Author Picks

FREE

# Exploring Data with Python

Chapters selected by Naomi Ceder

manning

# *Exploring Data With Python*

Selected by Naomi Ceder

**Manning Author Picks**

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

For more information, please contact

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



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

Cover designer: Leslie Haimes

ISBN: 9781617296048
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 23 22 21 20 19 18

# *contents*

# *about the authors*

**Davy Cielen**, **Arno D. B. Meysman**, and **Mohamed Ali** are the founders and managing partners of Optimately and Maiton, where they focus on developing data science projects and solutions in various sectors. Together, they wrote *Introduce Data Science*.

**Naomi Ceder** is chair of the Python Software Foundation and author of *The Quick Python Book, Third Edition*. She has been learning, using, and teaching Python since 2001. Naomi Ceder is the originator of the PyCon and PyCon UK poster sessions and the education summit.

**Henrik Brink**, **Joseph Richards**, and **Mark Fetherolf** are experienced data scientists who work with machine learning daily. They are the authors of *Real-World Machine Learning*.

# *introduction*

We may have always had data, but these days it seems we have never had quite so much data, nor so much demand for processes (and people) to put it to work. Yet in spite of the exploding interest in data engineering, data science, and machine learning it's not always easy to know where to start and how to choose among all of the languages, tools, and technologies currently available. And even once those decisions are made, finding real-world explanations and examples to use as guides in doing useful work with data are all too often lacking.

Fortunately, as the field evolves some popular (with good reason) choices and standards are appearing. Python, along with pandas, numpy, scikit-learn, and a whole ecosystem of data science tools, is increasingly being adopted as the language of choice for data engineering, data science, and machine learning. While it seems new approaches and processes for extracting meaning from data emerge every day, it's also true that general outlines of how one gets, cleans, and deals with data are clearer than they were a few years ago.

This sampler brings to together chapters from three Manning books to address these issues. First, we have a thorough discussion of the data science process from *Introducing Data Science*, by Davy Cielen, Arno D. B. Meysman, and Mohamed Ali, which lays out some of the considerations in starting a data science process and the elements that make up a successful project. Moving on from that base, I've selected two chapters from my book, *The Quick Python Book, 3rd Edition*, which focus on the using the Python language to handle data. While my book covers the range of Python from the basics through advanced features, the chapters included here cover ways to use Python for processing data files and cleaning data, as well as how to use Python for exploring data. Finally, I've chosen a chapter from *Real-world Machine Learning* by Henrik Brink, Joseph W. Richards, and Mark Fetherolf for some practical demonstrations of modelling and prediction with classification and regression.

Getting started in the wide and complex world of data engineering and data science is challenging. This collection of chapters comes to the rescue with an understanding of the process combined with practical tips on using Python and real-world illustrations.

# *The data science process*

There are a lot of factors to consider in extracting meaningful insights from data. Among other things, you need to know what sorts of questions you hope to answer, how you are going to go about it, what resources and how much time you'll need, and how you will measure the success of your project. Once you have answered those questions, you can consider what data you need, as well as where and how you'll get that data and what sort of preparation and cleaning it will need. Then after exploring the data comes the actual data modelling, arguably the "science" part of "data science." Finally, you're likely to present your results and possibly productionize your process.

Being able to think about data science with a framework like the above increases your chances of getting worthwhile results from the time and effort you spend on the project. This chapter, "The data science process" from *Introducing Data Science*, by Davy Cielen, Arno D. B. Meysman, and Mohamed Ali, lays out the steps in a mature data science process. While you don't need to be strictly bound by these steps, and may spend less time or even ignore some of them, depending on you project, this framework will help keep your project on track.

Chapter 2 from *Introducing Data Science*
by Davy Cielen, Arno D. B. Meysman,
and Mohamed Ali.

# *The data science process*

***This chapter covers***

- Understanding the flow of a data science process
- Discussing the steps in a data science process

The goal of this chapter is to give an overview of the data science process without diving into big data yet. You'll learn how to work with big data sets, streaming data, and text data in subsequent chapters.

## 2.1 Overview of the data science process

Following a structured approach to data science helps you to maximize your chances of success in a data science project at the lowest cost. It also makes it possible to take up a project as a team, with each team member focusing on what they do best. Take care, however: this approach may not be suitable for every type of project or be the only way to do good data science.

The typical data science process consists of six steps through which you'll iterate, as shown in figure 2.1.

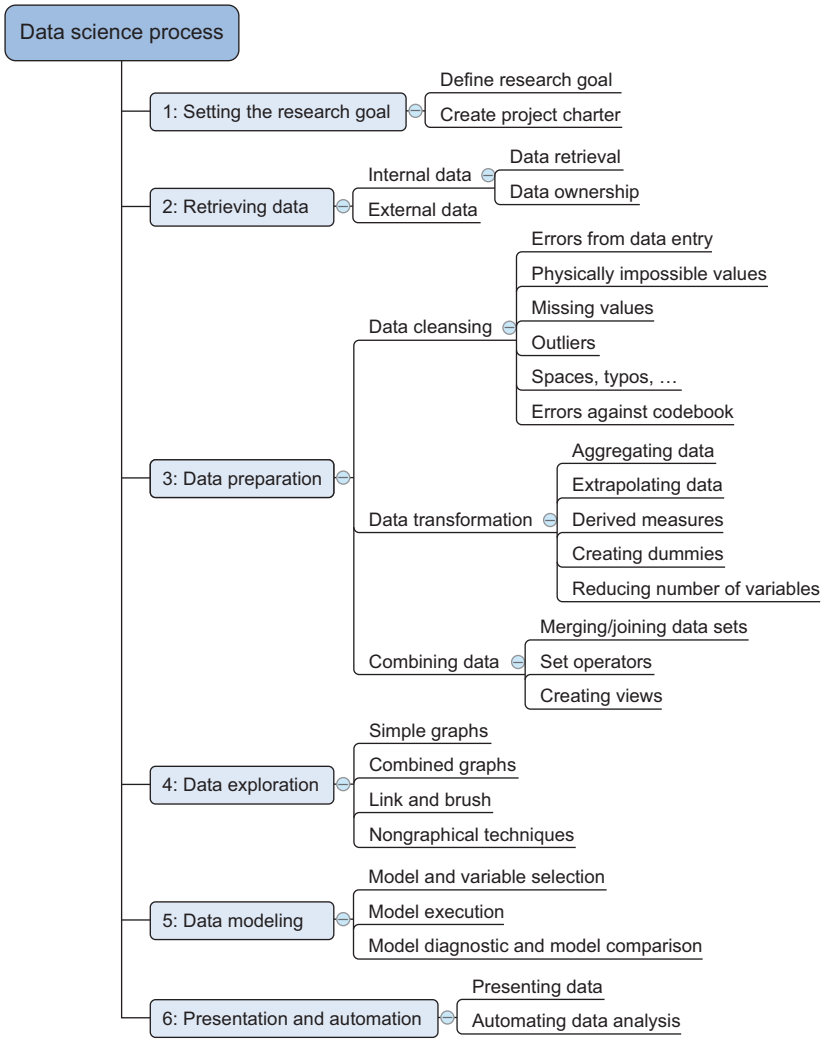



**Figure 2.1 The six steps of the data science process**

Figure 2.1 summarizes the data science process and shows the main steps and actions you'll take during a project. The following list is a short introduction; each of the steps will be discussed in greater depth throughout this chapter.

1. The first step of this process is setting a *research goal*. The main purpose here is making sure all the stakeholders understand the *what*, *how*, and *why* of the project. In every serious project this will result in a project charter.
2. The second phase is *data retrieval*. You want to have data available for analysis, so this step includes finding suitable data and getting access to the data from the

data owner. The result is data in its raw form, which probably needs polishing and transformation before it becomes usable.

3. Now that you have the raw data, it's time to *prepare* it. This includes transforming the data from a raw form into data that's directly usable in your models. To achieve this, you'll detect and correct different kinds of errors in the data, combine data from different data sources, and transform it. If you have successfully completed this step, you can progress to data visualization and modeling.
4. The fourth step is *data exploration*. The goal of this step is to gain a deep understanding of the data. You'll look for patterns, correlations, and deviations based on visual and descriptive techniques. The insights you gain from this phase will enable you to start modeling.
5. Finally, we get to the sexiest part: *model building (often referred to as "data modeling" throughout this book)*. It is now that you attempt to gain the insights or make the predictions stated in your project charter. Now is the time to bring out the heavy guns, but remember research has taught us that often (but not always) a combination of simple models tends to outperform one complicated model. If you've done this phase right, you're almost done.
6. The last step of the data science model is *presenting your results and automating the analysis,* if needed. One goal of a project is to change a process and/or make better decisions. You may still need to convince the business that your findings will indeed change the business process as expected. This is where you can shine in your influencer role. The importance of this step is more apparent in projects on a strategic and tactical level. Certain projects require you to perform the business process over and over again, so automating the project will save time.

In reality you won't progress in a linear way from step 1 to step 6. Often you'll regress and iterate between the different phases.

Following these six steps pays off in terms of a higher project success ratio and increased impact of research results. This process ensures you have a well-defined research plan, a good understanding of the business question, and clear deliverables before you even start looking at data. The first steps of your process focus on getting high-quality data as input for your models. This way your models will perform better later on. In data science there's a well-known saying: *Garbage in equals garbage out.*

Another benefit of following a structured approach is that you work more in *prototype mode* while you search for the best model. When building a *prototype,* you'll probably try multiple models and won't focus heavily on issues such as program speed or writing code against standards. This allows you to focus on bringing business value instead.

Not every project is initiated by the business itself. Insights learned during analysis or the arrival of new data can spawn new projects. When the data science team generates an idea, work has already been done to make a proposition and find a business sponsor.

Dividing a project into smaller stages also allows employees to work together as a team. It's impossible to be a specialist in everything. You'd need to know how to upload all the data to all the different databases, find an optimal data scheme that works not only for your application but also for other projects inside your company, and then keep track of all the statistical and data-mining techniques, while also being an expert in presentation tools and business politics. That's a hard task, and it's why more and more companies rely on a team of specialists rather than trying to find one person who can do it all.

The process we described in this section is best suited for a data science project that contains only a few models. It's not suited for every type of project. For instance, a project that contains millions of real-time models would need a different approach than the flow we describe here. A beginning data scientist should get a long way following this manner of working, though.

### 2.1.1 Don't be a slave to the process

Not every project will follow this blueprint, because your process is subject to the preferences of the data scientist, the company, and the nature of the project you work on. Some companies may require you to follow a strict protocol, whereas others have a more informal manner of working. In general, you'll need a structured approach when you work on a complex project or when many people or resources are involved.

The *agile* project model is an alternative to a sequential process with iterations. As this methodology wins more ground in the IT department and throughout the company, it's also being adopted by the data science community. Although the agile methodology is suitable for a data science project, many company policies will favor a more rigid approach toward data science.

Planning every detail of the data science process upfront isn't always possible, and more often than not you'll iterate between the different steps of the process. For instance, after the briefing you start your normal flow until you're in the exploratory data analysis phase. Your graphs show a distinction in the behavior between two groups—men and women maybe? You aren't sure because you don't have a variable that indicates whether the customer is male or female. You need to retrieve an extra data set to confirm this. For this you need to go through the approval process, which indicates that you (or the business) need to provide a kind of project charter. In big companies, getting all the data you need to finish your project can be an ordeal.

## 2.2 Step 1: Defining research goals and creating a project charter

A project starts by understanding the *what*, the *why*, and the *how* of your project (figure 2.2). What does the company expect you to do? And why does management place such a value on your research? Is it part of a bigger strategic picture or a "lone wolf" project originating from an opportunity someone detected? Answering these three

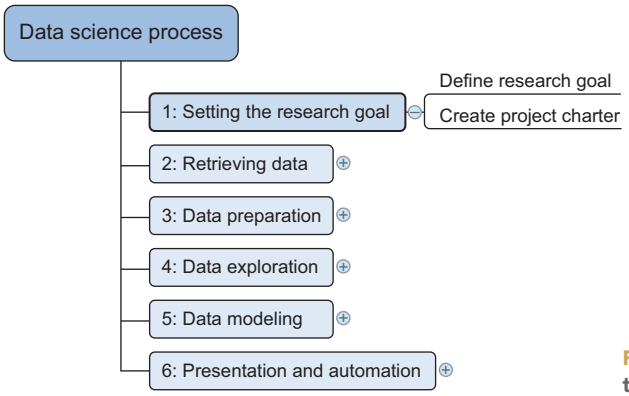



**Figure 2.2 Step 1: Setting the research goal**

questions (what, why, how) is the goal of the first phase, so that everybody knows what to do and can agree on the best course of action.

The outcome should be a clear research goal, a good understanding of the context, well-defined deliverables, and a plan of action with a timetable. This information is then best placed in a project charter. The length and formality can, of course, differ between projects and companies. In this early phase of the project, people skills and business acumen are more important than great technical prowess, which is why this part will often be guided by more senior personnel.

### 2.2.1 Spend time understanding the goals and context of your research

An essential outcome is the research goal that states the purpose of your assignment in a clear and focused manner. Understanding the business goals and context is critical for project success. Continue asking questions and devising examples until you grasp the exact business expectations, identify how your project fits in the bigger picture, appreciate how your research is going to change the business, and understand how they'll use your results. Nothing is more frustrating than spending months researching something until you have that one moment of brilliance and solve the problem, but when you report your findings back to the organization, everyone immediately realizes that you misunderstood their question. Don't skim over this phase lightly. Many data scientists fail here: despite their mathematical wit and scientific brilliance, they never seem to grasp the business goals and context.

### 2.2.2 Create a project charter

Clients like to know upfront what they're paying for, so after you have a good understanding of the business problem, try to get a formal agreement on the deliverables. All this information is best collected in a project charter. For any significant project this would be mandatory.

A project charter requires teamwork, and your input covers at least the following:

- A clear research goal
- The project mission and context
- How you're going to perform your analysis
- What resources you expect to use
- Proof that it's an achievable project, or proof of concepts
- Deliverables and a measure of success
- A timeline

Your client can use this information to make an estimation of the project costs and the data and people required for your project to become a success.

## 2.3 Step 2: Retrieving data

The next step in data science is to retrieve the required data (figure 2.3). Sometimes you need to go into the field and design a data collection process yourself, but most of the time you won't be involved in this step. Many companies will have already collected and stored the data for you, and what they don't have can often be bought from third parties. Don't be afraid to look outside your organization for data, because more and more organizations are making even high-quality data freely available for public and commercial use.

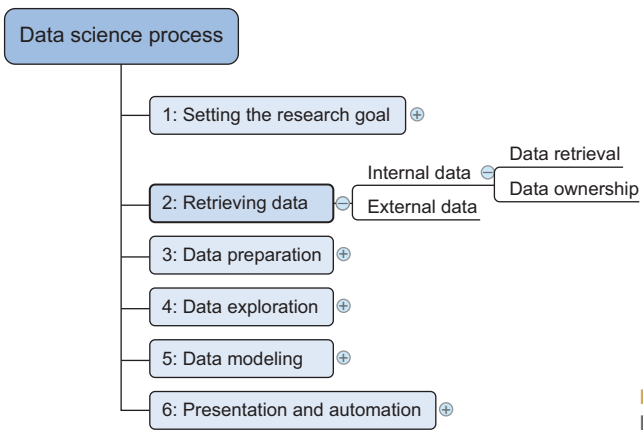



**Figure 2.3 Step 2: Retrieving data**

Data can be stored in many forms, ranging from simple text files to tables in a database. The objective now is acquiring all the data you need. This may be difficult, and even if you succeed, data is often like a diamond in the rough: it needs polishing to be of any use to you.

### 2.3.1 Start with data stored within the company

Your first act should be to assess the relevance and quality of the data that's readily available within your company. Most companies have a program for maintaining key data, so much of the cleaning work may already be done. This data can be stored in official data repositories such as *databases*, *data marts*, *data warehouses*, and *data lakes* maintained by a team of IT professionals. The primary goal of a *database* is data storage, while a *data warehouse* is designed for reading and analyzing that data. A data mart is a subset of the *data warehouse* and geared toward serving a specific business unit. While data warehouses and data marts are home to preprocessed data, *data lakes* contains data in its natural or raw format. But the possibility exists that your data still resides in Excel files on the desktop of a domain expert.

Finding data even within your own company can sometimes be a challenge. As companies grow, their data becomes scattered around many places. Knowledge of the data may be dispersed as people change positions and leave the company. Documentation and metadata aren't always the top priority of a delivery manager, so it's possible you'll need to develop some Sherlock Holmes–like skills to find all the lost bits.

Getting access to data is another difficult task. Organizations understand the value and sensitivity of data and often have policies in place so everyone has access to what they need and nothing more. These policies translate into physical and digital barriers called *Chinese walls*. These "walls" are mandatory and well-regulated for customer data in most countries. This is for good reasons, too; imagine everybody in a credit card company having access to your spending habits. Getting access to the data may take time and involve company politics.

### 2.3.2 Don't be afraid to shop around

If data isn't available inside your organization, look outside your organization's walls. Many companies specialize in collecting valuable information. For instance, Nielsen and GFK are well known for this in the retail industry. Other companies provide data so that you, in turn, can enrich their services and ecosystem. Such is the case with Twitter, LinkedIn, and Facebook.

Although data is considered an asset more valuable than oil by certain companies, more and more governments and organizations share their data for free with the world. This data can be of excellent quality; it depends on the institution that creates and manages it. The information they share covers a broad range of topics such as the number of accidents or amount of drug abuse in a certain region and its demographics. This data is helpful when you want to enrich proprietary data but also convenient when training your data science skills at home. Table 2.1 shows only a small selection from the growing number of open-data providers.

Table 2.1 A list of open-data providers that should get you started

| Open data site | Description |
|---|---|
| Data.gov | The home of the US Government's open data |
| https://open-data.europa.eu/ | The home of the European Commission's open data |
| Freebase.org | An open database that retrieves its information from sites like Wikipedia, MusicBrains, and the SEC archive |
| Data.worldbank.org | Open data initiative from the World Bank |
| Aiddata.org | Open data for international development |
| Open.fda.gov | Open data from the US Food and Drug Administration |

### 2.3.3 Do data quality checks now to prevent problems later

Expect to spend a good portion of your project time doing data correction and cleansing, sometimes up to 80%. The retrieval of data is the first time you'll inspect the data in the data science process. Most of the errors you'll encounter during the data-gathering phase are easy to spot, but being too careless will make you spend many hours solving data issues that could have been prevented during data import.

You'll investigate the data during the import, data preparation, and exploratory phases. The difference is in the goal and the depth of the investigation. During *data retrieval*, you check to see if the data is equal to the data in the source document and look to see if you have the right data types. This shouldn't take too long; when you have enough evidence that the data is similar to the data you find in the source document, you stop. With *data preparation*, you do a more elaborate check. If you did a good job during the previous phase, the errors you find now are also present in the source document. The focus is on the content of the variables: you want to get rid of typos and other data entry errors and bring the data to a common standard among the data sets. For example, you might correct USQ to USA and United Kingdom to UK. During the *exploratory phase* your focus shifts to what you can learn from the data. Now you assume the data to be clean and look at the statistical properties such as distributions, correlations, and outliers. You'll often iterate over these phases. For instance, when you discover outliers in the exploratory phase, they can point to a data entry error. Now that you understand how the quality of the data is improved during the process, we'll look deeper into the data preparation step.

## 2.4 Step 3: Cleansing, integrating, and transforming data

The data received from the data retrieval phase is likely to be "a diamond in the rough." Your task now is to sanitize and prepare it for use in the modeling and reporting phase. Doing so is tremendously important because your models will perform better and you'll lose less time trying to fix strange output. It can't be mentioned nearly enough times: garbage in equals garbage out. Your model needs the data in a specific

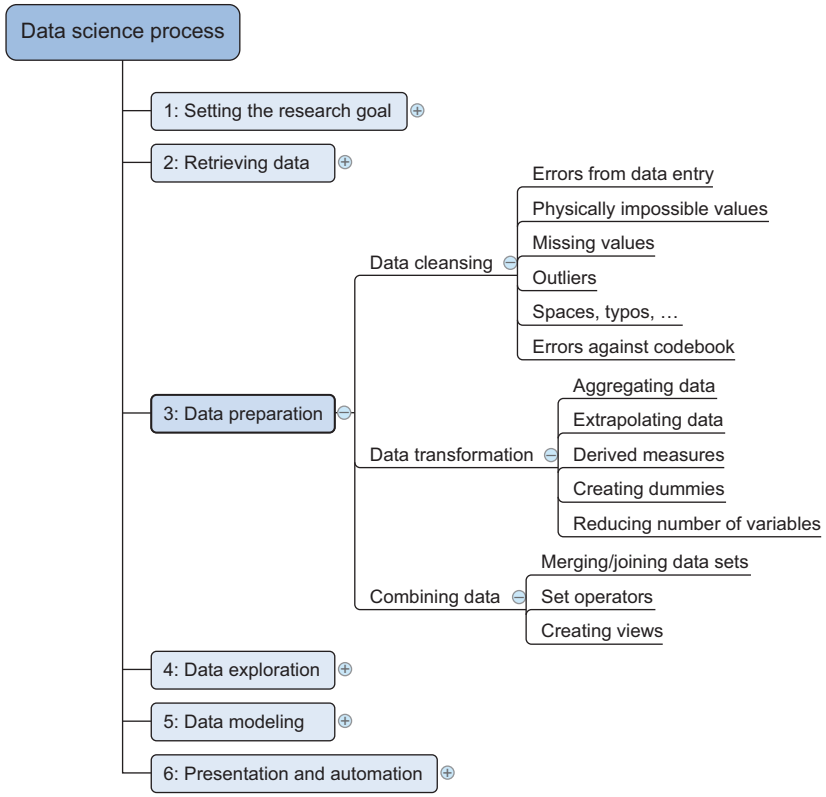



**Figure 2.4 Step 3: Data preparation**

format, so data transformation will always come into play. It's a good habit to correct data errors as early on in the process as possible. However, this isn't always possible in a realistic setting, so you'll need to take corrective actions in your program.

Figure 2.4 shows the most common actions to take during the data cleansing, integration, and transformation phase.

This mind map may look a bit abstract for now, but we'll handle all of these points in more detail in the next sections. You'll see a great commonality among all of these actions.

### 2.4.1 Cleansing data

Data cleansing is a subprocess of the data science process that focuses on removing errors in your data so your data becomes a true and consistent representation of the processes it originates from.

By "true and consistent representation" we imply that at least two types of errors exist. The first type is the *interpretation error*, such as when you take the value in your

data for granted, like saying that a person's age is greater than 300 years. The second type of error points to *inconsistencies* between data sources or against your company's standardized values. An example of this class of errors is putting "Female" in one table and "F" in another when they represent the same thing: that the person is female. Another example is that you use Pounds in one table and Dollars in another. Too many possible errors exist for this list to be exhaustive, but table 2.2 shows an overview of the types of errors that can be detected with easy checks—the "low hanging fruit," as it were.

**Table 2.2 An overview of common errors**

| General solution | |
|---|---|
| Try to fix the problem early in the data acquisition chain or else fix it in the program. | |
| **Error description** | **Possible solution** |
| *Errors pointing to false values within one data set* | |
| Mistakes during data entry | Manual overrules |
| Redundant white space | Use string functions |
| Impossible values | Manual overrules |
| Missing values | Remove observation or value |
| Outliers | Validate and, if erroneous, treat as missing value (remove or insert) |
| *Errors pointing to inconsistencies between data sets* | |
| Deviations from a code book | Match on keys or else use manual overrules |
| Different units of measurement | Recalculate |
| Different levels of aggregation | Bring to same level of measurement by aggregation or extrapolation |

Sometimes you'll use more advanced methods, such as simple modeling, to find and identify data errors; diagnostic plots can be especially insightful. For example, in figure 2.5 we use a measure to identify data points that seem out of place. We do a regression to get acquainted with the data and detect the influence of individual observations on the regression line. When a single observation has too much influence, this can point to an error in the data, but it can also be a valid point. At the data cleansing stage, these advanced methods are, however, rarely applied and often regarded by certain data scientists as overkill.

Now that we've given the overview, it's time to explain these errors in more detail.

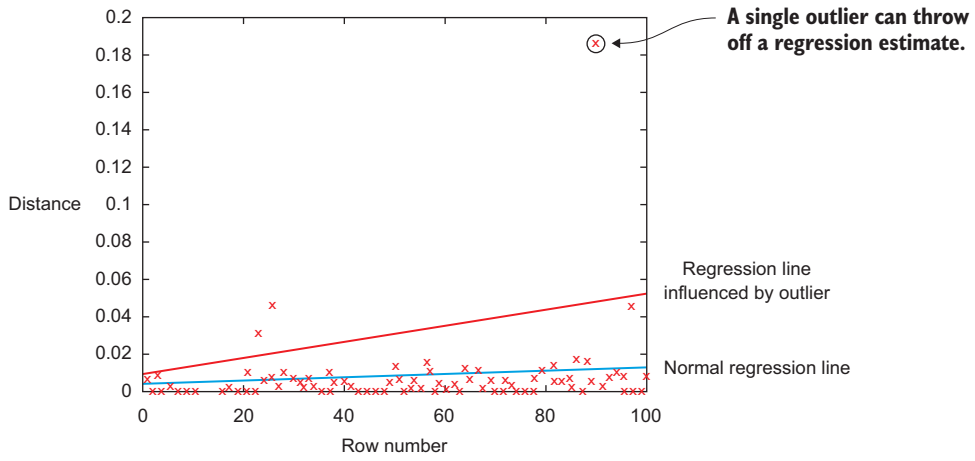



**Figure 2.5** **The encircled point influences the model heavily and is worth investigating because it can point to a region where you don't have enough data or might indicate an error in the data, but it also can be a valid data point.**

### DATA ENTRY ERRORS

Data collection and data entry are error-prone processes. They often require human intervention, and because humans are only human, they make typos or lose their concentration for a second and introduce an error into the chain. But data collected by machines or computers isn't free from errors either. Errors can arise from human sloppiness, whereas others are due to machine or hardware failure. Examples of errors originating from machines are transmission errors or bugs in the extract, transform, and load phase (ETL).

For small data sets you can check every value by hand. Detecting data errors when the variables you study don't have many classes can be done by tabulating the data with counts. When you have a variable that can take only two values: "Good" and "Bad", you can create a frequency table and see if those are truly the only two values present. In table 2.3, the values "Godo" and "Bade" point out something went wrong in at least 16 cases.

**Table 2.3** **Detecting outliers on simple variables with a frequency table**

| Value | Count |
|---|---|
| Good | 1598647 |
| Bad | 1354468 |
| Godo | 15 |
| Bade | 1 |

Most errors of this type are easy to fix with simple assignment statements and if-then-else rules:

```
if x == "Godo":
    x = "Good"
if x == "Bade":
    x = "Bad"
```

#### Redundant whitespace

Whitespaces tend to be hard to detect but cause errors like other redundant characters would. Who hasn't lost a few days in a project because of a bug that was caused by whitespaces at the end of a string? You ask the program to join two keys and notice that observations are missing from the output file. After looking for days through the code, you finally find the bug. Then comes the hardest part: explaining the delay to the project stakeholders. The cleaning during the ETL phase wasn't well executed, and keys in one table contained a whitespace at the end of a string. This caused a mismatch of keys such as "FR " – "FR", dropping the observations that couldn't be matched.

