Author Picks

FREE

Exploring Data Science

Chapters selected by
John Mount and Nina Zumel

manning

Exploring Data Science

Selections by John Mount and Nina Zumel

Manning Author Picks

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

♾ 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 9781617294181
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

Data science is a broad field that touches on aspects of statistics, machine learning, and data engineering. What the tools, methods, and work look like depend a lot on your problem domain and point of view. Our book, *Practical Data Science with R*, introduces readers to basic predictive modeling in the R language. But it was never our intent to imply that data scientists can restrict themselves to one problem domain or one implementation language.

This is a great time to get into data science. The number of free tools and materials has exploded. Storing and managing large data sets is now markedly easier. However, this diversity can seem overwhelming and divisive. A traditional statistician may not consider text analytics to be data science, and similarly somebody using neural nets to analyze images may not appreciate classic statistical inference.

We believe your problem helps to choose your technique. To illustrate this concept, we have put together this sampler of chapters from our book and other Manning titles. They cover diverse topics relevant to data science, highlighting a variety of domains and programming languages. We hope these selections give you a better picture of the many tools available to solve your specific data science problems. In fact we hope that after you appreciate these clear demonstrations of different domains and methodologies, for your own domain you have an "Aha!" moment where you see how one particular methodology applies to a problem you care about.

Exploring data

Data exploration and data cleaning are oft-neglected topics, yet they are also the most crucial—and most time-consuming—parts of the data science process. Without good data, you can't develop effective models or glean important insights. R's interactive environment and visualization capabilities make it particularly well-suited for the task of understanding your data. The following chapter covers some common data issues and how to detect them via visualization and other exploration techniques in R. The chapter also provides an introduction to R's ggplot2 visualization package, a flexible grammar for creating rich and informative plots.

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

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 sco⑦e 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

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

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

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

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

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

About 84% of the customers have health insurance.

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

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).

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 factors[2]). 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 Income = 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 income in units of $1000."

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.

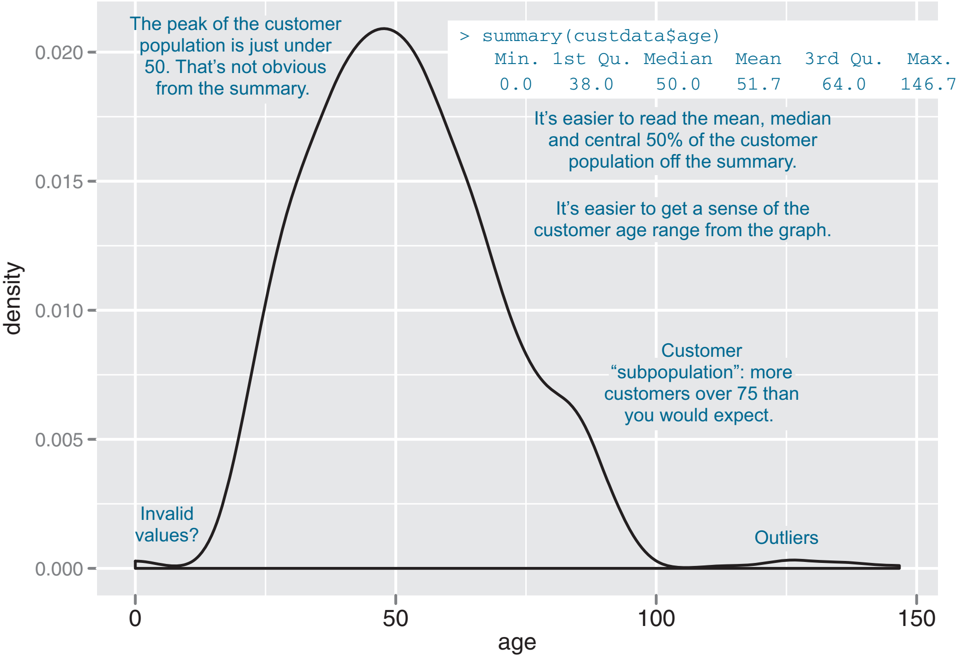
> *We cannot expect a small number of numerical values [summary statistics] to consistently convey the wealth of information that exists in data. Numerical reduction methods do not retain the information in the data.*
>
> —William Cleveland
> *The Elements of Graphing Data*

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.



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

The peak of the customer population is just under 50. That's not obvious from the summary.

It's easier to read the mean, median and central 50% of the customer population off the summary.

It's easier to get a sense of the customer age range from the graph.

Customer "subpopulation": more customers over 75 than you would expect.

