Build a portfolio of real-life projects

Machine Learning Bookcamp

Alexey Grigorev

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Machine Learning Bookcamp
Build a portfolio of real-life projects
Version 1

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Machine Learning Bookcamp*.

This book will teach you machine learning in a project-based way, touching on a broad range of topics from the basics to the latest deep learning techniques. In addition, it covers often overlooked topics, from productionizing machine learning models to collecting datasets. It's focused on the practical, so it's ideal for software engineers looking to deep dive into machine learning.

By the end of the book, you will have implemented a wide variety of projects that will serve as a great portfolio. The knowledge from the book together with the portfolio will help you launch a career in machine learning, as a data scientist or a machine learning engineer.

The main language of the book is Python, and we will use the standard PyData stack: NumPy, SciPy, Pandas, and Scikit-Learn. In addition, we will learn how to use other libraries, like Keras with TensorFlow for deep learning. Finally, we will cover infrastructure and deployment technologies like Flask, Docker, and AWS.

Three chapters available now, and after reading them, you will understand the problems that machine learning can solve and will have finished two projects: predicting the price of a car and determining whether a customer is going to churn.

As additional chapters become available, you will also learn how to evaluate machine learning models, how to make them available as a web service, and much more!

I hope you find this book useful, and I invite you to share your comments, questions, and suggestions in Manning's Author Forum for my book.

-- Alexey Grigorev

brief contents

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 deep dive into one of the steps of CRISP-DM, which is 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 description of cars and their prices, for example. Then we provide a machine learning model with this dataset and "teach" it by showing cars and their prices. This process is called *training* or sometimes *fitting* (figure 1.1).

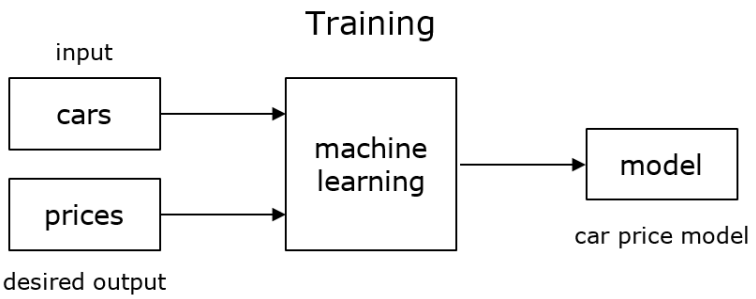


Figure 1.1 A machine learning algorithm takes in input data (descriptions of cars) and desired output (the cars' prices). Based on that data, it produces a model.

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).

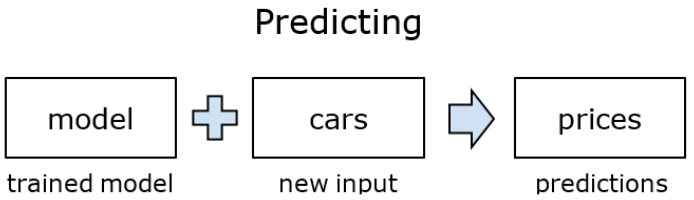


Figure 1.2 When training is done, we have a model that can be applied to new input data (cars without prices) to produce the output (predictions of prices).

As we see, 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 there 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 with a programming language such as Java or Python, and the result is called s*oftware*. So, in contrast with machine learning, a human does all the difficult work in traditional programming (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.

We can summarize the difference between a traditional software system and a system based on machine learning with figure 1.4, below. 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). By 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).

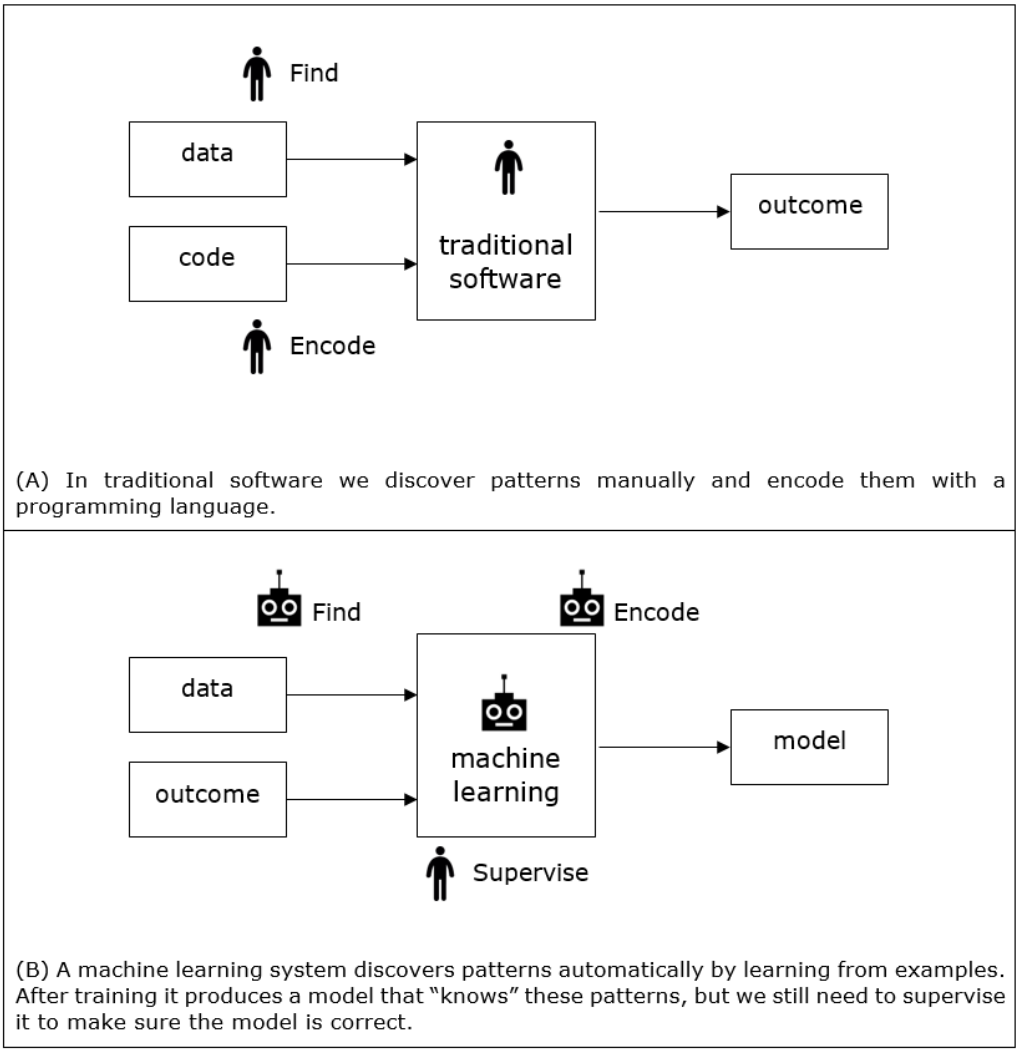


Figure 1.4 The difference between a traditional software system and a machine learning system. In traditional software engineering, we do all the work, whereas in machine learning, we delegate pattern discovery to a machine.

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 will talk about a spam-detection system to show this difference.

Suppose that 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 these emails 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 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 *Nigerian*. 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 "nigerian" 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 *Nigerian* with good intention, 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 "nigerian" 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.

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 will be marked as spam and good messages won't be.

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 "nigerian"? true/false

In this particular case, we describe all emails with a set of six features. Coincidently, 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.

Now imagine that we have a mail that users marked as spam (figure 1.5).

```
Subject: Waiting for your reply
From: promotions@test.com
The Nigerian prince text

Spam: true
```

Figure 1.5 A mail that a user marked as spam

We can express this mail 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 when for false (figure 1.6). Because our users marked the message as spam, the target variable is 1 (true).

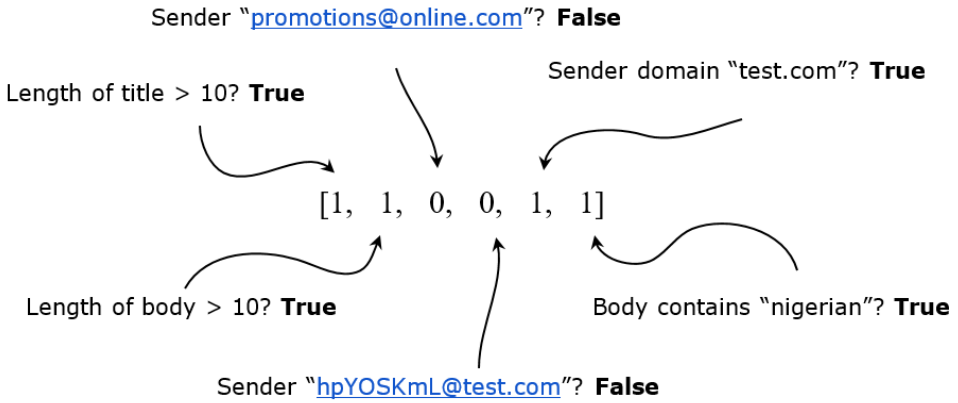


Figure 1.6 The six-dimensional feature vector for a spam email. Each of the six features is represented by a number. In this case, we use 1 if the feature is true and 0 if the feature is false.

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).

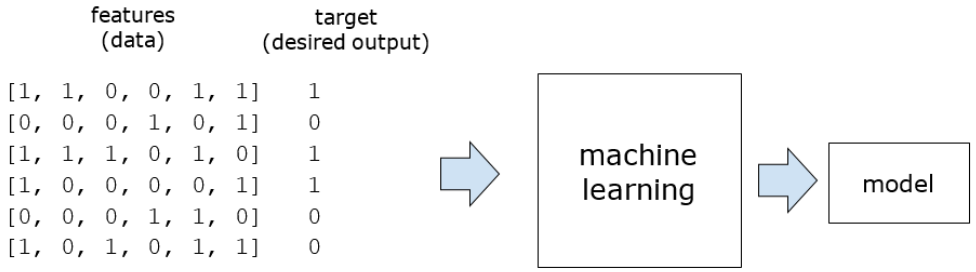


Figure 1.7 The input to a machine learning algorithm consists of multiple feature vectors and the target variable for each vector.

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 vandalic 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. Literally anything that we can express as (input data, desired output), we can use 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, no machine learning is 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 (typically, a 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).

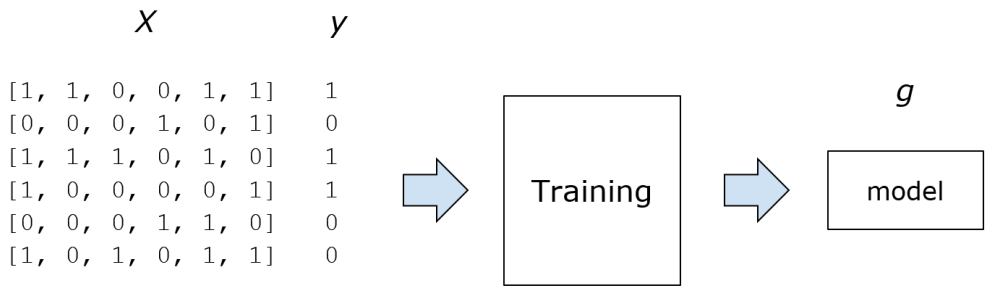


Figure 1.8 When we train a model, an algorithm takes in a matrix X in which feature vectors are rows and the desired output is the vector y, with all the values we want to predict. The result of training is g, the model. After training, g should produce y when applied to X — or, in short, $g(X) \approx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

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 will 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, and we will cover it in multiple chapters throughout the book, starting with chapter 3, in which we predict whether a customer is going to churn.
- *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 identifying model and make from a picture of a car.

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.

There are other steps, starting from identifying the problem that machine learning can solve and finish by 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

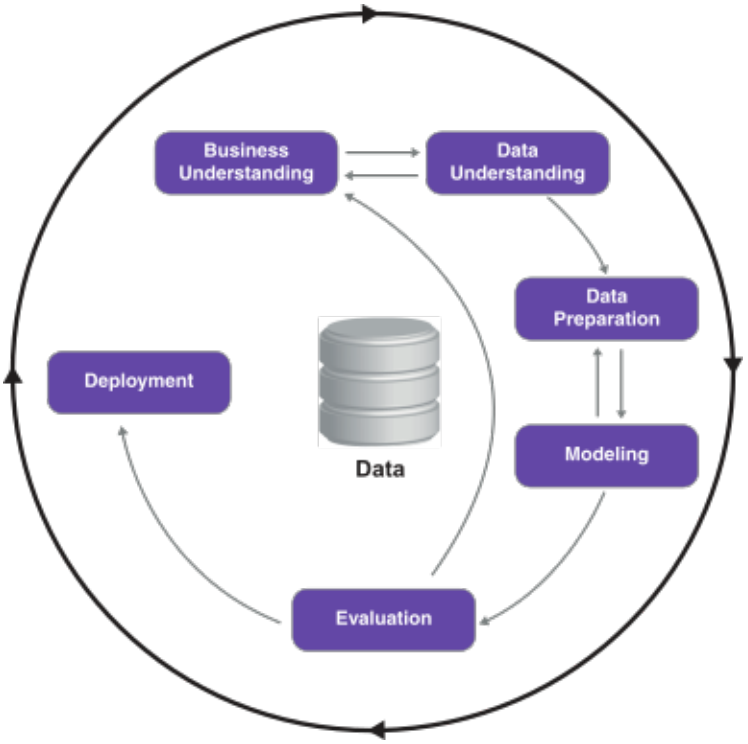


Figure 1.9 The CRISP-DM process. A machine learning project starts with understanding the problem and then moves into data preparation, training the model, and evaluating the results. Finally, the model goes to deployment. This process is iterative, and at each step, it's possible to go back to the previous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

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.
- In the data preparation step, we transform the data into 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 step

Let's consider the spam-detection example for an email service provider. We see more spam messages than 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 step

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 the data generated by the 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 be not 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, collecting 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 step

In this step, we clean the data, transforming it in such a way that it can be used as input to 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 step

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 will 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 step

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 spam-detection case, 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 step

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 in 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 very common misconception that machine learning engineers and data scientists spent their entire day training machine learning models. In reality, this idea is incorrect, as we can see in the CRISP-DM diagram. 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 mails, 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 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 decided 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 very 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 and take a part out of it, keep the part away and train the model on the rest 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* (figure 1.10).

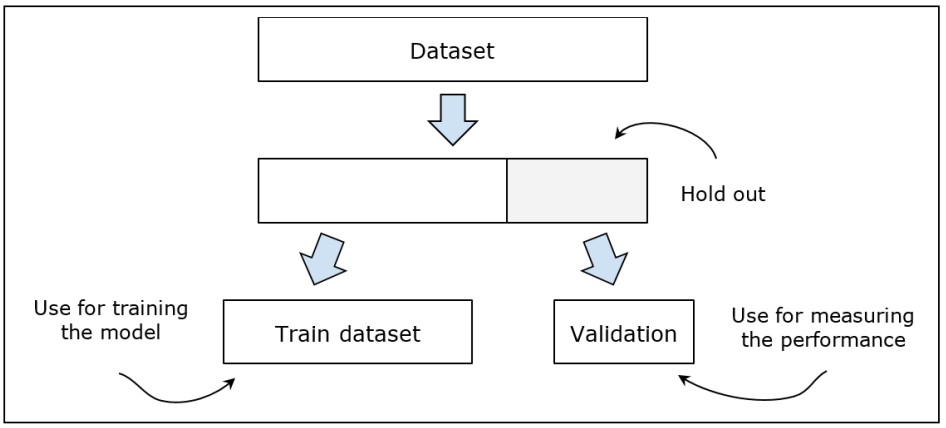


Figure 1.10 To evaluate the performance of a model, we set some data aside and use it only for validation purposes.

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 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 in 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).

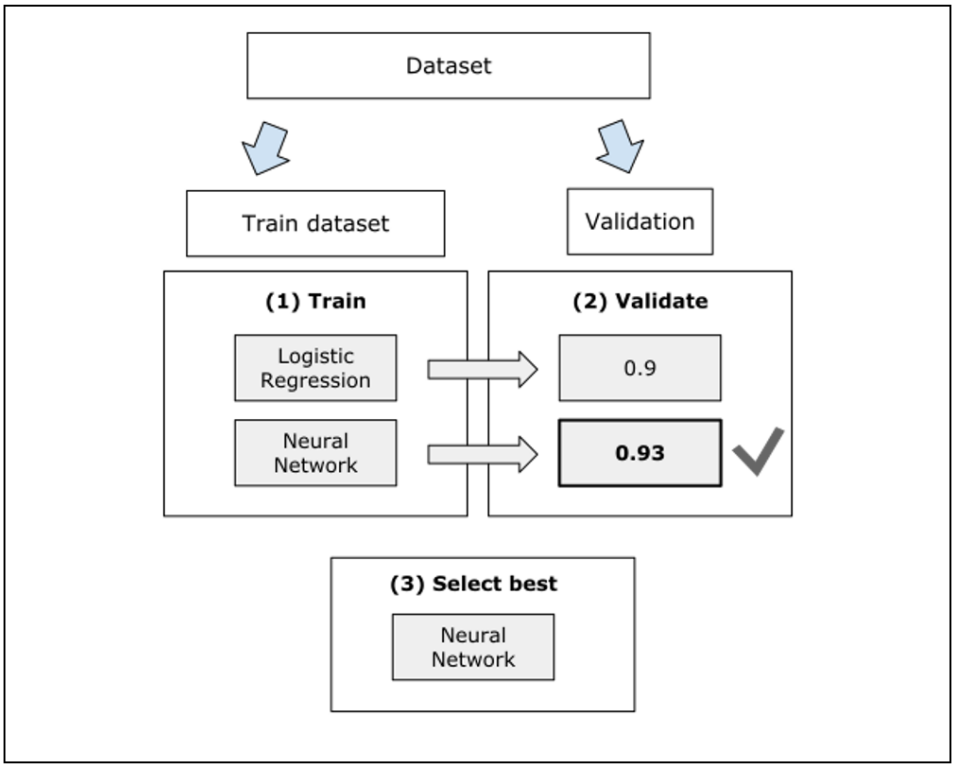


Figure 1.11 The validation process. We split the dataset into two parts, train models on the training part, and evaluate performance on the validation part. Using the evaluation results, we can choose the best model.

Often, we don't have 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. What's more, there are 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).

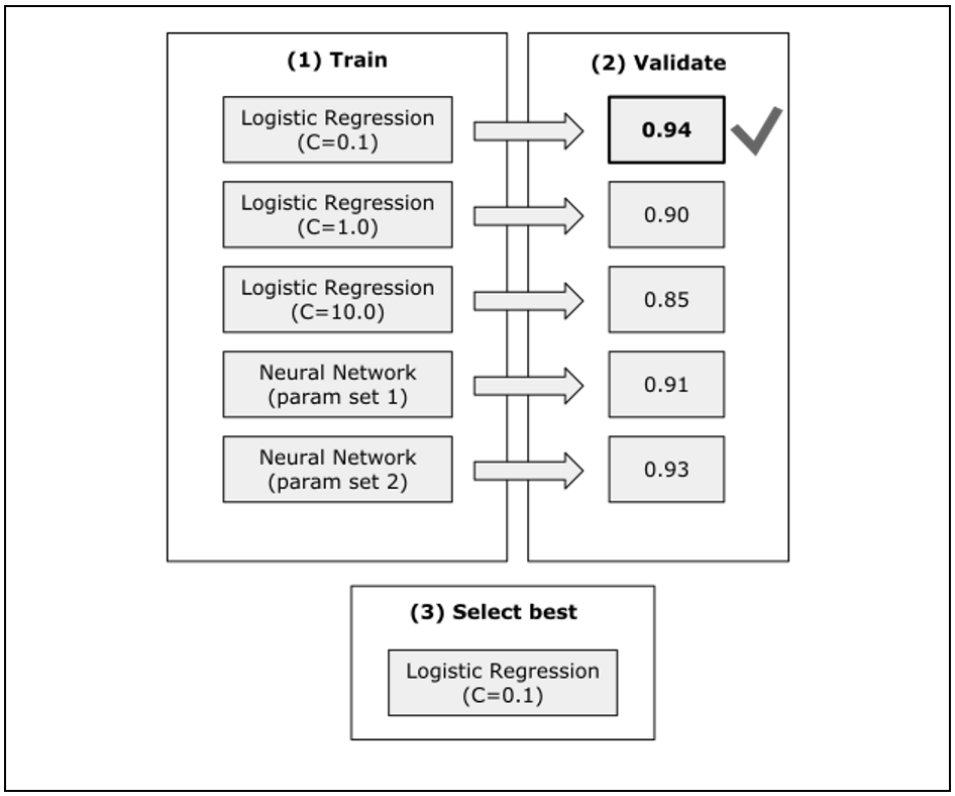


Figure 1.12 Using the validation dataset to select the best model with the best parameters

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 there just by chance. In other words, the "best" model may simply get lucky in predicting the outcomes for this particular dataset.

> **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 and use it rarely, only to test the model that we selected as the best (figure 1.13).

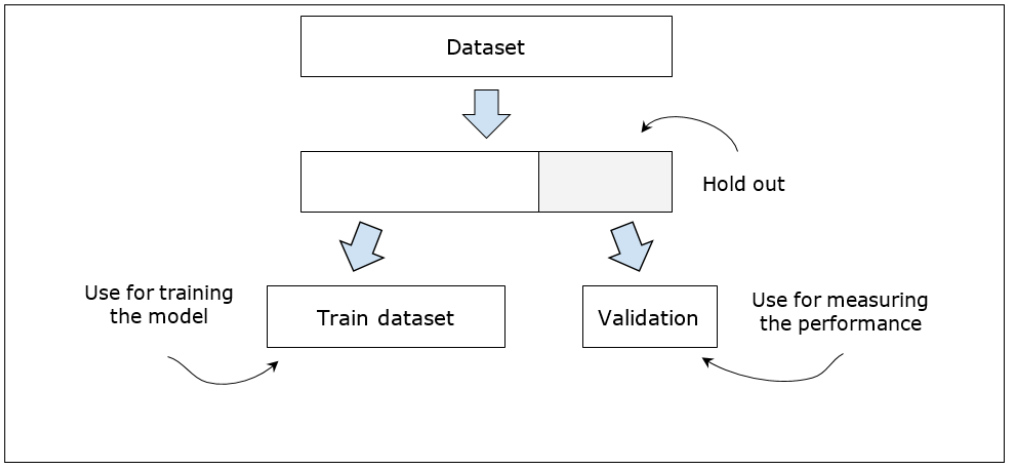


Figure 1.13 Splitting the data into training, testing, and validation parts.

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).

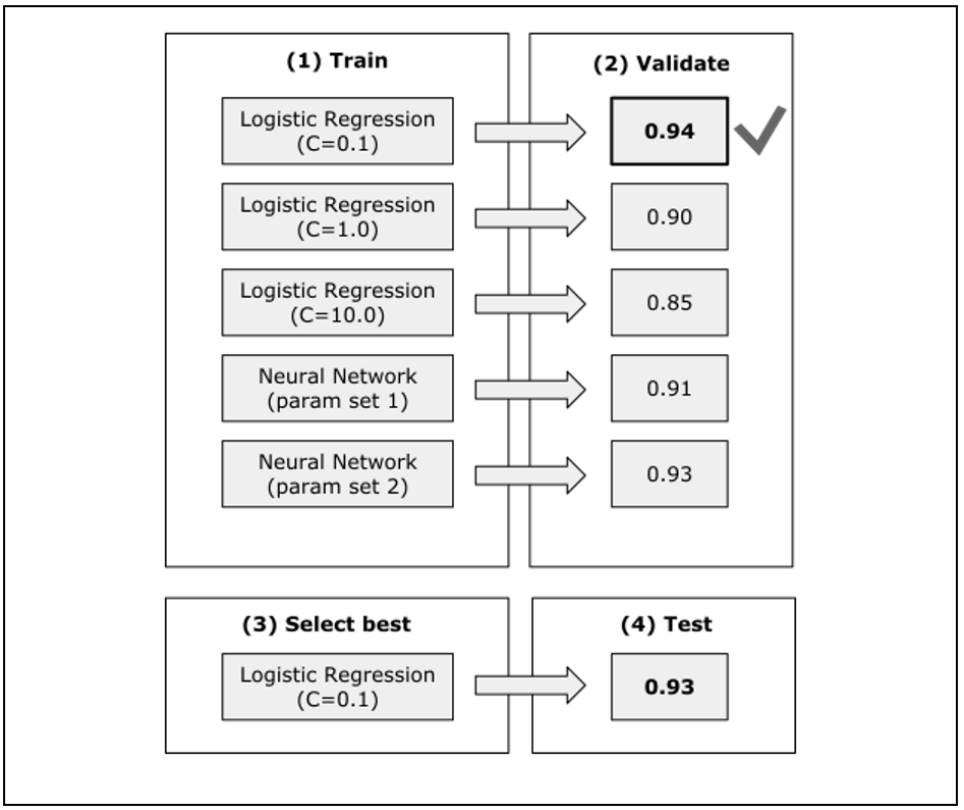


Figure 1.14 We use the test dataset to confirm that the performance of the best model on the validation set is good.

> **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.

