

Author Picks

FREE



Exploring the Data Jungle

Finding, Preparing, and Using Real-World Data

Chapters selected by Brian Godsey





Exploring the Data Jungle
Finding, Preparing, and Using Real-World Data

Selected by Brian Godsey

Manning Author Picks

Copyright 2017 Manning Publications
To pre-order or learn more about these books go to www.manning.com

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

©2017 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

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

Cover designer: Leslie Haimes

ISBN 9781617295065
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

Introduction iv

DATA ALL AROUND US: THE VIRTUAL WILDERNESS 1

Data all around us: the virtual wilderness

Chapter 3 *from Think Like a Data Scientist* by Brian Godsey 2

EXPLORING DATA 33

Exploring data

Chapter 3 *from Practical Data Science with R*
by Nina Zumel and John Mount. 34

REAL-WORLD DATA 64

Real-world data

Chapter 2 *from Real-World Machine Learning*
by Henrik Brink, Joseph W. Richards, and Mark Fetherolf 65

index 91

introduction

Some people believe that data comes in simple, organized tables—numbers and text stacked in neat little rows, each value separated from its neighbor by a comma. And other people believe that all data values are 100% correct, because they originated from an autonomous data collection machine that never makes mistakes. Some people believe that every database, straight out of the box, can store anything and everything and make it easily retrievable in a fraction of a second. In a perfect world, these beliefs may align with the truth, but—in this world—far from it.

Data in the wild is unkempt and unruly. It's not always where you want it to be. It may exist in an odd format. It may have missing or incorrect values. It may be skewed or not representative of the population you wish to study. There may be way too much of it to manage. It may not exist. A good data scientist is no stranger to problems like these; they come with the territory. In order to avoid or solve these problems and others like them, it is helpful to be familiar with data in its many locations, formats, and qualities.

The three chapters in this collection each give a perspective of what you might find when you go looking for data. The first chapter—from my own book *Think Like a Data Scientist*—describes the world of data as a wilderness worthy of exploration and meticulous investigation. Here, the roles of data in the modern world have grown to the point that they can no longer be ignored; thus we need to prepare for the many ways and forms in which we might find the data we want. The second chapter—from *Practical Data Science with R* by Nina Zumel and John Mount—gives a thorough introduction to the many ways you can inspect data you have. The directives and suggestions of this *R*-specific viewpoint can be generalized to understand your data comprehensively using any statistical software, not just *R*. The third chapter—from *Real-World Machine Learning* by Henrik Brink, Joseph W. Richards, and Mark Fetherolf—gives a thoughtful blueprint for what to do as you prepare real-world data for machine learning. Data from the wild isn't usually ready to be fed to a highly intelligent, but coldly deterministic, algorithm without a little cleaning, organizing, and dressing-up.

Data isn't always approachable. It can be messy, wrong, or hard to access. But despite all of that, it can still answer real business questions and solve meaningful problems. This collection of chapters shows you how to approach data in the wild for maximum insight and benefit.

Data All Around Us: The Virtual Wilderness

The underlying basis for any discussion about data is the nature of data itself. What forms might data take? Where is it? How can I access it? What might it contain? Why do I need it? For some data science project goals, the answers to these questions are obvious. But for others, data scientists benefit from expanding their ideas to seek other possible answers to these questions. You are never stuck using only the data in the database that you have. There is a whole world of data possibilities out there if you do a little exploration and can make the connections between what you find and what your project can use.

This chapter from my book, *Think Like a Data Scientist*, first attempts to describe world of data and its vastness in order to motivate you to use it to your advantage. Second, it helps remove some technical barriers to making use of the things that you find. Last, the chapter presents some strategies for navigating the world of data, seeking and capturing exactly those specimens that will help you achieve success in your project.

Data all around us: the virtual wilderness

This chapter covers

- Discovering data you may need
- Interacting with data in various environments
- Combining disparate data sets

This chapter discusses the principal species of study of the data scientist: data. Having possession of data—namely, useful data—is often taken as a foregone conclusion, but it’s not usually a good idea to assume anything of the sort. As with any topic worthy of scientific examination, data can be hard to find and capture and is rarely completely understood. Any mistaken notion about a data set that you possess or would like to possess can lead to costly problems, so in this chapter, I discuss the treatment of data as an object of scientific study.

3.1 Data as the object of study

In recent years, there has been a seemingly never-ending discussion about whether the field of data science is merely a reincarnation or an offshoot—in the Big Data Age—of any of a number of older fields that combine software engineering

and data analysis: operations research, decision sciences, analytics, data mining, mathematical modeling, or applied statistics, for example. As with any trendy term or topic, the discussion over its definition and concept will cease only when the popularity of the term dies down. I don't think I can define *data science* any better than many of those who have done so before me, so let a definition from Wikipedia (https://en.wikipedia.org/wiki/Data_science), paraphrased, suffice:

Data science is the extraction of knowledge from data.

Simple enough, but that description doesn't distinguish *data science* from the many other similar terms, except perhaps to claim that *data science* is an umbrella term for the whole lot. On the other hand, this era of data science has a property that no previous era had, and it is, to me, a fairly compelling reason to apply a new term to the types of things that data scientists do that previous applied statisticians and data-oriented software engineers did not. This reason helps me underscore an often-overlooked but very important aspect of data science, as shown in figure 3.1.

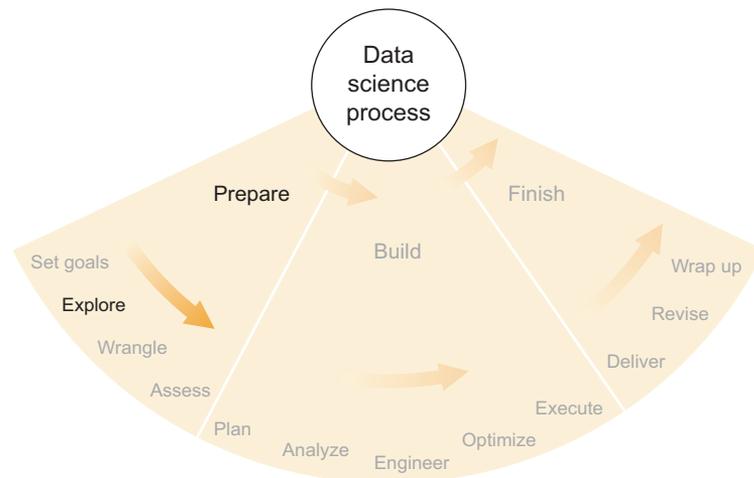


Figure 3.1 The second step of the preparation phase of the data science process: exploring available data

3.1.1 The users of computers and the internet became data generators

Throughout recent history, computers have made incredible advances in computational power, storage, and general capacity to accomplish previously unheard-of tasks. Every generation since the invention of the modern computer nearly a century ago has seen ever-shrinking machines that are orders of magnitude more powerful than the most powerful supercomputers of the previous generation.

The time period including the second half of the twentieth century through the beginning of the twenty-first, and including the present day, is often referred to as the Information Age. The Information Age, characterized by the rise to ubiquity of

computers and then the internet, can be divided into several smaller shifts that relate to analysis of data.

First, early computers were used mainly to make calculations that previously took an unreasonable amount of time. Cracking military codes, navigating ships, and performing simulations in applied physics were among the computationally intensive tasks that were performed by early computers.

Second, people began using computers to communicate, and the internet developed in size and capacity. It became possible for data and results to be sent easily across a large distance. This enabled a data analyst to amass larger and more varied data sets in one place for study. Internet access for the average person in a developed country increased dramatically in the 1990s, giving hundreds of millions of people access to published information and data.

Third, whereas early use of the internet by the populace consisted mainly of consuming published content and communication with other people, soon the owners of many websites and applications realized that the aggregation of actions of their users provided valuable insight into the success of their own product and sometimes human behavior in general. These sites began to collect user data in the form of clicks, typed text, site visits, and any other actions a user might take. Users began to produce more data than they consumed.

Fourth, the advent of mobile devices and smartphones that are connected to the internet made possible an enormous advance in the amount and specificity of user data being collected. At any given moment, your mobile device is capable of recording and transmitting every bit of information that its sensors can collect (location, movement, camera image and video, sound, and so on) as well as every action that you take deliberately while using the device. This can potentially be a huge amount of information, if you enable or allow its collection.

Fifth—though this isn't necessarily subsequent to the advent of personal mobile devices—is the inclusion of data collection and internet connectivity in almost everything electronic. Often referred to as the Internet of Things (IoT), these can include everything from your car to your wristwatch to the weather sensor on top of your office building. Certainly, collecting and transmitting information from devices began well before the twenty-first century, but its ubiquity is relatively new, as is the availability of the resultant data on the internet in various forms, processed or raw, for free or for sale.

Through these stages of growth of computing devices and the internet, the online world became not merely a place for consuming information but a data-collection tool in and of itself. A friend of mine in high school in the late 1990s set up a website offering electronic greeting cards as a front for collecting email addresses. He sold the resulting list of millions of email addresses for a few hundred thousand dollars. This is a primitive example of the value of user data for purposes completely unrelated to the website itself and a perfect example of something I'm sorry to have missed out on in my youth. By the early 2000s, similar-sized collections of email addresses were no longer

worth nearly this much money, but other sorts of user data became highly desirable and could likewise fetch high prices.

3.1.2 Data for its own sake

As people and businesses realized that user data could be sold for considerable sums of money, they began to collect it indiscriminately. Very large quantities of data began to pile up in data stores everywhere. Online retailers began to store not only everything you bought but also every item you viewed and every link you clicked. Video games stored every step your avatar ever took and which opponents it vanquished. Various social networks stored everything you and your friends ever did.

The purpose of collecting all of this data wasn't always to sell it, though that happens frequently. Because virtually every major website and application uses its own data to optimize the experience and effectiveness of users, site and app publishers are typically torn between the value of the data as something that can be sold and the value of the data when held and used internally. Many publishers are afraid to sell their data because that opens up the possibility that someone else will figure out something lucrative to do with it. Many of them keep their data to themselves, hoarding it for the future, when they supposedly will have enough time to wring all value out of it.

Internet juggernauts Facebook and Amazon collect vast amounts of data every minute of every day, but in my estimation, the data they possess is largely unexploited. Facebook is focused on marketing and advertising revenue, when they have one of the largest data sets comprising human behavior all around the world. Product designers, marketers, social engineers, and sociologists alike could probably make great advances in their fields, both academic and industrial, if they had access to Facebook's data. Amazon, in turn, has data that could probably upend many beloved economic principles and create several new ones if it were turned over to academic institutions. Or it might be able to change the way retail, manufacturing, and logistics work throughout the entire industry.

These internet behemoths know that their data is valuable, and they're confident that no one else possesses similar data sets of anywhere near the same size or quality. Innumerable companies would gladly pay top dollar for access to the data, but Facebook and Amazon have—I surmise—aspirations of their own to use their data to its fullest extent and therefore don't want anyone else to grab the resulting profits. If these companies had unlimited resources, surely they would try to wring every dollar out of every byte of data. But no matter how large and powerful they are, they're still limited in resources, and they're forced to focus on the uses of the data that affect their bottom lines most directly, to the exclusion of some otherwise valuable efforts.

On the other hand, some companies have elected to provide access to their data. Twitter is a notable example. For a fee, you can access the full stream of data on the Twitter platform and use it in your own project. An entire industry has developed

around brokering the sale of data, for profit. A prominent example of this is the market of data from various major stock exchanges, which has long been available for purchase.

Academic and nonprofit organizations often make data sets available publicly and for free, but there may be limitations on how you can use them. Because of the disparity of data sets even within a single scientific field, there has been a trend toward consolidation of both location and format of data sets. Several major fields have created organizations whose sole purpose is to maintain databases containing as many data sets as possible from that field. It's often a requirement that authors of scientific articles submit their data to one of these canonical data repositories prior to publication of their work.

In whichever form, data is now ubiquitous, and rather than being merely a tool that analysts might use to draw conclusions, it has become a purpose of its own. Companies now seem to collect data as an end, not a means, though many of them claim to be planning to use the data in the future. Independent of other defining characteristics of the Information Age, data has gained its own role, its own organizations, and its own value.

3.1.3 Data scientist as explorer

In the twenty-first century, data is being collected at unprecedented rates, and in many cases it's not being collected for a specific purpose. Whether private, public, for free, for sale, structured, unstructured, big, normal size, social, scientific, passive, active, or any other type, data sets are accumulating everywhere. Whereas for centuries data analysts collected their own data or were given a data set to work on, for the first time in history many people across many industries are collecting data first and then asking, "What can I do with this?" Still others are asking, "Does the data already exist that can solve my problem?"

In this way data—all data everywhere, as a hypothetical aggregation—has become an entity worthy of study and exploration. In years past, data sets were usually collected deliberately, so that they represented some intentional measurement of the real world. But more recently the internet, ubiquitous electronic devices, and a latent fear of missing out on hidden value in data have led us to collect as much data as possible, often on the loose premise that we *might* use it later.

Figure 3.2 shows an interpretation of four major innovation types in computing history: computing power itself, networking and communication between computers, collection and use of big data, and rigorous statistical analysis of that big data. By *big data*, I mean merely the recent movement to capture, organize, and use any and all data possible. Each of these computing innovations begins with a problem that begs to be addressed and then goes through four phases of development, in a process that's similar to the technological surge cycle of Carlota Perez (*Technological Revolutions and Financial Capital*, Edward Elgar Publishing, 2002) but with a focus on computing innovation and its effect on computer users and the general public.

Innovation type	The stages of computing innovation				
	Problem	Innovation	Proof/ recognition	Adoption	Refinement
Computing	Cracking codes High-powered physics Ship navigation	Pre-1950s • Purpose-built computing machines	~1950s • Enigma • ENIAC	~1970s • First PCs • Computers in schools and libraries	~1980s • Supercomputers • Consumer PCs
Networking	Communicating and sending text and files	~1970s • pre-internet • ARPANET	~1980s • Academic networks • IRC	1990s • Prodigy • CompuServe • AOL	2000s • Mobile devices • Social networks • Cloud services
Big data collection and use	Too much useful data being thrown away	~2000 • Web crawling • Click tracking • Early, big social networks	2000s • Google search • Big retailers tracking users	2010s • Twitter firehose • Hadoop	2015+ • Massive API development • Format standardization
Big data statistical analysis	Even basic statistics are hard to calculate on large data sets	2000s • Google search • Amazon streamlining processes	2010s • Netflix challenge • Kaggle.com	2015+ • Google Analytics • Budding analytics start-ups	2025+? • Ubiquitous intelligent, integrated systems

Figure 3.2 We're currently in the refinement phase of big data collection and use and in the widespread adoption phase of statistical analysis of big data.

For each innovation included in the figure, there are five stages:

- 1 *Problem*—There is a problem that computers can address in some way.
- 2 *Invention*—The computing technology that can address that problem is created.
- 3 *Proof/recognition*—Someone uses the computing technology in a meaningful way, and its value is proven or at least recognized by some experts.
- 4 *Adoption*—The newly proven technology is widely put to use in industry.
- 5 *Refinement*—People develop new versions, more capabilities, higher efficiency, integrations with other tools, and so on.

Because we're currently in the refinement phase of big data collection and the widespread adoption phase of statistical analysis of that data, we've created an entire data ecosystem wherein the knowledge that has been extracted is only a very small portion of the total knowledge contained. Not only has much of the knowledge not been extracted yet, but in many cases the full extent and properties of the data set are not understood by anyone except maybe a few software engineers who set up the system; the only people who might understand what's contained in the data are people who

are probably too busy or specialized to make use of it. The aggregation of all of this underutilized or poorly understood data to me is like an entirely new continent with many undiscovered species of plants and animals, some entirely unfamiliar organisms, and possibly a few legacy structures left by civilizations long departed.

There are exceptions to this characterization. Google, Amazon, Facebook, and Twitter are good examples of companies that are ahead of the curve. They are, in some cases, engaging in behavior that matches a later stage of innovation. For example, by allowing access to its entire data set (often for a fee), Twitter seems to be operating within the *refinement* stage of big data collection and use. People everywhere are trying to squeeze every last bit of knowledge out of users' tweets. Likewise, Google seems to be doing a good job of analyzing its data in a rigorous statistical manner. Its work on search-by-image, Google Analytics, and even its basic text search are good examples of solid statistics on a large scale. One can easily argue that Google has a long way to go, however. If today's ecosystem of data is like a largely unexplored continent, then the data scientist is its explorer. Much like famous early European explorers of the Americas or Pacific islands, a good explorer is good at several things:

- Accessing interesting areas
- Recognizing new and interesting things
- Recognizing the signs that something interesting might be close
- Handling things that are new, unfamiliar, or sensitive
- Evaluating new and unfamiliar things
- Drawing connections between familiar things and unfamiliar things
- Avoiding pitfalls

An explorer of a jungle in South America may have used a machete to chop through the jungle brush, stumbled across a few loose-cut stones, deduced that a millennium-old temple was nearby, found the temple, and then learned from the ruins about the religious rituals of the ancient tribe. A data scientist might hack together a script that pulls some social networking data from a public API, realize that a few people compose major hubs of social activity, discover that those people often mention a new photo-sharing app in their posts on the social network, pull more data from the photo-sharing app's public API, and in combining the two data sets with some statistical analysis learn about the behavior of network influencers in online communities. Both cases derive previously unknown information about how a society operates. Like an explorer, a modern data scientist typically must survey the landscape, take careful note of surroundings, wander around a bit, and dive into some unfamiliar territory to see what happens. When they find something interesting, they must examine it, figure out what it can do, learn from it, and be able to apply that knowledge in the future. Although analyzing data isn't a new field, the existence of data everywhere—often regardless of whether anyone is making use of it—enables us to apply the scientific method to discovery and analysis of a pre-existing world of data. This, to me, is the differentiator between data science and all of its predecessors. There's so much data that no one can possibly understand it all, so we treat it as a world unto itself, worthy of exploration.

This idea of data as a wilderness is one of the most compelling reasons for using the term *data science* instead of any of its counterparts. To get real truth and useful answers from data, we must use the scientific method, or in our case, the *data scientific method*:

- 1 Ask a question.
- 2 State a hypothesis about the answer to the question.
- 3 Make a testable prediction that would provide evidence in favor of the hypothesis if correct.
- 4 Test the prediction via an experiment involving data.
- 5 Draw the appropriate conclusions through analyses of experimental results.

In this way, data scientists are merely doing what scientists have been doing for centuries, albeit in a digital world. Today, some of our greatest explorers spend their time in virtual worlds, and we can gain powerful knowledge without ever leaving our computers.

3.2 Where data might live, and how to interact with it

Before we dive in to the unexplored wilderness that is the state of data today, I'd like to discuss the forms that data might take, what those forms mean, and how we might treat them initially. Flat files, XML, and JSON are a few data formats, and each has its own properties and idiosyncrasies. Some are simpler than others or more suited to certain purposes. In this section, I discuss several types of formats and storage methods, some of their strengths and weaknesses, and how you might take advantage of them.

Although plenty of people will object to this, I decided to include in this section a discussion of databases and APIs as well. Commingling a discussion of file formats with software tools for data storage makes sense to me because at the beginning of a data science project, any of these formats or data sources is a valid answer to the question "Where is the data now?" File, database, or API, what the data scientist needs to know is "How do I access and extract the data I need?" and so that's my purpose here.

Figure 3.3 shows three basic ways a data scientist might access data. It could be a file on a file system, and the data scientist could read the file into their favorite analysis tool. Or the data could be in a database, which is also on a file system, but in order to access the data, the data scientist has to use the database's interface, which is a software layer that helps store and extract data. Finally, the data could be behind an application programming interface (API), which is a software layer between the data scientist and some system that might be completely unknown or foreign. In all three cases, the data can be stored and/or delivered to the data scientist in any format I discuss in this section or any other. Storage and delivery of data are so closely intertwined in some systems that I choose to treat them as a single concept: getting data into your analysis tools.

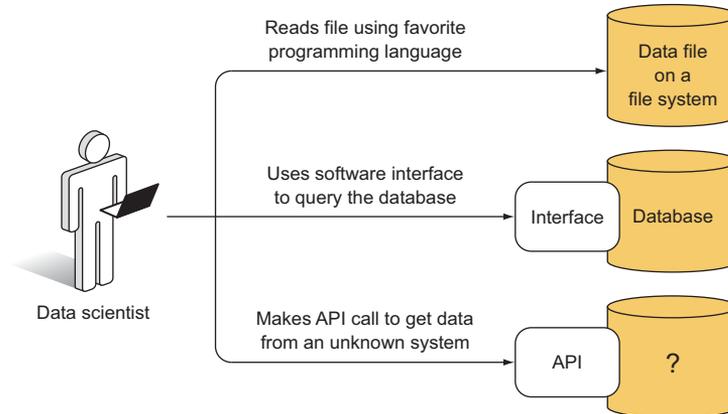


Figure 3.3 Three ways a data scientist might access data: from a file system, database, or API

In no way do I purport to cover all possible data formats or systems, nor will I list all technical details. My goal here is principally to give descriptions that would make a reader feel comfortable talking about and approaching each one. I can still remember when extracting data from a conventional database was daunting for me, and with this section I'd like to put even beginners at ease. Only if you're fairly comfortable with these basic forms of data storage and access can you move along to the most important part of data science: what the data can tell you.

3.2.1 Flat files

Flat files are plain-vanilla data sets, the default data format in most cases if no one has gone through the effort to implement something else. Flat files are self-contained, and you don't need any special programs to view the data contained inside. You can open a flat file for viewing in a program typically called a text editor, and many text editors are available for every major operating system. Flat files contain ASCII (or UTF-8) text, each character of text using (most likely) 8 bits (1 byte) of memory/storage. A file containing only the word `DATA` will be of size 32 bits. If there is an end-of-line (EOL) character after the word `DATA`, the file will be 40 bits, because an EOL character is needed to signify that a line has ended. My explanation of this might seem simplistic to many people, but even some of these basic concepts will become important later on as we discuss other formats, so I feel it's best to outline some basic properties of the flat file so that we might compare other data formats later.

There are two main subtypes of the flat file: plain text and delimited. *Plain text* is words, as you might type them on your keyboard. It could look like this:

```
This is what a plain text flat file looks like. It's just plain ASCII text.
Lines don't really end unless there is an end-of-line character, but some
text editors will wrap text around anyway, for convenience.
```

Usually, every character is a byte, and so there are only 256 possible characters, but there are a lot of caveats to that statement, so if you're really interested, consult a reference about ASCII and UTF-8.

This file would contain seven lines, or technically eight if there's an end-of-line character at the end of the final line of text. A plain text flat file is a bunch of characters stored in one of two (or so) possible very common formats. This is not the same as a text document stored in a word processor format, such as Microsoft Word or OpenOffice Writer. (See the subsection "Common bad formats.") Word processor file formats potentially contain much more information, including overhead such as style formats and metadata about the file format itself, as well as objects like images and tables that may have been inserted into a document. Plain text is the minimal file format for containing words and only words—no style, no fancy images. Numbers and some special characters are OK too.

But if your data contains numerous entries, a delimited file might be a better idea. A *delimited file* is plain text but with the stipulation that every so often in the file a delimiter will appear, and if you line up the delimiters properly, you can make something that looks like a table, with rows, columns, and headers. A delimited file might look like this:

NAME	ID	COLOR	DONE
Alison	1	'blue'	FALSE
Brian	2	'red'	TRUE
Clara	3	'brown'	TRUE

Let's call this table JOBS_2015, because it represents a fictional set of house-painting jobs that started in 2015, with the customer name, ID, paint color, and completion status.

This table happens to be tab-delimited—or tab-separated value (TSV)—meaning that columns are separated by the tab character. If opened in a text editor, such a file would usually appear as it does here, but it might optionally display the text `\t` where a tab would otherwise appear. This is because a tab, like an end-of-line character, can be represented with a single ASCII character, and that character is typically represented with `\t`, if not rendered as variable-length whitespace that aligns characters into a tabular format.

If JOBS_2015 were stored as a comma-separated value (CSV) format, it would appear like this in a standard text editor:

```
NAME, ID, COLOR, DONE
Alison, 1, 'blue', FALSE
Brian, 2, 'red', TRUE
Clara, 3, 'brown', TRUE
```

The commas have taken the place of the tab characters, but the data is still the same. In either case, you can see that the data in the file can be interpreted as a set of rows and columns. The rows represent one job each for Alison, Brian, and Clara, and the

column names on the header (first) line are `NAME`, `ID`, `COLOR`, and `DONE`, giving the types of details of the job contained within the table.

Most programs, including spreadsheets and some programming languages, require the same number of delimiters on each line (except possibly the header line) so that when they try to read the file, the number of columns is consistent, and each line contributes exactly one entry to each column. Some software tools don't require this, and they each have specific ways of dealing with varying numbers of entries on each line.

I should note here that delimited files are typically interpreted as tables, like spreadsheets. Furthermore, as plain text files can be read and stored using a word processing program, delimited files can typically be loaded into a spreadsheet program like Microsoft Excel or OpenOffice Calc.

Any common program for manipulating text or tables can read flat files. Popular programming languages all include functions and methods that can read such files. My two most familiar languages, Python (the `csv` package) and R (the `read.table` function and its variants), contain methods that can easily load a CSV or TSV file into the most relevant data types in those languages. For plain text also, Python (`readlines`) and R (`readLines`) have methods that read a file line by line and allow for the parsing of the text via whatever methods you see fit. Packages in both languages—and many others—provide even more functionality for loading files of related types, and I suggest looking at recent language and package documentation to find out whether another file-loading method better suits your needs.

Without compressing files, flat files are more or less the smallest and simplest common file formats for text or tables. Other file formats contain other information about the specifics of the file format or the data structure, as appropriate. Because they're the simplest file formats, they're usually the easiest to read. But because they're so lean, they provide no additional functionality other than showing the data, so for larger data sets, flat files become inefficient. It can take minutes or hours for a language like Python to scan a flat file containing millions of lines of text. In cases where reading flat files is too slow, there are alternative data storage systems designed to parse through large amounts of data quickly. These are called *databases* and are covered in a later section.

3.2.2 **HTML**

A *markup language* is plain text marked up with tags or specially denoted instructions for how the text should be interpreted. The very popular Hypertext Markup Language (HTML) is used widely on the internet, and a snippet might look like this:

```
<html>
  <body>
    <div class="column">
      <h1>Column One</h1>
      <p>This is a paragraph</p>
    </div>
```

```
<div class="column">
  <h1>Column Two</h1>
  <p>This is another paragraph</p>
</div>
</body>
</html>
```

An HTML interpreter knows that everything between the `<html>` and `</html>` tags should be considered and read like HTML. Similarly, everything between the `<body>` and `</body>` tags will be considered the body of the document, which has special meaning in HTML rendering. Most HTML tags are of the format `<TAGNAME>` to begin the annotation and `</TAGNAME>` to end it, for an arbitrary TAGNAME. Everything between the two tags is now treated as being annotated by TAGNAME, which an interpreter can use to render the document. The two `<div>` tags in the example show how two blocks of text and other content can be denoted, and a class called `column` is applied to the `div`, allowing the interpreter to treat a `column` instance in a special way.

