need a dataset with loans, where for each application, we know whether or not it was paid back successfully. Using this data, we can build a model for predicting the probability of default, and we can use this model to assess the risk of future borrowers not repaying the money.

This is what we do in this chapter: use machine learning to calculate the risk of default. The plan for the project is the following:

- First, we get the data and do some initial preprocessing.
- Next, we train a decision tree model from Scikit-learn for predicting the probability of default.
- After that, we explain how decision trees work and which parameters the model has and show how to adjust these parameters to get the best performance.
- Then we combine multiple decision trees into one model — a random forest. We look at its parameters and tune them to achieve the best predictive performance.
- Finally, we explore a different way of combining decision trees — gradient boosting. We use XGBoost, a highly efficient library that implements gradient boosting. We'll train a model and tune its parameters.

Credit risk scoring is a binary classification problem: the target is positive ("1") if the customer defaults and negative ("0") otherwise. For evaluating our solution, we'll use AUC (area under the ROC curve), which we covered in chapter 4. AUC describes how well our model can separate the cases into positive and negative ones.

The code for this project is available in the book's GitHub repository at https://github.com/alexeygrigorev/mlbookcamp-code (in the chapter-06-trees folder).

6.1.1 Credit scoring dataset

For this project, we use a dataset from a data mining course at the Polytechnic University of Catalonia (https://www.cs.upc.edu/~belanche/Docencia/mineria/mineria.html). The dataset describes the customers (seniority, age, marital status, income, and other characteristics), the loan (the requested amount, the price of the item), and its status (paid back or not).

We use a copy of this dataset available on GitHub at https://github.com/gastonstat/CreditScoring/. Let's download it.

First, create a folder for our project (e.g., chapter-06-credit-risk), and then use `wget` to get it:

```
wget https://github.com/gastonstat/CreditScoring/raw/master/CreditScoring.csv
```

Alternatively, you can enter the link to your browser and save it to the project folder.

Next, start a Jupyter Notebook server if it's not started yet:

```
jupyter notebook
```

Go to the project folder, and create a new notebook (e.g., chapter-06-credit-risk).

As usual, we begin by importing Pandas, NumPy, Seaborn, and Matplotlib:

```
import pandas as pd
import numpy as np

import seaborn as sns
from matplotlib import pyplot as plt
%matplotlib inline
```

After we press Ctrl-Enter, the libraries are imported and we're ready to read the data with Pandas:

```
df = pd.read_csv('CreditScoring.csv')
```

Now the data is loaded, so let's take an initial look at it and see if we need to do any preprocessing before we can use it.

6.1.2 Data cleaning

To use a dataset for our task, we need to look for any issues in the data and fix them.

Let's start by looking at the first rows of the DataFrame, generated by the `df.head()` function (figure 6.1).

```
df.head()
```

	Status	Seniority	Home	Time	Age	Marital	Records	Job	Expenses	Income	Assets	Debt	Amount	Price
0	1	9	1	60	30	2	1	3	73	129	0	0	800	846
1	1	17	1	60	58	3	1	1	48	131	0	0	1000	1658
2	2	10	2	36	46	2	2	3	90	200	3000	0	2000	2985
3	1	0	1	60	24	1	1	1	63	182	2500	0	900	1325
4	1	0	1	36	26	1	1	1	46	107	0	0	310	910

Figure 6.1 The first five rows of the credit scoring dataset

First, we can see that all the column names start with a capital letter. Before doing anything else, let's lowercase all the column names and make it consistent with other projects (figure 6.2):

```
df.columns = df.columns.str.lower()
```

```
df.columns = df.columns.str.lower()
df.head()
```

	status	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price
0	1	9	1	60	30	2	1	3	73	129	0	0	800	846
1	1	17	1	60	58	3	1	1	48	131	0	0	1000	1658
2	2	10	2	36	46	2	2	3	90	200	3000	0	2000	2985
3	1	0	1	60	24	1	1	1	63	182	2500	0	900	1325
4	1	0	1	36	26	1	1	1	46	107	0	0	310	910

Figure 6.2 The DataFrame with lowercase column names

We can see that the DataFrame has the following columns:

- status: whether the customer managed to pay back the loan (1) or not (2)
- seniority: job experience in years
- home: type of homeownership: renting (1), a homeowner (2), and others
- time: period planned for the loan (in months)
- age: age of the client
- marital [status]: single (1), married (2), and others
- records: whether the client has any previous records: no (1), yes (2) (It's not clear from the dataset description what kind of records we have in this column. For the purposes of this project, we may assume that it's about records in the bank's database.)
- job: type of job: full-time (1), part-time (2), and others
- expenses: how much the client spends per month
- income: how much the client earns per month
- assets: total worth of all the assets of the client
- debt: amount of credit debt
- amount: requested amount of the loan
- price: price of an item the client wants to buy

Although most of the columns are numerical, some are categorical: status, home, marital [status], records, and job. The values we see in the DataFrame, however, are numbers, not strings. This means that we need to translate them to their actual names. In the GitHub repository with the dataset is a script that decodes the numbers to categories (https://github.com/gastonstat/CreditScoring/blob/master/Part1_CredScoring_Processing.R). Originally, this script was written in R, so we need to translate it to Pandas.

We start with the status column. The value "1" means "OK," the value "2" means "default," and "0" means that the value is missing — let's replace it with "unk" (short for "unknown").

In Pandas, we can use `map` for converting the numbers to strings. For that, we first define the dictionary with mapping from the current value (number) to the desired value (string):

```
status_values = {
    1: 'ok',
    2: 'default',
    0: 'unk'
}
```

Now we can use this dictionary to do the mapping:

```
df.status = df.status.map(status_values)
```

It creates a new series, which we immediately write back to the DataFrame. As a result, the values in the status column are overwritten and look more meaningful (figure 6.3).

```
status_values = {
    1: 'ok',
    2: 'default',
    0: 'unk'
}

df.status = df.status.map(status_values)
df.head()
```

	status	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price
0	ok	9	1	60	30	2	1	3	73	129	0	0	800	846
1	ok	17	1	60	58	3	1	1	48	131	0	0	1000	1658
2	default	10	2	36	46	2	2	3	90	200	3000	0	2000	2985
3	ok	0	1	60	24	1	1	1	63	182	2500	0	900	1325
4	ok	0	1	36	26	1	1	1	46	107	0	0	310	910

Figure 6.3 To translate the original values in the status column (numbers) to a more meaningful representation (strings), we use the `map` method.

We repeat the same procedure for all the other columns. First, we'll do it for the home column:

```
home_values = {
    1: 'rent',
    2: 'owner',
    3: 'private',
    4: 'ignore',
    5: 'parents',
    6: 'other',
```

```
    0: 'unk'
}

df.home = df.home.map(home_values)
```

Next, let's do it for the marital, records, and job columns:

```
marital_values = {
    1: 'single',
    2: 'married',
    3: 'widow',
    4: 'separated',
    5: 'divorced',
    0: 'unk'
}

df.marital = df.marital.map(marital_values)


records_values = {
    1: 'no',
    2: 'yes',
    0: 'unk'
}

df.records = df.records.map(records_values)


job_values = {
    1: 'fixed',
    2: 'parttime',
    3: 'freelance',
    4: 'others',
    0: 'unk'
}

df.job = df.job.map(job_values)
```

After these transformations, the columns with categorical variables contain the actual values, not numbers (figure 6.4).

```
df.head()
```

	status	seniority	home	time	age	marital	records	job	expenses	income	assets	debt	amount	price
0	ok	9	rent	60	30	married	no	freelance	73	129	0	0	800	846
1	ok	17	rent	60	58	widow	no	fixed	48	131	0	0	1000	1658
2	default	10	owner	36	46	married	yes	freelance	90	200	3000	0	2000	2985
3	ok	0	rent	60	24	single	no	fixed	63	182	2500	0	900	1325
4	ok	0	rent	36	26	single	no	fixed	46	107	0	0	310	910

Figure 6.4 The values of categorical variables are translated from integers to strings.

As the next step, let's take a look at numerical columns. First, let's check the summary statistics for each of the columns: min, mean, max, and others. To do so, we can use the `describe` method of the DataFrame:

```
df.describe().round()
```

> **NOTE** The output of `describe` may be confusing. In our case, there are values in scientific notation like 1.000000e+08 or 8.703625e+06. To force Pandas to use a different notation, we use `round`: it removes the fractional part of a number and rounds it to the closest integer.