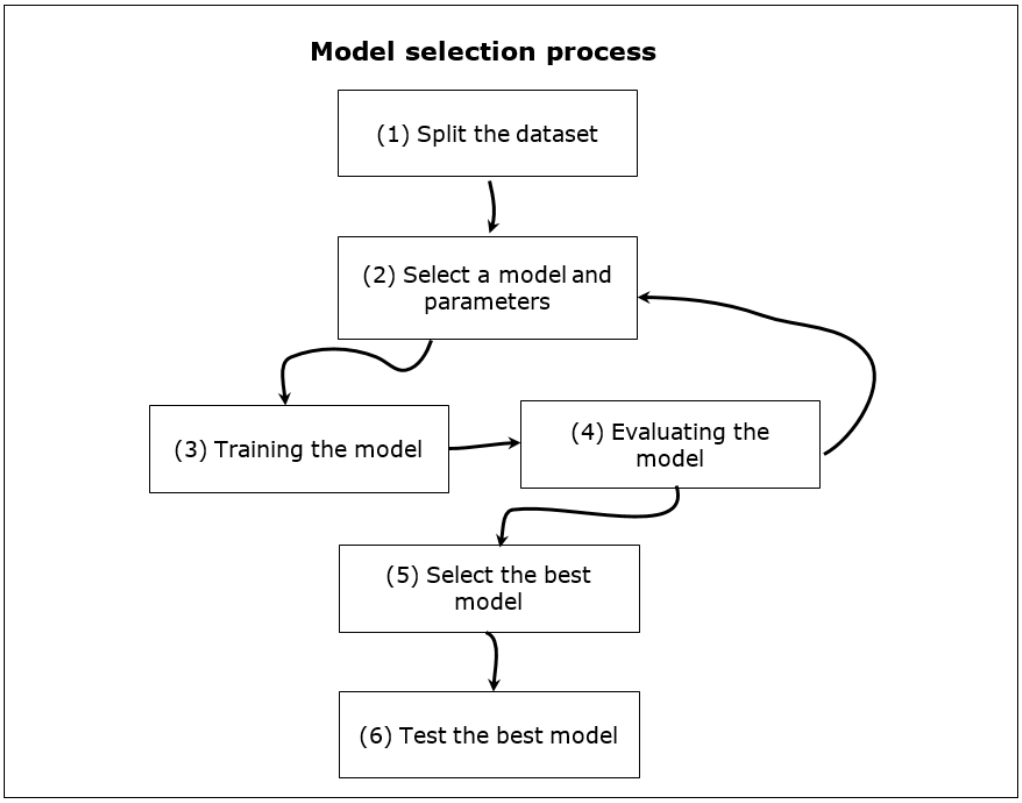


Figure 1.15. The model selection process. First, we split the dataset, select a model, and train it only on the training part of the data. Then we evaluate the model on the validation part. We repeat the process many times until we find the best model.

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 on real users.

1.4 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 the process down 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 will create a regression model, and we will start with the simplest one: linear regression. We will 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 will 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 will 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, the sellers often struggle to come up with a meaningful price. We want to help our users with automatic price recommendation. We ask the sellers to specify 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 contains all the important features as well as the recommended price — exactly what we need 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 to 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. There are multiple options for doing this. You can download it manually through the kaggle web interface. It's available at https://www.kaggle.com/CooperUnion/cardataset.[1] Go there, open it, and click the download link. The other option is using the kaggle command-line interface (CLI), which is

[1] You can read more about the dataset and the way it was collected at https://www.kaggle.com/jshih7/car-price-prediction.

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/ml-projects 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 do 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 to execute a piece of code and immediately see the outcome. In addition to that 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 only use it for Python.
- Matplotlib is a library for plotting. It is very powerful and allows you to create different types of visualizations, such as line chars, bar charts, histograms and many more.
- 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).

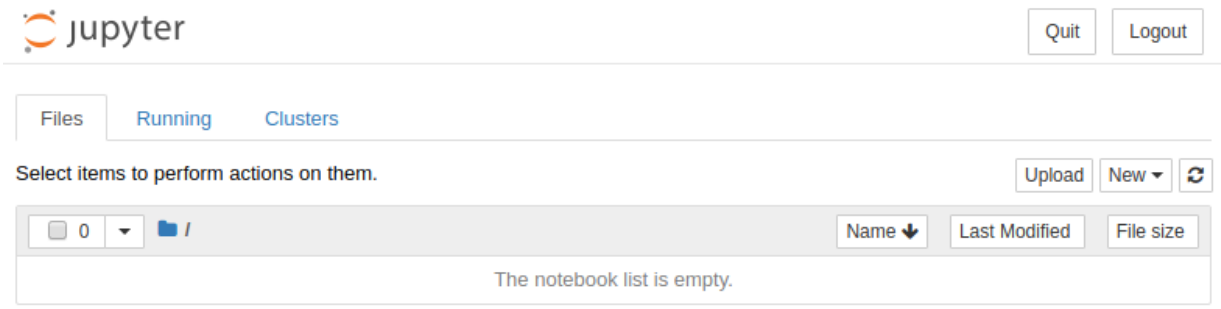


Figure 2.1 The starting screen of the jupyter notebook service

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:

```
import pandas as pd #A
import numpy as np #B

from matplotlib import pyplot as plt #C
import seaborn as sns #C
%matplotlib inline #D
```

#A Import NumPy: a library for numerical operations
#B Import Pandas: a library for tabular data
#C Import plotting libraries: matplotlib and seaborn
#D Make sure that plots are rendered correctly in jupyter notebooks.

The first two lines, A and B, 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 very common in the Python machine learning community, and everybody follows it.

The next two lines, C, 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 D 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[2] 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 are there. Let's use the `len` function:

[2] https://jupyter.org

```
len(df)
```

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

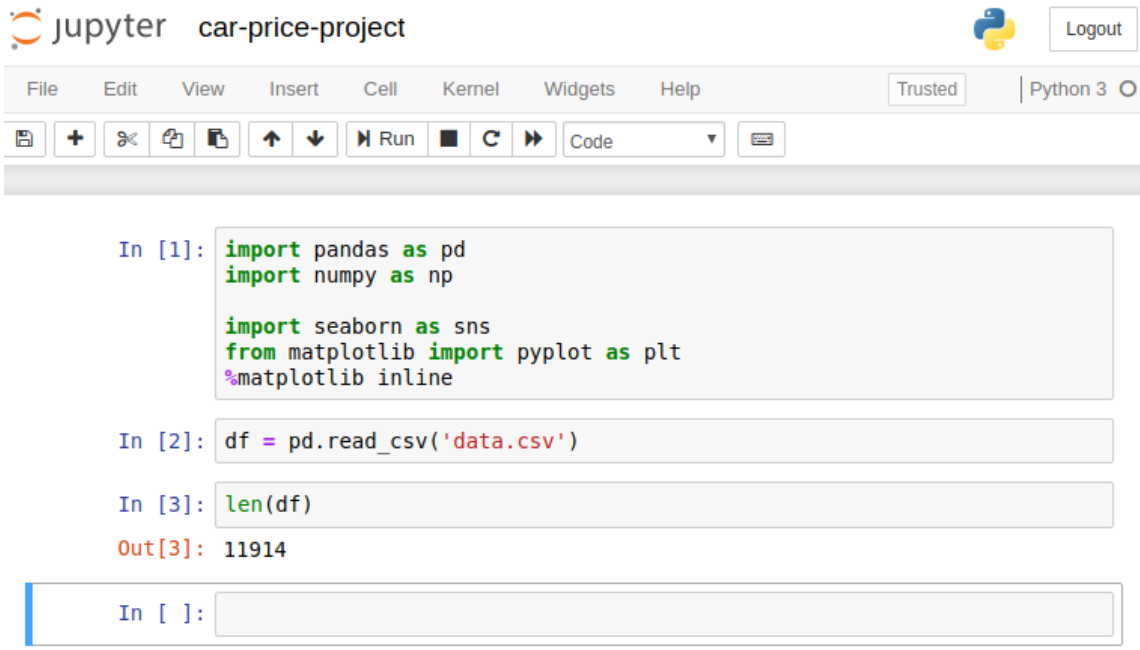


Figure 2.2. Jupyter notebooks are interactive. We can type some code in a cell, execute it, and see the results immediately, which is ideal for exploratory data analysis.

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	Lu
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: replace all spaces with underscores and lowercase all letters:

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

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

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

#A Lowercase all the column names, and replace spaces with underscores.
#B Select only columns with string values.
#C Lowercase and replace spaces with underscores for values in all string columns of the dataframe.

In A and C, 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 B.

> **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 [8]: df.head()
```

Out[8]:

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

Figure 2.4: The result of preprocessing the data. The column names and values are normalized: they are lowercased, 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). This column is our target variable — the y, which is the value that we want to learn to predict. Let's have a closer look at it.

2.2.3 Target variable analysis

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.distplot(df.msrp, kde=False)
```

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).

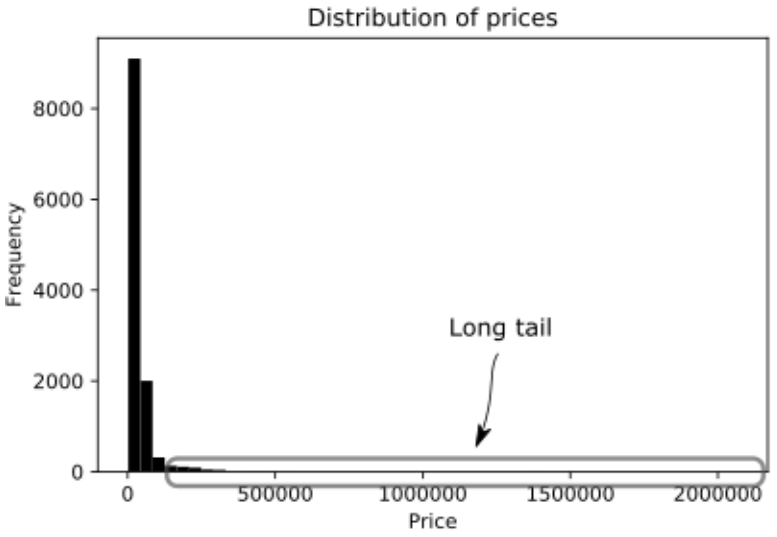


Figure 2.5 The distribution of the prices in the dataset. We see many values at the low end of the price axis and almost nothing at the high end. This is a long tail distribution, which is a typical situation for prices. There are many items with low prices and very few expensive ones.

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

```
sns.distplot(df.msrp[df.msrp < 100000], kde=False)
```

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).

$$y\_new = log(y + 1)$$

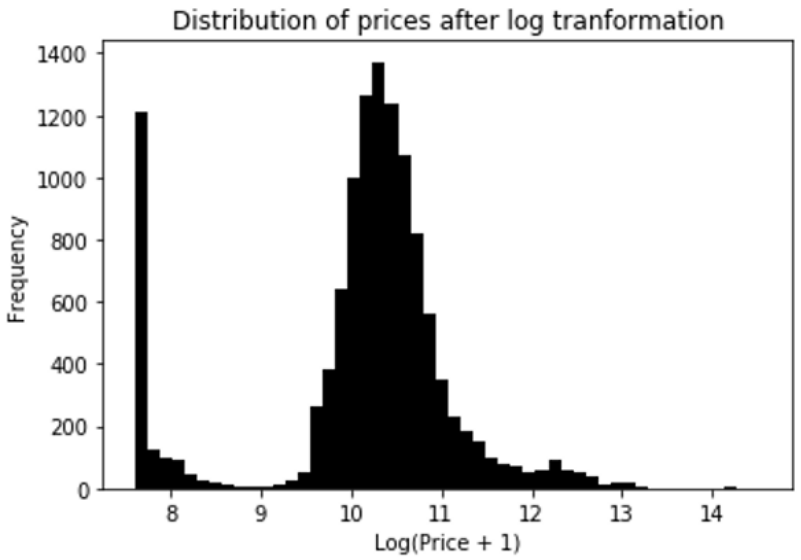


Figure 2.7 The logarithm of the price. The effect of the long tail is removed, and we can see the entire distribution in one plot.

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 distplot function (figure 2.7):

```
sns.distplot(log_price, kde=False)
```

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.

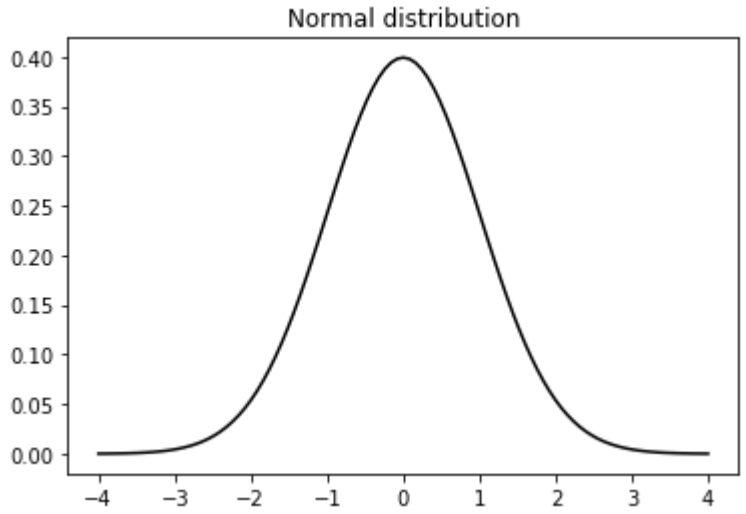


Figure 2.8: The normal distribution, also known as Gaussian, follows the bell-shaped curve, which is symmetric and has a peak in the center.

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 is 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 will need to deal with missing values later, when we train the model, so we should keep this problem in mind. For now, we won't do anything else with these features and will 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.

It's 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. So when we apply the model to this data, it emulates the case of applying the model to new data that we never saw. 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).

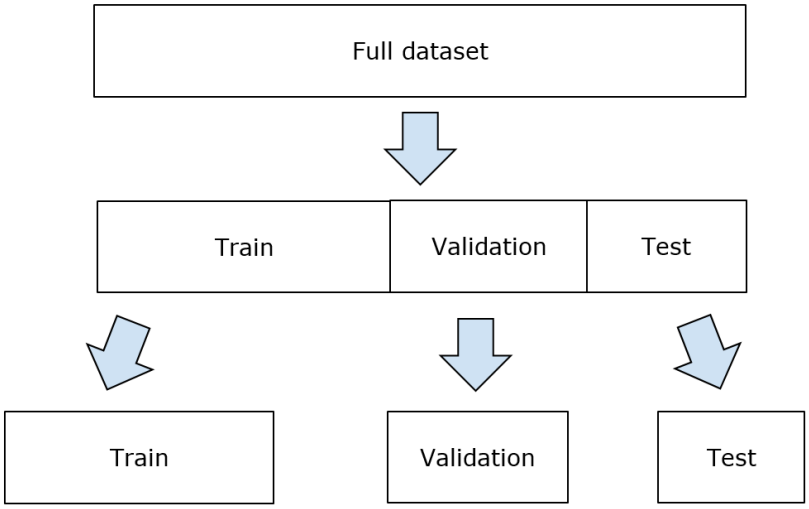


Figure 2.9 The entire dataset is split into three parts: train, validation and test.

Let's split the dataframe such that:

- 20% of data goes to validation,
- 20% goes to test, and
- the remaining 60% goes to train.

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

```
n = len(df) #A

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

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

df_shuffled = df.iloc[idx] #E

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

#A Get the number of rows in the dataframe.
#B Calculate how many rows should go to train, validation, and test.
#C Fix the random seed to make sure that the results are reproducible.
#D Create a NumPy array with indices from 0 to (n-1) and shuffle it.

#E Use the array with indices to get a shuffled dataframe.
#F Split the shuffled dataframe into train, validation, and test.

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

In D, 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 C 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 use this number as the starting seed for all the generated data inside NumPy's random package.

When we execute this code, it prints the following:

```
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 previously:

```
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.

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 E, 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]`

	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

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

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 F 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']
```

When the validation split is done, we can do 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 formula above 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:

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

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:

$$y_i \approx g(x_i) = g(x_{i1}, x_{i2}, \ldots, x_{in})$$

For our case, we have 7,150 cars in the training dataset. This means that $m = 7150$, 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 3 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 above, `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, so 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:

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

The variables $w_0, w_1, w_2,\ldots,w_n$ are the parameters of the model:

- w_0 is the *bias* term.
- $w_1, w_2,\ldots,w_n$ 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.

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

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

These weights are what the model learns when we train it. We can have a model with the following weights, for example (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, this is how it will look:

```
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
```

As we did with `xi`, which is a list with multiple elements, we put all the feature weights inside a single list `w`. Then all we need to do is loop over these weights and multiply them by the feature values — the direct translation of the formula above to Python.

This is easy to see. First, 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

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

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 this prediction is the logarithm of the price, so to get the actual prediction, we need to take the exponent of it, which is approximately $220,000.

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 mpg 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).

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

bias	horsepower	mpg	popularity
	4.53	0.44	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 that x_i is a vector with n components:

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

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

$$w = (w_1, w_2, \ldots, w_n)$$

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 like for our example:

$$x_i = (453, 11, 86)$$

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

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:

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

The dot product is a way of multiplying two vectors. We multiply corresponding elements of the vectors and then sum the results.

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_1:

$$w = (w_0, w_1, w_2, \ldots, w_n)$$

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 old 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 4 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}:

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

Or, in code:

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

So 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, $x_1, \ldots, x_i, \ldots, x_m$, and each of these vectors consists of $n+1$ features:

$$x_1 = (1, x_{11}, x_{12},\ldots,x_{1n})$$

$$x_2 = (1, x_{21}, x_{22},\ldots,x_{2n})$$

$$\ldots$$

$$x_i = (1, x_{i1}, x_{i2},\ldots,x_{in})$$

$$\ldots$$

$$x_m = (1, x_{m1}, x_{m2},\ldots,x_{mn})$$

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

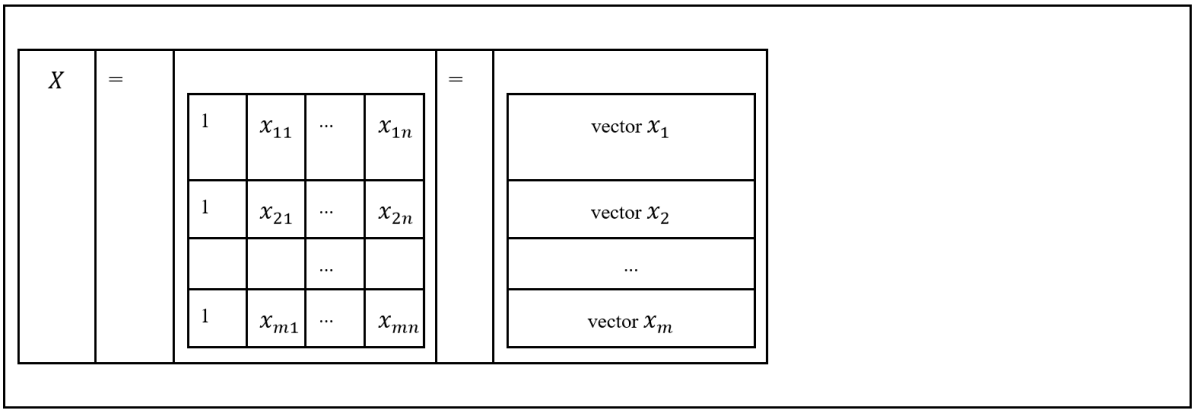


Figure 2.12 Matrix X, in which observations x_1, x_2, ..., x_m are rows.

Let's see how it looks in code. We can take a few rows for 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:

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

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

$$g(X) = Xw$$

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.

There are 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.

This piece of math may seem to be 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 NumPy and linear algebra.

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 linear_regression(X, y):
    # adding the dummy column
    ones = np.ones(X.shape[0]) # A
    X = np.column_stack([ones, X]) # B

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

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

#A Create an array that contains only 1s.
#B Add the array of 1s as the first column of X.
#C Compute X^TX .
#D Compute the inverse of X^TX .
#E Compute the rest of the normal equation.
#F Split 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 A, we create a vector containing only ones, which we append to the matrix X as the first column; this is the dummy feature in B. Next, we compute X^TX in C and its inverse in D, and we put them together to calculate w in E. Finally, we split the weights into the bias w_0 and the remaining weights w in F)

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).

```
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.

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

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

This is still very easy to translate to NumPy:

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

Let's use it for our project!

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, selecting a few numerical features and forming the matrix X from them. In the example previously, we used only three features. This time, let's 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 zero. 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. Recall the formula for linear regression:

$$g(x_i) = w_0 + x_{i1}w_1 + x_{i2}w_2 + x_{i3}w_3 + \ldots + x_{in}w_n$$

If feature 3 is missing, and we fill it with zero, x_{i3} becomes zero. 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.

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, to be more exact. Either way, it's something we can use as input to our `linear_regresson` function. Let's call it:

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

We have just trained the first model! Let's apply it to the training data to see how well it predicts:

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

Now we can plot these predicted values and compare them with the actual prices:

```
sns.distplot(y_pred, label='pred')
sns.distplot(y_train, label='y')
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.

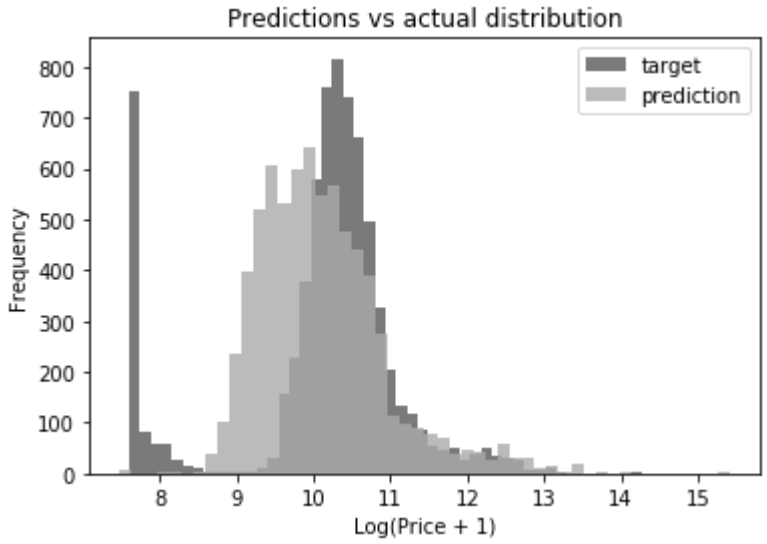


Figure 2.14 The distribution of the predicted values (light gray) and the actual values (dark gray). We see that our predictions aren't quite good; they are very different from the actual distribution.

2.4.2 RMSE: evaluating model quality

Looking at such plots 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 are the errors 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 $g(x_i)$ — the prediction we make for the observation x_i — and the actual target value y_i for that observation, one for each i (figure 2.15).

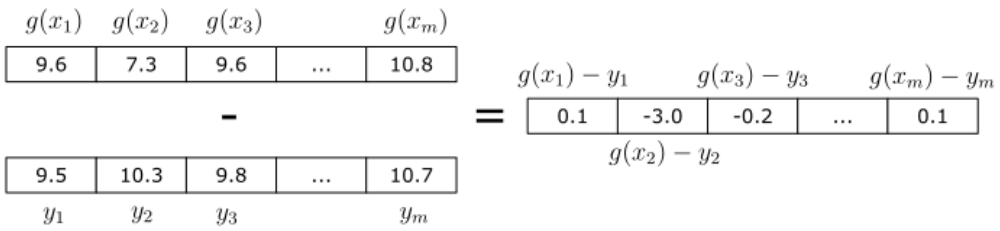


Figure 2.15 The difference between the predictions $g(x_i)$ and the actual values y_i.

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 square of the difference is 9 (figure 2.16).

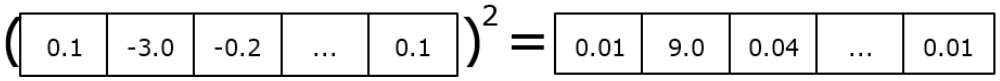


Figure 2.16 The square of the difference between the predictions and the actual values. For large differences, the square is quite big.

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

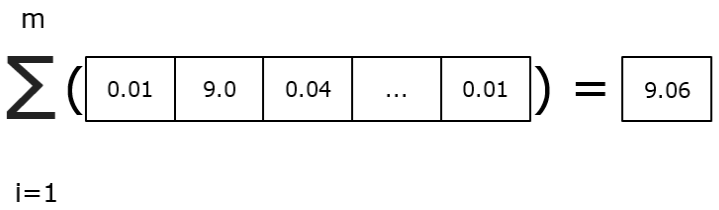


Figure 2.17 The result of the summation of all the square differences is a single number.

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

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

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

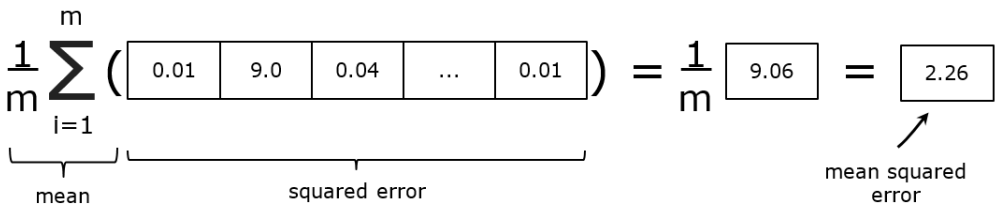


Figure 2.18 MSE is computed by calculating the mean of the squared errors.

Finally, we take the square root of that:

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

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

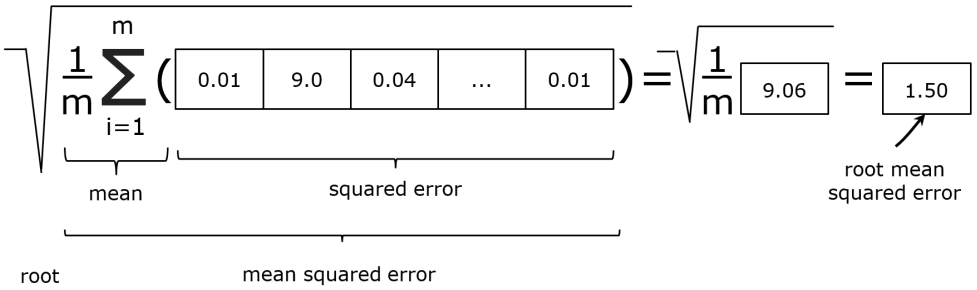


Figure 2.19 RMSE: we first compute MSE and then calculate its square root.