HTML is used primarily to create web pages, and so it usually looks more like a document than a data set, with a header, body, and some style and formatting information. HTML is not typically used to store raw data, but it's certainly capable of doing so. In fact, the concept of *web scraping* usually entails writing code that can fetch and read web pages, interpreting the HTML, and scraping out the specific pieces of the HTML page that are of interest to the scraper.

For instance, suppose we're interested in collecting as many blog posts as possible and that a particular blogging platform uses the `<div class="column">` tag to denote columns in blog posts. We could write a script that systematically visits a blog, interprets the HTML, looks for the `<div class="column">` tag, captures all text between it and the corresponding `</div>` tag, and discards everything else, before proceeding to another blog to do the same. This is web scraping, and it might come in handy if the data you need isn't contained in one of the other more friendly formats. Web scraping is sometimes prohibited by website owners, so it's best to be careful and check the copyright and terms of service of the site before scraping.

3.2.3 XML

Extensible Markup Language (XML) can look a lot like HTML but is generally more suitable for storing and transmitting documents and data other than web pages. The previous snippet of HTML can be valid XML, though most XML documents begin with a tag that declares a particular XML version, such as the following:

```
<?xml version="1.0" encoding="UTF-8"?>
```

This declaration helps ensure that an XML interpreter reads tags in the appropriate way. Otherwise, XML works similarly to HTML but without most of the overhead associated with web pages. XML is now used as a standard format for offline documents such as the OpenOffice and Microsoft Office formats. Because the XML specification

is designed to be machine-readable, it also can be used for data transmission, such as through APIs. For example, many official financial documents are available publicly in the Extensible Business Reporting Language (XBRL), which is XML-based.

This is a representation of the first two rows of the table `JOBS_2015` in XML:

```
<JOB>
  <NAME>Alison</NAME>
  <ID>1</ID>
  <COLOR>'blue'</COLOR>
  <DONE>FALSE</DONE>
</JOB>
<JOB>
  <NAME>Brian</NAME>
  <ID>2</ID>
  <COLOR>'red'</COLOR>
  <DONE>TRUE</DONE>
</JOB>
```

You can see that each row of the table is denoted by a `<JOB>` tag, and within each `JOB`, the table's column names have been used as tags to denote the various fields of information. Clearly, storing data in this format takes up more disk space than a standard table because XML tags take up disk space, but XML is much more flexible, because it's not limited to a row-and-column format. For this reason, it has become popular in applications and documents using non-tabular data and other formats requiring such flexibility.

3.2.4 JSON

Though not a markup language, JavaScript Object Notation (JSON) is functionally similar, at least when storing or transmitting data. Instead of describing a document, JSON typically describes something more like a data structure, such as a list, map, or dictionary in many popular programming languages. Here's the data from the first two rows of the table `JOBS_2015` in JSON:

```
[
  {
    NAME: "Alison",
    ID: 1,
    COLOR: "blue",
    DONE: False
  },
  {
    NAME: "Brian",
    ID: 2,
    COLOR: "red",
    DONE: True
  }
]
```

In terms of structure, this JSON representation looks a lot like the XML representation you've already seen. But the JSON representation is leaner in terms of the number of

characters needed to express the data, because JSON was designed to represent data objects and not as a document markup language. Therefore, for transmitting data, JSON has become very popular. One huge benefit of JSON is that it can be read directly as JavaScript code, and many popular programming languages including Python and Java have natural representations of JSON as native data objects. For interoperability between programming languages, JSON is almost unparalleled in its ease of use.

3.2.5 Relational databases

Databases are data storage systems that have been optimized to store and retrieve data as efficiently as possible within various contexts. Theoretically, a relational database (the most common type of database) contains little more than a set of tables that could likewise be represented by a delimited file, as already discussed: row and column names and one data point per row-column pair. But databases are designed to search—or *query*, in the common jargon—for specific values or ranges of values within the entries of the table.

For example, let's revisit the JOBS_2015 table:

NAME	ID	COLOR	DONE
Alison	1	'blue'	FALSE
Brian	2	'red'	TRUE
Clara	3	'brown'	TRUE

But this time assume that this table is one of many stored in a database. A database query could be stated in plain English as follows:

```
From JOBS_2015, show me all NAME in rows where DONE=TRUE
```

This query should return the following:

```
Brian  
Clara
```

That's a basic query, and every database has its own language for expressing queries like this, though many databases share the same basis query language, the most common being Structured Query Language (SQL).

Now imagine that the table contains millions of rows and you'd like to do a query similar to the one just shown. Through some tricks of software engineering, which I won't discuss here, a well-designed database is able to retrieve a set of table rows matching certain criteria (a query) much faster than a scan of a flat file would. This means that if you're writing an application that needs to search for specific data very often, you may improve retrieval speed by orders of magnitude if you use a database instead of a flat file.

The main reason why databases are good at retrieving specific data quickly is the database index. A *database index* is itself a data structure that helps that database software find relevant data quickly. It's like a structural map of the database content,

which has been sorted and stored in a clever way and might need to be updated every time data in the database changes. Database indexes are not universal, however, meaning that the administrator of the database needs to choose which columns of the tables are to be indexed, if the default settings aren't appropriate. The columns that are chosen to be indexed are the ones upon which querying will be most efficient, and so the choice of index is an important one for the efficiency of your applications that use that database.

Besides querying, another operation that databases are typically good at is joining tables. Querying and joining aren't the only two things that databases are good at, but they're by far the most commonly utilized reasons to use a database over another data storage system. *Joining*, in database jargon, means taking two tables of data and combining them to make another table that contains some of the information of both of the original tables.

For example, assume you have the following table, named `CUST_ZIP_CODES`:

<code>CUST_ID</code>	<code>ZIP_CODE</code>
1	21230
2	45069
3	21230
4	98033

You'd like to investigate which paint colors have been used in which ZIP codes in 2015. Because the colors used on the various jobs are in `JOBS_2015` and the customers' ZIP codes are in `CUST_ZIP_CODES`, you need to join the tables in order to match colors with ZIP codes. An inner join matching `ID` from table `JOBS_2015` and `CUST_ID` from table `CUST_ZIP_CODES` could be stated in plain English:

```
JOIN tables JOBS_2015 and CUST_ZIP_CODES where ID equals CUST_ID, and
show me ZIP_CODE and COLOR.
```

You're telling the database to first match up the customer ID numbers from the two tables and then show you only the two columns you care about. Note that there are no duplicate column names between the two tables, so there's no ambiguity in naming. But in practice you'd normally have to use a notation like `CUST_ZIP_CODES.CUST_ID` to denote the `CUST_ID` column of `CUST_ZIP_CODES`. I use the shorter version here for brevity.

The result of the join would look like this:

<code>ZIP_CODE</code>	<code>COLOR</code>
21230	'blue'
45069	'red'
21230	'brown'

Joining can be a very big operation if the original tables are big. If each table had millions of different IDs, it could take a long time to sort them all and match them up. Therefore, if you're joining tables, you should minimize the size of those tables (primarily by number of rows) because the database software will have to shuffle all rows

of both tables based on the join criteria until all appropriate combinations of rows have been created in the new table. Joins should be done sparingly and with care.

It's a good general rule, if you're going to query *and* join, to query the data *first* before joining. For example, if you care only about the COLOR in ZIP_CODE 21230, it's usually better to query CUST_ZIP_CODES for ZIP_CODE=21230 first and join the result with JOBS_2015 instead of joining first and then querying. This way, there might be far less matching to do, and the execution of the operation will be much faster overall. For more information and guidance on optimizing database operations, you'll find plenty of practical database books in circulation.

You can think of databases in general as well-organized libraries, and their indexes are good librarians. A librarian can find the book you need in a matter of seconds, when it might have taken you quite a long time to find it yourself. If you have a relatively large data set and find that your code or software tool is spending a lot of time searching for the data it needs at any given moment, setting up a database is certainly worth considering.

3.2.6 Non-relational databases

Even if you don't have tabular data, you might still be able to make use of the efficiency of database indexing. An entire genre of databases called *NoSQL* (often interpreted as "Not only SQL") allows for database schemas outside the more traditional SQL-style relational databases. Graph databases and document databases are typically classified as NoSQL databases.

Many NoSQL databases return query results in familiar formats. Elasticsearch and MongoDB, for instance, return results in JSON format (discussed in section 3.2.4). Elasticsearch in particular is a document-oriented database that's very good at indexing the contents of text. If you're working with numerous blog posts or books, for example, and you're performing operations such as counting the occurrences of words within each blog post or book, then Elasticsearch is typically a good choice, if indexed properly.

Another possible advantage of some NoSQL databases is that, because of the flexibility of the schema, you can put almost anything into a NoSQL database without much hassle. Strings? Maps? Lists? Sure! Why not? MongoDB, for instance, is extremely easy to set up and use, but then you do lose some performance that you might have gained by setting up a more rigorous index and schema that apply to your data.

All in all, if you're working with large amounts of non-tabular data, there's a good chance that someone has developed a database that's good at indexing, querying, and retrieving your type of data. It's certainly worth a quick look around the internet to see what others are using in similar cases.

3.2.7 APIs

An *application programming interface* (API) in its most common forms is a set of rules for communicating with a piece of software. With respect to data, think of an API as a

gateway through which you can make requests and then receive the data, using a well-defined set of terms. Databases have APIs; they define the language that you must use in your query, for instance, in order to receive the data you want.

Many websites also have APIs. Tumblr, for instance, has a public API that allows you to ask for and receive information about Tumblr content of certain types, in JSON format. Tumblr has huge databases containing all the billions of posts hosted on its blogging service. But it has decided what you, as a member of the public, can and can't access within the databases. The methods of access and the limitations are defined by the API.

Tumblr's API is a REST API accessible via HTTP. I've never found the technical definition of *REST API* to be helpful, but it's a term that people use when discussing APIs that are accessible via HTTP—meaning you can usually access them from a web browser—and that respond with information in a familiar format. For instance, if you register with Tumblr as a developer (it's free), you can get an API key. This API key is a string that's unique to you, and it tells Tumblr that you're the one using the API whenever you make a request. Then, in your web browser, you can paste the URL http://api.tumblr.com/v2/blog/good.tumblr.com/info?api_key=API_KEY, which will request information about a particular blog on Tumblr (replacing `API_KEY` with the API key that you were given). After you press Enter, the response should appear in your browser window and look something like this (after some reformatting):

```
{
  meta:
  {
    status: 200,
    msg: "OK"
  },
  response:
  {
    blog:
    {
      title: "",
      name: "good",
      posts: 2435,
      url: "http://good.tumblr.com/",
      updated: 1425428288,
      description: "<font size='6'>
        GOOD is a magazine for the global citizen.
      </font>",
      likes: 429
    }
  }
}
```

This is JSON with some HTML in the description field. It contains some metadata about the status of the request and then a response field containing the data that was requested. Assuming you know how to parse JSON strings (and likewise HTML), you can use this in a programmatic way. If you were curious about the number of likes of

Tumblr blogs, you could use this API to request information about any number of blogs and compare the numbers of likes that they have received. You wouldn't want to do that, though, from your browser window, because it would take a very long time.

In order to capture the Tumblr API response programmatically, you need to use an HTTP or URL package in your favorite programming language. In Python there is `urllib`, in Java `URLConnection`, and R has `url`, but there are many other packages for each of these languages that perform similar tasks. In any case, you'll have to assemble the request URL (as a string object/variable) and then pass that request to the appropriate URL retrieval method, which should return a response similar to the previous one that can be captured as another object/variable. Here's an example in Python:

```
import urllib
requestURL = \
    'http://api.tumblr.com/v2/blog/good.tumblr.com/info?api_key=API_KEY'
response = urllib.urlopen(requestURL)
```

After running these lines, the variable `response` should contain a JSON string that looks similar to the response shown.

I remember learning how to use an API like this one from Python, and I was a bit confused and overwhelmed at first. Getting the request URL exactly right can be tricky if you're assembling it programmatically from various parts (for example, base URL, parameters, API key, and so on). But being adept at using APIs like this one can be one of the most powerful tools in data collection, because so much data is available through these gateways.

3.2.8 Common bad formats

It's no secret that I'm not a fan of the typical suites of office software: word processing programs, spreadsheets, mail clients. Thankfully, I'm not often required to use them. I avoid them whenever possible and never more so than when doing data science. That doesn't mean that I won't deal with those files; on the contrary, I wouldn't throw away free data. But I make sure to get away from any inconvenient formats as quickly as possible. There usually isn't a good way to interact with them unless I'm using the highly specialized programs that were built for them, and these programs typically aren't capable of the analysis that a data scientist usually needs. I can't remember the last time I did (or saw) a solid bit of data science in Microsoft Excel; to me, Excel's methods for analysis are limited, and the interface is unwieldy for anything but looking at tables. But I know I'm biased, so don't mind me if you're convinced you can do rigorous analysis within a spreadsheet. OpenOffice Calc and Microsoft Excel both allow you to export individual sheets from a spreadsheet into CSV formats. If a Microsoft Word document contains text I'd like to use, I export it either into plain text or maybe HTML or XML.

A PDF can be a tricky thing as well. I've exported lots of text (or copied and pasted) from PDFs into plain text files that I then read into a Python program. This is

one of my favorite examples of *data wrangling*, a topic I devote an entire chapter to, and so for now it will suffice to say that exporting or scraping text from a PDF (where possible) is usually a good idea whenever you want to analyze that text.

3.2.9 **Unusual formats**

This is the umbrella category for all data formats and storage systems with which I'm unfamiliar. All sorts of formats are available, and I'm sure someone had a good reason to develop them, but for whatever reason they're not familiar to me. Sometimes they're archaic; maybe they were superseded by another format, but some legacy data sets haven't yet been updated.

Sometimes the formats are highly specialized. I once participated in a project exploring the chemical structure of a compound and its connection to the way the compound smelled (its aroma). The `RDKit` package (www.rdkit.org) provided a ton of helpful functionality for parsing through chemical structures and substructures. But much of this functionality was highly specific to chemical structure and its notation. Plus the package made heavy use of a fairly sophisticated binary representation of certain aspects of chemical structure that greatly improved the computational efficiency of the algorithms but also made them extremely difficult to understand.

Here's what I do when I encounter a data storage system unlike anything I've seen before:

- 1 Search and search (and search) online for a few examples of people doing something similar to what I want to do. How difficult might it be to adapt these examples to my needs?
- 2 Decide how badly I want the data. Is it worth the trouble? What are the alternatives?
- 3 If it's worth it, I try to generalize from the similar examples I found. Sometimes I can gradually expand from examples by fiddling with parameters and methods. I try a few things and see what happens.

Dealing with completely unfamiliar data formats or storage systems can be its own type of exploration, but rest assured that someone somewhere has accessed the data before. If no one has ever accessed the data, then someone was completely mistaken in creating the data format in the first place. When in doubt, send a few emails and try to find someone who can help you.

3.2.10 **Deciding which format to use**

Sometimes you don't have a choice. The data comes in a certain format, and you have to deal with it. But if you find that format inefficient, unwieldy, or unpopular, you're usually free to set up a secondary data store that might make things easier, but at the additional cost of the time and effort it takes you to set up the secondary data store. For applications where access efficiency is critical, the cost can be worth it. For smaller projects, maybe not. You'll have to cross that bridge when you get there.

I'll conclude this section with a few general rules about what data formats to use, when you have a choice, and in particular when you're going to be accessing the data from a programming language. Table 3.1 gives the most common good format for interacting with data of particular types.

Table 3.1 Some common types of data and formats that are good for storing them

Type of data	Good, common format
Tabular data, small amount	Delimited flat file
Tabular data, large amount with lots of searching/querying	Relational database
Plain text, small amount	Flat file
Plain text, large amount	Non-relational database with text search capabilities
Transmitting data between components	JSON
Transmitting documents	XML

And here are a few guidelines for choosing or converting data formats:

- For spreadsheets and other office documents, export!
- More common formats are usually better for your data type and application.
- Don't spend too much time converting from a certain format to your favorite; weigh the costs and benefits first.

Now that I've covered many of the forms in which data might be presented to you, hopefully you'll feel somewhat comfortable in a high-level conversation about data formats, stores, and APIs. As always, never hesitate to ask someone for details about a term or system you haven't heard of before. New systems are being developed constantly, and in my experience, anyone who recently learned about a system is usually eager to help others learn about it.

3.3 Scouting for data

The previous section discussed many of the common forms that data takes, from file formats to databases to APIs. I intended to make these data forms more approachable, as well as to increase awareness about the ways you might look for data. It's not hard to find data, much like it's not hard to find a tree or a river (in certain climates). But finding the data that can help you solve your problem is a different story. Or maybe you already have data from an internal system. It may seem like that data can answer the major questions of your project, but you shouldn't take it for granted. Maybe a data set out there will perfectly complement the data you already have and greatly improve results. There's so much data on the internet and elsewhere; some part of it should be able to help you. Even if not, a quick search is certainly worth it, even for a long-shot possibility.

In this section, I discuss the act of looking for data that might help you with your project. This is the exploration I talked about at the beginning of this chapter. Now that you have some exposure to common forms of data from the previous section, you can focus less on the format and more on the content and whether it can help you.

3.3.1 First step: Google search

This may seem obvious, but I still feel like mentioning it: Google searches are not perfect. To make them work as well as possible, you have to know what to search for and what you're looking for in the search results. Given the last section's introduction to data formats, you now have a little more ammunition for Google searches than before.

A Google search for "Tumblr data" gives different results from a search for "Tumblr API." I'm not sure which I prefer, given that I don't have a specific project involving Tumblr at the moment. The former returns results involving the term *data* as used on Tumblr posts as well as third parties selling historical Tumblr data. The latter returns results that deal almost exclusively with the official Tumblr API, which contains considerable up-to-the-minute information about Tumblr posts. Depending on your project, one might be better than the other.

But it's definitely worth keeping in mind that terms such as *data* and *API* do make a difference in web searches. Try the searches "social networking" and "social networking API." There's a dramatic difference in results.

Therefore, when searching for data related to your project, be sure to include modifying terms like *historical*, *API*, *real time*, and so on, because they do make a difference. Likewise, watch out for them in the search results. This may be obvious, but it makes a considerable difference in your ability to find what you're looking for, and so it's worth repeating.

3.3.2 Copyright and licensing

I've talked about searching for, accessing, and using data, but there's another very important concern: are you *allowed* to use it?

As with software licenses, data may have licensing, copyright, or other restrictions that can make it illegal to use the data for certain purposes. If the data comes from academic sources, for example (universities, research institutions, and the like), then there's often a restriction that the data can't be used for profit. Proprietary data, such as that of Tumblr or Twitter, often comes with the restriction that you can't use the data to replicate functionality that the platform itself provides. You may not be able to make a Tumblr client that does the same things as the standard Tumblr platform, but perhaps if you offer other functionality not included in the platform, there would be no restriction. Restrictions like these are tricky, and it's best to read any legal documentation that the data provider offers. In addition, it's usually good to search for other examples of people and companies using the data in a similar way and see if there are any references to legal concerns. Precedent is no guarantee that a particular

use of the data is legally sound, but it may provide guidance in your decision to use the data or not.

All in all, you should remain keenly aware that most data sets not owned by you or your organization come with restrictions on use. Without confirming that your use case is legal, you remain at risk of losing access to the data or, even worse, a lawsuit.

3.3.3 The data you have: is it enough?

Let's say you've found data and confirmed that you're allowed to use it for your project. Should you keep looking for more data, or should you attack the data you have immediately? The answer to this question is—like pretty much everything in data science—tricky. In this case, the answer is tricky because data sets aren't always what they seem to be or what you want them to be. Take the example of Uber, the taxi service app publisher. I recently read that Uber was compelled (upon losing an appeal) to turn over trip data to New York City's Taxi and Limousine Commission (TLC). Suppose you're an employee of the TLC, and you'd like to compare Uber with traditional taxi services in regard to the number of trips taken by riders over many specific routes. Given that you have data from both Uber and traditional taxis, it may seem straightforward to compare the number of trips for similar routes between the two types of car services. But once you begin your analysis, you realize that Uber had provided pick-up and drop-off locations in terms of ZIP codes, which happen to be the minimum specificity required by the TLC. ZIP codes can cover large areas, though admittedly less so in New York City than anywhere else. Addresses, or at least city blocks, would have been considerably better from a data analysis perspective, but requiring such specificity presents legal troubles regarding the privacy of personal data of the users of taxi services, so it's understandable.

So what should you do? After the first waves of disappointment wear off, you should probably check to see whether your data will suffice after all or if you need to supplement this data and/or amend your project plans. There's often a simple way to accomplish this: can you run through a few specific examples of your intended analyses and see if it makes a significant difference?

In this taxi-versus-Uber example, you'd like to find out whether the relative non-specificity of ZIP code can still provide a useful approximation for the many routes you'd like to evaluate. Pick a specific route, say Times Square (ZIP code: 10036) to the Brooklyn Academy of Music (ZIP code: 11217). If a car travels between 10036 and 11217, what other specific routes might the rider have taken? In this case, those same ZIP codes could also describe a trip from the Intrepid Sea, Air & Space Museum to Grand Army Plaza, or likewise a trip from a restaurant in Hell's Kitchen to an apartment in Park Slope. These probably don't mean much to people outside the New York City area, but for our purposes it suffices to say that these other locations are up to a kilometer from the origin and destination of the chosen route, a distance that's about a ten-minute walk and, by NYC standards, not very short. It's up to you, the data scientist, to make a decision about whether these other locations in the same ZIP codes are

close enough or too far from their intended targets. And this decision, in turn, should be made based on the project's goals and the precise questions (from chapter 2) that you're hoping to answer.

3.3.4 Combining data sources

If you find that your data set is insufficient to answer your questions, and you can't find a data set that is sufficient, it might still be possible to combine data sets to find answers. This is yet another point that seems obvious at times but is worth mentioning because of its importance and because of a few tricky points that might pop up.

Combining two (or more) data sets can be like fitting puzzle pieces together. If the puzzle is, metaphorically, the complete data set you wish you had, then each piece of the puzzle—a data set—needs to cover precisely what the other pieces don't. Sure, unlike puzzle pieces, data sets can overlap in some sense, but any gap left after all the present pieces have been assembled is an obstacle that needs to be overcome or circumvented, either by changing the plan or some other reevaluation of how you're going to answer your questions.

Your multiple data sets might be coming in multiple formats. If you're adept at manipulating each of these formats, this doesn't usually present a problem, but it can be tough to conceptualize how the data sets relate to one another if they're in vastly different forms. A database table and a CSV file are similar to me—they both have rows and columns—and so I can typically imagine how they might fit together, as in the database example earlier in this chapter, with one data set (one of the tables) providing the customer's color choice and another data set (the other table) providing the customer's ZIP code. These two can be combined easily because both data sets are based on the same set of customer IDs. If you can imagine how you might match up the customer IDs between the two data sets and then combine the accompanying information—a *join*, in database parlance—then you can imagine how to combine these two data sets meaningfully.

On the other hand, combining data sets might not be so simple. During my time as the lead data scientist at a Baltimore analytic software firm, I took part in a project in which our team was analyzing email data sets as part of a legal investigation. The collection of emails was delivered to us in the form of a few dozen files in PST format, which is Microsoft Outlook's archival format. I'd seen this format before, because I'd worked previously with the now-public and commonly studied Enron email data set. Each archive file comprised the email from one person's computer, and because the people under investigation often emailed each other, the data sets overlapped. Each email, excepting deleted emails, was present in each of the senders' and recipients' archives. It's tempting to think that it would be easy to combine all of the email archives in to a single file—I chose a simple, large CSV file as the goal format—and then analyze this file. But it wasn't so easy.

Extracting individual emails from each archive and turning each of them into a row of a CSV file was, comparatively, the easy part. The hard part, I quickly realized,

was making sure I could keep all of the senders and recipients straight. As it turns out, the names listed in the sender and recipient fields of emails are not standardized—when you send an email, what appears in the `SENDER` field isn't always the same as what appears in the `RECIPIENT` field when someone writes an email to you. In fact, even within each of these two fields, names are not consistent. If Nikola Tesla sent an email to Thomas Edison (names have been changed to protect the innocent), the `SENDER` and `RECIPIENT` fields might be any of the following:

SENDER	RECIPIENT
Nikola Tesla <nikola@ac.org>	Thomas Edison, CEO <thomas@coned.com>
Nikola <nikola.tesla@ac.org>	thomas.edison@dc.com
ntesla@gmail.com	tommyed@comcast.com
nikola@tesla.me	Tom <t@coned.com>
wirelesspower@tesla.me	litebulbz@hotmail.com

Some of these would be recognizable as Tesla or Edison, even out of context, but others would not. To be sure each email is attributed to the right person, you'd also need a list of email addresses matched to the correct names. I didn't have that list, so I did the best I could, made some assumptions, and used some fuzzy string matching with spot-checking (discussed more in the next chapter on data wrangling) to match as many emails as possible with the appropriate names. I thought the multiple email data sets would merge nicely together, but I soon found out that this was not the case.

Data sets can differ in any number of ways; format, nomenclature, and scope (geographic, temporal, and so on) are a few. As in section 3.3.3 on finding out whether your data is enough, before you spend too much time manipulating your multiple data sets or diving into analyses, it's usually extremely helpful and informative to spot-check a few data points and attempt a quick analysis on a small scale. A quick look into a few PST files in the email example made me aware of the disparate naming schemes across files and fields and allowed me to plan within the project for the extra time and inevitable matching errors that arose.

Now imagine combining this email data set with internal chat messages in a JSON format—potentially containing a different set of user names—with a set of events/appointments in a proprietary calendar format. Assembling them into a single timeline with unambiguous user names is no simple task, but it might be possible with care and awareness of the potential pitfalls.

3.3.5 **Web scraping**

Sometimes you can find the information you need on the internet, but it's not what you might call a data set in the traditional sense. Social media profiles, like those on Facebook or LinkedIn, are great examples of data that's viewable on the internet but not readily available in a standard data format. Therefore, some people choose to scrape it from the web.

I should definitely mention that web scraping is against the terms of service for many websites. And some sites have guards in place that will shut down your access if

they detect a scraper. Sometimes they detect you because you're visiting web pages much more quickly than a human being can, such as several thousand pages in few minutes or even a few hours. Regardless, people have used scraping techniques to gather useful data they wouldn't have otherwise, in some cases circumventing any guards by adapting the scraper to act more human.

Two important things that a web scraper must do well are visit lots of URLs programmatically and capture the right information from the pages. If you wanted to know about your friend network on Facebook, you could theoretically write a script that visits the Facebook profiles of all of your friends, saves the profile pages, and then parses the pages to get lists of their friends, visits their friends' profiles, and so on. This works only for people who have allowed you to view their profiles and friend lists, and would not work for private profiles.

An example of web scraping that became popular in early 2014 is that of mathematician Chris McKinlay, who used a web scraper to capture data from thousands of profiles on the popular dating website OKCupid. He used the information he gathered—mostly women's answers to multiple-choice questions on the site—to cluster the women into a few types and subsequently optimize a separate profile for himself for each of the types he found generally attractive. Because he optimized each profile for a certain cluster/type of women, women in that cluster had a high matching score (according to OKCupid's own algorithms) for the respective profile and were therefore more likely to engage him in conversation and ultimately to go out on a date with him. It seems to have worked out well for him, earning him dozens of dates before he met the woman with whom he wanted to start a monogamous relationship.