It gives us an idea of how the distribution of the values in each column looks (figure 6.5).

```
df.describe().round()
```

	seniority	time	age	expenses	income	assets	debt	amount	price
count	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0
mean	8.0	46.0	37.0	56.0	763317.0	1060341.0	404382.0	1039.0	1463.0
std	8.0	15.0	11.0	20.0	8703625.0	10217569.0	6344253.0	475.0	628.0
min	0.0	6.0	18.0	35.0	0.0	0.0	0.0	100.0	105.0
25%	2.0	36.0	28.0	35.0	80.0	0.0	0.0	700.0	1118.0
50%	5.0	48.0	36.0	51.0	120.0	3500.0	0.0	1000.0	1400.0
75%	12.0	60.0	45.0	72.0	166.0	6000.0	0.0	1300.0	1692.0
max	48.0	72.0	68.0	180.0	99999999.0	99999999.0	99999999.0	5000.0	11140.0

Figure 6.5 The summary of all numerical columns of the dataframe. We notice that some of them have 99999999 as the max value.

One thing we notice immediately is that the max value is `99999999` in some cases. This is quite suspicious. As it turns out, it's an artificial value — this is how missing values are encoded in this dataset.

Three columns have this problem: income, assets, and debt. Let's replace this big number with NaN for these columns:

```
for c in ['income', 'assets', 'debt']:
    df[c] = df[c].replace(to_replace=99999999, value=np.nan)
```

We use the `replace` method, which takes two values:

- `to_replace`: the original value ("99999999," in our case)
- `value`: the target value ("NaN," in our case)

After this transformation, no more suspicious numbers appear in the summary (figure 6.6).

```
df.describe().round()
```

	seniority	time	age	expenses	income	assets	debt	amount	price
count	4455.0	4455.0	4455.0	4455.0	4421.0	4408.0	4437.0	4455.0	4455.0
mean	8.0	46.0	37.0	56.0	131.0	5403.0	343.0	1039.0	1463.0
std	8.0	15.0	11.0	20.0	86.0	11573.0	1246.0	475.0	628.0
min	0.0	6.0	18.0	35.0	0.0	0.0	0.0	100.0	105.0
25%	2.0	36.0	28.0	35.0	80.0	0.0	0.0	700.0	1118.0
50%	5.0	48.0	36.0	51.0	120.0	3000.0	0.0	1000.0	1400.0
75%	12.0	60.0	45.0	72.0	165.0	6000.0	0.0	1300.0	1692.0
max	48.0	72.0	68.0	180.0	959.0	300000.0	30000.0	5000.0	11140.0

Figure 6.6 The summary statistics after replacing large values with NaN

Before we finish with the dataset preparation, let's look at our target variable `status`:

```
df.status.value_counts()
```

The output of `value_counts` shows the count of each value:

```
ok         3200
default    1254
unk           1
Name: status, dtype: int64
```

Notice that there's one row with "unknown" status: we don't know whether or not this client managed to pay back the loan. For our project, this row is not useful, so let's remove it from the dataset:

```
df = df[df.status != 'unk']
```

In this case, we don't really "remove" it: we create a new DataFrame where we don't have records with "unknown" status.

By looking at the data, we have identified a few important issues in the data and addressed them.

For this project, we skip a more detailed exploratory data analysis like we did for chapter 2 (the car-price prediction project) and chapter 3 (churn prediction project), but you're free to repeat the steps we covered there for this project as well.

6.1.3 Dataset preparation

Now our dataset is cleaned, and we're almost ready to use it for model training. Before we can do that, we need to do a few more steps:

- Split the dataset into train, validation, and test.
- Handle missing values.

- Use one-hot encoding to encode categorical variables.
- Create the feature matrix X and the target variable y.

Let's start by splitting the data. We will split the data into three parts:

- Training data (60% of the original dataset)
- Validation data (20%)
- Test data (20%)

Like previously, we'll use `train_test_split` from Scikit-learn for that. Because we cannot split it into three datasets at once, we'll need to split two times (figure 6.7). First we'll hold out 20% of data for testing, and then split the remaining 80% into training and validation:

```
from sklearn.model_selection import train_test_split

df_train_full, df_test = train_test_split(df, test_size=0.2, random_state=11)
df_train, df_val = train_test_split(df_train_full, test_size=0.25,
     random_state=11)
```

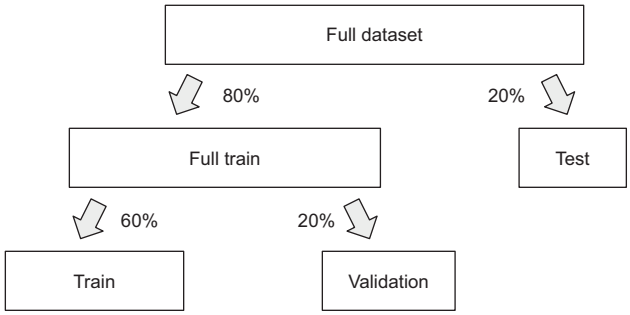


Figure 6.7 Because `train_test_split` can split a dataset into only two parts, but we need three, we perform the split two times.

When splitting for the second time, we put aside 25% of data instead of 20% (`test_size=0.25`). Because `df_train_full` contains 80% of records, one-quarter (i.e., 25%) of 80% corresponds to 20% of the original dataset.

To check the size of our datasets, we can use the `len` function:

```
len(df_train), len(df_val), len(df_test)
```

When running it, we get the following output:

```
(2672, 891, 891)
```

So, for training, we will use approximately 2,700 examples and almost 900 for validation and testing.

The outcome we want to predict is `status`. We will use it to train a model, so it's our *y* — the target variable. Because our objective is to determine if somebody fails to pay back their loan, the positive class is `default`. This means that *y* is "1" if the client defaulted and "0" otherwise. It's quite simple to implement:

```
y_train = (df_train.status == 'default').values
y_val = (df_val.status == 'default').values
```

Now we need to remove `status` from the DataFrames. If we don't do it, we may accidentally use this variable for training. For that, we use the `del` operator:

```
del df_train['status']
del df_val['status']
```

Next, we'll take care of *X* — the feature matrix.

From the initial analysis, we know our data contains missing values — we added these NaNs ourselves. We can replace the missing values with zero:

```
df_train = df_train.fillna(0)
df_val = df_val.fillna(0)
```

To use categorical variables, we need to encode them. In chapter 3, we applied the one-hot encoding technique for that. In one-hot encoding, each value is encoded as "1" if it's present ("hot") or "0" if it's absent ("cold"). To implement it, we used `DictVectorizer` from Scikit-learn.

`DictVectorizer` needs a list of dictionaries, so we first need to convert the DataFrames into this format:

```
dict_train = df_train.to_dict(orient='records')
dict_val = df_val.to_dict(orient='records')
```

Each dictionary in the result represents a row from the DataFrame. For example, the first record in `dict_train` looks like this:

```
{'seniority': 10,
 'home': 'owner',
 'time': 36,
 'age': 36,
 'marital': 'married',
 'records': 'no',
 'job': 'freelance',
 'expenses': 75,
 'income': 0.0,
 'assets': 10000.0,
 'debt': 0.0,
 'amount': 1000,
 'price': 1400}
```

This list of dictionaries now can be used as input to `DictVectorizer`:

```
from sklearn.feature_extraction import DictVectorizer

dv = DictVectorizer(sparse=False)

X_train = dv.fit_transform(dict_train)
X_val = dv.transform(dict_val)
```

As a result, we have feature matrices for both train and validation datasets. Please refer to chapter 3 for more details on doing one-hot encoding with Scikit-learn.

Now we're ready to train a model! In the next section, we cover the simplest tree model: decision tree.

6.2 Decision trees

A *decision tree* is a data structure that encodes a series of if-then-else rules. Each node in a tree contains a condition. If the condition is satisfied, we go to the right side of the tree; otherwise, we go to the left. In the end we arrive at the final decision (figure 6.8).

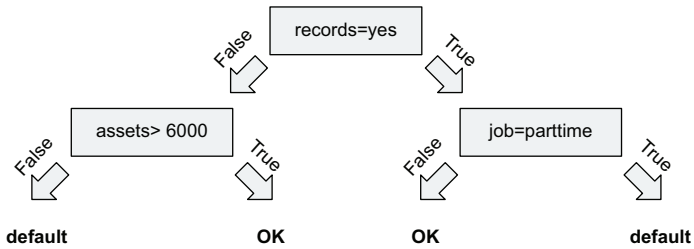


Figure 6.8 A decision tree consists of nodes with conditions. If the condition in a node is satisfied, we go right; otherwise, we go left.