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 # A
    mse = (error ** 2).mean() # B
    return np.sqrt(mse) # C
```

#A Compute the difference between the prediction and the target.
#B Compute MSE: first compute the squared error and then calculate its mean.
#C Take the square root to get RMSE.

In A, we compute elementwise 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 B, 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 C, we compute the square root to get RMSE.

Elementwise 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).

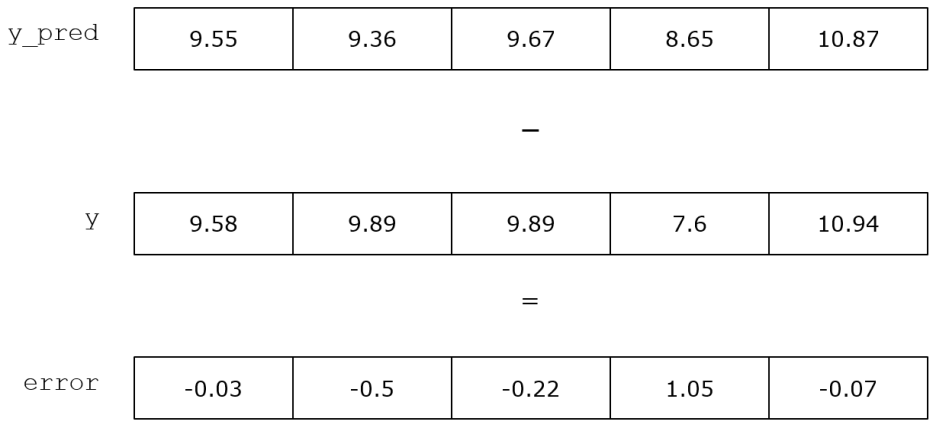


Figure 2.20 The elementwise difference between `y_pred` and `y`. The result is written to the `error` array.

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).

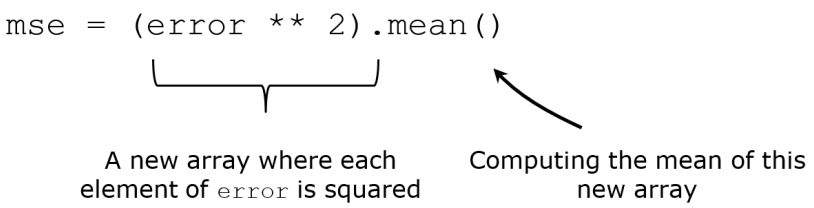


Figure 2.21 To calculate MSE, we first compute the square of each element in the error array and them compute the mean value of the result.

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

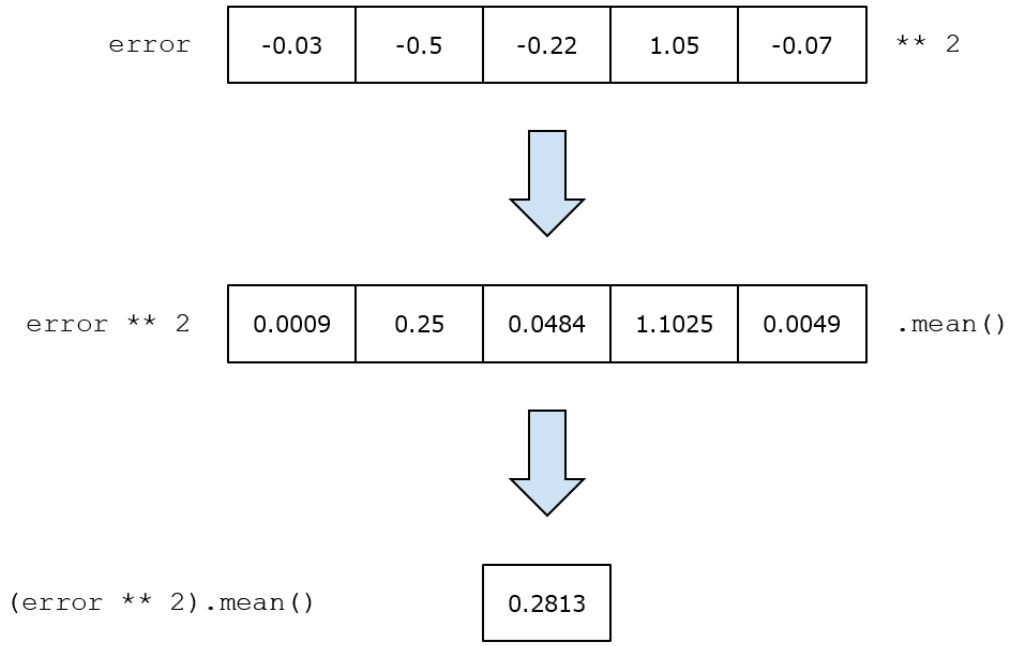


Figure 2.22 The power operator (**) applied elementwise to the error array. The result is another array in which each element is squared. Then we compute the mean of the array with the squared error to compute MSE.

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 this model is better.

2.4.3 Validating the model

In this example, 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 the validation dataset; we intentionally didn't use it for training and kept 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`. 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 it:

```
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 code above we already see some duplication: Training and validation tests require the same preprocessing. Thus, it makes sense to move this logic to a separate function. We can call this function `prepare_X` because it takes in a dataframe and creates a matrix `X` from it.

Listing 2.4 The prepare_X 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:

```
# train the model
X_train = prepare_X(df_train)
w_0, w = linear_regression(X_train, y_train)

# apply it to validation dataset to check RMSE
X_val = prepare_X(df_val)
y_pred = w_0 + X_val.dot(w)
print('validation:', rmse(y_val, y_pred))
```

This gives us a way to check whether anything we do with the model leads to an improvement. 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." 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 out from 2017:

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

This operation is an elementwise 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 2.4 Creating the "age" feature in the prepare_X function

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

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

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

#A Create a copy of the input parameter to prevent side effects.
#B Create a copy of the base list with the basic features.
#C Compute the age feature.
#D Append 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 A, 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 C. 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 B, we create a copy for the list with the base features for the same reason. Later, we extend this list with new features D, but we don't want to change the original list. The rest of the code is the same as previously.

Let's test the new feature:

```
X_train = prepare_X(df_train)
w_0, w = 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. We can also look at the distribution of the predicted values:

```
sns.distplot(y_pred, label='pred')
sns.distplot(y_val, label='y')
plt.legend()
```

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

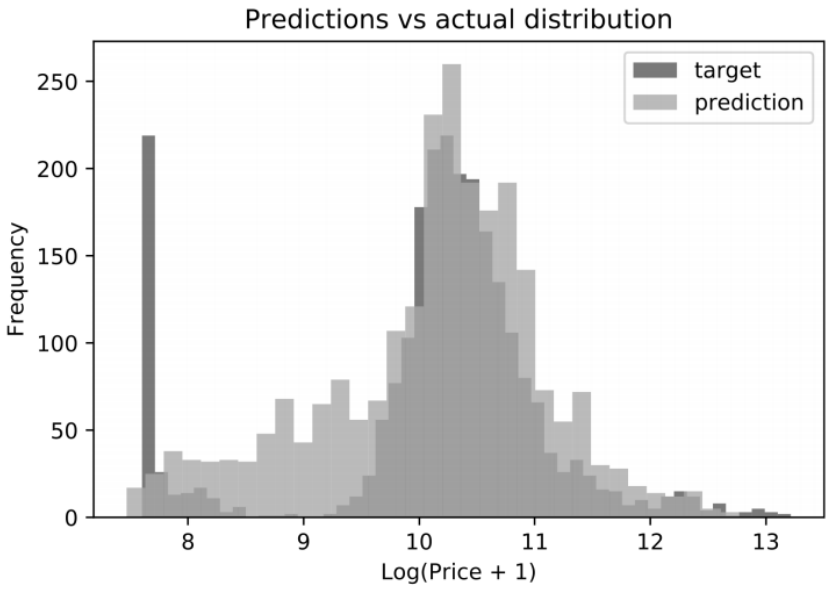


Figure 2.23 The distribution of predicted (light gray) versus actual (dark gray). With the new features, the model follows the original distribution closer than previously.

2.4.5 Handling categorical variables

Let's continue adding features. One of the columns we can use 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:

```
for v in [2, 3, 4]: # A
    feature = 'num_doors_%s' % v # B
    value = (df['number_of_doors'] == v).astype(int) # C
    df[feature] = value #D
    features.append(feature)
```

#A Iterate over possible values of the "number of doors" variable.
#B Give a feature a meaningful name, such as "num_doors_2" for v=2.
#C Create the one-hot encoding feature.
#D Add the feature back to the dataframe, using the name from B.

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 C:

```
(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 elementwise 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; the elements are False otherwise. The operation creates a series of True/False values. Because `v` has three values (2, 3, and 4), the operation creates three series (figure 2.24).

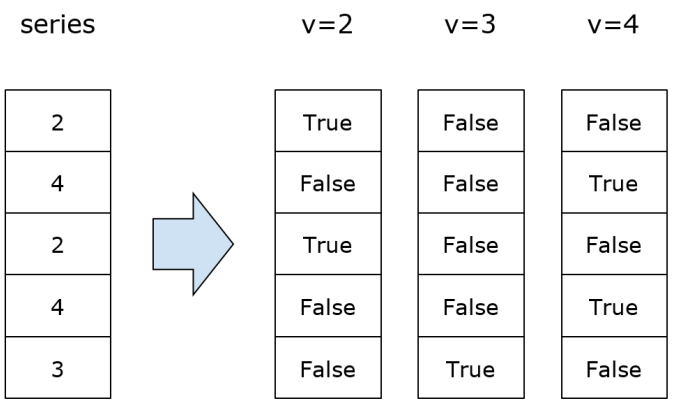


Figure 2.24 We use the == operator to create the new series from the original one: one for two doors, one for three doors, and one for four doors.

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).

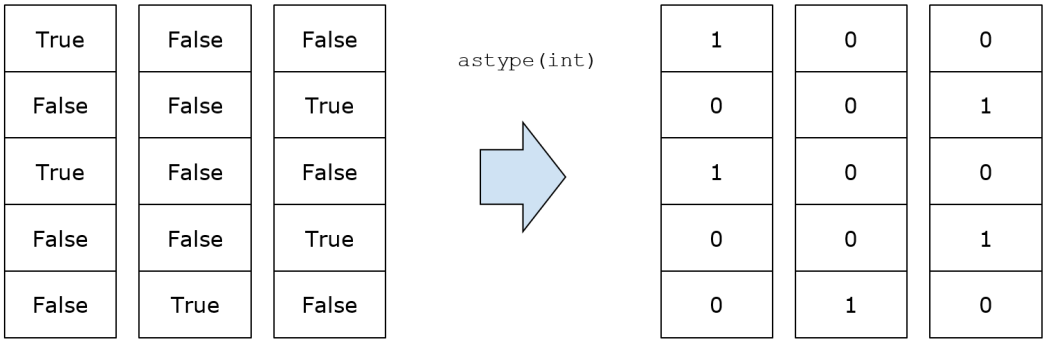


Figure 2.25 Using `astype(int)` to convert series with boolean values to integers

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 encoding 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. 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 2.5 Handling categorical variables "number of doors" and "make" 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]: #A
        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']: #B
        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
```

#A Encode the number of doors variable.
#B Encode the make variable.

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

```
X_train = prepare_X(df_train)
w_0, w = 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 2.6 Handling categorical variables "number of doors" and "make" 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)']: #A
        feature = 'is_type_%s' % v
        df[feature] = (df['engine_fuel_type'] == v).astype(int)
        features.append(feature)

    for v in ['automatic', 'manual', 'automated_manual']: #B
        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']: #C
        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']: #D
        feature = 'is_mc_%s' % v
        df[feature] = (df['market_category'] == v).astype(int)
        features.append(feature)

    for v in ['compact', 'midsize', 'large']: #E
        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']: #F
        feature = 'is_style_%s' % v
        df[feature] = (df['vehicle_style'] == v).astype(int)
        features.append(feature)

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

#A Encode the type variable.
#B Encode the transmission variable.
#C Encode the number of driven wheels.
#D Encode the market category.
#E Encode the size.
#F Encode the style.

Let's test it:

```
X_train = prepare_X(df_train)
w_0, w = 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, NumPy version, the version of NumPy dependencies, OS, and other factors. But a 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. Remember 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. 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. 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:

$$X^TX+ \alpha I$$

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^TX, which adds α to all the diagonal elements of X^TX.

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).

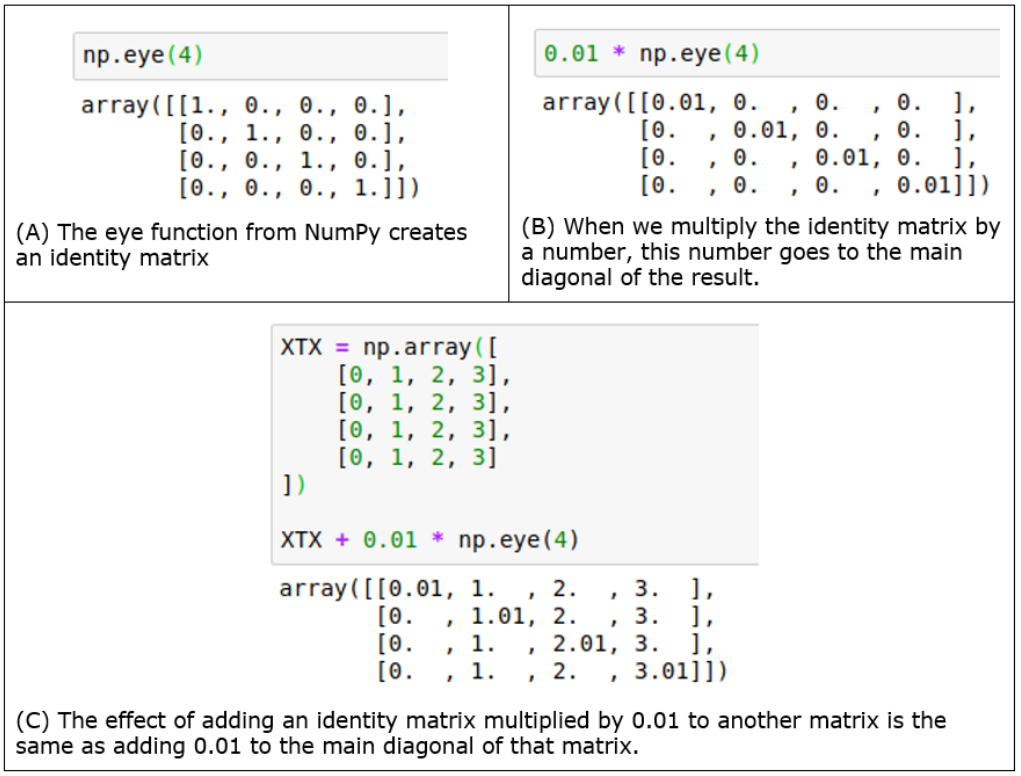


Figure 2.26 Using an identity matrix to add 0.01 to the main diagonal of a square matrix.

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

Listing 2.7 Linear regression with regularization

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

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

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

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

#A Control the amount of regularization by using the parameter r.
#B Add 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 = 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 unregularized solution, and get very large numbers. Then we try 0.001 and increase it by 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 = 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 run a grid search 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 = 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 = 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.

2.4.7 Using the model

As 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:

```
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 can use our model to suggest the price:

```
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 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

There are a few other things you can try to make the model better. For example,

- *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 re-evaluate the quality of the model in terms of RMSE.

- *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.

2.5.2 Other projects

There are other projects you can do now:

- *Predict the price of a house.* You can take the Boston house prices dataset from https://www.kaggle.com/vikrishnan/boston-house-prices or https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_boston.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.

2.6 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.

3

Machine learning for classification

This chapter covers

- Doing 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 a process in which 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 and this way keep the users.

Naturally, we can use machine learning for that: we can use the past data about customers who churned and, based on that, create a model for identifying present customers who are about to go away. 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 the previous chapter, we learned that there are many supervised machine learning models, and we specifically mentioned ones that can be used for binary classification, including logistic regression, decision trees, and neural networks. In this chapter, we will start with the simplest one: logistic regression. Even though it's indeed the simplest, it's still very powerful and has many advantages over other models: it's fast and easy to understand, and the 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. There is a problem: some of our customers are churning. They no longer are using our services and going to a different provider. We would like to prevent that from happening. For that, 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 predictions of the model.

We have collected a dataset where we recorded some information about our customers: what type of services they use, 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 is the following:

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 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, such as 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 can 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. This is not a large dataset but should be quite enough to train a decent model.

Next, let's look at the first couple of rows using `df.head()` (figure 3.1). There are quite a few columns there, so they all won't fit on a screen.

```
df.head()
```

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

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

Instead, we can transpose the dataframe, 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). We can do the transpose using the `T` function:

```
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 `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.

We see that the dataset has a few columns:

- *CustomerID* — the ID of the customer

- *Gender* — male/female
- *Senior Citizen* — 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
- *Phone service* — whether they have phone service (yes/no)
- *Multiple lines* — whether they have multiple phone lines (yes/no/no phone service)
- *Internet service* — the type of internet service (no/fiber/optic)
- *Online security* — if online security is enabled (yes/no/no internet)
- *Online backup* — if online backup service is enabled (yes/no/no internet)
- *Device protection* — if the device protection service is enabled (yes/no/no internet)
- *Tech support* — if the customer has tech support (yes/no/no internet)
- *Streaming TV* — if the TV streaming service is enabled (yes/no/no internet)
- *Streaming movies* — if the movie streaming service is enabled (yes/no/no internet)
- *Contract* — the type of contract (monthly/yearly/two years)
- *Paperless billing* — if the billing is paperless (yes/no)
- *Payment method* — payment method (electronic check, mailed check, bank transfer, credit card)
- *Monthly charges* — the amount charged monthly (numeric)
- *Total charges* — the total amount charged (numeric)
- *Churn* — if the client has canceled the contract (yes/no)

The most interesting one for us is churn. This is the target variable for our model, and 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
```

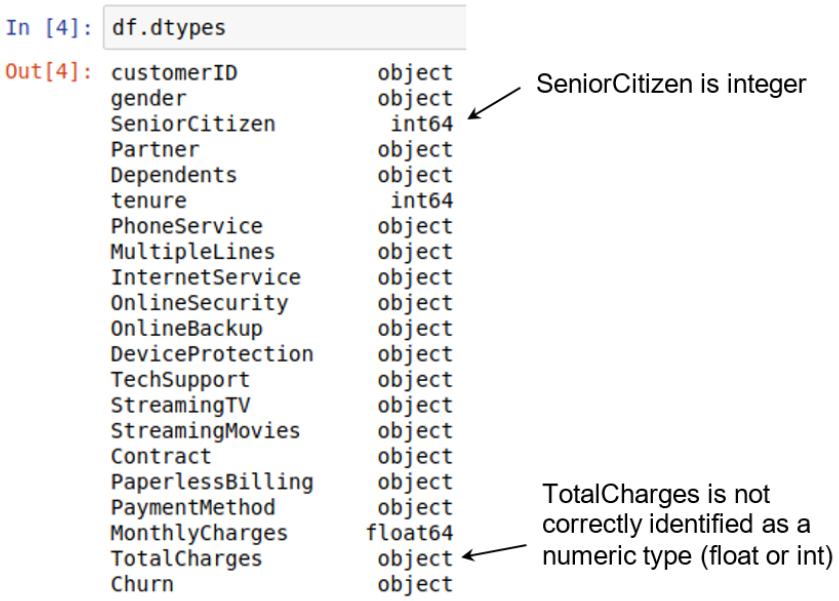


Figure 3.3 Automatically inferred types for all the columns of the dataframe. Object means a string. TotalCharges is incorrectly identified as object, but should be float.

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 type 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.

The other thing 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 non-numeric characters, Pandas has no other option but to declare the column as 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 pre-processing.

We can force this column to be numeric by converting it to numbers. There's a special function in Pandas that we can use: `to_numeric`. By default, this function raises an exception when it sees non-numeric data (such as spaces), but we can make it skip these cases by specifying

the `errors='coerce'` option. This way Pandas will replace all non-numeric values with a `NaN` (not a number):

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

To confirm that there are indeed non-numeric characters in the data, 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.X).

```
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 A We can spot non-numeric data in a column by parsing the content as numeric and see at the rows there the parsing fails.

Now it's up to us to decide what to do with these missing values. There are many things we can do with them, but we can 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, while others start with a capital letter. 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 will 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", what this does is convert "yes" to `True` and "no" to `False` (figure 3.5B). When we perform casting by using the `astype(int)` function, we 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.

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

```
0     no
1     no
2    yes
3     no
4    yes
```

(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

We did 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 there's 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 first by shuffling the original dataset and then splitting it in such a way that the test set contains 20% of 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.

	customerid	gender	tenure
0	7590-vhveg	female	1
1	5575-gnvde	male	34
2	3668-qpybk	male	2
3	7795-cfocw	male	45
4	9237-hqitu	female	2
5	9305-cdskc	female	8
6	1452-kiovk	male	22
7	6713-okomc	female	10
8	7892-pookp	female	28
9	6388-tabgu	male	62

	customerid	gender	tenure
8	7892-pookp	female	28
2	3668-qpybk	male	2
5	9305-cdskc	female	8
6	1452-kiovk	male	22
3	7795-cfocw	male	45
1	5575-gnvde	male	34
0	7590-vhveg	female	1
7	6713-okomc	female	10
4	9237-hqitu	female	2
9	6388-tabgu	male	62

	customerid	gender	tenure
8	7892-pookp	female	28
2	3668-qpybk	male	2
5	9305-cdskc	female	8
6	1452-kiovk	male	22
3	7795-cfocw	male	45
1	5575-gnvde	male	34
0	7590-vhveg	female	1
7	6713-okomc	female	10

	customerid	gender	tenure
4	9237-hqitu	female	2
9	6388-tabgu	male	62

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

There are a few parameters in this function:

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 making sure 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; that's why it's important to fix the random seed. This way we ensure that every time we shuffle the data, the final arrangement of rows will be the same.

	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.

There's 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). It will lead to problems when we need to split the dataframes again later, so we need to reset the index for each of the dataframes:

```
df_train_full = df_train_full.reset_index(drop=True)
df_test = df_test.reset_index(drop=True)
```

After the indices are reset, we can see that they are again consecutive numbers that start with 0 and then incrementally increase with each row.

In the previous chapter, we split the data into three parts: train, validation, and test. However, the `train_test_split` function can split the data into 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).

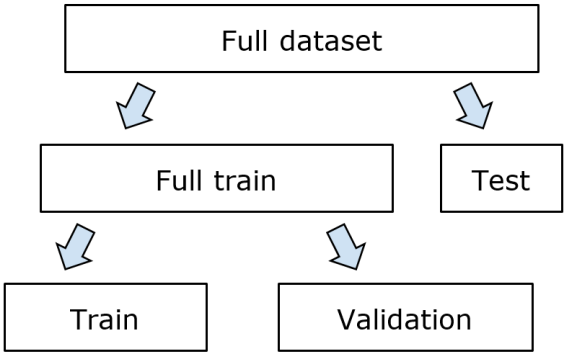


Figure 3.8 Because `train_test_split` can split a dataset into only two parts, but we need three, we perform the split two times. First, we split the entire dataset into full train and test, and then we split full train into train and validation.

So 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) #A

df_train = df_train.reset_index(drop=True) #B
df_val = df_val.reset_index(drop=True) #B

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

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

#A Set the random seed when doing the split to make sure every time we run the code, the result is the same.
#B Reset the indexes.
#C Take the column with the target variable churn and save it outside the dataframe.
#D Delete the churn columns from both the 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 model we can build afterward.

One of the things we should always do is check for any missing values in the dataset. This is important because many machine learning models cannot easily deal with missing data. We have already found a problem with the Total Charges 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 there are no missing values in the dataset and we 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 There is no need 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 1521 of 5634 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. So as we see in the training dataset, approximately 27% of the customers stopped using our services, and the rest remained as customers.

For churn, 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()
```

The value we get is also 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 the following

$$(1/n) \cdot \sum_{i=1}^{n} y_i$$

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, and this 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 are three times more people who didn't churn in our data than who did churn, and we say that the non-churn 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.

There are both categorical and numerical variables in our dataset. Both 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 that it should be just a few for each column, but let's verify it:

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

```
In [31]: df_train_full[categorical].nunique()

Out[31]: 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.

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 time doing extra preparing and cleaning the data; everything is already good to go.

Now we will come to another important part of exploratory data analysis: understanding which features may be important for our model.

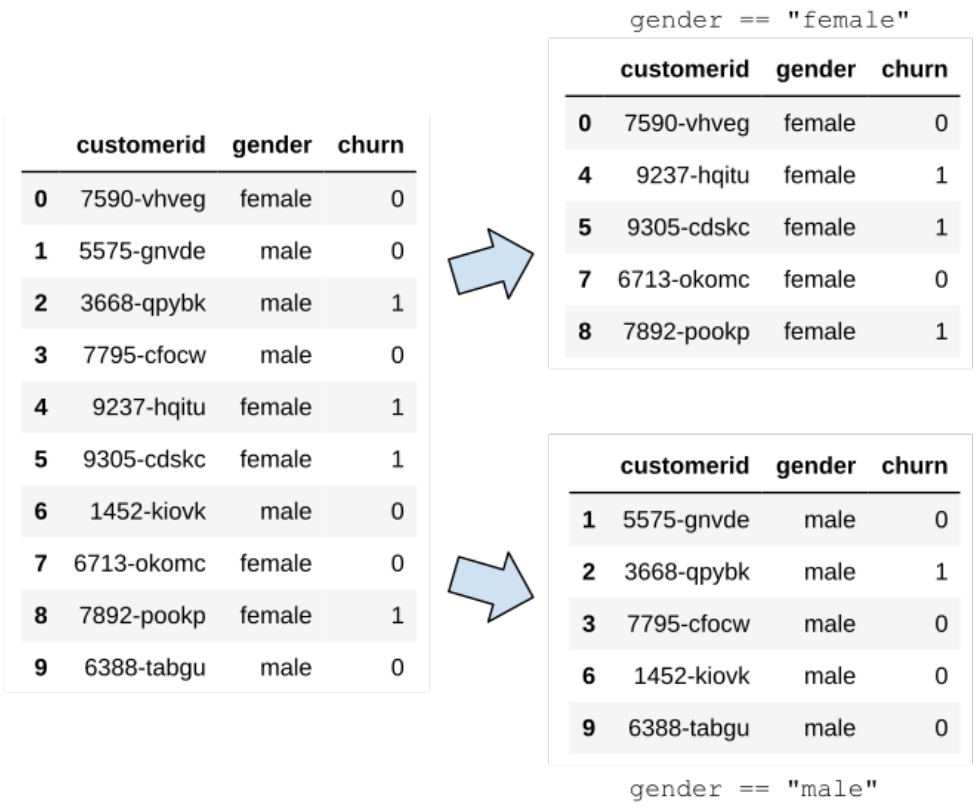
3.1.4 Feature importance

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.



	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"`.

The gender variable can take two values, female and male. So there are two groups of customers: ones that have `gender == 'female'` and ones that have `gender == 'male'` (figure 3.12).