For more on the practicalities of building a web scraper, see the documentation for the HTTP- and HTML-related utilities of your favorite programming language and any number of online guides, as well as section 3.2 on data formats, particularly the discussion of HTML.

3.3.6 *Measuring or collecting things yourself*

Contrary to the principal way I've presented data in this chapter—a product of a culture that wants data for its own sake, existing regardless of whether someone intends to use it—you sometimes have the opportunity to collect data the old-fashioned way. Methods could be as simple as personally counting the number of people crossing a street at a particular crosswalk or perhaps emailing a survey to a group of interest. When starting a new project, if you ever ask yourself, "Does the data I need exist?" and find that the answer is "No" or "Yes, but I can't get access to it," then maybe it would be helpful to ask, "*Can* the data exist?"

The question "Can the data exist?" is intended to draw attention to the potential for simple measures you can take that can create the data set you want. These include the following:

- *Measuring things in real life*—Using tape measures, counting, asking questions personally, and so on may seem outmoded, but it's often underrated.

- *Measuring things online*—Clicking around the internet and counting relevant web pages, numbers of relevant Google search results, and number of occurrences of certain terms on certain Wikipedia pages, among others, can benefit your project.
- *Scripting and web scraping*—Repeated API calls or web scraping of certain pages over a period of time can be useful when certain elements in the API or web page are constantly changing but you don't have access to the history.
- *Data-collection devices*—Today's concept of the Internet of Things gets considerable media buzz partially for its value in creating data from physical devices, some of which are capable of recording the physical world—for example, cameras, thermometers, and gyroscopes. Do you have a device (your mobile phone?) that can help you? Can you buy one?
- *Log files or archives*—Sometimes jargonized into *digital trail* or *exhaust*, log files are (or can be) left behind by many software applications. Largely untouched, they're usually called to action only in exceptional circumstances (crashes! bugs!). Can you put them to good use in your project?

For that last bullet, much like web scraping, the primary tasks are to identify manually whether and where the log files contain data that can help you and to learn how to extract this useful data programmatically from a set of log files that contain, in most cases, a bunch of other data that you'll never need. This, perhaps, is the frontier of the data wilderness: creating conceptually new data sets using other data that exists for an entirely different purpose. I believe *data alchemy* has been put forth as a possible name for this phenomenon, but I'll leave you to judge whether your own data extractions and transformations merit such a mystical title.

3.4 Example: microRNA and gene expression

When I was a PhD student, most of my research was related to quantitative modeling of gene expression. I mentioned working in genetics previously, but I haven't delved deeply until now. I find it to be an incredibly interesting field.

Genetics is the study of the code from which all living things are built. This code is present in every organism's genome, which is composed of DNA or RNA, and copies of it are present in every cell. If an organism's genome has been sequenced, then its genome has been parsed into genes and other types of non-gene sequences. Here I focus only on the genes and their expression. Biologists' concept of *gene expression* involves the activity of known genes within a biological sample, and we measure gene expression using any of several tools that can measure the *copy number*, or concentration of specific RNA sequence fragments that are related directly to these genes. If an RNA fragment contains a sequence that's known to match a certain gene but not other genes, then that sequence can be used as an indicator of the expression of the gene. If that RNA sequence occurs very often (high copy number or concentration) in the biological sample, then the expression of the corresponding gene is said to be

high, and a sequence that occurs rarely indicates that its associated gene is expressed at a low level.

Two technologies, known as *microarrays* and *sequencing*, are common methods for measuring gene expression via the concentration or copy number of RNA sequences found in biological samples. Sequencing tends to be favored now, but at the time of my PhD research, I was analyzing data from microarrays. The data had been given to me by a collaborator at the University of Maryland School of Medicine, who had been studying the stem cells of *Mus musculus*—a common mouse—through various stages of development. In the earliest stages, stem cells are known to be of a general type that can subsequently develop into any of a number of *differentiated*, or specialized, cell types. The progression of cells through these stages of undifferentiated and then specific differentiated stem cell types isn't fully understood, but it had been hypothesized by my collaborators and others that a special class of RNA sequences called microRNA might be involved.

MicroRNAs (or miRs) are short RNA sequences (about 20 base pairs, vastly shorter than most genes) that are known to be present in virtually all organisms. To help determine whether miRs help regulate the development of stem cells and differentiation, my collaborators used microarrays to measure the expression of both genes and miRs throughout the early stages of development of stem cells.

The data set consisted of microarray data for both genes and miRs for each of the seven stem cell types. A single microarray measures several thousand genes or, alternatively, a few hundred miRs. And for each stem cell type, there were two to three replicates, meaning that each biological sample was analyzed using two to three gene-oriented microarrays and two to three miR-oriented microarrays. Replicates are helpful for analyzing variance between samples as well as identifying outliers. Given 7 stem cell types and 2 to 3 replicates each for genes and miRs, I had 33 microarrays in total.

Because miRs are thought mainly to inhibit expression of genes—they apparently bind to complementary sections of genetic RNA and block that RNA from being copied—the main question I asked of the data set was “Can I find any evidence of specific miRs inhibiting the expression of specific genes?” Is the expression of any certain gene routinely low whenever the expression of a specific miR is high? In addition, I wanted to know whether the expression and inhibiting activity of any miRs could be highly correlated with particular stages of stem cell development and differentiation.

Though no one had previously studied this specific topic—the effect of miRs in mouse stem cell development—a fair amount of work had been done on related topics. Of particular note was the class of statistical algorithms that attempted to characterize whether a particular miR would target (inhibit) a specific section of genetic RNA, based solely on the sequence information alone. If a miR's base sequence looks like this

ACATGTAACCTGTAGATAGAT

(again, I use *T* in place of *U* for convenience), then a perfectly complementary genetic RNA sequence would be

TGTACATTGGACATCTATCTA

because, within an RNA sequence, the nucleotide A is complementary to U, and C is complementary to G. Because these miRs are floating around in a cell's cytoplasm, as are genetic RNA sequences, there's no guarantee that even a perfect match will bind and inhibit gene expression. Under perfect conditions, such complementary sequences will bind, but nothing in biology is perfect. It's also likely that a miR and its perfect match will float past each other like two ships passing in the night, as they say. Also, it's a funny quirk of all RNA sequences that sometimes they bend a little too much and get stuck to themselves—for miRs the result is known as a *hairpin* because of the shape that's easy to imagine. In any case, it's not a foregone conclusion that perfectly complementary sequences will bind; nor is it true that imperfect matches won't bind. Many researchers have explored this and developed algorithms that assign miR-gene pairs matching scores based on complementarity and other features of the sequences. These are generally referred to as *target prediction* algorithms, and I made use of two such algorithms in my work: one called TargetScan (www.targetscan.org) and another called miRanda (www.microrna.org).

Both TargetScan and miRanda are widely viewed as the products of solid scientific research, and both of these algorithms and their predictions are freely available on the internet. For any miR-gene pair in my microarray data sets, I had at least two target prediction scores indicating whether the miR is likely to inhibit expression of the gene. The files I obtained from TargetScan look like this (with some columns removed for clarity):

Gene ID	miRNA	context+ score	percentile
71667	xtr-miR-9b	-0.259	89
71667	xtr-miR-9	-0.248	88
71667	xtr-miR-9a	-0.248	88

As you can see, for each gene and miR/miRNA, TargetScan has given a score representing the likelihood that the miR will target the genetic RNA. miRanda provides similar files. These scores are known to be imperfect, but they are informative, so I decided to include them as evidence but not certainty of inhibition of the gene's expression by the miR.

My main data set was still the set of microarrays I had from my collaborators' lab, and from these I would be able to analyze all expression values of genes and miRs and determine positive and negative correlations between them. Also, I could use the target predictions as further evidence in favor of certain miR-gene target pairs. In the framework of Bayesian statistics—discussed more in chapter 8—the target predictions can be considered a priori knowledge, and I could adjust that knowledge based on the new data I collected—the new microarray data I received from my collaborators. In

this way, neither the prediction nor the noisy data set was taken as truth, but both informed the final estimates of which miR-gene pairs are most likely true targeting interactions.

So far in this section, I've talked about combining gene expression data with microRNA data to search for targeting interactions between them and to analyze the effects of miRs on stem cell development. In addition, I included two target prediction data sets as further evidence that certain miRs target certain genes. As I completed analysis based on these data sets, I needed to be able to show that the miRs and genes that my models indicated as being related to stem cell development made sense in some way. There were two ways I might typically do this: ask my biologist collaborators to test some of my results in the lab to confirm that they were correct, or find more data sets online somewhere that were already validated and that supported my results in some way.

If I'd had no experience working with this sort of data, I might have Googled "validated microRNA targeting" or "stem cell development gene annotations," but because I knew from past projects that a large public set of annotations of genes known as Gene Ontology (GO) terms was available, as well as a database of validated miR-gene targeting interactions already reported in scientific publications, I didn't have to search much. GO term annotation can be accessed via a few web-based tools (geneontology.org) as well as a package for the R language, among others. I had previously used these annotations for analyzing groups of genes to see whether they have some things in common. In the case of this project, it would help to confirm my results if any group of genes found significant within my model with respect to stem cell development also had a significant number of GO annotations related to stem cells and stem cell development.

Also, I obviously preferred that any miR-gene target pairs that my model found significant would have already been validated in some other reliable way. This is where the data set of *validated* targeting interactions on www.microrna.org comes in. It's certainly a useful data set, but one important aspect of it is that, although some miR-gene target pairs have been confirmed, just because a pair *hasn't* been confirmed doesn't mean that it *isn't* a true target pair. If my model found a particular target pair significant, but it hadn't been validated yet, that didn't indicate at all that the model was wrong. On the other hand, if a validated target pair did not appear significant according to my model, then there was some reason for concern. Overall, in the validation step of my project, I hoped that all or most of the validated target pairs appeared significant according to the model, but I didn't necessarily need to see validations for my most significant results.

Lastly, my collaborators had some interest in which families of microRNAs (groups of miRs with partially matching sequences) contributed to which stages of stem cell development. It turned out that TargetScan provided a nicely formatted file matching miRs with their families. In addition to the gene expression microarrays, the microRNA expression microarrays, two target prediction algorithm results, a set of gene annotations, and some validated miR-gene target pairs, I added a miR family data set.

Needless to say, there were many moving parts in this project. Also, as happens quite often in academia, the resulting scientific papers couldn't include every piece of analysis. There's one paper describing the model and application to a public data set ("Inferring MicroRNA Regulation of mRNA with Partially Ordered Samples of Paired Expression Data and Exogenous Prediction Algorithms," PLoS ONE, 2012) and another paper describing the application to the mouse data set ("Correlated miR-mRNA Expression Signatures of Mouse Hematopoietic Stem and Progenitor Cell Subsets Predict 'Stemness' and 'Myeloid' Interaction Networks," PLoS ONE, 2014).

I won't describe all results here in detail, but I was quite satisfied with the project. After matching miRs and genes from their respective expression data sets with their predicted target pairs from TargetScan and miRanda, I analyzed them via a Bayesian model incorporating all of this data and validated it using GO annotations and known target pair validations, with some miR family analysis tacked on. The results weren't perfect; bioinformatics is notoriously complex, not to mention imperfect in its data quality. But most validated target pairs were significant, and some relevant GO annotations were overrepresented in significant groups of genes. In later chapters, I'll delve more deeply into statistical models, their significance, and drawing conclusions from results, but for now I'd like to leave this example as one in which various data sets have been combined in ways that make new analyses possible.

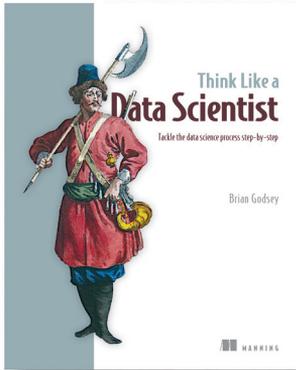
Exercises

Continuing with the Filthy Money Forecasting personal finance app scenario from the last chapter's exercises, try these exercises:

- 1 List three potential data sources that you expect would be good to examine for this project. For each, also list how you would expect to access the data (for example, database, API, and so on).
- 2 Consider the three project goals you listed in exercise 3 in the last chapter. What data would you need in order to achieve them?

Summary

- Data science treats *data* as something that might exist independently of any specific purpose. Much like nature itself, and all of the organisms and species that exist within it, discovered or undiscovered, the realm of data is worth exploration and study.
- Data can be found in many places with varying accessibility, complexity, and reliability.
- It's best to become familiar with some of the forms that data might take, as well as how to view and manipulate these forms.
- Before assuming that a data set contains what you want, it's best to evaluate data extent and quality.
- Finding and recognizing data sets that are useful for your project aren't always straightforward, but some preliminary investigation can help.



Data collected from customers, scientific measurements, IoT sensors, and so on is valuable only if you understand it. Data scientists revel in the interesting and rewarding challenge of observing, exploring, analyzing, and interpreting this data. Getting started with data science means more than mastering analytic tools and techniques, however; the real magic happens when you begin to think like a data scientist. This book will get you there.

Think Like a Data Scientist teaches you a step-by-step approach to solving real-world data-centric problems. By breaking down carefully crafted examples, you'll

learn to combine analytic, programming, and business perspectives into a repeatable process for extracting real knowledge from data. As you read, you'll discover (or remember) valuable statistical techniques and explore powerful data science software. More importantly, you'll put this knowledge together using a structured process for data science. When you've finished, you'll have a strong foundation for a lifetime of data science learning and practice.

What's inside

- The data science process, step-by-step
- How to anticipate problems
- Dealing with uncertainty
- Best practices in software and scientific thinking

Readers need beginner programming skills and knowledge of basic statistics.

Exploring Data

You can't be very effective with data if you don't know what you have. To get to know your data, it's best to run it through some diagnostics that summarize the data and expose any oddities or outliers. Often, we call these *descriptive statistics*. The goal is to be aware of when the data conforms to the general expectations that you and your colleagues have of it and, alternatively, when the data deviates and might cause problems down the line. The best descriptive statistics are the ones that check your most important assumptions and/or have the potential to expose significant bias or deviation.

This chapter, from *Practical Data Science with R* describes in detail how you might get to know your data using summary statistics, plots, and other descriptive statistics. The examples, complete with excellent figures and code (in R), show you exactly how to do them and—more importantly—why. It is the *why* that is most crucial to data science, making this chapter generalizable beyond the R language. Data scientists working in any language can benefit from the suggestions and insights of this chapter.

Exploring data

This chapter covers

- Using summary statistics to explore data
- Exploring data using visualization
- Finding problems and issues during data exploration

In the last two chapters, you learned how to set the scope and goal of a data science project, and how to load your data into R. In this chapter, we'll start to get our hands into the data.

Suppose your goal is to build a model to predict which of your customers don't have health insurance; perhaps you want to market inexpensive health insurance packages to them. You've collected a dataset of customers whose health insurance status you know. You've also identified some customer properties that you believe help predict the probability of insurance coverage: age, employment status, income, information about residence and vehicles, and so on. You've put all your data into a single data frame called *custdata* that you've input into R.¹ Now you're ready to start building the model to identify the customers you're interested in.

¹ We have a copy of this synthetic dataset available for download from <https://github.com/WinVector/zmPDSwR/tree/master/Custdata>, and once saved, you can load it into R with the command `custdata <- read.table('custdata.tsv', header=T, sep='\t')`.

It's tempting to dive right into the modeling step without looking very hard at the dataset first, especially when you have a lot of data. Resist the temptation. No dataset is perfect: you'll be missing information about some of your customers, and you'll have incorrect data about others. Some data fields will be dirty and inconsistent. If you don't take the time to examine the data before you start to model, you may find yourself redoing your work repeatedly as you discover bad data fields or variables that need to be transformed before modeling. In the worst case, you'll build a model that returns incorrect predictions—and you won't be sure why. By addressing data issues early, you can save yourself some unnecessary work, and a lot of headaches!

You'd also like to get a sense of who your customers are: Are they young, middle-aged, or seniors? How affluent are they? Where do they live? Knowing the answers to these questions can help you build a better model, because you'll have a more specific idea of what information predicts the probability of insurance coverage more accurately.

In this chapter, we'll demonstrate some ways to get to know your data, and discuss some of the potential issues that you're looking for as you explore. Data exploration uses a combination of *summary statistics*—means and medians, variances, and counts—and *visualization*, or graphs of the data. You can spot some problems just by using summary statistics; other problems are easier to find visually.

Organizing data for analysis

For most of this book, we'll assume that the data you're analyzing is in a single data frame. This is not how that data is usually stored. In a database, for example, data is usually stored in *normalized form* to reduce redundancy: information about a single customer is spread across many small tables. In log data, data about a single customer can be spread across many log entries, or sessions. These formats make it easy to add (or in the case of a database, modify) data, but are not optimal for analysis. You can often join all the data you need into a single table in the database using SQL, but in appendix A we'll discuss commands like `join` that you can use within R to further consolidate data.

3.1 Using summary statistics to spot problems

In R, you'll typically use the `summary` command to take your first look at the data.

Listing 3.1 The `summary()` command

```
> summary(custdata)
custid      sex
Min.   :    2068  F:440
1st Qu.: 345667  M:560
Median : 693403
Mean   : 698500
3rd Qu.:1044606
Max.   :1414286
```

```
is.employed      income
Mode :logical    Min.   : -8700
FALSE:73         1st Qu.: 14600
TRUE :599        Median : 35000
NA's :328        Mean   : 53505
                 3rd Qu.: 67000
                 Max.   : 615000
```

← The variable `is.employed` is missing for about a third of the data. The variable `income` has negative values, which are potentially invalid.

```
marital.stat
Divorced/Separated:155
Married           :516
Never Married     :233
Widowed          : 96
```

```
health.ins
Mode :logical
FALSE:159
TRUE :841
NA's :0
```

← About 84% of the customers have health insurance.

```
housing.type
Homeowner free and clear :157
Homeowner with mortgage/loan:412
Occupied with no rent    : 11
Rented                   :364
NA's                     : 56
```

← The variables `housing.type`, `recent.move`, and `num.vehicles` are each missing 56 values.

```
recent.move      num.vehicles
Mode :logical    Min.   :0.000
FALSE:820        1st Qu.:1.000
TRUE :124        Median :2.000
NA's :56         Mean   :1.916
                 3rd Qu.:2.000
                 Max.   :6.000
                 NA's   :56
```

← The average value of the variable `age` seems plausible, but the minimum and maximum values seem unlikely. The variable `state.of.res` is a categorical variable; `summary()` reports how many customers are in each state (for the first few states).

```
age              state.of.res
Min.   : 0.0    California :100
1st Qu.: 38.0   New York   : 71
Median : 50.0   Pennsylvania: 70
Mean   : 51.7   Texas     : 56
3rd Qu.: 64.0   Michigan  : 52
Max.   :146.7   Ohio      : 51
                 (Other)  :600
```

The `summary` command on a data frame reports a variety of summary statistics on the numerical columns of the data frame, and count statistics on any categorical columns (if the categorical columns have already been read in as factors²). You can also ask for summary statistics on specific numerical columns by using the commands `mean`, `variance`, `median`, `min`, `max`, and `quantile` (which will return the quartiles of the data by default).

² Categorical variables are of class `factor` in R. They can be represented as strings (class `character`), and some analytical functions will automatically convert string variables to factor variables. To get a summary of a variable, it needs to be a factor.

As you see from listing 3.1, the summary of the data helps you quickly spot potential problems, like missing data or unlikely values. You also get a rough idea of how categorical data is distributed. Let's go into more detail about the typical problems that you can spot using the summary.

3.1.1 Typical problems revealed by data summaries

At this stage, you're looking for several common issues: missing values, invalid values and outliers, and data ranges that are too wide or too narrow. Let's address each of these issues in detail.

MISSING VALUES

A few missing values may not really be a problem, but if a particular data field is largely unpopulated, it shouldn't be used as an input without some repair (as we'll discuss in chapter 4, section 4.1.1). In R, for example, many modeling algorithms will, by default, quietly drop rows with missing values. As you see in listing 3.2, all the missing values in the `is.employed` variable could cause R to quietly ignore nearly a third of the data.

Listing 3.2 Will the variable `is.employed` be useful for modeling?

```
is.employed
  Mode :logical
FALSE:73
 TRUE :599
NA's :328
```



The variable `is.employed` is missing for about a third of the data. Why? Is employment status unknown? Did the company start collecting employment data only recently? Does NA mean “not in the active workforce” (for example, students or stay-at-home parents)?

```
                housing.type
Homeowner free and clear :157
Homeowner with mortgage/loan:412
Occupied with no rent    : 11
Rented                   :364
NA's                     : 56
```



The variables `housing.type`, `recent.move`, and `num.vehicles` are only missing a few values. It's probably safe to just drop the rows that are missing values—especially if the missing values are all the same 56 rows.

```
recent.move    num.vehicles
Mode :logical  Min.   :0.000
FALSE:820     1st Qu.:1.000
 TRUE :124     Median :2.000
NA's :56       Mean   :1.916
                3rd Qu.:2.000
                Max.   :6.000
                NA's   :56
```

If a particular data field is largely unpopulated, it's worth trying to determine why; sometimes the fact that a value is missing is informative in and of itself. For example, why is the `is.employed` variable missing so many values? There are many possible reasons, as we noted in listing 3.2.

Whatever the reason for missing data, you must decide on the most appropriate action. Do you include a variable with missing values in your model, or not? If you

decide to include it, do you drop all the rows where this field is missing, or do you convert the missing values to 0 or to an additional category? We'll discuss ways to treat missing data in chapter 4. In this example, you might decide to drop the data rows where you're missing data about housing or vehicles, since there aren't many of them. You probably don't want to throw out the data where you're missing employment information, but instead treat the NAs as a third employment category. You will likely encounter missing values when model scoring, so you should deal with them during model training.

INVALID VALUES AND OUTLIERS

Even when a column or variable isn't missing any values, you still want to check that the values that you do have make sense. Do you have any invalid values or outliers? Examples of invalid values include negative values in what should be a non-negative numeric data field (like age or income), or text where you expect numbers. Outliers are data points that fall well out of the range of where you expect the data to be. Can you spot the outliers and invalid values in listing 3.3?

Listing 3.3 Examples of invalid values and outliers

```
> summary(custdata$income)
  Min. 1st Qu.  Median    Mean 3rd Qu.
-8700  14600   35000   53500   67000
  Max.
615000
```

Negative values for income could indicate bad data. They might also have a special meaning, like "amount of debt."

Either way, you should check how prevalent the issue is, and decide what to do: Do you drop the data with negative income? Do you convert negative values to zero?

```
> summary(custdata$age)
  Min. 1st Qu.  Median    Mean 3rd Qu.
  0.0   38.0   50.0   51.7   64.0
  Max.
146.7
```

Customers of age zero, or customers of an age greater than about 110 are outliers. They fall out of the range of expected customer values.

Outliers could be data input errors. They could be special sentinel values: zero might mean "age unknown" or "refuse to state." And some of your customers might be especially long-lived.

Often, invalid values are simply bad data input. Negative numbers in a field like age, however, could be a *sentinel value* to designate "unknown." Outliers might also be data errors or sentinel values. Or they might be valid but unusual data points—people do occasionally live past 100.

As with missing values, you must decide the most appropriate action: drop the data field, drop the data points where this field is bad, or convert the bad data to a useful value. Even if you feel certain outliers are valid data, you might still want to omit them from model construction (and also collar allowed prediction range), since the usual achievable goal of modeling is to predict the typical case correctly.

DATA RANGE

You also want to pay attention to how much the values in the data vary. If you believe that age or income helps to predict the probability of health insurance coverage, then

you should make sure there is enough variation in the age and income of your customers for you to see the relationships. Let's look at income again, in listing 3.4. Is the data range wide? Is it narrow?

Listing 3.4 Looking at the data range of a variable

```
> summary(custdata$income)
  Min. 1st Qu.  Median    Mean 3rd Qu.
-8700  14600   35000   53500  67000
  Max.
 615000
```

Income ranges from zero to over half a million dollars; a very wide range.

Even ignoring negative income, the `income` variable in listing 3.4 ranges from zero to over half a million dollars. That's pretty wide (though typical for income). Data that ranges over several orders of magnitude like this can be a problem for some modeling methods. We'll talk about mitigating data range issues when we talk about logarithmic transformations in chapter 4.

Data can be too narrow, too. Suppose all your customers are between the ages of 50 and 55. It's a good bet that age range wouldn't be a very good predictor of the probability of health insurance coverage for that population, since it doesn't vary much at all.

How narrow is "too narrow" a data range?

Of course, the term *narrow* is relative. If we were predicting the ability to read for children between the ages of 5 and 10, then age probably is a useful variable as-is. For data including adult ages, you may want to transform or bin ages in some way, as you don't expect a significant change in reading ability between ages 40 and 50. You should rely on information about the problem domain to judge if the data range is narrow, but a rough rule of thumb is the ratio of the standard deviation to the mean. If that ratio is very small, then the data isn't varying much.

We'll revisit data range in section 3.2, when we talk about examining data graphically.

One factor that determines apparent data range is the unit of measurement. To take a nontechnical example, we measure the ages of babies and toddlers in weeks or in months, because developmental changes happen at that time scale for very young children. Suppose we measured babies' ages in years. It might appear numerically that there isn't much difference between a one-year-old and a two-year-old. In reality, there's a dramatic difference, as any parent can tell you! Units can present potential issues in a dataset for another reason, as well.

UNITS

Does the income data in listing 3.5 represent hourly wages, or yearly wages in units of \$1000? As a matter of fact, it's the latter, but what if you thought it was the former? You might not notice the error during the modeling stage, but down the line someone will start inputting hourly wage data into the model and get back bad predictions in return.

Listing 3.5 Checking units can prevent inaccurate results later

```
> summary(Income)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-8.7    14.6    35.0    53.5    67.0    615.0
```

The variable `Income` is defined as `custdata$Income/1000`. But suppose you didn't know that. Looking only at the summary, the values could plausibly be interpreted to mean either "hourly wage" or "yearly

Are time intervals measured in days, hours, minutes, or milliseconds? Are speeds in kilometers per second, miles per hour, or knots? Are monetary amounts in dollars, thousands of dollars, or 1/100 of a penny (a customary practice in finance, where calculations are often done in fixed-point arithmetic)? This is actually something that you'll catch by checking data definitions in data dictionaries or documentation, rather than in the summary statistics; the difference between hourly wage data and annual salary in units of \$1000 may not look that obvious at a casual glance. But it's still something to keep in mind while looking over the value ranges of your variables, because often you can spot when measurements are in unexpected units. Automobile speeds in knots look a lot different than they do in miles per hour.

3.2 Spotting problems using graphics and visualization

As you've seen, you can spot plenty of problems just by looking over the data summaries. For other properties of the data, pictures are better than text.

We cannot expect a small number of numerical values [summary statistics] to consistently convey the wealth of information that exists in data. Numerical reduction methods do not retain the information in the data.