If you know to watch out for them, fixing redundant whitespaces is luckily easy enough in most programming languages. They all provide string functions that will remove the leading and trailing whitespaces. For instance, in Python you can use the `strip()` function to remove leading and trailing spaces.

**Fixing capital letter mismatches** Capital letter mismatches are common. Most programming languages make a distinction between "Brazil" and "brazil". In this case you can solve the problem by applying a function that returns both strings in lowercase, such as `.lower()` in Python. `"Brazil".lower() == "brazil".lower()` should result in `true`.

#### Impossible values and sanity checks

Sanity checks are another valuable type of data check. Here you check the value against physically or theoretically impossible values such as people taller than 3 meters or someone with an age of 299 years. Sanity checks can be directly expressed with rules:

```
check = 0 <= age <= 120
```

#### Outliers

An outlier is an observation that seems to be distant from other observations or, more specifically, one observation that follows a different logic or generative process than the other observations. The easiest way to find outliers is to use a plot or a table with the minimum and maximum values. An example is shown in figure 2.6.

The plot on the top shows no outliers, whereas the plot on the bottom shows possible outliers on the upper side when a normal distribution is expected. The normal distribution, or Gaussian distribution, is the most common distribution in natural sciences.

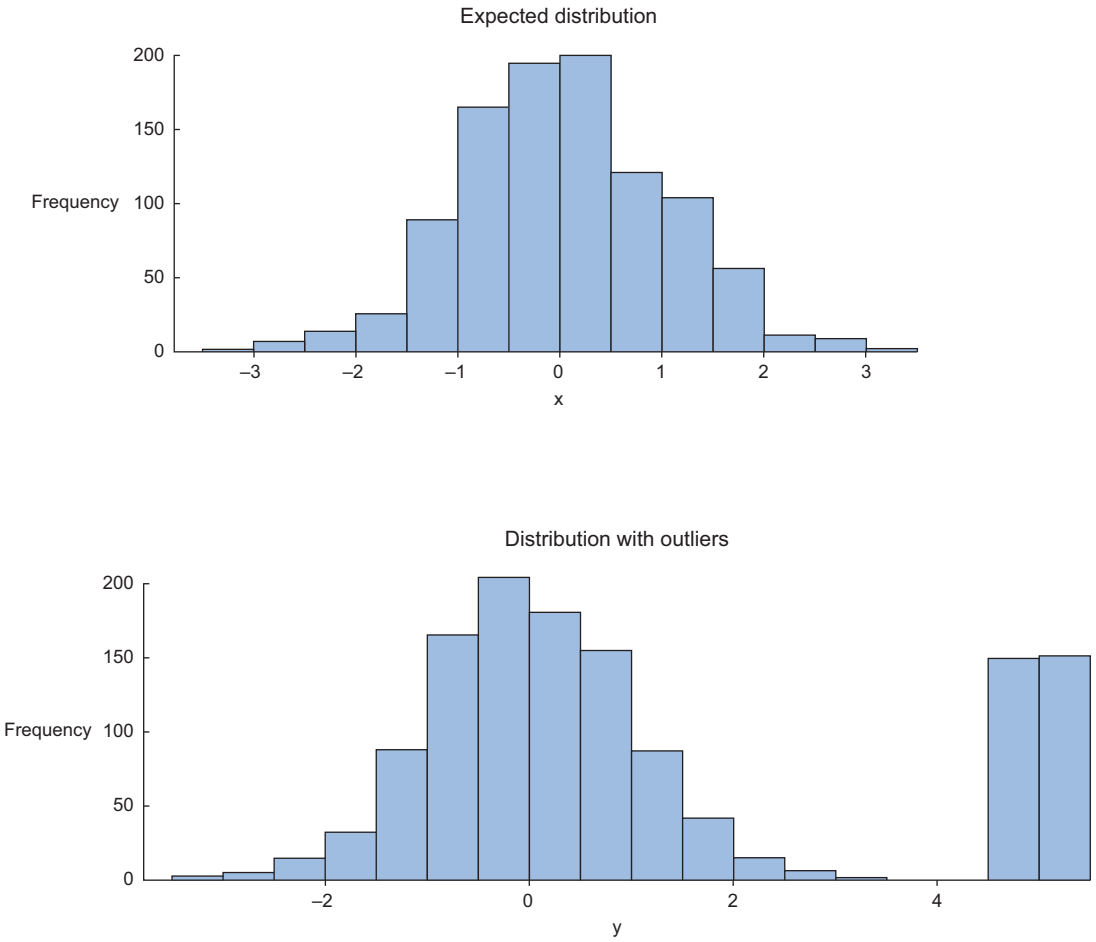



**Figure 2.6 Distribution plots are helpful in detecting outliers and helping you understand the variable.**

It shows most cases occurring around the average of the distribution and the occurrences decrease when further away from it. The high values in the bottom graph can point to outliers when assuming a normal distribution. As we saw earlier with the regression example, outliers can gravely influence your data modeling, so investigate them first.

#### DEALING WITH MISSING VALUES

Missing values aren't necessarily wrong, but you still need to handle them separately; certain modeling techniques can't handle missing values. They might be an indicator that something went wrong in your data collection or that an error happened in the ETL process. Common techniques data scientists use are listed in table 2.4.

Table 2.4 An overview of techniques to handle missing data

| Technique | Advantage | Disadvantage |
|---|---|---|
| Omit the values | Easy to perform | You lose the information from an observation |
| Set value to `null` | Easy to perform | Not every modeling technique and/or implementation can handle `null` values |
| Impute a static value such as 0 or the mean | Easy to perform<br>You don't lose information from the other variables in the observation | Can lead to false estimations from a model |
| Impute a value from an estimated or theoretical distribution | Does not disturb the model as much | Harder to execute<br>You make data assumptions |
| Modeling the value (nondependent) | Does not disturb the model too much | Can lead to too much confidence in the model<br>Can artificially raise dependence among the variables<br>Harder to execute<br>You make data assumptions |

Which technique to use at what time is dependent on your particular case. If, for instance, you don't have observations to spare, omitting an observation is probably not an option. If the variable can be described by a stable distribution, you could impute based on this. However, maybe a missing value actually means "zero"? This can be the case in sales for instance: if no promotion is applied on a customer basket, that customer's promo is missing, but most likely it's also 0, no price cut.

#### Deviations from a code book

Detecting errors in larger data sets against a code book or against standardized values can be done with the help of set operations. A *code book* is a description of your data, a form of metadata. It contains things such as the number of variables per observation, the number of observations, and what each encoding within a variable means. (For instance "0" equals "negative", "5" stands for "very positive".) A code book also tells the type of data you're looking at: is it hierarchical, graph, something else?

You look at those values that are present in set A but not in set B. These are values that should be corrected. It's no coincidence that *sets* are the data structure that we'll use when we're working in code. It's a good habit to give your data structures additional thought; it can save work and improve the performance of your program.

If you have multiple values to check, it's better to put them from the code book into a table and use a difference operator to check the discrepancy between both tables. This way, you can profit from the power of a database directly. More on this in chapter 5.

#### DIFFERENT UNITS OF MEASUREMENT

When integrating two data sets, you have to pay attention to their respective units of measurement. An example of this would be when you study the prices of gasoline in the world. To do this you gather data from different data providers. Data sets can contain prices per gallon and others can contain prices per liter. A simple conversion will do the trick in this case.

#### DIFFERENT LEVELS OF AGGREGATION

Having different levels of aggregation is similar to having different types of measurement. An example of this would be a data set containing data per week versus one containing data per work week. This type of error is generally easy to detect, and *summarizing* (or the inverse, *expanding*) the data sets will fix it.

After cleaning the data errors, you combine information from different data sources. But before we tackle this topic we'll take a little detour and stress the importance of cleaning data as early as possible.

### 2.4.2 Correct errors as early as possible

A good practice is to mediate data errors as early as possible in the data collection chain and to fix as little as possible inside your program while fixing the origin of the problem. Retrieving data is a difficult task, and organizations spend millions of dollars on it in the hope of making better decisions. The data collection process is error-prone, and in a big organization it involves many steps and teams.

Data should be cleansed when acquired for many reasons:

- Not everyone spots the data anomalies. Decision-makers may make costly mistakes on information based on incorrect data from applications that fail to correct for the faulty data.
- If errors are not corrected early on in the process, the cleansing will have to be done for every project that uses that data.
- Data errors may point to a business process that isn't working as designed. For instance, both authors worked at a retailer in the past, and they designed a couponing system to attract more people and make a higher profit. During a data science project, we discovered clients who abused the couponing system and earned money while purchasing groceries. The goal of the couponing system was to stimulate cross-selling, not to give products away for free. This flaw cost the company money and nobody in the company was aware of it. In this case the data wasn't technically wrong but came with unexpected results.
- Data errors may point to defective equipment, such as broken transmission lines and defective sensors.
- Data errors can point to bugs in software or in the integration of software that may be critical to the company. While doing a small project at a bank we discovered that two software applications used different local settings. This caused problems with numbers greater than 1,000. For one app the number 1.000 meant one, and for the other it meant one thousand.

Fixing the data as soon as it's captured is nice in a perfect world. Sadly, a data scientist doesn't always have a say in the data collection and simply telling the IT department to fix certain things may not make it so. If you can't correct the data at the source, you'll need to handle it inside your code. Data manipulation doesn't end with correcting mistakes; you still need to combine your incoming data.

As a final remark: always keep a copy of your original data (if possible). Sometimes you start cleaning data but you'll make mistakes: impute variables in the wrong way, delete outliers that had interesting additional information, or alter data as the result of an initial misinterpretation. If you keep a copy you get to try again. For "flowing data" that's manipulated at the time of arrival, this isn't always possible and you'll have accepted a period of tweaking before you get to use the data you are capturing. One of the more difficult things isn't the data cleansing of individual data sets however, it's combining different sources into a whole that makes more sense.

### 2.4.3 Combining data from different data sources

Your data comes from several different places, and in this substep we focus on integrating these different sources. Data varies in size, type, and structure, ranging from databases and Excel files to text documents.

We focus on data in table structures in this chapter for the sake of brevity. It's easy to fill entire books on this topic alone, and we choose to focus on the data science process instead of presenting scenarios for every type of data. But keep in mind that other types of data sources exist, such as key-value stores, document stores, and so on, which we'll handle in more appropriate places in the book.

#### The different ways of combining data

You can perform two operations to combine information from different data sets. The first operation is *joining*: enriching an observation from one table with information from another table. The second operation is *appending* or *stacking*: adding the observations of one table to those of another table.

When you combine data, you have the option to create a new physical table or a virtual table by creating a view. The advantage of a view is that it doesn't consume more disk space. Let's elaborate a bit on these methods.

#### Joining tables

Joining tables allows you to combine the information of one observation found in one table with the information that you find in another table. The focus is on enriching a single observation. Let's say that the first table contains information about the purchases of a customer and the other table contains information about the region where your customer lives. Joining the tables allows you to combine the information so that you can use it for your model, as shown in figure 2.7.

To join tables, you use variables that represent the same object in both tables, such as a date, a country name, or a Social Security number. These common fields are known as keys. When these keys also uniquely define the records in the table they

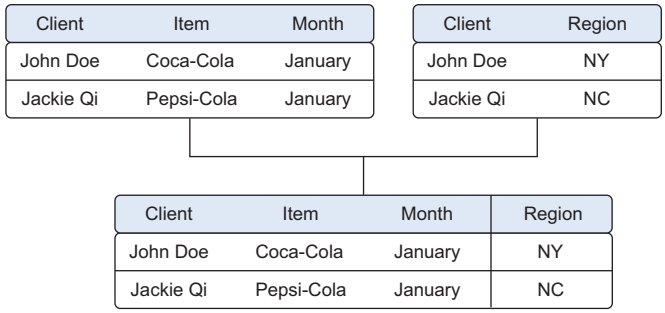



**Figure 2.7 Joining two tables on the Item and Region keys**

are called *primary keys*. One table may have buying behavior and the other table may have demographic information on a person. In figure 2.7 both tables contain the client name, and this makes it easy to enrich the client expenditures with the region of the client. People who are acquainted with Excel will notice the similarity with using a lookup function.

The number of resulting rows in the output table depends on the exact join type that you use. We introduce the different types of joins later in the book.

### APPENDING TABLES

Appending or stacking tables is effectively adding observations from one table to another table. Figure 2.8 shows an example of appending tables. One table contains the observations from the month January and the second table contains observations from the month February. The result of appending these tables is a larger one with the observations from January as well as February. The equivalent operation in set theory would be the union, and this is also the command in SQL, the common language of relational databases. Other set operators are also used in data science, such as set difference and intersection.

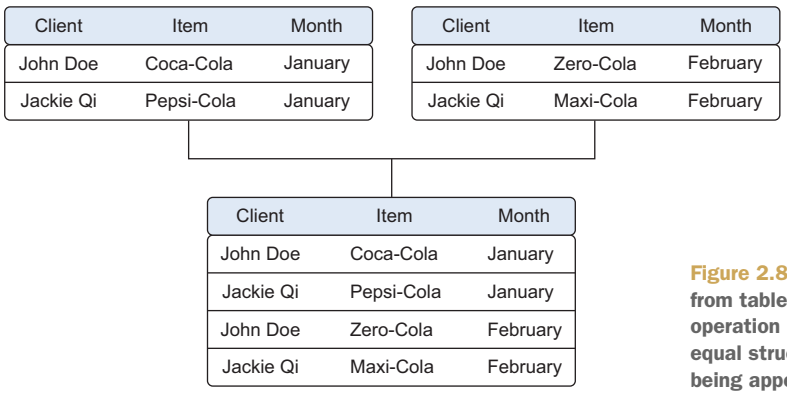



**Figure 2.8 Appending data from tables is a common operation but requires an equal structure in the tables being appended.**

### Using views to simulate data joins and appends

To avoid duplication of data, you virtually combine data with views. In the previous example we took the monthly data and combined it in a new physical table. The problem is that we duplicated the data and therefore needed more storage space. In the example we're working with, that may not cause problems, but imagine that every table consists of terabytes of data; then it becomes problematic to duplicate the data. For this reason, the concept of a view was invented. *A view* behaves as if you're working on a table, but this table is nothing but a virtual layer that combines the tables for you. Figure 2.9 shows how the sales data from the different months is combined virtually into a yearly sales table instead of duplicating the data. Views do come with a drawback, however. While a table join is only performed once, the join that creates the view is recreated every time it's queried, using more processing power than a pre-calculated table would have.

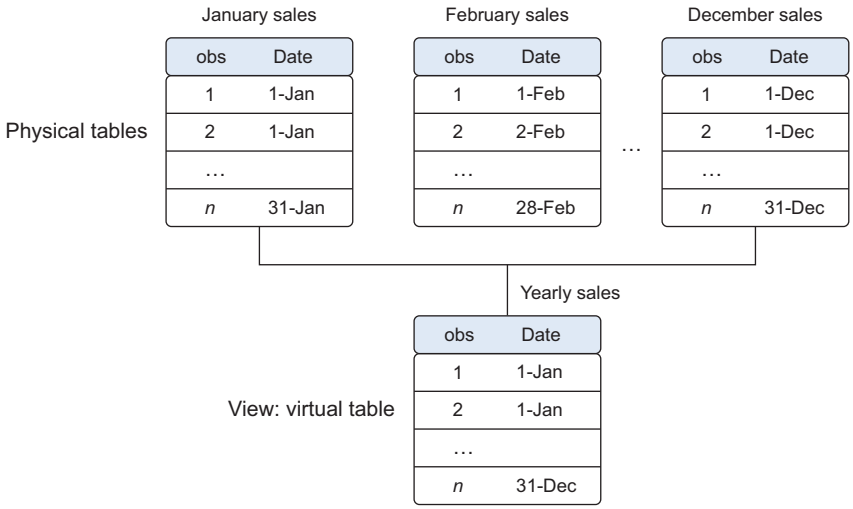



**Figure 2.9 A view helps you combine data without replication.**

### Enriching aggregated measures

Data enrichment can also be done by adding calculated information to the table, such as the total number of sales or what percentage of total stock has been sold in a certain region (figure 2.10).

Extra measures such as these can add perspective. Looking at figure 2.10, we now have an aggregated data set, which in turn can be used to calculate the participation of each product within its category. This could be useful during data exploration but more so when creating data models. As always this depends on the exact case, but from our experience models with "relative measures" such as % sales (quantity of

| Product class | Product | Sales in $ | Sales t-1 in $ | Growth | Sales by product class | Rank sales |
|---|---|---|---|---|---|---|
| A | B | X | Y | (X-Y) / Y | AX | NX |
| Sport | Sport 1 | 95 | 98 | –3.06% | 215 | 2 |
| Sport | Sport 2 | 120 | 132 | –9.09% | 215 | 1 |
| Shoes | Shoes 1 | 10 | 6 | 66.67% | 10 | 3 |

**Figure 2.10 Growth, sales by product class, and rank sales are examples of derived and aggregate measures.**

product sold/total quantity sold) tend to outperform models that use the raw numbers (quantity sold) as input.

### 2.4.4 Transforming data

Certain models require their data to be in a certain shape. Now that you've cleansed and integrated the data, this is the next task you'll perform: transforming your data so it takes a suitable form for data modeling.

#### TRANSFORMING DATA

Relationships between an input variable and an output variable aren't always linear. Take, for instance, a relationship of the form $y = ae^{bx}$. Taking the log of the independent variables simplifies the estimation problem dramatically. Figure 2.11 shows how

| $x$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| log($x$) | 0.00 | 0.43 | 0.68 | 0.86 | 1.00 | 1.11 | 1.21 | 1.29 | 1.37 | 1.43 |
| $y$ | 0.00 | 0.44 | 0.69 | 0.87 | 1.02 | 1.11 | 1.24 | 1.32 | 1.38 | 1.46 |

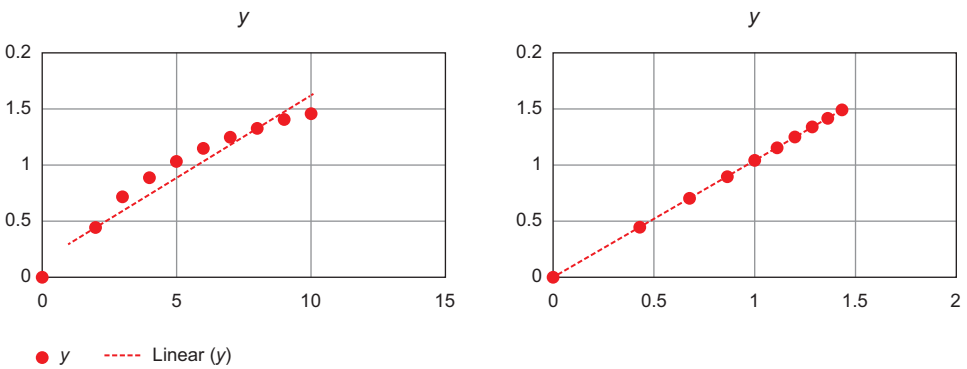



**Figure 2.11 Transforming x to log x makes the relationship between x and y linear (right), compared with the non-log x (left).**

transforming the input variables greatly simplifies the estimation problem. Other times you might want to combine two variables into a new variable.

#### REDUCING THE NUMBER OF VARIABLES

Sometimes you have too many variables and need to reduce the number because they don't add new information to the model. Having too many variables in your model makes the model difficult to handle, and certain techniques don't perform well when you overload them with too many input variables. For instance, all the techniques based on a Euclidean distance perform well only up to 10 variables.

### Euclidean distance

Euclidean distance or "ordinary" distance is an extension to one of the first things anyone learns in mathematics about triangles (trigonometry): Pythagoras's leg theorem. If you know the length of the two sides next to the 90° angle of a right-angled triangle you can easily derive the length of the remaining side (hypotenuse). The formula for this is hypotenuse = $\sqrt{(side1 + side2)^2}$. The Euclidean distance between two points in a two-dimensional plane is calculated using a similar formula: distance = $\sqrt{((x1 - x2)^2 + (y1 - y2)^2)}$. If you want to expand this distance calculation to more dimensions, add the coordinates of the point within those higher dimensions to the formula. For three dimensions we get distance = $\sqrt{((x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2}$.

Data scientists use special methods to reduce the number of variables but retain the maximum amount of data. We'll discuss several of these methods in chapter 3. Figure 2.12 shows how reducing the number of variables makes it easier to understand the

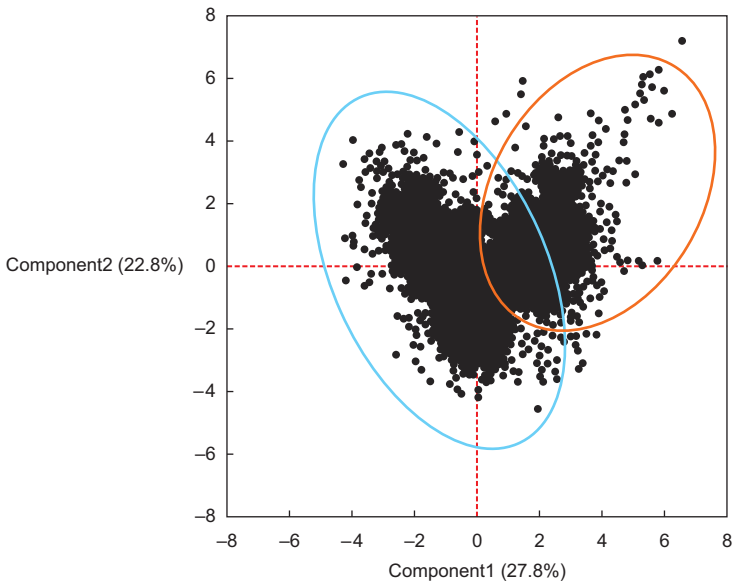



**Figure 2.12 Variable reduction allows you to reduce the number of variables while maintaining as much information as possible.**

key values. It also shows how two variables account for 50.6% of the variation within the data set (component1 = 27.8% + component2 = 22.8%). These variables, called "component1" and "component2," are both combinations of the original variables. They're the *principal components* of the underlying data structure. If it isn't all that clear at this point, don't worry, principal components analysis (PCA) will be explained more thoroughly in chapter 3. What you can also see is the presence of a third (unknown) variable that splits the group of observations into two.

### TURNING VARIABLES INTO DUMMIES

Variables can be turned into dummy variables (figure 2.13). *Dummy variables* can only take two values: true(1) or false(0). They're used to indicate the absence of a categorical effect that may explain the observation. In this case you'll make separate columns for the classes stored in one variable and indicate it with 1 if the class is present and 0 otherwise. An example is turning one column named Weekdays into the columns Monday through Sunday. You use an indicator to show if the observation was on a Monday; you put 1 on Monday and 0 elsewhere. Turning variables into dummies is a technique that's used in modeling and is popular with, but not exclusive to, economists.

In this section we introduced the third step in the data science process—cleaning, transforming, and integrating data—which changes your raw data into usable input for the modeling phase. The next step in the data science process is to get a better

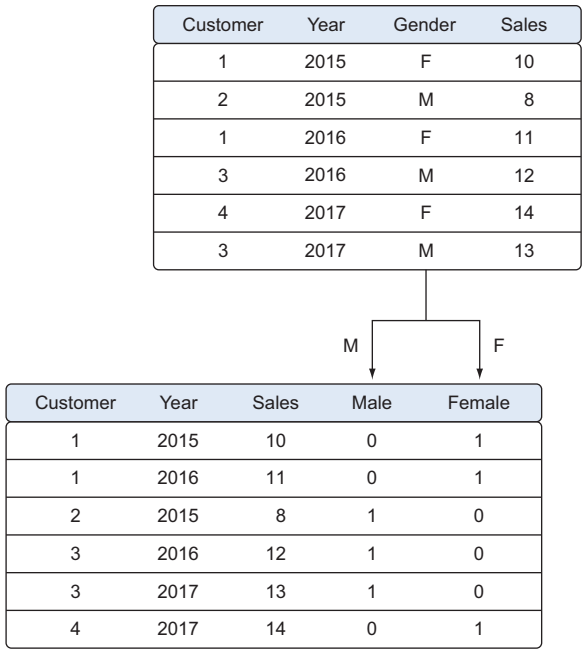



**Figure 2.13 Turning variables into dummies is a data transformation that breaks a variable that has multiple classes into multiple variables, each having only two possible values: 0 or 1.**

understanding of the content of the data and the relationships between the variables and observations; we explore this in the next section.

## 2.5 Step 4: Exploratory data analysis

During exploratory data analysis you take a deep dive into the data (see figure 2.14). Information becomes much easier to grasp when shown in a picture, therefore you mainly use graphical techniques to gain an understanding of your data and the interactions between variables. This phase is about exploring data, so keeping your mind open and your eyes peeled is essential during the exploratory data analysis phase. The goal isn't to cleanse the data, but it's common that you'll still discover anomalies you missed before, forcing you to take a step back and fix them.

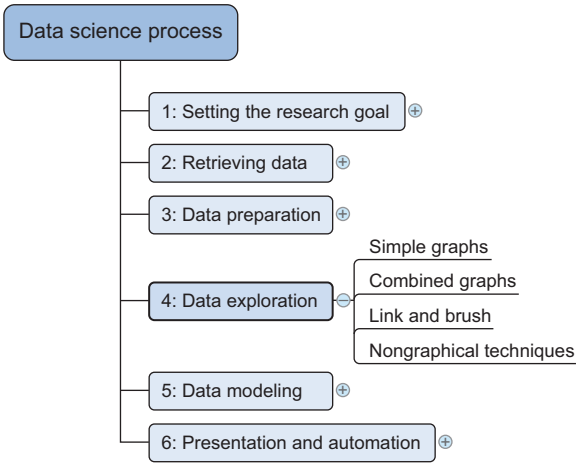



**Figure 2.14 Step 4: Data exploration**

The visualization techniques you use in this phase range from simple line graphs or histograms, as shown in figure 2.15, to more complex diagrams such as Sankey and network graphs. Sometimes it's useful to compose a composite graph from simple graphs to get even more insight into the data. Other times the graphs can be animated or made interactive to make it easier and, let's admit it, way more fun. An example of an interactive Sankey diagram can be found at http://bost.ocks.org/mike/sankey/.

Mike Bostock has interactive examples of almost any type of graph. It's worth spending time on his website, though most of his examples are more useful for data presentation than data exploration.