So 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 will be the group churn rate. When we have it, we can compare it with the global churn rate — 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. To compute the churn rate for all 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 can 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%, while the global churn rate is 27% (figure 3.13). The difference between group rate 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.

```
In [18]: global_mean = df_train_full.churn.mean()
         round(global_mean, 3)

Out[18]: 0.27

In [19]: 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 very useful variable when predicting churn.

Now let's take a look at another variable: partner. It takes values 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 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()
```

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).

```
In [20]: 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 *risk ratio*, where *risk* refers to the risk of having the effect. In our case, the effect is churn, so it's risk of churning:

$$risk = group\ rate\ /\ global\ rate$$

For "gender == female", for example, the risk of churning is 1.02:

$$risk = 27.7\%\ /\ 27\% = 1.02$$

Risk is a number between 0 and infinity. It has a nice interpretation that tells u, 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 will be close to 1. This group has the same level of risk as the rest of the population. So customers in the group are as likely to churn as anyone else. In other words, a group with risk close to 1 is not risky at all (figure 3.12, 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, for example, means that the clients in this group are two times less likely to churn than clients in general (figure 3.12, 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).

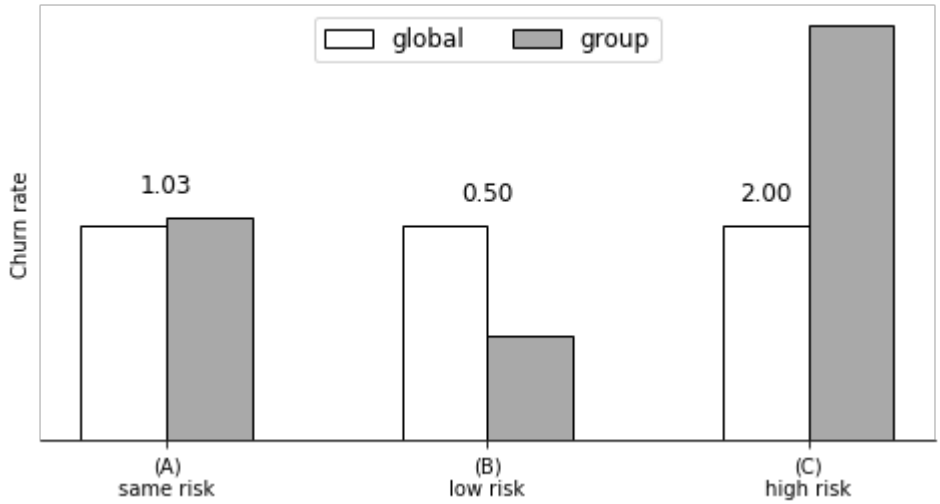


Figure 3.15 Churn rate of different groups compared with the global churn rate. In group (A) the rates are approximately the same, so the risk of churn is around 1. In group (B) the group churn rate is smaller than the global rate, so the risk is around 0.5. Finally, in group (C) the group churn rate 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:

$$risk = negative\ outcome\ rate\ in\ group\ 1\ /\ negative\ outcome\ rate\ in\ group\ 2$$

If medicine turns out to be effective, it's said to reduce the risk of having the negative outcome, and the value of 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).

Table 3.1 Churn rates and risks for 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

We did this only from two variables. Let's now do this for all the categorical variables. To do that, we will need a piece of code that checks all the values a variable has, and computes churn rate for each of these values.

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 not difficult to translate to Pandas:

```
df_group = df_train_full.groupby(by='gender').churn.mean() #A
df_group['diff'] = df_group['mean'] - global_mean #B
df_group['risk'] = df_group['mean'] / global_mean #C
df_group
```

#A Compute the AVG(churn).
#B Calculate the difference between group churn rate and global rate.
#C Calculate the risk of churning.

In (A) we calculate the `AVG(churn)` part. We do this in the same way we compute the mean of a dataframe, and because it comes after the groupby expression, Pandas knows that it should be computed for each group. In (B) we create another column, diff, where we will keep the

difference between the group mean and the global mean. Likewise, in (C) 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: #A
    df_group = df_train_full.groupby(by=col).churn.agg(['mean']) #B
    df_group['diff'] = df_group['mean'] - global_mean
    df_group['rate'] = df_group['mean'] / global_mean
    display(df_group) #C
```

#A Loop over all categorical variables.
#B Perform group by for each categorical variable .
#C Display the resulting dataframe.

Two things are different in this code. The first thing is that instead of manually specifying the column name, we iterate over all categorical variables.

The second one 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.

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: senior citizen

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: phone service

Figure 3.17 Churn rate difference and risk for four categorical variables: gender, senior citizen, partner, and phone service.

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 less 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.

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: tech support

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 tech support and contract. People with no tech support and month-to-month contracts tend to churn a lot more than clients from other groups, while people with tech support and two-year contracts are very low-risk clients.

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.

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 very 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 is the most important feature and whether tech support is more useful than 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 this, machine learning models will not work. 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 get to 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, it means that the categorical variable and the target are independent and thus the variable will not be very 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): #A
    return mutual_info_score(series, df_train_full.churn) #B

df_mi = df_train_full[categorical].apply(calculate_mi) #C
df_mi = df_mi.sort_values(ascending=False).to_frame(name='MI') #D
df_mi
```

#A Create a stand-alone function for calculating mutual information.
#B Use the mutual_info_score function from scikit-learn.
#C Apply the function from (A) to each categorical column of the dataset.
#D Sort values of the result.

In C we use the apply method to apply the `calculate_mi` function we defined in A 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 A takes only one parameter: series. This is a column from the dataframe on which we invoked the `apply()` method. In B 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 services 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, online security, and tech support are among the most important features (figure 3.19). Indeed, we've already noted that contract and tech support are quite informative. It's also not surprising that gender is among the least important features, so we shouldn't expect it to be very 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, while 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.

But there are ways to measure the dependency between a binary target variable and a numerical variable. We can pretend that the binary variable is numerical (containing only numbers 0 and 1) 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 case of 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 while 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:

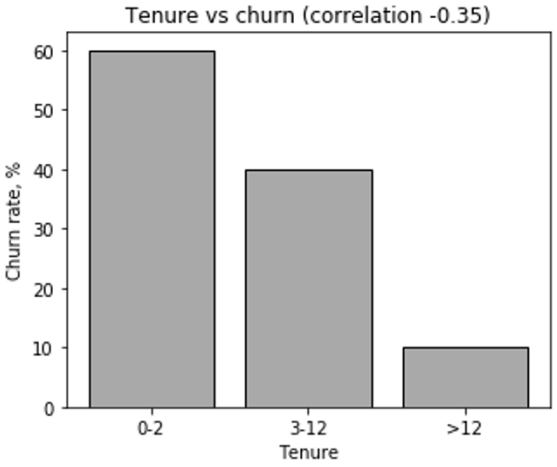
```
df_train_full[numerical].corrwith(df_train_full.churn)
```

We see the results in figure 3.20:

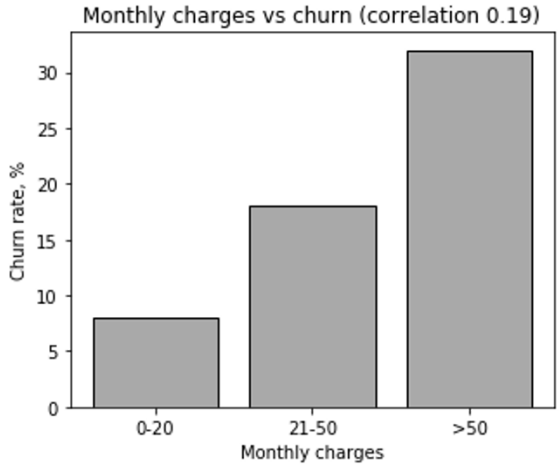
- The correlation between tenure and churn is -0.35: it has a negative sign, so the longer customers stay, the less often they tend to churn. For customers staying with the company for two months or less, the churn rate is 60%; for customers with tenure between 3 and 12 months, the churn rate is 40%; 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).
- Monthly charges 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).
- Total charges 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. Monthly charges has positive correlation: the more customers pay, the more likely they are to churn.



(A) Churn rate for different values of Tenure. The correlation coefficient is negative, so the trend is downwards: for higher values of tenure the churn rate is smaller.



(B) Churn rate for different values of monthly charges. The correlation coefficient is positive, so the trend is upwards: for higher values of monthly charges the churn rate is higher.

Figure 3.21 Churn rate for tenure (negative correlation of -0.35) and monthly charges (positive correlation of 0.19)

After doing initial exploratory data analysis, identifying important feature,s and getting some insights into the problem, we are ready to do the next step: feature engineering and model training.

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. Only after that will we 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 first 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 a very simple way of doing it. In this chapter, we will spend more time understanding and using it.

If a variable contract has possible values (monthly, yearly, and 2-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, while the remaining values are not active or *cold*, so they are 0.

To understand it 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 2-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=2-year

Let's consider two customers (table 3.3):

- 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, while the remaining contract columns, contract=monthly and contract=2-year, get 0.

As for the second customer, gender=male and contract=monthly get ones, while the rest of the columns get zeros (figure 3.22).

gender	contract
male	monthly
female	yearly

gender		contract		
female	**male**	**month**	**yearly**	**2-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 pretty 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, this process becomes tedious.

Luckily, we don't need to implement this by hand: we can use Scikit-Learn. There are multiple ways we can perform one-hot encoding in scikit-learn, and we will use `DictVectorizer`.

As the name suggests, `DictVectorizer` takes in a dictionary and *vectorizes* it — that is, 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).

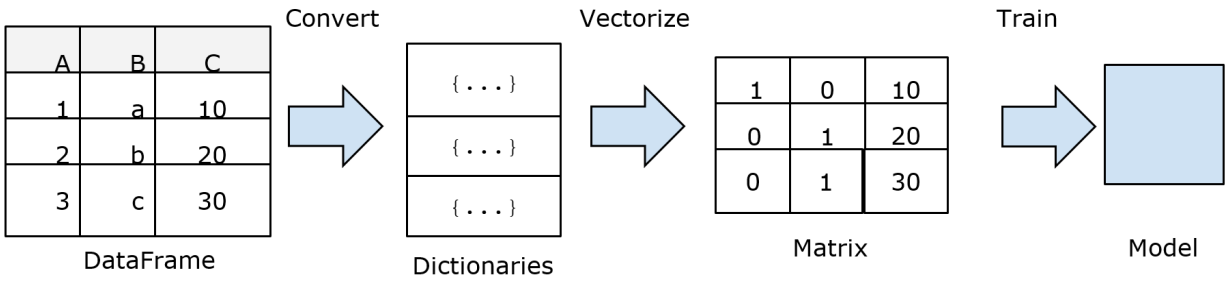


Figure 3.23 The process of creating a model. First, we convert a dataframe to a list of dictionaries; then we vectorize the list to a matrix; and finally, we use the matrix to train a model.

To use it, we need to convert our dataframe to a list of dictionaries. It's very simple to do in Pandas. Use the `to_dict` method with the `orient="rows"` parameter:

```
train_dict = df_train[categorical + numerical].to_dict(orient='rows')
```

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: monthly charges, tenure and total charges.

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.

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 to 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 will 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 will 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 very simple to implement, train, and use. Because of their simplicity, they are also very 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 0 and 1. 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, that 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 0 and 1. We use a special mathematical function for this purpose. This function is called *sigmoid*, and the full formula for the logistic regression model is

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

If we compare it with the linear regression formula, the only difference is this sigmoid function: in case of linear regression, we only have $w_0 + x_i^T w$. 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 0 and 1 (figure 3.24). It's defined this way:

$$sigmoid(x) = 1 / (1 + exp(-x))$$

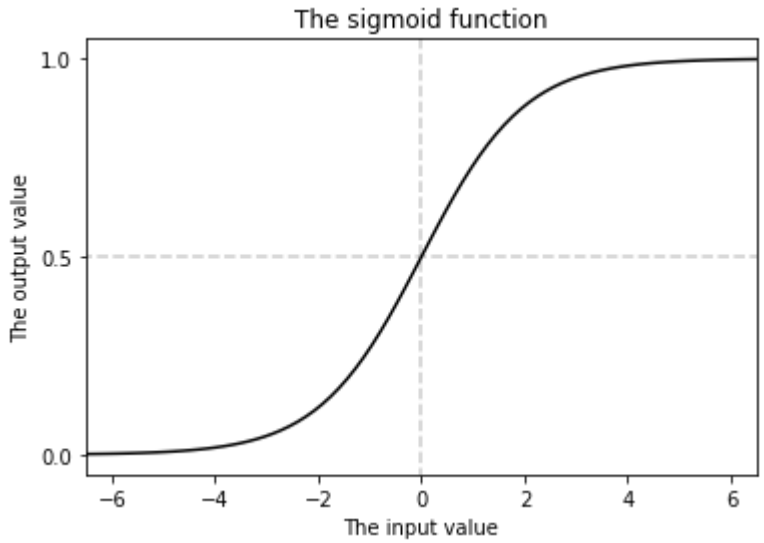


Figure 3.24 The sigmoid function outputs values that are always between 0 and 1. When the input is 0, the result of sigmoid is 0.5; for negative values the results are below 0.5 and start approaching 0 for input values less than -6. When the input is positive, the result of sigmoid is above 0.5 and approaches 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

$$g(x_i) = sigmoid\ (w_0 + x_{i1}w_1 + x_{i2}w_2 + \ldots + x_{in}w_n)$$

Or, using sum notation, as

$$g(x_i) = sigmoid(w_0 + \sum_{j=1}^{n} x_{ij}w_j)$$

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 0 and 1.

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 it now using Scikit-Learn.

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 there are quite a few parameters that we can change. 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,[3] 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 will 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='rows') #A
X_val = dv.transform(val_dict) #B
```

#A We perform one-hot encoding exactly in the same way as during training.
#B 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).

[3] https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

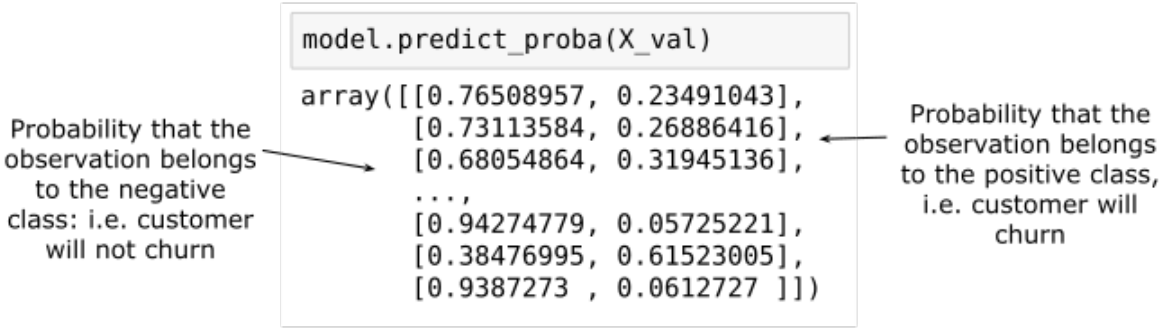


Figure 3.25 The predictions of the model: a two-column matrix. The first column contains the probability that the target is 0 (the client won't churn). The second column contains the opposite probability (the target is 1 and the client will churn).

These columns convey the same information. We know the probability of churn — it's *p*, then the probability of not churning is always *1 - p*. So we don't need both columns and can keep only one.

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. There are two positions 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. They tell us the probability of churning as a number between 0 and 1. 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 ones 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.

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 elementwise, 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).

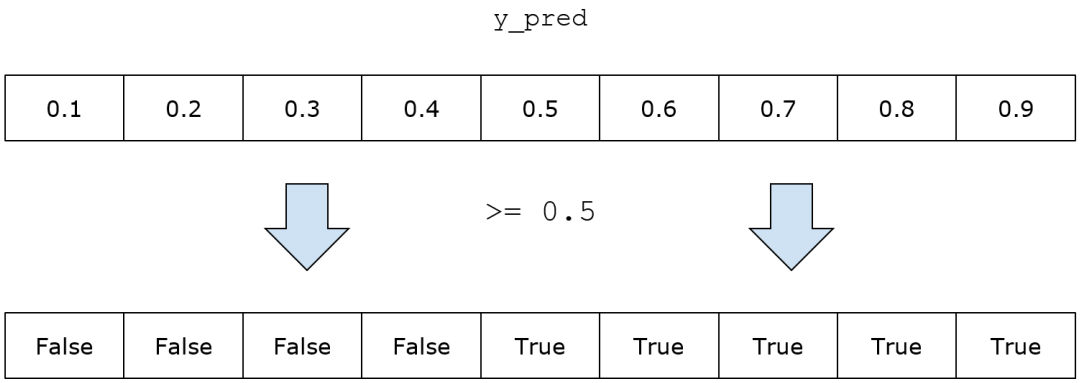


Figure 3.26. The >= operator is applied elementwise in NumPy. For every element, it performs the comparison, and the result is another array with `True` or `False` values, depending on the result of the comparison.

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 to 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: 1 if the customer churned and 0 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 3.28).

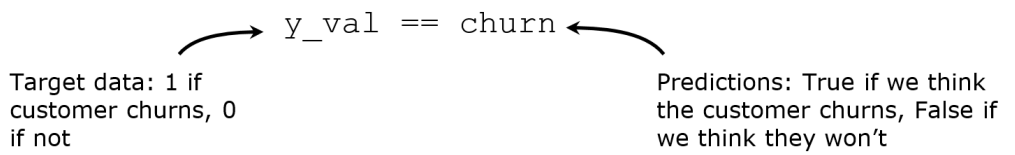


Figure 3.27. Applying the == operator to compare the target data with our predictions

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 3.29).

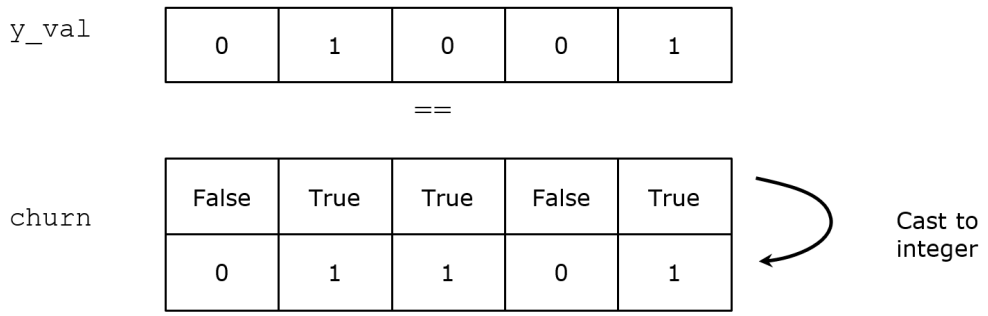


Figure 3.28 To compare the prediction with the target data, the array with predictions is cast to integer.

Like the `>=` operator, the `==` operator is applied elementwise. 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).

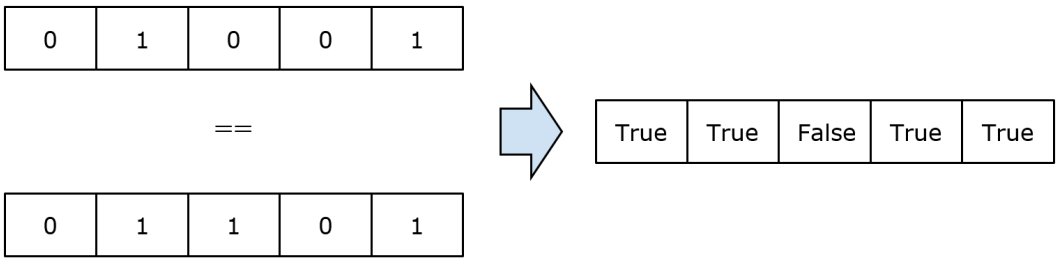


Figure 3.29 The `==` operator from NumPy is applied elementwise for two NumPy arrays.

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 by 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).

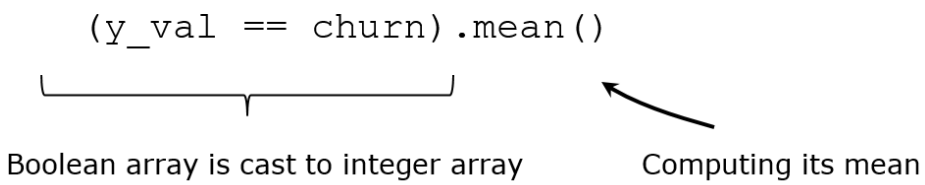


Figure 3.30 When computing the mean of a boolean array, NumPy first casts it to integers and then computes the mean.

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% cases. This is exactly 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 will 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)))
```

It 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 total charges.

Tenure and total charges are numerical variables, 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='rows')
dv_small = DictVectorizer(sparse=False)
dv_small.fit(train_dict_small)

X_small_train = dv_small.transform(train_dict_small)
```

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 what are the features that the small model will use. For that, as previously, we use `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 total charges, 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. There are three distinct values for contract: 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).

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

Table 3.2 The weights of a logistic regression model

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, and if we look at the sigmoid function (figure 3.31), we can see that for a negative value, the result is a probability lower than 0.5. In our case, this is the probability of a customer churning, and because it's less than 50% the negative sign of the bias term means that on average, we'd expect a customer to stay rather than churn.

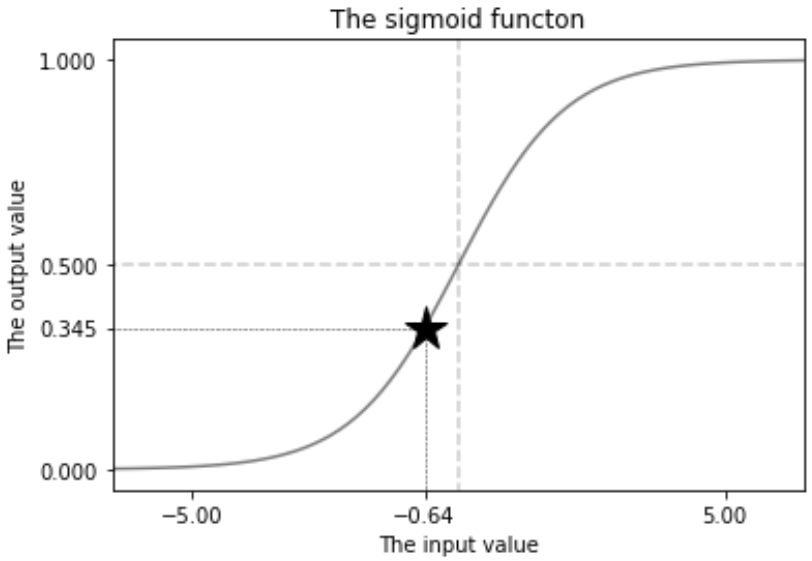


Figure 3.31 The bias term -0.639 on the sigmoid curve. The resulting probability is less than 0.5, which means that on average, we expect customers not to churn.

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, so the probability of churn 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 of 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).

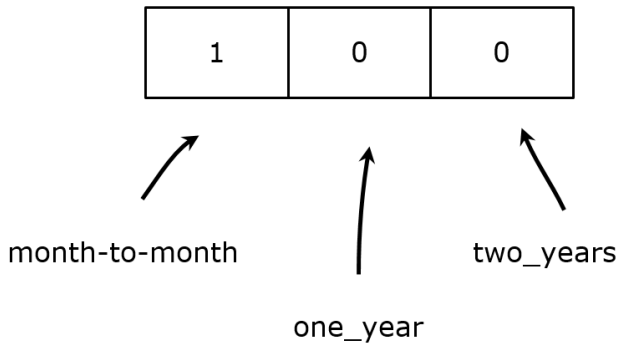


Figure 3.32 The one-hot encoding representation for a customer with a month-to-month contract

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).

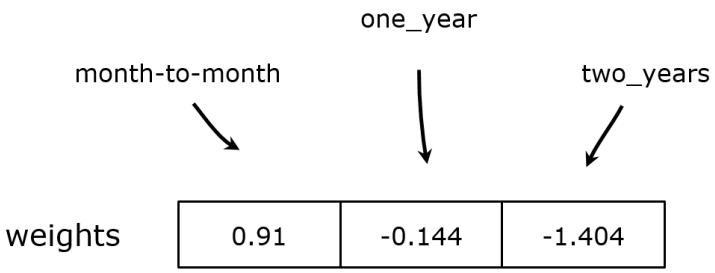


Figure 3.33 The weights of the contract=month-to-month, contract=one_year, and contract=two_years features

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).

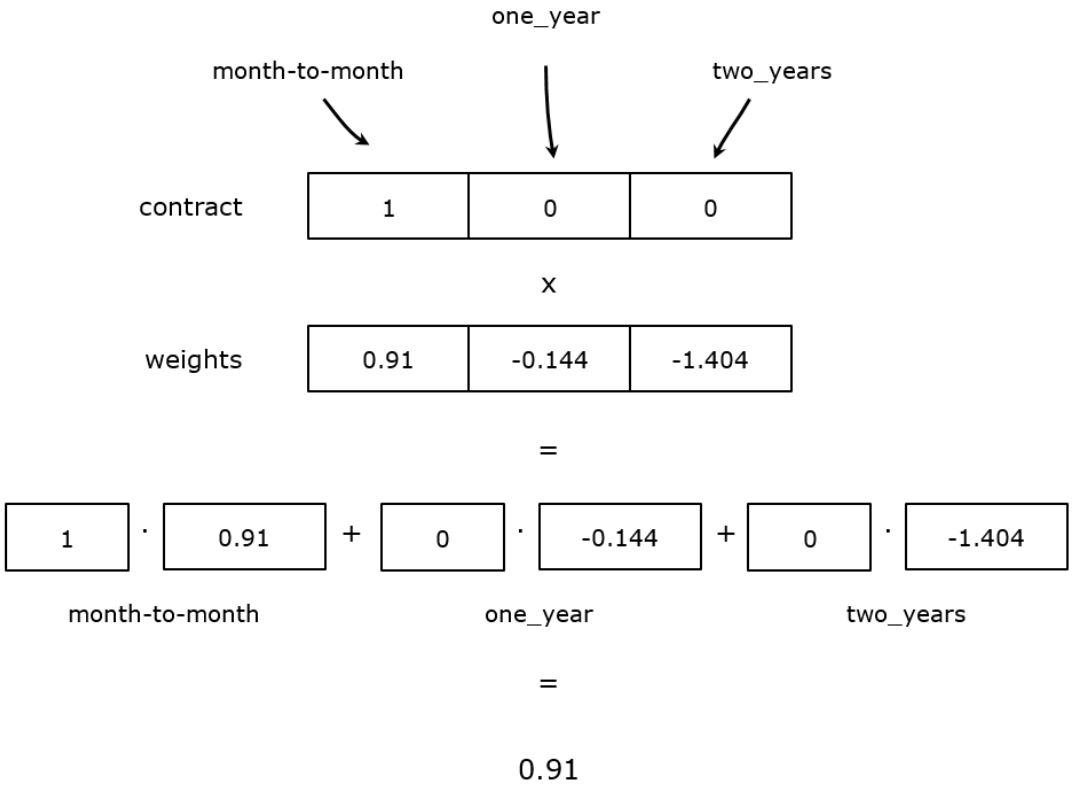


Figure 3.34 The dot product between the one-hot encoding representation of the contract variable and the corresponding weights. The result is 0.91, which is the weight of the hot feature.