It's quite easy to represent a decision tree as a set of `if-else` statements in Python. For example:

```
def assess_risk(client):
    if client['records'] == 'yes':
        if client['job'] == 'parttime':
            return 'default'
        else:
            return 'ok'
    else:
        if client['assets'] > 6000:
            return 'ok'
        else:
            return 'default'
```

With machine learning, we can extract these rules from data automatically. Let's see how we can do it.

6.2.1 Decision tree classifier

We'll use Scikit-learn for training a decision tree. Because we're solving a classification problem, we need to use `DecisionTreeClassifier` from the `tree` package. Let's import it:

```
from sklearn.tree import DecisionTreeClassifier
```

Training the model is as simple as invoking the `fit` method:

```
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)
```

To check if the result is good, we need to evaluate the predictive performance of the model on the validation set. Let's use AUC (area under the ROC curve) for that.

Credit risk scoring is a binary classification problem, and for cases like that, AUC is one of the best evaluation metrics. As you may recall from our discussion in chapter 4, AUC shows how well a model separates positive examples from negative examples. It has a nice interpretation: it describes the probability that a randomly chosen positive example ("default") has a higher score than a randomly chosen negative example ("OK"). This is a relevant metric for the project: we want risky clients to have higher scores than nonrisky ones. For more details on AUC, refer to chapter 4.

Like previously, we'll use an implementation from Scikit-learn, so let's import it:

```
from sklearn.metrics import roc_auc_score
```

First, we evaluate the performance on the training set. Because we chose AUC as the evaluation metric, we need scores, not hard predictions. As we know from chapter 3, we need to use the `predict_proba` method for that:

```
y_pred = dt.predict_proba(X_train)[:, 1]
roc_auc_score(y_train, y_pred)
```

When we execute it, we see that the score is 100% — the perfect score. Does it mean that we can predict default without errors? Let's check the score on validation before jumping to conclusions:

```
y_pred = dt.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

After running, we see that AUC on validation is only 65%.

We just observed a case of *overfitting*. The tree learned the training data so well that it simply memorized the outcome for each customer. However, when we applied it to the validation set, the model failed. The rules it extracted from the data turned out to be too specific to the training set, so it worked poorly for customers it didn't see during training. In such cases, we say that the model cannot *generalize*.

Overfitting happens when we have a complex model with enough power to remember all the training data. If we force the model to be simpler, we can make it less powerful and improve the model's ability to generalize.

We have multiple ways to control the complexity of a tree. One option is to restrict its size: we can specify the `max_depth` parameter, which controls the maximum number of levels. The more levels a tree has, the more complex rules it can learn (figure 6.9).

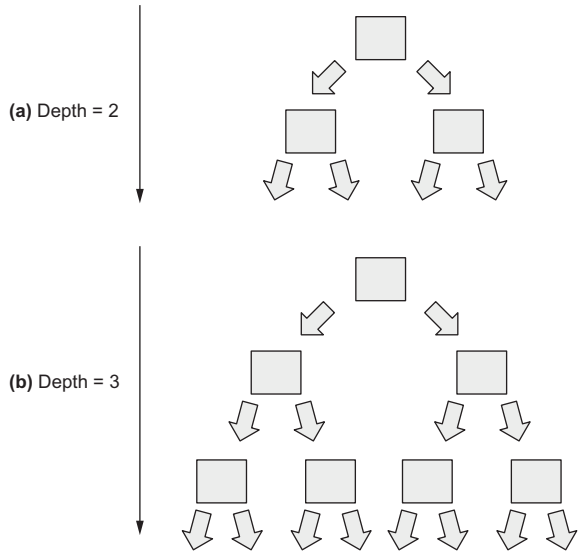


Figure 6.9 A tree with more levels can learn more complex rules. A tree with two levels is less complex than a tree with three levels and, thus, less prone to overfitting.

The default value for the `max_depth` parameter is `None`, which means that the tree can grow as large as possible. We can try a smaller value and compare the results.

For example, we can change it to 2:

```
dt = DecisionTreeClassifier(max_depth=2)
dt.fit(X_train, y_train)
```

To visualize the tree we just learned, we can use the `export_text` function from the `tree` package:

```
from sklearn.tree import export_text

tree_text = export_text(dt, feature_names=dv.feature_names_)
print(tree_text)
```

We only need to specify the names of features using the `feature_names` parameter. We can get it from the `DictVectorizer`. When we print it, we get the following:

```
|--- records=no <= 0.50
|   |--- seniority <= 6.50
|   |   |--- class: True
|   |--- seniority >  6.50
|   |   |--- class: False
|--- records=no >  0.50
|   |--- job=parttime <= 0.50
|   |   |--- class: False
|   |--- job=parttime >  0.50
|   |   |--- class: True
```

Each line in the output corresponds to a node with a condition. If the condition is true, we go inside and repeat the process until we arrive at the final decision. At the end, if class is `True`, then the decision is "default," and otherwise it's "OK."

The condition `records=no > 0.50` means that a customer has no records. Recall that we use one-hot encoding to represent `records` with two features: `records=yes` and `records=no`. For a customer with no records, `records=no` is set to "1" and `records=yes` to "0." Thus, "`records=no > 0.50` is true when the value for `records` is `no` (figure 6.10).

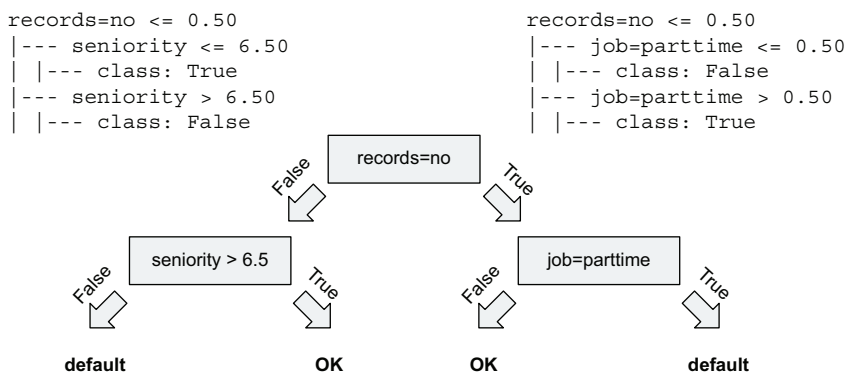


Figure 6.10 The tree we learned with `max_depth` set to 2

Let's check the score:

```
y_pred = dt.predict_proba(X_train)[:, 1]
auc = roc_auc_score(y_train, y_pred)
print('train auc', auc)

y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('validation auc', auc)
```

We see that the score on train dropped:

```
train auc: 0.705
val auc: 0.669
```

Previously, the performance on the training set was 100%, but now it's only 70.5%. It means that the model can no longer memorize all the outcomes from the training set.

However, the score on the validation set is better: it's 66.9%, which is an improvement over the previous result (65%). By making it less complex, we improved the ability of our model to generalize. Now it's better at predicting the outcomes for customers it hasn't seen previously.

However, this tree has another problem — it's too simple. To make it better, we need to tune the model: try different parameters, and see which ones lead to the best AUC. In addition to `max_depth`, we can control other parameters. To understand what these parameters mean and how they influence the model, let's take a step back and look at how decision trees learn rules from data.

6.2.2 Decision tree learning algorithm

To understand how a decision tree learns from data, let's simplify the problem. First, we'll use a much smaller dataset with just one feature: `assets` (figure 6.11).

	assets	status
0	8000	default
1	2000	OK
2	0	OK
3	6000	OK
4	6000	default
5	9000	default

Figure 6.11 A smaller dataset with one feature: `assets`. The target variable is `status`.

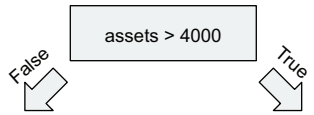
Second, we'll grow a very small tree, with a single node.

The only feature we have in the dataset is `assets`. This is why the condition in the node will be `assets > T`, where *T* is a threshold value that we need to determine. If the condition is true, we'll predict "OK," and if it's false, our prediction will be "default" (figure 6.12).

The condition `assets > T` is called a *split*. It splits the dataset into two groups: the data points that satisfy the condition and the data points that do not.

If `T` is 4000, then we have customers with more than $4,000 in assets (on the right) and the customers with less than $4,000 in assets (on the left) (figure 6.13).

Figure 6.12 A simple decision tree with only one node. The node contains a condition `assets > T`. We need to find the best value for *T*.

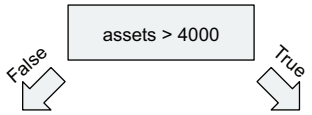


	assets	status
1	2000	default
2	0	default
5	4000	OK
7	3000	default

	assets	status
0	8000	default
3	5000	OK
4	5000	OK
6	9000	OK

Figure 6.13 The condition in a node splits the dataset into two parts: data points that satisfy the condition (on the right) and data points that don't (on the left).

Now we turn these groups into *leaves* — the decision nodes — by taking the most frequent status in each group and using it as the final decision. In our example, "default" is the most frequent outcome in the left group and "OK" in the right (figure 6.14).



3 default
1 OK

	assets	status
1	2000	default
2	0	default
5	4000	OK
7	3000	default

	assets	status
0	8000	default
3	5000	OK
4	5000	OK
6	9000	OK

1 default
3 OK

Figure 6.14 The most frequent outcome on the left is "default." For the group on the right, it's "OK."

Thus, if a customer has more than $4,000 in assets, our decision is "OK," and, otherwise, it's "default" `assets > 4000` (figure 6.15).

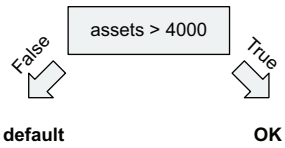


Figure 6.15 By taking the most frequent outcome in each group and assigning it to leaves, we get the final decision tree

IMPURITY

These groups should be as homogeneous as possible. Ideally, each group should contain only observations of one class. In this case, we call these groups *pure*.

For example, if we have a group of four customers with outcomes ["default," "default," "default," "default"], it's pure: it contains only customers who defaulted. But a group ["default," "default," "default," "OK"] is impure: there's one customer who didn't default.

When training a decision tree model, we want to find such T that the *impurity* of both groups is minimal.

So, the algorithm for finding T is quite simple:

- Try all possible values of T.
- For each T, split the dataset into left and right groups and measure their impurity.
- Select T that has the lowest degree of impurity.

We can use different criteria for measuring impurity. The easiest one to understand is the *misclassification rate*, which says how many observations in a group don't belong to the majority class.

> **NOTE** Scikit-learn uses more advanced split criteria such as entropy and the Gini impurity. We do not cover them in this book, but the idea is the same: they measure the degree of impurity of the split.

Let's calculate the misclassification rate for the split $T = 4000$ (figure 6.16):

- For the left group, the majority class is "default." There are four data points in total, and one doesn't belong to "default." The misclassification rate is 25% (1/4).
- For the right group, "OK" is the majority class, and there's one "default." Thus, the misclassification rate is also 25% (1/4).
- To calculate the overall impurity of the split, we can take the average across both groups. In this case, the average is 25%.

> **NOTE** In reality, instead of taking the simple average across both groups, we take a weighted average — we weight each group proportionally to its size. To simplify calculations, we use the simple average in this chapter.

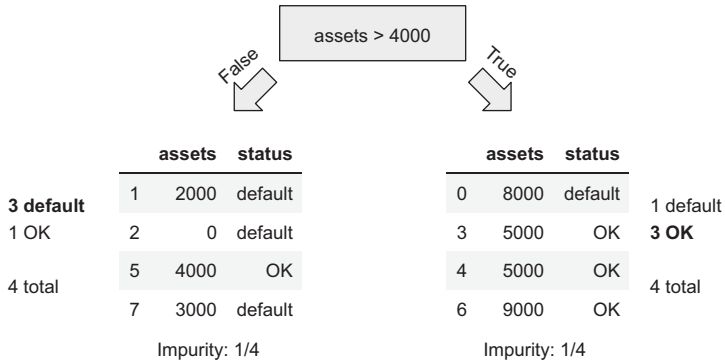


Figure 6.16 For `assets > 4000`, the misclassification rate for both groups is one-quarter.

$T = 4000$ is not the only possible split for `assets`. Let's try other values for T such as 2000, 3000, and 5000 (figure 6.17):

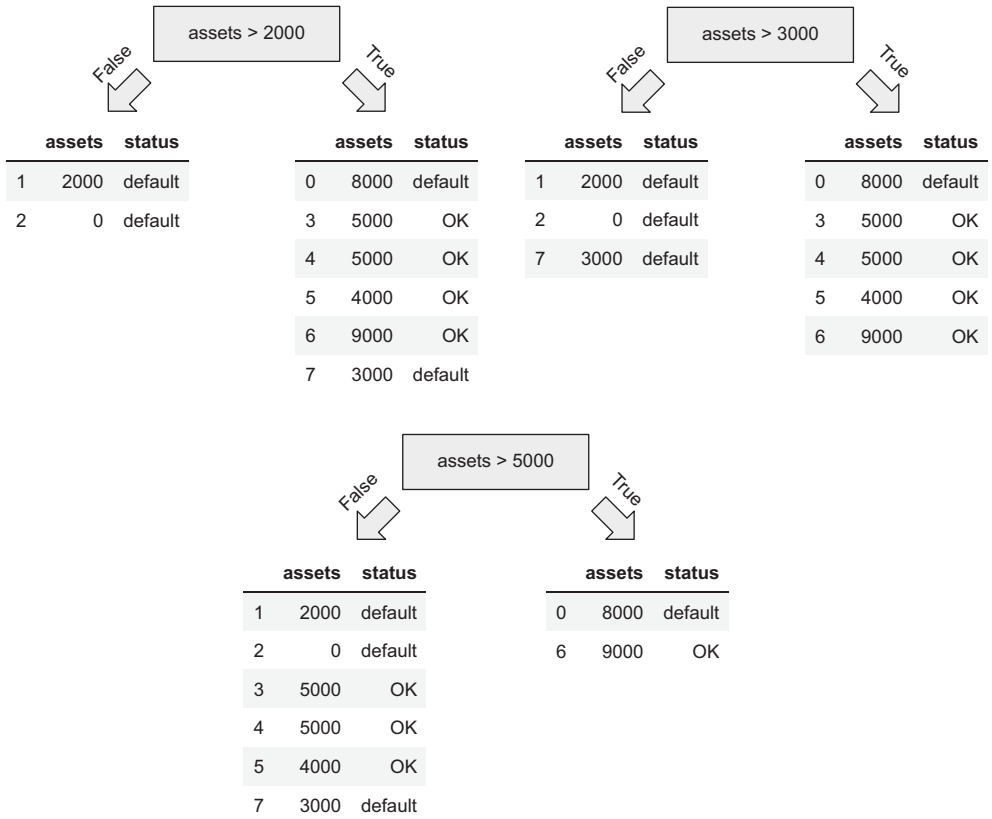


Figure 6.17 In addition to `assets > 4000`, we can try other values of T, such as 2000, 3000, and 5000.

- For $T = 2000$, we have 0% impurity on the left (0/2, all are "default") and 33.3% impurity on the right (2/6, 2 out of 6 are "default," the rest are "OK"). The average is 16.6%.
- For $T = 3000$, 0% on the left and 20% (1/5) on the right. The average is 10%.
- For $T = 5000$, 50% (3/6) on the left and 50% (1/2) on the right. The average is 50%.

The best average impurity is 10% for $T = 3000$: we got zero mistakes for the left tree and only one (out of five rows) for the right. So, we should select 3000 as the threshold for our final model (figure 6.18).

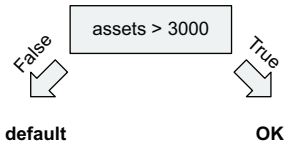


Figure 6.18 The best split for this dataset is `assets > 3000`.

SELECTING THE BEST FEATURE FOR SPLITTING

Now let's make the problem a bit more complex and add another feature to the dataset: `debt` (figure 6.19).

	assets	debt	status
0	8000	3000	default
1	2000	1000	default
2	0	1000	default
3	5000	1000	OK
4	5000	1000	OK
5	4000	1000	OK
6	9000	500	OK
7	3000	2000	default

Figure 6.19 A dataset with two features: `assets` and `debt`. The target variable is `status`.

Previously we had only one feature: `assets`. We knew for sure that it would be used for splitting the data. Now we have two features, so in addition to selecting the best threshold for splitting, we need to figure out which feature to use.

The solution is simple: we try all the features, and for each feature select the best threshold.

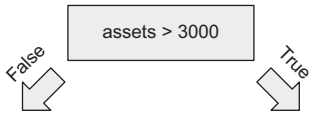
Let's modify the training algorithm to include this change:

- For each feature, try all possible thresholds.
- For each threshold value T, measure the impurity of the split.
- Select the feature and the threshold with the lowest impurity possible.

Let's apply this algorithm to our dataset:

- We already identified that for `assets`, the best T is 3000. The average impurity of this split is 10%.
- For `debt`, the best T is 1000. In this case, the average impurity is 17%.

So, the best split is `asset > 3000` (figure 6.20).



	assets	debt	status
1	2000	1000	default
2	0	1000	default
7	3000	2000	default

	assets	debt	status
0	8000	3000	default
3	5000	1000	OK
4	5000	1000	OK
5	4000	1000	OK
6	9000	500	OK

Figure 6.20 The best split is `assets > 3000`, which has the average impurity of 10%.

The group on the left is already pure, but the group on the right is not. We can make it less impure by repeating the process: split it again!

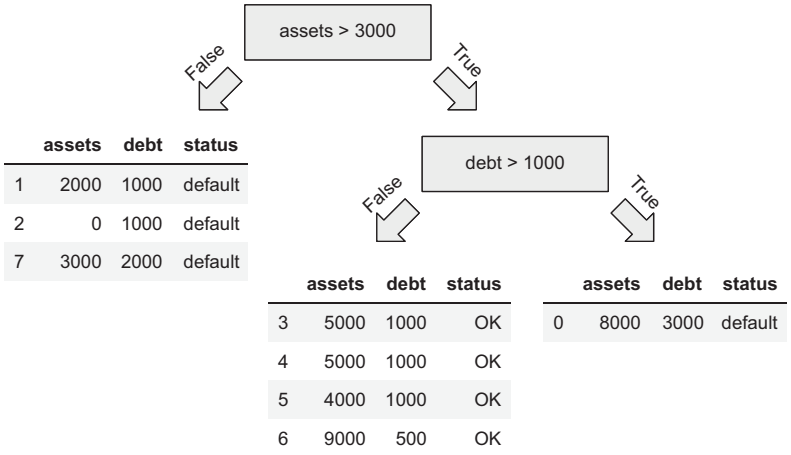
When we apply the same algorithm to the dataset on the right, we find that the best split condition is `debt > 1000`. We have two levels in the tree now — or we can say that the depth of this tree is 2 (figure 6.21).

Before the decision tree is ready, we need to do the last step: convert the groups into decision nodes. For that, we take the most frequent status in each group. This way, we get a decision tree (figure 6.22).

STOPPING CRITERIA

When training a decision tree, we can keep splitting the data until all the groups are pure. This is exactly what happens when we don't put any restrictions on the trees in Scikit-learn. As we've seen, the resulting model becomes too complex, which leads to overfitting.

We solved this problem by using the `max_depth` parameter — we restricted the tree size and didn't let it grow too big.



	assets	debt	status
1	2000	1000	default
2	0	1000	default
7	3000	2000	default

	assets	debt	status
3	5000	1000	OK
4	5000	1000	OK
5	4000	1000	OK
6	9000	500	OK

	assets	debt	status
0	8000	3000	default

Figure 6.21 By repeating the algorithm recursively to the group on the right, we get a tree with two levels.

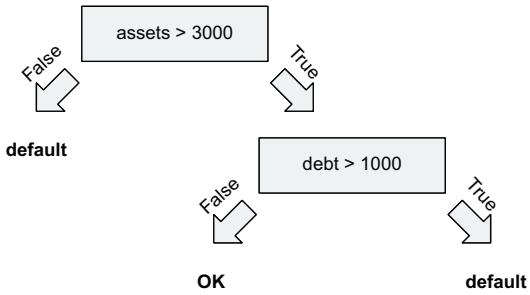


Figure 6.22 The groups are already pure, so the most frequent status is the only status each group has. We take this status as the final decision in each leaf.

To decide if we want to continue splitting the data, we use *stopping criteria* — criteria that describe if we should add another split in the tree or stop.

The most common stopping criteria are

- The group is already pure.
- The tree reached the depth limit (controlled by the `max_depth` parameter).
- The group is too small to continue splitting (controlled by the `min_samples_leaf` parameter).

By using these criteria to stop earlier, we force our model to be less complex and, therefore, reduce the risk of overfitting.

Let's use this information to adjust the training algorithm:

- Find the best split:
  - For each feature try all possible threshold values.
  - Use the one with the lowest impurity.
- If the maximum allowed depth is reached, stop.
- If the group on the left is sufficiently large and it's not pure yet, repeat on the left.
- If the group on the right is sufficiently large and it's not pure yet, repeat on the right.

Even though this is a simplified version of the decision tree learning algorithm, it should provide you enough intuition about the internals of the learning process.

Most important, we know two parameters control the complexity of the model. By changing these parameters, we can improve the performance of the model.

Exercise 6.1

We have a dataset with 10 features and need to add another feature to this dataset. What happens with the speed of training?

a With one more feature, training takes longer.
b The number of features does not affect the speed of training.

6.2.3 Parameter tuning for decision tree

The process of finding the best set of parameters is called *parameter tuning*. We usually do it by changing the model and checking its score on the validation dataset. In the end, we use the model with the best validation score.

As we have just learned, we can tune two parameters:

- `max_depth`
- `min_leaf_size`

These two are the most important ones, so we will adjust only them. You can check the other parameters in the official documentation (https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html).

When we trained our model previously, we restricted the depth of the tree to 2, but we didn't touch `min_leaf_size`. With this, we got an AUC of 66% on the validation set.

Let's find the best parameters.

We start by tuning max_depth. For that, we iterate over a few reasonable values and see what works best:

```
for depth in [1, 2, 3, 4, 5, 6, 10, 15, 20, None]:
    dt = DecisionTreeClassifier(max_depth=depth)
```

```
dt.fit(X_train, y_train)
y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('%4s -> %.3f' % (depth, auc))
```

The value `None` means that there's no restriction on depth, so the tree will grow as large as it can.

When we run this code, we see that `max_depth` of 5 gives the best AUC (76.6%), followed by 4 and 6 (figure 6.23).

```
   1 -> 0.606
   2 -> 0.669
   3 -> 0.739
   4 -> 0.761
   5 -> 0.766
   6 -> 0.754
  10 -> 0.685
  15 -> 0.671
  20 -> 0.657
None -> 0.657
```

Optimal values for max_depth

Figure 6.23 The optimal value for depth is 5 (76.6%) followed by 4 (76.1%) and 6 (75.4%).

Next, we tune `min_leaf_size`. For that, we iterate over the three best parameters of `max_depth`, and for each, go over different values of `min_leaf_size`:

```
for m in [4, 5, 6]:
    print('depth: %s' % m)

    for s in [1, 5, 10, 15, 20, 50, 100, 200]:
        dt = DecisionTreeClassifier(max_depth=m, min_samples_leaf=s)
        dt.fit(X_train, y_train)
        y_pred = dt.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)
        print('%s -> %.3f' % (s, auc))

    print()
```

After running it, we see that the best AUC is 78.5% with parameters `min_sample_leaf=15` and `max_depth=6` (table 6.1).

NOTE As we see, the value we use for `min_leaf_size` influences the best value of `max_depth`. You can experiment with a wider range of values for `max_depth` to tweak the performance further.

Table 6.1 AUC on validation set for different values of `min_leaf_size` (rows) and `max_depth` (columns)

	depth=4	depth=5	depth=6
1	0.761	0.766	0.754
5	0.761	0.768	0.760
10	0.761	0.762	0.778
15	0.764	0.772	**0.785**
20	0.761	0.774	0.774
50	0.753	0.768	0.770
100	0.756	0.763	0.776
200	0.747	0.759	0.768

We have found the best parameters, so let's use them to train the final model:

```
dt = DecisionTreeClassifier(max_depth=6, min_samples_leaf=15)
dt.fit(X_train, y_train)
```

Decision trees are simple and effective models, but they become even more powerful when we combine many trees together. Next, we'll see how we can do it to achieve even better predictive performance.

6.3 Random forest

For a moment, let's suppose that we don't have a machine learning algorithm to help us with credit risk scoring. Instead, we have a group of experts.

Each expert can independently decide if we should approve a loan application or reject it. An individual expert may make a mistake. However, it's less likely that all the experts together decide to accept the application, but the customer fails to pay the money back.

Thus, we can ask all the experts independently and then combine their verdicts into the final decision, for example, by using the majority vote (figure 6.24).

This idea also applies to machine learning. One model individually may be wrong, but if we combine the output of multiple models into one, the chance of an incorrect answer is smaller. This concept is called *ensemble learning*, and a combination of models is called an *ensemble*.

For this to work, the models need to be different. If we train the same decision tree model 10 times, they will all predict the same output, so it's not useful at all.

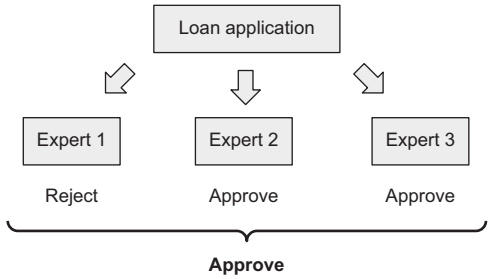


Figure 6.24 A group of experts can make a decision better than a single expert individually.

The easiest way to have different models is to train each tree on a different subset of features. For example, suppose we have three features: `assets`, `debts`, and `price`. We can train three models:

- The first will use `assets` and `debts`.
- The second will use `debts` and `price`.
- The last one will use `assets` and `price`.

With this approach, we'll have different trees, each making its own decisions (figure 6.25). But when we put their predictions together, their mistakes average out, and combined, they have more predictive power.

This way of putting together multiple decision trees into an ensemble is called a *random forest*. To train a random forest, we can do this (figure 6.26):

- Train N independent decision tree models.
- For each model, select a random subset of features, and use only them for training.
- When predicting, combine the output of N models into one.

NOTE This is a very simplified version of the algorithm. It's enough to illustrate the main idea, but in reality, it's more complex.

Scikit-learn contains an implementation of a random forest, so we can use it for solving our problem. Let's do it.

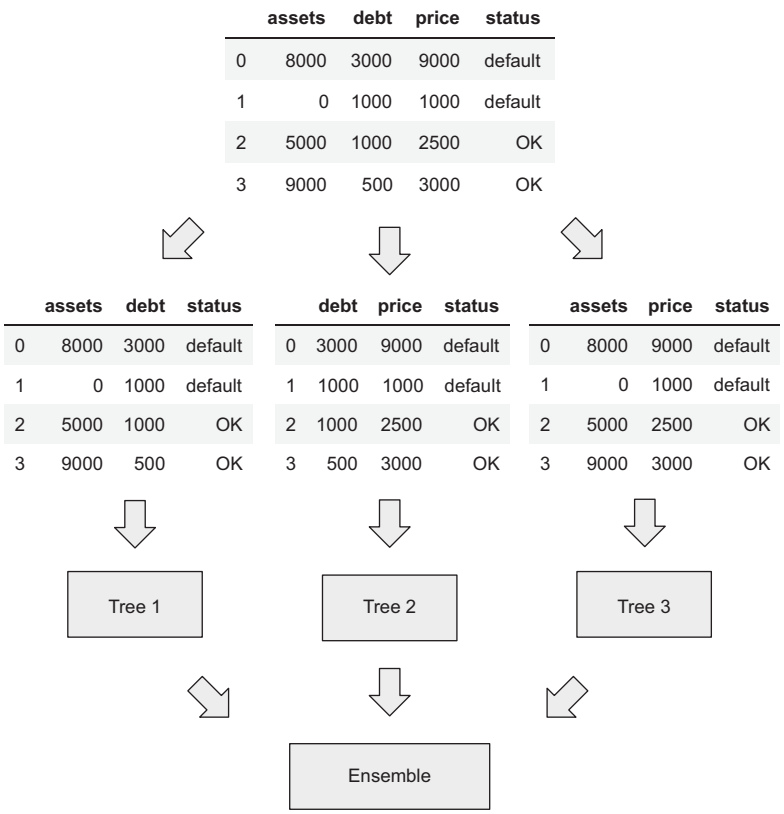


Figure 6.25 Models we want to combine in an ensemble should not be the same. We can make sure they are different by training each tree on a different subset of features.

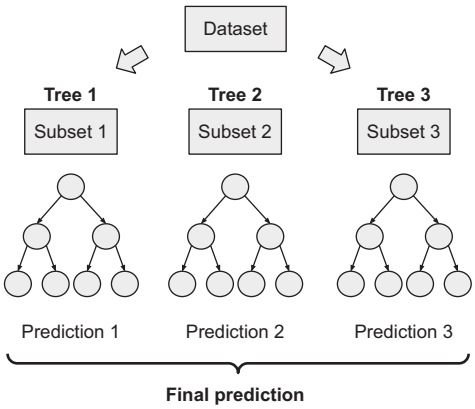


Figure 6.26 Training a random forest model: for training each tree, randomly select a subset of features. When making the final prediction, combine all the predictions into one.

6.3.1 Training a random forest

To use random forest in Scikit-learn, we need to import `RandomForestClassifier` from the `ensemble` package:

```
from sklearn.ensemble import RandomForestClassifier
```

When training a model, the first thing we need to specify is the number of trees we want to have in the ensemble. We do it with the `n_estimators` parameter:

```
rf = RandomForestClassifier(n_estimators=10)
rf.fit(X_train, y_train)
```

After training finishes, we can evaluate the performance of the result:

```
y_pred = rf.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

It shows 77.9%. However, the number you see may be different. Every time we retrain the model, the score changes: it varies from 77% to 80%.

The reason for this is randomization: to train a tree, we randomly select a subset of features. To make the results consistent, we need to fix the seed for the random-number generator by assigning some value to the `random_state` parameter:

```
rf = RandomForestClassifier(n_estimators=10, random_state=3)
rf.fit(X_train, y_train)
```

Now we can evaluate it:

```
y_pred = rf.predict_proba(X_val)[:, 1]
roc_auc_score(y_val, y_pred)
```

This time, we get an AUC of 78%. This score doesn't change, no matter how many times we retrain the model.

The number of trees in the ensemble is an important parameter, and it influences the performance of the model. Usually, a model with more trees is better than a model with fewer trees. On the other hand, adding too many trees is not always helpful.

To see how many trees we need, we can iterate over different values for `n_estimators` and see its effect on AUC:

```
aucs = []

for i in range(10, 201, 10):
    rf = RandomForestClassifier(n_estimators=i, random_state=3)
    rf.fit(X_train, y_train)

    y_pred = rf.predict_proba(X_val)[:, 1]
    auc = roc_auc_score(y_val, y_pred)
    print('%s -> %.3f' % (i, auc))

    aucs.append(auc)
```

Creates a list with AUC results

Trains progressively more trees in each iteration

Evaluates the score

Adds the score to the list with other scores

In this code, we try different numbers of trees: from 10 to 200, going by steps of 10 (10, 20, 30, …). Each time we train a model, we calculate its AUC on the validation set and record it.

After we finish, we can plot the results:

```
plt.plot(range(10, 201, 10), aucs)
```

In figure 6.27, we can see the results.

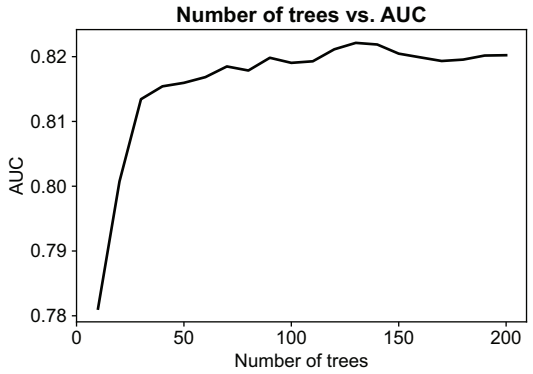


Figure 6.27 The performance of the random forest model with different values for the `n_estimators` parameter

The performance grows rapidly for the first 25–30 trees; then the growth slows down. After 130, adding more trees is not helpful anymore: the performance stays approximately at the level of 82%.

The number of trees is not the only parameter we can change to get better performance. Next, we see which other parameters we should also tune to improve the model.