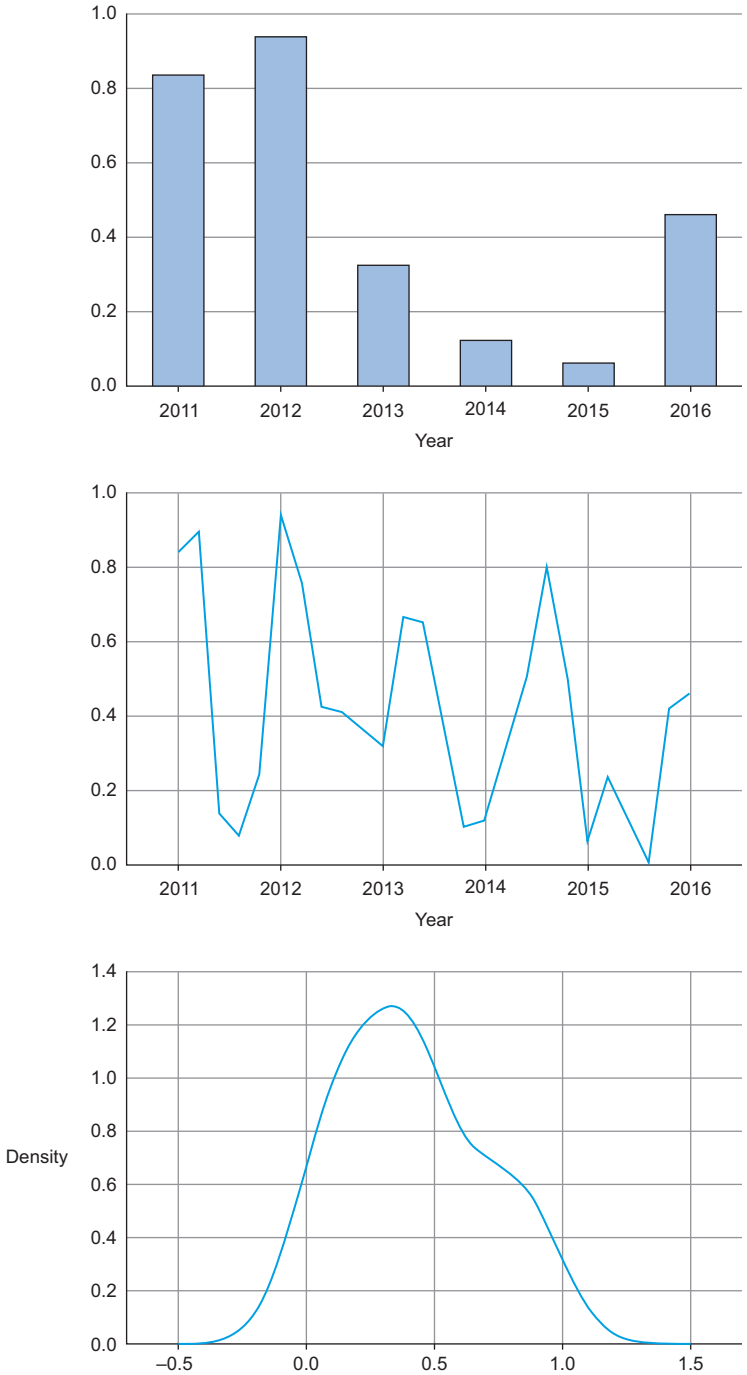



**Figure 2.15** **From top to bottom, a bar chart, a line plot, and a distribution are some of the graphs used in exploratory analysis.**

These plots can be combined to provide even more insight, as shown in figure 2.16.

Overlaying several plots is common practice. In figure 2.17 we combine simple graphs into a Pareto diagram, or 80-20 diagram.

Figure 2.18 shows another technique: *brushing and linking*. With brushing and linking you combine and link different graphs and tables (or views) so changes in one graph are automatically transferred to the other graphs. An elaborate example of this can be found in chapter 9. This interactive exploration of data facilitates the discovery of new insights.

Figure 2.18 shows the average score per country for questions. Not only does this indicate a high correlation between the answers, but it's easy to see that when you select several points on a subplot, the points will correspond to similar points on the other graphs. In this case the selected points on the left graph correspond to points on the middle and right graphs, although they correspond better in the middle and right graphs.

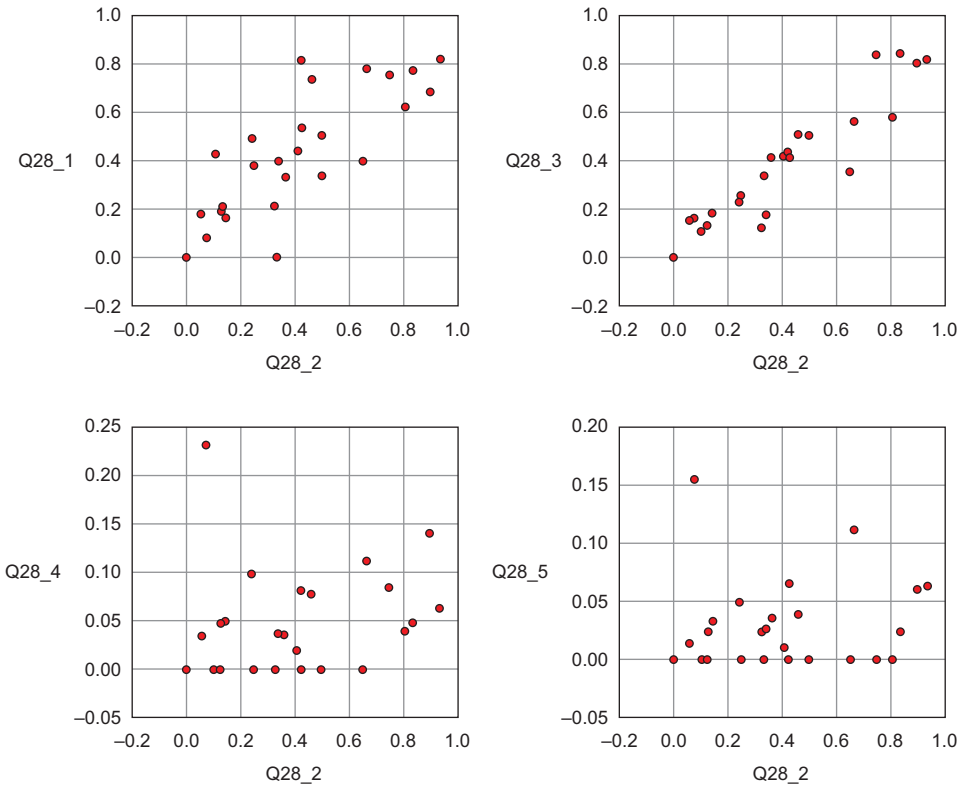



**Figure 2.16** **Drawing multiple plots together can help you understand the structure of your data over multiple variables.**

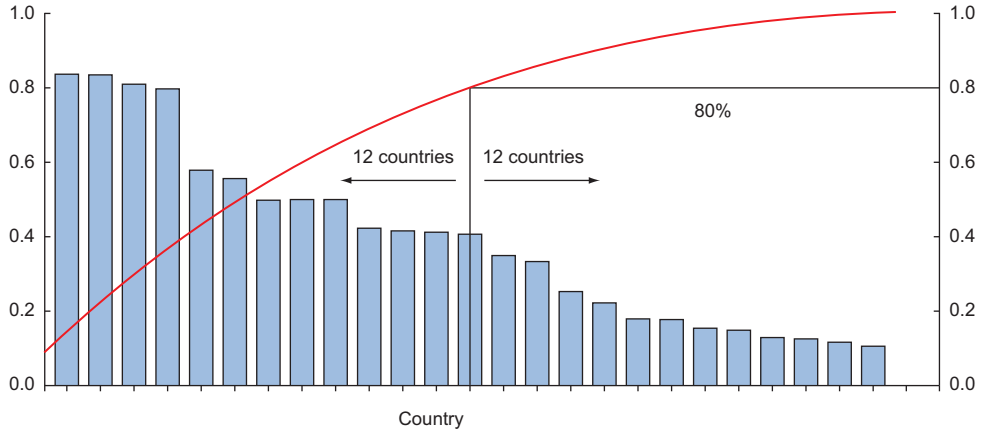



**Figure 2.17** **A Pareto diagram is a combination of the values and a cumulative distribution. It's easy to see from this diagram that the first 50% of the countries contain slightly less than 80% of the total amount. If this graph represented customer buying power and we sell expensive products, we probably don't need to spend our marketing budget in every country; we could start with the first 50%.**

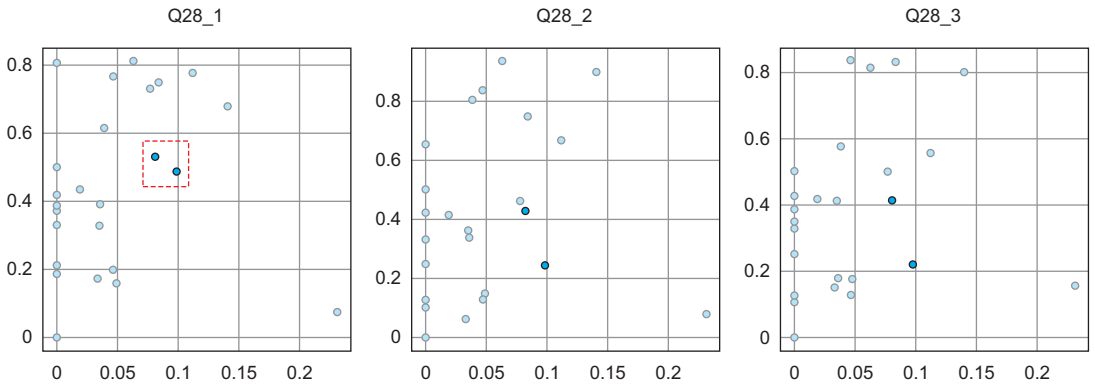



**Figure 2.18** **Link and brush allows you to select observations in one plot and highlight the same observations in the other plots.**

Two other important graphs are the histogram shown in figure 2.19 and the boxplot shown in figure 2.20.

In a histogram a variable is cut into discrete categories and the number of occurrences in each category are summed up and shown in the graph. The boxplot, on the other hand, doesn't show how many observations are present but does offer an impression of the distribution within categories. It can show the maximum, minimum, median, and other characterizing measures at the same time.

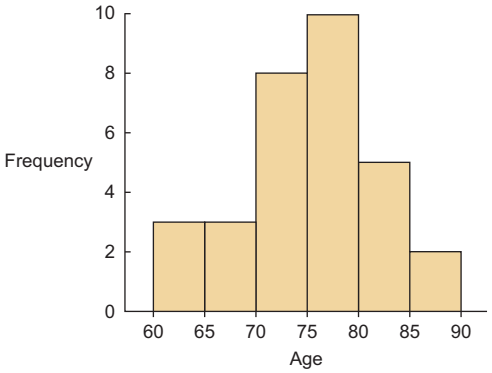



**Figure 2.19 Example histogram: the number of people in the age-groups of 5-year intervals**

The techniques we described in this phase are mainly visual, but in practice they're certainly not limited to visualization techniques. Tabulation, clustering, and other modeling techniques can also be a part of exploratory analysis. Even building simple models can be a part of this step.

Now that you've finished the data exploration phase and you've gained a good grasp of your data, it's time to move on to the next phase: building models.

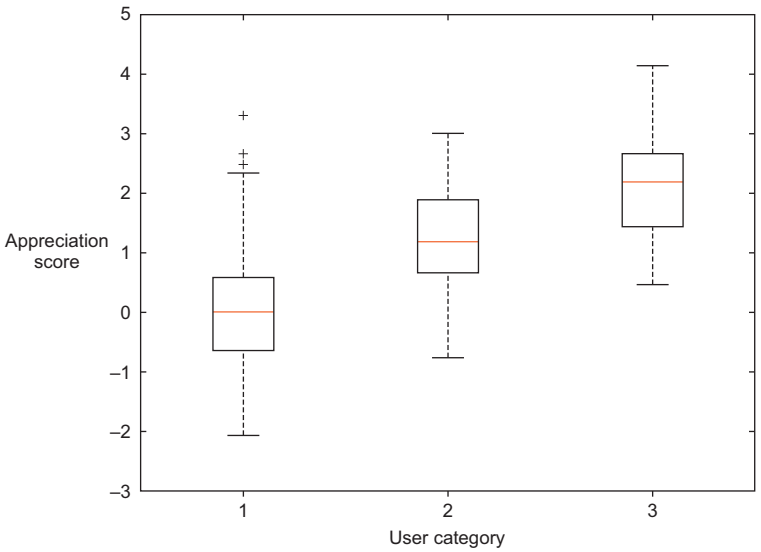



**Figure 2.20 Example boxplot: each user category has a distribution of the appreciation each has for a certain picture on a photography website.**

## 2.6 Step 5: Build the models

With clean data in place and a good understanding of the content, you're ready to build models with the goal of making better predictions, classifying objects, or gaining an understanding of the system that you're modeling. This phase is much more focused than the exploratory analysis step, because you know what you're looking for and what you want the outcome to be. Figure 2.21 shows the components of model building.

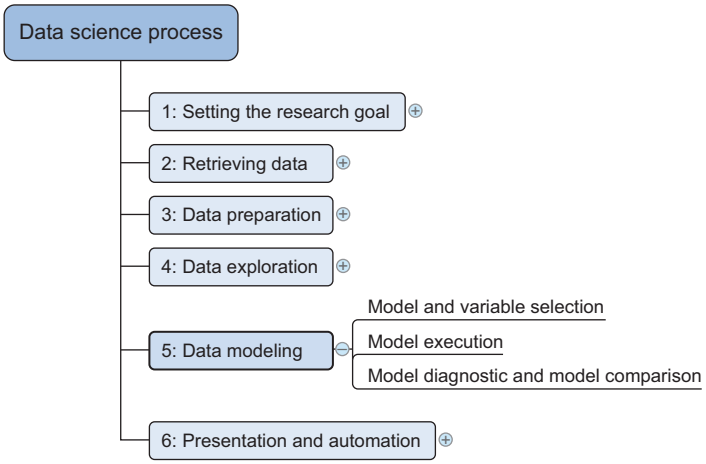



**Figure 2.21 Step 5: Data modeling**

The techniques you'll use now are borrowed from the field of machine learning, data mining, and/or statistics. In this chapter we only explore the tip of the iceberg of existing techniques, while chapter 3 introduces them properly. It's beyond the scope of this book to give you more than a conceptual introduction, but it's enough to get you started; 20% of the techniques will help you in 80% of the cases because techniques overlap in what they try to accomplish. They often achieve their goals in similar but slightly different ways.

Building a model is an iterative process. The way you build your model depends on whether you go with classic statistics or the somewhat more recent machine learning school, and the type of technique you want to use. Either way, most models consist of the following main steps:

1. Selection of a modeling technique and variables to enter in the model
2. Execution of the model
3. Diagnosis and model comparison

### 2.6.1 Model and variable selection

You'll need to select the variables you want to include in your model and a modeling technique. Your findings from the exploratory analysis should already give a fair idea

of what variables will help you construct a good model. Many modeling techniques are available, and choosing the right model for a problem requires judgment on your part. You'll need to consider model performance and whether your project meets all the requirements to use your model, as well as other factors:

- Must the model be moved to a production environment and, if so, would it be easy to implement?
- How difficult is the maintenance on the model: how long will it remain relevant if left untouched?
- Does the model need to be easy to explain?

When the thinking is done, it's time for action.

### 2.6.2 Model execution

Once you've chosen a model you'll need to implement it in code.

> **REMARK** This is the first time we'll go into actual Python code execution so make sure you have a virtual env up and running. Knowing how to set this up is required knowledge, but if it's your first time, check out *appendix D*.
>
> All code from this chapter can be downloaded from https://www.manning.com/books/introducing-data-science. This chapter comes with an ipython (.ipynb) notebook and Python (.py) file.

Luckily, most programming languages, such as Python, already have libraries such as StatsModels or Scikit-learn. These packages use several of the most popular techniques. Coding a model is a nontrivial task in most cases, so having these libraries available can speed up the process. As you can see in the following code, it's fairly easy to use linear regression (figure 2.22) with StatsModels or Scikit-learn. Doing this yourself would require much more effort even for the simple techniques. The following listing shows the execution of a linear prediction model.

**Listing 2.1 Executing a linear prediction model on semi-random data**

```
import statsmodels.api as sm
import numpy as np
predictors = np.random.random(1000).reshape(500,2)
target = predictors.dot(np.array([0.4, 0.6])) + np.random.random(500)
lmRegModel = sm.OLS(target,predictors)
result = lmRegModel.fit()
result.summary()
```

Imports required Python modules.

Creates random data for predictors (x-values) and semi-random data for the target (y-values) of the model. We use predictors as input to create the target so we infer a correlation here.

Fits linear regression on data.

Shows model fit statistics.

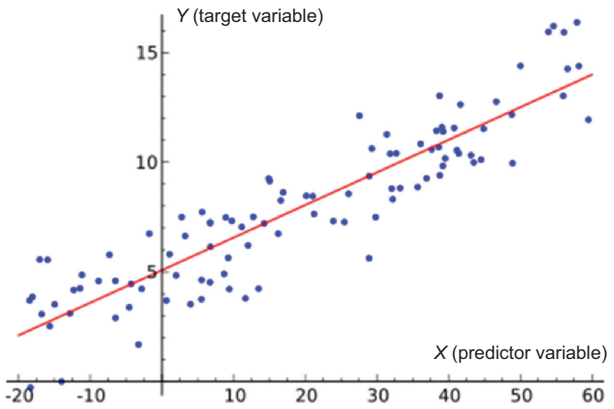



**Figure 2.22** Linear regression tries to fit a line while minimizing the distance to each point

Okay, we cheated here, quite heavily so. We created predictor values that are meant to predict how the target variables behave. For a linear regression, a "linear relation" between each x (predictor) and the y (target) variable is assumed, as shown in figure 2.22.

We, however, created the target variable, based on the predictor by adding a bit of randomness. It shouldn't come as a surprise that this gives us a well-fitting model. The `results.summary()` outputs the table in figure 2.23. Mind you, the exact outcome depends on the random variables you got.

| | | | |
|---|---|---|---|
| **Dep. Variable:** | y | **R-squared:** | 0.893 |
| **Model:** | OLS | **Adj. R-squared:** | 0.893 |
| **Method:** | Least Squares | **F-statistic:** | 2088. |
| **Date:** | Fri, 30 Oct 2015 | **Prob (F-statistic):** | 7.13e-243 |
| **Time:** | 12:44:31 | **Log-Likelihood:** | -176.74 |
| **No. Observations:** | 500 | **AIC:** | 357.5 |
| **Df Residuals:** | 498 | **BIC:** | 365.9 |
| **Df Model:** | 2 | | |
| **Covariance Type:** | nonrobust | | |

**Model fit: higher is better but too high is suspicious.**

| | coef | std err | t | P>\|t\| | [95.0% Conf. Int.] |
|---|---|---|---|---|---|
| **x1** | 0.7658 | 0.040 | 19.130 | 0.000 | 0.687 0.844 |
| **x2** | 1.1252 | 0.039 | 28.603 | 0.000 | 1.048 1.202 |

**p-value to show whether a predictor variable has a significant influence on the target. Lower is better and <0.05 is often considered "significant."**

| | | | |
|---|---|---|---|
| **Omnibus:** | 34.269 | **Durbin-Watson:** | 1.943 |
| **Prob(Omnibus):** | 0.000 | **Jarque-Bera (JB):** | 13.480 |
| **Skew:** | -0.125 | **Prob(JB):** | 0.00118 |
| **Kurtosis:** | 2.235 | **Cond. No.** | 2.51 |

**Linear equation coefficients.**
**y = 0.7658xl + l.l252x2.**

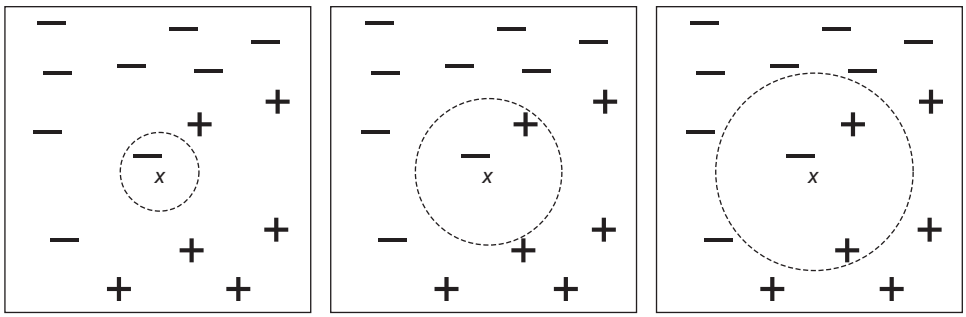
**Figure 2.23** Linear regression model information output

Let's ignore most of the output we got here and focus on the most important parts:

- *Model fit*—For this the R-squared or adjusted R-squared is used. This measure is an indication of the amount of variation in the data that gets captured by the model. The difference between the adjusted R-squared and the R-squared is minimal here because the adjusted one is the normal one + a penalty for model complexity. A model gets complex when many variables (or features) are introduced. You don't need a complex model if a simple model is available, so the adjusted R-squared punishes you for overcomplicating. At any rate, 0.893 is high, and it should be because we cheated. Rules of thumb exist, but for models in businesses, models above 0.85 are often considered good. If you want to win a competition you need in the high 90s. For research however, often very low model fits (<0.2 even) are found. What's more important there is the influence of the introduced predictor variables.
- *Predictor variables have a coefficient*—For a linear model this is easy to interpret. In our example if you add "1" to x1, it will change y by "0.7658". It's easy to see how finding a good predictor can be your route to a Nobel Prize even though your model as a whole is rubbish. If, for instance, you determine that a certain gene is significant as a cause for cancer, this is important knowledge, even if that gene in itself doesn't determine whether a person will get cancer. The example here is classification, not regression, but the point remains the same: detecting influences is more important in scientific studies than perfectly fitting models (not to mention more realistic). But when do we know a gene has that impact? This is called significance.
- *Predictor significance*—Coefficients are great, but sometimes not enough evidence exists to show that the influence is there. This is what the p-value is about. A long explanation about type 1 and type 2 mistakes is possible here but the short explanations would be: if the p-value is lower than 0.05, the variable is considered significant for most people. In truth, this is an arbitrary number. It means there's a 5% chance the predictor doesn't have any influence. Do you accept this 5% chance to be wrong? That's up to you. Several people introduced the extremely significant ($p<0.01$) and marginally significant thresholds ($p<0.1$).

Linear regression works if you want to predict a value, but what if you want to classify something? Then you go to classification models, the best known among them being k-nearest neighbors.

As shown in figure 2.24, k-nearest neighbors looks at labeled points nearby an unlabeled point and, based on this, makes a prediction of what the label should be.




(a) 1-nearest neighbor (b) 2-nearest neighbor (c) 3-nearest neighbor

**Figure 2.24** **K-nearest neighbor techniques look at the k-nearest point to make a prediction.**

Let's try it in Python code using the Scikit learn library, as in this next listing.

**Listing 2.2 Executing k-nearest neighbor classification on semi-random data**

```
from sklearn import neighbors
predictors = np.random.random(1000).reshape(500,2)
target = np.around(predictors.dot(np.array([0.4, 0.6])) +
         np.random.random(500))
clf = neighbors.KNeighborsClassifier(n_neighbors=10)
knn = clf.fit(predictors,target)
knn.score(predictors, target)
```

**Imports modules.**

**Creates random predictor data and semi-random target data based on predictor data.**

**Fits 10-nearest neighbors model.**

**Gets model fit score: what percent of the classification was correct?**

As before, we construct random correlated data and surprise, surprise we get 85% of cases correctly classified. If we want to look in depth, we need to score the model. Don't let `knn.score()` fool you; it returns the model accuracy, but by "scoring a model" we often mean applying it on data to make a prediction.

```
prediction = knn.predict(predictors)
```

Now we can use the prediction and compare it to the real thing using a *confusion matrix*.

```
metrics.confusion_matrix(target,prediction)
```

We get a 3-by-3 matrix as shown in figure 2.25.

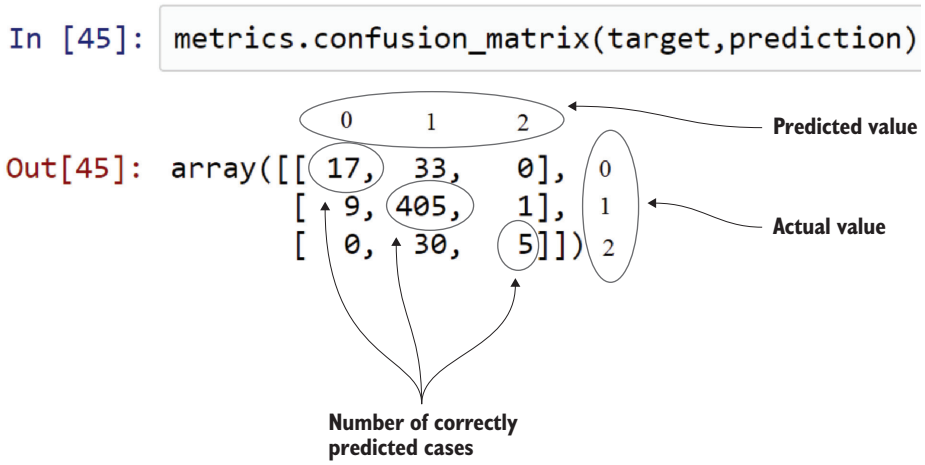



**Figure 2.25** **Confusion matrix: it shows how many cases were correctly classified and incorrectly classified by comparing the prediction with the real values. Remark: the classes (0,1,2) were added in the figure for clarification.**

The confusion matrix shows we have correctly predicted 17+405+5 cases, so that's good. But is it really a surprise? No, for the following reasons:

- For one, the classifier had but three options; marking the difference with last time `np.around()` will round the data to its nearest integer. In this case that's either 0, 1, or 2. With only 3 options, you can't do much worse than 33% correct on 500 guesses, even for a real random distribution like flipping a coin.
- Second, we cheated again, correlating the response variable with the predictors. Because of the way we did this, we get most observations being a "1". By guessing "1" for every case we'd already have a similar result.
- We compared the prediction with the real values, true, but we never predicted based on fresh data. The prediction was done using the same data as the data used to build the model. This is all fine and dandy to make yourself feel good, but it gives you no indication of whether your model will work when it encounters truly new data. For this we need a holdout sample, as will be discussed in the next section.

Don't be fooled. Typing this code won't work miracles by itself. It might take a while to get the modeling part and all its parameters right.

To be honest, only a handful of techniques have industry-ready implementations in Python. But it's fairly easy to use models that are available in R within Python with the help of the RPy library. RPy provides an interface from Python to R. *R* is a free software environment, widely used for statistical computing. If you haven't already, it's worth at least a look, because in 2014 it was still one of the most popular

(if not the most popular) programming languages for data science. For more information, see http://www.kdnuggets.com/polls/2014/languages-analytics-data-mining-data-science.html.

### 2.6.3 Model diagnostics and model comparison

You'll be building multiple models from which you then choose the best one based on multiple criteria. Working with a holdout sample helps you pick the best-performing model. A *holdout sample* is a part of the data you leave out of the model building so it can be used to evaluate the model afterward. The principle here is simple: the model should work on unseen data. You use only a fraction of your data to estimate the model and the other part, the holdout sample, is kept out of the equation. The model is then unleashed on the unseen data and error measures are calculated to evaluate it. Multiple error measures are available, and in figure 2.26 we show the general idea on comparing models. The error measure used in the example is the mean square error.

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

**Figure 2.26 Formula for mean square error**

Mean square error is a simple measure: check for every prediction how far it was from the truth, square this error, and add up the error of every prediction.