—William Cleveland
The Elements of Graphing Data

Figure 3.1 shows a plot of how customer ages are distributed. We'll talk about what the y-axis of the graph means later; for right now, just know that the height of the graph corresponds to how many customers in the population are of that age. As you can see, information like the peak age of the distribution, the existence of subpopulations, and the presence of outliers is easier to absorb visually than it is to determine textually.

The use of graphics to examine data is called *visualization*. We try to follow William Cleveland's principles for scientific visualization. Details of specific plots aside, the key points of Cleveland's philosophy are these:

- A graphic should display as much information as it can, with the lowest possible cognitive strain to the viewer.
- Strive for clarity. Make the data stand out. Specific tips for increasing clarity include
 - Avoid too many superimposed elements, such as too many curves in the same graphing space.

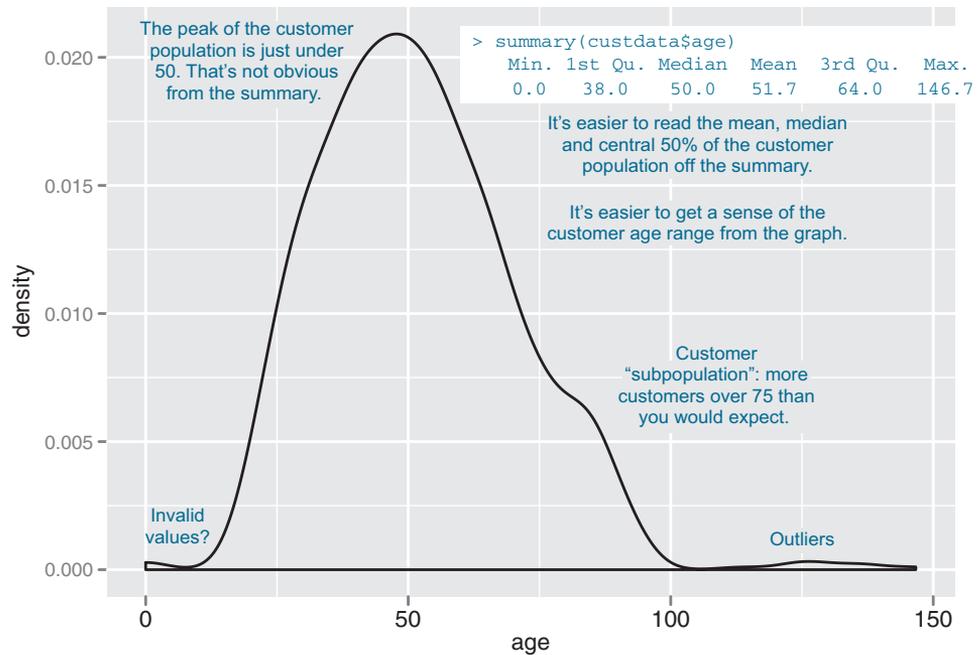


Figure 3.1 Some information is easier to read from a graph, and some from a summary.

- Find the right aspect ratio and scaling to properly bring out the details of the data.
- Avoid having the data all skewed to one side or the other of your graph.
- Visualization is an iterative process. Its purpose is to answer questions about the data.

During the visualization stage, you graph the data, learn what you can, and then regraph the data to answer the questions that arise from your previous graphic. Different graphics are best suited for answering different questions. We'll look at some of them in this section.

In this book, we use `ggplot2` to demonstrate the visualizations and graphics; of course, other R visualization packages can produce similar graphics.

A note on `ggplot2`

The theme of this section is how to use visualization to explore your data, not how to use `ggplot2`. We chose `ggplot2` because it excels at combining multiple graphical elements together, but its syntax can take some getting used to. The key points to understand when looking at our code snippets are these:

- Graphs in `ggplot2` can only be defined on data frames. The variables in a graph—the x variable, the y variable, the variables that define the color or the

size of the points—are called *aesthetics*, and are declared by using the `aes` function.

- The `ggplot()` function declares the graph object. The arguments to `ggplot()` can include the data frame of interest and the aesthetics. The `ggplot()` function doesn't of itself produce a visualization; visualizations are produced by *layers*.
- Layers produce the plots and plot transformations and are added to a given graph object using the `+` operator. Each layer can also take a data frame and aesthetics as arguments, in addition to plot-specific parameters. Examples of layers are `geom_point` (for a scatter plot) or `geom_line` (for a line plot).

This syntax will become clearer in the examples that follow. For more information, we recommend Hadley Wickham's reference site <http://ggplot2.org>, which has pointers to online documentation, as well as to Dr. Wickham's *ggplot2: Elegant Graphics for Data Analysis (Use R!)* (Springer, 2009).

In the next two sections, we'll show how to use pictures and graphs to identify data characteristics and issues. In section 3.2.2, we'll look at visualizations for two variables. But let's start by looking at visualizations for single variables.

3.2.1 Visually checking distributions for a single variable

The visualizations in this section help you answer questions like these:

- What is the peak value of the distribution?
- How many peaks are there in the distribution (unimodality versus bimodality)?
- How normal (or lognormal) is the data? We'll discuss normal and lognormal distributions in appendix B.
- How much does the data vary? Is it concentrated in a certain interval or in a certain category?

One of the things that's easier to grasp visually is the shape of the data distribution. Except for the blip to the right, the graph in figure 3.1 (which we've reproduced as the gray curve in figure 3.2) is almost shaped like the normal distribution (see appendix B). As that appendix explains, many summary statistics assume that the data is approximately normal in distribution (at least for continuous variables), so you want to verify whether this is the case.

You can also see that the gray curve in figure 3.2 has only one peak, or that it's *unimodal*. This is another property that you want to check in your data.

Why? Because (roughly speaking), a unimodal distribution corresponds to one population of subjects. For the gray curve in figure 3.2, the mean customer age is about 52, and 50% of the customers are between 38 and 64 (the first and third quartiles). So you can say that a "typical" customer is middle-aged and probably possesses many of the demographic qualities of a middle-aged person—though of course you have to verify that with your actual customer information.

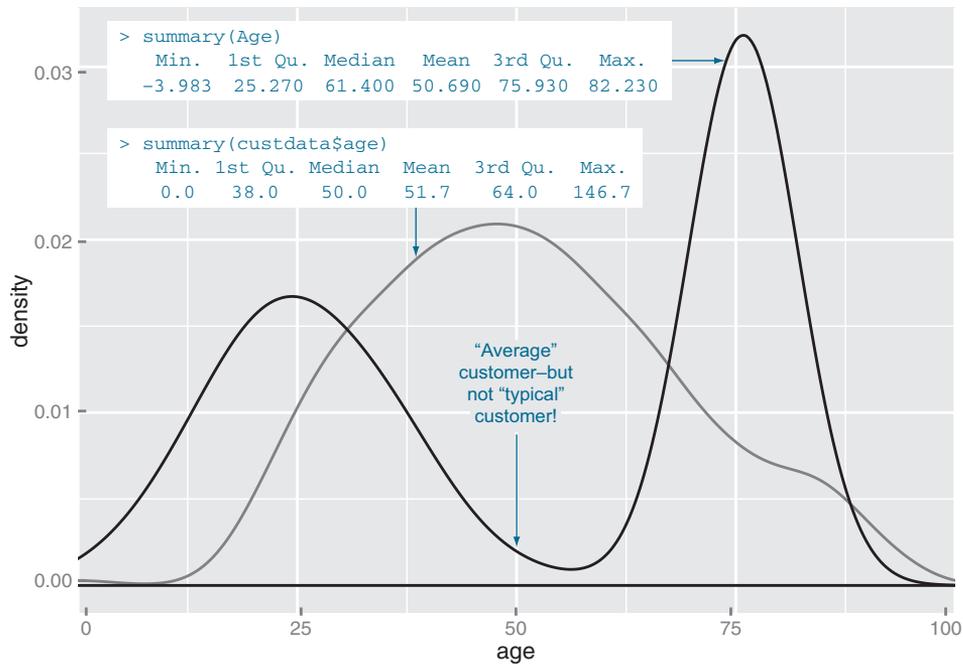


Figure 3.2 A unimodal distribution (gray) can usually be modeled as coming from a single population of users. With a bimodal distribution (black), your data often comes from two populations of users.

The black curve in figure 3.2 shows what can happen when you have two peaks, or a *bimodal distribution*. (A distribution with more than two peaks is *multimodal*.) This set of customers has about the same mean age as the customers represented by the gray curve—but a 50-year-old is hardly a “typical” customer! This (admittedly exaggerated) example corresponds to two populations of customers: a fairly young population mostly in their 20s and 30s, and an older population mostly in their 70s. These two populations probably have very different behavior patterns, and if you want to model whether a customer probably has health insurance or not, it wouldn’t be a bad idea to model the two populations separately—especially if you’re using linear or logistic regression.

The histogram and the density plot are two visualizations that help you quickly examine the distribution of a numerical variable. Figures 3.1 and 3.2 are density plots. Whether you use histograms or density plots is largely a matter of taste. We tend to prefer density plots, but histograms are easier to explain to less quantitatively-minded audiences.

HISTOGRAMS

A basic histogram bins a variable into fixed-width buckets and returns the number of data points that falls into each bucket. For example, you could group your customers by age range, in intervals of five years: 20–25, 25–30, 30–35, and so on. Customers at a

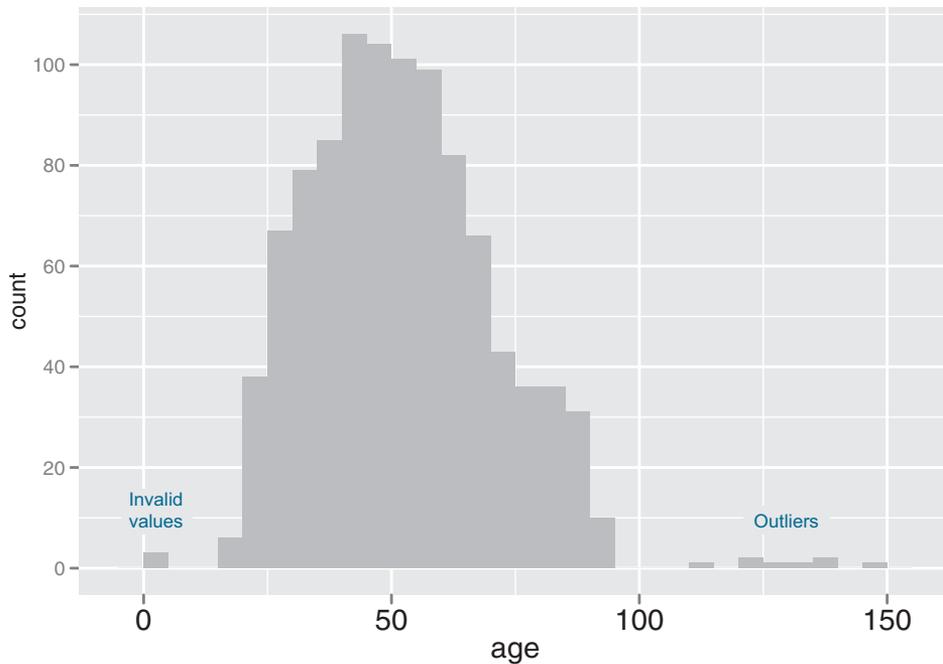


Figure 3.3 A histogram tells you where your data is concentrated. It also visually highlights outliers and anomalies.

boundary age would go into the higher bucket: 25-year-olds go into the 25–30 bucket. For each bucket, you then count how many customers are in that bucket. The resulting histogram is shown in figure 3.3.

You create the histogram in figure 3.3 in `ggplot2` with the `geom_histogram` layer.

Listing 3.6 Plotting a histogram

```
library(ggplot2)
ggplot(custdata) +
  geom_histogram(aes(x=age),
    binwidth=5, fill="gray")
```

← Load the `ggplot2` library, if you haven't already done so.

← The `binwidth` parameter tells the `geom_histogram` call how to make bins of five-year intervals (default is `datarange/30`). The `fill` parameter specifies the color of the histogram bars (default: black).

The primary disadvantage of histograms is that you must decide ahead of time how wide the buckets are. If the buckets are too wide, you can lose information about the shape of the distribution. If the buckets are too narrow, the histogram can look too noisy to read easily. An alternative visualization is the density plot.

DENSITY PLOTS

You can think of a *density plot* as a “continuous histogram” of a variable, except the area under the density plot is equal to 1. A point on a density plot corresponds to the

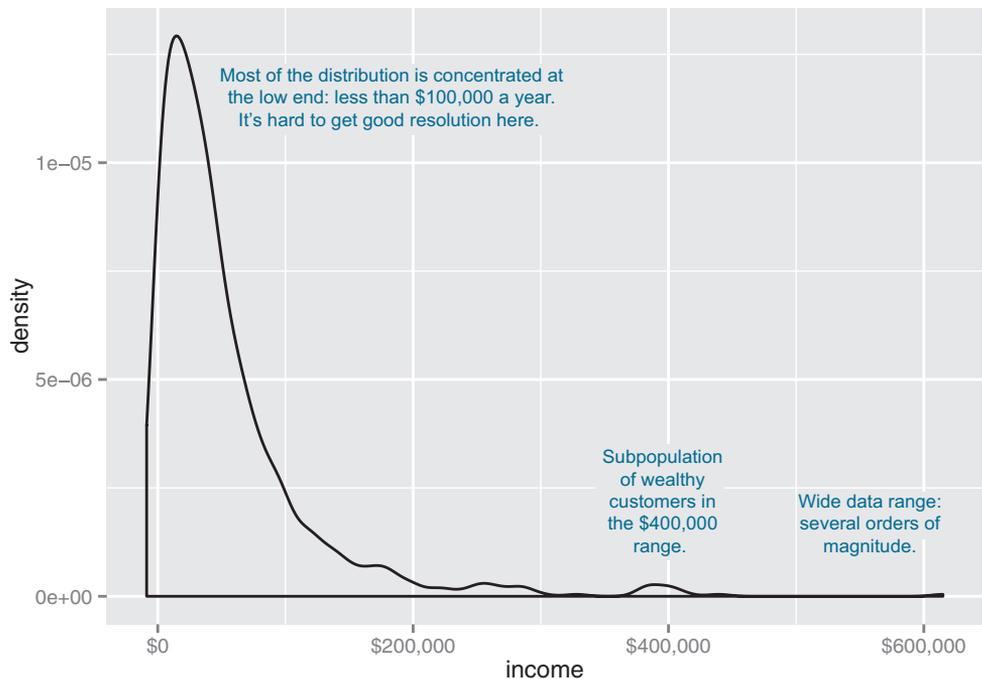


Figure 3.4 Density plots show where data is concentrated. This plot also highlights a population of higher-income customers.

fraction of data (or the percentage of data, divided by 100) that takes on a particular value. This fraction is usually very small. When you look at a density plot, you're more interested in the overall shape of the curve than in the actual values on the y-axis. You've seen the density plot of age; figure 3.4 shows the density plot of income. You produce figure 3.4 with the `geom_density` layer, as shown in the following listing.

Listing 3.7 Producing a density plot

```
library(scales)
ggplot(custdata) + geom_density(aes(x=income)) +
  scale_x_continuous(labels=dollar)
```

← The `scales` package brings in the dollar scale notation.

← Set the x-axis labels to dollars.

When the data range is very wide and the mass of the distribution is heavily concentrated to one side, like the distribution in figure 3.4, it's difficult to see the details of its shape. For instance, it's hard to tell the exact value where the income distribution has its peak. If the data is non-negative, then one way to bring out more detail is to plot the distribution on a logarithmic scale, as shown in figure 3.5. This is equivalent to plotting the density plot of `log10(income)`.

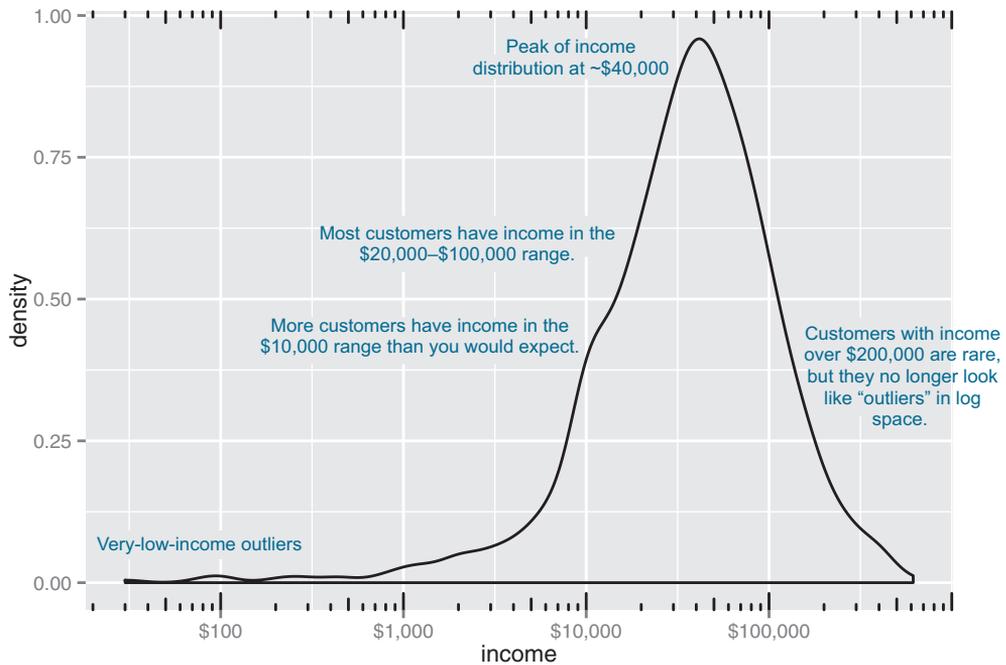


Figure 3.5 The density plot of income on a log10 scale highlights details of the income distribution that are harder to see in a regular density plot.

In `ggplot2`, you can plot figure 3.5 with the `geom_density` and `scale_x_log10` layers, such as in the next listing.

Listing 3.8 Creating a log-scaled density plot

```
ggplot(custdata) + geom_density(aes(x=income)) +
  scale_x_log10(breaks=c(100,1000,10000,100000), labels=dollar) +
  annotation_logticks(sides="bt")
```

Set the x-axis to be in log10 scale, with manually set tick points and labels as dollars.

Add log-scaled tick marks to the top and bottom of the graph.

When you issued the preceding command, you also got back a warning message:

Warning messages:

```
1: In scale$trans$trans(x) : NaNs produced
2: Removed 79 rows containing non-finite values (stat_density).
```

This tells you that `ggplot2` ignored the zero- and negative-valued rows (since $\log(0) = \text{Infinity}$), and that there were 79 such rows. Keep that in mind when evaluating the graph.

In log space, income is distributed as something that looks like a “normalish” distribution, as will be discussed in appendix B. It’s not exactly a normal distribution (in fact, it appears to be at least two normal distributions mixed together).

When should you use a logarithmic scale?

You should use a logarithmic scale when percent change, or change in orders of magnitude, is more important than changes in absolute units. You should also use a log scale to better visualize data that is heavily skewed.

For example, in income data, a difference in income of five thousand dollars means something very different in a population where the incomes tend to fall in the tens of thousands of dollars than it does in populations where income falls in the hundreds of thousands or millions of dollars. In other words, what constitutes a “significant difference” depends on the order of magnitude of the incomes you’re looking at. Similarly, in a population like that in figure 3.5, a few people with very high income will cause the majority of the data to be compressed into a relatively small area of the graph. For both those reasons, plotting the income distribution on a logarithmic scale is a good idea.

BAR CHARTS

A *bar chart* is a histogram for discrete data: it records the frequency of every value of a categorical variable. Figure 3.6 shows the distribution of marital status in your customer dataset. If you believe that marital status helps predict the probability of health insurance coverage, then you want to check that you have enough customers with different marital statuses to help you discover the relationship between being married (or not) and having health insurance.

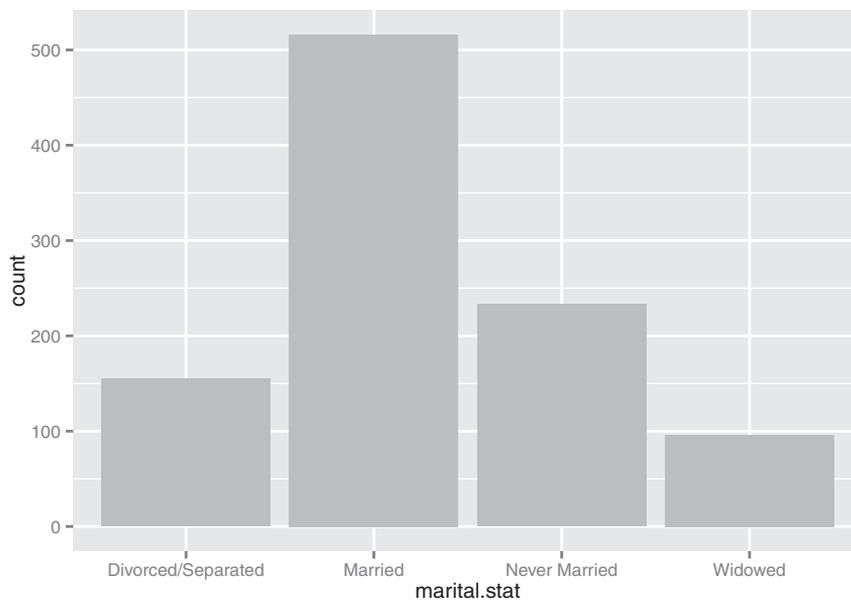


Figure 3.6 Bar charts show the distribution of categorical variables.

The `ggplot2` command to produce figure 3.6 uses `geom_bar`:

```
ggplot(custdata) + geom_bar(aes(x=marital.stat), fill="gray")
```

This graph doesn't really show any more information than `summary(custdata$marital.stat)` would show, although some people find the graph easier to absorb than the text. Bar charts are most useful when the number of possible values is fairly large, like state of residence. In this situation, we often find that a horizontal graph is more legible than a vertical graph.

The `ggplot2` command to produce figure 3.7 is shown in the next listing.

Listing 3.9 Producing a horizontal bar chart

Flip the
x and y
axes:
state.of.res
is now on
the y-axis.

```
ggplot(custdata) +  
  geom_bar(aes(x=state.of.res), fill="gray") +  
  coord_flip() +  
  theme(axis.text.y=element_text(size=rel(0.8)))
```

Plot bar chart as before:
state.of.res is on x axis,
count is on y-axis.

Reduce the size of the y-axis
tick labels to 80% of default
size for legibility.

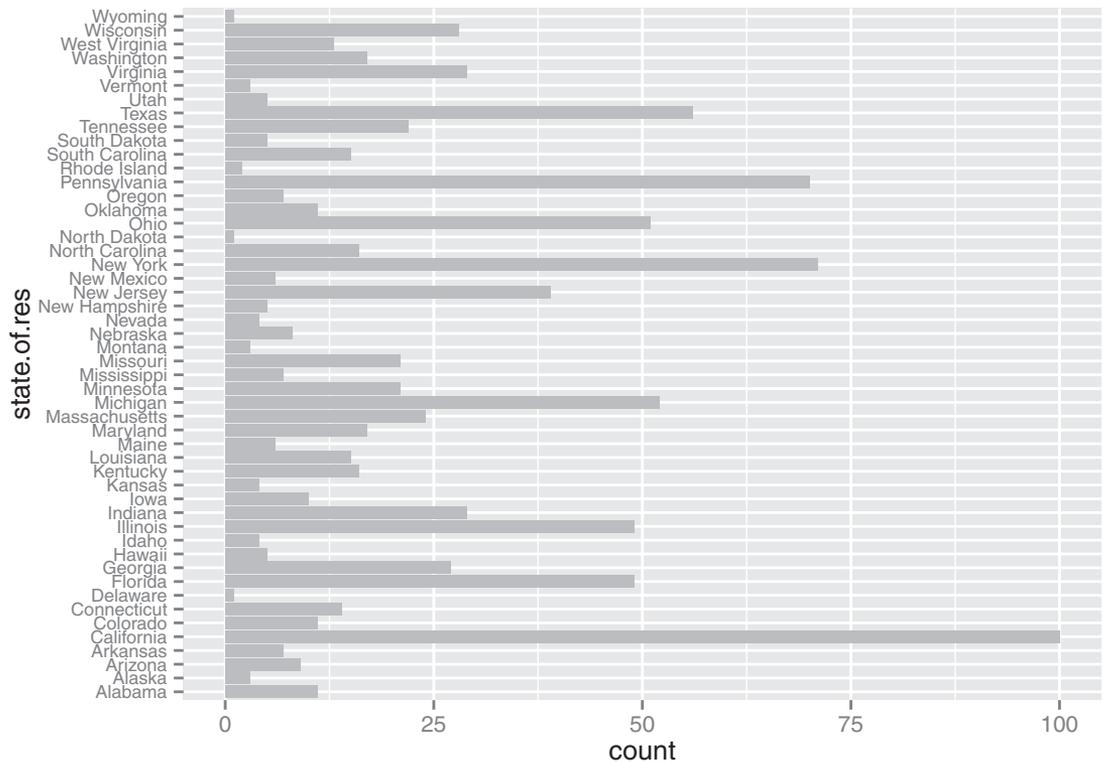


Figure 3.7 A horizontal bar chart can be easier to read when there are several categories with long names.

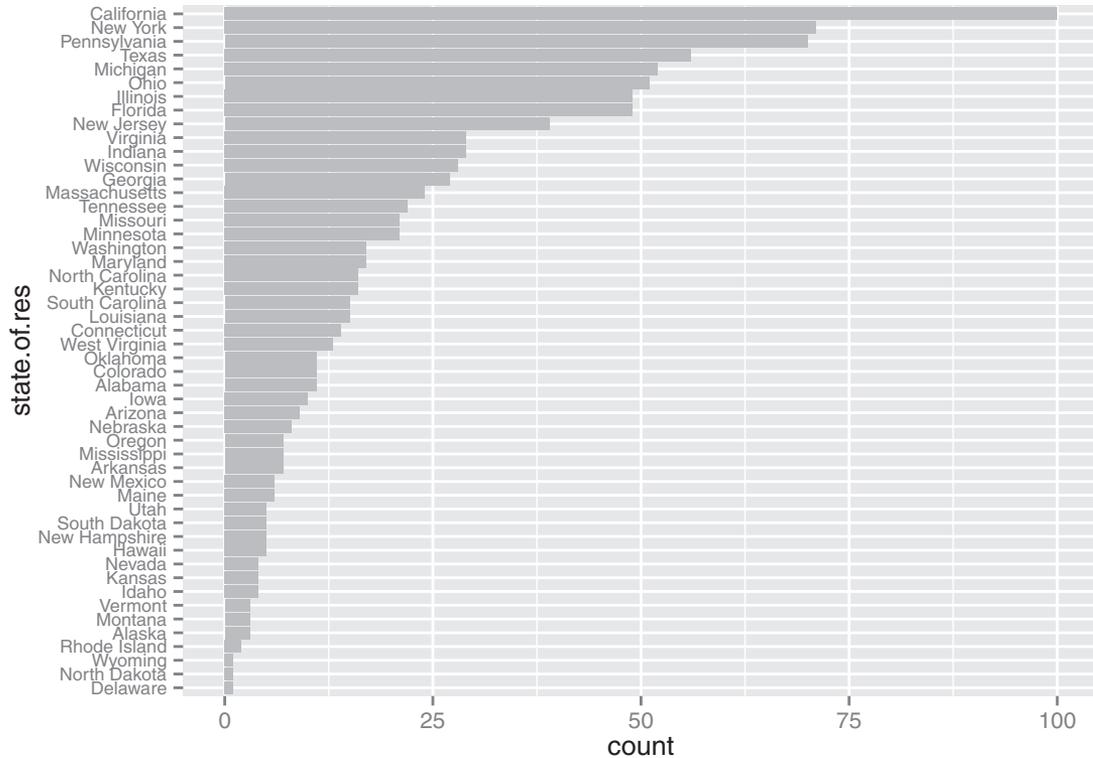


Figure 3.8 Sorting the bar chart by count makes it even easier to read.