6.3.2 Parameter tuning for random forest

A random forest ensemble consists of multiple decision trees, so the most important parameters we need to tune for random forest are the same:

- `max_depth`
- `min_leaf_size`

We can change other parameters, but we won't cover them in detail in this chapter. Refer to the official documentation for more information (https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html).

Let's start with `max_depth`. We already know that this parameter significantly affects the performance of a decision tree. This is also the case for random forest: larger trees tend to overfit more than smaller trees.

Let's test a few values for `max_depth` and see how AUC evolves as the number of trees grows:

```
all_aucs = {}

for depth in [5, 10, 20]:
    print('depth: %s' % depth)
    aucs = []

    for i in range(10, 201, 10):
        rf = RandomForestClassifier(n_estimators=i, max_depth=depth,
     random_state=1)
        rf.fit(X_train, y_train)
        y_pred = rf.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)
        print('%s -> %.3f' % (i, auc))
        aucs.append(auc)

    all_aucs[depth] = aucs
    print()
```

Creates a dictionary with AUC results

Iterates over different depth values

Creates a list with AUC results for the current depth level

Iterates over different n_estimator values

Evaluates the model

Save the AUCs for the current depth level in the dictionary

Now for each value of `max_depth`, we have a series of AUC scores. We can plot them now:

```
num_trees = list(range(10, 201, 10))
plt.plot(num_trees, all_aucs[5], label='depth=5')
plt.plot(num_trees, all_aucs[10], label='depth=10')
plt.plot(num_trees, all_aucs[20], label='depth=20')
plt.legend()
```

In figure 6.28 we see the result.

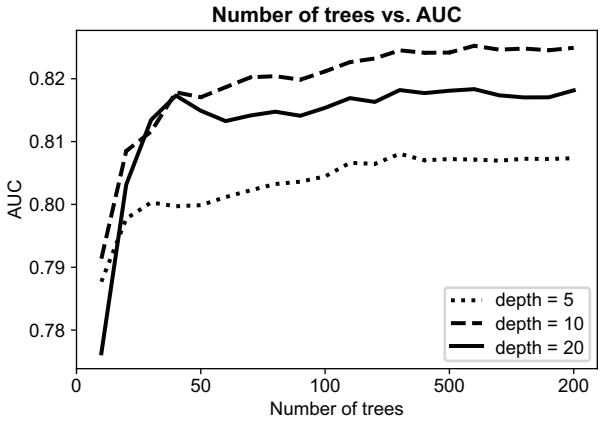


Figure 6.28 The performance of random forest with different values of the `max_depth` parameter

With `max_depth=10`, AUC goes over 82%, whereas for other values it performs worse.

Now let's tune `min_samples_leaf`. We set the value for the `max_depth` parameter from the previous step and then follow the same approach as previously for determining the best value for `min_samples_leaf`:

```
all_aucs = {}

for m in [3, 5, 10]:
    print('min_samples_leaf: %s' % m)
    aucs = []

    for i in range(10, 201, 20):
        rf = RandomForestClassifier(n_estimators=i, max_depth=10,
     min_samples_leaf=m, random_state=1)
        rf.fit(X_train, y_train)
        y_pred = rf.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)
        print('%s -> %.3f' % (i, auc))
        aucs.append(auc)

    all_aucs[m] = aucs
    print()
```

Let's plot it:

```
num_trees = list(range(10, 201, 20))
plt.plot(num_trees, all_aucs[3], label='min_samples_leaf=3')
plt.plot(num_trees, all_aucs[5], label='min_samples_leaf=5')
plt.plot(num_trees, all_aucs[10], label='min_samples_leaf=10')
plt.legend()
```

Then review the results (figure 6.29).

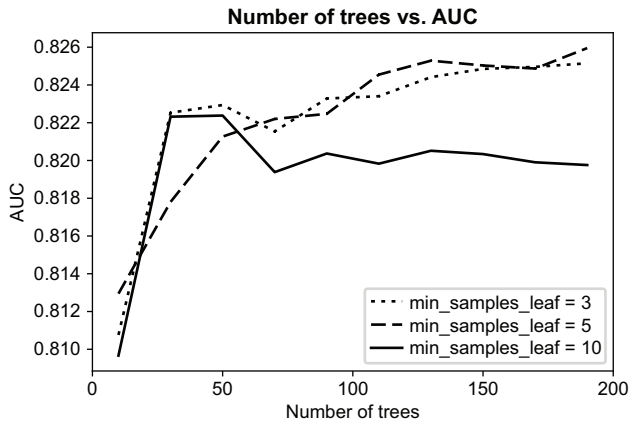


Figure 6.29 The performance of random forest with different values of `min_samples_leaf` (with `max_depth=10`)

We see that AUC is slightly better for small values of `min_samples_leaf` and the best value is 5.

Thus, the best parameters for random forest for our problem are

- `max_depth=10`
- `min_samples_leaf=5`

We achieved the best AUC with 200 trees, so we should set the `n_estimators` parameter to 200.

Let's train the final model:

```
rf = RandomForestClassifier(n_estimators=200, max_depth=10,
     min_samples_leaf=5, random_state=1)
```

Random forest is not the only way to combine multiple decision trees. There's a different approach: gradient boosting. We cover that next.

Exercise 6.2

To make an ensemble useful, trees in a random forest should be different from each other. This is done by

a. Selecting different parameters for each individual tree
b. Randomly selecting a different subset of features for each tree
c. Randomly selecting values for splitting

6.4 Gradient boosting

In a random forest, each tree is independent: it's trained on a different set of features. After individual trees are trained, we combine all their decisions together to get the final decision.

It's not the only way to combine multiple models together in one ensemble, however. Alternatively, we can train models sequentially — each next model tries to fix errors from the previous one:

- Train the first model.
- Look at the errors it makes.
- Train another model that fixes these errors.
- Look at the errors again; repeat sequentially.

This way of combining models is called *boosting*. *Gradient boosting* is a particular variation of this approach that works especially well with trees (figure 6.30).

Let's have a look at how we can use it for solving our problem.

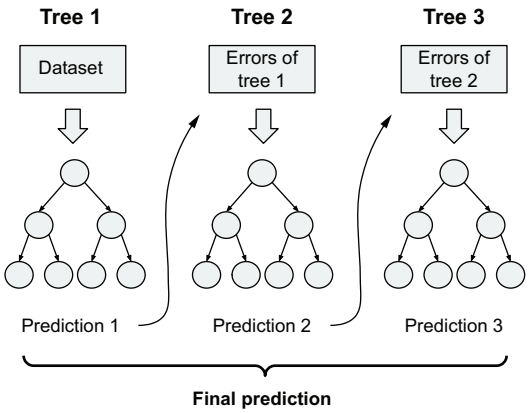


Figure 6.30 In gradient boosting, we train the models sequentially, and each next tree fixes the errors of the previous one.

6.4.1 XGBoost: Extreme gradient boosting

We have many good implementations of the gradient boosting model: `GradientBoostingClassifier` from Scikit-learn, XGBoost, LightGBM and CatBoost. In this chapter, we use XGBoost (short for "Extreme Gradient Boosting"), which is the most popular implementation.

XGBoost doesn't come with Anaconda, so to use it, we need to install it. The easiest way is to install it with `pip`:

```
pip install xgboost
```

Next, open the notebook with our project and import it:

```
import xgboost as xgb
```

NOTE In some cases, importing XGBoost may give you a warning like `YMLLoadWarning`. You shouldn't worry about it; the library will work without problems.

Using the alias `xgb` when importing XGBoost is a convention, just like with other popular machine learning packages in Python.

Before we can train an XGBoost model, we need to wrap our data into `DMatrix` — a special data structure for finding splits efficiently. Let's do it:

```
dtrain = xgb.DMatrix(X_train, label=y_train, feature_names=dv.feature_names_)
```

When creating an instance of `DMatrix`, we pass three parameters:

- `X_train`: the feature matrix
- `y_train`: the target variable
- `feature_names`: the names of features in `X_train`

Let's do the same for the validation dataset:

```
dval = xgb.DMatrix(X_val, label=y_val, feature_names=dv.feature_names_)
```

The next step is specifying the parameters for training. We're using only a small subset of the default parameters of XGBoost (check the official documentation for the entire list of parameter: https://xgboost.readthedocs.io/en/latest/parameter.html):

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

For us, the most important parameter now is `objective`: it specifies the learning task. We're solving a binary classification problem — that's why we need to choose `binary :logistic`. We cover the rest of these parameters later in this section.