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).

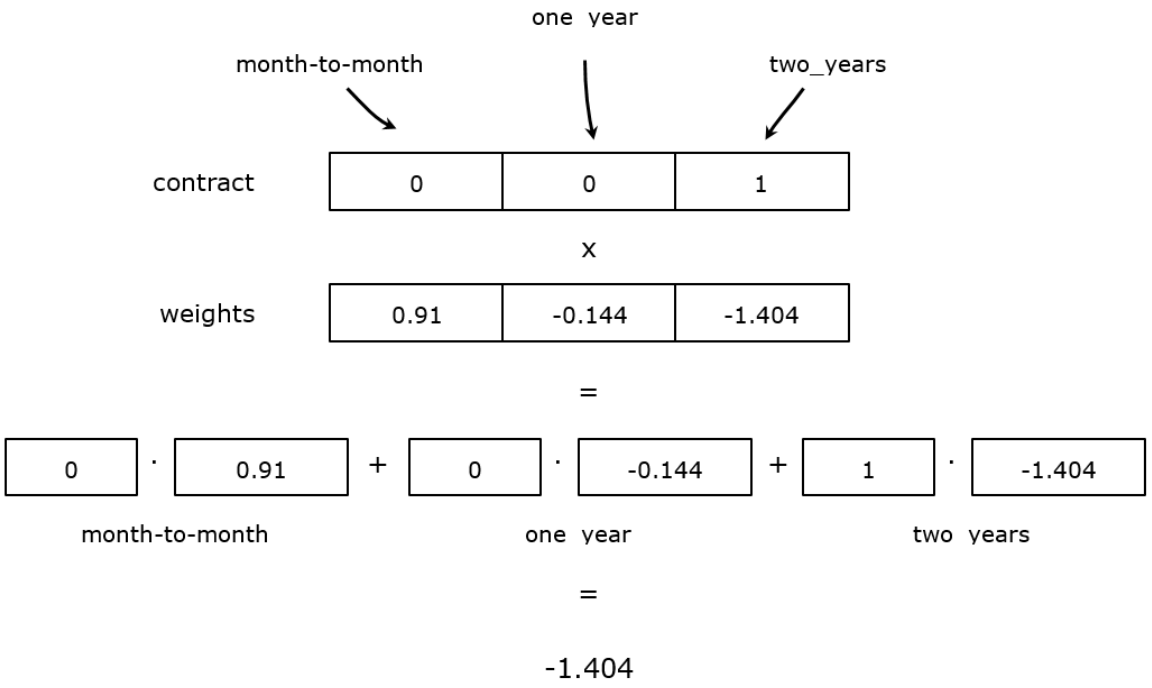


Figure 3.35 For a customer with a two-year contract, the result of the dot-product is -1.404

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 0, and when we multiply by 0, we get 0 again (figure 3.36).

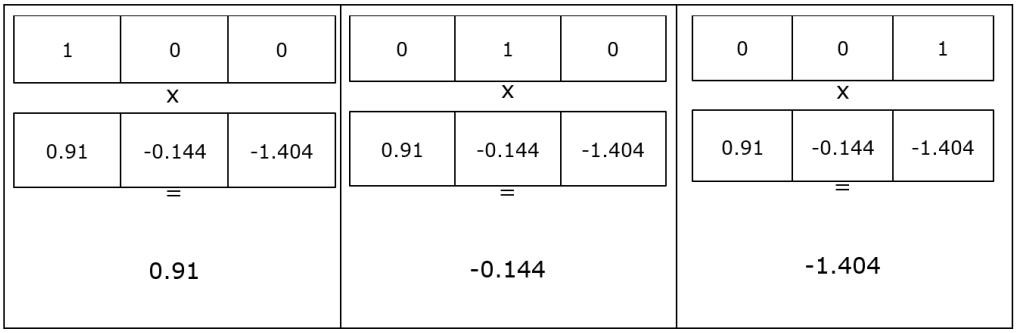


Figure 3.36 When we multiply the one-hot encoding representation of a variable by the weight vector from the model, the result is the weight corresponding to the hot feature.

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).

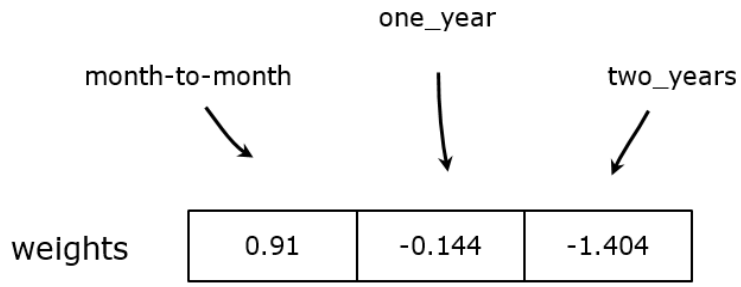


Figure 3.37 The sign of the weight matters. If it's positive, it's a good indicator of churn; if it's negative, it indicates a loyal customer.

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).

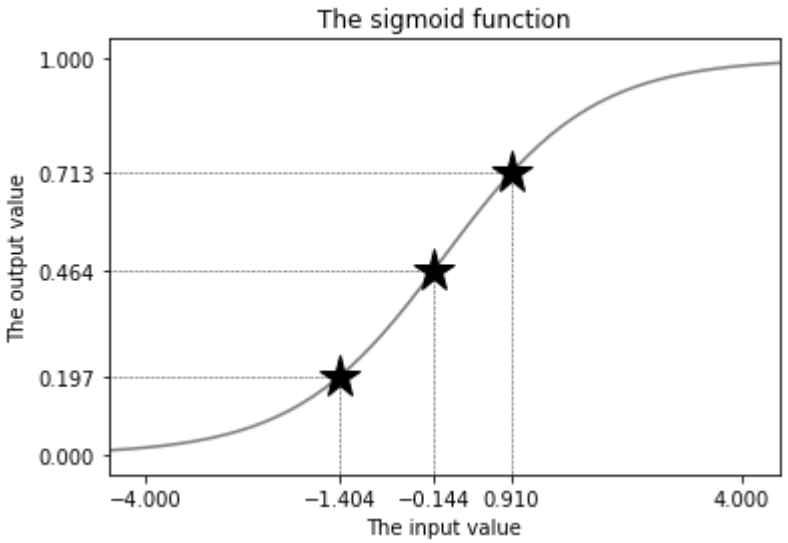


Figure 3.38. The weights for the contract features and their translation to probabilities. For contract=two-year, the weight is -1.404, which translates to very low probability of churn. For contract=one-year, the weight is -0.144, so the probability is moderate. And for contract=month-to-month, the weight is 0.910, and the probability is quite high.

Now let's have a look at the numerical features. We have two of them: tenure and total charges. 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 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, total changes, has weight 0. Because it's 0, 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.

$$-0.639 + 0.91 - 12 \cdot 0.097 + 0 \cdot 1000 = -0.893$$

bias	monthly contract	12 months of tenure	total charges don't matter	negative, so low likelihood of churn

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 * 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 (total charges) multiplied by the weight of this feature, but because it's 0, we don't do anything. The result stays -0.893 (figure 3.39).
- 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 (figure 3.41).

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).

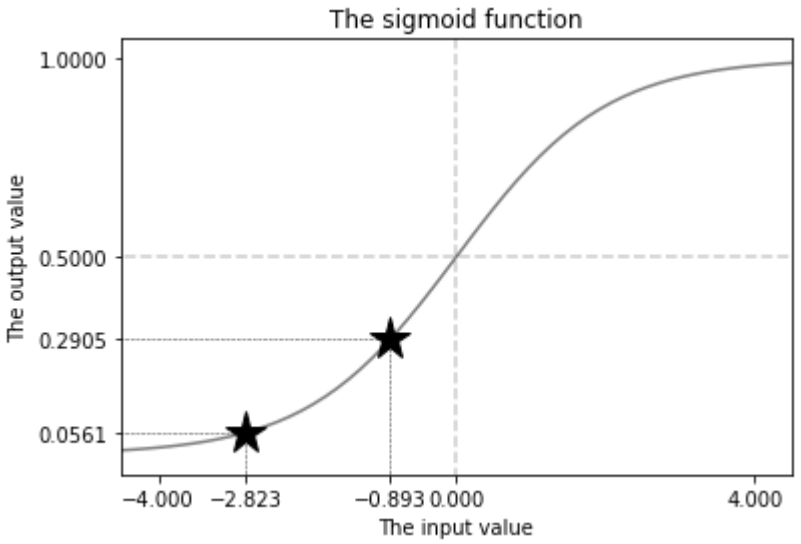


Figure 3.41 The scores of -2.823 and -0.893 translated to probability: 0.05 and 0.29, respectively

3.3.4 Using the model

Now we know how logistic regression works a lot better, 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 example above, 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 (monthly charges, tenure, and total charges).

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 her.

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 next 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 had a look 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 not-useful features from the dataset to make the model simpler,

faster, and potentially better. Try to exclude the two least useful features (gender and phone services) from the training data matrix, and see what happens with validation accuracy. What if we remove the most useful feature (contract)?

3.4.2 Other projects

There are numerous ways in which classification can be used 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 (became 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 very 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 returning 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). There are many datasets online that you can use 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).

3.5 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, while 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 won'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 0 and 1). 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.

A

Installing the libraries

This appendix covers:

- Installing Anaconda, a Python distribution that includes most of the scientific libraries we need
- Running a Jupyter Notebook service from a remote machine
- Installing and configuring the Kaggle command line interface tool for accessing datasets from Kaggle
- Creating an EC2 machine on AWS using the web interface and the command-line interface

A.1 Installing Python and Anaconda

For the projects in the book we will use Anaconda, a Python distribution that comes with most of the required machine learning packages that you'll need to use: NumPy, SciPy, Scikit-Learn, Pandas, and many more.

A.1.1 Installing Python and Anaconda on Linux

The instructions in this section will work regardless of whether you're installing Anaconda on a remote machine or your laptop. Although we tested it only on Ubuntu 18.04 LTS, this process should work fine for most Linux distributions.

> **NOTE** Using Ubuntu Linux is recommended for the examples in this book. It's not a strict requirement, however, and you should not have problems running the examples in other operating systems. If you don't have a computer with Ubuntu, it's possible to rent one online in a cloud. Please refer to the "Renting a server on AWS" section for more detailed instructions.

Almost every Linux distribution comes with a Python interpreter installed, but it's always a good idea to have a separate installation of Python to avoid trouble with the system Python. Using Anaconda is a great option: it's installed in the user directory and it doesn't interfere with the system Python.

To install Anaconda, you first need to download it. Go to https://www.anaconda.com/ and click Download. This should take you to https://www.anaconda.com/distribution/.

Select 64-Bit (x86) installer and the latest available version - 3.7 at the moment of writing (figure A.1).

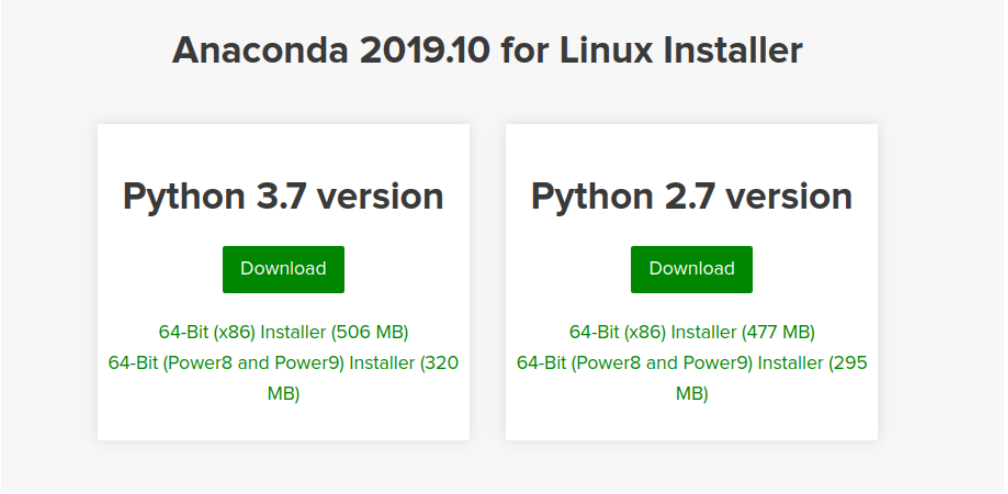


Figure A.1 Downloading the Linux Installer for Anaconda

> **NOTE** At the moment of writing, Python 2.7 was still available, but you should not use it. It won't be maintained past 2020, and most of the scientific packages, like NumPy, have already stopped supporting Python 2.

Next, copy the link to the installation package. In our case it was https://repo.anaconda.com/archive/Anaconda3-2019.07-Linux-x86_64.sh.

Now go to the terminal to download it:

```
wget https://repo.anaconda.com/archive/Anaconda3-2019.07-Linux-x86_64.sh
```

And then install it:

```
bash Anaconda3-2019.07-Linux-x86_64.sh
```

Read the agreement, type "yes" if you accept it, and then select the location where you want to install Anaconda. You can use the default location but don't have to.

During the installation, you'll be asked if you want to initialize Anaconda. Type "yes" and it will do everything automatically:

```
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>> yes
```

If you don't want to let the installer initialize it, you can do it manually by adding the location with Anaconda's binaries to the `PATH` variable. Open the ".bashrc" file in the home directory and add this line at the end:

```
export PATH=~/anaconda3/bin:$PATH
```

After the installation has completed, you can delete the installer:

```
rm Anaconda3-2019.07-Linux-x86_64.sh
```

Next, open a new terminal shell. If you're using a remote machine, you can simply exit the current session by pressing Ctrl-D and then log in again using the same `ssh` command as previously.

Now everything should work. You can test that your system picks the right binary by using the `which` command:

```
which python
```

If you're running on an EC2 instance from AWS, you should see something similar to this:

```
/home/ubuntu/anaconda3/bin/python
```

Of course, the path may be different, but it should be the path to the Anaconda installation.

Now you're ready to use Python and Anaconda.

A.1.2 Installing Python and Anaconda on Windows

LINUX SUBSYSTEM FOR WINDOWS

The recommended way to install Anaconda on Windows is to use the Linux Subsystem for Windows.

To install Ubuntu on Windows, open the Microsoft Store and look for "ubuntu" in the search box and then select "Ubuntu 18.04 LTS" (figure A.2).

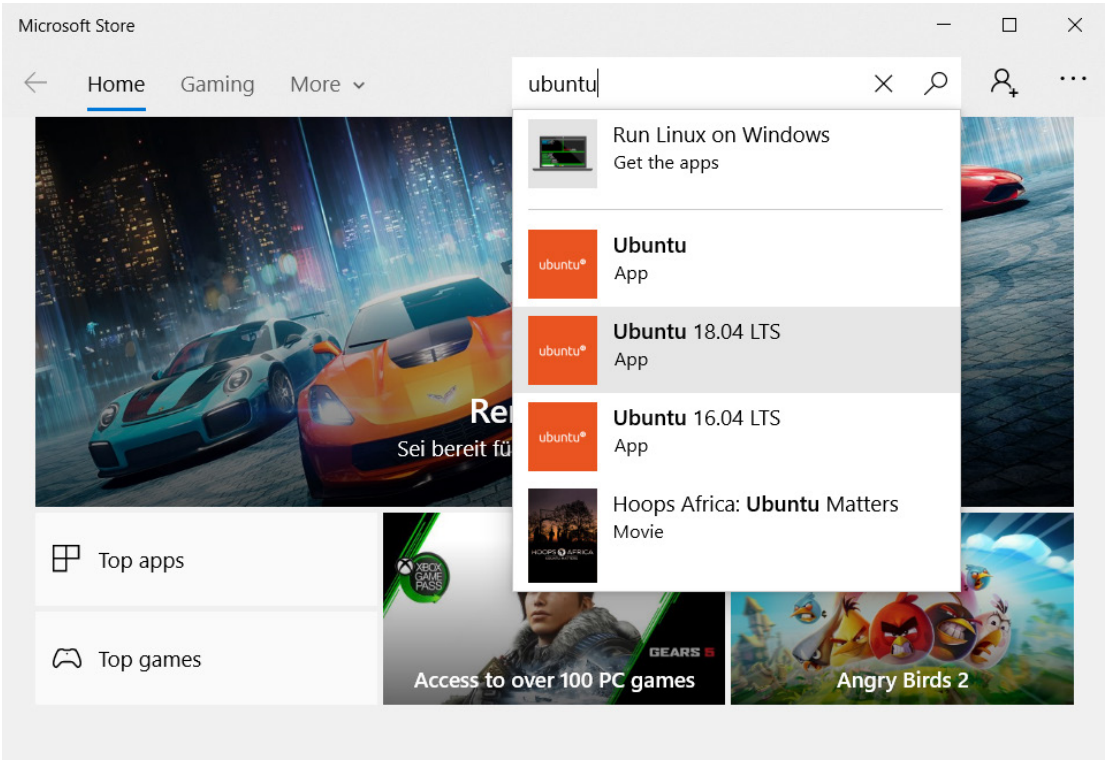


Figure A.2 Use Microsoft Store to install Ubuntu on Windows

To install it, we simply click “Get” in the next window (figure A.3).

Figure A.3 To install Ubuntu 18.04 for Windows, click "Get".

Once it's installed, we can use it by clicking the "Launch" button (figure A.4).

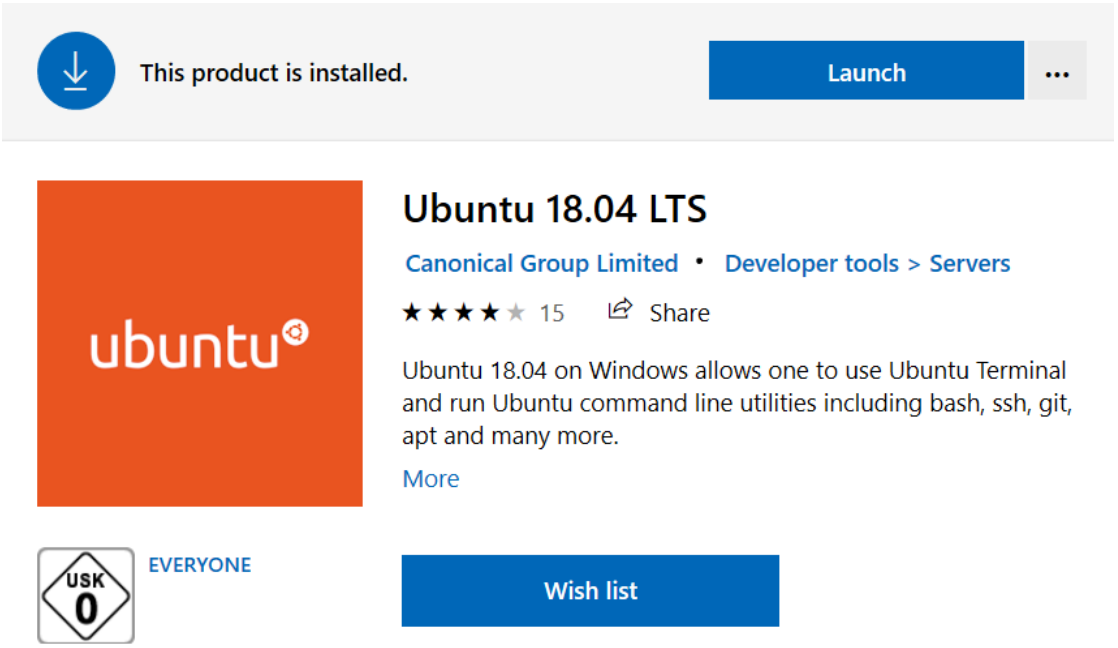


Figure A.4 Click "Launch" to run the Ubuntu terminal

When running it for the first time, it will ask you to specify the username and password (figure A.5). After that, the terminal is ready to use.

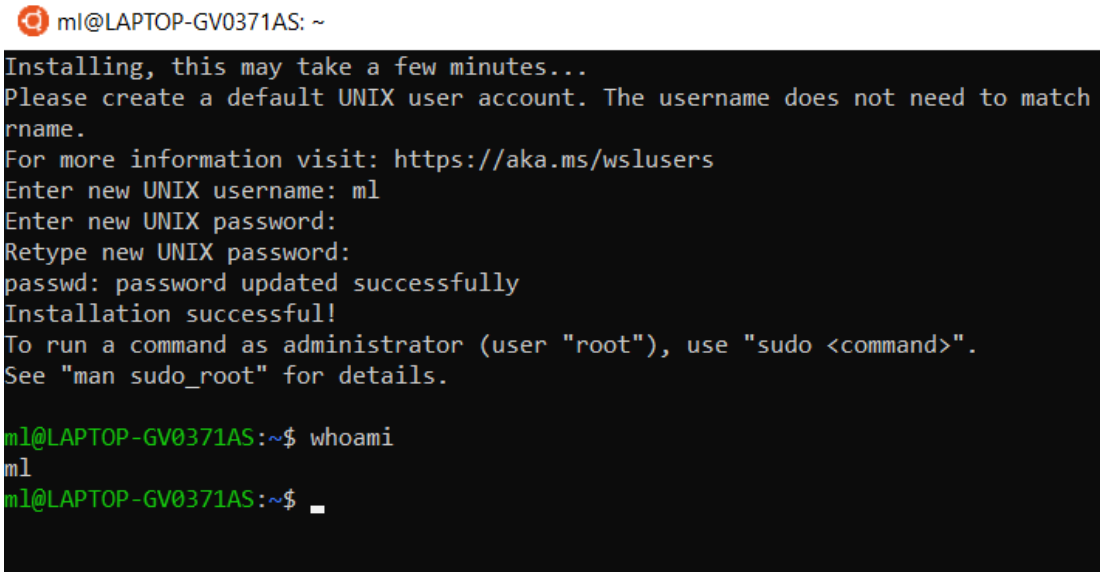


Figure A.5 The Ubuntu Terminal running on Windows

Now you can use the Ubuntu Terminal and follow the instructions for Linux to install Anaconda.

ANACONDA WINDOWS INSTALLER

Alternatively, we can use the Windows Installer for Anaconda. First, we need to download it from https://anaconda.com/distribution (figure A.6). Navigate to the "Windows Installer" section and download the 64-Bit Graphical Installer (or the 32 bit version, if you're using an older computer).

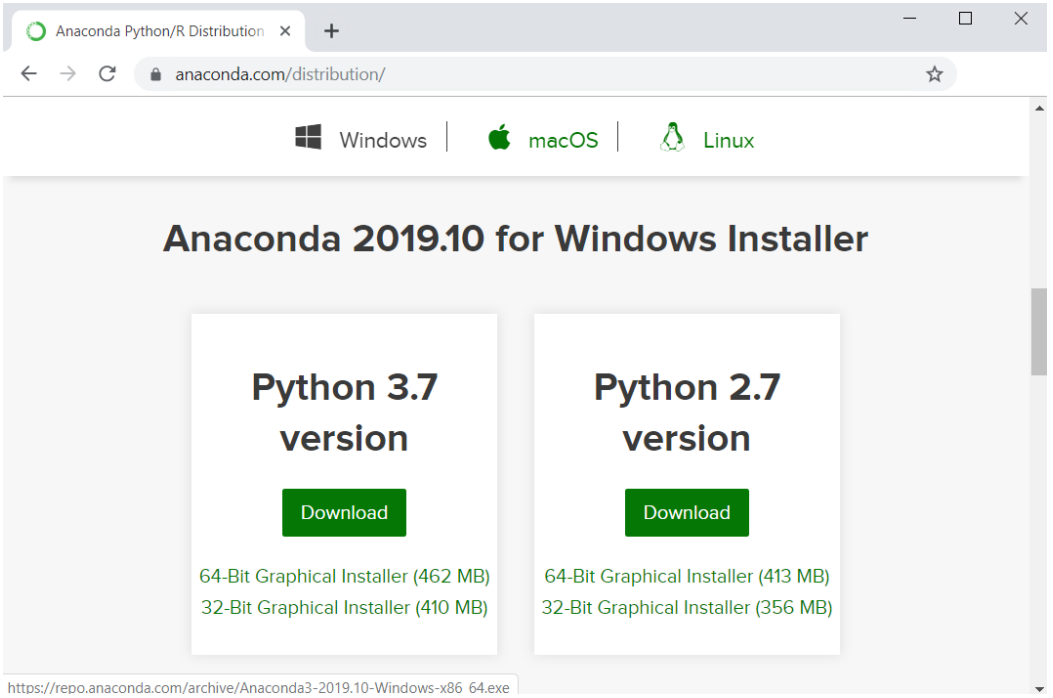


Figure A.6 Downloading the Windows Installer for Anaconda

Once you've downloaded the installer, simply run it and follow the setup guide (figure A.7).

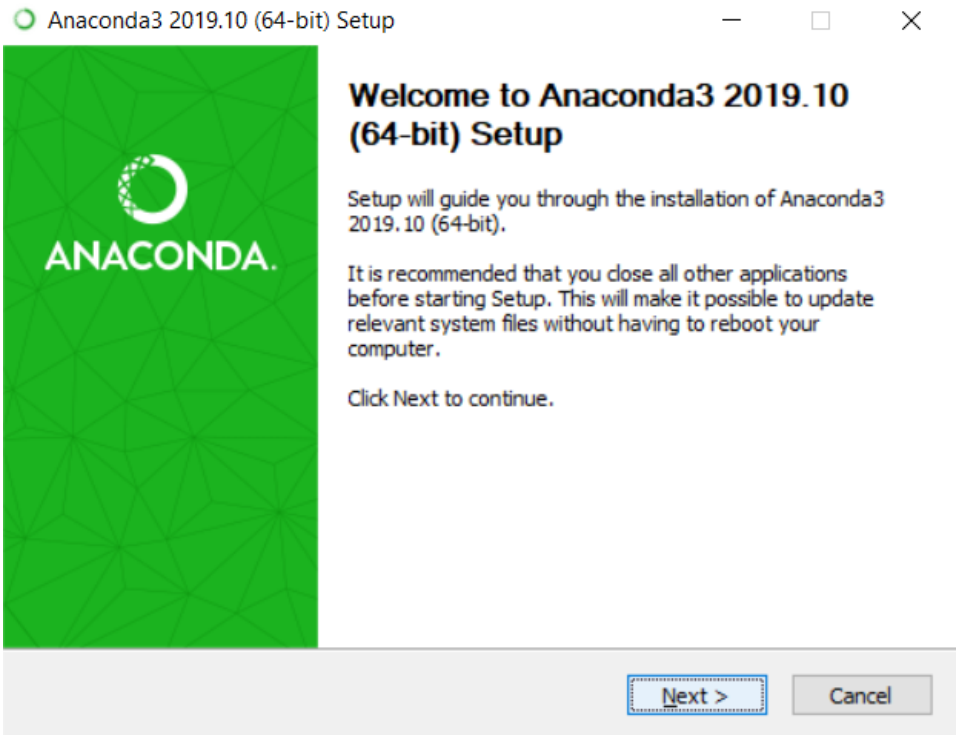


Figure A.7 The Installer for Anaconda

It's pretty straightforward and you should have no problems running it. After the installation is successful, you should be able to run it by choosing "Anaconda Navigator" from the start menu.