Cleveland³ recommends that the data in a bar chart (or in a *dot plot*, Cleveland's preferred visualization in this instance) be sorted, to more efficiently extract insight from the data. This is shown in figure 3.8.

This visualization requires a bit more manipulation, at least in `ggplot2`, because by default, `ggplot2` will plot the categories of a factor variable in alphabetical order. To change this, we have to manually specify the order of the categories—in the factor variable, not in `ggplot2`.

Listing 3.10 Producing a bar chart with sorted categories

```

> statesums <- table(custdata$state.of.res)
> statef <- as.data.frame(statesums)
> colnames(statef) <- c("state.of.res", "count")
> summary(statef)

```

Rename the columns for readability.

Convert the table object to a data frame using `as.data.frame()`. The default column names are `Var1` and `Freq`.

The `table()` command aggregates the data by state of residence—exactly the information the bar chart plots.

Notice that the default ordering for the `state.of.res` variable is alphabetical.

³ See William S. Cleveland, *The Elements of Graphing Data*, Hobart Press, 1994.

```

state.of.res      count
Alabama   : 1      Min.    :  1.00
Alaska    : 1      1st Qu.:  5.00
Arizona   : 1      Median : 12.00
Arkansas  : 1      Mean    : 20.00
California: 1      3rd Qu.: 26.25
Colorado  : 1      Max.    :100.00
(Other)   :44
> statef <- transform(statef,
  state.of.res=reorder(state.of.res, count))
> summary(statef)
  state.of.res      count
Delaware   : 1      Min.    :  1.00
North Dakota: 1      1st Qu.:  5.00
Wyoming    : 1      Median : 12.00
Rhode Island: 1      Mean    : 20.00
Alaska     : 1      3rd Qu.: 26.25
Montana    : 1      Max.    :100.00
(Other)    :44
> ggplot(statef)+ geom_bar(aes(x=state.of.res,y=count),
  stat="identity",
  fill="gray") +
  coord_flip() +
  theme(axis.text.y=element_text(size=rel(0.8)))

```

Use the `reorder()` function to set the `state.of.res` variable to be count ordered. Use the `transform()` function to apply the transformation to the `state.of.res` data frame.

The `state.of.res` variable is now count ordered.

Since the data is being passed to `geom_bar` pre-aggregated, specify both the `x` and `y` variables, and use `stat="identity"` to plot the data exactly as given.

Flip the axes and reduce the size of the label text as before.

Before we move on to visualizations for two variables, in table 3.1 we'll summarize the visualizations that we've discussed in this section.

Table 3.1 Visualizations for one variable

Graph type	Uses
Histogram or density plot	Examines data range Checks number of modes Checks if distribution is normal/lognormal Checks for anomalies and outliers
Bar chart	Compares relative or absolute frequencies of the values of a categorical variable

3.2.2 Visually checking relationships between two variables

In addition to examining variables in isolation, you'll often want to look at the relationship between two variables. For example, you might want to answer questions like these:

- Is there a relationship between the two inputs *age* and *income* in my data?
- What kind of relationship, and how strong?
- Is there a relationship between the input *marital status* and the output *health insurance*? How strong?

You'll precisely quantify these relationships during the modeling phase, but exploring them now gives you a feel for the data and helps you determine which variables are the best candidates to include in a model.

First, let's consider the relationship between two continuous variables. The most obvious way (though not always the best) is the line plot.

LINE PLOTS

Line plots work best when the relationship between two variables is relatively clean: each *x* value has a unique (or nearly unique) *y* value, as in figure 3.9. You plot figure 3.9 with `geom_line`.

Listing 3.11 Producing a line plot

Plot
the
line
plot.

```
x <- runif(100)
y <- x^2 + 0.2*x
ggplot(data.frame(x=x,y=y), aes(x=x,y=y)) + geom_line()
```

First, generate the data for this example. The *x* variable is uniformly randomly distributed between 0 and 1.

The *y* variable is a quadratic function of *x*.

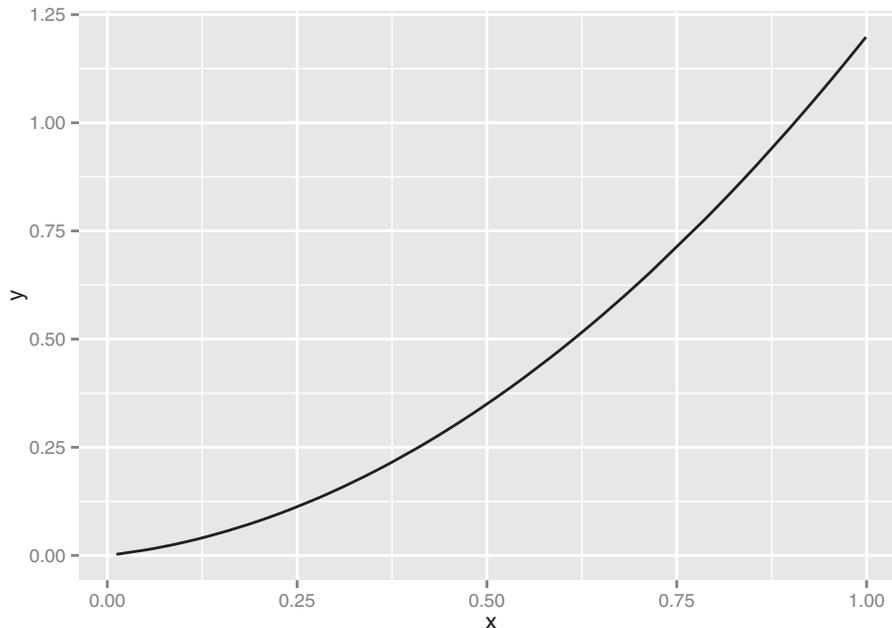


Figure 3.9 Example of a line plot

When the data is not so cleanly related, line plots aren't as useful; you'll want to use the scatter plot instead, as you'll see in the next section.

SCATTER PLOTS AND SMOOTHING CURVES

You'd expect there to be a relationship between age and health insurance, and also a relationship between income and health insurance. But what is the relationship between age and income? If they track each other perfectly, then you might not want to use both variables in a model for health insurance. The appropriate summary statistic is the correlation, which we compute on a safe subset of our data.

Listing 3.12 Examining the correlation between age and income

```

custdata2 <- subset(custdata,
  (custdata$age > 0 & custdata$age < 100
  & custdata$income > 0))
cor(custdata2$age, custdata2$income)
[1] -0.02240845

```

Only consider a subset of data with reasonable age and income values.

Get correlation of age and income.

Resulting correlation.

The negative correlation is surprising, since you'd expect that income should increase as people get older. A visualization gives you more insight into what's going on than a single number can. Let's try a scatter plot first; you plot figure 3.10 with `geom_point`:

```

ggplot(custdata2, aes(x=age, y=income)) +
  geom_point() + ylim(0, 200000)

```

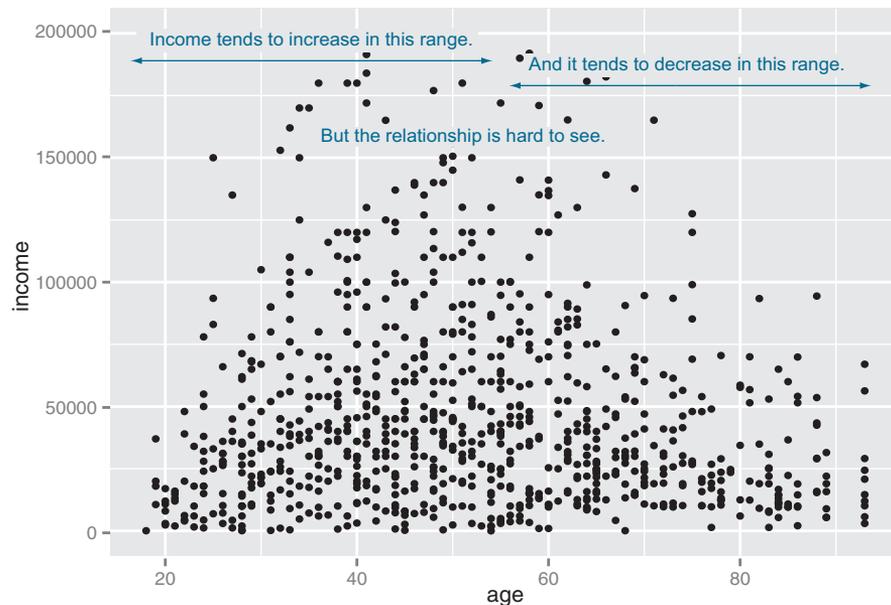


Figure 3.10 A scatter plot of income versus age

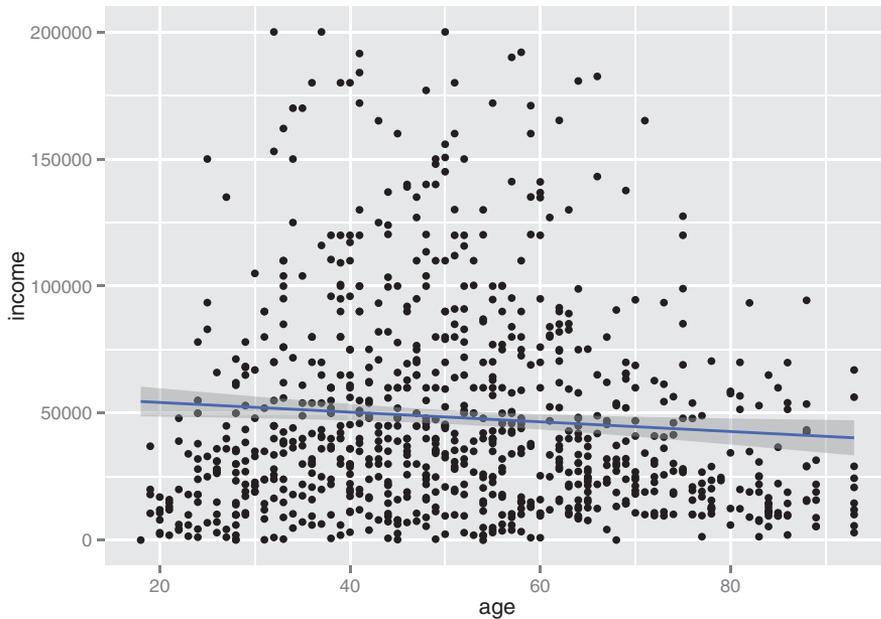


Figure 3.11 A scatter plot of income versus age, with a linear fit

The relationship between age and income isn't easy to see. You can try to make the relationship clearer by also plotting a linear fit through the data, as shown in figure 3.11.

You plot figure 3.11 using the `stat_smooth` layer:⁴

```
ggplot(custdata2, aes(x=age, y=income)) + geom_point() +
  stat_smooth(method="lm") +
  ylim(0, 200000)
```

In this case, the linear fit doesn't really capture the shape of the data. You can better capture the shape by instead plotting a smoothing curve through the data, as shown in figure 3.12.

In R, smoothing curves are fit using the `loess` (or `lowess`) functions, which calculate smoothed local linear fits of the data. In `ggplot2`, you can plot a smoothing curve to the data by using `geom_smooth`:

```
ggplot(custdata2, aes(x=age, y=income)) +
  geom_point() + geom_smooth() +
  ylim(0, 200000)
```

A scatter plot with a smoothing curve also makes a good visualization of the relationship between a continuous variable and a Boolean. Suppose you're considering using age as an input to your health insurance model. You might want to plot health insurance

⁴ The *stat* layers in `ggplot2` are the layers that perform transformations on the data. They're usually called under the covers by the *geom* layers. Sometimes you have to call them directly, to access parameters that aren't accessible from the *geom* layers. In this case, the default smoothing curve used `geom_smooth`, which is a loess curve, as you'll see shortly. To plot a linear fit we must call `stat_smooth` directly.

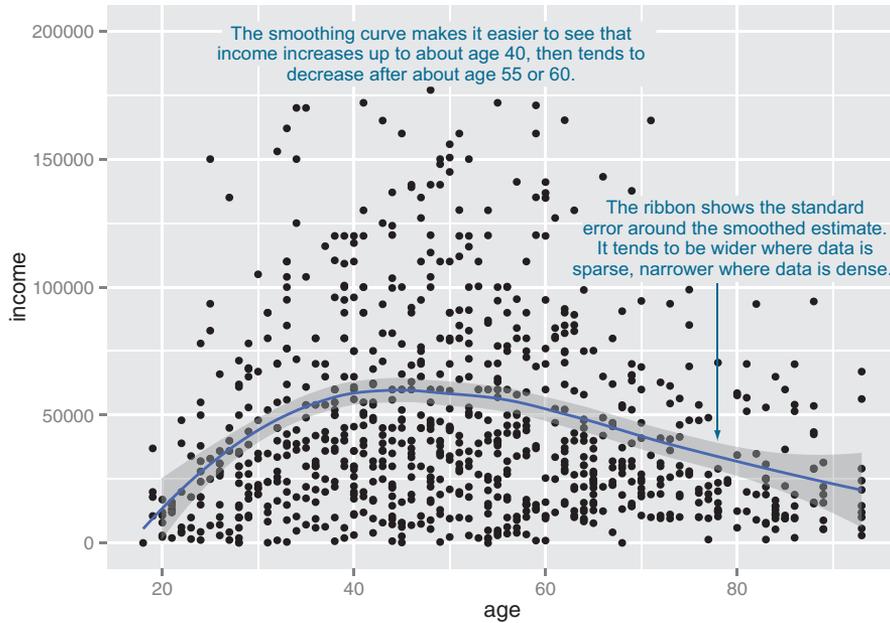


Figure 3.12 A scatter plot of income versus age, with a smoothing curve

coverage as a function of age, as shown in figure 3.13. This will show you that the probability of having health insurance increases as customer age increases.

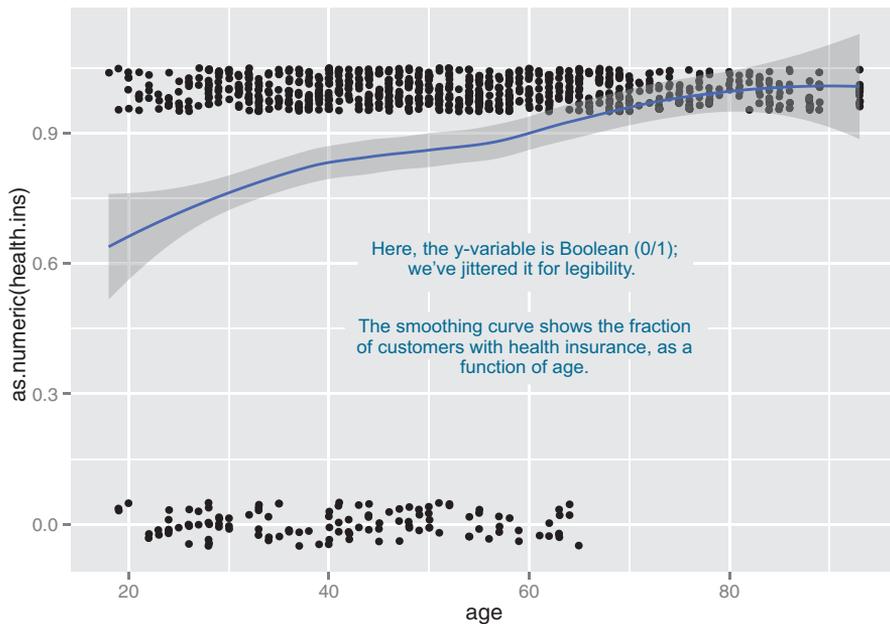


Figure 3.13 Distribution of customers with health insurance, as a function of age

You plot figure 3.13 with the command shown in the next listing.

Listing 3.13 Plotting the distribution of `health.ins` as a function of age

Add
smoothing
curve. →

```
ggplot(custdata2, aes(x=age, y=as.numeric(health.ins))) +
  geom_point(position=position_jitter(w=0.05, h=0.05)) +
  geom_smooth()
```

The Boolean variable `health.ins` must be converted to a 0/1 variable using `as.numeric`.

Since `y` values can only be 0 or 1, add a small jitter to get a sense of data density.

In our health insurance examples, the dataset is small enough that the scatter plots that you've created are still legible. If the dataset were a hundred times bigger, there would be so many points that they would begin to plot on top of each other; the scatter plot would turn into an illegible smear. In high-volume situations like this, try an aggregated plot, like a hexbin plot.

HEXBIN PLOTS

A *hexbin plot* is like a two-dimensional histogram. The data is divided into bins, and the number of data points in each bin is represented by color or shading. Let's go back to the income versus age example. Figure 3.14 shows a hexbin plot of the data. Note how the smoothing curve traces out the shape formed by the densest region of data.

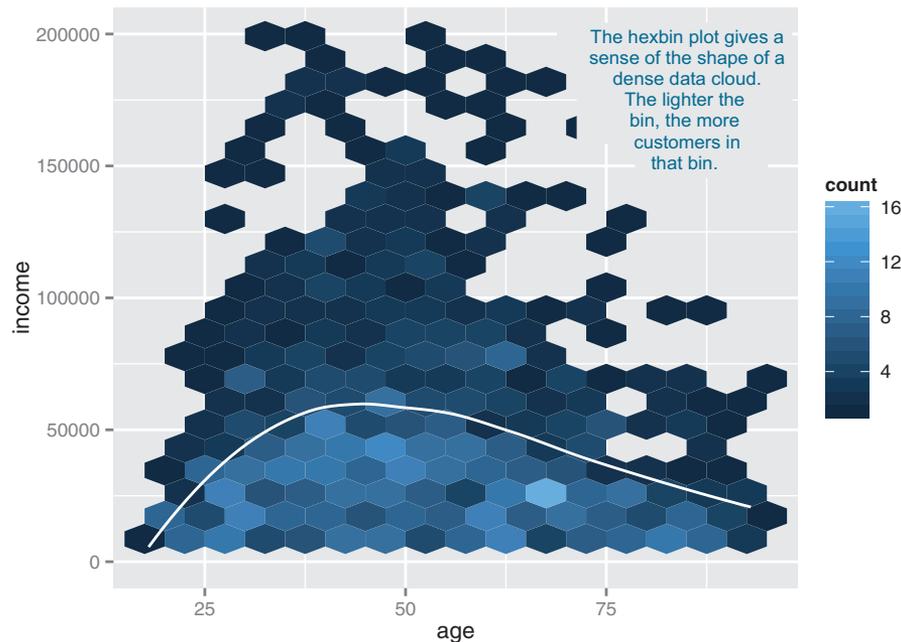


Figure 3.14 Hexbin plot of income versus age, with a smoothing curve superimposed in white

To make a hexbin plot in R, you must have the `hexbin` package installed. We'll discuss how to install R packages in appendix A. Once `hexbin` is installed and the library loaded, you create the plots using the `geom_hex` layer.

Listing 3.14 Producing a hexbin plot

```
library(hexbin)                                     ← Load hexbin library.
ggplot(custdata2, aes(x=age, y=income)) +
  geom_hex(binwidth=c(5, 10000)) +                 ← Add smoothing
  geom_smooth(color="white", se=F) +               ← curve in white;
  ylim(0, 200000)                                  suppress
                                                    standard error
                                                    ribbon (se=F).
```

Create hexbin with age binned into 5-year increments, income in increments of \$10,000.

In this section and the previous section, we've looked at plots where at least one of the variables is numerical. But in our health insurance example, the output is categorical, and so are many of the input variables. Next we'll look at ways to visualize the relationship between two categorical variables.

BAR CHARTS FOR TWO CATEGORICAL VARIABLES

Let's examine the relationship between marital status and the probability of health insurance coverage. The most straightforward way to visualize this is with a *stacked bar chart*, as shown in figure 3.15.

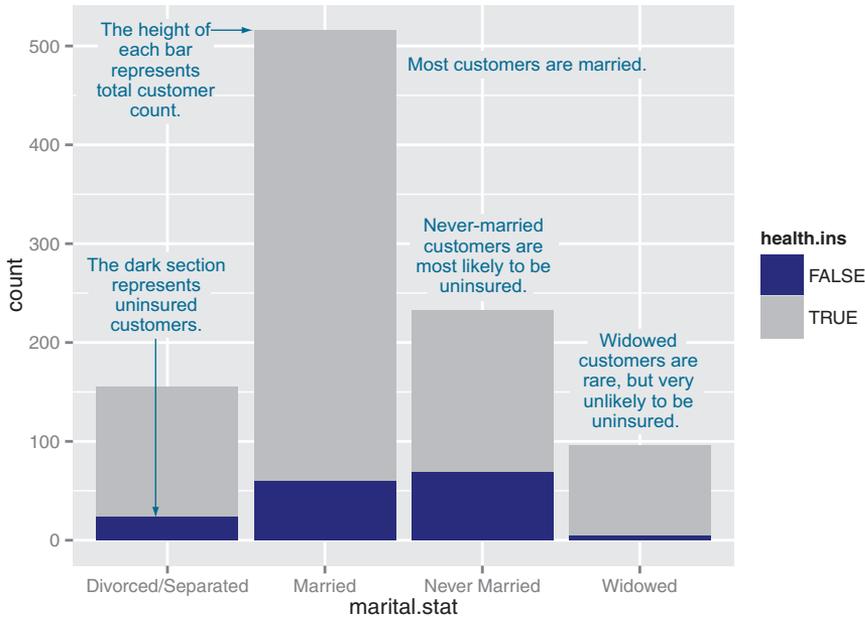


Figure 3.15 Health insurance versus marital status: stacked bar chart

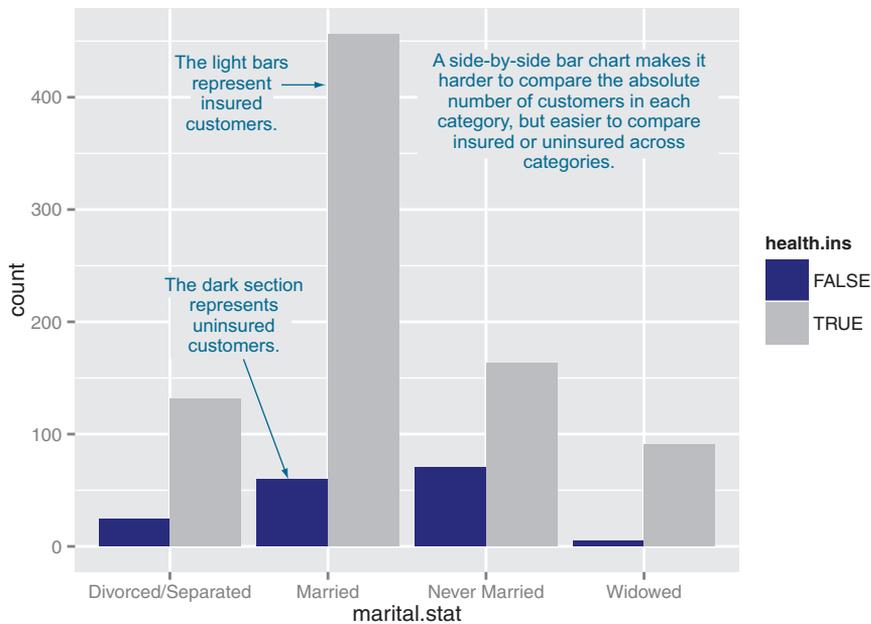


Figure 3.16 Health insurance versus marital status: side-by-side bar chart

Some people prefer the side-by-side bar chart, shown in figure 3.16, which makes it easier to compare the number of both insured and uninsured across categories.

The main shortcoming of both the stacked and side-by-side bar charts is that you can't easily compare the ratios of insured to uninsured across categories, especially for rare categories like *Widowed*. You can use what *ggplot2* calls a *filled bar chart* to plot a visualization of the ratios directly, as in figure 3.17.

The filled bar chart makes it obvious that divorced customers are slightly more likely to be uninsured than married ones. But you've lost the information that being widowed, though highly predictive of insurance coverage, is a rare category.

Which bar chart you use depends on what information is most important for you to convey. The *ggplot2* commands for each of these plots are given next. Note the use of the *fill* aesthetic; this tells *ggplot2* to color (fill) the bars according to the value of the variable *health.ins*. The *position* argument to *geom_bar* specifies the bar chart style.

Listing 3.15 Specifying different styles of bar chart

```
ggplot(custdata) + geom_bar(aes(x=marital.stat,
  fill=health.ins))                                     ← Stacked bar chart, the default

ggplot(custdata) + geom_bar(aes(x=marital.stat,
  fill=health.ins),
  position="dodge")                                     ← Side-by-side bar chart

ggplot(custdata) + geom_bar(aes(x=marital.stat,
  fill=health.ins),
  position="fill")                                     ← Filled bar chart
```

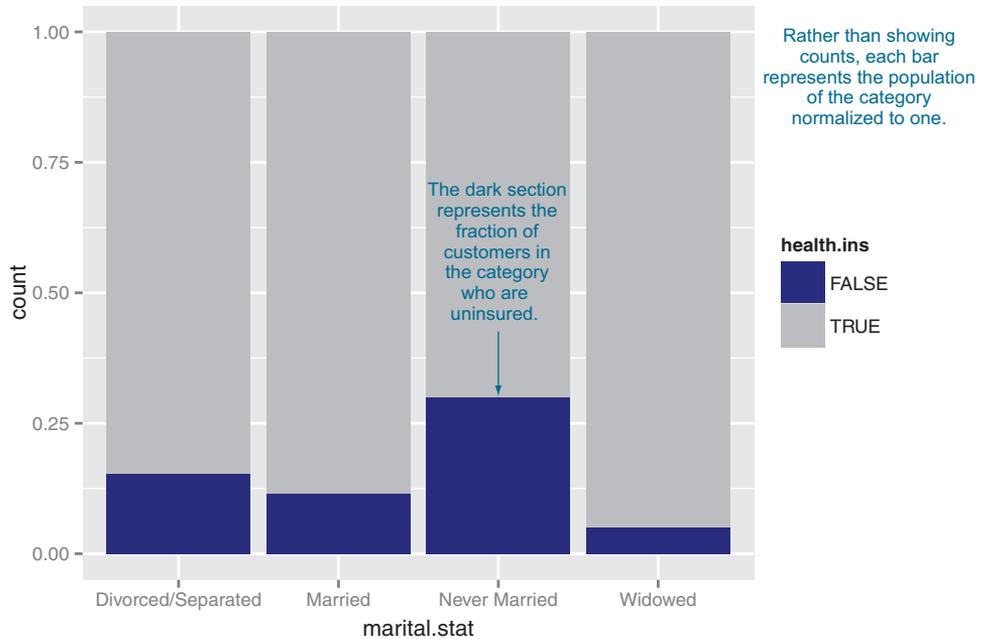


Figure 3.17 Health insurance versus marital status: filled bar chart

To get a simultaneous sense of both the population in each category and the ratio of insured to uninsured, you can add what's called a *rug* to the filled bar chart. A rug is a series of ticks or points on the x-axis, one tick per datum. The rug is dense where you have a lot of data, and sparse where you have little data. This is shown in figure 3.18. You generate this graph by adding a `geom_point` layer to the graph.