For training an XGBoost model, we use the `train` function. Let's start with 10 trees:

```
model = xgb.train(xgb_params, dtrain, num_boost_round=10)
```

We provide three arguments to `train`:

- `xgb_params`: the parameters for training
- `dtrain`: the dataset for training (an instance of `DMatrix`)
- `num_boost_round=10`: the number of trees to train

After a few seconds, we get a model. To evaluate it, we need to make a prediction on the validation dataset. For that, use the `predict` method with the validation data wrapped in `DMatrix`:

```
y_pred = model.predict(dval)
```

The result, `y_pred`, is a one-dimensional NumPy array with predictions: the risk score for each customer in the validation dataset (figure 6.31).

```
y_pred = model.predict(dval)
y_pred[:10]
```

```
array([0.08926772, 0.0468099 , 0.09692743, 0.17261842, 0.05435968,
       0.12576081, 0.08033007, 0.61870354, 0.486538  , 0.04056795],
      dtype=float32)
```

Figure 6.31 The predictions of XGBoost

Next, we calculate AUC using the same approach as previously:

```
roc_auc_score(y_val, y_pred)
```

After executing it, we get 81.5%. This is quite a good result, but it's still slightly worse than our best random forest model (82.5%).

Training an XGBoost model is simpler when we can see how its performance changes when the number of trees grows. We see how to do it next.

6.4.2 Model performance monitoring

To get an idea of how AUC changes as the number of trees grows, we can use a watchlist — a built-in feature in XGBoost for monitoring model performance.

A watchlist is a Python list with tuples. Each tuple contains a DMatrix and its name. This is how we typically do it:

```
watchlist = [(dtrain, 'train'), (dval, 'val')]
```

Additionally, we modify the list of parameters for training: we need to specify the metric we use for evaluation. In our case, it's the AUC:

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

Sets the evaluation metric to the AUC

To use the watchlist during training, we need to specify two extra arguments for the `train` function:

- `evals`: the watchlist.
- `verbose_eval`: how often we print the metric. If we set it to "10," we see the result after each 10th step.

Let's train it:

```
model = xgb.train(xgb_params, dtrain,
                  num_boost_round=100,
                  evals=watchlist, verbose_eval=10)
```

While training, XGBoost prints the scores to the output:

```
[0]  train-auc:0.862996  val-auc:0.768179
[10] train-auc:0.950021  val-auc:0.815577
[20] train-auc:0.973165  val-auc:0.817748
```

```
[30] train-auc:0.987718  val-auc:0.817875
[40] train-auc:0.994562  val-auc:0.813873
[50] train-auc:0.996881  val-auc:0.811282
[60] train-auc:0.998887  val-auc:0.808006
[70] train-auc:0.999439  val-auc:0.807316
[80] train-auc:0.999847  val-auc:0.806771
[90] train-auc:0.999915  val-auc:0.806371
[99] train-auc:0.999975  val-auc:0.805457
```

As the number of trees grows, the score on the training set goes up (figure 6.32).

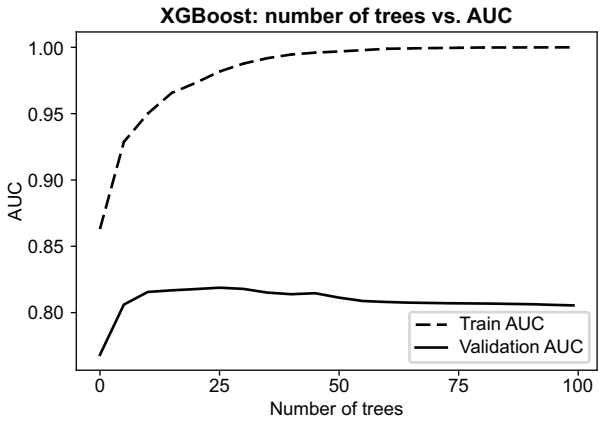


Figure 6.32 The effect of the number of trees on the AUC from train and validation sets. To see how to plot these values, check the notebook in the book's GitHub repository.

This behavior is expected: in boosting, every next model tries to fix the mistakes from the previous step, so the score is always improving.

For the validation score, however, this is not the case. It goes up initially but then starts to decrease. This is the effect of overfitting: our model becomes more and more complex until it simply memorizes the entire training set. It's not helpful for predicting the outcome for the customers outside of the training set, and the validation score reflects that.

We get the best AUC on the 30th iteration (81.7%), but it's not so different from the score we got on the 10th iteration (81.5%).

Next, we'll see how to get the best out of XGBoost by tuning its parameters.

6.4.3 Parameter tuning for XGBoost

Previously, we used a subset of default parameters for training a model:

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,
```

```
    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

We're mostly interested in the first three parameters. These parameters control the training process:

- `eta`: Learning rate. Decision trees and random forest don't have this parameter. We cover it later in this section when we tune it.
- `max_depth`: The maximum allowed depth of each tree; the same as `max_depth` in `DecisionTreeClassifier` from Scikit-learn.
- `min_child_weight`: The minimal number of observations in each group; the same as `min_leaf_size` in `DecisionTreeClassifier` from Scikit-learn.

Other parameters:

- `objective`: The type of task we want to solve. For classification, it should be `binary:logistic`.
- `eval_metric`: The metric we use for evaluation. For this project, it's "AUC."
- `nthread`: The number of threads we use for training the model. XGBoost is very good at parallelizing training, so set it to the number of cores your computer has.
- `seed`: The seed for the random-number generator; we need to set it to make sure the results are reproducible.
- `silent`: The verbosity of the output. When we set it to "1," it outputs only warnings.

This is not the full list of parameters, only the basic ones. You can learn more about all the parameters in the official documentation (https://xgboost.readthedocs.io/en/latest/parameter.html).

We already know `max_depth` and `min_child_weight` (`min_leaf_size`), but we haven't previously come across `eta` — the learning rate parameter. Let's talk about it and see how we can optimize it.

Learning rate

In boosting, each tree tries to correct the mistakes from the previous iterations. Learning rate determines the weight of this correction. If we have a large value for `eta`, the correction overweights the previous predictions significantly. On the other hand, if the value is small, only a small fraction of this correction is used.

In practice it means

- If `eta` is too large, the model starts to overfit quite early without realizing its full potential.
- If it's too small, we need to train too many trees before it can produce good results.

The default value of 0.3 is reasonably good for large datasets, but for smaller datasets like ours, we should try smaller values like 0.1 or even 0.05.

Let's do it and see if it helps to improve the performance:

```
xgb_params = {
    'eta': 0.1,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'eval_metric': 'auc',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
```

Changes eta from 0.3 to 0.1

Because now we can use a watchlist to monitor the performance of our model, we can train for as many iterations as we want. Previously we used 100 iterations, but this may be not enough for smaller `eta`. So let's use 500 rounds for training:

```
model = xgb.train(xgb_params, dtrain,
                  num_boost_round=500, verbose_eval=10,
                  evals=watchlist)
```

When running it, we see that the best validation score is 82.4%:

```
[60] train-auc:0.976407  val-auc:0.824456
```

Previously, we could achieve AUC of 81.7% when `eta` was set to the default value of 0.3. Let's compare these two models (figure 6.33).

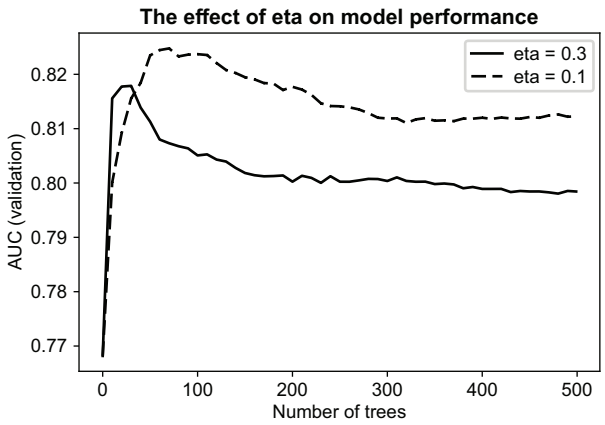


Figure 6.33 The effect of the `eta` parameter on the validation score

When `eta` is 0.3, we get the best AUC pretty quickly, but then it starts to overfit. After the 30th iteration, the performance on the validation set goes down.