A.1.3 Installing Python and Anaconda on macOS

The instructions for macOS should be similar to Linux and Windows: select the installer with the latest version of Python and execute it.

A.2 Running Jupyter

A.2.1 Running Jupyter on Linux

Once Anaconda is installed, you can run Jupyter. First, you need to create a directory that Jupyter will use for all the notebooks:

```
mkdir notebooks
```

Then `cd` to this directory to run Jupyter from there:

```
cd notebooks
```

It will use this directory for creating notebooks. Now let's run Jupyter:

```
jupyter notebook
```

This should be enough if you want to run Jupyter on a local computer. If you want to run it on a remote server, such as an EC2 instance from AWS, you need to add a few extra command-line options:

```
jupyter notebook --ip=0.0.0.0 --no-browser
```

In this case, you must specify two things:

- The IP address Jupyter will use to accept incoming HTTP requests (`--ip=0.0.0.0`). By default it uses localhost, meaning that it's possible to access the Notebook service only from within the computer.
- The `--no-browser` parameter, so Jupyter won't attempt to use the default web browser to open the URL with the notebooks. Of course, there's no web browser on the remote machine, only a terminal.

NOTE **In the case of EC2 instances on AWS, you will also need to configure the security rules to allow the instance to receive requests on the port 8888. Please refer to the "Renting a server on AWS" section for more details.**

When you run this command, you should see something similar to this:

```
[C 04:50:30.099 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///run/user/1000/jupyter/nbserver-3510-open.html
    Or copy and paste one of these URLs:
        http://(ip-172-31-21-255 or
       127.0.0.1):8888/?token=670dfec7558c9a84689e4c3cdbb473e158d3328a40bf6bba
```

When starting, Jupyter generates a random token. You need this token to access the web page. This is for security purposes, so no one can access the Notebook service but you.

Copy the URL from the terminal, and replace "(ip-172-31-21-255 or 127.0.0.1)" with the instance URL. You should end up with something like this:

http://ec2-18-217-172-167.us-east-2.compute.amazonaws.com:8888/?token=f04317713e74e65289fe5a43dac43d5bf164c144d05ce613

This URL consists of three parts:

- The DNS name of the instance. If you use AWS, you can get it from the AWS console or by using the AWS CLI.
- The port (8888, which is the default port for the Jupyter notebooks service).
- The token you just copied from the terminal.

After that, you should be able to see the Jupyter Notebooks service and create a new notebook (figure A.8).

Figure A.8 The Jupyter Notebook service. Now you can create a new notebook.

If you're using a remote machine, when you exit the SSH session the Jupyter Notebook service will stop working. The internal process is attached to the SSH session, and it will be terminated. To avoid this, you can run the service inside *screen*, a tool for managing multiple virtual terminals:

```
screen -R jupyter
```

This command will attempt to connect to a screen with the name "jupyter", but if no such screen exists, it will create one.

Then, inside the screen, you can type the same command for starting Jupyter Notebook:

```
jupyter notebook --ip=0.0.0.0 --no-browser
```

Check that it's working by trying to access it from your web browser. After verifying that it works, you can detach the screen by pressing Ctrl-A followed by D: first press Ctrl-A, wait a bit, and then press D (for macOS, first press Ctrl-A and then press Ctrl-D). Anything running inside the screen is not attached to the current SSH session, so when you detach the screen and exit the session, the Jupyter process will keep running.

You can now disconnect from SSH (by pressing Ctrl-D) and verify that the Jupyter URL is still working.

A.2.2 Running Jupyter on Windows

As with Python and Anaconda, if you use the Linux Subsystem for Windows to install Jupyter, the instructions for Linux should work for Windows too, with no additional changes.

However, if you didn't use the Linux Subsystem and installed Anaconda using the Windows Installer, starting the Jupyter Notebook service is different.

First, we need to open the Anaconda Navigator in the start menu. Once it's open, find "Jupyter" in the Applications tab and click "Launch" (figure A.9).

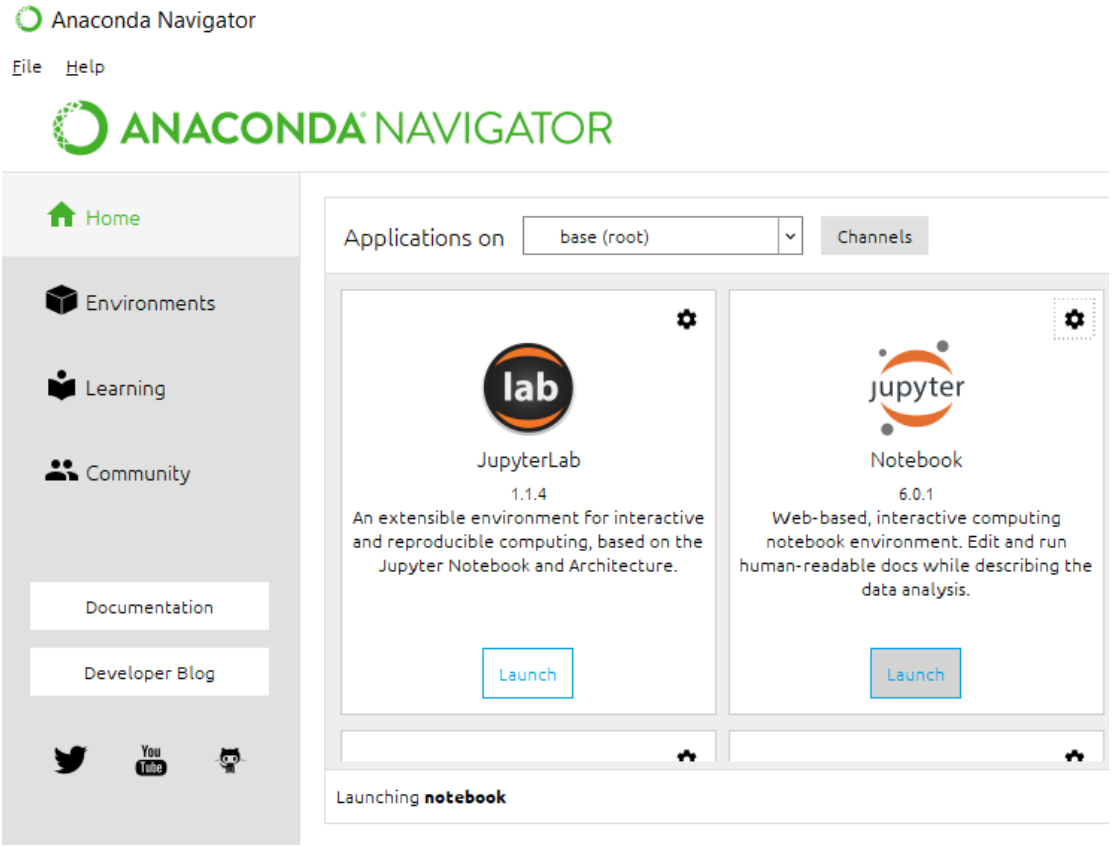


Figure A.9 To run the Jupyter Notebook service, find "Jupyter" in the applications tab and click "Launch"

After the service launches successfully, the browser with Jupyter should open automatically (figure A.10).

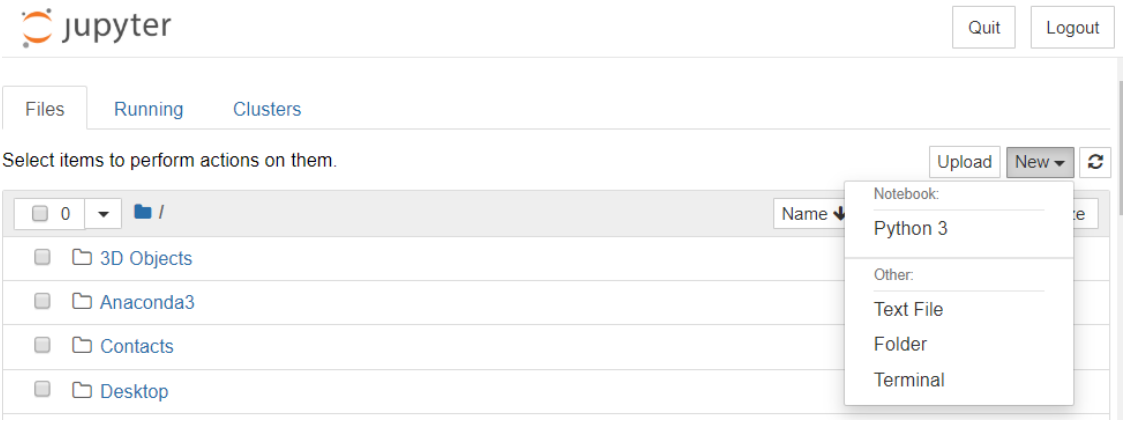


Figure A.10 The Jupyter Notebook service launched using Anaconda Navigator

A.2.3 Running Jupyter on MacOS

The instructions for Linux should also work for macOS, with no additional changes.

A.3 Installing the Kaggle CLI

The Kaggle CLI is the command-line interface for accessing the Kaggle platform, which includes data from Kaggle competitions and Kaggle datasets.

You can install it using pip:

```
pip install kaggle --upgrade
```

Then you need to configure it. First, you need to get credentials from Kaggle. For that, go to your Kaggle profile (create one if you don't have one yet), located at https://www.kaggle.com/<username>/account. The URL will be something like https://www.kaggle.com/agrigorev/account.

In the "API" section, click "Create New API Token" (figure A.11).

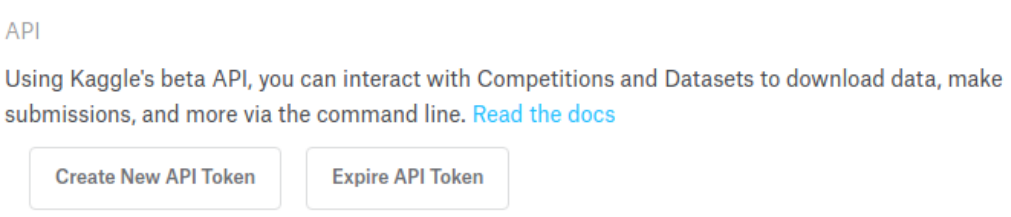


Figure A.11 To generate an API token to use from the Kaggle CLI, click "Create New API Token" on your Kaggle account page.

This will download a file called kaggle.json, which is a JSON file with two fields: `username` and `key`. If you're configuring the Kaggle CLI on the same computer you used to download the file, you should simply move this file to the location where the Kaggle CLI expects it:

```
mkdir ~/.kaggle
mv kaggle.json ~/.kaggle/kaggle.json
```

If you're configuring it on a remote machine, such as an EC2 instance, you need to copy the content of this file and paste them into the terminal. Open the file using nano (this will create the file if it doesn't exist):

```
mkdir ~/.kaggle
nano ~/.kaggle/kaggle.json
```

Paste in the content of the kaggle.json file you downloaded. Save the file by pressing Ctrl-O and exit nano by pressing Ctrl-X.

Now test that it's working by trying to list the available datasets:

```
kaggle datasets list
```

You can also test that it can download datasets, by trying the dataset from chapter 2:

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

It should download a file called cardataset.zip.

A.4 Accessing the source code

We've stored the source code for this book on GitHub, a platform for hosting source code. You can see it here: https://github.com/alexeygrigorev/ml-projects.

GitHub uses Git to manage code, so you'll need a Git client to access the code for this book.

Git comes preinstalled in all the major Linux distributions. For example, the AMI we used for creating an instance with Ubuntu on AWS already has it.

If your distribution doesn't have Git, it's easy to install it. For example, for Debian-based distributions (such as Ubuntu), you need to run the following command:

```
sudo apt-get install git
```

On macOS, to use Git you need to install Command Line Tools or, alternatively, download the installer at https://sourceforge.net/projects/git-osx-installer/.

For Windows, you can download Git at https://git-scm.com/download/win.

Once you have Git installed, you can use it to get the book's code. To access it, you need to run the following command:

```
git clone https://github.com/alexeygrigorev/ml-projects.git
```

Now you can run Jupyter Notebook:

```
cd ml-projects
```

```
jupyter notebook
```

If you don't have Git and don't want to install it, it's also possible to access the code without it. You can download the latest code in a zip archive and unpack it. On Linux, you can do that by executing these commands:

```
wget -O ml-projects.zip \
    https://github.com/alexeygrigorev/ml-projects/archive/master.zip
unzip ml-projects.zip
rm ml-projects.zip
```

You can also just use your web browser: type the URL, download the zip archive, and extract the content.

A.5 Renting a server on AWS

Using a cloud service is the easiest way of getting a remote machine that you can use for following the examples in the book.

There are quite a few options nowadays, including cloud computing providers like Amazon Web Services (AWS), Google Cloud Platform, Microsoft Azure, and Digital Ocean. Rather than having to rent a server for a long time, in the cloud you can use it for a short period and typically pay per hour, per minute, or even per second. You can select the best machine for your needs in terms of computing power (number of CPUs or GPUs) and RAM.

It's also possible to rent a dedicated server for a longer time and pay per month. If you intend to use the server for a long time — say, six months or more — renting a dedicated server will be cheaper. Hetzner.com might be a good option in this case. They also offer servers with GPUs.

To make it easier for you to set up the environment with all the required libraries for the book, we provide instructions here on setting up an EC2 (Elastic Compute Cloud) machine on AWS. EC2 is part of AWS and allows you to rent a server of any configuration for any duration of time.

> **NOTE** We're not affiliated with Amazon or AWS. We chose to use it in this book because at the time of writing it's the most commonly used cloud provider.

If you don't have an AWS account or only recently created it, you're eligible for the free tier: you have a 12-month trial period in which to check out most of the AWS products for free. We try to use the free tier whenever possible, and we will specifically mention if something isn't covered by this tier.

Note that the instructions in this section are optional, and you don't have to use AWS or any other cloud. The code should work on any Linux machine, so if you have a laptop with Linux, it should be enough to work through the book. A Mac or Windows computer should also be fine, but we haven't tested the code thoroughly on these platforms.

A.5.1 Registering on AWS

The first thing you need to do is create an account. To do this, go to https://aws.amazon.com and click the "Create an AWS Account" button (see figure A.12).

Figure A.12 To create an account, click "Create an AWS Account" on the main AWS page.

> **NOTE** This chapter was written in October 2019 and the screenshots were taken at that time. Please be aware that content on the AWS web site and the appearance of the management console could change.

Follow the instructions and fill in the required details. It should be a straightforward process, similar to the process of registering on any website.

> **NOTE** Please be aware that AWS will ask you to provide the details of a bank card during the registration process.

Once you've completed the registration and verified your account, you should see the main page — the AWS Management Console (figure A.13).

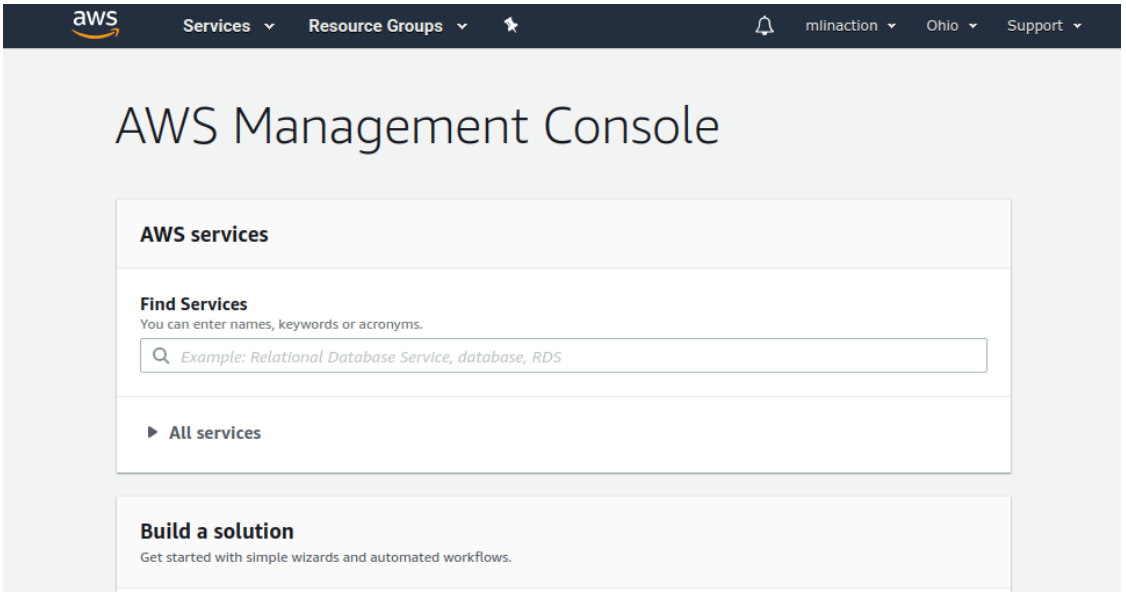


Figure A.13 The AWS Management Console is the starting page for AWS.

Congratulations! You've just created a root account. However, it's not advised to use the root account for anything: it has very broad permissions that allow you to do anything and everything on your AWS accunt. Typically, you use the root account to create less powerful accounts and then use them for your day-to-day tasks.

To create such an account, type "IAM" in the "Find Services" box and click on that item in the drop-down list. Select "Users" in the menu on the left, and click "Add User" (see figure A.14).

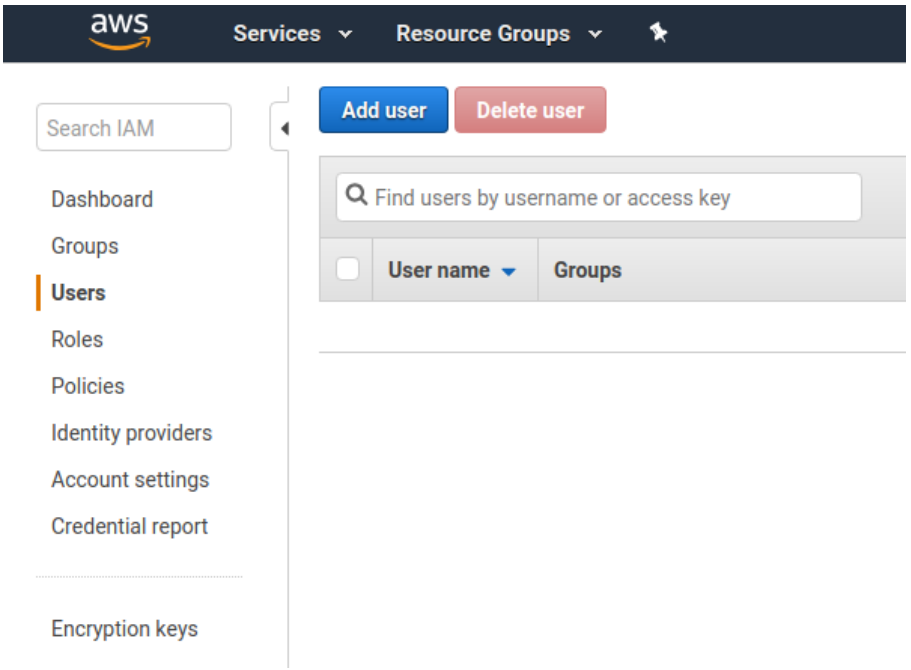


Figure A.14 Adding a user in the AWS Identity and Access Management (IAM) service.

Now you just need to follow the instructions and answer the questions. At some point, it will ask about an access type: you'll need to select both "Programmatic Access" and "AWS Management Console Access" (see figure A.15). We will use both the command-line interface (CLI) and the web interface for working with AWS.

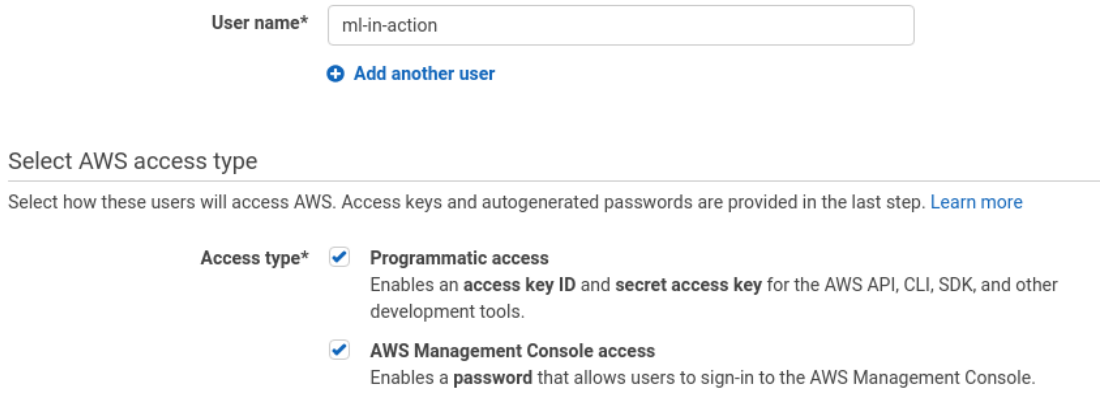


Figure A.15 We will use both the web interface and the command-line interface for working with AWS, so you need to select both access types.

In the "Set Permissions" step, you specify what this new user will be able to do. You want the user to have full privileges, so select "Attach Existing Policies Directly" at the top and choose "AdministratorAccess" in the list of policies (see figure A.16).

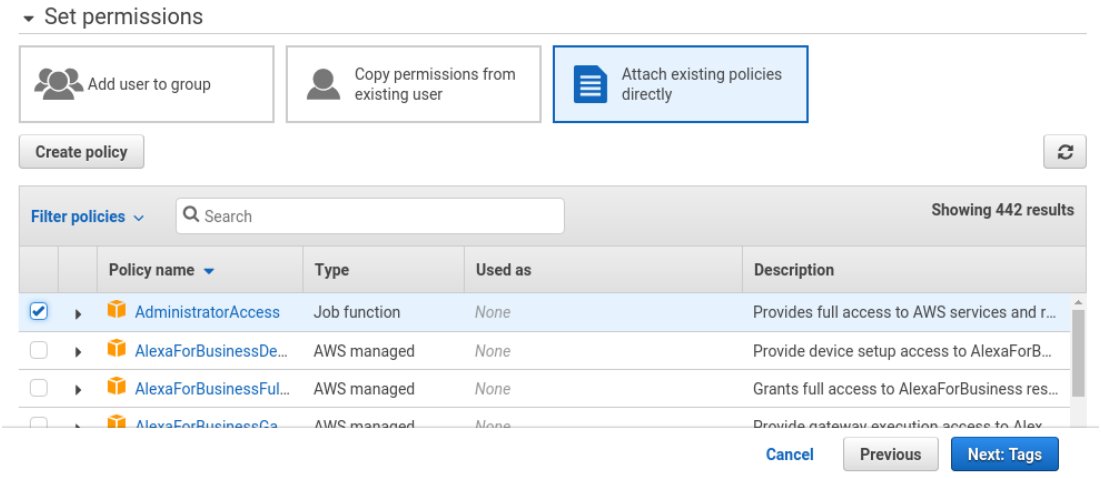


Figure A.16 Select the "AdministratorAccess" policy to enable the new user to access everything on AWS.

As the next step, the system will ask you about tags — you can safely ignore these for now. Tags are needed for companies where multiple people work on the same AWS account, mostly

for expense management purposes, so they shouldn't be a concern for the projects you'll do in this book.

At the end, when you've successfully created the new user, the wizard will suggest that you download the credentials (figure A.17). Download them and keep them safe: you'll use these later for programmatic access.

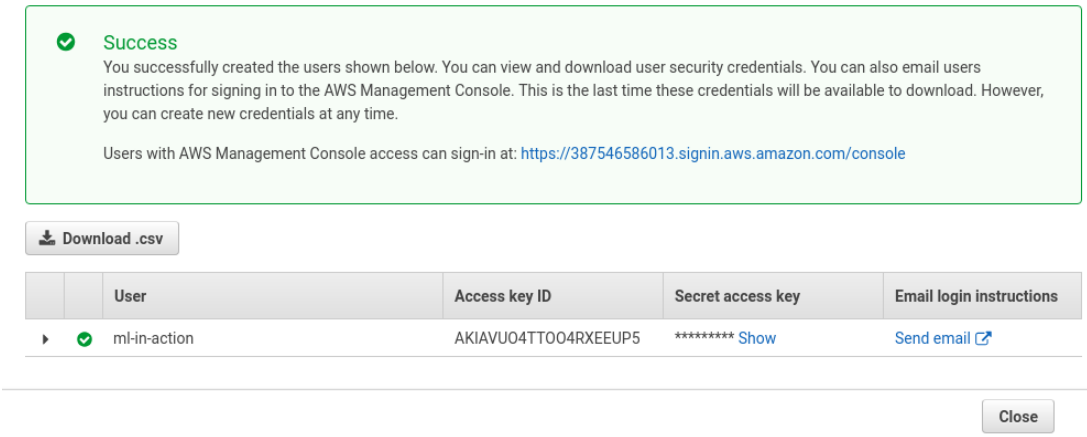


Figure A.17 The details for the newly created user. You can see the sign-in URL and download the credentials for programmatic access.

To access the management console, you can use the link AWS has generated for you. It appears in the "Success" box and follows this pattern:

```
https://<accountid>.signin.aws.amazon.com/console
```

It might be a good idea to bookmark this link. Once AWS has validated the account (which can take a little while), you can use it to log in: simply provide the username and password you specified when creating the user.

You can now start using the services of AWS. Most importantly, you can create an EC2 machine.