Listing 3.16 Plotting data with a rug

```
ggplot(custdata, aes(x=marital.stat)) +
  geom_bar(aes(fill=health.ins), position="fill") +
  geom_point(aes(y=-0.05), size=0.75, alpha=0.3,
    position=position_jitter(h=0.01))
```

← **Jitter the points slightly for legibility.**

Set the points just under the y-axis, three-quarters of default size, and make them slightly transparent with the alpha parameter.

In the preceding examples, one of the variables was binary; the same plots can be applied to two variables that each have several categories, but the results are harder to read. Suppose you're interested in the distribution of marriage status across housing types. Some find the side-by-side bar chart easiest to read in this situation, but it's not perfect, as you see in figure 3.19.

A graph like figure 3.19 gets cluttered if either of the variables has a large number of categories. A better alternative is to break the distributions into different graphs, one for each housing type. In `ggplot2` this is called *faceting* the graph, and you use the `facet_wrap` layer. The result is in figure 3.20.

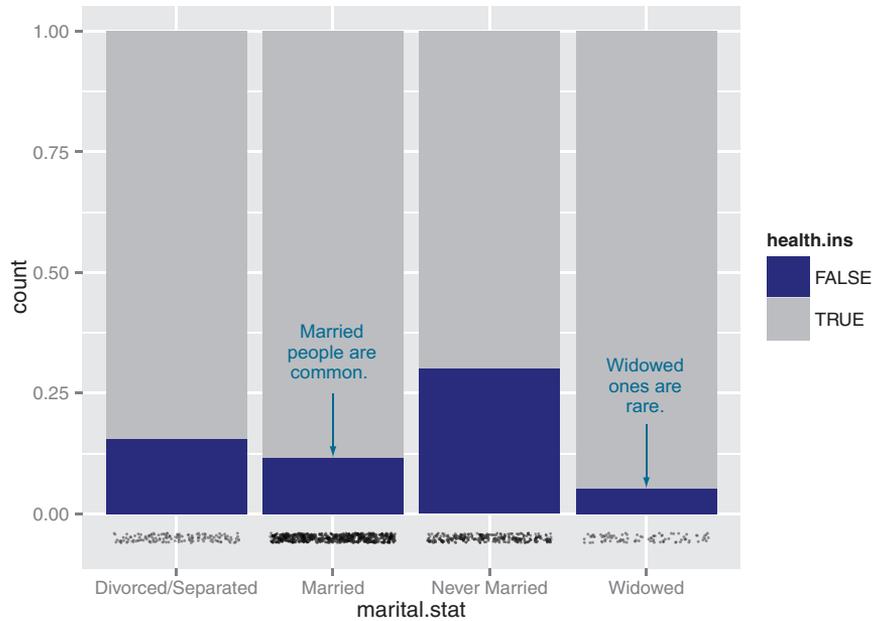


Figure 3.18 Health insurance versus marital status: filled bar chart with rug

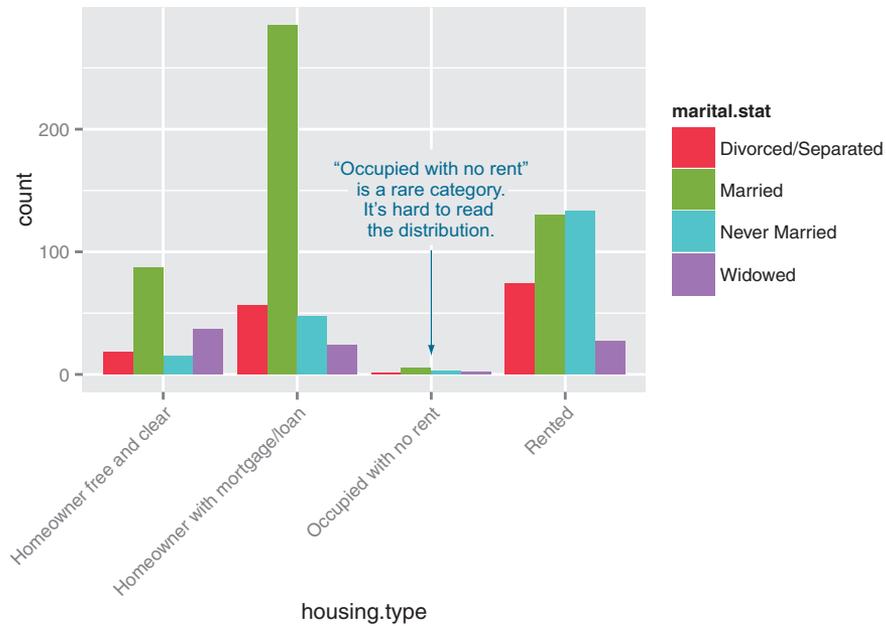


Figure 3.19 Distribution of marital status by housing type: side-by-side bar chart

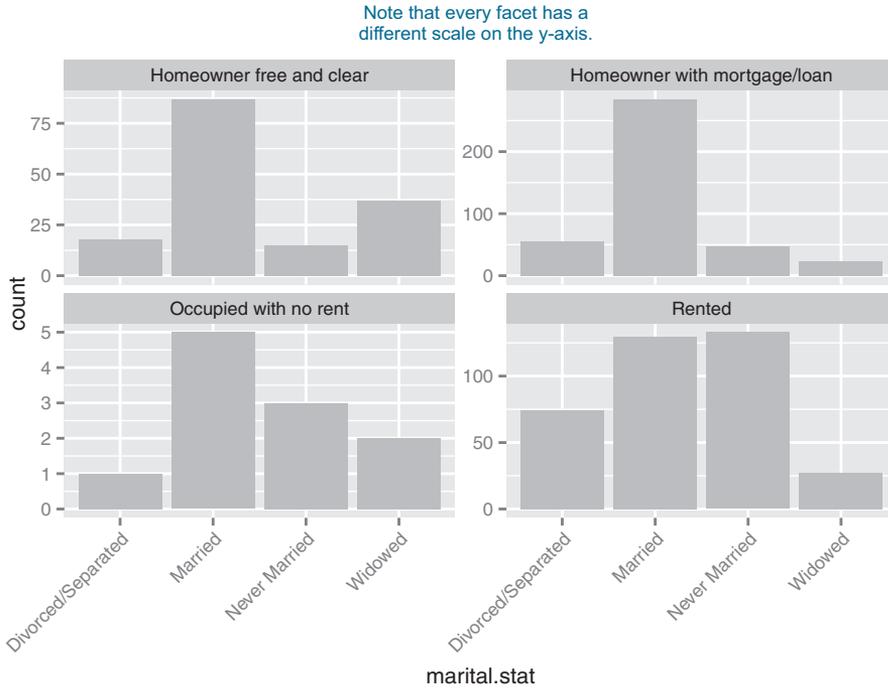


Figure 3.20 Distribution of marital status by housing type: faceted side-by-side bar chart

The code for figures 3.19 and 3.20 looks like the next listing.

Listing 3.17 Plotting a bar chart with and without facets

Side-by-side bar chart.

```
ggplot(custdata2) +
  geom_bar(aes(x=housing.type, fill=marital.stat),
    position="dodge") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

Tilt the x-axis labels so they don't overlap. You can also use `coord_flip()` to rotate the graph, as we saw previously. Some prefer `coord_flip()` because the `theme()` layer is complicated to use.

The faceted bar chart.

```
ggplot(custdata2) +
  geom_bar(aes(x=marital.stat), position="dodge",
    fill="darkgray") +
  facet_wrap(~housing.type, scales="free_y") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

Facet the graph by housing.type. The `scales="free_y"` argument specifies that each facet has an independently scaled y-axis (the default is that all facets have the same scales on both axes). The argument `free_x` would free the x-axis scaling, and the argument `free` frees both axes.

As of this writing, `facet_wrap` is incompatible with `coord_flip`, so we have to tilt the x-axis labels.

Table 3.2 summarizes the visualizations for two variables that we’ve covered.

Table 3.2 Visualizations for two variables

Graph type	Uses
Line plot	Shows the relationship between two continuous variables. Best when that relationship is functional, or nearly so.
Scatter plot	Shows the relationship between two continuous variables. Best when the relationship is too loose or cloud-like to be easily seen on a line plot.
Smoothing curve	Shows underlying “average” relationship, or trend, between two continuous variables. Can also be used to show the relationship between a continuous and a binary or Boolean variable: the fraction of <i>true</i> values of the discrete variable as a function of the continuous variable.
Hexbin plot	Shows the relationship between two continuous variables when the data is very dense.
Stacked bar chart	Shows the relationship between two categorical variables (<code>var1</code> and <code>var2</code>). Highlights the frequencies of each value of <code>var1</code> .
Side-by-side bar chart	Shows the relationship between two categorical variables (<code>var1</code> and <code>var2</code>). Good for comparing the frequencies of each value of <code>var2</code> across the values of <code>var1</code> . Works best when <code>var2</code> is binary.
Filled bar chart	Shows the relationship between two categorical variables (<code>var1</code> and <code>var2</code>). Good for comparing the relative frequencies of each value of <code>var2</code> within each value of <code>var1</code> . Works best when <code>var2</code> is binary.
Bar chart with faceting	Shows the relationship between two categorical variables (<code>var1</code> and <code>var2</code>). Best for comparing the relative frequencies of each value of <code>var2</code> within each value of <code>var1</code> when <code>var2</code> takes on more than two values.

There are many other variations and visualizations you could use to explore the data; the preceding set covers some of the most useful and basic graphs. You should try different kinds of graphs to get different insights from the data. It’s an interactive process. One graph will raise questions that you can try to answer by replotting the data again, with a different visualization.

Eventually, you’ll explore your data enough to get a sense of it and to spot most major problems and issues. In the next chapter, we’ll discuss some ways to address common problems that you may discover in the data.

3.3 Summary

At this point, you’ve gotten a feel for your data. You’ve explored it through summaries and visualizations; you now have a sense of the quality of your data, and of the relationships among your variables. You’ve caught and are ready to correct several kinds of data issues—although you’ll likely run into more issues as you progress.

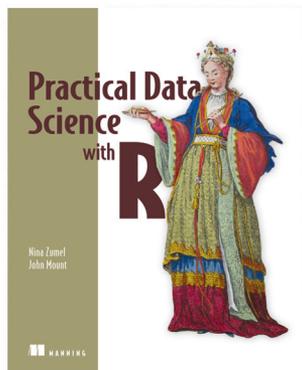
Maybe some of the things you’ve discovered have led you to reevaluate the question you’re trying to answer, or to modify your goals. Maybe you’ve decided that you

need more or different types of data to achieve your goals. This is all good. As we mentioned in the previous chapter, the data science process is made of loops within loops. The data exploration and data cleaning stages (we'll discuss cleaning in the next chapter) are two of the more time-consuming—and also the most important—stages of the process. Without good data, you can't build good models. Time you spend here is time you don't waste elsewhere.

In the next chapter, we'll talk about fixing the issues that you've discovered in the data.

Key takeaways

- Take the time to examine your data before diving into the modeling.
- The `summary` command helps you spot issues with data range, units, data type, and missing or invalid values.
- Visualization additionally gives you a sense of data distribution and relationships among variables.
- Visualization is an iterative process and helps answer questions about the data. Time spent here is time not wasted during the modeling process.



Business analysts and developers are increasingly collecting, curating, analyzing, and reporting on crucial business data. The R language and its associated tools provide a straightforward way to tackle day-to-day data science tasks without a lot of academic theory or advanced mathematics.

Practical Data Science with R shows you how to apply the R programming language and useful statistical techniques to everyday business situations. Using examples from marketing, business intelligence, and decision support, it shows you how to design experiments (such as A/B tests), build predictive models,

and present results to audiences of all levels.

What's inside

- Data science for the business professional
- Statistical analysis using the R language
- Project lifecycle, from planning to delivery
- Numerous instantly familiar use cases
- Keys to effective data presentations

This book is accessible to readers without a background in data science. Some familiarity with basic statistics, R, or another scripting language is assumed.

Real-world Data

Just like real life can sometimes be surprising and crazy and chaotic, real-life data can do all sorts of unexpected things. We, with our years of experience living in the real world, have conditioned ourselves to handle it all. That's a virtue of being human: we are fundamentally malleable and adaptive. Machine learning tools aren't quite as lucky. They do only what their code says to do, and—adaptive as machine learning tools are—if that code doesn't account for real-world outliers and other oddities, the machine learning tool will either grind to a halt or continue as normal while producing very abnormal and very problematic results in the face of adversity. Both of these outcomes are bad.

This chapter from *Real-World Machine Learning* tells you what might happen when you try to stuff data from real life into a sophisticated but decidedly unworldly learning algorithm. In addition to trying to scare you with the complexity of the task, the chapter demonstrates important and indispensable methods for avoiding the most serious pitfalls of machine learning. Following the guidelines and advice given here, your next machine-learning project might just be able to handle what the world throws at it.

Real-world data

This chapter covers

- Getting started with machine learning
- Collecting training data
- Using data-visualization techniques
- Preparing your data for ML

In supervised machine learning, you use data to teach automated systems how to make accurate decisions. ML algorithms are designed to discover patterns and associations in historical training data; they learn from that data and encode that learning into a model to accurately predict a data attribute of importance for new data. Training data, therefore, is fundamental in the pursuit of machine learning. With high-quality data, subtle nuances and correlations can be accurately captured and high-fidelity predictive systems can be built. But if training data is of poor quality, the efforts of even the best ML algorithms may be rendered useless.

This chapter serves as your guide to collecting and compiling training data for use in the supervised machine-learning workflow (figure 2.1). We give general guidelines for preparing training data for ML modeling and warn of some of the common pitfalls. Much of the art of machine learning is in exploring and visualizing training data to assess data quality and guide the learning process. To that end,

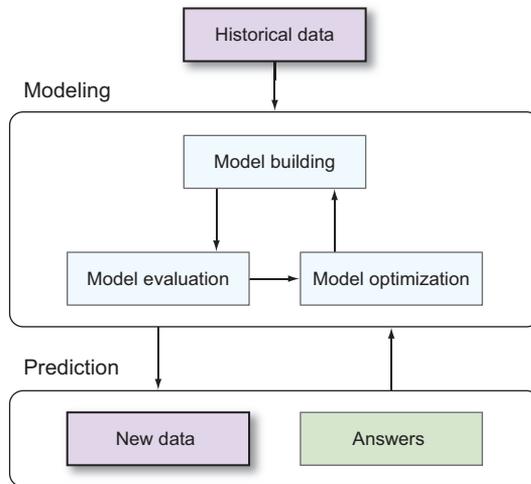


Figure 2.1 The basic ML workflow. Because this chapter covers data, we've highlighted the boxes indicating historical data and new data.

we provide an overview of some of the most useful data-visualization techniques. Finally, we discuss how to prepare a training dataset for ML model building, which is the subject of chapter 3.

This chapter uses a real-world machine-learning example: *churn prediction*. In business, *churn* refers to the act of a customer canceling or unsubscribing from a paid service. An important, high-value problem is to predict which customers are likely to churn in the near future. If a company has an accurate idea of which customers may unsubscribe from their service, then they may intervene by sending a message or offering a discount. This intervention can save companies millions of dollars, as the typical cost of new customer acquisition largely outpaces the cost of intervention on churners. Therefore, a machine-learning solution to churn prediction—whereby those users who are likely to churn are predicted weeks in advance—can be extremely valuable.

This chapter also uses datasets that are available online and widely used in machine-learning books and documentation: Titanic Passengers and Auto MPG datasets.

2.1 **Getting started: data collection**

To get started with machine learning, the first step is to ask a question that's suited for an ML approach. Although ML has many flavors, most real-world problems in machine learning deal with *predicting a target variable (or variables) of interest*. In this book, we cover primarily these supervised ML problems. Questions that are well suited for a supervised ML approach include the following:

- Which of my customers will churn this month?
- Will this user click my advertisement?
- Is this user account fraudulent?

- Is the sentiment of this tweet negative, positive, or neutral?
- What will demand for my product be next month?

You'll notice a few commonalities in these questions. First, they all require making assessments on one or several instances of interest. These instances can be people (such as in the churn question), events (such as the tweet sentiment question), or even periods of time (such as in the product demand question).

Second, each of these problems has a well-defined target of interest, which in some cases is binary (churn versus not churn, fraud versus not fraud), in some cases takes on multiple classes (negative versus positive versus neutral), or even hundreds or thousands of classes (picking a song out of a large library) and in others takes on numerical values (product demand). Note that in statistics and computer science, the *target* is also commonly referred to as the *response* or *dependent variable*. These terms may be used interchangeably.

Third, each of these problems can have sets of historical data in which the target is known. For instance, over weeks or months of data collection, you can determine which of your subscribers churned and which people clicked your ads. With some manual effort, you can assess the sentiment of different tweets. In addition to known target values, your historical data files will contain information about each instance that's knowable at the time of prediction. These are *input features* (also commonly referred to as the *explanatory* or *independent variables*). For example, the product usage history of each customer, along with the customer's demographics and account information, would be appropriate input features for churn prediction. The input features, together with the known values of the target variable, compose the *training set*.

Finally, each of these questions comes with an implied *action* if the target were knowable. For example, if you knew that a user would click your ad, you would bid on that user and serve the user an ad. Likewise, if you knew precisely your product demand for the upcoming month, you would position your supply chain to match that demand. The role of the ML algorithm is to use the training set to determine how the set of input features can most accurately predict the target variable. The result of this "learning" is encoded in a machine-learning model. When new instances (with an unknown target) are observed, their features are fed into the ML model, which generates predictions on those instances. Ultimately, those predictions enable the end user to take smarter (and faster) actions. In addition to producing predictions, the ML model allows the user to draw inferences about the relationships between the input features and the target variable.

Let's put all this in the context of the churn prediction problem. Imagine that you work for a telecom company and that the question of interest is, "Which of my current cell-phone subscribers will unsubscribe in the next month?" Here, each instance is a current subscriber. Likewise, the target variable is the binary outcome of whether each subscriber cancelled service during that month. The input features can consist of any information about each customer that's knowable at the beginning of the month, such as the current duration of the account, details on the subscription plan, and

usage information such as total number of calls made and minutes used in the previous month. Figure 2.2 shows the first four rows of an example training set for telecom churn prediction.

Features									Target
Cust. ID	State	Acct length	Area code	Int'l plan	Voicemail plan	Total messages	Total mins.	Total calls	Churned?
502	FL	124	561	No	Yes	28	251.4	104	False
1007	OR	48	503	No	No	0	190.4	92	False
1789	WI	63	608	No	Yes	34	152.2	119	False
2568	KY	58	606	No	No	0	247.2	116	True

Figure 2.2 Training data with four instances for the telecom churn problem

The aim of this section is to give a basic guide for properly collecting training data for machine learning. Data collection can differ tremendously from industry to industry, but several common questions and pain points arise when assembling training data. The following subsections provide a practical guide to addressing four of the most common data-collection questions:

- Which input features should I include?
- How do I obtain known values of my target variable?
- How much training data do I need?
- How do I know if my training data is good enough?

2.1.1 Which features should be included?

In machine-learning problems, you'll typically have dozens of features that you could use to predict the target variable. In the telecom churn problem, input attributes about each customer's demographics (age, gender, location), subscription plan (status, time remaining, time since last renewal, preferred status), and usage (calling history, text-messaging data and data usage, payment history) may all be available to use as input features. Only two practical restrictions exist on whether something may be used as an input feature:

- The value of the feature must be known at the time predictions are needed (for example, at the beginning of the month for the telecom churn example).
- The feature must be numerical or categorical in nature (chapter 5 shows how non-numerical data can be transformed into features via feature engineering).

Data such as Calling History data streams can be processed into a set of numerical and/or categorical features by computing summary statistics on the data, such as total minutes used, ratio of day/night minutes used, ratio of week/weekend minutes used, and proportion of minutes used in network.

Given such a broad array of possible features, which should you use? As a simple rule of thumb, features should be included only if they're suspected to be related to the target variable. Insofar as the goal of supervised ML is to predict the target, features that obviously have nothing to do with the target should be excluded. For example, if a distinguishing identification number was available for each customer, it shouldn't be used as an input feature to predict whether the customer will unsubscribe. Such useless features make it more difficult to detect the true relationships (signals) from the random perturbations in the data (noise). The more uninformative features are present, the lower the signal-to-noise ratio and thus the less accurate (on average) the ML model will be.

Likewise, excluding an input feature because it wasn't previously known to be related to the target can also hurt the accuracy of your ML model. Indeed, it's the role of ML to discover new patterns and relationships in data! Suppose, for instance, that a feature counting the number of current unopened voicemail messages was excluded from the feature set. Yet, some small subset of the population has ceased to check their voicemail because they decided to change carriers in the following month. This signal would express itself in the data as a slightly increased conditional probability of churn for customers with a large number of unopened voicemails. Exclusion of that input feature would deprive the ML algorithm of important information and therefore would result in an ML system of lower predictive accuracy. Because ML algorithms are able to discover subtle, nonlinear relationships, features beyond the known, first-order effects can have a substantial impact on the accuracy of the model.

In selecting a set of input features to use, you face a trade-off. On one hand, throwing every possible feature that comes to mind ("the kitchen sink") into the model can drown out the handful of features that contain any signal with an overwhelming amount of noise. The accuracy of the ML model then suffers because it can't distinguish true patterns from random noise. On the other extreme, hand-selecting a small subset of features that you already know are related to the target variable can cause you to omit other highly predictive features. As a result, the accuracy of the ML model suffers because the model doesn't know about the neglected features, which are predictive of the target.

Faced with this trade-off, the most practical approach is the following:

- 1 Include all the features that you suspect to be predictive of the target variable. Fit an ML model. If the accuracy of the model is sufficient, stop.
- 2 Otherwise, expand the feature set by including other features that are less obviously related to the target. Fit another model and assess the accuracy. If performance is sufficient, stop.
- 3 Otherwise, starting from the expanded feature set, run an ML *feature selection algorithm* to choose the best, most predictive subset of your expanded feature set.

We further discuss feature selection algorithms in chapter 5. These approaches seek the most accurate model built on a subset of the feature set; they retain the signal in

the feature set while discarding the noise. Though computationally expensive, they can yield a tremendous boost in model performance.

To finish this subsection, it's important to note that in order to use an input feature, that feature doesn't have to be present for each instance. For example, if the ages of your customers are known for only 75% of your client base, you could still use age as an input feature. We discuss ways to handle missing data later in the chapter.

2.1.2 How can we obtain ground truth for the target variable?

One of the most difficult hurdles in getting started with supervised machine learning is the aggregation of training instances with a known target variable. This process often requires running an existing, suboptimal system for a period of time, until enough training data is collected. For example, in building out an ML solution for telecom churn, you first need to sit on your hands and watch over several weeks or months as some customers unsubscribe and others renew. After you have enough training instances to build an accurate ML model, you can flip the switch and start using ML in production.

Each use case will have a different process by which ground truth—the actual or observed value of the target variable—can be collected or estimated. For example, consider the following training-data collection processes for a few selected ML use cases:

- *Ad targeting*—You can run a campaign for a few days to determine which users did/didn't click your ad and which users converted.
- *Fraud detection*—You can pore over your past data to figure out which users were fraudulent and which were legitimate.
- *Demand forecasting*—You can go into your historical supply-chain management data logs to determine the demand over the past months or years.
- *Twitter sentiment*—Getting information on the true intended sentiment is considerably harder. You can perform manual analysis on a set of tweets by having people read and opine on tweets (or use crowdsourcing).

Although the collection of instances of known target variables can be painful, both in terms of time and money, the benefits of migrating to an ML solution are likely to more than make up for those losses. Other ways of obtaining ground-truth values of the target variable include the following:

- Dedicating analysts to manually look through past or current data to determine or estimate the ground-truth values of the target
- Using crowdsourcing to use the “wisdom of crowds” in order to attain estimates of the target
- Conducting follow-up interviews or other hands-on experiments with customers
- Running controlled experiments (for example, A/B tests) and monitoring the responses

Each of these strategies is labor-intensive, but you can accelerate the learning process and shorten the time required to collect training data by collecting only target variables

for the instances that have the most influence on the machine-learning model. One example of this is a method called *active learning*. Given an existing (small) training set and a (large) set of data with unknown response variable, active learning identifies the subset of instances from the latter set whose inclusion in the training set would yield the most accurate ML model. In this sense, active learning can accelerate the production of an accurate ML model by focusing manual resources. For more information on active learning and related methods, see the 2009 presentation by Dasgupta and Langford from ICML.¹

2.1.3 How much training data is required?

Given the difficulty of observing and collecting the response variable for data instances, you might wonder how much training data is required to get an ML model up and running. Unfortunately, this question is so problem-specific that it's impossible to give a universal response or even a rule of thumb.

These factors determine the amount of training data needed:

- The complexity of the problem. Does the relationship between the input features and target variable follow a simple pattern, or is it complex and nonlinear?
- The requirements for accuracy. If you require only a 60% success rate for your problem, less training data is required than if you need to achieve a 95% success rate.
- The dimensionality of the feature space. If only two input features are available, less training data will be required than if there were 2,000 features.

One guiding principle to remember is that, as the training set grows, the models will (on average) get more accurate. (This assumes that the data remains *representative* of the ongoing data-generating process, which you'll learn more about in the next section.) More training data results in higher accuracy because of the data-driven nature of ML models. Because the relationship between the features and target is learned entirely from the training data, the more you have, the higher the model's ability to recognize and capture more-subtle patterns and relationships.

Using the telecom data from earlier in the chapter, we can demonstrate how the ML model improves with more training data and also offer a strategy to assess whether more training data is required. The telecom training dataset consists of 3,333 instances, each containing 19 features plus the binary outcome of unsubscribed versus renewed. Using this data, it's straightforward to assess whether you need to collect more data. Do the following:

- 1 Using the current training set, choose a grid of subsample sizes to try. For example, with this telecom training set of 3,333 instances of training data, your grid could be 500; 1,000; 1,500; 2,000; 2,500; 3,000.
- 2 For each sample size, randomly draw that many instances (without replacement) from the training set.

¹ See http://videlectures.net/icml09_dasgupta_langford_actl/.