Figure 2.27 compares the performance of two models to predict the order size from the price. The first model is *size = 3 * price* and the second model is *size = 10*. To

| | *n* | Size | Price | Predicted model 1 | Predicted model 2 | Error model 1 | Error model 2 |
|---|---|---|---|---|---|---|---|
| | 1 | 10 | 3 | | | | |
| | 2 | 15 | 5 | | | | |
| | 3 | 18 | 6 | | | | |
| | 4 | 14 | 5 | | | | |
| | … | … | | | | | |
| 80% train | 800 | 9 | 3 | | | | |
| | 801 | 12 | 4 | 12 | 10 | 0 | 2 |
| | 802 | 13 | 4 | 12 | 10 | 1 | 3 |
| | … | | | | | | |
| | 999 | 21 | 7 | 21 | 10 | 0 | 11 |
| 20% test | 1000 | 10 | 4 | 12 | 10 | –2 | 0 |
| | | | | Total | | 5861 | 110225 |

**Figure 2.27 A holdout sample helps you compare models and ensures that you can generalize results to data that the model has not yet seen.**

estimate the models, we use 800 randomly chosen observations out of 1,000 (or 80%), without showing the other 20% of data to the model. Once the model is trained, we predict the values for the other 20% of the variables based on those for which we already know the true value, and calculate the model error with an error measure. Then we choose the model with the lowest error. In this example we chose model 1 because it has the lowest total error.

Many models make strong assumptions, such as independence of the inputs, and you have to verify that these assumptions are indeed met. This is called *model diagnostics*.

This section gave a short introduction to the steps required to build a valid model. Once you have a working model you're ready to go to the last step.

## 2.7 Step 6: Presenting findings and building applications on top of them

After you've successfully analyzed the data and built a well-performing model, you're ready to present your findings to the world (figure 2.28). This is an exciting part; all your hours of hard work have paid off and you can explain what you found to the stakeholders.

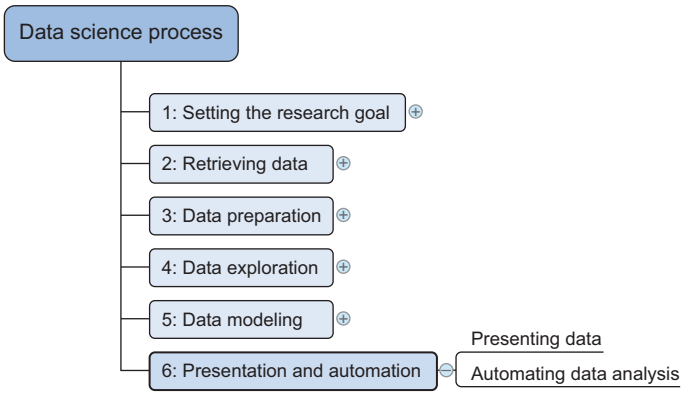



**Figure 2.28 Step 6: Presentation and automation**

Sometimes people get so excited about your work that you'll need to repeat it over and over again because they value the predictions of your models or the insights that you produced. For this reason, you need to automate your models. This doesn't always mean that you have to redo all of your analysis all the time. Sometimes it's sufficient that you implement only the model scoring; other times you might build an application that automatically updates reports, Excel spreadsheets, or PowerPoint presentations. The last stage of the data science process is where your *soft skills* will be most useful, and yes, they're extremely important. In fact, we recommend you find dedicated books and other information on the subject and work through them, because why bother doing all this tough work if nobody listens to what you have to say?

If you've done this right, you now have a working model and satisfied stakeholders, so we can conclude this chapter here.

## 2.8 Summary

In this chapter you learned the data science process consists of six steps:

- *Setting the research goal*—Defining the what, the why, and the how of your project in a project charter.
- *Retrieving data*—Finding and getting access to data needed in your project. This data is either found within the company or retrieved from a third party.
- *Data preparation*—Checking and remediating data errors, enriching the data with data from other data sources, and transforming it into a suitable format for your models.
- *Data exploration*—Diving deeper into your data using descriptive statistics and visual techniques.
- *Data modeling*—Using machine learning and statistical techniques to achieve your project goal.
- *Presentation and automation*—Presenting your results to the stakeholders and industrializing your analysis process for repetitive reuse and integration with other tools.

# *Processing data files*

One of the key ingredients in data science projects is, obviously, data. Much of the time that data will be stored in files. Those files might be found in different places, and they probably will have different formats and different structures. It will be your job to get those files and decide how to extract their data and combine in way that is meaningful for your project. You will also almost certainly need to massage and clean that data in various ways.

The following chapter, "Processing data files," is from my book, *The Quick Python Book, 3rd edition*. While it's intended to be a book introducing all of the Python language, for the 3rd edition I chose to use the last 5 chapters to focus a bit more on how to use Python to handle data.

This chapter introduces several aspects of reading and writing data files, from plain text files and delimited files to more structured formats like JSON and XML, even to spreadsheet files. It also discusses several common scenarios for massaging and cleaning the data extracted from those files, including handling incorrectly encoded files, null bytes, and other common hassles.

Chapter 21 from *The Quick Python Book, 3rd edition* by Naomi Ceder.

# *Processing data files*

***This chapter covers***

- Using ETL (extract-transform-load)
- Reading text data files (plain text and CSV)
- Reading spreadsheet files
- Normalizing, cleaning, and sorting data
- Writing data files

Much of the data available is contained in text files. This data can range from unstructured text, such as a corpus of tweets or literary texts, to more structured data in which each row is a record and the fields are delimited by a special character, such as a comma, a tab, or a pipe (|). Text files can be huge; a data set can be spread over tens or even hundreds of files, and the data in it can be incomplete or horribly dirty. With all the variations, it's almost inevitable that you'll need to read and use data from text files. This chapter gives you strategies for using Python to do exactly that.

## 21.1 Welcome to ETL

The need to get data out of files, parse it, turn it into a useful format, and then do something with it has been around for as long as there have been data files. In fact, there is a standard term for the process: extract-transform-load (ETL). The extraction refers to the process of reading a data source and parsing it, if necessary. The transformation can be cleaning and normalizing the data, as well as combining, breaking up, or reorganizing the records it contains. The loading refers to storing the transformed data in a new place, either a different file or a database. This chapter deals with the basics of ETL in Python, starting with text-based data files and storing the transformed data in other files. I look at more structured data files in chapter 22 and storage in databases in chapter 23.

## 21.2 Reading text files

The first part of ETL—the "extract" portion—involves opening a file and reading its contents. This process seems like a simple one, but even at this point there can be issues, such as the file's size. If a file is too large to fit into memory and be manipulated, you need to structure your code to handle smaller segments of the file, possibly operating one line at a time.

### 21.2.1 Text encoding: ASCII, Unicode, and others

Another possible pitfall is in the encoding. This chapter deals with text files, and in fact, much of the data exchanged in the real world is in text files. But the exact nature of *text* can vary from application to application, from person to person, and of course from country to country.

Sometimes, *text* means something in the ASCII encoding, which has 128 characters, only 95 of which are printable. The good news about ASCII encoding is that it's the lowest common denominator of most data exchange. The bad news is that it doesn't begin to handle the complexities of the many alphabets and writing systems of the world. Reading files using ASCII encoding is almost certain to cause trouble and throw errors on character values that it doesn't understand, whether it's a German ü, a Portuguese ç, or something from almost any language other than English.

These errors arise because ASCII is based on 7-bit values, whereas the bytes in a typical file are 8 bits, allowing 256 possible values as opposed to the 128 of a 7-bit value. It's routine to use those additional values to store additional characters—anything from extra punctuation (such as the printer's en dash and em dash) to symbols (such as the trademark, copyright, and degree symbols) to accented versions of alphabetical characters. The problem has always been that if, in reading a text file, you encounter a character in the 128 outside the ASCII range, you have no way of knowing for sure how it was encoded. Is the character value of 214, say, a division symbol, an Ö, or something else? Short of having the code that created the file, you have no way to know.

### UNICODE AND UTF-8

One way to mitigate this confusion is Unicode. The Unicode encoding called UTF-8 accepts the basic ASCII characters without any change but also allows an almost unlimited set of other characters and symbols according to the Unicode standard. Because of its flexibility, UTF-8 was used in more 85% of web pages served at the time I wrote this chapter, which means that your best bet for reading text files is to assume UTF-8 encoding. If the files contain only ASCII characters, they'll still be read correctly, but you'll also be covered if other characters are encoded in UTF-8. The good news is that the Python 3 string data type was designed to handle Unicode by default.

Even with Unicode, there'll be occasions when your text contains values that can't be successfully encoded. Fortunately, the `open` function in Python accepts an optional errors parameter that tells it how to deal with encoding errors when reading or writing files. The default option is `'strict'`, which causes an error to be raised whenever an encoding error is encountered. Other useful options are `'ignore'`, which causes the character causing the error to be skipped; `'replace'`, which causes the character to be replaced by a marker character (often, `?`); `'backslashreplace'`, which replaces the character with a backslash escape sequence; and `'surrogateescape'`, which translates the offending character to a private Unicode code point on reading and back to the original sequence of bytes on writing. Your particular use case will determine how strict you need to be in handling or resolving encoding issues.

Look at a short example of a file containing an invalid UTF-8 character, and see how the different options handle that character. First, write the file, using bytes and binary mode:

```
>>> open('test.txt', 'wb').write(bytes([65, 66, 67, 255, 192,193]))
```

This code results in a file that contains "ABC" followed by three non-ASCII characters, which may be rendered differently depending on the encoding used. If you use vim to look at the file, you see

```
ABCÿÀÁ
~
```

Now that you have the file, try reading it with the default `'strict'` errors option:

```
>>> x = open('test.txt').read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python3.6/codecs.py", line 321, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xff in position 3:
     invalid start byte
```

The fourth byte, which had a value of 255, isn't a valid UTF-8 character in that position, so the `'strict'` errors setting raises an exception. Now see how the other error

options handle the same file, keeping in mind that the last three characters raise an error:

```
>>> open('test.txt', errors='ignore').read()
'ABC'
>>> open('test.txt', errors='replace').read()
'ABC'
>>> open('test.txt', errors='surrogateescape').read()
'ABC\udcff\udcc0\udcc1'
>>> open('test.txt', errors='backslashreplace').read()
'ABC\\xff\\xc0\\xc1'
>>>
```

If you want any problem characters to disappear, `'ignore'` is the option to use. The `'replace'` option only marks the place occupied by the invalid character, and the other options in different ways attempt to preserve the invalid characters without interpretation.

### 21.2.2 Unstructured text

Unstructured text files are the easiest sort of data to read but the hardest to extract information from. Processing unstructured text can vary enormously, depending on both the nature of the text and what you want to do with it, so any comprehensive discussion of text processing is beyond the scope of this book. A short example, however, can illustrate some of the basic issues and set the stage for a discussion of structured text data files.

One of the simplest issues is deciding what forms a basic logical unit in the file. If you have a corpus of thousands of tweets, the text of *Moby Dick*, or a collection of news stories, you need to be able to break them up into cohesive units. In the case of tweets, each may fit onto a single line, and you can read and process each line of the file fairly simply.

In the case of *Moby Dick* or even a news story, the problem can be trickier. You may not want to treat all of a novel or news item as a single item in many cases. But if that's the case, you need to decide what sort of unit you do want and then come up with a strategy to divide the file accordingly. Perhaps you want to consider the text paragraph by paragraph. In that case, you need to identify how paragraphs are separated in your file and create your code accordingly. If a paragraph is the same as a line in the text file, the job is easy. Often, however, the line breaks in a text file are shorter, and you need to do a bit more work.

Now look at a couple of examples:

```
Call me Ishmael.  Some years ago--never mind how long precisely--
having little or no money in my purse, and nothing particular
to interest me on shore, I thought I would sail about a little
and see the watery part of the world.  It is a way I have
of driving off the spleen and regulating the circulation.
Whenever I find myself growing grim about the mouth;
whenever it is a damp, drizzly November in my soul; whenever I
find myself involuntarily pausing before coffin warehouses,
and bringing up the rear of every funeral I meet;
and especially whenever my hypos get such an upper hand of me,
```

```
that it requires a strong moral principle to prevent me from
deliberately stepping into the street, and methodically knocking
people's hats off--then, I account it high time to get to sea
as soon as I can.  This is my substitute for pistol and ball.
With a philosophical flourish Cato throws himself upon his sword;
I quietly take to the ship.  There is nothing surprising in this.
If they but knew it, almost all men in their degree, some time
or other, cherish very nearly the same feelings towards
the ocean with me.

There now is your insular city of the Manhattoes, belted round by wharves
as Indian isles by coral reefs--commerce surrounds it with her surf.
Right and left, the streets take you waterward.  Its extreme downtown
is the battery, where that noble mole is washed by waves, and cooled
by breezes, which a few hours previous were out of sight of land.
Look at the crowds of water-gazers there.
```

In the sample, which is indeed the beginning of *Moby Dick*, the lines are broken more or less as they might be on the page, and paragraphs are indicated by a single blank line. If you want to deal with each paragraph as a unit, you need to break the text on the blank lines. Fortunately, this task is easy if you use the string `split()` method. Each newline character in a string can represented by `"\n"`. Naturally, the last line of a paragraph's text ends with a newline, and if the next line is blank, it's immediately followed by a second newline for the blank line:

**Reads all of file as a single string**

**Splits on two newlines together**

```
>>> moby_text = open("moby_01.txt").read()
>>> moby_paragraphs = moby_text.split("\n\n")
>>> print(moby_paragraphs[1])
There now is your insular city of the Manhattoes, belted round by wharves
as Indian isles by coral reefs--commerce surrounds it with her surf.
Right and left, the streets take you waterward.  Its extreme downtown
is the battery, where that noble mole is washed by waves, and cooled
by breezes, which a few hours previous were out of sight of land.
Look at the crowds of water-gazers there.
```

Splitting the text into paragraphs is a very simple first step in handling unstructured text. You might also need to do more normalization of the text before processing. Suppose that you want to count the rate of occurrence of every word in a text file. If you just split the file on whitespace, you get a list of words in the file. Counting their occurrences accurately will be hard, however, because *This*, *this*, *this.*, and *this,* are not the same. The way to make this code work is to normalize the text by removing the punctuation and making everything the same case before processing. For the example text above, the code for a normalized list of words might look like this:

**Reads all of the file as a single string**

**Makes everything lowercase**

```
>>> moby_text = open("moby_01.txt").read()
>>> moby_paragraphs = moby_text.split("\n\n")
>>> moby = moby_paragraphs[1].lower()
```

```
>>> moby = moby.replace(".", "")
>>> moby = moby.replace(",", "")
>>> moby_words = moby.split()
>>> print(moby_words)
['there', 'now', 'is', 'your', 'insular', 'city', 'of', 'the', 'manhattoes,',
     'belted', 'round', 'by', 'wharves', 'as', 'indian', 'isles', 'by',
     'coral', 'reefs--commerce', 'surrounds', 'it', 'with', 'her', 'surf',
     'right', 'and', 'left,', 'the', 'streets', 'take', 'you', 'waterward',
     'its', 'extreme', 'downtown', 'is', 'the', 'battery,', 'where', 'that',
     'noble', 'mole', 'is', 'washed', 'by', 'waves,', 'and', 'cooled', 'by',
     'breezes,', 'which', 'a', 'few', 'hours', 'previous', 'were', 'out',
     'of', 'sight', 'of', 'land', 'look', 'at', 'the', 'crowds', 'of',
     'water-gazers', 'there']
```

Removes periods

Removes commas

**QUICK CHECK: NORMALIZATION** Look closely at the list of words generated. Do you see any issues with the normalization so far? What other issues do you think you might encounter in a longer section of text? How do you think you might deal with those issues?

### 21.2.3 Delimited flat files

Although reading unstructured text files is easy, the downside is their very lack of structure. It's often much more useful to have some organization in the file to help with picking out individual values. The simplest way is to break the file into lines and have one element of information per line. You may have a list of the names of files to be processed, a list of people's names that need to be printed (on name tags, say), or maybe a series of temperature readings from a remote monitor. In such cases, the data parsing is very simple: You read in the line and convert it to the right type, if necessary. Then the file is ready to use.

Most of the time, however, things aren't not quite so simple. Usually, you need to group multiple related bits of information, and you need your code to read them in together. The common way to do this is to put the related pieces of information on the same line, separated by a special character. That way, as you read each line of the file, you can use the special characters to split the file into its different fields and put the values of those fields in variables for later processing.

This file is a simple example of temperature data in delimited format:

```
State|Month Day, Year Code|Avg Daily Max Air Temperature (F)|Record Count for
     Daily Max Air Temp (F)
Illinois|1979/01/01|17.48|994
Illinois|1979/01/02|4.64|994
Illinois|1979/01/03|11.05|994
Illinois|1979/01/04|9.51|994
Illinois|1979/05/15|68.42|994
Illinois|1979/05/16|70.29|994
Illinois|1979/05/17|75.34|994
Illinois|1979/05/18|79.13|994
Illinois|1979/05/19|74.94|994
```

This data is pipe-delimited, meaning that each field in the line is separated by the pipe (|) character, in this case giving you four fields: the state of the observations, the

date of the observations, the average high temperature, and the number of stations reporting. Other common delimiters are the tab character and the comma. The comma is perhaps the most common, but the delimiter could be any character you don't expect to occur in the values. (More about that issue next.) Comma delimiters are so common that this format is often called CSV (comma-separated values), and files of this type often have a .csv extension as a hint of their format.

Whatever character is being used as the delimiter, if you know what character it is, you can write your own code in Python to break each line into its fields and return them as a list. In the previous case, you can use the string `split()` method to break a line into a list of values:

```
>>> line = "Illinois|1979/01/01|17.48|994"
>>> print(line.split("|"))
['Illinois', '1979/01/01', '17.48', '994']
```

Note that this technique is very easy to do but leaves all the values as strings, which might not be convenient for later processing.

> **TRY THIS: READ A FILE** Write the code to read a text file (assume temp_data_pipes_00a.txt, as shown in the example), split each line of the file into a list of values, and add that list to a single list of records.
>
> What issues or problems did you encounter in implementing this code? How might you go about converting the last three fields to the correct date, real, and int types?

### 21.2.4 The csv module

If you need to do much processing of delimited data files, you should become familiar with the `csv` module and its options. When I've been asked to name my favorite module in the Python standard library, more than once I've cited the `csv` module—not because it's glamorous (it isn't), but because it has probably saved me more work and kept me from more self-inflicted bugs over my career than any other module.

The `csv` module is a perfect case of Python's "batteries included" philosophy. Although it's perfectly possible, and in many cases not even terribly hard, to roll your own code to read delimited files, it's even easier and much more reliable to use the Python module. The `csv` module has been tested and optimized, and it has features that you probably wouldn't bother to write if you had to do it yourself, but that are truly handy and time-saving when available.

Look at the previous data, and decide how you'd read it by using the `csv` module. The code to parse the data has to do two things: read each line and strip off the trailing newline character, and then break up the line on the pipe character and append that list of values to a list of lines. Your solution to the exercise might look something like this:

```
>>> results = []
>>> for line in open("temp_data_pipes_00a.txt"):
...     fields = line.strip().split("|")
```

```
...     results.append(fields)
...
>>> results
[['State', 'Month Day, Year Code', 'Avg Daily Max Air Temperature (F)',
     'Record Count for Daily Max Air Temp (F)'], ['Illinois', '1979/01/01',
     '17.48', '994'], ['Illinois', '1979/01/02', '4.64', '994'], ['Illinois',
     '1979/01/03', '11.05', '994'], ['Illinois', '1979/01/04', '9.51',
     '994'], ['Illinois', '1979/05/15', '68.42', '994'], ['Illinois',
     '1979/05/16', '70.29', '994'], ['Illinois', '1979/05/17', '75.34',
     '994'], ['Illinois', '1979/05/18', '79.13', '994'], ['Illinois',
     '1979/05/19', '74.94', '994']]
```

To do the same thing with the `csv` module, the code might be something like this:

```
>>> import csv
>>> results = [fields for fields in
     csv.reader(open("temp_data_pipes_00a.txt", newline=''), delimiter="|")]
>>> results
[['State', 'Month Day, Year Code', 'Avg Daily Max Air Temperature (F)',
     'Record Count for Daily Max Air Temp (F)'], ['Illinois', '1979/01/01',
     '17.48', '994'], ['Illinois', '1979/01/02', '4.64', '994'], ['Illinois',
     '1979/01/03', '11.05', '994'], ['Illinois', '1979/01/04', '9.51',
     '994'], ['Illinois', '1979/05/15', '68.42', '994'], ['Illinois',
     '1979/05/16', '70.29', '994'], ['Illinois', '1979/05/17', '75.34',
     '994'], ['Illinois', '1979/05/18', '79.13', '994'], ['Illinois',
     '1979/05/19', '74.94', '994']]
```

In this simple case, the gain over rolling your own code doesn't seem so great. Still, the code is two lines shorter and a bit clearer, and there's no need to worry about stripping off newline characters. The real advantages come when you want to deal with more challenging cases.

The data in the example is real, but it's actually been simplified and cleaned. The real data from the source is more complex. The real data has more fields, some fields are in quotes while others are not, and the first field is empty. The original is tab-delimited, but for the sake of illustration, I present it as comma-delimited here:

```
"Notes","State","State Code","Month Day, Year","Month Day, Year Code",Avg
     Daily Max Air Temperature (F),Record Count for Daily Max Air Temp
     (F),Min Temp for Daily Max Air Temp (F),Max Temp for Daily Max Air Temp
     (F),Avg Daily Max Heat Index (F),Record Count for Daily Max Heat Index
     (F),Min for Daily Max Heat Index (F),Max for Daily Max Heat Index
     (F),Daily Max Heat Index (F) % Coverage

,"Illinois","17","Jan 01, 1979","1979/01/
     01",17.48,994,6.00,30.50,Missing,0,Missing,Missing,0.00%
,"Illinois","17","Jan 02, 1979","1979/01/02",4.64,994,-
     6.40,15.80,Missing,0,Missing,Missing,0.00%
,"Illinois","17","Jan 03, 1979","1979/01/03",11.05,994,-
     0.70,24.70,Missing,0,Missing,Missing,0.00%
,"Illinois","17","Jan 04, 1979","1979/01/
     04",9.51,994,0.20,27.60,Missing,0,Missing,Missing,0.00%
,"Illinois","17","May 15, 1979","1979/05/
     15",68.42,994,61.00,75.10,Missing,0,Missing,Missing,0.00%
,"Illinois","17","May 16, 1979","1979/05/
     16",70.29,994,63.40,73.50,Missing,0,Missing,Missing,0.00%
```

```
,"Illinois","17","May 17, 1979","1979/05/
     17",75.34,994,64.00,80.50,82.60,2,82.40,82.80,0.20%
,"Illinois","17","May 18, 1979","1979/05/
     18",79.13,994,75.50,82.10,81.42,349,80.20,83.40,35.11%
,"Illinois","17","May 19, 1979","1979/05/
     19",74.94,994,66.90,83.10,82.87,78,81.60,85.20,7.85%
```

Notice that some fields include commas. The convention in that case is to put quotes around a field to indicate that it's not supposed to be parsed for delimiters. It's quite common, as here, to quote only some fields, especially those in which a value might contain the delimiter character. It also happens, as here, that some fields are quoted even if they're not likely to contain the delimiting character.

In a case like this one, your home-grown code becomes cumbersome. Now you can no longer split the line on the delimiting character; you need to be sure that you look only at delimiters that aren't inside quoted strings. Also, you need to remove the quotes around quoted strings, which might occur in any position or not at all. With the `csv` module, you don't need to change your code at all. In fact, because the comma is the default delimiter, you don't even need to specify it:

```
>>> results2 = [fields for fields in csv.reader(open("temp_data_01.csv",
     newline=''))]
>>> results2
[['Notes', 'State', 'State Code', 'Month Day, Year', 'Month Day, Year Code',
     'Avg Daily Max Air Temperature (F)', 'Record Count for Daily Max Air
     Temp (F)', 'Min Temp for Daily Max Air Temp (F)', 'Max Temp for Daily
     Max Air Temp (F)', 'Avg Daily Min Air Temperature (F)', 'Record Count
     for Daily Min Air Temp (F)', 'Min Temp for Daily Min Air Temp (F)', 'Max
     Temp for Daily Min Air Temp (F)', 'Avg Daily Max Heat Index (F)',
     'Record Count for Daily Max Heat Index (F)', 'Min for Daily Max Heat
     Index (F)', 'Max for Daily Max Heat Index (F)', 'Daily Max Heat Index
     (F) % Coverage'], ['', 'Illinois', '17', 'Jan 01, 1979', '1979/01/01',
     '17.48', '994', '6.00', '30.50', '2.89', '994', '-13.60', '15.80',
     'Missing', '0', 'Missing', 'Missing', '0.00%'], ['', 'Illinois', '17',
     'Jan 02, 1979', '1979/01/02', '4.64', '994', '-6.40', '15.80', '-9.03',
     '994', '-23.60', '6.60', 'Missing', '0', 'Missing', 'Missing', '0.00%'],
     ['', 'Illinois', '17', 'Jan 03, 1979', '1979/01/03', '11.05', '994', '-
     0.70', '24.70', '-2.17', '994', '-18.30', '12.90', 'Missing', '0',
     'Missing', 'Missing', '0.00%'], ['', 'Illinois', '17', 'Jan 04, 1979',
     '1979/01/04', '9.51', '994', '0.20', '27.60', '-0.43', '994', '-16.30',
     '16.30', 'Missing', '0', 'Missing', 'Missing', '0.00%'], ['',
     'Illinois', '17', 'May 15, 1979', '1979/05/15', '68.42', '994', '61.00',
     '75.10', '51.30', '994', '43.30', '57.00', 'Missing', '0', 'Missing',
     'Missing', '0.00%'], ['', 'Illinois', '17', 'May 16, 1979',
     '1979/05/16', '70.29', '994', '63.40', '73.50', '48.09', '994', '41.10',
     '53.00', 'Missing', '0', 'Missing', 'Missing', '0.00%'], ['',
     'Illinois', '17', 'May 17, 1979', '1979/05/17', '75.34', '994', '64.00',
     '80.50', '50.84', '994', '44.30', '55.70', '82.60', '2', '82.40',
     '82.80', '0.20%'], ['', 'Illinois', '17', 'May 18, 1979', '1979/05/18',
     '79.13', '994', '75.50', '82.10', '55.68', '994', '50.00', '61.10',
     '81.42', '349', '80.20', '83.40', '35.11%'], ['', 'Illinois', '17', 'May
     19, 1979', '1979/05/19', '74.94', '994', '66.90', '83.10', '58.59',
     '994', '50.90', '63.20', '82.87', '78', '81.60', '85.20', '7.85%']]
```

Notice that the extra quotes have been removed and that any field values with commas have the commas intact inside the fields—all without any more characters in the command.

**QUICK CHECK: HANDLING QUOTING** Consider how you'd approach the problems of handling quoted fields and embedded delimiter characters if you didn't have the `csv` library. Which would be easier to handle: the quoting or the embedded delimiters?