A.5.2 Accessing billing information

When using a cloud service provider, you are typically charged per second: for every second you use a particular AWS service, you pay a pre-defined rate. At the end of each month you get a bill, which is typically processed automatically: the money is withdrawn from the bank card you linked to the AWS account.

> **IMPORTANT** Even though we use the free tier to follow most of the examples in the book, you should periodically check the billing page to make sure you're not accidentally using billable services.

To understand how much you will need to pay at the end of the month, you can access the billing page of AWS.

If you use the root account (the account you created first), simply type "billing" on the homepage of the AWS console to navigate to the billing page (figure A.18).

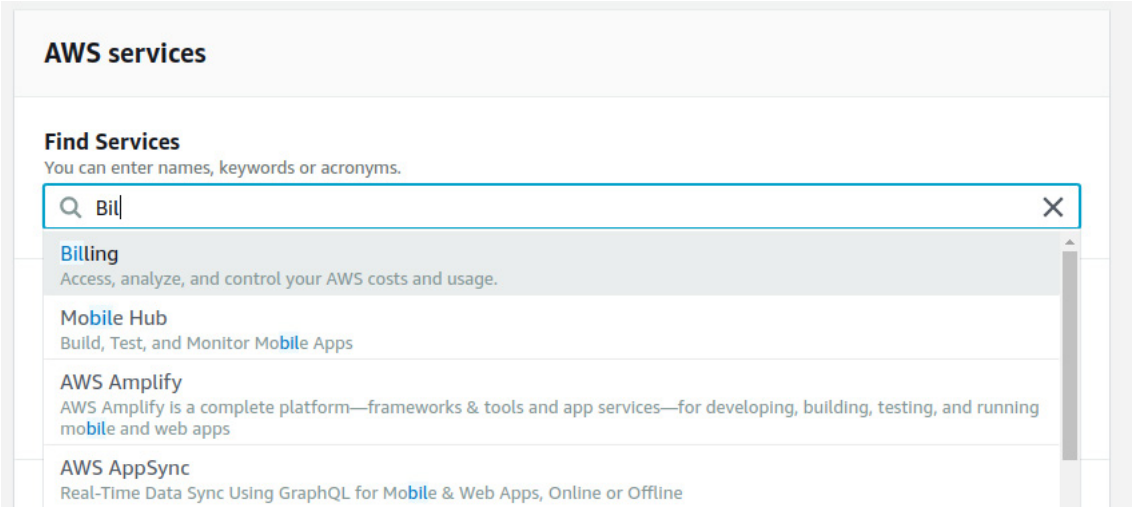


Figure A.18 To get to the billing page, type in "Billing" in the quick access search box.

If you try to access the same page from the user account (or "IAM user" — the one we created after creating the root account), you will notice that it's not allowed. To fix that, you need to

- Allow the billing page to be accessed by all IAM user, and
- Give the AMI user permissions to access the billing page.

Allowing all the IAM users access the billing page is simple: go to "My Account" (figure A.19 A), go to the "IAM User and Role Access to Billing Information" section and click "Edit" (figure A.19 B), and then select the "Activate IAM Access" option and click "Update" (figure A.19 C).

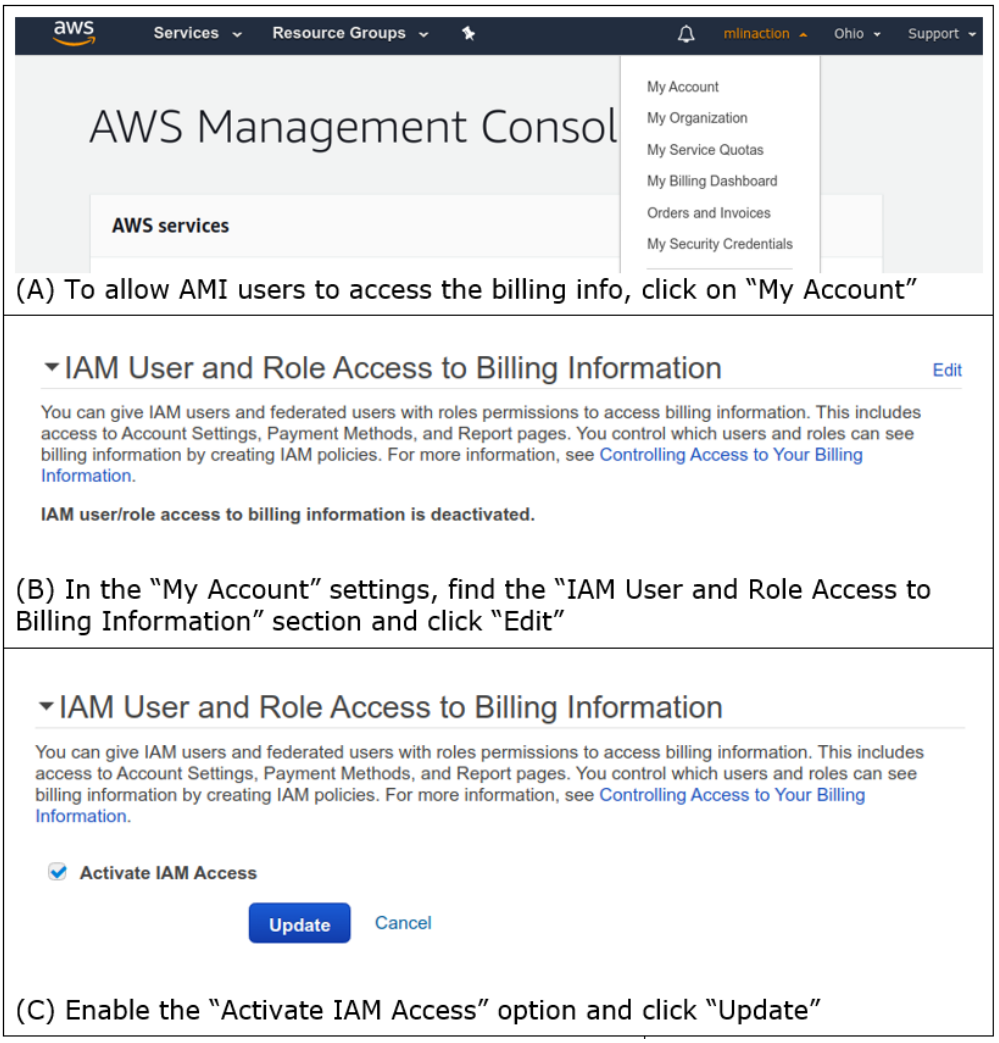


Figure A.19 Enabling access to billing information to IAM users

After that, go to the "IAM" service and find the IAM user we previously created and click on it. Next, click on the "Add permissions" button (figure A.20).

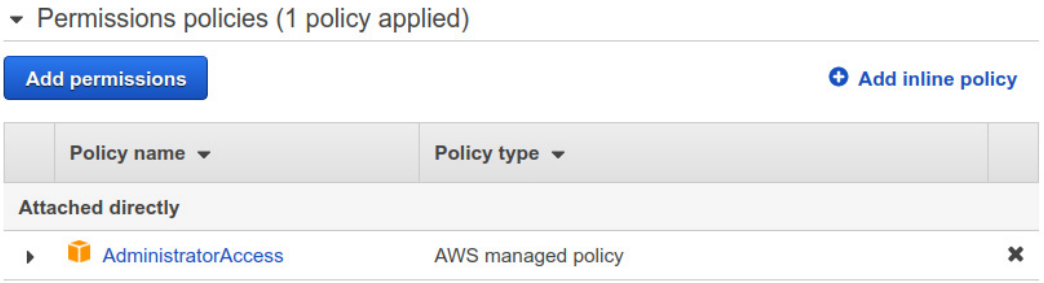


Figure A.20 To allow the IAM user access the billing information, we need to add special permissions for that. To do it, click on the "Add permissions" button.

Then attach the existing "Billing" policy to the user (figure A.21).

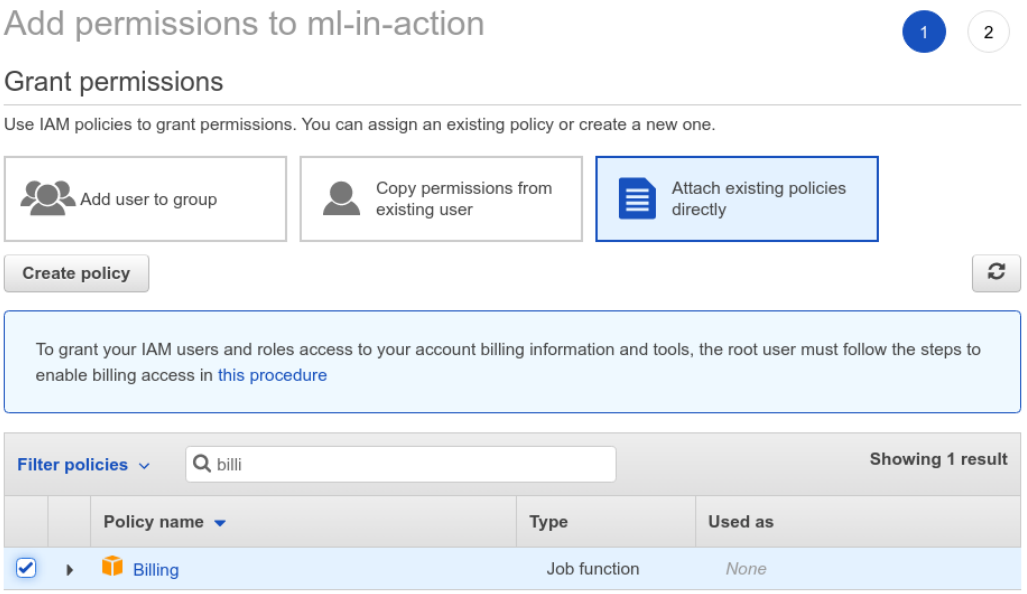


Figure A.21 After clicking on the "Add permissions" button, select the "Attach existing policies directly" option and select "Billing" in the list.

After that, the IAM user should be able to access the billing information page.

A.5.3 Creating an EC2 instance

EC2 is a service for renting a machine from AWS. You can use it to create a Linux machine to use for the projects in this book. To do this, first go to the EC2 page in AWS. The easiest way to do this is by typing "EC2" in the "Find Services" box on the home page of the AWS Management Console; select "EC2" from the drop-down list and press Enter (figure A.22).

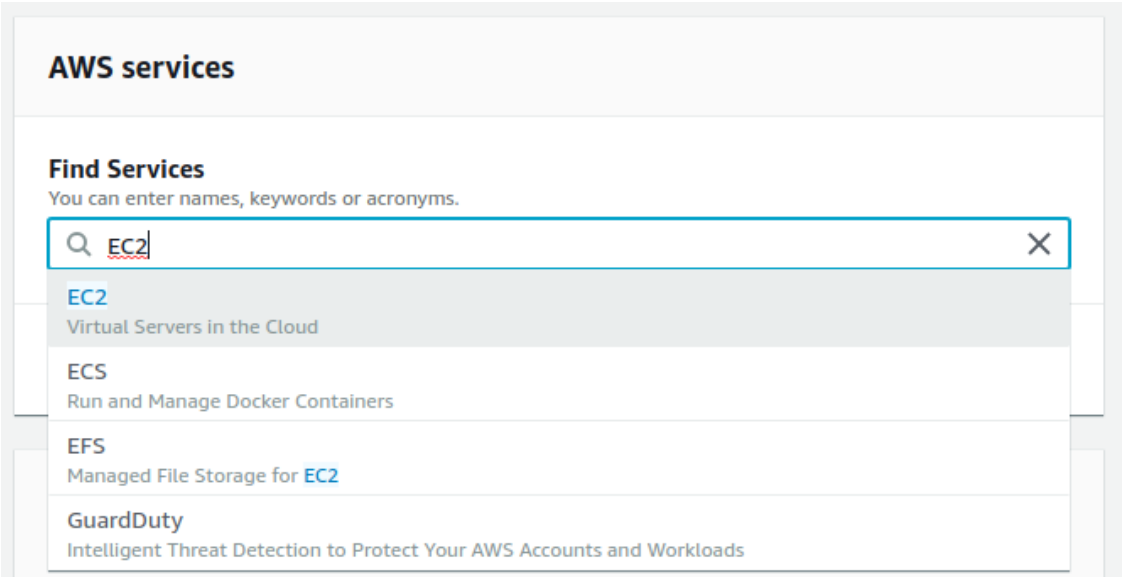


Figure A.22 To go to the EC2 service's page, type EC2 in the "Find Services" box on the main page of the AWS Management Console and press Enter.

On the EC2 page, choose "Instances" from the menu on the left and then click "Launch Instance" (figure A.23).

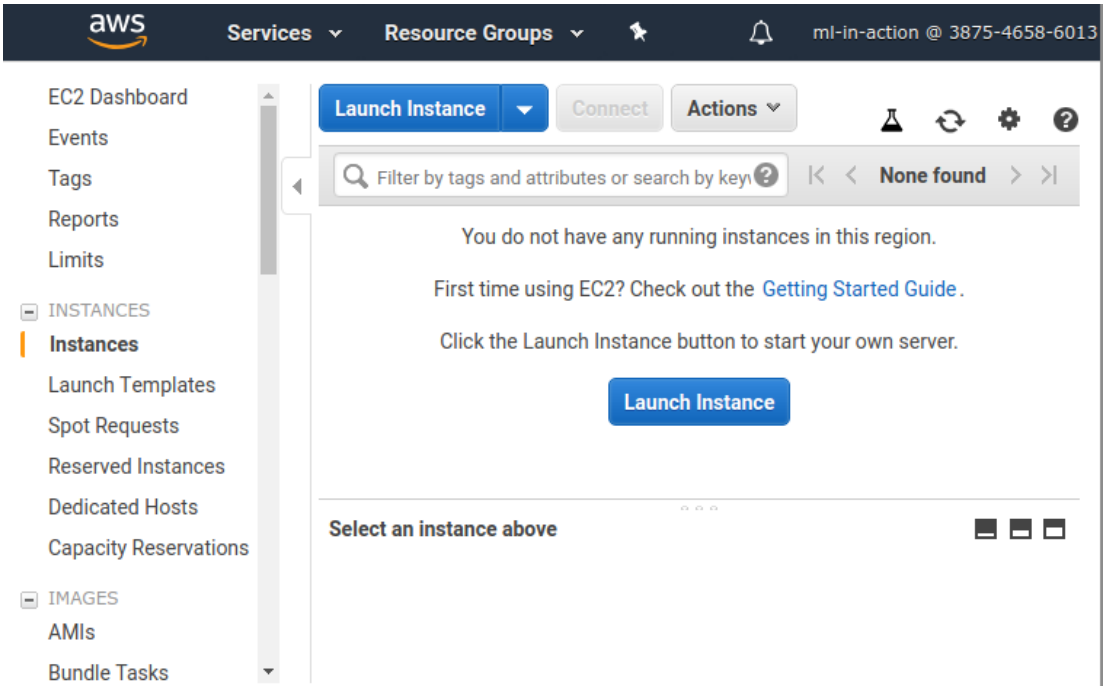


Figure A.23 To create an EC2 instance, select “Instances” in the menu on the left and click “Launch Instance”.

This brings you to a six-step form.

The first step is to specify the AMI (Amazon Machine Image) you’ll use for the instance. We recommend Ubuntu: it’s one of the most popular Linux distributions and we used it for all the examples in this book. Other images should also work fine, but we haven’t tested them.

At the time of writing, Ubuntu Server 18.04 LTS is available (figure A.24), so use that one: find it in the list and then click “Select”.

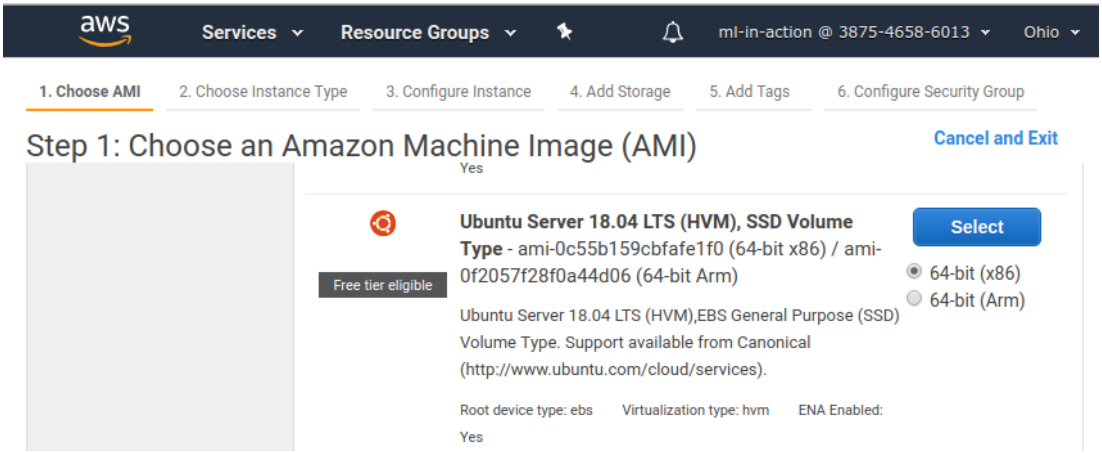


Figure A.24 Your instance will be based on Ubuntu Server 18.04 LTS.

You should take note of the AMI's ID: it this example it's "ami-0c55b159cbfafe1f0", but it might be different for you, depending on your AWS region and version of Ubuntu. You'll need this later for creating EC2 instances programmatically.

Also note that this AMI is "Free tier eligible," which means that if you use the free tier for testing AWS, you won't be charged for using this AMI.

After that you need to select the instance type. There are many options, with different numbers of CPU cores and different amounts of RAM. If you want to stay within the free tier, select "t2.micro" (figure A.25). It's a rather small machine: it has only 1 CPU and 1 GB RAM. Of course, it's not the best instance in terms of computing power, but it should be enough for many projects in this book.

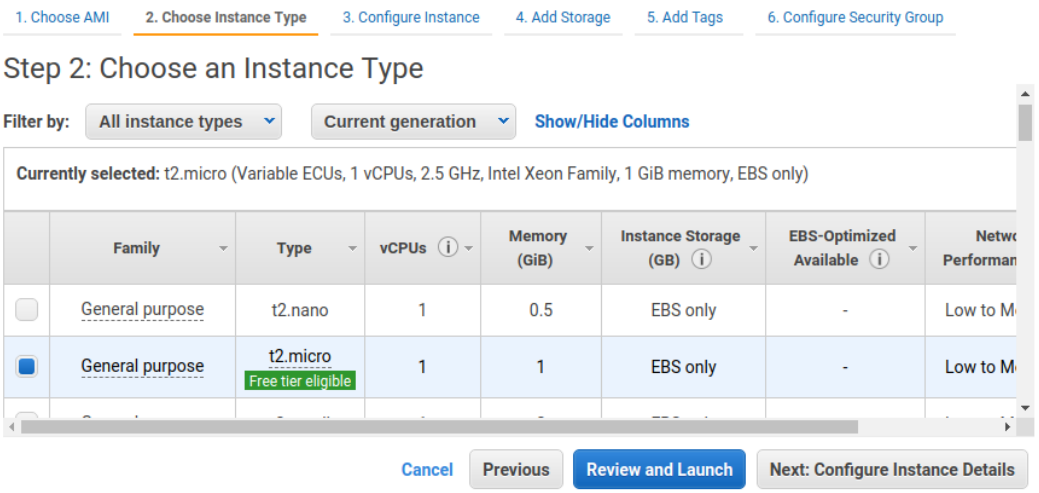


Figure A.25 The "t2.micro" is a rather small instance with only 1 CPU and 1 GB RAM, but it can be used for free.

The next step is where you configure the instance details. You don't need to change anything here and can simply go on to the next step, adding storage (figure A.26). Here, you specify how much space you need on the instance. The default suggestion is 8 GB, which is good enough for most of the projects we'll do, so you can simply click "Next: Add Tags".

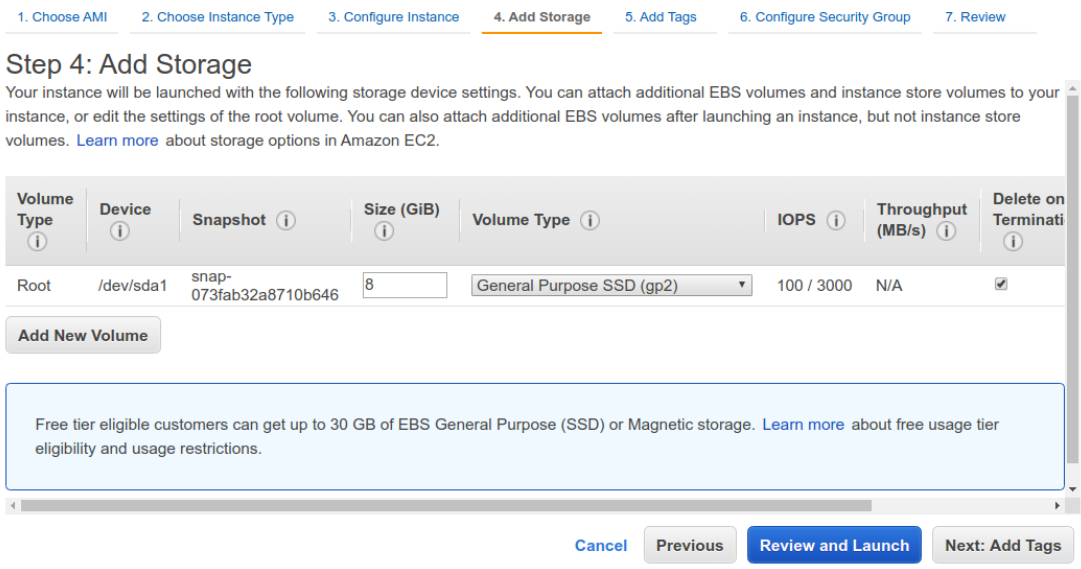


Figure A.26 The fourth step of creating an EC2 instance in AWS. You can stick with the default of 8 GB.

In the next step, you add tags to your new instance. The only tag you should add is "Name", which allows you to give an instance a human-readable name. Add the key "Name" and the value "ml-in-action-instance" (or any other name you prefer), as seen in figure A.27.

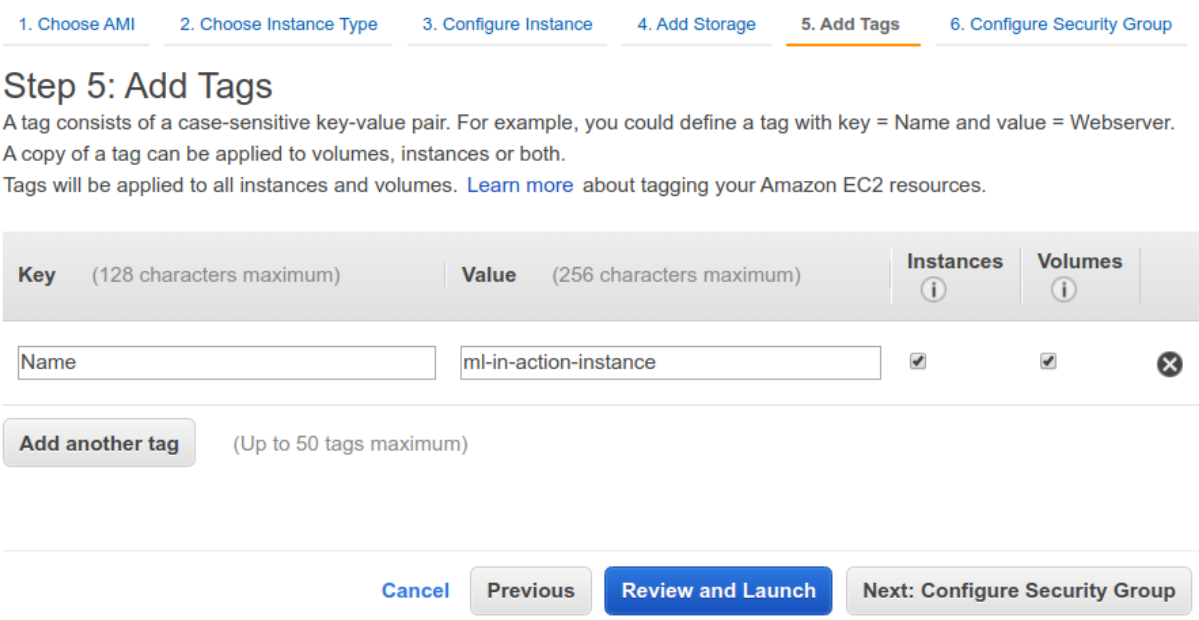


Figure A.27 The only tag you may want to specify in step 5 is "Name": it allows you to give a human-readable name to the instance.

The next step is quite an important one: here you choose the security group. This allows you to configure the network firewall and specify how the instance can be accessed and which ports are open. You'll want to host Jupyter Notebook on the instance, so you need to make sure its port is open and you can log in to the remote machine.

Because you don't yet have any security groups in your AWS account, you'll need to create a new one now: choose "Create a New Security Group" and give it the name "jupyter" (figure A.28). You'll want to use SSH to connect to the instance from your computers, so you need to make sure SSH connections are allowed. To enable this, select SSH in the "Type" drop-down list in the first row.

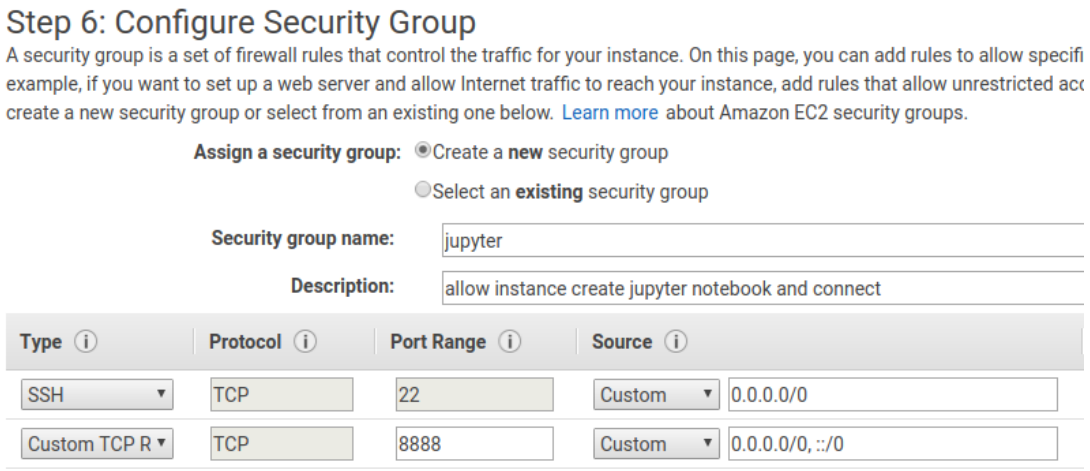


Figure A.28 Creating a security group for running Jupyter Notebook on EC2 instances

Typically the Jupyter Notebook service runs on port 8888, so you need to add a custom TCP rule saying that port 8888 can be accessed from anywhere on the internet (figure A.29).