- 3 With each subsample of training data, build an ML model and assess the accuracy of that model (we talk about ML evaluation metrics in chapter 4).
- 4 Assess how the accuracy changes as a function of sample size. If it seems to level off at the higher sample sizes, the existing training set is probably sufficient. But if the accuracy continues to rise for the larger samples, the inclusion of more training instances would likely boost accuracy.

Alternatively, if you have a clear accuracy target, you can use this strategy to assess whether that target has been fulfilled by your current ML model built on the existing training data (in which case it isn't necessary to amass more training data).

Figure 2.3 demonstrates how the accuracy of the fitted ML model changes as a function of the number of training instances used with the telecom dataset. In this case, it's clear that the ML model improves as you add training data: moving from 250 to 500 to 750 training examples produces significant improvements in the accuracy level. Yet, as you increase the number of training instances beyond 2,000, the accuracy

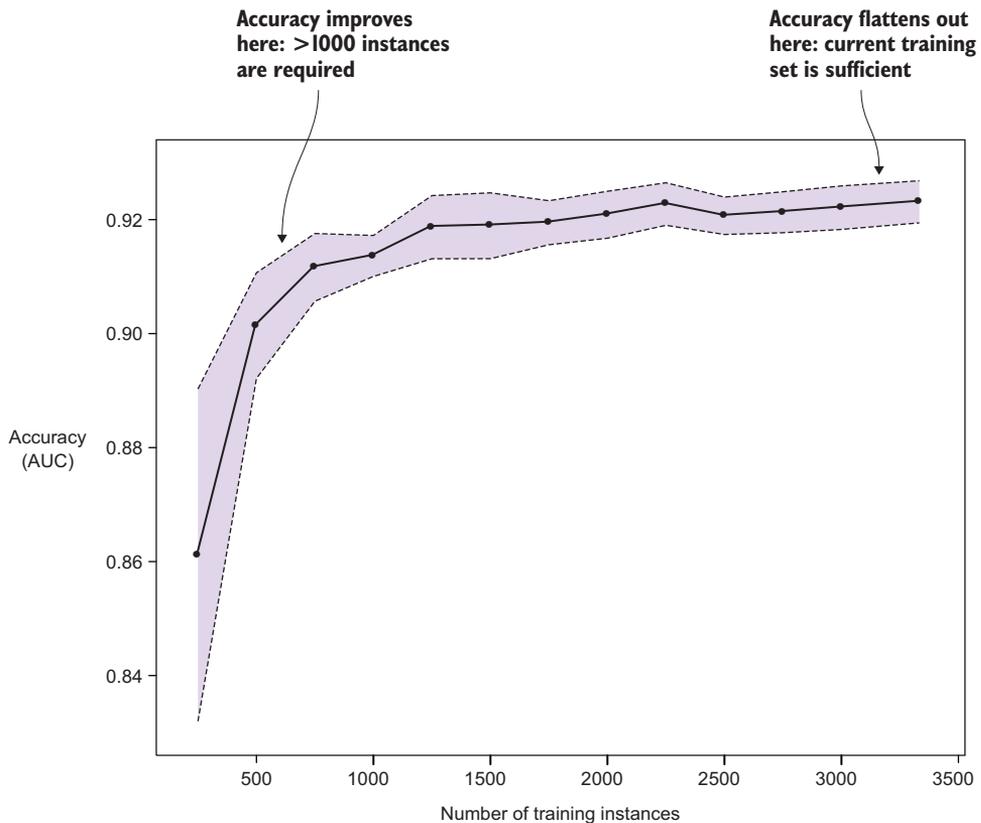


Figure 2.3 Testing whether the existing sample of 3,333 training instances is enough data to build an accurate telecom churn ML model. The black line represents the average accuracy over 10 repetitions of the assessment routine, and the shaded bands represent the error bands.

levels off. This is evidence that the ML model won't improve substantially if you add more training instances. (This doesn't mean that significant improvements couldn't be made by using more features.)

2.1.4 Is the training set representative enough?

Besides the size of the training set, another important factor for generating accurate predictive ML models is the *representativeness* of the training set. How similar are the instances in the training set to the instances that will be collected in the future? Because the goal of supervised machine learning is to generate accurate predictions on new data, it's fundamental that the training set be representative of the sorts of instances that you ultimately want to generate predictions for. A training set that consists of a nonrepresentative sample of what future data will look like is called *sample-selection bias* or *covariate shift*.

A training sample could be nonrepresentative for several reasons:

- It was possible to obtain ground truth for the target variable for only a certain, biased subsample of data. For example, if instances of fraud in your historical data were detected only if they cost the company more than \$1,000, then a model trained on that data will have difficulty identifying cases of fraud that result in losses less than \$1,000.
- The properties of the instances have changed over time. For example, if your training example consists of historical data on medical insurance fraud, but new laws have substantially changed the ways in which medical insurers must conduct their business, then your predictions on the new data may not be appropriate.
- The input feature set has changed over time. For example, say the set of location attributes that you collect on each customer has changed; you used to collect ZIP code and state, but now collect IP address. This change may require you to modify the feature set used for the model and potentially discard old data from the training set.

In each of these cases, an ML model fit to the training data may not extrapolate well to new data. To borrow an adage: you wouldn't necessarily want to use your model trained on apples to try to predict on oranges! The predictive accuracy of the model on oranges would likely not be good.

To avoid these problems, it's important to attempt to make the training set as representative of future data as possible. This entails structuring your training-data collection process in such a way that biases are removed. As we mention in the following section, visualization can also help ensure that the training data is representative.

Now that you have an idea of how to collect training data, your next task is to structure and assemble that data to get ready for ML model building. The next section shows how to preprocess your training data so you can start building models (the topic of chapter 3).

2.2 Preprocessing the data for modeling

Collecting data is the first step toward preparing the data for modeling, but sometimes you must run the data through a few preprocessing steps, depending on the composition of the dataset. Many machine-learning algorithms work only on numerical data—integers and real-valued numbers. The simplest ML datasets come in this format, but many include other types of features, such as categorical variables, and some have missing values. Sometimes you need to construct or compute features through feature engineering. Some numeric features may need to be rescaled to make them comparable or to bring them into line with a frequency distribution (for example, grading on the normal curve). In this section, you’ll look at these common data preprocessing steps needed for real-world machine learning.

2.2.1 Categorical features

The most common type of non-numerical feature is the categorical feature. A feature is *categorical* if values can be placed in buckets and the order of values isn’t important. In some cases, this type of feature is easy to identify (for example, when it takes on only a few string values, such as `spam` and `ham`). In other cases, whether a feature is a numerical (integer) feature or categorical isn’t so obvious. Sometimes either may be a valid representation, and the choice can affect the performance of the model. An example is a feature representing the day of the week, which could validly be encoded as either numerical (number of days since Sunday) or as categorical (the names Monday, Tuesday, and so forth). You aren’t going to look at model building and performance until chapters 3 and 4, but this section introduces a technique for dealing with categorical features. Figure 2.4 points out categorical features in a few datasets.

Person	Name	Age	Income	Marital status
1	Jane Doe	24	81,200	Single
2	John Smith	41	121,000	Married

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/O2. 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 2.4 Identifying categorical features. At the top is the simple `Person` dataset, which has a `Marital Status` categorical feature. At the bottom is a dataset with information about Titanic passengers. The features identified as categorical here are `Survived` (whether the passenger survived or not), `Pclass` (what class the passenger was traveling on), `Gender` (male or female), and `Embarked` (from which city the passenger embarked).

Some machine-learning algorithms use categorical features natively, but generally they need data in numerical form. You can encode categorical features as numbers (one number per category), but you can't use this encoded data as a true categorical feature because you've then introduced an (arbitrary) order of categories. Recall that one of the properties of categorical features is that they aren't ordered. Instead, you can convert each of the categories into a separate binary feature that has value 1 for instances for which the category appeared, and value 0 when it didn't. Hence, each categorical feature is converted to a set of binary features, one per category. Features constructed in this way are sometimes called *dummy variables*. Figure 2.5 illustrates this concept further.

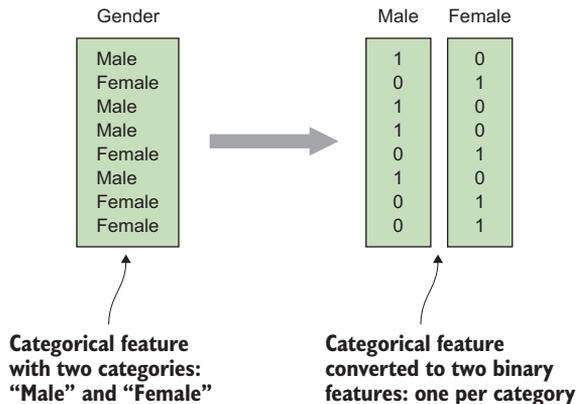


Figure 2.5 Converting categorical columns to numerical columns

The pseudocode for converting the categorical features in figure 2.5 to binary features looks like the following listing. Note that `categories` is a special NumPy type (www.numpy.org) such that `(data == cat)` yields a list of Boolean values.

Listing 2.1 Convert categorical features to numerical binary features

```
def cat_to_num(data):
    categories = unique(data)
    features = []
    for cat in categories:
        binary = (data == cat)
        features.append(binary.astype("int"))
    return features
```

NOTE Readers familiar with the Python programming language may have noticed that the preceding example isn't just pseudocode, but also valid Python. You'll see this a lot throughout the book: we introduce a code snippet as pseudocode, but unless otherwise noted, it's working code. To make the code simpler, we implicitly import a few helper libraries, such as `numpy` and `scipy`. Our examples will generally work if you include `from numpy import *`,

and from `scipy import *`. Note that although this approach is convenient for trying out examples interactively, you should never use it in real applications, because the `import *` construct may cause name conflicts and unexpected results. All code samples are available for inspection and direct execution in the accompanying GitHub repository: <https://github.com/brinkar/real-world-machine-learning>.

The categorical-to-numerical conversion technique works for most ML algorithms. But a few algorithms (such as certain types of decision-tree algorithms and related algorithms such as random forests) can use categorical features natively. This will often yield better results for highly categorical datasets, and we discuss this further in the next chapter. Our simple Person dataset, after conversion of the categorical feature to binary features, is shown in figure 2.6.

Person	Name	Age	Income	Marital status: Single	Marital status: Married
1	Jane Doe	24	81,200	1	0
2	John Smith	41	121,000	0	1

Figure 2.6 The simple Person dataset after conversion of the categorical Marital Status feature to binary numerical features. (The original dataset is shown in figure 2.4.)

2.2.2 Dealing with missing data

You’ve already seen a few examples of datasets with missing data. In tabular datasets, missing data often appears as empty cells, or cells with NaN (Not a Number), N/A, or None. Missing data is usually an artifact of the data-collection process; for some reason, a particular value couldn’t be measured for a data instance. Figure 2.7 shows an example of missing data in the Titanic Passengers dataset.

There are two main types of missing data, which you need to handle in different ways. First, for some data, *the fact that it’s missing* can carry meaningful information that could be useful for the ML algorithm. The other possibility is that the data is missing

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/O2. 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

Missing values

Figure 2.7 The Titanic Passengers dataset has missing values in the Age and Cabin columns. The passenger information has been extracted from various historical sources, so in this case the missing values stem from information that couldn’t be found in the sources.

only because its measurement was impossible, and the unavailability of the information isn't otherwise meaningful. In the Titanic Passengers dataset, for example, missing values in the Cabin column may indicate that those passengers were in a lower social or economic class, whereas missing values in the Age column carry no useful information (the age of a particular passenger at the time simply couldn't be found).

Let's first consider the case of *informative* missing data. When you believe that information is missing from the data, you usually want the ML algorithm to be able to use this information to potentially improve the prediction accuracy. To achieve this, you want to convert the missing values into the same format as the column in general. For numerical columns, this can be done by setting missing values to -1 or -999 , depending on typical values of non-null values. Pick a number at one end of the numerical spectrum that will denote missing values, and remember that order is important for numerical columns. You don't want to pick a value in the middle of the distribution of values.

For a categorical column with potentially informative missing data, you can create a new category called Missing, None, or similar, and then handle the categorical feature in the usual way (for example, using the technique described in the previous section). Figure 2.8 shows a simple diagram of what to do with meaningful missing data.

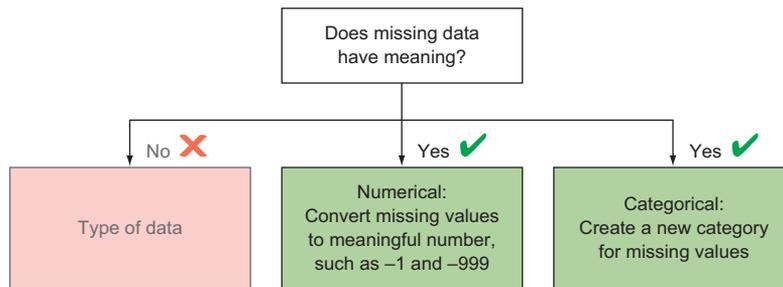


Figure 2.8 What to do with meaningful missing data

When the absence of a value for a data item has no informative value in itself, you proceed in a different way. In this case, you can't introduce a special number or category because you might introduce data that's flat-out wrong. For example, if you were to change any missing values in the Age column of the Titanic Passengers dataset to -1 , you'd probably hurt the model by messing with the age distribution for no good reason. Some ML algorithms will be able to deal with these truly missing values by ignoring them. If not, you need to preprocess the data to either eliminate missing values or replace them by guessing the true value. This concept of replacing missing data is called *imputation*.

If you have a large dataset and only a handful of missing values, dropping the observations with missing data is the easiest approach. But when a larger portion of

your observations contain missing values, the loss of perfectly good data in the dropped observations will reduce the predictive power of your model. Furthermore, if the observations with missing values aren't randomly distributed throughout your dataset, this approach may introduce unexpected bias.

Another simple approach is to assume some temporal order to the data instances and replace missing values with the column value of the preceding row. With no other information, you're making a guess that a measurement hasn't changed from one instance to the next. Needless to say, this assumption will often be wrong, but less wrong than, for example, filling in zeros for the missing values, especially if the data is a series of sequential observations (yesterday's temperature isn't an unreasonable estimate of today's). And for extremely big data, you won't always be able to apply more-sophisticated methods, and these simple methods can be useful.

When possible, it's usually better to use a larger portion of the existing data to guess the missing values. You can replace missing column values by the mean or median value of the column. With no other information, you assume that the average will be closest to the truth. Depending on the distribution of column values, you might want to use the median instead; the mean is sensitive to outliers. These are widely used in machine learning today and work well in many cases. But when you set all missing values to a single new value, you diminish the visibility of potential correlation with other variables that may be important in order for the algorithm to detect certain patterns in the data.

What you want to do, if you can, is use all the data at your disposal to predict the value of the missing variable. Does this sound familiar? This is exactly what machine learning is about, so you're basically thinking about building ML models in order to be able to build ML models. In practice, you'll typically use a simple algorithm (such as linear or logistic regression, described in chapter 3) to impute the missing data. This isn't necessarily the same as the main ML algorithm used. In any case, you're creating a pipeline of ML algorithms that introduces more knobs to turn in order to optimize the model in the end.

Again, it's important to realize that there's no single best way to deal with truly missing data. We've discussed a few ways in this section, and figure 2.9 summarizes the possibilities.

2.2.3 Simple feature engineering

Chapter 5 covers domain-specific and advanced feature-engineering techniques, but it's worth mentioning the basic idea of simple data preprocessing in order to make the model better.

You'll use the Titanic example again in this section. Figure 2.10 presents another look at part of the data, and in particular the Cabin feature. Without processing, the Cabin feature isn't necessarily useful. Some values seem to include multiple cabins, and even a single cabin wouldn't seem like a good categorical feature because all cabins would be separate "buckets." If you want to predict, for example, whether a certain

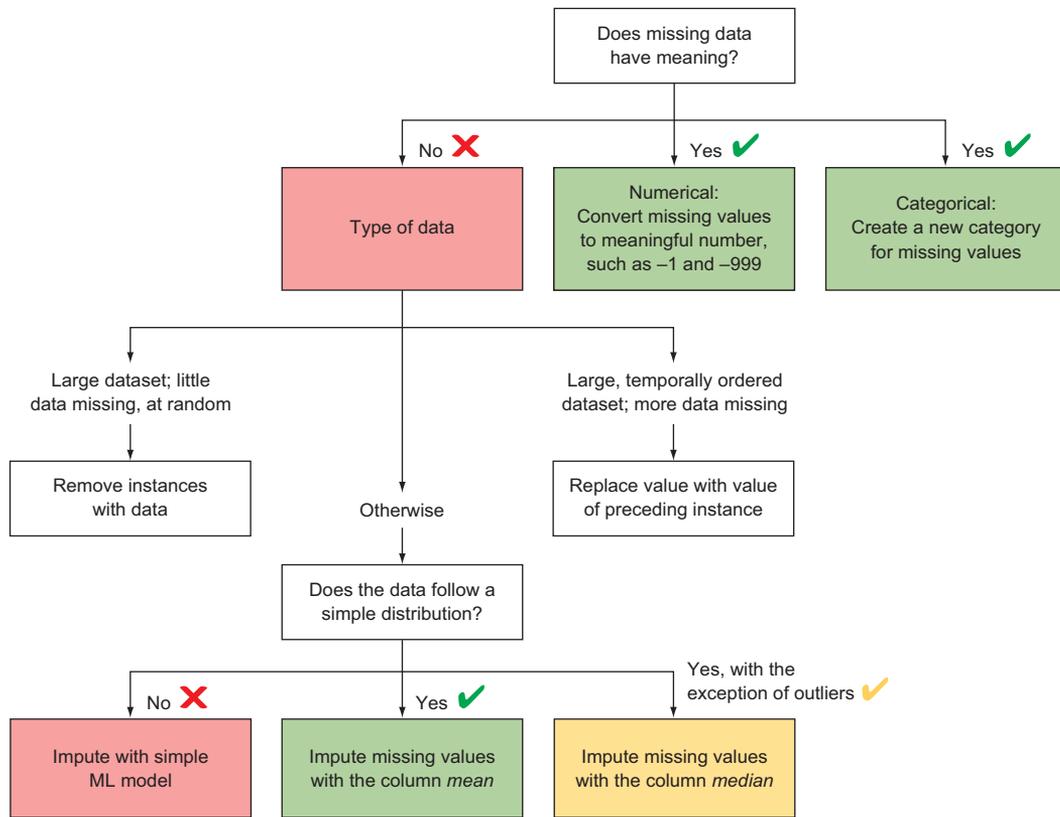


Figure 2.9 Full decision diagram for handling missing values when preparing data for ML modeling

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/O2. 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 2.10 In the Titanic Passengers dataset, some Cabin values include multiple cabins, whereas others are missing. And cabin identifiers themselves may not be good categorical features.

passenger survived, living in a particular cabin instead of the neighboring cabin may not have any predictive power.

Living in a particular section of the ship, though, could be important for survival. For single cabin IDs, you could extract the letter as a categorical feature and the number as a numerical feature, assuming they denote different parts of the ship. You could even find a layout map of the Titanic and map each cabin to the level and side of the ship, ocean-facing versus interior, and so forth. These approaches don't handle multiple cabin IDs, but because it looks like all multiple cabins are close to each other, extracting only the first cabin ID should be fine. You could also include the number of cabins in a new feature, which could also be relevant.

All in all, you'll create three new features from the Cabin feature. The following listing shows the code for this simple extraction.

Listing 2.2 Simple feature extraction on Titanic cabins

```
def cabin_features(data):
    features = []
    for cabin in data:
        cabins = cabin.split(" ")
        n_cabins = len(cabins)
        # First char is the cabin_char
        try:
            cabin_char = cabins[0][0]
        except IndexError:
            cabin_char = "X"
            n_cabins = 0
        # The rest is the cabin number
        try:
            cabin_num = int(cabins[0][1:])
        except:
            cabin_num = -1
        # Add 3 features for each passenger
        features.append( [cabin_char, cabin_num, n_cabins] )
    return features
```

By now it should be no surprise what we mean by *feature engineering*: using the existing features to create new features that increase the value of the original data by applying our knowledge of the data or domain in question. As mentioned earlier, you'll look at advanced feature-engineering concepts and common types of data that need to be processed to be used by most algorithms. These include free-form text features for things such as web pages or tweets. Other important features can be extracted from images, video, and time-series data as well.

2.2.4 Data normalization

Some ML algorithms require data to be *normalized*, meaning that each individual feature has been manipulated to reside on the same numeric scale. The value range of a feature can influence the importance of the feature compared to other features. If

one feature has values between 0 and 10, and another has values between 0 and 1, the weight of the first feature is 10, compared to the second. Sometimes you'll want to force a particular feature weight, but typically it's better to let the ML algorithm figure out the relative weights of the features. To make sure all features are considered equally, you need to normalize the data. Often data is normalized to be in the range from 0 to 1, or from -1 to 1.

Let's consider how this normalization is performed. The following code listing implements this function. For each feature, you want the data to be distributed between a minimum value (typically -1) and a maximum value (typically +1). To achieve this, you divide the data by the total range of the data in order to get the data into the 0-1 range. From here, you can re-extend to the required range (2, in the case of -1 to +1) by multiplying with this transformed value. At last, you move the starting point from 0 to the minimum required value (for example, -1).

Listing 2.3 Feature normalization

```
def normalize_feature(data, f_min=-1.0, f_max=1.0):
    d_min, d_max = min(data), max(data)
    factor = (f_max - f_min) / (d_max - d_min)
    normalized = f_min + (data - d_min)*factor
    return normalized, factor
```

Note that you return both the normalized data and the factor with which the data was normalized. You do this because any new data (for example, for prediction) will have to be normalized in the same way in order to yield meaningful results. This also means that the ML modeler will have to remember how a particular feature was normalized, and save the relevant values (factor and minimum value).

We leave it up to you to implement a function that takes new data, the normalization factor, and the normalized minimum value and reapplies the normalization.

As you expand your data-wrangling toolkit and explore a variety of data, you'll begin to see that each dataset has qualities that make it uniquely interesting, and often challenging. But large collections of data with many variables are hard to fully understand by looking at tabular representations. Graphical data-visualization tools are indispensable for understanding the data from which you hope to extract hidden information.

2.3 Using data visualization

Between data collection/preprocessing and ML model building lies the important step of data visualization. Data visualization serves as a sanity check of the training features and target variable before diving into the mechanics of machine learning and prediction. With simple visualization techniques, you can begin to explore the relationship between the input features and the output target variable, which will guide you in model building and assist in your understanding of the ML model and predictions.

		Input feature	
		Categorical	Numerical
Response variable	Categorical	Mosaic plots Section 2.3.1	Box plots Section 2.3.2
		Numerical	Density plots Section 2.3.3

Figure 2.11 Four visualization techniques, arranged by the type of input feature and response variable to be plotted

Further, visualization techniques can tell you how representative the training set is and inform you of the types of instances that may be lacking.

This section focuses on methods for visualizing the association between the target variable and the input features. We recommend four visualization techniques: mosaic plots, box plots, density plots, and scatter plots. Each technique is appropriate for a different type (numeric or categorical) of input feature and target variable, as shown in figure 2.11.

Further reading

A plethora of books are dedicated to statistical visualization and plotting data. If you'd like to dive deeper into this topic, check out the following:

- The classic textbook *The Visual Display of Quantitative Information* by Edward Tufte (Graphics Press, 2001) presents a detailed look into visualizing data for analysis and presentation.
- For R users, *R Graphics Cookbook* by Winston Chang (O'Reilly, 2013) covers data visualization in R, from the basics to advanced topics, with code samples to follow along.
- For Python users, *Python Data Visualization Cookbook* by Igor Milovanović, Dimitry Foures, and Giuseppe Vettigli (Packt Publishing, 2015) covers the basics to get you up and running with Matplotlib.

2.3.1 Mosaic plots

Mosaic plots allow you to visualize the relationship between two or more categorical variables. Plotting software for mosaic plots is available in R, SAS, Python, and other scientific or statistical programming languages.

To demonstrate the utility of mosaic plots, you'll use one to display the relationship between passenger gender and survival in the Titanic Passengers dataset. The mosaic plot begins with a square whose sides each have length 1. The square is then

divided, by vertical lines, into a set of rectangles whose widths correspond to the proportion of the data belonging to each of the categories of the input feature. For example, in the Titanic data, 24% of passengers were female, so you split the unit square along the x-axis into two rectangles corresponding to a width 24% / 76% of the area.

Next, each vertical rectangle is split by horizontal lines into subrectangles whose relative areas are proportional to the percent of instances belonging to each category of the response variable. For example, of Titanic passengers who were female, 74% survived (this is the *conditional probability* of survival, given that the passenger was female). Therefore, the Female rectangle is split by a horizontal line into two subrectangles that contain 74% / 26% of the area of the rectangle. The same is repeated for the Male rectangle (for males, the breakdown is 19% / 81%).

What results is a quick visualization of the relationship between gender and survival. If there is no relationship, the horizontal splits would occur at similar locations on the y-axis. If a strong relationship exists, the horizontal splits will be far apart. To enhance the visualization, the rectangles are shade-coded to assess the statistical significance of the relationship, compared to independence of the input feature and response variable, with large negative residuals (“lower count than expected”) shaded dark gray, and large positive residuals (“higher count than expected”) shaded light gray; see figure 2.12.

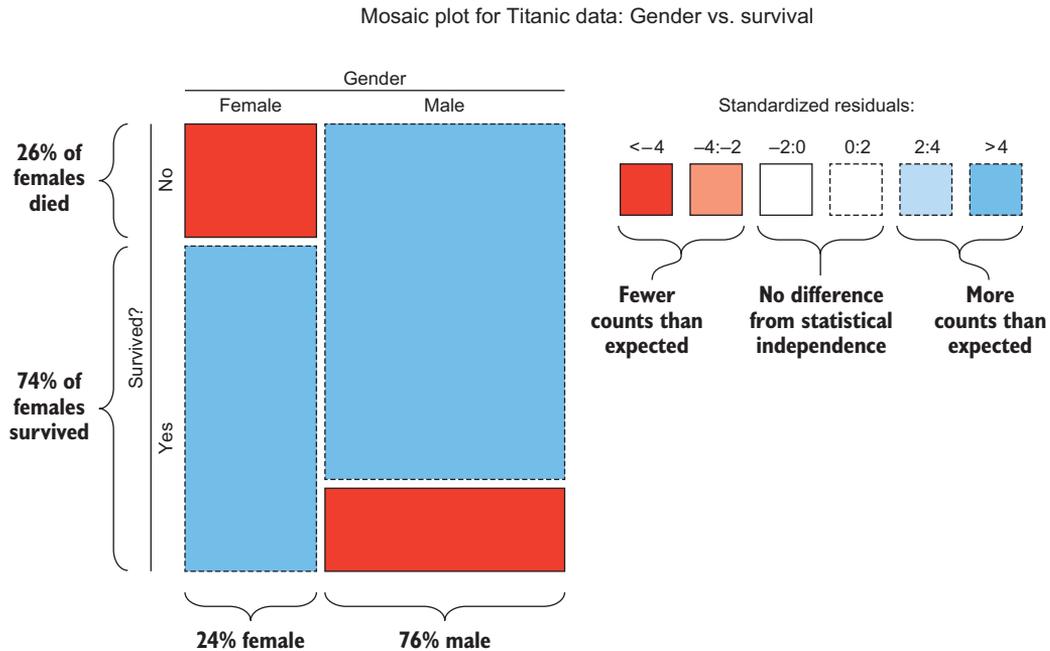


Figure 2.12 Mosaic plot showing the relationship between gender and survival on the Titanic. The visualization shows that a much higher proportion of females (and much smaller proportion of males) survived than would have been expected if survival were independent of gender. “Women and children first.”