### 21.2.5 Reading a csv file as a list of dictionaries

In the preceding examples, you got a row of data back as a list of fields. This result works fine in many cases, but sometimes it may be handy to get the rows back as dictionaries where the field name is the key. For this use case, the `csv` library has a `DictReader`, which can take a list of fields as a parameter or can read them from the first line of the data. If you want to open the data with a `DictReader`, the code would look like this:

```
>>> results = [fields for fields in csv.DictReader(open("temp_data_01.csv",
     newline=''))]
>>> results[0]
OrderedDict([('Notes', ''), ('State', 'Illinois'), ('State Code', '17'),
     ('Month Day, Year', 'Jan 01, 1979'), ('Month Day, Year Code',
     '1979/01/01'), ('Avg Daily Max Air Temperature (F)', '17.48'), ('Record
     Count for Daily Max Air Temp (F)', '994'), ('Min Temp for Daily Max Air
     Temp (F)', '6.00'), ('Max Temp for Daily Max Air Temp (F)', '30.50'),
     ('Avg Daily Min Air Temperature (F)', '2.89'), ('Record Count for Daily
     Min Air Temp (F)', '994'), ('Min Temp for Daily Min Air Temp (F)', '-
     13.60'), ('Max Temp for Daily Min Air Temp (F)', '15.80'), ('Avg Daily
     Max Heat Index (F)', 'Missing'), ('Record Count for Daily Max Heat Index
     (F)', '0'), ('Min for Daily Max Heat Index (F)', 'Missing'), ('Max for
     Daily Max Heat Index (F)', 'Missing'), ('Daily Max Heat Index (F) %
     Coverage', '0.00%')])
```

Note that the `csv.DictReader` returns `OrderedDicts`, so the fields stay in their original order. Although their representation is a little different, the fields still behave like dictionaries:

```
>>> results[0]['State']
'Illinois'
```

If the data is particularly complex, and specific fields need to be manipulated, a `DictReader` can make it much easier to be sure you're getting the right field; it also makes your code somewhat easier to understand. Conversely, if your data set is quite large, you need to keep in mind that `DictReader` can take on the order of twice as long to read the same amount of data.

## 21.3 Excel files

The other common file format that I discuss in this chapter is the Excel file, which is the format that Microsoft Excel uses to store spreadsheets. I include Excel files here

because the way you end up treating them is very similar to the way you treat delimited files. In fact, because Excel can both read and write CSV files, the quickest and easiest way to extract data from an Excel spreadsheet file often is to open it in Excel and then save it as a CSV file. This procedure doesn't always make sense, however, particularly if you have a lot of files. In that case, even though you could theoretically automate the process of opening and saving each file in CSV format, it's probably faster to deal with the Excel files directly.

It's beyond the scope of this book to have an in-depth discussion of spreadsheet files, with their options for multiple sheets in the same file, macros, and various formatting options. Instead, in this section I look at an example of reading a simple one-sheet file simply to extract the data from it.

As it happens, Python's standard library doesn't have a module to read or write Excel files. To read that format, you need to install an external module. Fortunately, several modules are available to do the job. For this example, you use one called OpenPyXL, which is available from the Python package repository. You can install it with the following command from a command line:

```
$pip install openpyxl
```

Here's a view of the previous data, but in a spreadsheet:

| | A | B | C | D | E | F | G | H | I | J | K | L | M | N | C |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Notes | State | State Code | Month Day, Year | Month Day, Year Code | Avg Daily M | Record Co | Min Temp | Max Temp | Avg Daily M | Record Co | Min for Dai | Max for Da | Daily Max | Heat Inc |
| 2 | | Illinois | 17 | Jan 01, 1979 | 1979/01/01 | 17.48 | 994 | 6 | 30.5 | Missing | 0 | Missing | Missing | 0.00% | |
| 3 | | Illinois | 17 | Jan 02, 1979 | 1979/01/02 | 4.64 | 994 | -6.4 | 15.8 | Missing | 0 | Missing | Missing | 0.00% | |
| 4 | | Illinois | 17 | Jan 03, 1979 | 1979/01/03 | 11.05 | 994 | -0.7 | 24.7 | Missing | 0 | Missing | Missing | 0.00% | |
| 5 | | Illinois | 17 | Jan 04, 1979 | 1979/01/04 | 9.51 | 994 | 0.2 | 27.6 | Missing | 0 | Missing | Missing | 0.00% | |
| 6 | | Illinois | 17 | May 15, 1979 | 1979/05/15 | 68.42 | 994 | 61 | 75.1 | Missing | 0 | Missing | Missing | 0.00% | |
| 7 | | Illinois | 17 | May 16, 1979 | 1979/05/16 | 70.29 | 994 | 63.4 | 73.5 | Missing | 0 | Missing | Missing | 0.00% | |
| 8 | | Illinois | 17 | May 17, 1979 | 1979/05/17 | 75.34 | 994 | 64 | 80.5 | 82.6 | 2 | 82.4 | 82.8 | 0.20% | |
| 9 | | Illinois | 17 | May 18, 1979 | 1979/05/18 | 79.13 | 994 | 75.5 | 82.1 | 81.42 | 349 | 80.2 | 83.4 | 35.11% | |
| 10 | | Illinois | 17 | May 19, 1979 | 1979/05/19 | 74.94 | 994 | 66.9 | 83.1 | 82.87 | 78 | 81.6 | 85.2 | 7.85% | |
| 11 | | | | | | | | | | | | | | | |
| 12 | | | | | | | | | | | | | | | |

Reading the file is fairly simple, but it's still more work than CSV files require. First, you need to load the workbook; next, you need to get the specific sheet; then you can iterate over the rows; and from there, you extract the values of the cells. Some sample code to read the spreadsheet looks like this:

```
>>> from openpyxl import load_workbook
>>> wb = load_workbook('temp_data_01.xlsx')
>>> results = []
>>> ws = wb.worksheets[0]
>>> for row in ws.iter_rows():
...     results.append([cell.value for cell in row])
...
>>> print(results)
[['Notes', 'State', 'State Code', 'Month Day, Year', 'Month Day, Year Code',
     'Avg Daily Max Air Temperature (F)', 'Record Count for Daily Max Air
     Temp (F)', 'Min Temp for Daily Max Air Temp (F)', 'Max Temp for Daily
```

```
Max Air Temp (F)', 'Avg Daily Max Heat Index (F)', 'Record Count for
Daily Max Heat Index (F)', 'Min for Daily Max Heat Index (F)', 'Max for
Daily Max Heat Index (F)', 'Daily Max Heat Index (F) % Coverage'],
[None, 'Illinois', 17, 'Jan 01, 1979', '1979/01/01', 17.48, 994, 6,
30.5, 'Missing', 0, 'Missing', 'Missing', '0.00%'], [None, 'Illinois',
17, 'Jan 02, 1979', '1979/01/02', 4.64, 994, -6.4, 15.8, 'Missing', 0,
'Missing', 'Missing', '0.00%'], [None, 'Illinois', 17, 'Jan 03, 1979',
'1979/01/03', 11.05, 994, -0.7, 24.7, 'Missing', 0, 'Missing',
'Missing', '0.00%'], [None, 'Illinois', 17, 'Jan 04, 1979',
'1979/01/04', 9.51, 994, 0.2, 27.6, 'Missing', 0, 'Missing', 'Missing',
'0.00%'], [None, 'Illinois', 17, 'May 15, 1979', '1979/05/15', 68.42,
994, 61, 75.1, 'Missing', 0, 'Missing', 'Missing', '0.00%'], [None,
'Illinois', 17, 'May 16, 1979', '1979/05/16', 70.29, 994, 63.4, 73.5,
'Missing', 0, 'Missing', 'Missing', '0.00%'], [None, 'Illinois', 17,
'May 17, 1979', '1979/05/17', 75.34, 994, 64, 80.5, 82.6, 2, 82.4, 82.8,
'0.20%'], [None, 'Illinois', 17, 'May 18, 1979', '1979/05/18', 79.13,
994, 75.5, 82.1, 81.42, 349, 80.2, 83.4, '35.11%'], [None, 'Illinois',
17, 'May 19, 1979', '1979/05/19', 74.94, 994, 66.9, 83.1, 82.87, 78,
81.6, 85.2, '7.85%']]
```

This code gets you the same results as the much simpler code did for a csv file. It's not surprising that the code to read a spreadsheet is more complex, because spreadsheets are themselves much more complex objects. You should also be sure that you understand the way that data has been stored in the spreadsheet. If the spreadsheet contains formatting that has some significance, if labels need to be disregarded or handled differently, or if formulas and references need to be processed, you need to dig deeper into how those elements should be processed, and you need to write more-complex code.

Spreadsheets also often have other possible issues. At this writing, it's common for spreadsheets to be limited to around a million rows. Although that limit sounds large, more and more often you'll need to handle data sets that are larger. Also, spreadsheets sometimes automatically apply inconvenient formatting. One company I worked for had part numbers that consisted of a digit and at least one letter followed by some combination of digits and letters. It was possible to get a part number such as 1E20. Most spreadsheets automatically interpret 1E20 as scientific notation and save it as 1.00E+20 (1 times 10 to the $20^{th}$ power) while leaving 1F20 as a string. For some reason, it's rather difficult to keep this from happening, and particularly with a large data set, the problem won't be detected until farther down the pipeline, if all. For these reasons, I recommend using CSV or delimited files when at all possible. Users usually can save a spreadsheet as CSV, so there's usually no need put up with the extra complexity and formatting hassles that spreadsheets involve.

## 21.4 Data cleaning

One common problem you'll encounter in processing text-based data files is dirty data. By *dirty*, I mean that there are all sorts of surprises in the data, such as null values, values that aren't legal for your encoding, or extra whitespace. The data may also be unsorted or in an order that makes processing difficult. The process of dealing with situations like these is called *data cleaning*.

### 21.4.1 Cleaning

In a very simple example data clean, you might need to process a file that was exported from a spreadsheet or other financial program, and the columns dealing with money may have percentage and currency symbols (such as %, $, £, and ?), as well as extra groupings that use a period or comma. Data from other sources may have other surprises that make processing tricky if they're not caught in advance. Look again at the temperature data you saw previously. The first data line looks like this:

```
[None, 'Illinois', 17, 'Jan 01, 1979', '1979/01/01', 17.48, 994, 6, 30.5,
    2.89, 994, -13.6, 15.8, 'Missing', 0, 'Missing', 'Missing', '0.00%']
```

Some columns, such as `'State'` (field 2) and `'Notes'` (field 1), are clearly text, and you wouldn't be likely to do much with them. There are also two date fields in different formats, and you might well want to do calculations with the dates, possibly to change the order of the data and to group rows by month or day, or possibly to calculate how far apart in time two rows are.

The rest of the fields seem to be different types of numbers; the temperatures are decimals, and the record counts columns are integers. Notice, however, that the heat index temperatures have a variation: When the value for the `'Max Temp for Daily Max Air Temp (F)'` field is below 80, the values for the heat index fields aren't reported, but instead are listed as `'Missing'`, and the record count is 0. Also note that the `'Daily Max Heat Index (F) % Coverage'` field is expressed as a percentage of the number of temperature records that also qualify to have a heat index. Both of these issues will be problematic if you want to do any math calculations on the values in those fields, because both `'Missing'` and any number ending with % will be parsed as strings, not numbers.

Cleaning data like this can be done at different steps in the process. Quite often, I prefer to clean the data as it's being read from the file, so I might well replace the `'Missing'` with a None value or an empty string as the lines are being processed. You could also leave the `'Missing'` strings in place and write your code so that no math operations are performed on a value if it is `'Missing'`.

> **TRY THIS: CLEANING DATA** How would you handle the fields with `'Missing'` as possible values for math calculations? Can you write a snippet of code that averages one of those columns?
>
> What would you do with the average column at the end so that you could also report the average coverage? In your opinion, would the solution to this problem be at all linked to the way that the `'Missing'` entries were handled?

### 21.4.2 Sorting

As I mentioned earlier, it's often useful to have data in the text file sorted before processing. Sorting the data makes it easier to spot and handle duplicate values, and it can also help bring together related rows for quicker or easier processing. In one case, I received a 20 million–row file of attributes and values, in which arbitrary numbers of

them needed to be matched with items from a master SKU list. Sorting the rows by the item ID made gathering each item's attributes much faster. How you do the sorting depends on the size of the data file relative to your available memory and on the complexity of the sort. If all the lines of the file can fit comfortably into available memory, the easiest thing may be to read all of the lines into a list and use the list's sort method:

```
>>> lines = open("datafile").readlines()
>>> lines.sort()
```

You could also use the `sorted()` function, as in `sorted_lines = sorted(lines)`. This function preserves the order of the lines in your original list, which usually is unnecessary. The drawback to using the `sorted()` function is that it creates a new copy of the list. This process takes slightly longer and consumes twice as much memory, which might be a bigger concern.

If the data set is larger than memory and the sort is very simple (just by an easily grabbed field), it may be easier to use an external utility, such as the UNIX `sort` command, to preprocess the data:

```
$ sort data > data.srt
```

In either case, sorting can be done in reverse order and can be keyed by values, not the beginning of the line. For such occasions, you need to study the documentation of the sorting tool you choose to use. A simple example in Python would be to make a sort of lines of text case-insensitive. To do this, you give the `sort` method a key function that makes the element lowercase before making a comparison:

```
>>> lines.sort(key=str.lower)
```

This example uses a `lambda` function to ignore the first five characters of each string:

```
>>> lines.sort(key=lambda x: x[5:])
```

Using key functions to determine the behavior of sorts in Python is very handy, but be aware that the key function is called a lot in the process of sorting, so a complex key function could mean a real performance slowdown, particularly with a large data set.

### 21.4.3 Data cleaning issues and pitfalls

It seems that there are as many types of dirty data as there are sources and use cases for that data. Your data will always have quirks that do everything from making processing less accurate to making it impossible to even load the data. As a result, I can't provide an exhaustive list of the problems you might encounter and how to deal with them, but I can give you some general hints.

- *Beware of whitespace and null characters.* The problem with whitespace characters is that you can't see them, but that doesn't mean that they can't cause troubles. Extra whitespace at the beginning and end of data lines, extra whitespace around individual fields, and tabs instead of spaces (or vice versa) can all make

your data loading and processing more troublesome, and these problems aren't always easily apparent. Similarly, text files with null characters (ASCII 0) may seem okay on inspection but break on loading and processing.

- *Beware punctuation.* Punctuation can also be a problem. Extra commas or periods can mess up CSV files and the processing of numeric fields, and unescaped or unmatched quote characters can also confuse things.
- *Break down and debug the steps.* It's easier to debug a problem if each step is separate, which means putting each operation on a separate line, being more verbose, and using more variables. But the work is worth it. For one thing, it makes any exceptions that are raised easier to understand, and it also makes debugging easier, whether with print statements, logging, or the Python debugger. It may also be helpful to save the data after each step and to cut the file size to just a few lines that cause the error.

## 21.5 Writing data files

The last part of the ETL process may involve saving the transformed data to a database (which I discuss in chapter 22), but often it involves writing the data to files. These files may be used as input for other applications and analysis, either by people or by other applications. Usually, you have a particular file specification listing what fields of data should be included, what they should be named, what format and constraints there should be for each, and so on

### 21.5.1 CSV and other delimited files

Probably the easiest thing of all is to write your data to CSV files. Because you've already loaded, parsed, cleaned, and transformed the data, you're unlikely to hit any unresolved issues with the data itself. And again, using the `csv` module from the Python standard library makes your work much easier.

Writing delimited files with the `csv` module is pretty much the reverse of the read process. Again, you need to specify the delimiter that you want to use, and again, the `csv` module takes care of any situations in which your delimiting character is included in a field:

```
>>> temperature_data = [['State', 'Month Day, Year Code', 'Avg Daily Max Air
     Temperature (F)', 'Record Count for Daily Max Air Temp (F)'],
     ['Illinois', '1979/01/01', '17.48', '994'], ['Illinois', '1979/01/02',
     '4.64', '994'], ['Illinois', '1979/01/03', '11.05', '994'], ['Illinois',
     '1979/01/04', '9.51', '994'], ['Illinois', '1979/05/15', '68.42',
     '994'], ['Illinois', '1979/05/16', '70.29', '994'], ['Illinois',
     '1979/05/17', '75.34', '994'], ['Illinois', '1979/05/18', '79.13',
     '994'], ['Illinois', '1979/05/19', '74.94', '994']]
>>> csv.writer(open("temp_data_03.csv", "w",
     newline='')).writerows(temperature_data)
```

This code results in the following file:

```
State,"Month Day, Year Code",Avg Daily Max Air Temperature (F),Record Count
     for Daily Max Air Temp (F)
```

```
Illinois,1979/01/01,17.48,994
Illinois,1979/01/02,4.64,994
Illinois,1979/01/03,11.05,994
Illinois,1979/01/04,9.51,994
Illinois,1979/05/15,68.42,994
Illinois,1979/05/16,70.29,994
Illinois,1979/05/17,75.34,994
Illinois,1979/05/18,79.13,994
Illinois,1979/05/19,74.94,994
```

Just as when reading from a CSV file, it's possible to write dictionaries instead of lists if you use a `DictWriter`. If you do use a `DictWriter`, be aware of a couple of points: You must specify the fields names in a list when you create the writer, and you can use the `DictWriter`'s `writeheader` method to write the header at the top of the file. So assume that you have the same data as previously, but in dictionary format:

```
{'State': 'Illinois', 'Month Day, Year Code': '1979/01/01', 'Avg Daily Max
     Air Temperature (F)': '17.48', 'Record Count for Daily Max Air Temp
     (F)': '994'}
```

You can use a `DictWriter` object from the `csv` module to write each row, a dictionary, to the correct fields in the CSV file:

```
>>> fields = ['State', 'Month Day, Year Code', 'Avg Daily Max Air Temperature
     (F)', 'Record Count for Daily Max Air Temp (F)']
>>> dict_writer = csv.DictWriter(open("temp_data_04.csv", "w"),
     fieldnames=fields)
>>> dict_writer.writeheader()
>>> dict_writer.writerows(data)
>>> del dict_writer
```

### 21.5.2 Writing Excel files

Writing spreadsheet files is unsurprisingly similar to reading them. You need to create a workbook, or spreadsheet file; then you need to create a sheet or sheets; and finally, you write the data in the appropriate cells. You could create a new spreadsheet from your CSV data file like this:

```
>>> from openpyxl import Workbook
>>> data_rows = [fields for fields in csv.reader(open("temp_data_01.csv"))]
>>> wb = Workbook()
>>> ws = wb.active
>>> ws.title = "temperature data"
>>> for row in data_rows:
...     ws.append(row)
...
>>> wb.save("temp_data_02.xlsx")
```

It's also possible to add formatting to cells as you write them to the spreadsheet file. For more on how to add formatting, please refer to the `xlswriter` documentation.

### 21.5.3 Packaging data files

If you have several related data files, or if your files are large, it may make sense to package them in a compressed archive. Although various archive formats are in use, the zip file remains popular and almost universally accessible to users on almost every platform. For hints on how to create zip-file packages of your data files, please refer to chapter 20.

**LAB 21: WEATHER OBSERVATIONS** The file of weather observations provided here is by month and then by county for the state of Illinois from 1979 to 2011. Write the code to process this file to extract the data for Chicago (Cook County) into a single CSV or spreadsheet file. This process includes replacing the `'Missing'` strings with empty strings and translating the percentage to a decimal. You may also consider what fields are repetitive (and therefore can be omitted or stored elsewhere). The proof that you've got it right occurs when you load the file into a spreadsheet. You can download a solution with the book's source code.

## Summary

- ETL (extract-transform-load) is the process of getting data from one format, making sure that it's consistent, and then putting it in a format you can use. ETL is the basic step in most data processing.
- Encoding can be problematic with text files, but Python lets you deal with some encoding problems when you load files.
- Delimited or CSV files are common, and the best way to handle them is with the `csv` module.
- Spreadsheet files can be more complex than CSV files but can be handled much the same way.
- Currency symbols, punctuation, and null characters are among the most common data cleaning issues; be on the watch for them.
- Presorting your data file can make other processing steps faster.

# *Exploring data*

In order to choose and construct useful models in a data science project you need to get to know the data. What bits of data do you have? What problems might there be? Given the nature of the data, what approaches might work best?

In the last chapter of *The Quick Python Book, 3rd edition* I introduce a key Python tool for exploring data, Jupyter notebook. Based on an earlier project called IPython, Jupyter notebooks are a daily staple for many data professionals.

It's hard to explain all of the features that make Jupyter notebooks so useful for data exploration – they execute Python code, they make testing easy to write chunks of Python, then execute them on your data, see what the result is, and adapt it as needed. Jupyter works well with the excellent Python data handling framework, Pandas, as well as various graphing tools. Jupyter notebooks are web-based, it's to share them with others, and they can even be used as a presentation tool.

With all of these features in mind, I've selected this chapter to give an idea of ways to explore data as part of a data science project.

Chapter 24 from *The Quick Python Book, 3rd edition* by Naomi Ceder.

# *Exploring data*

***This chapter covers***

- Python's advantages for handling data
- Jupyter Notebook
- pandas
- Data aggregation
- Plots with matplotlib

Over the past few chapters, I've dealt with some aspects of using Python to get and clean data. Now it's time to look at a few of the things that Python can help you do to manipulate and explore data.

## 24.1 Python tools for data exploration

In this chapter, we'll look at some common Python tools for data exploration: Jupyter notebook, pandas, and matplotlib. I can only touch briefly on a few features of these tools, but the aim is to give you an idea of what is possible and some initial tools to use in exploring data with Python.

### 24.1.1 Python's advantages for exploring data

Python has become one of the leading languages for data science and continues to grow in that area. As I've mentioned, however, Python isn't always the fastest language in terms of raw performance. Conversely, some data-crunching libraries, such as NumPy, are largely written in C and heavily optimized to the point that speed isn't an issue. In addition, considerations such as readability and accessibility often outweigh pure speed; minimizing the amount of developer time needed is often more important. Python is readable and accessible, and both on its own and in combination with tools developed in the Python community, it's an enormously powerful tool for manipulating and exploring data.

### 24.1.2 Python can be better than a spreadsheet

Spreadsheets have been the tools of choice for ad-hoc data manipulation for decades. People who are skilled with spreadsheets can make them do truly impressive tricks: spreadsheets can combine different but related data sets, pivot tables, use lookup tables to link data sets, and much more. But although people everywhere get a vast amount of work done with them every day, spreadsheets do have limitations, and Python can help you go beyond those limitations.

One limitation that I've already alluded to is the fact that most spreadsheet software has a row limit—currently, about 1 million rows, which isn't enough for many data sets. Another limitation is the central metaphor of the spreadsheet itself. Spreadsheets are two-dimensional grids, rows and columns, or at best stacks of grids, which limits the ways you can manipulate and think about complex data.

With Python, you can code your way around the limitations of spreadsheets and manipulate data the way you want. You can combine Python data structures such as lists, tuples, sets, and dictionaries in endlessly flexible ways, or you can create your own classes to package both data and behavior exactly the way you need.

## 24.2 Jupyter notebook

Probably one of the most compelling tools for exploring data with Python doesn't augment what the language itself does, but changes the way you use the language to interact with your data. Jupyter notebook is a web application that allows you to create and share documents that contain live code, equations, visualizations, and explanatory text. Although several other languages are now supported, it originated in connection with IPython, an alternative shell for Python developed by the scientific community.

What makes Jupyter such a convenient and powerful tool is the fact that you interact with it in a web browser. It lets you combine text and code, as well as modify and execute your code interactively. You can not only run and modify code in chunks, but also save and share the notebooks with others.

The best way to get a feel for what Jupyter notebook can do is start playing with it. It's fairly easy to run a Jupyter process locally on your machine, or you can access online versions. For some options, see the sidebar on ways to run Jupyter.

**Ways to run Jupyter**

*Jupyter online:* Accessing online instances of Jupyter is one of the easiest ways to get started. Currently, Project Jupyter, the community behind Jupyter, hosts free notebooks at https://jupyter.org/try. You can also find demo notebooks and kernels for other languages. At this writing, you can also access free notebooks on Microsoft's Azure platform at https://notebooks.azure.com, and many other ways are available.

*Jupyter locally:* Although using an online instance is quite convenient, it's not very much work to set up your own instance of Jupyter on your computer. Usually for local versions, you point your browser to localhost:8888.

If you use Docker, you have several containers to choose among. To run the data science notebook container, use something like this:

```
docker run -it --rm -p 8888:8888 jupyter/datascience-notebook
```

If you'd rather run directly on your system, it's easy to install and run Jupyter in a virtualenv.

*macOS and Linux systems:* First, open a command window, and enter the following commands:

```
> python3 -m venv jupyter
> cd jupyter
> source bin/activate
> pip install jupyter
> jupyter-notebook
```

Windows systems:

```
> python3 -m venv jupyter
> cd jupyter
> Scripts/bin/activate
> pip install jupyter
> Scripts/jupyter-notebook
```

The last command should run the Jupyter notebook web app and open a browser window pointing at it.

### 24.2.1 Starting a kernel

When you have Jupyter installed, running, and open in your browser, you need to start a Python kernel. One nice thing about Jupyter is that it lets you run multiple kernels at the same time. You can run kernels for different versions of Python and for other languages such as R, Julia, and even Ruby.


Figure 24.1 Starting a Python kernel

Starting a kernel is easy. Just click the new button and select Python 3 (figure 24.1).

### 24.2.2 Executing code in a cell

When you have a kernel running, you can start entering and running Python code. Right away, you'll notice a few differences from the ordinary Python command shell. You won't get the >>> prompt that you see in the standard Python shell, and pressing Enter just adds new lines in the cell. To execute the code in a cell, illustrated in figure 24.2, choose Cell > Run Cells, click the Run button immediately to the left of the down arrow on the button bar, or use the key combination Alt-Enter. After you use Jupyter notebook a little bit, it's quite likely that the Alt-Enter key combination will become quite natural to you.

You can test how it works by entering some code or an expression into the first cell of your new notebook and then pressing Alt-Enter.

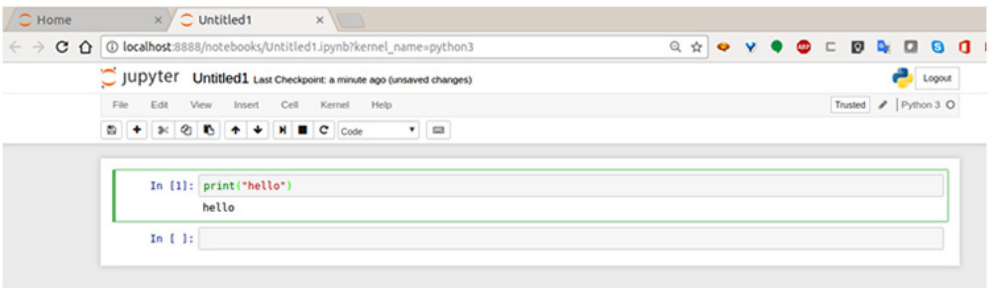



Figure 24.2 Executing code in a notebook cell

As you can see, any output is shown immediately below the cell, and a new cell is created and ready for your next input. Also note that each cell that's executed is numbered in the order in which it's executed.

**TRY THIS: USING JUPYTER NOTEBOOK** Enter some code in the notebook and experiment with running it. Check out the Edit, Cell, and Kernel menus to see what options are there. When you have a little code running, use the Kernel menu to restart the kernel, repeat your steps, and then use the Cell menu to rerun the code in all of the cells.

## 24.3 Python and pandas

In the course of exploring and manipulating data, you perform quite a few common operations, such as loading data into a list or dictionary, cleaning data, and filtering data. Most of these operations are repeated often, have to be done in standard patterns, and are simple and often tedious. If you think that this combination is a strong reason to automate those tasks you're not alone. One of the now-standard tools for handling data in Python—pandas—was created to automate the boring heavy lifting of handling data sets.