When you do this, you may see a warning telling you that this might not be safe (figure A.14). It's not a problem for us: we are not running anything critical on the instances. Implementing proper security is not trivial and is out of scope for this book.

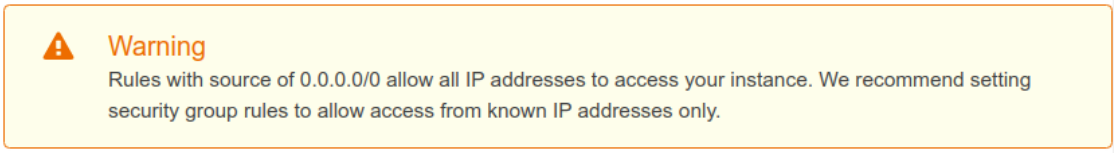


Figure A.29 AWS warns us that the rules we added are not strict. For our case it's not a problem and we can safely ignore the warning.

The next time you create an instance, you'll be able to reuse this security group instead of creating a new one: choose "Select an Existing Security Group" and select it from the list (figure A.30).

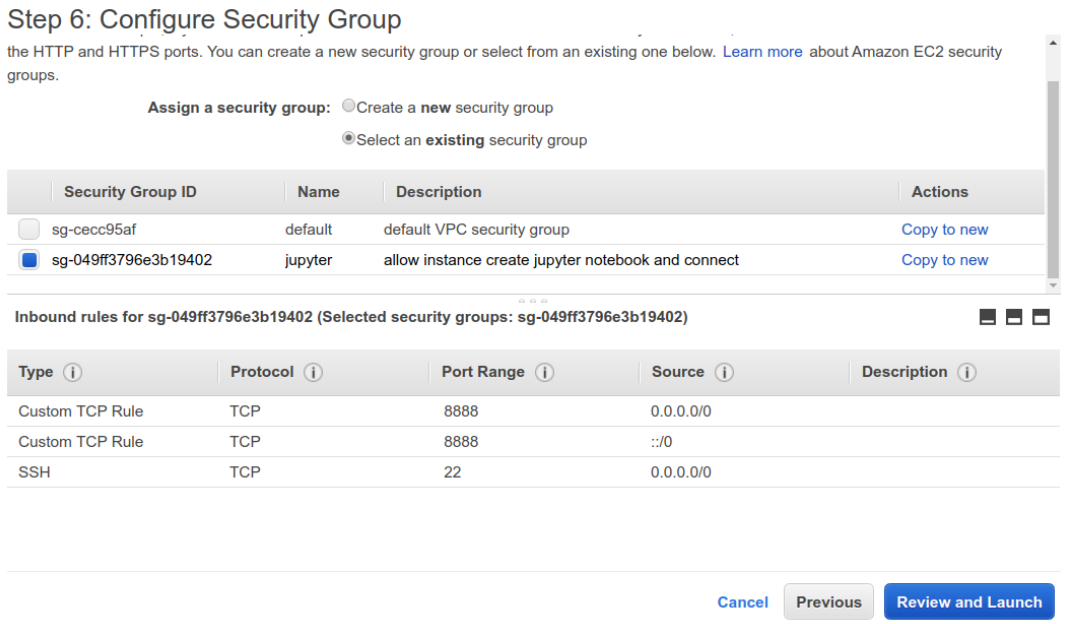


Figure A.30 When creating an instance, it's also possible to assign an existing security group to the instance.

Configuring the security group is the last step. Verify that everything is fine, and click "Review and Launch".

AWS won't let you launch the instance yet: you still need to configure the SSH keys for logging into the instance. Because your AWS account is still fresh and doesn't have keys yet, you need to create a new key pair. Choose "Create a New Key Pair" from the drop-down list and give it the name "jupyter" (figure A.31).

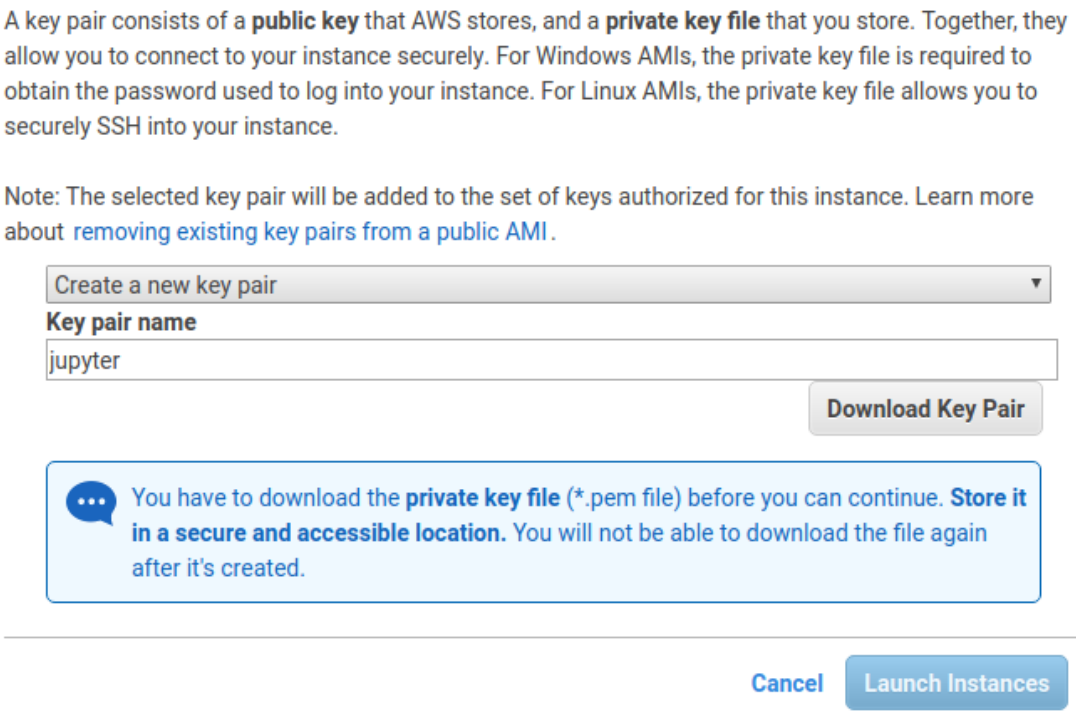


Figure A.31 To be able to use SSH to log in to the instance, you need to create a key pair.

Click "Download Key Pair" and save the file somewhere on your computer. Make sure you can access this file later: it's important for being able to connect to the instance.

The next time you create an instance, you can reuse this key. Select "Choose an Existing Key Pair" in the first drop-down list, choose the key you want to use and click the checkbox to confirm that you still have the key (figure A.32).

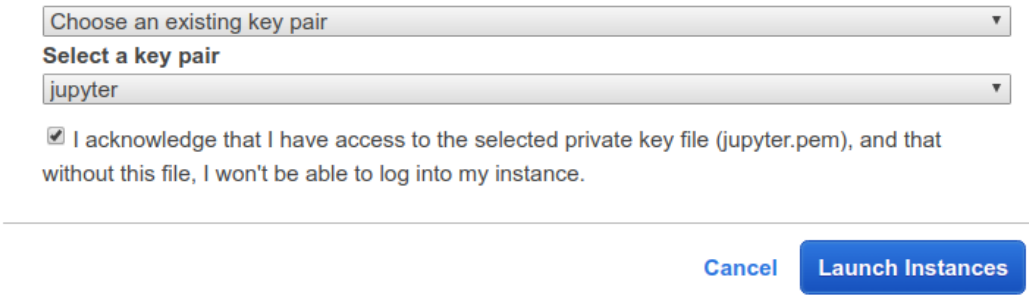


Figure A.32 You can also use an existing key when creating an instance.

Now you can launch the instance by clicking "Launch Instances". You should see a confirmation that everything is good and the instance is launching (figure A.33).

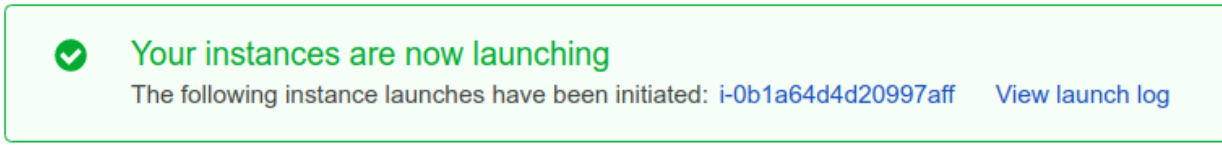


Figure A.33 AWS tells us that everything went well and now the instance is launching.

In this message, you can see the ID of the instance. In our case, it's "i-0b1a64d4d20997aff". You can click on it now to see the details of the instance (figure A.34). Because you want to use SSH to connect to your instance, you need to get the public DNS name to do it. You can find this on the "Description" tab.

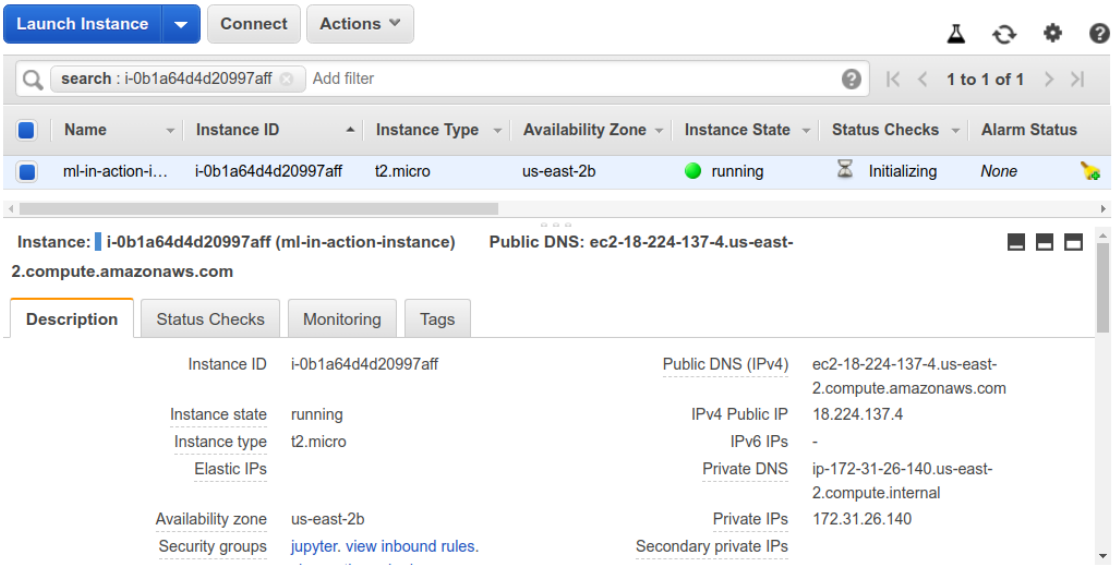


Figure A.34 The details of the newly created instance. To use SSH to connect to it, you need the public DNS name.

A.5.4 Connecting to the instance

In the previous section you created an instance on EC2. Now you'd like to log in to this instance to install all the required software. You will use SSH for this.

CONNECTING TO THE INSTANCE ON LINUX

You already know the public DNS name of your instance. In our example, it's "ec2-18-191-156-172.us-east-2.compute.amazonaws.com". In your case the name will be different: the first part of the name ("ec2-18-191-156-172") depends on the IP that the instance gets and the second ("us-east-2") on the region where it's running. To use SSH to enter the instance, you will need this name.

When using the key you downloaded from AWS for the first time, you need to make sure the permissions on the file are set correctly. Execute this command:

```
chmod 400 jupyter.pem
```

Now you can use the key to log in to the instance:

```
ssh -i "jupyter.pem" \
    ubuntu@ec2-18-191-156-172.us-east-2.compute.amazonaws.com
```

Of course, you should replace the DNS name shown here with the one you copied from the instance description.

Before allowing you to enter the machine, the SSH client will ask you to confirm that you trust the remote instance:

```
The authenticity of host 'ec2-18-191-156-172.us-east-2.compute.amazonaws.com
       (18.191.156.172)' can't be established.
ECDSA key fingerprint is SHA256:S5doTJOGwXVF3i1IFjB10RuHufaVSe+EDqKbGpIN0wI.
Are you sure you want to continue connecting (yes/no)?
```

Type "yes" to confirm.

Now you should be able to log in to the instance and see the welcome message (figure A.35).

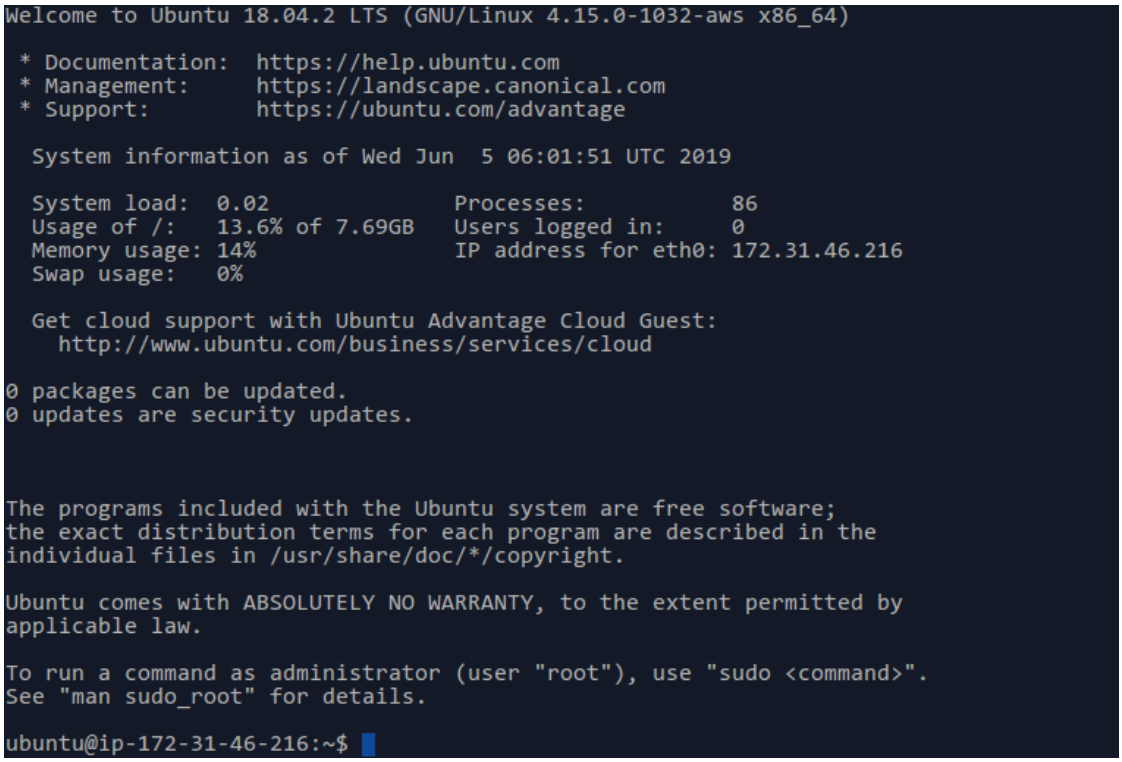


Figure A.35 After successfully logging into the EC2 instance, you should see the welcome message.

Now it's possible to do anything you want with the machine.

CONNECTING TO THE INSTANCE ON WINDOWS

Using the Linux subsystem on Windows is the easiest way for connecting to the EC2 instance: you can use ssh there and follow the same instructions as for Linux.

CONNECTING TO THE INSTANCE ON MACOS

SSH is built in on macOS, so the steps for Linux should work on a Mac.

A.5.5 Shutting down the instance

After you've finished working with the instance, you should turn it off.

> **IMPORTANT** It's very important to turn off the instance after the work is finished. For each second you use the instance, you get billed, even if you no longer need the machine and it's idle. That doesn't apply in the first

12 months of using AWS if the requested instance is free-tier eligible, but nonetheless, it's good to develop the habit of periodically checking your account status and disabling unneeded services.

You can do this from the terminal:

```
sudo shutdown now
```

It's also possible to do it from the web interface: select the instance you want to turn off, go to "Actions", and select "Instance State" > "Stop" (figure A.36).

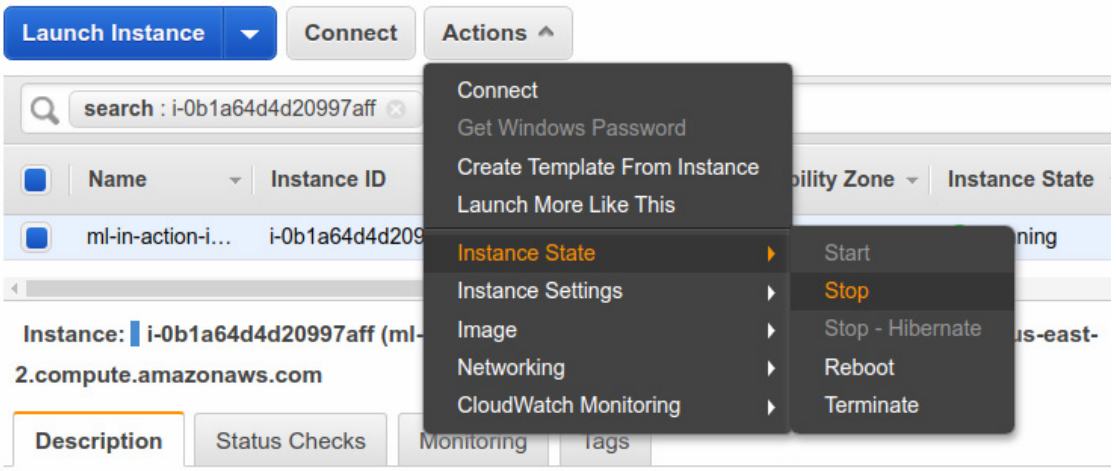


Figure A.36 Stopping the instance from the AWS console

Once the instance has been stopped, you can start it again by choosing "Start" from the same submenu. It's also possible to completely remove the instance: for this you need to use the "Terminate" option.

A.5.6 Creating an EC2 instance with the AWS CLI

Using the AWS console is a convenient way of creating EC2 instances, and it's a good option if you need to do this only occasionally. However, if you need to do it often, it may become tedious. There's a way to automate the manual work: use the AWS CLI, which is the command-line interface for AWS.

To use the CLI, you need to have Python. If you use Linux or macOS, you should already have a Python distribution built in. Alternatively, you can install Anaconda using the instructions in the next section.

Just having Python is not enough: you also need to install the AWS CLI itself. You can do this by running the following command in the terminal:

```
pip install awscli
```

If you already have it, it's a good idea to update it:

```
pip install -U awscli
```

After the installation finishes you need to configure the tool, specifying the access token and secret you downloaded earlier when creating a user.

One way to do this is to use the `configure` command:

```
aws configure
```

It will ask you for the keys:

```
$ aws configure
AWS Access Key ID [None]: <ENTER_ACCESS_KEY>
AWS Secret Access Key [None]: <ENTER_SECRET_KEY>
Default region name [None]: us-east-2
Default output format [None]:
```

The region name used here is "us-east-2", which is located in Ohio.

When you're finished configuring the tool, verify that it works. You can ask the CLI to return your identity, which should match the details of your user:

```
$ aws sts get-caller-identity
{
    "UserId": "AIDAVUO4TTOO55WN6WHZ4",
    "Account": "387546586013",
    "Arn": "arn:aws:iam::387546586013:user/ml-in-action"
}
```

Another tool you should install to make interaction with AWS easier is `jq`, a command-line library for parsing JSON data. You will use it to parse the responses from AWS and extract the parts you need.

To install `jq` on Ubuntu Linux, type the following command in the terminal:

```
sudo apt install jq
```

`jq` is also available for other distributions of Linux, as well as macOS and Windows. You can check its official page (https://stedolan.github.io/jq/download/) for more details.

Now you have all the tools you need to request an EC2 instance programmatically.

To request an instance, you will use the `aws ec2 run-instances` command. It has a few parameters you have to specify. To see the full list of parameters, invoke it with `help` at the end:

```
aws ec2 run-instances help
```

This brings up the manual page, shown in figure A.37.

```
NAME
       run-instances -

DESCRIPTION
       Launches  the  specified number of instances using an AMI for which you
       have permissions.

       You can specify a number of options, or leave the default options.  The
       following rules apply:

       o [EC2-VPC]  If you don't specify a subnet ID, we choose a default sub-
         net from your default VPC for you. If you don't have a  default  VPC,
         you must specify a subnet ID in the request.

       o [EC2-Classic]  If  don't  specify an Availability Zone, we choose one
         for you.

       o Some instance types must be launched into a VPC. If you do not have a
         default VPC, or if you do not specify a subnet ID, the request fails.
         For more information, see Instance Types Available Only in a VPC .

       o [EC2-VPC] All instances have a network interface with a primary  pri-
         vate  IPv4  address. If you don't specify this address, we choose one
         from the IPv4 range of your subnet.

       o Not all instance types support IPv6 addresses. For more  information,
         see Instance Types .

       o If you don't specify a security group ID, we use the default security
:
```

Figure A.37 The `run-instances` manual page. You can see the manual page for any AWS CLI command by simply adding `help` at the end.

After inspecting the manual page, we see that in order to create an instance, you need to specify a few things:

- Instance type — This specifies the number of CPUs and the amount of RAM on the instance. You used t2.micro previously to stay within the free tier, so you will do the same here.
- Number of instances — In this case, you just want to create one instance.
- AMI — The ID of the image to use for creating the instance. Previously we used "ami-0c55b159cbfafe1f0" with Ubuntu Linux AMI, so we'll use the same one here.
- Key name — The name of the key you'll use to SSH to the instance. We'll use "jupyter", the key created previously.
- Security group ID — The security group specifies which ports are open. We'll use the one we created earlier.
- Subnet ID — The ID of the network where the instance will run. We'll need to use the AWS CLI to get this.

You can get all this information using the AWS console, but this time we'll use the CLI. To get the list of available security groups, use the `describe-security-groups` command:

```
aws ec2 describe-security-groups
```

The response will look similar to this:

```
{
    "SecurityGroups": [
        {
            "Description": "allow instance to create Jupyter notebook and connect",
            "GroupName": "jupyter",
            "IpPermissions": [...],
            "OwnerId": "387546586013",
            "GroupId": "sg-049ff3796e3b19402",
            ...
        },
        ...
    ]
}
```

The result contains a list of all available security groups for the account in the region specified during the configuration (Ohio — which is "us-east-2"). In this case it contains the group named "jupyter" which you created using the web interface. What you need from this is the group ID, which is stored in the `GroupId` field.

You can simply copy this ID and then use it later when requesting an instance. Alternatively, you can use `jq` to extract the content of the `GroupId` field:

```
aws ec2 describe-security-groups \
    | jq '.SecurityGroups[] | select(.GroupName=="jupyter") | .GroupId' -r
```

The `jq` query syntax might be a bit confusing. The command performs these actions:

1. Get the content of the `SecurityGroups` field, which is an array.
2. Select all the groups that have the name "jupyter". In this case there's just one group.
3. Get the `GroupId` of that group.

You can read more about the syntax of `jq` in the official documentation (https://stedolan.github.io/jq/manual/).

When you run this query, you get the group ID:

```
sg-049ff3796e3b19402
```

The next parameter you need to fill in is the subnet ID. There are several of these, but for our purposes it doesn't matter which one you choose, so you can just pick the first one. You can do this with two commands piped together:

```
aws ec2 describe-subnets | jq ".Subnets[0].SubnetId" -r
```

The first command asks AWS to return all the available subnets (in the Ohio region), and the second one selects the first one and gets the content of the `SubnetId` field. When you run this, you get the ID:

```
subnet-8eed95f4
```

Now you have all the information required to create an instance programmatically with the AWS CLI. For that you will use the `run-instances` command. Save the response from AWS to the variable `EC2_RESPONSE` — you'll need to parse this response to extract some information about the created instance:

```
EC2_RESPONSE=$(aws ec2 run-instances \
    --image-id ami-0c55b159cbfafe1f0 \
    --count 1 \
    --instance-type t2.micro \
    --key-name jupyter \
    --security-group-ids sg-049ff3796e3b19402  \
    --subnet-id subnet-8eed95f4)
```

To log in to the instance using SSH you need to know the public DNS name, and to get the DSN name you need to know the ID of the instance. You can get the ID from the response, using `jq` to parse it:

```
INSTANCE_ID=$(echo $EC2_RESPONSE | jq ".Instances[0].InstanceId" -r)
echo ${INSTANCE_ID}
```

When you execute this in the terminal, you get the instance ID:

```
i-0ae9e52cd95f890db
```

You can use this to get the DNS name, with another command: `describe-instances`. This command takes in a list of instances and returns information about them, just like the Description tab in the AWS Console:

```
INSTANCE_DESC=$(aws ec2 describe-instances --instance-ids "${INSTANCE_ID}")
INSTANCE_DNS=$(echo ${INSTANCE_DESC} \
    | jq ".Reservations[0].Instances[0].PublicDnsName" -r)
echo ${INSTANCE_DNS}
```

When you execute it, you get the name:

```
ec2-18-217-172-167.us-east-2.compute.amazonaws.com
```

You can now log in to the instance using this name:

```
ssh -o "StrictHostKeyChecking no" \
    -i "jupyter.pem" \
    ubuntu@${INSTANCE_DNS}
```

You should see the welcome message and be able to execute commands on the instance.

To terminate the instance, you can also use the AWS CLI:

```
aws ec2 terminate-instances --instance-ids ${INSTANCE_ID}
```

A.6 Summary

- The best way of getting Python with most of the required libraries is to use Anaconda.
- The source code from the book can be accessed at https://github.com/alexeygrigorev/ml-projects.
- AWS EC2 provides an easy way of renting a server.
- Shutting down the EC2 instance after finishing the work is important to avoid unnecessary bills.
- When we need to request EC2 instances often, instead of doing it manually using the web console, we can automate it with AWS CLI.