This tells you that when building a machine-learning model to predict survival on the Titanic, gender is an important factor to include. It also allows you to perform a sanity check on the relationship between gender and survival: indeed, it's common knowledge that a higher proportion of women survived the disaster. This gives you an extra layer of assurance that your data is legitimate. Such data visualizations can also help you interpret and validate your machine-learning models, after they've been built.

Figure 2.13 shows another mosaic plot for survival versus passenger class (first, second, and third). As expected, a higher proportion of first-class passengers (and a lower proportion of third-class passengers) survived the sinking. Obviously, passenger class is also an important factor in an ML model to predict survival, and the relationship is exactly as you should expect: higher-class passengers had a higher probability of survival.

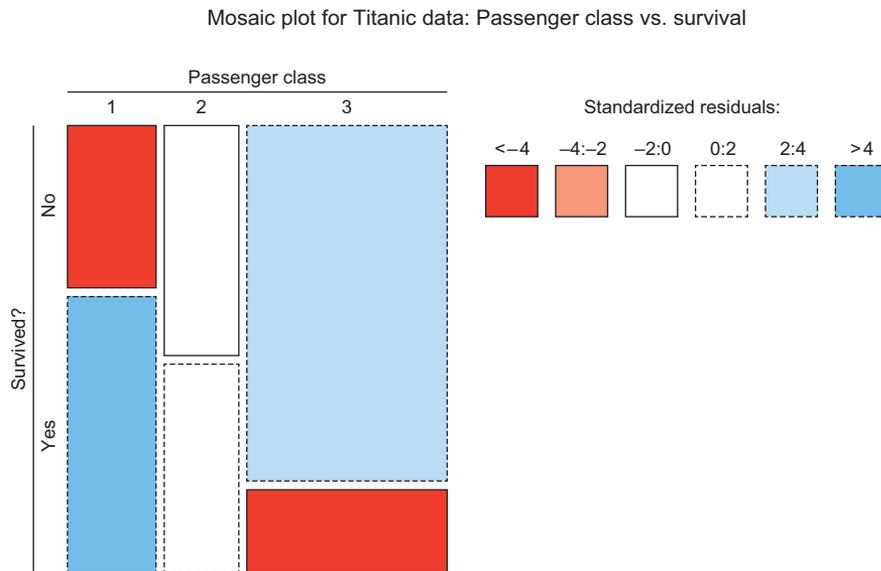


Figure 2.13 Mosaic plot showing the relationship between passenger class and survival on the Titanic

2.3.2 **Box plots**

Box plots are a standard statistical plotting technique for visualizing the distribution of a numerical variable. For a single variable, a box plot depicts the *quartiles* of its distribution: the minimum, 25th percentile, median, 75th percentile, and maximum of the values. Box-plot visualization of a single variable is useful to get insight into the center, spread, and skew of its distribution of values plus the existence of any outliers.

You can also use box plots to compare distributions when plotted in parallel. In particular, they can be used to visualize the difference in the distribution of a numerical feature as a function of the various categories of a categorical response variable.

Returning to the Titanic example, you can visualize the difference in ages between survivors and fatalities by using parallel box plots, as in figure 2.14. In this case, it's not clear that any differences exist in the distribution of passenger ages of survivors versus fatalities, as the two box plots look fairly similar in shape and location.

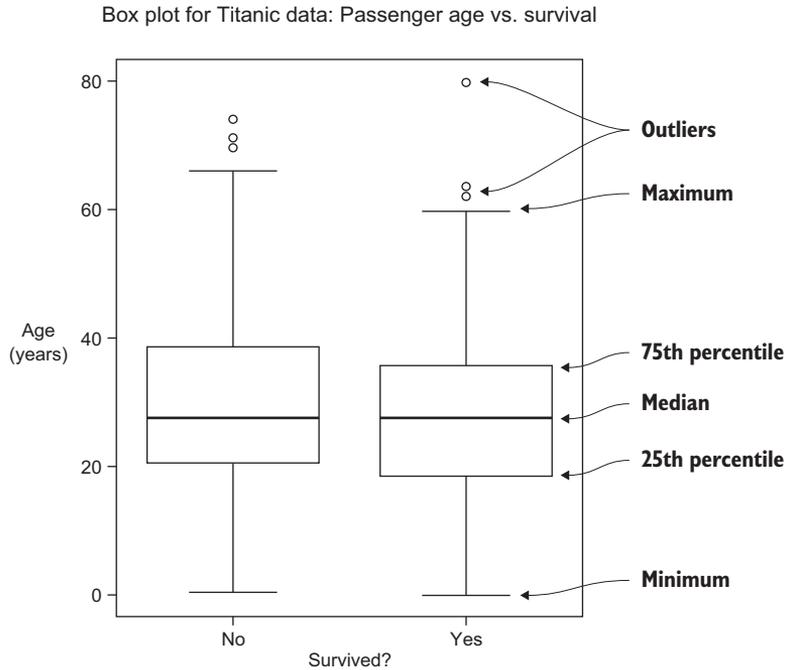


Figure 2.14 Box plot showing the relationship between passenger age and survival on the Titanic. No noticeable differences exist between the age distributions for survivors versus fatalities. (This alone *shouldn't* be a reason to exclude age from the ML model, as it may still be a predictive factor.)

It's important to recognize the limitations of visualization techniques. Visualizations aren't a substitute for ML modeling! Machine-learning models can find and exploit subtle relationships hidden deep inside the data that aren't amenable to being exposed via simple visualizations. You shouldn't automatically exclude features whose visualizations don't show clear associations with the target variable. These features could still carry a strong association with the target when used in association with other input features. For example, although age doesn't show a clear relationship with survival, it could be that for third-class passengers, age is an important predictor (perhaps for third-class passengers, the younger and stronger passengers could make their way to the deck of the ship more readily than older passengers). A good ML model will discover and expose such a relationship, and thus the visualization alone isn't meant to exclude age as a feature.

Figure 2.15 displays box plots exploring the relationship between passenger fare paid and survival outcome. In the left panel, it's clear that the distributions of fare paid are highly skewed (many small values and a few large outliers), making the differences difficult to visualize. This is remedied by a simple transformation of the fare (square root, in the right panel), making the differences easy to spot. Fare paid has an obvious relationship with survival status: those paying higher fares were more likely to survive, as is expected. Thus, fare amount should be included in the model, as you expect the ML model to find and exploit this positive association.

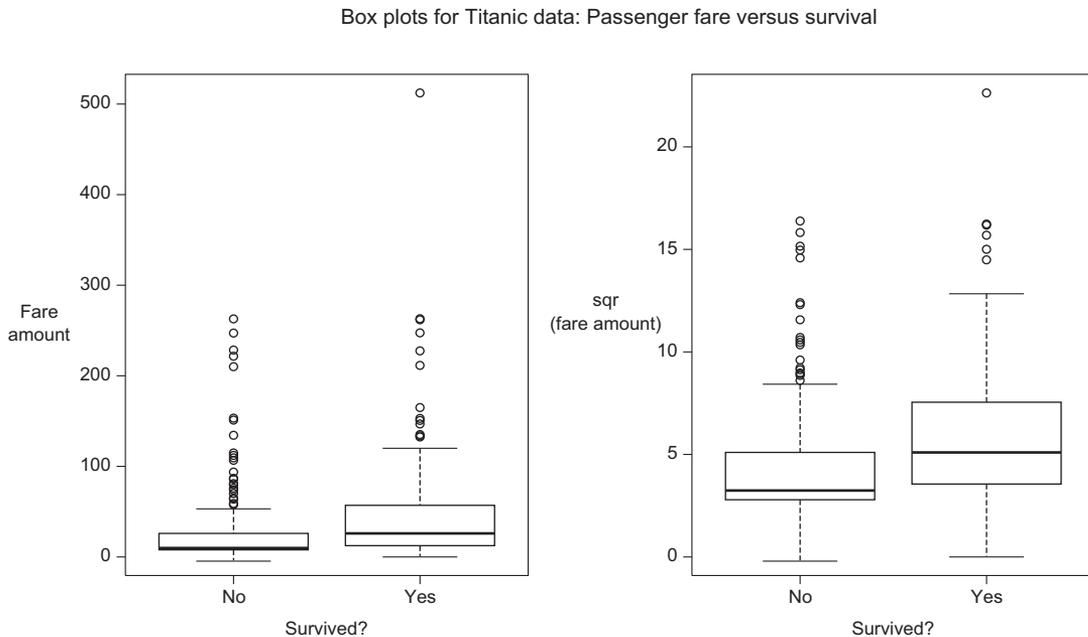


Figure 2.15 Box plots showing the relationship between passenger fare paid and survival on the Titanic. The square-root transformation makes it obvious that passengers who survived paid higher fares, on average.

2.3.3 *Density plots*

Now, we move to numerical, instead of categorical, response variables. When the input variable is categorical, you can use box plots to visualize the relationship between two variables, just as you did in the preceding section. You can also use density plots.

Density plots display the distribution of a single variable in more detail than a box plot. First, a smoothed estimate of the probability distribution of the variable is estimated (typically using a technique called *kernel smoothing*). Next, that distribution is plotted as a curve depicting the values that the variable is likely to have. By creating a single density plot of the response variable for each category that the input feature takes, you can easily visualize any discrepancies in the values of the response variable for differences in the categorical input feature. Note that density plots are similar to

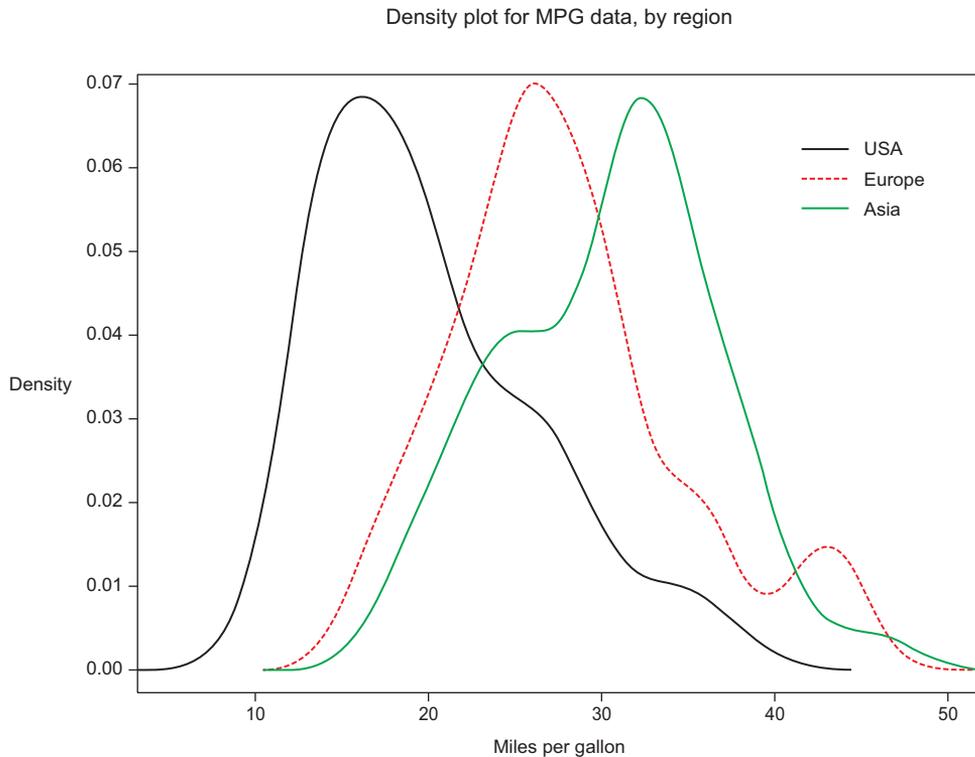


Figure 2.16 Density plot for the Auto MPG dataset, showing the distribution of vehicle MPG for each manufacturer region. It's obvious from the plot that Asian cars tend to have the highest MPG and that cars made in the United States have the lowest. Region is clearly a strong indicator of MPG.

histograms, but their smooth nature makes it much simpler to visualize multiple distributions in a single figure.

In the next example, you'll use the Auto MPG dataset.¹ This dataset contains the miles per gallon (MPG) attained by each of a large collection of automobiles from 1970–82, plus attributes about each auto, including horsepower, weight, location of origin, and model year. Figure 2.16 presents a density plot for MPG versus location of origin (United States, Europe, or Asia). It's clear from the plot that Asian cars tend to have higher MPG, followed by European and then American cars. Therefore, location should be an important predictor in our model. Further, a few secondary “bumps” in the density occur for each curve, which may be related to different types of automobile (for example, truck versus sedan versus hybrid). Thus, extra exploration of these secondary bumps is warranted to understand their nature and to use as a guide for further feature engineering.

¹ The Auto MPG dataset is available at <https://archive.ics.uci.edu/ml/datasets/Auto+MPG> and is standard in the R programming language, by entering `data(mtcars)`.

2.3.4 Scatter plots

A *scatter plot* is a simple visualization of the relationship between two numerical variables and is one of the most popular plotting tools in existence. In a scatter plot, the value of the feature is plotted versus the value of the response variable, with each instance represented as a dot. Though simple, scatter plots can reveal both linear and nonlinear relationships between the input and response variables.

Figure 2.17 shows two scatter plots: one of car weight versus MPG, and one of car model year versus MPG. In both cases, clear relationships exist between the input features and the MPG of the car, and hence both should be used in modeling. In the left panel is a clear banana shape in the data, showing a nonlinear decrease in MPG for increasing vehicle weight. Likewise, the right panel shows an increasing, linear relationship between MPG and the model year. Both plots clearly indicate that the input features are useful in predicting MPG, and both have the expected relationship.

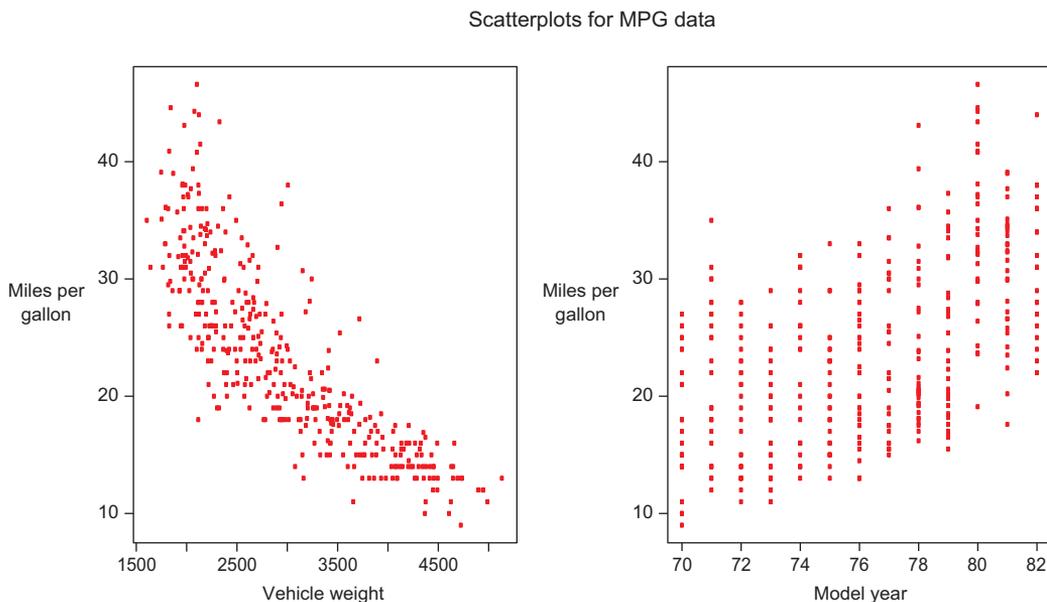


Figure 2.17 Scatter plots for the relationship of vehicle miles per gallon versus vehicle weight (left) and vehicle model year (right)

2.4 Summary

In this chapter, you've looked at important aspects of data in the context of real-world machine learning:

- Steps in compiling your training data include the following:
 - Deciding which input features to include
 - Figuring out how to obtain ground-truth values for the target variable

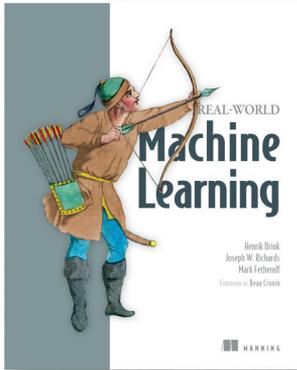
- Determining when you've collected enough training data
- Keeping an eye out for biased or nonrepresentative training data
- Preprocessing steps for training data include the following:
 - Recoding categorical features
 - Dealing with missing data
 - Feature normalization (for some ML approaches)
 - Feature engineering
- Four useful data visualizations are mosaic plots, density plots, box plots, and scatter plots:

		Input Feature	
		Categorical	Numerical
Response Variable	Categorical	Mosaic plots	Box plots
	Numerical	Density plots	Scatter plots

With our data ready for modeling, let's now start building machine-learning models!

2.5 Terms from this chapter

Word	Definition
dummy variable	A binary feature that indicates that an observation is (or isn't) a member of a category
ground truth	The value of a known target variable or label for a training or test set
missing data imputation	Those features with unknown values for a subset of instances Replacement of the unknown values of missing data with numerical or categorical values



Machine learning systems help you find valuable insights and patterns in data, which you'd never recognize with traditional methods. In the real world, ML techniques give you a way to identify trends, forecast behavior, and make fact-based recommendations. It's a hot and growing field, and up-to-speed ML developers are in demand.

Real-World Machine Learning will teach you the concepts and techniques you need to be a successful machine learning practitioner without overdosing you on abstract theory and complex mathematics. By working through immediately relevant examples in Python, you'll build skills in data acquisition and modeling, classification, and regression. You'll also explore the most important tasks like model validation, optimization, scalability, and real-time streaming. When you're done, you'll be ready to successfully build, deploy, and maintain your own powerful ML systems.

What's inside

- Predicting future behavior
- Performance evaluation and optimization
- Analyzing sentiment and making recommendations

No prior machine learning experience assumed. Readers should know Python.

Symbols

+ operator 42

A

active learning method 71
ad targeting 70
aesthetics 41–42
API keys 18
APIs (application programming interface)
17–19
ASCII text 10
Auto MPG dataset 87

B

bar charts
checking distributions for single variable
47–50
checking relationships between two
variables 56–61
big data 6–9
bimodal distribution 43
binwidth parameter 44
box plots 84

C

categorical variables 74, 82
categories type 75
churn prediction 66–67
combining data sources 24–25
comma-separated value. *See* CSV
computers users, as data generators 3–5

conditional probability 69, 83
coord_flip command 60
copyright 22–23
covariate shift 73
CSV (comma-separated value) 11–12

D

data
indiscriminate collection of 5–6
scouting for 21–27
combining data sources 24–25
copyright and licensing 22–23
measuring or collecting things yourself
26–27
using Google search 22
web scraping 25–26
data collection
amount of training data required 71
deciding which features to include 68
obtaining ground truth for target variable 70
whether training set is representative
enough 73
data generators, computers and internet users
as 3–5
data normalization 80
data science
defined 3
data scientists
as explorer 6–9
data visualization
box plots 84
density plots 86
mosaic plots 82
scatter plots 88

data wrangling
 PDFs and 20
 database indexes 15–16
 databases
 non-relational 17
 relational 15–17
 delimited files 11–12
 demand forecasting 70
 density plots 44–47, 86
 dependent variable 67
 distribution shape 42
 dot plot 49
 dummy variables 75, 89

E

Elasticsearch 17
 encoding categorical features 75
 EOL (end-of-line) character 10
 error-checking data
 checking distributions for single variable
 bar charts 47
 density plots 44
 histograms 43
 checking relationships between two variables
 bar charts 56
 hexbin plots 55
 line plots 51
 scatter plots 52
 summary command
 data ranges 38
 invalid values 38
 missing values 37
 outliers 38
 overview 35
 units 39
 using visualizations 40
 Excel. *See* Microsoft Excel
 explanatory variables 67
 exploring data
 checking distributions for single variable
 bar charts 47–50
 density plots 44–47
 histograms 43–44
 checking relationships between two variables
 bar charts 56–61
 hexbin plots 55–56
 line plots 51–52
 scatter plots 52–55
 summary command
 data ranges 38–39
 invalid values 38
 missing values 37–38

outliers 38
 overview 35–37
 units 39–40
 using visualizations 40–42
 Extensible Business Reporting Language. *See*
 XBRL

F

faceting graph 58
 factor
 summary command 36
 file formats
 bad 19–20
 deciding which to use 20–21
 flat files 10–12
 HTML 12–13
 JSON 14–15
 unusual 20
 XML 13–14
 filled bar chart 57
 flat files 10–12
 fraud detection 70

G

generators of data. *See* data generators
 geom layers 53
 ggplot2 41–42
 Google, big data use by 8
 ground truth 73, 89
 guessing missing values 78

H

hexbin plots 55–56
 histogram
 checking distributions for single variable 43–44
 defined 44
 HTML (hypertext markup language) 12–13
 HttpURLConnection package 19

I

imputation 77, 89
 independent variables 67
 indexing in databases 15–16
 informative missing data 77
 input features 67–69, 71, 81–82, 85, 88
 input variables 88
 integer features 74
 internet users, as data generators 3–5

invalid values 38
 IoT (Internet of Things) 4, 27

J

joining tables 16–17
 JSON (JavaScript object notation) 14–15, 18

K

kernel smoothing 86

L

legality of data usage 22–23
 licensing 22–23
 line plots 51–52
 loess function 53
 log files 27
 logarithmic scale
 density plot 46
 when to use 47
 lowess function 53

M

max command 36
 McKinlay, Chris 26
 mean command 36
 median command 36
 microRNA example 27–31
 Microsoft Excel 19
 Microsoft Word 19
 min command 36
 miRanda algorithm 29–31
 missing data 70, 76–77, 89
 missing values 74, 76–79
 checking data using summary command 37–38
 MongoDB 17
 mosaic plots 82
 multimodal distribution 43

N

NaN (Not a Number) 76
 narrow data ranges 39
 non-relational databases 17
 normalization
 organizing data for analysis 35
 normalized data 80
 NoSQL 17

Not a Number. *See* NaN
 numerical features 74

O

OpenOffice Calc 19
 organizing data for analysis 35
 outliers 38

P

PDFs
 overview 19–20
 plain text 10–11
 plots
 box plots 84
 density plots 86
 mosaic plots 82–84
 scatter plots 88–89
 preprocessing data 74–81
 categorical features 74–76
 data normalization 80–81
 dealing with missing data 76–78
 simple feature engineering 78–80
 Python programming language
 reading of flat files by 12

Q

quantile() function 36
 quartiles 84
 querying, using databases 15–17

R

R programming language
 reading of flat files by 12
 RDKit package 20
 relational databases 15–17
 relationships
 visually checking
 bar charts 56–61
 hexbin plots 55–56
 line plots 51–52
 scatter plots 52–55
 representative data 71
 representativeness training sets 73
 response variables 67, 88
 REST API 18–19
 rug, defined 58

S

sample-selection bias 73
 scatter plot 52–55
 scatter plots 88–89
 scouting for data 21–27
 combining data sources 24–25
 copyright and licensing 22–23
 measuring or collecting things yourself 26–27
 using Google search 22
 web scraping 25–26
 scraping web 13, 25–27
 scripts
 overview 27
 shape of distribution 42
 smoothing curves 53
 SQL (Structured Query Language) 15
 stacked bar chart 56
 stat layers 53
 Structured Query Language. *See* SQL
 summary() function
 checking data for errors
 data ranges 38–39
 invalid values 38
 missing values 37–38
 outliers 38
 overview 35–37
 units 39–40

T

tables
 joining 16–17
 tab-separated value. *See* TSV
 TargetScan algorithm 29–31
 telecom churn 68, 70, 72
 temporal data order 78
 training set 67–68, 71–73, 82
 TSV (tab-separated value) 11
 Tumblr, API of 18–19
 Twitter, big data use by 8

U

Uber example 23–24

unimodal distribution 42
 units
 checking data using summary command 39–40
 url package 19
 urllib package 19
 UTF-8 text 10

V

variables
 checking distributions for visually
 bar charts 47–50
 density plots 44–47
 histograms 43–44
 overview 42–43
 factor class and summary command 36
 visualizations for one 50–51
 visualizations for two 61
 variables of interest 66
 variance command 36
 visualizations
 checking distributions for single variable
 bar charts 47–50
 density plots 44–47
 histograms 43–44
 overview 42–43
 checking relationships between two variables
 bar charts 56–61
 hexbin plots 55–56
 line plots 51–52
 scatter plots 52–55
 overview 40–42

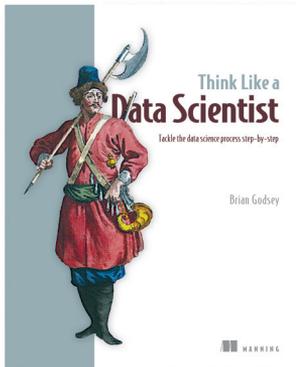
W

web scraping 13, 25–27
 Word. *See* Microsoft Word

X

XBRL (Extensible Business Reporting
 Language) 14
 XML (extensible markup language) 13–14

Save 50% on these selected books—eBook, pBook, and MEAP. Just enter **feedj50** in the Promotional Code box when you check out. Only at mannings.com.



Think Like a Data Scientist

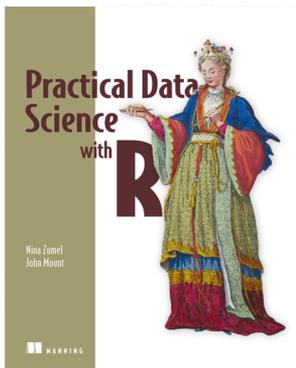
by Brian Godsey

ISBN: 9781633430273

328 pages

\$44.99

March 2017



Practical Data Science with R

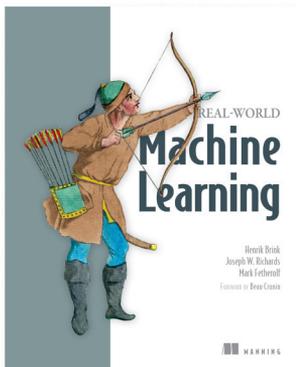
by Nina Zumel and John Mount

ISBN: 9781617291562

416 pages

\$49.99

March 2014



Real-World Machine Learning

by Henrik Brink, Joseph W. Richards, and Mark Fetherolf

ISBN: 9781617291920

264 pages

\$49.99

September 2016