### 24.3.1 Why you might want to use pandas

pandas was created to make manipulating and analyzing tablular or relational data easy by providing a standard framework for holding the data, with convenient tools for frequent operations. As a result, it's almost more of an extension to Python than a library, and it changes the way you can interact with data. The plus side is that after you grok how pandas work, you can do some impressive things and save a lot of time. It does take time to learn how to get the most from pandas, however. As with many tools, if you use pandas for what it was designed for, it excels. The simple examples I show you in the following sections should give you a rough idea whether pandas is a tool that's suited for your use cases.

### 24.3.2 Installing pandas

pandas is easy to install with `pip`. It's often used along with matplotlib for plotting, so you can install both tools from the command line of your Jupyter virtual environment with this code:

```
> pip install pandas matplotlib
```

From a cell in a Jupyter notebook, you can use

```
In [ ]: !pip install pandas matplotlib
```

If you use pandas, life will be easier if you use the following three lines:

```
%matplotlib inline
import pandas as pd
import numpy as np
```

The first line is a Jupyter "magic" function that enables matplotlib to plot data in the cell where your code is (which is very useful). The second line imports pandas with the alias of pd, which is both easier to type and common among pandas users; the last

line also imports numpy. Although pandas depends quite a bit on numpy, you won't use it explicitly in the following examples, but it's reasonable to get into the habit of importing it anyway.

### 24.3.3 Data frames

One basic structure that you get with pandas is a data frame. A *data frame* is a two-dimensional grid, rather similar to a relational database table except in memory. Creating a data frame is easy; you give it some data. To keep things absolutely simple, give it a 3 x 3 grid of numbers as the first example. In Python, such a grid is a list of lists:

```
grid = [[1,2,3], [4,5,6], [7,8,9]]
print(grid)

[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Sadly, in Python the grid won't look like a grid unless you make some additional effort. So see what you can do with the same grid as a pandas data frame:

```
import pandas as pd
df = pd.DataFrame(grid)
print(df)

   0  1  2
0  1  2  3
1  4  5  6
2  7  8  9
```

That code is fairly straightforward; all you needed to do was turn your grid into a data frame. You've gained a more gridlike display, and now you have both row and column numbers. It's often rather bothersome to keep track of what column number is what, of course, so give your columns names:

```
df = pd.DataFrame(grid, columns=["one", "two", "three"] )
print(df)

   one  two  three
0    1    2      3
1    4    5      6
2    7    8      9
```

You may wonder whether naming the columns has any benefit, but the column names can be put to use with another pandas trick: the ability to select columns by name. If you want the contents only of column `"two"`, for example, you can get it very simply:

```
print(df["two"])
0    2
1    5
2    8
Name: two, dtype: int64
```

Here, you've already saved time in comparison to Python. To get only column two of your grid, you'd need to use a list comprehension while also remembering to use a zero-based index (and you still wouldn't get the nice output):

```
print([x[1] for x in grid])
[2, 5, 8]
```

You can loop over data frame column values just as easily as the list you got by using a comprehension:

```
for x in df["two"]:
    print(x)
2
5
8
```

That's not bad for a start, but by using a list of columns in double brackets, you can do better, getting a subset of the data frame that's another data frame. Instead of getting the middle column, get the first and last columns of your data frame as another data frame:

```
edges = df[["one", "three"]]
print(edges)
   one  three
0    1      3
1    4      6
2    7      9
```

A data frame also has several methods that apply the same operation and argument to every item in the frame. If you want to add two to every item in the data frame's edges, you could use the `add()` method:

```
print(edges.add(2))
   one  three
0    3      5
1    6      8
2    9     11
```

Here again, it's possible to get the same result by using list comprehensions and/or nested loops, but those techniques aren't as convenient. It's pretty easy to see how such functionality can make life easier, particularly for someone who's more interested in the information that the data contains than in the process of manipulating it.

## 24.4 Data cleaning

In earlier chapters, I discussed a few ways to use Python to clean data. Now that I've added pandas to the mix, I'll show you examples of how to use its functionality to clean data. As I present the following operations, I also refer to ways that the same operation might be done in plain Python, both to illustrate how using pandas is different and to show why pandas isn't right for every use case (or user, for that matter).

### 24.4.1 Loading and saving data with pandas

pandas has an impressive collection of methods to load data from different sources. It supports several file formats (including fixed-width and delimited text files, spreadsheets, JSON, XML, and HTML), but it's also possible to read from SQL databases, Google BiqQuery, HDF, and even clipboard data. You should be aware that many of these operations aren't actually part of pandas itself; pandas relies on having other libraries installed to handle those operations, such as SQLAlchemy for reading from SQL databases. This distinction matters mostly if something goes wrong; quite often, the problem that needs to be fixed is outside pandas, and you're left to deal with the underlying library.

Reading a JSON file with the `read_json()` method is simple:

```
mars = pd.read_json("mars_data_01.json")
```

This code gives you a data frame like this:

```
                                 report
abs_humidity                       None
atmo_opacity                      Sunny
ls                                  296
max_temp                             -1
max_temp_fahrenheit                30.2
min_temp                            -72
min_temp_fahrenheit               -97.6
pressure                            869
pressure_string                  Higher
season                         Month 10
sol                                1576
sunrise             2017-01-11T12:31:00Z
sunset              2017-01-12T00:46:00Z
terrestrial_date              2017-01-11
wind_direction                        --
wind_speed                         None
```

For another example of how simple reading data into pandas is, load some data from the CSV file of temperature data from chapter 21 and from the JSON file of Mars weather data used in chapter 22. In the first case, use the `read_csv()` method:

```
temp = pd.read_csv("temp_data_01.csv")
```

**Note that the \ at the end of the header line is an indication that the table is too long to be printed on one line and more columns are printed below.**

```
          4      5    6     7     8      9    10    11    12      13    14 \

0  1979/01/01  17.48  994   6.0  30.5   2.89  994 -13.6  15.8     NaN     0
1  1979/01/02   4.64  994  -6.4  15.8  -9.03  994 -23.6   6.6     NaN     0
2  1979/01/03  11.05  994  -0.7  24.7  -2.17  994 -18.3  12.9     NaN     0
3  1979/01/04   9.51  994   0.2  27.6  -0.43  994 -16.3  16.3     NaN     0
4  1979/05/15  68.42  994  61.0  75.1  51.30  994  43.3  57.0     NaN     0
5  1979/05/16  70.29  994  63.4  73.5  48.09  994  41.1  53.0     NaN     0
6  1979/05/17  75.34  994  64.0  80.5  50.84  994  44.3  55.7   82.60     2
```

```
7  1979/05/18  79.13  994  75.5  82.1  55.68  994  50.0  61.1  81.42  349
8  1979/05/19  74.94  994  66.9  83.1  58.59  994  50.9  63.2  82.87   78

     15    16      17
0   NaN   NaN  0.0000
1   NaN   NaN  0.0000
2   NaN   NaN  0.0000
3   NaN   NaN  0.0000
4   NaN   NaN  0.0000
5   NaN   NaN  0.0000
6  82.4  82.8  0.0020
7  80.2  83.4  0.3511
8  81.6  85.2  0.0785
```

Clearly, loading the file in a single step is appealing, and you can see that pandas had no issues loading the file. You can also see that the empty first column has been translated into `NaN` (not a number). You do still have the same issue with `'Missing'` for some values, and in fact it might make sense to have those `'Missing'` values converted to `NaN`:

```
temp = pd.read_csv("temp_data_01.csv", na_values=['Missing'])
```

The addition of the `na_values` parameter controls what values will be translated to `NaN` on load. In this case, you added the string `'Missing'` so that the row of the data frame was translated from

```
NaN  Illinois  17  Jan 01, 1979  1979/01/01  17.48  994  6.0  30.5  2.89994
     -13.6  15.8  Missing  0  Missing  Missing  0.00%
```

to

```
NaN  Illinois  17  Jan 01, 1979  1979/01/01  17.48  994  6.0  30.5  2.89994
     -13.6  15.8  NaN0  NaN  NaN  0.00%
```

This technique can be particularly useful if you have one of those data files in which, for whatever reason, "no data" is indicated in a variety of ways: `NA`, `N/A`, `?`, `-`, and so on. To handle a case like that, you can inspect the data to find out what's used and then reload it, using the `na_values` parameter to standardize all those variations as `NaN`.

### SAVING DATA

If you want to save the contents of a data frame, a pandas data frame has a similarly broad collection of methods. If you take your simple grid data frame, you can write it in several ways. This line

```
df.to_csv("df_out.csv", index=False)
```

**Setting index to False means that the row indexes will not be written.**

writes a file that looks like this:

```
one,two,three
1,2,3
4,5,6
7,8,9
```

Similarly, you can transform a data grid to a JSON object or write it to a file:

Supplying a file path as an argument writes the JSON to that file rather than returning it.

```
df.to_json()
'{"one":{"0":1,"1":4,"2":7},"two":{"0":2,"1":5,"2":8},"three":{"0":3,"1":6,"2
    ":9}}'
```

### 24.4.2 Data cleaning with a data frame

Converting a particular set of values to `NaN` on load is a very simple bit of data cleaning that pandas makes trivial. Going beyond that, data frames support several operations that can make data cleaning less of a chore. To see how this works, reopen the temperature CSV file, but this time, instead of using the headers to name the columns, use the `range()` function with the `names` parameter to give them numbers, which will make referring to them easier. You also may recall from an earlier example that the first field of every line—the `"Notes"` field—is empty and loaded with `NaN` values. Although you could ignore this column, it would be even easier if you didn't have it. You can use the `range()` function again, this time starting from 1, to tell pandas to load all columns except the first one. But if you know that all of your values are from Illinois and you don't care about the long-form date field, you could start from 4 to make things much more manageable:

Setting header=0 turns off reading the header for column labels.

```
temp = pd.read_csv("temp_data_01.csv", na_values=['Missing'], header=0,
    names=range(18), usecols=range(4,18))
print(temp)
```

```
           4      5    6     7     8      9    10    11    12     13   14  \
0  1979/01/01  17.48  994   6.0  30.5   2.89  994 -13.6  15.8    NaN    0
1  1979/01/02   4.64  994  -6.4  15.8  -9.03  994 -23.6   6.6    NaN    0
2  1979/01/03  11.05  994  -0.7  24.7  -2.17  994 -18.3  12.9    NaN    0
3  1979/01/04   9.51  994   0.2  27.6  -0.43  994 -16.3  16.3    NaN    0
4  1979/05/15  68.42  994  61.0  75.1  51.30  994  43.3  57.0    NaN    0
5  1979/05/16  70.29  994  63.4  73.5  48.09  994  41.1  53.0    NaN    0
6  1979/05/17  75.34  994  64.0  80.5  50.84  994  44.3  55.7  82.60    2
7  1979/05/18  79.13  994  75.5  82.1  55.68  994  50.0  61.1  81.42  349
8  1979/05/19  74.94  994  66.9  83.1  58.59  994  50.9  63.2  82.87   78

     15    16      17
0   NaN   NaN   0.00%
1   NaN   NaN   0.00%
2   NaN   NaN   0.00%
3   NaN   NaN   0.00%
4   NaN   NaN   0.00%
5   NaN   NaN   0.00%
6  82.4  82.8   0.20%
7  80.2  83.4  35.11%
8  81.6  85.2   7.85%
```

Now you have a data frame that has only the columns you might want to work with. But you still have an issue: the last column, which lists the percentage of coverage for

the heat index, is still a string ending with a percentage sign rather than an actual percentage. This problem is apparent if you look at the first row's value for column 17:

```
temp[17][0]
'0.00%'
```

To fix this problem, you need to do two things: Remove the `%` from the end of the value and then cast the value from string to a number. Optionally, if you want to represent the resulting percentage as a fraction, you need to divide it by 100. The first bit is simple because pandas lets you use a single command to repeat an operation on a column:

```
temp[17] = temp[17].str.strip("%")
temp[17][0]
'0.00'
```

This code takes the column and calls a string `strip()` operation on it to remove the trailing `%`. Now when you look at the first value in the column (or any of the other values), you see that the offending percentage sign is gone. It's also worth noting that you could have used other operations, such as `replace("%", "")`, to achieve the same result.

The second operation is to convert the string to a numeric value. Again, pandas lets you perform this operation with one command:

```
temp[17] = pd.to_numeric(temp[17])
temp[17][0]
0.0
```

Now the values in column 17 are numeric, and if you want to, you can use the `div()` method to finish the job of turning those values into fractions:

```
temp[17] = temp[17].div(100)
temp[17]

0    0.0000
1    0.0000
2    0.0000
3    0.0000
4    0.0000
5    0.0000
6    0.0020
7    0.3511
8    0.0785
Name: 17, dtype: float64
```

In fact, it would be possible to achieve the same result in a single line by chaining the three operations together:

```
temp[17] = pd.to_numeric(temp[17].str.strip("%")).div(100)
```

This example is very simple, but it gives you an idea of the convenience that pandas can bring to cleaning your data. pandas has a wide variety of operations for transforming

data, as well as the ability to use custom functions, so it would be hard to think of a scenario in which you couldn't streamline data cleaning with pandas.

Although the number of options is almost overwhelming, a wide variety of tutorials and videos is available, and the documentation at http://pandas.pydata.org is excellent.

**TRY THIS: CLEANING DATA WITH AND WITHOUT PANDAS** Experiment with the operations. When the final column has been converted to a fraction, can you think of a way to convert it back to a string with the trailing percentage sign?

By contrast, load the same data into a plain Python list by using the `csv` module, and apply the same changes by using plain Python.

## 24.5 Data aggregation and manipulation

The preceding examples probably gave you some idea of the many options pandas gives you for performing fairly complex operations on your data with only a few commands. As you might expect, this level of functionality is also available for aggregating data. In this section, I walk through a few simple examples of aggregating data to illustrate some of the many possibilities. Although many options are available, I focus on merging data frames, performing simple data aggregation, and grouping and filtering.

### 24.5.1 Merging data frames

Quite often in the course of handling data, you need to relate two data sets. Suppose that you have one file containing the number of sales calls made per month by members of a sales team, and in another file, you have the dollar amounts of the sales in each of their territories:

```
calls = pd.read_csv("sales_calls.csv")
print(calls)
```

```
   Team member  Territory  Month  Calls
0        Jorge          3      1    107
1        Jorge          3      2     88
2        Jorge          3      3     84
3        Jorge          3      4    113
4          Ana          1      1     91
5          Ana          1      2    129
6          Ana          1      3     96
7          Ana          1      4    128
8          Ali          2      1    120
9          Ali          2      2     85
10         Ali          2      3     87
11         Ali          2      4     87
```

```
revenue = pd.read_csv("sales_revenue.csv")
print(revenue)
```

```
    Territory  Month  Amount
0           1      1   54228
1           1      2   61640
2           1      3   43491
```

```
3           1       4   52173
4           2       1   36061
5           2       2   44957
6           2       3   35058
7           2       4   33855
8           3       1   50876
9           3       2   57682
10          3       3   53689
11          3       4   49173
```

Clearly, it would be very useful to link revenue and team-member activity. These two files are very simple, yet merging them with plain Python isn't entirely trivial. pandas has a function to merge two data frames:

```
calls_revenue = pd.merge(calls, revenue, on=['Territory', 'Month'])
```

The `merge` function creates a new data frame by joining the two frames on the columns specified in the column field. The `merge` function works similarly to a relational-database join, giving you a table that combines the columns from the two files:

```
print(calls_revenue)
   Team member  Territory  Month  Calls  Amount
0        Jorge          3      1    107   50876
1        Jorge          3      2     88   57682
2        Jorge          3      3     84   53689
3        Jorge          3      4    113   49173
4          Ana          1      1     91   54228
5          Ana          1      2    129   61640
6          Ana          1      3     96   43491
7          Ana          1      4    128   52173
8          Ali          2      1    120   36061
9          Ali          2      2     85   44957
10         Ali          2      3     87   35058
11         Ali          2      4     87   33855
```

In this case, you have a one-to-one correspondence between the rows in the two fields, but the `merge` function can also do one-to-many and many-to-many joins, as well as right and left joins.

**QUICK CHECK: MERGING DATA SETS** How would you go about merging to data sets like the ones in the Python example?

### 24.5.2 Selecting data

It can also be useful to select or filter the rows in a data frame based on some condition. In the example sales data, you may want to look only at territory 3, which is also easy:

```
print(calls_revenue[calls_revenue.Territory==3])
```

```
  Team member  Territory  Month  Calls  Amount
0       Jorge          3      1    107   50876
1       Jorge          3      2     88   57682
2       Jorge          3      3     84   53689
3       Jorge          3      4    113   49173
```

In this example, you select only rows in which the territory is equal to 3 but using exactly that expression, `revenue.Territory==3`, as the index for the data frame. From the point of view of plain Python, such use is nonsense and illegal, but for a pandas data frame, it works and makes for a much more concise expression.

More complex expressions are also allowed, of course. If you want to select only rows in which the amount per call is greater than 500, you could use this expression instead:

```
print(calls_revenue[calls_revenue.Amount/calls_revenue.Calls>500])
```

```
  Team member  Territory  Month  Calls  Amount
1       Jorge          3      2     88   57682
2       Jorge          3      3     84   53689
4         Ana          1      1     91   54228
9         Ali          2      2     85   44957
```

Even better, you could calculate and add that column to your data frame by using a similar operation:

```
calls_revenue['Call_Amount'] = calls_revenue.Amount/calls_revenue.Calls
print(calls_revenue)
```

```
   Team member  Territory  Month  Calls  Amount  Call_Amount
0        Jorge          3      1    107   50876   475.476636
1        Jorge          3      2     88   57682   655.477273
2        Jorge          3      3     84   53689   639.154762
3        Jorge          3      4    113   49173   435.159292
4          Ana          1      1     91   54228   595.912088
5          Ana          1      2    129   61640   477.829457
6          Ana          1      3     96   43491   453.031250
7          Ana          1      4    128   52173   407.601562
8          Ali          2      1    120   36061   300.508333
9          Ali          2      2     85   44957   528.905882
10         Ali          2      3     87   35058   402.965517
11         Ali          2      4     87   33855   389.137931
```

Again, note that pandas's built-in logic replaces a more cumbersome structure in plain Python.

**QUICK CHECK: SELECTING IN PYTHON** What Python code structure would you use to select only rows meeting certain conditions?

### 24.5.3 Grouping and aggregation

As you might expect, pandas has plenty of tools to summarize and aggregate data as well. In particular, getting the sum, mean, median, minimum, and maximum values from a column uses clearly named column methods:

```
print(calls_revenue.Calls.sum())
print(calls_revenue.Calls.mean())
print(calls_revenue.Calls.median())
print(calls_revenue.Calls.max())
print(calls_revenue.Calls.min())
```

```
1215
101.25
93.5
129
84
```

If, for example, you want to get all of the rows in which the amount per call is above the median, you can combine this trick with the selection operation:

```
print(calls_revenue.Call_Amount.median())
print(calls_revenue[calls_revenue.Call_Amount >=
     calls_revenue.Call_Amount.median()])
```

```
464.2539427570093
  Team member  Territory  Month  Calls  Amount  Call_Amount
0       Jorge          3      1    107   50876   475.476636
1       Jorge          3      2     88   57682   655.477273
2       Jorge          3      3     84   53689   639.154762
4         Ana          1      1     91   54228   595.912088
5         Ana          1      2    129   61640   477.829457
9         Ali          2      2     85   44957   528.905882
```

In addition to being able to pick out summary values, it's often useful to group the data based on other columns. In this simple example, you can use the `groupby` method to group your data. You may want to know the total calls and amounts by month or by territory, for example. In those cases, use those fields with the data frame's `groupby` method:

```
print(calls_revenue[['Month', 'Calls', 'Amount']].groupby(['Month']).sum())
```

```
       Calls  Amount
Month
1        318  141165
2        302  164279
3        267  132238
4        328  135201
```

```
print(calls_revenue[['Territory', 'Calls',
     'Amount']].groupby(['Territory']).sum())
```

```
           Calls  Amount
Territory
1            444  211532
2            379  149931
3            392  211420
```

In each case, you select the columns that you want to aggregate, group them by the values in one of those columns, and (in this case) sum the values for each group. You could also use any of the other methods mentioned earlier in this chapter.

Again, all these examples are simple, but they illustrate a few of the options you have for manipulating and selecting data with pandas. If these ideas resonate with your needs, you can learn more by studying the pandas documentation at http://pandas.pydata.org.

**TRY THIS: GROUPING AND AGGREGATING** Experiment with pandas and the data in previous examples. Can you get the calls and amounts by both team member and month?

## 24.6 Plotting data

Another very attractive feature of pandas is the ability to plot the data in a data frame very easily. Although you have many options for plotting data in Python and Jupyter notebook, pandas can use matplotlib directly from a data frame. You may recall that when you started your Jupyter session, one of the first commands you gave was the Jupyter "magic" command to enable matplotlib for inline plotting:

```
%matplotlib inline
```

Because you have the ability to plot, see how you might plot some data (figure 24.3). To continue with the sales example, if you want to plot the quarter's mean sales by territory, you can get a graph right in your notebook just by adding `.plot.bar()`:

```
calls_revenue[['Territory', 'Calls']].groupby(['Territory']).sum().plot.bar()
```

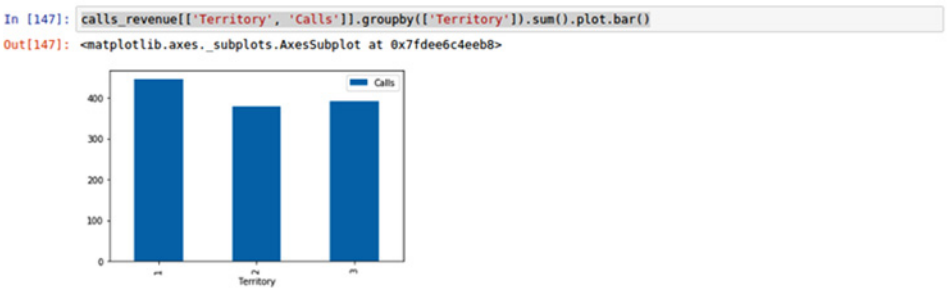



Figure 24.3 **Bar plot of a pandas data frame in Jupyter notebook**

Other options are available. `plot()` alone or `.plot.line()` creates a line graph, `.plot.pie()` creates a pie chart, and so on.

Thanks to the combination of pandas and matplotlib, plotting data in a Jupyter notebook is quite easy. I should also note that although such plotting is easy, there are many things that it doesn't do extremely well.

**TRY THIS: PLOTTING** Plot a line graph of the monthly average amount per call.

## 24.7 Why you might not want to use pandas

The preceding examples illustrate only a tiny fraction of the tools pandas can offer you in cleaning, exploring, and manipulating data. As I mentioned at the beginning of this chapter, pandas is an excellent tool set that excels in what it was designed to do. That doesn't mean, however, that pandas is the tool for all situations or for all people.

There are reasons why you might elect to use plain old Python (or some other tool) instead. For one thing, as I mention earlier, learning to fully use pandas is in some ways like learning another language, which may not be something you have the time or inclination for. Also, pandas may not be ideal in all production situations, particularly with very large data sets that don't require much in the way of math operations or with data that isn't easy to put into the formats that work best with pandas. Munging large collections of product information, for example, probably wouldn't benefit so much from pandas; neither would basic processing of a stream of transactions.

The point is that you should choose your tools thoughtfully based on the problems at hand. In many cases, pandas will truly make your life easier as you work with data, but in other cases, plain old Python may be your best bet.

## Summary

- Python offers many benefits for data handling, including the ability to handle very large data sets and the flexibility to handle data in ways that match your needs.
- Jupyter notebook is a useful way to access Python via a web browser, which also makes improved presentation easier.
- pandas is a tool that makes many common data-handling operations much easier, including cleaning, combining, and summarizing data.
- pandas also makes simple plotting much easier.

# *Modeling and prediction*

The "science" part of "data science" is the modelling, the part that creates the predictions, correlations, or classifications that form the information we hope to get out a data science project.

For the final chapter of this ebook, I've chosen a chapter from *Real-world Machine Learning* by Henrik Brink, Joseph W. Richards, and Mark Fetherolf on "Modeling and prediction." This chapter lays out the basics of machine learning modelling and explains the difference between supervised and unsupervised learning. From there it uses the Python package scikit-learn to demonstrate several common machine learning approaches, including logistic regression, support vector machines, k-nearest neighbors, linear regression, and random forest algorithms to model problems like the survival chances of passengers on the Titanic, number recognition, and mileage prediction.

Chapter 3 from *Real-world Machine Learning* by Henrik Brink, Joseph W. Richards, and Mark Fetherolf.

# *Modeling and prediction*

***This chapter covers***

- Discovering relationships in data through ML modeling
- Using models for prediction and inference
- Building classification models
- Building regression models

The previous chapter covered guidelines and principles of data collection, preprocessing, and visualization. The next step in the machine-learning workflow is to use that data to begin exploring and uncovering the relationships that exist between the input features and the target. In machine learning, this process is done by building statistical models based on the data. This chapter covers the basics required to understand ML modeling and to start building your own models. In contrast to most machine-learning textbooks, we spend little time discussing the various approaches to ML modeling, instead focusing attention on the big-picture concepts. This will help you gain a broad understanding of machine-learning model building and quickly get up to speed on building your own models to solve real-world problems. For those seeking more information about specific ML modeling techniques, please see the appendix.

We begin the chapter with a high-level overview of statistical modeling. This discussion focuses on the big-picture concepts of ML modeling, such as the purpose of models, the ways in which models are used in practice, and a succinct look at types of modeling techniques in existence and their relative strengths and weaknesses. From there, we dive into the two most common machine-learning models: classification and regression. In these sections, we give more details about how to build models on your data. We also call attention to a few of the most common algorithms used in practice in the "Algorithm highlight" boxes scattered throughout the chapter.

## 3.1 Basic machine-learning modeling

The objective of machine learning is to discover patterns and relationships in data and to put those discoveries to use. This process of discovery is achieved through the use of modeling techniques that have been developed over the past 30 years in statistics, computer science, and applied mathematics. These various approaches can range from simple to tremendously complex, but all share a common goal: to estimate the functional relationship between the input features and the target variable.

These approaches also share a common workflow, as illustrated in figure 3.1: use of historical data to build and optimize a model that is, in turn, used to make predictions based on new data. This section prepares you for the practical sections later in the chapter. You'll look at the general goal of machine learning modeling in the next section, and move on to seeing how the end product can be used and a few important aspects for differentiating between ML algorithms.

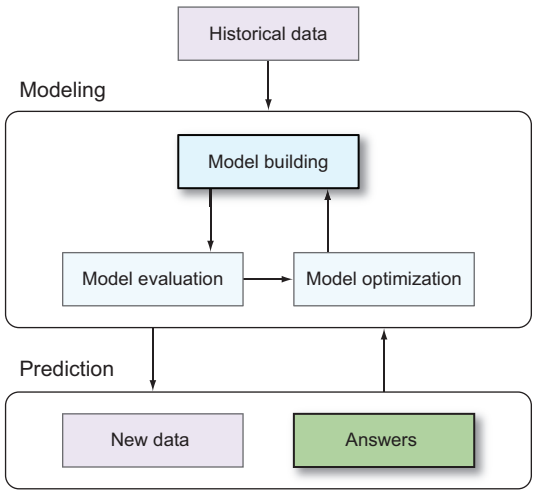



Figure 3.1 The basic ML workflow

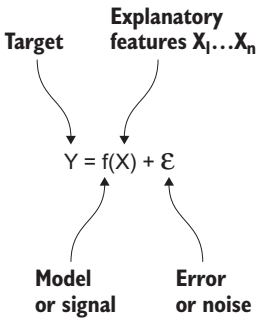
### 3.1.1 Finding the relationship between input and target

Let's frame the discussion of ML modeling around an example. Recall the Auto MPG dataset from chapter 2. The dataset contains metrics about automobiles, such as

manufacturer region, model year, vehicle weight, horsepower, and number of cylinders. The purpose of the dataset is to understand the relationship between the input features and a vehicle's miles per gallon (MPG) rating.

Input features are typically referred to using the symbol X, with subscripts differentiating inputs when multiple input features exist. For instance, we'll say that $X_1$ refers to manufacturer region, $X_2$ to model year, $X_3$ to vehicle weight, and so forth. The collection of all the input features is referred to as the bold **X**. Likewise, the target variable is typically referred to as Y.

The relationship between the inputs, **X**, and output, Y, can be succinctly represented by this simple formula:




In this equation, f represents the unknown function that relates the input variables to the target, Y. *The goal of ML modeling is to accurately estimate f by using data.* The symbol ε represents random noise in the data that's unrelated to the function f. The function f is commonly referred to as the *signal*, whereas the random variable ε is called the *noise*. The challenge of machine learning is to use data to determine what the true signal is, while ignoring the noise.

In the Auto MPG example, the function f describes the true MPG rating for an automobile as a function of that car's many input features. If you knew that function perfectly, you could know the MPG rating for any car, real or fictional. But you could have numerous sources of noise, ε, including (and certainly not limited to) the following:

- Imperfect measurement of each vehicle's MPG rating caused by small inaccuracies in the measuring devices—measurement noise
- Variations in the manufacturing process, causing each car in the fleet to have slightly different MPG measurements—manufacturing process noise
- Noise in the measurement of the input features, such as weight and horsepower
- Lack of access to the broader set of features that would exactly determine MPG

Using the noisy data that you have from hundreds of vehicles, the ML approach is to use modeling techniques to find a good estimate for f. This resultant estimate is referred to as an *ML model*.

In sections 3.2 and 3.3, we describe in further detail how these ML modeling techniques work. Indeed, the bulk of the academic literature on machine learning deals with how to best estimate f.

### 3.1.2 The purpose of finding a good model

Assuming that you have a good estimate of f, what next? Machine learning has two main goals: *prediction* and *inference*.

#### Prediction

After you have a model, you can use that model to generate predictions of the target, Y, for new data, $\mathbf{X}_{new}$, by plugging those new features into the model. In mathematical notation, if $f_{est}$ denotes your machine-learning estimate of f (recall that f denotes the true relationship between the features and the target), then predictions for new data can be obtained by plugging the new data into this formula:

$$Y_{pred} = f_{est}(\mathbf{X}_{new})$$

These predictions can then be used to make decisions about the new data or may be fed into an automated workflow.

Going back to the Auto MPG example, suppose that you have an ML model, $f_{est}$, that describes the relationship between MPG and the input metrics of an automobile. Prediction allows you to ask the question, "What would the MPG of a certain automobile with known input metrics be?" Such a predictive ability would be useful for designing automobiles, because it would allow engineers to assess the MPG rating of different design concepts and to ensure that the individual concepts meet MPG requirements.

Prediction is the most common use of machine-learning systems. Prediction is central to many ML use cases, including these:

- Deciphering handwritten digits or voice recordings
- Predicting the stock market
- Forecasting
- Predicting which users are most likely to click, convert, or buy
- Predicting which users will need product support and which are likely to unsubscribe
- Determining which transactions are fraudulent
- Making recommendations

Because of the high levels of predictive accuracy attained by machine-learning approaches and the rapid speed by which ML predictions can be generated, ML is used every day by thousands of companies for predictive purposes.

#### Inference

In addition to making predictions on new data, you can use machine-learning models to better understand the relationships between the input features and the output target.

A good estimate of f can enable you to answer deep questions about the associations between the variables at hand. For example:

- Which input features are most strongly related to the target variable?
- Are those relationships positive or negative?
- Is f a simple relationship, or is it a function that's more nuanced and nonlinear?

These inferences can tell you a lot about the data-generating process and give clues to the factors driving relationships in the data. Returning to the Auto MPG example, you can use inference to answer questions such as these: Does manufacturer region have an effect on MPG? Which of the inputs are most strongly related to MPG? And are they negatively or positively related? Answers to these questions can give you an idea of the driving factors in automobile MPG and give clues about how to engineer vehicles with higher MPG.

### 3.1.3 Types of modeling methods

Now the time has come to dust off your statistics knowledge and dive into some of the mathematical details of ML modeling. Don't worry—we'll keep the discussion relatively broad and understandable for those without much of a statistics background!

Statistical modeling has a general trade-off between predictive accuracy and model interpretability. Simple models are easy to interpret, yet won't produce accurate predictions (particularly for complicated relationships). Complex models may produce accurate predictions, but may be black-box and hard to interpret.

In addition, the machine-learning model has two main types: parametric and nonparametric. The essential difference is that parametric models assume that f takes a specific functional form, whereas nonparametric models don't make such strict assumptions. Therefore, parametric approaches tend to be simple and interpretable, but less accurate. Likewise, nonparametric approaches are usually less interpretable but more accurate across a broad range of problems. Let's take a closer look at both parametric and nonparametric approaches to ML modeling.

#### PARAMETRIC METHODS

The simplest example of a parametric approach is linear regression. In linear regression, f is assumed to be a linear combination of the numerical values of the inputs. The standard linear regression model is as follows:

$$f(\mathbf{X}) = \beta_0 + X_1 \times \beta_1 + X_2 \times \beta_2 + \ldots$$

In this equation, the unknown parameters, $\beta_0$, $\beta_1$,… can be interpreted as the intercept and slope parameters (with respect to each of the inputs). When you fit a parametric model to some data, you estimate the best values of each of the unknown parameters. Then you can turn around and plug those estimates into the formula for $f(\mathbf{X})$ along with new data to generate predictions.

Other examples of commonly used parametric models include logistic regression, polynomial regression, linear discriminant analysis, quadratic discriminant analysis,

(parametric) mixture models, and naïve Bayes (when parametric density estimation is used). Approaches often used in conjunction with parametric models for model selection purposes include ridge regression, lasso, and principal components regression. Further details about some of these methods are given later in this chapter, and a description of each approach is given in the appendix.

The drawback of parametric approaches is that they make strong assumptions about the true form of the function f. In most real-world problems, f doesn't assume such a simple form, especially when there are many input variables (X). In these situations, parametric approaches will fit the data poorly, leading to inaccurate predictions. Therefore, most real-world approaches to machine learning depend on nonparametric machine-learning methods.

#### Nonparametric methods

In *nonparametric* models, f doesn't take a simple, fixed function. Instead, the form and complexity of f adapts to the complexity of the data. For example, if the relationship between X and Y is wiggly, a nonparametric approach will choose a function f that matches the curvy patterns. Likewise, if the relationship between the input and output variable is smooth, a simple function f will be chosen.

A simple example of a nonparametric model is a classification tree. A *classification tree* is a series of recursive binary decisions on the input features. The classification tree learning algorithm uses the target variable to learn the optimal series of splits such that the terminal leaf nodes of the tree contain instances with similar values of the target.

Take, for example, the Titanic Passengers dataset. The classification tree algorithm first seeks the best input feature to split on, such that the resulting leaf nodes contain passengers who either mostly lived or mostly died. In this case, the best split is on the sex (male/female) of the passenger. The algorithm continues splitting on other input features in each of the subnodes until the algorithm can no longer detect any good subsequent splits.

Classification trees are nonparametric because the depth and complexity of the tree isn't fixed in advance, but rather is learned from the data. If the relationship between the target variable and the input features is complex and there's a sufficient amount of data, then the tree will grow deeper, uncovering more-nuanced patterns. Figure 3.2 shows two classification trees learned from different subsets of the Titanic Passengers dataset. In the left panel is a tree learned from only 400 passengers: the resultant model is simple, consisting of only a single split. In the right panel is a tree learned from 891 passengers: the larger amount of data enables the model to grow in complexity and find more-detailed patterns in the data.

Other examples of nonparametric approaches to machine learning include k-nearest neighbors, splines, basis expansion methods, kernel smoothing, generalized additive models, neural nets, bagging, boosting, random forests, and support vector machines. Again, more details about some of these methods are given later in this chapter, and a description of each approach is given in the appendix.

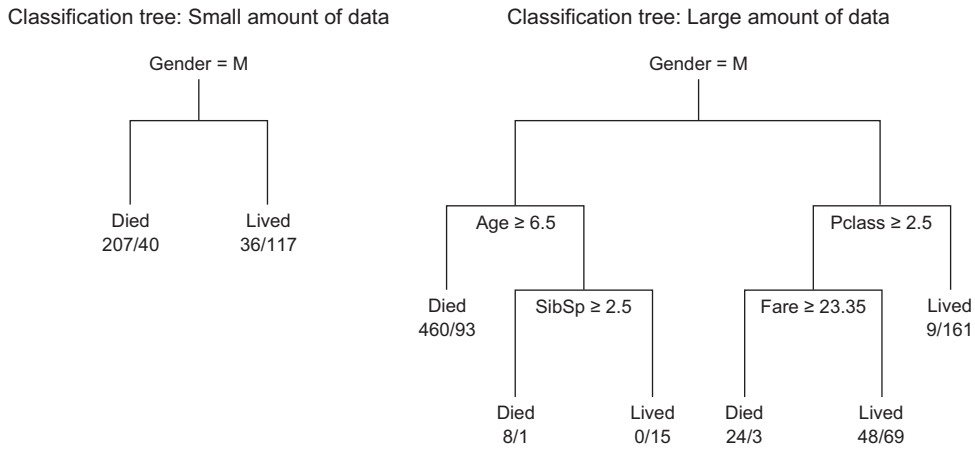



**Figure 3.2 A decision tree is an example of a nonparametric ML algorithm, because its functional form isn't fixed. The tree model can grow in complexity with larger amounts of data to capture more-complicated patterns. In each terminal node of the tree, the ratio represents the number of training instances in that node that died versus lived.**

### 3.1.4 Supervised versus unsupervised learning

Machine-learning problems fall into two camps: supervised and unsupervised. *Supervised problems* are ones in which you have access to the target variable for a set of training data, and *unsupervised problems* are ones in which there's no identified target variable.

All the examples so far in this book fall in the supervised camp. These problems each contain a target of interest (Did the Titanic passenger survive? Did the customer churn? What's the MPG?) and a set of training data with known values of the target. Indeed, most problems in machine learning are supervised in nature, and most ML techniques are designed to solve supervised problems. We spend the vast majority of this book describing how to solve supervised problems.

In unsupervised learning, you have access to only input features, and don't have an associated target variable. So what kinds of analyses can you perform if there's no target available? The unsupervised learning approach has two main classes:

- *Clustering*—Use the input features to discover natural groupings in the data and to divide the data into those groups. Methods: k-means, Gaussian mixture models, and hierarchical clustering.
- *Dimensionality reduction*—Transform the input features into a small number of coordinates that capture most of the variability of the data. Methods: principal component analysis (PCA), multidimensional scaling, manifold learning.

Both clustering and dimensionality reduction have wide popularity (particularly, k-means and PCA), yet are often abused and used inappropriately when a supervised approach is warranted.

But unsupervised problems do play a significant role in machine learning, often in support of supervised problems, either to help compile training data for learning or to derive new input features on which to learn. You'll return to the topic of unsupervised learning in chapter 8.

Now, let's transition to the more practical aspects of ML modeling. Next we describe the steps needed to start building models on your own data and the practical considerations of choosing which algorithm to use. We break up the rest of the chapter into two sections corresponding to the two most common problems in machine learning: classification and regression. We begin with the topic of classification.

## 3.2 Classification: predicting into buckets

In machine learning, *classification* describes the prediction of new data into buckets (classes) by using a *classifier* built by the machine-learning algorithm. Spam detectors put email into Spam and No Spam buckets, and handwritten digit recognizers put images into buckets from 0 through 9, for example. In this section, you'll learn how to build classifiers based on the data at hand. Figure 3.3 illustrates the process of classification.

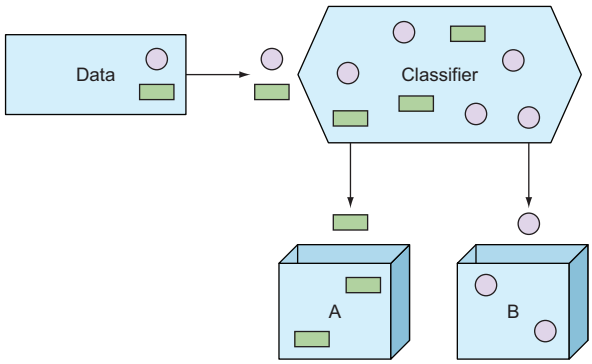



**Figure 3.3 A classification process. Rectangles and circles are divided by a classifier into classes A and B. This is a case of binary classification with only two classes.**

Let's again use an example. In chapter 2, you looked at the Titanic Passengers dataset for predicting survival of passengers onboard the ill-fated ship. Figure 3.4 shows a subset of this data.

| PassengerId | Survived | Pclass | Gender | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 3 | Male | 22 | 1 | 0 | A/5 21171 | 7.25 | | S |
| 2 | 1 | 1 | Female | 38 | 1 | 0 | PC 17599 | 71.2833 | C85 | C |
| 3 | 1 | 3 | Female | 26 | 0 | 0 | STON/02. 3101282 | 7.925 | | S |
| 4 | 1 | 1 | Female | 35 | 1 | 0 | 113803 | 53.1 | C123 | S |
| 5 | 0 | 3 | Male | 35 | 0 | 0 | 373450 | 8.05 | | S |
| 6 | 0 | 3 | Male | | 0 | 0 | 330877 | 8.4583 | | Q |

**Figure 3.4 A subset of the Titanic Passengers dataset**

As we've previously discussed, typically the best way to start an ML project is to get a feel for the data by visualizing it. For example, it's considered common knowledge that more women than men survived the Titanic, and you can see that this is the case from the mosaic plot in figure 3.5 (if you've forgotten about mosaic plots, look back at section 2.3.1).

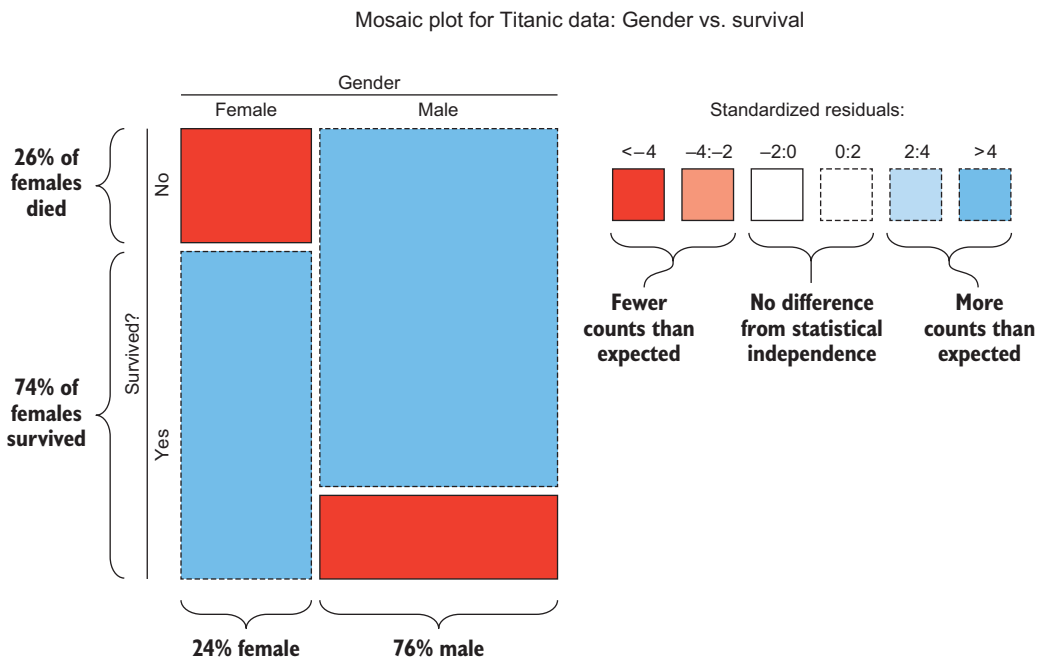



**Figure 3.5 Mosaic plot showing overwhelming support for the idea that more women than men survived the disaster.**

By using the visualization techniques in section 2.3, you can get a feeling for the performance of each feature in the Titanic Passengers dataset. But it's important to realize that just because a single feature looks good or bad, it doesn't necessarily show the performance of the feature in combination with one or more other features. Maybe the age together with the sex and social status divides the passengers much better than any single feature. In fact, this is one of the main reasons to use machine-learning algorithms in the first place: to find signals in many dimensions that humans can't discover easily.

The following subsections introduce the methodology for building classification models and making predictions. You'll look at a few specific algorithms and the difference between linear and nonlinear algorithms.

### 3.2.1 Building a classifier and making predictions

The first order of business is to choose the classification algorithm to use for building the classifier. Many algorithms are available, and each has pros and cons for different data and deployment requirements. The appendix provides a table of algorithms and a comparison of their properties. You'll use this table throughout the book for selecting algorithms to try for different problems. In this section, the choice of algorithm isn't essential; in the next chapter, you'll learn how to properly measure the performance of the algorithm and choose the best for the job.

The next step is to ready the data for modeling. After exploring some of the features in the dataset, you may want to preprocess the data to deal with categorical features, missing values, and so on (as discussed in chapter 2). The preprocessing requirements are also dependent on the specific algorithm, and the appendix lists these requirements for each algorithm.

For the Titanic survival model, you'll start by choosing a simple classification algorithm: logistic regression.[1] For logistic regression, you need to do the following:

1. Impute missing values.
2. Expand categorical features.
3. From chapter 2, you know that the Fare feature is heavily skewed. In this situation, it's advantageous (for some ML models) to transform the variable to make the feature distribution more symmetric and to reduce the potentially harmful impact of outliers. Here, you'll choose to transform Fare by taking the square root.

The final dataset that you'll use for modeling is shown in figure 3.6.

| Pclass | Age | SibSp | Parch | sqrt_Fare | Gender = female | Gender = male | Embarked = C | Embarked = Q | Embarked = S |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 22 | 1 | 0 | 2.692582 | 0 | 1 | 0 | 0 | 1 |
| 1 | 38 | 1 | 0 | 8.442944 | 1 | 0 | 1 | 0 | 0 |
| 3 | 26 | 0 | 0 | 2.815138 | 1 | 0 | 0 | 0 | 1 |
| 1 | 35 | 1 | 0 | 7.286975 | 1 | 0 | 0 | 0 | 1 |
| 3 | 35 | 0 | 0 | 2.837252 | 0 | 1 | 0 | 0 | 1 |

**Figure 3.6 The first five rows of the Titanic Passengers dataset after processing categorical features and missing values, and transforming the Fare variable by taking the square root (see the `prepare_data` function in the source code repository). All features are now numerical, which is the preferred format for most ML algorithms.**

You can now go ahead and build the model by running the data through the logistic regression algorithm. This algorithm is implemented in the scikit-learn Python package, and the model-building and prediction code look like the following listing.

[1] The *regression* in logistic regression doesn't mean it's a regression algorithm. Logistic regression expands linear regression with a logistic function to make it suitable for classification.

**Listing 3.1 Building a logistic regression classifier with scikit-learn**

```
from sklearn.linear_model import LogisticRegression as Model

def train(features, target):
  model = Model()
  model.fit(features, target)
  return model

def predict(model, new_features):
  preds = model.predict(new_features)
  return preds

# Assume Titanic data is loaded into titanic_feats,
# titanic_target and titanic_test
model = train(titanic_feats, titanic_target)
predictions = predict(model, titanic_test)
```

Imports the logistic regression algorithm

Fits the logistic regression algorithm using features and target data

Makes predictions on a new set of features using the model

Returns predictions (0 or 1)

Returns the model built by the algorithm

After building the model, you predict the survival of previously unseen passengers based on their features. The model expects features in the format given in figure 3.6, so any new passengers will have to be run through exactly the same processes as the training data. The output of the `predict` function will be 1 if the passenger is predicted to survive, and 0 otherwise.

It's useful to visualize the classifier by plotting the decision boundary. Given two of the features in the dataset, you can plot the boundary that separates surviving passengers from the dead, according to the model. Figure 3.7 shows this for the Age and square-root Fare features.

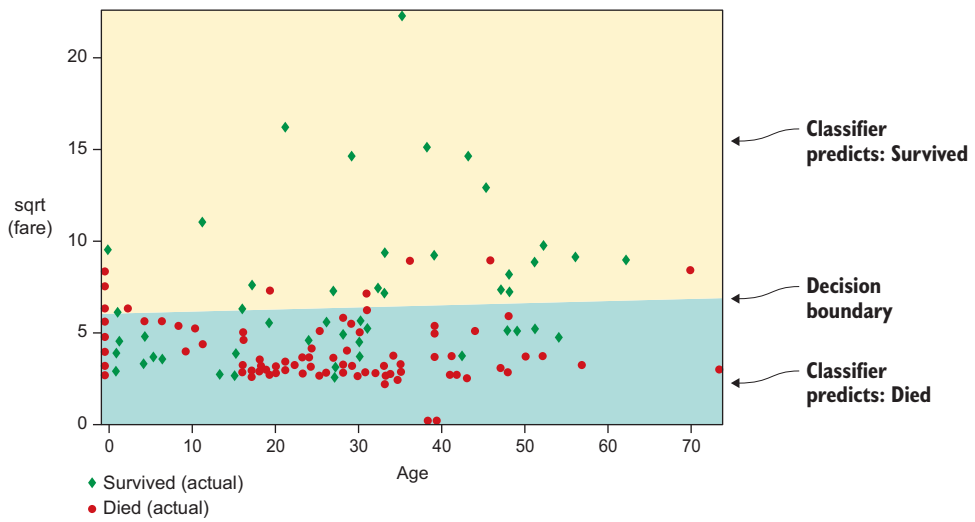



**Figure 3.7 The decision boundary for the Age and sqrt(Fare) features. The diamonds show passengers who survived, whereas circles denote passengers who died. The light background denotes the combinations of Age and Fare that are predicted to yield survival. Notice that a few instances overlap the boundary. The classifier isn't perfect, but you're looking in only two dimensions. The algorithm on the full dataset finds this decision boundary in 10 dimensions, but that becomes harder to visualize.**

### Algorithm highlight: logistic regression

In these "Algorithm highlight" boxes, you'll take a closer look at the basic ideas behind the algorithms used throughout the book. This allows curious readers to try to code up, with some extra research, basic working versions of the algorithms. Even though we focus mostly on the use of existing packages in this book, understanding the basics of a particular algorithm can sometimes be important to fully realize the predictive potential.

The first algorithm you'll look at is the *logistic regression algorithm*, arguably the simplest ML algorithm for classification tasks. It's helpful to think about the problem as having only two features and a dataset divided into two classes. Figure 3.7 shows an example, with the features Age and sqrt(Fare); the target is Survived or Died. To build the classifier, you want to find the line that best splits the data into the target classes. A line in two dimensions can be described by two parameters. These two numbers are the parameters of the model that you need to determine.

The algorithm then consists of the following steps:

1. You can start the search by picking the parameter values at random, hence placing a random line in the two-dimensional figure.
2. Measure how well this line separates the two classes. In logistic regression, you use the statistical *deviance* for the goodness-of-fit measurement.
3. Guess new values of the parameters and measure the separation power.
4. Repeat until there are no better guesses. This is an *optimization* procedure that can be done with a range of optimization algorithms. Gradient descent is a popular choice for a simple optimization algorithm.

This approach can be extended to more dimensions, so you're not limited to two features in this model. If you're interested in the details, we strongly encourage you to research further and try to implement this algorithm in your programming language of choice. Then look at an implementation in a widely used ML package. We've left out plenty of details, but the preceding steps remain the basis of the algorithm.

Some properties of logistic regression include the following:

- The algorithm is relatively simple to understand, compared to more-complex algorithms. It's also computationally simple, making it scalable to large datasets.
- The performance will degrade if the decision boundary that separates the classes needs to be highly nonlinear. See section 3.2.2.
- Logistic regression algorithms can sometimes overfit the data, and you often need to use a technique called *regularization* that limits this danger. See section 3.2.2 for an example of overfitting.

### Further reading

If you want to learn more about logistic regression and its use in the real world, check out *Applied Logistic Regression* by David Hosmer et al. (Wiley, 2013).

### 3.2.2 Classifying complex, nonlinear data

Looking at figure 3.7, you can understand why logistic regression is a linear algorithm: the decision boundary is a straight line. Of course, your data might not be well separated by a straight line, so for such datasets you should use a nonlinear algorithm. But nonlinear algorithms are typically more demanding computationally and don't scale well to large datasets. You'll look further at the scalability of various types of algorithms in chapter 8.

Looking again at the appendix, you can pick a nonlinear algorithm for modeling the Titanic Passengers dataset. A popular method for nonlinear problems is a support vector machine with a nonlinear kernel. Support vector machines are linear by nature, but by using a kernel, this model becomes a powerful nonlinear method. You can change a single line of code in listing 3.1 to use this new algorithm, and the decision boundary is plotted in figure 3.8:

```
from sklearn.svm import SVC as Model
```

You can see that the decision boundary in figure 3.8 is different from the linear one in figure 3.7. What you see here is a good example of an important concept in machine learning: overfitting. The algorithm is capable of fitting well to the data, almost at the single-record level, and you risk losing the ability to make good predictions on new data that wasn't included in the training set; the more complex you allow the model to become, the higher the risk of overfitting.

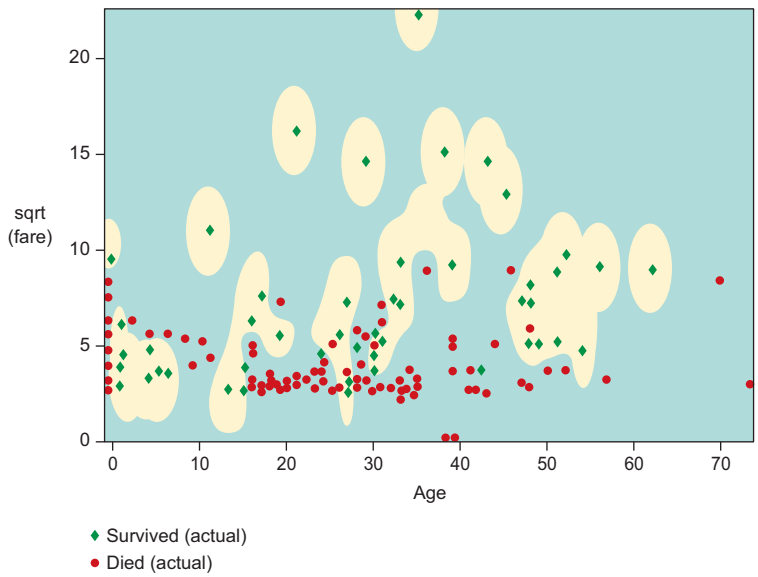



**Figure 3.8 Nonlinear decision boundary of the Titanic survival support vector machine classifier with a nonlinear kernel. The light background denotes the combinations of Age and Fare that are predicted to yield survival.**

Usually, you can avoid overfitting a nonlinear model by using model parameters built into the algorithm. By tweaking the parameters of the model, keeping the data unchanged, you can obtain a better decision boundary. Note that you're currently using intuition to determine when something is overfitting; in chapter 4, you'll learn how to use data and statistics to quantify this intuition. For now, you'll use our (the authors') experience and tweak a certain parameter called *gamma*. You don't need to know what gamma is at this point, only that it helps control the risk of overfitting. In chapter 5, you'll see how to optimize the model parameters without only guessing at better values. Setting gamma = 0.1 in the SVM classifier, you obtain the much improved decision boundary shown in figure 3.9.

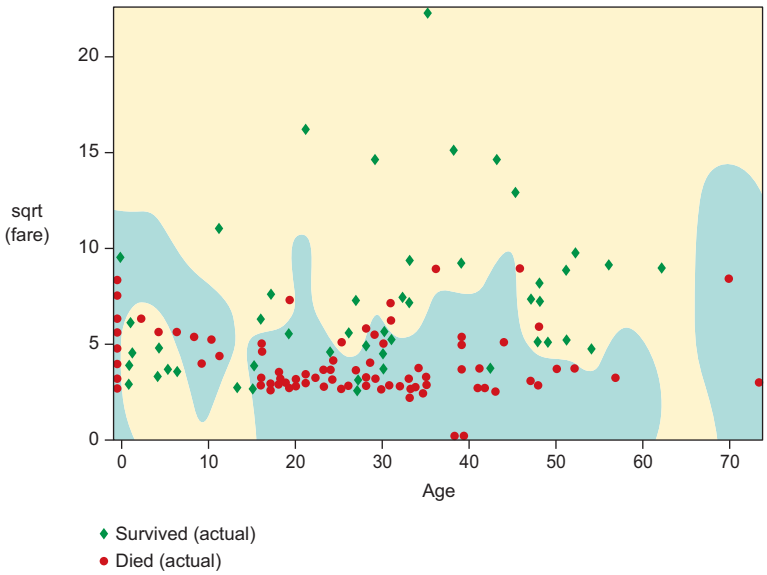



**Figure 3.9** **Decision boundary of nonlinear RBF-kernel SVM with gamma = 0.1**

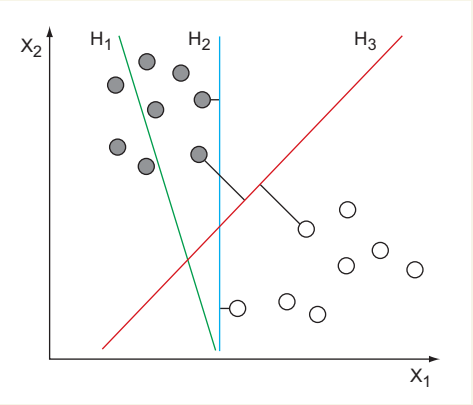
### Algorithm highlight: support vector machines

The support vector machine (SVM) algorithm is a popular choice for both linear and nonlinear problems. It has some interesting theoretical and practical properties that make it useful in many scenarios.

The main idea behind the algorithm is, as with logistic regression discussed previously, to find the line (or equivalent in higher dimensions) that separates the classes optimally. Instead of measuring the distance to all points, SVMs try to find the largest *margin* between only the points on either side of the decision line. The idea is that there's no reason to worry about points that are well within the boundary, only ones that are close. In the following image, you can see that lines $H_1$ and $H_2$ are bad separation

***(continued)***

boundaries, because the distance to the closest point on both sides of the line isn't the largest it can be. $H_3$ is the optimal line.




**An SVM decision boundary ($H_3$) is often superior to decision boundaries found by other ML algorithms.**

Although this algorithm is also linear in the sense that the separation boundary is linear, SVMs are capable of fitting to nonlinear data, as you saw earlier in this section. SVMs use a clever technique in order to fit to nonlinear data: the kernel trick. A *kernel* is a mathematical construct that can "warp" the space where the data lives. The algorithm can then find a linear boundary in this warped space, making the boundary nonlinear in the original space.

### Further reading

Hundreds of books have been written about machine-learning algorithms, covering everything from their theoretical foundation and efficient implementation to their practical use. If you're looking for a more rigorous treatment of these topics, we recommend two classic texts on ML algorithms:

### 3.2.3 Classifying with multiple classes

Up to this point, you've looked at classification into only two classes. In some cases, you'll have more than two classes. A good real-world example of multiclass classification is the handwritten digit recognition problem. Whenever you send old-school mail to your family, a robot reads the handwritten ZIP code and determines where to send the letter, and good digit recognition is essential in this process. A public dataset, the

MNIST database,[1] is available for research into these types of problems. This dataset consists of 60,000 images of handwritten digits. Figure 3.10 shows a few of the handwritten digit images.

**Figure 3.10 Four randomly chosen handwritten digits from the MNIST database**

The images are 28 × 28 pixels each, but we convert each image into $28^2 = 784$ features, one feature for each pixel. In addition to being a multiclass problem, this is also a high-dimensional problem. The pattern that the algorithm needs to find is a complex combination of many of these features, and the problem is nonlinear in nature.

To build the classifier, you first choose the algorithm to use from the appendix. The first nonlinear algorithm on the list that natively supports multiclass problems is the k-nearest neighbors classifier, which is another simple but powerful algorithm for nonlinear ML modeling. You need to change only one line in listing 3.1 to use the new algorithm, but you'll also include a function for getting the full prediction probabilities instead of just the final prediction:

```
from sklearn.neighbors import KNeighborsClassifier as Model

def predict_probabilities(model, new_features):
  preds = model.predict_proba(new_features)
  return preds
```

Building the k-nearest neighbors classifier and making predictions on the four digits shown in figure 3.10, you obtain the table of probabilities shown in figure 3.11.

You can see that the predictions for digits 1 and 3 are spot on, and there's only a small (10%) uncertainty for digit 4. Looking at the second digit (3), it's not surprising that this is hard to classify perfectly. This is the main reason to get the full probabilities in the first place: to be able to take action on things that aren't perfectly certain. This is easy to understand in the case of a post office robot routing letters; if the robot is sufficiently uncertain about some digits, maybe we should have a good old human look at it before we send it out wrong.

[1] You can find the MNIST Database of Handwritten Digits at http://yann.lecun.com/exdb/mnist/.

| | Actual value | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Digit 1 | 7 | 0 | 0 | 0 | 0.0 | 0 | 0.0 | 0 | 1 | 0 | 0.0 |
| Digit 2 | 3 | 0 | 0 | 0 | 0.7 | 0 | 0.2 | 0 | 0 | 0 | 0.1 |
| Digit 3 | 9 | 0 | 0 | 0 | 0.0 | 0 | 0.0 | 0 | 0 | 0 | 1.0 |
| Digit 4 | 5 | 0 | 0 | 0 | 0.0 | 0 | 0.9 | 0 | 0 | 0 | 0.1 |

Predicted digit

Digit 2 has a probability of 0.2 of being 5.

**Figure 3.11** Table of predicted probabilities from a k-nearest neighbors classifier, as applied to the MNIST dataset

### Algorithm highlight: k-nearest neighbors

The *k-nearest neighbors algorithm* is a simple yet powerful nonlinear ML method. It's often used when model training should be quick, but predictions are typically slower. You'll soon see why this is the case.

The basic idea is that you can classify a new data record by comparing it with similar records from the training set. If a dataset record consists of a set of numbers, $n_i$, you can find the *distance* between records via the usual distance formula:

$$d = \sqrt{n_1^2 + n_2^2 \cdots + n_r^2}$$

When making predictions on new records, you find the *closest* known record and assign that class to the new record. This would be a 1-nearest neighbor classifier, as you're using only the closest neighbor. Usually you'd use 3, 5, or 9 neighbors and pick the class that's most common among neighbors (you use odd numbers to avoid ties).

The training phase is relatively quick, because you index the known records for fast distance calculations to new data. The prediction phase is where most of the work is done, finding the closest neighbors from the entire dataset.

The previous simple example uses the usual Euclidean distance metric. You can also use more-advanced distance metrics, depending on the dataset at hand.

K-nearest neighbors is useful not only for classification, but for regression as well. Instead of taking the most common class of neighbors, you take the average or median values of the target values of the neighbors. Section 3.3 further details regression.

## 3.3 Regression: predicting numerical values

Not every machine-learning problem is about putting records into classes. Sometimes the target variable takes on numerical values—for example, when predicting dollar

values in a financial model. We call the act of predicting numerical values *regression*, and the model itself a *regressor*. Figure 3.12 illustrates the concept of regression.

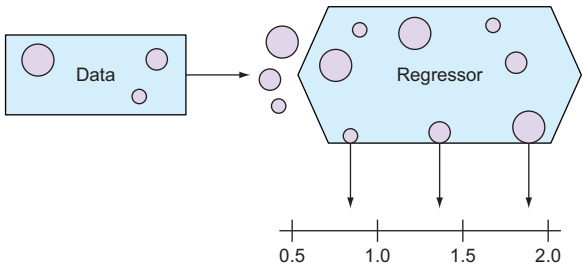



**Figure 3.12** In this regression process, the regressor is predicting the numerical value of a record.

As an example of a regression analysis, you'll use the Auto MPG dataset introduced in chapter 2. The goal is to build a model that can predict the average miles per gallon of a car, given various properties of the car such as horsepower, weight, location of origin, and model year. Figure 3.13 shows a small subset of this data.

| | MPG | Cylinders | Displacement | Horsepower | Weight | Acceleration | Model/year | Origin |
|---|---|---|---|---|---|---|---|---|
| **0** | 18 | 8 | 307 | 130 | 3504 | 12.0 | 70 | 1 |
| **1** | 15 | 8 | 350 | 165 | 3693 | 11.5 | 70 | 1 |
| **2** | 18 | 8 | 318 | 150 | 3436 | 11.0 | 70 | 1 |
| **3** | 16 | 8 | 304 | 150 | 3433 | 12.0 | 70 | 1 |
| **4** | 17 | 8 | 302 | 140 | 3449 | 10.5 | 70 | 1 |

**Figure 3.13** Small subset of the Auto MPG data

In chapter 2, you discovered useful relationships between the MPG rating, the car weight, and the model year. These relationships are shown in figure 3.14.

In the next section, you'll look at how to build a basic linear regression model to predict the miles per gallon values of this dataset of vehicles. After successfully building a basic model, you'll look at more-advanced algorithms for modeling nonlinear data.

### 3.3.1 Building a regressor and making predictions

Again, you'll start by choosing an algorithm to use and getting the data into a suitable format. Arguably, the linear regression algorithm is the simplest regression algorithm. As the name indicates, this is a linear algorithm, and the appendix shows the data preprocessing needed in order to use this algorithm. You need to (1) impute missing values and (2) expand categorical features. Our Auto MPG dataset has no missing values,

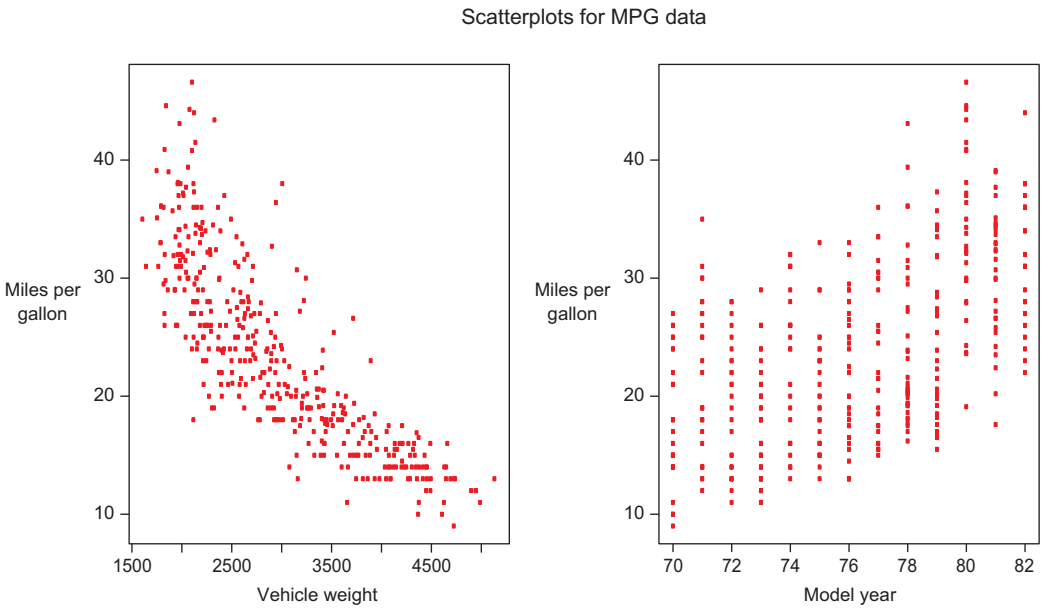



**Figure 3.14 Using scatter plots, you can see that Vehicle Weight and Model Year are useful for predicting MPG. See chapter 2 for more details.**

but there's one categorical column: Origin. After expanding the Origin column (as described in section 2.2.1 in chapter 2), you obtain the data format shown in figure 3.15.

You can now use the algorithm to build the model. Again, you can use the code structure defined in listing 3.1 and change this line:

```
from sklearn.linear_model import LinearRegression as Model
```

With the model in hand, you can make predictions. In this example, however, you'll split the dataset into a training set and a testing set before building the model. In chapter 4, you'll learn much more about how to evaluate models, but you'll use some

| | MPG | Cylinders | Displacement | Horsepower | Weight | Acceleration | Model/year | Origin = 1 | Origin = 2 | Origin = 3 |
|---|---|---|---|---|---|---|---|---|---|---|
| **387** | 27 | 4 | 140 | 86 | 2790 | 15.6 | 82 | 1 | 0 | 0 |
| **388** | 44 | 4 | 97 | 52 | 2130 | 24.6 | 82 | 0 | 1 | 0 |
| **389** | 32 | 4 | 135 | 84 | 2295 | 11.6 | 82 | 1 | 0 | 0 |
| **390** | 28 | 4 | 120 | 79 | 2625 | 18.6 | 82 | 1 | 0 | 0 |
| **391** | 31 | 4 | 119 | 82 | 2720 | 19.4 | 82 | 1 | 0 | 0 |

**Figure 3.15 The Auto MPG data after expanding the categorical Origin column**

simple techniques in this section. By training a model on only some of the data while holding out a testing set, you can subsequently make predictions on the testing set and see how close your predictions come to the actual values. If you were training on all the data and making predictions on some of that training data, you'd be cheating, as the model is more likely to make good predictions if it's seen the data while training.

Figure 3.16 shows the results of making predictions on a held-out testing set, and how they compare to the known values. In this example, you train the model on 80% of the data and use the remaining 20% for testing.

| Origin = 1 | Origin = 3 | Origin = 2 | MPG | Predicted MPG |
|---|---|---|---|---|
| 0 | 0 | 1 | 26.0 | 27.172795 |
| 1 | 0 | 0 | 23.8 | 24.985776 |
| 1 | 0 | 0 | 13.0 | 13.601050 |
| 1 | 0 | 0 | 17.0 | 15.181120 |
| 1 | 0 | 0 | 16.9 | 16.809079 |

**Figure 3.16 Comparing MPG predictions on a held-out testing set to actual values**

A useful way to compare more than a few rows of predictions is to use our good friend, the scatter plot, once again. For regression problems, both the actual target values and the predicted values are numeric. Plotting the predictions against each other in a scatter plot, introduced in chapter 2, you can visualize how well the predictions follow the actual values. This is shown for the held-out Auto MPG test set in figure 3.17. This figure

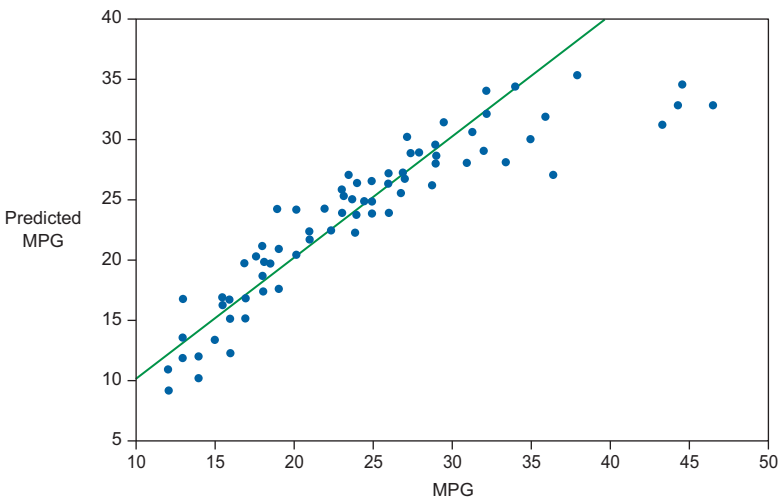


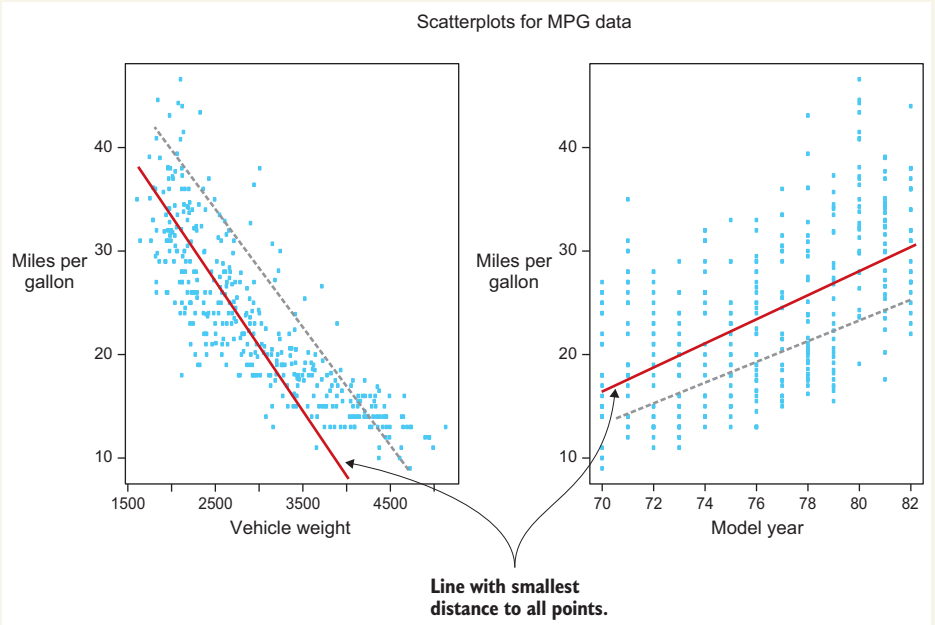

**Figure 3.17 A scatter plot of the actual versus predicted values on the held-out test set. The diagonal line shows the perfect regressor. The closer all of the predictions are to this line, the better the model.**

shows great prediction performance, as the predictions all fall close to the optimal diagonal line. By looking at this figure, you can get a sense of how your ML model might perform on new data. In this case, a few of the predictions for higher MPG values seem to be underestimated, and this may be useful information for you. For example, if you want to get better at estimating high MPG values, you might need to find more examples of high MPG vehicles, or you might need to obtain higher-quality data in this regime.

### Algorithm highlight: linear regression

Like logistic regression for classification, *linear regression* is arguably the simplest and most widely used algorithm for building regression models. The main strengths are linear scalability and a high level of interpretability.

This algorithm plots the dataset records as points, with the target variable on the y-axis, and fits a straight line (or plane, in the case of two or more features) to these points. The following figure illustrates the process of optimizing the distance from the points to the straight line of the model.




**Demonstration of how linear regression determines the best-fit line. Here, the dark line is the optimal linear regression fitted line on this dataset, yielding a smaller mean-squared deviation from the data to any other possible line (such as the dashed line shown).**

A straight line can be described by two parameters for lines in two dimensions, and so on. You know this from the a and b in $y = a \times x + b$ from the basic math. These parameters are fitted to the data, and when optimized, they completely describe the model and can be used to make predictions on new data.

### 3.3.2 Performing regression on complex, nonlinear data

In some datasets, the relationship between features can't be fitted by a linear model, and algorithms such as linear regression may not be appropriate if accurate predictions are required. Other properties, such as scalability, may make lower accuracy a necessary trade-off. Also, there's no guarantee that a nonlinear algorithm will be more accurate, as you risk overfitting to the data. As an example of a nonlinear regression model, we introduce the random forest algorithm. Random forest is a popular method for highly nonlinear problems for which accuracy is important. As evident in the appendix, it's also easy to use, as it requires minimal preprocessing of data. In figures 3.18 and 3.19, you can see the results of making predictions on the Auto MPG test set via the random forest model.

| Origin = 1 | Origin = 3 | Origin = 2 | MPG | Predicted MPG |
|---|---|---|---|---|
| 0 | 0 | 1 | 26.0 | 27.1684 |
| 1 | 0 | 0 | 23.8 | 23.4603 |
| 1 | 0 | 0 | 13.0 | 13.6590 |
| 1 | 0 | 0 | 17.0 | 16.8940 |
| 1 | 0 | 0 | 16.9 | 15.5060 |

**Figure 3.18** Table of actual versus predicted MPG values for the nonlinear random forest regression model

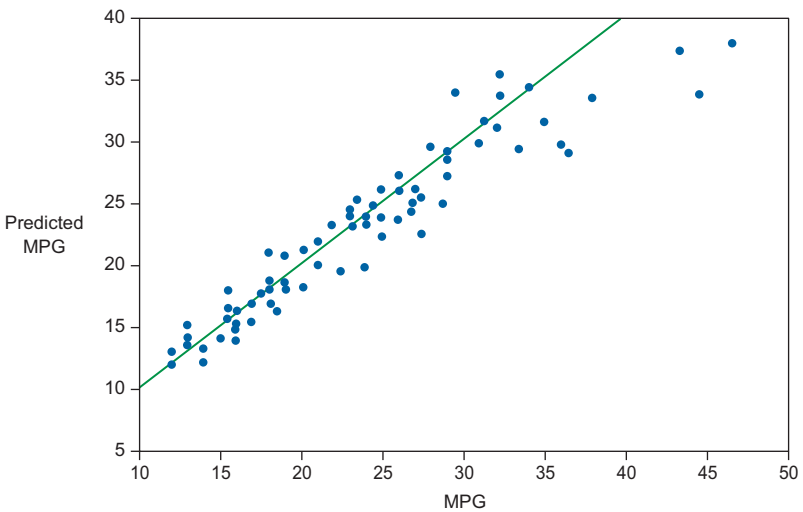



**Figure 3.19** Comparison of MPG data versus predicted values for the nonlinear random forest regression model

This model isn't much different from the linear algorithm, at least visually. It's not clear which of the algorithms performs the best in terms of accuracy. In the next chapter, you'll learn how to quantify the performance (often called the *accuracy score* of the model) so you can make meaningful measurements of how good the prediction accuracy is.

### Algorithm highlight: random forest

For the last algorithm highlight of this chapter, we introduce the *random forest* (RF) algorithm. This highly accurate nonlinear algorithm is widely used in real-world classification and regression problems.

The basis of the RF algorithm is the decision tree. Imagine that you need to make a decision about something, such as what to work on next. Some variables can help you decide the best course of action, and some variables weigh higher than others. In this case, you might ask first, "How much money will this make me?" If the answer is less than $10, you can choose to not go ahead with the task. If the answer is more than $10, you might ask the next question in the decision tree, "Will working on this make me happy?" and answer with a yes/no. You can continue to build out this tree until you've reached a conclusion and chosen a task to work on.

The decision tree algorithm lets the computer figure out, based on the training set, which variables are the most important, and put them in the top of the tree, and then gradually use less-important variables. This allows it to combine variables and say, "If the amount is greater than $10 and makes me happy, and amount of work less than 1 hour, then yes."

A problem with decision trees is that the top levels of the tree have a huge impact on the answer, and if the new data doesn't follow exactly the same distribution as the training set, the ability to generalize might suffer. This is where the random forest method comes in. By building a collection of decision trees, you mitigate this risk. When making the answer, you pick the majority vote in the case of classification, or take the mean in case of regression. Because you use votes or means, you can also give back full probabilities in a natural way that not many algorithms share.

Random forests are also known for other kinds of advantages, such as their immunity to unimportant features, noisy datasets in terms of missing values, and mislabeled records.

## 3.4 Summary

In this chapter, we introduced machine-learning modeling. Here we list the main takeaways from the chapter:

- The purpose of modeling is to describe the relationship between the input features and the target variable.
- You can use models either to generate predictions for new data (whose target is unknown) or to infer the true associations (or lack thereof) present in the data.

- There are hundreds of methods for ML modeling. Some are parametric, meaning that the form of the mathematical function relating the features to the target is fixed in advance. Parametric models tend to be more highly interpretable yet less accurate than nonparametric approaches, which are more flexible and can adapt to the true complexity of the relationship between the features and the target. Because of their high levels of predictive accuracy and their flexibility, nonparametric approaches are favored by most practitioners of machine learning.
- Machine-learning methods are further broken into supervised and unsupervised methods. Supervised methods require a training set with a known target, and unsupervised methods don't require a target variable. Most of this book is dedicated to supervised learning.
- The two most common problems in supervised learning are classification, in which the target is categorical, and regression, in which the target is numerical. In this chapter, you learned how to build both classification and regression models and how to employ them to make predictions on new data.
- You also dove more deeply into the problem of classification. Linear algorithms can define linear decision boundaries between classes, whereas nonlinear methods are required if the data can't be separated linearly. Using nonlinear models usually has a higher computational cost.
- In contrast to classification (in which a categorical target is predicted), you predict a numerical target variable in regression models. You saw examples of linear and nonlinear methods and how to visualize the predictions of these models.

## 3.5 Terms from this chapter

| Word | Definition |
|---|---|
| model | The base product from using an ML algorithm on training data. |
| prediction | Predictions are performed by pulling new data through the model. |
| inference | The act of gaining insight into the data by building the model and not making predictions. |
| (non)parametric | Parametric models make assumptions about the structure of the data. Nonparametric models don't. |
| (un)supervised | Supervised models, such as classification and regression, find the mapping between the input features and the target variable. Unsupervised models are used to find patterns in the data without a specified target variable. |
| clustering | A form of unsupervised learning that puts data into self-defined clusters. |
| dimensionality reduction | Another form of unsupervised learning that can map high-dimensional datasets to a lower-dimensional representation, usually for plotting in two or three dimensions. |
| classification | A supervised learning method that predicts data into buckets. |
| regression | The supervised method that predicts numerical target values. |

In the next chapter, you'll look at creating and testing models, the exciting part of machine learning. You'll see whether your choice of algorithms and features is going to work to solve the problem at hand. You'll also see how to rigorously validate a model to see how good its predictions are likely to be on new data. And you'll learn about validation methods, metrics, and some useful visualizations for assessing your models' performance.

# index

## D



## E

## O



## P



## Q



## R



## S

## T



## U



## V



## W



## X