Invalid values?

Outliers

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

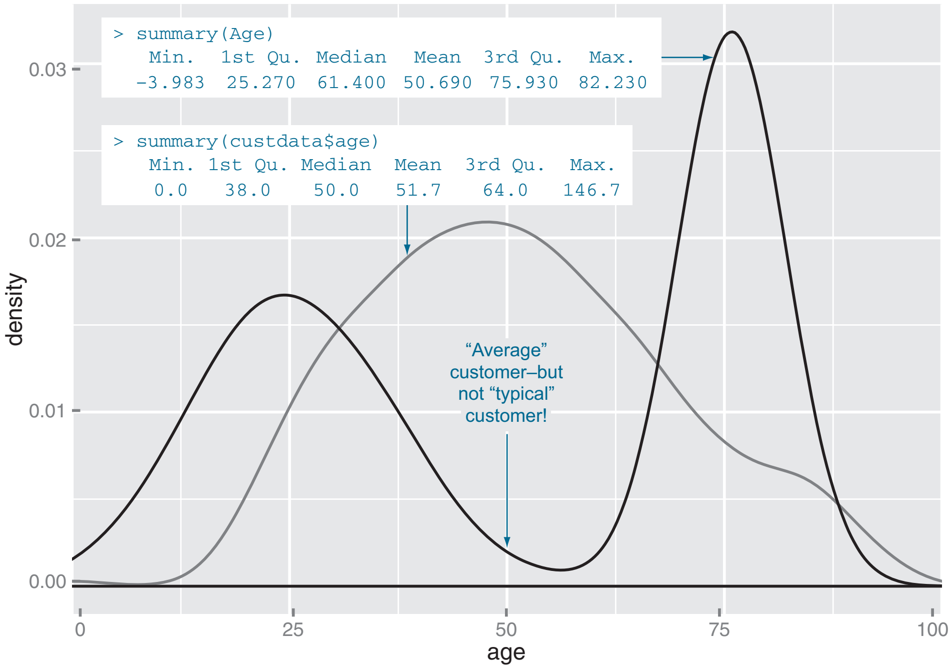
The visualizations in this section help you answer questions like these:

- 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.



```
> summary(Age)
   Min.  1st Qu.  Median   Mean  3rd Qu.   Max.
 -3.983  25.270  61.400  50.690  75.930  82.230
```

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

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

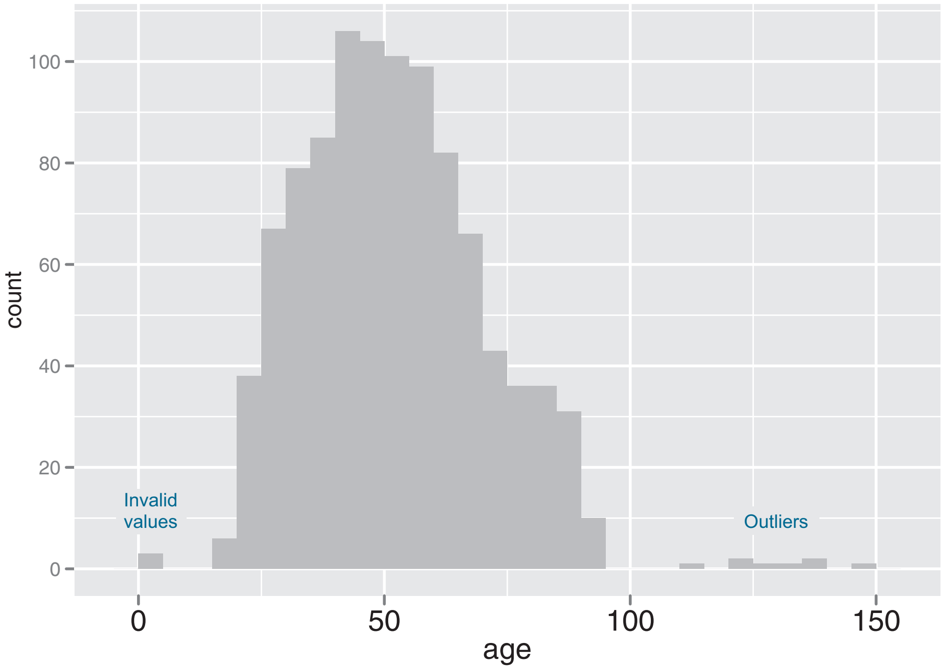


Figure 3.3 A histogram tells you where your data is concentrated. It also visually highlights outliers and anomalies.

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")
```

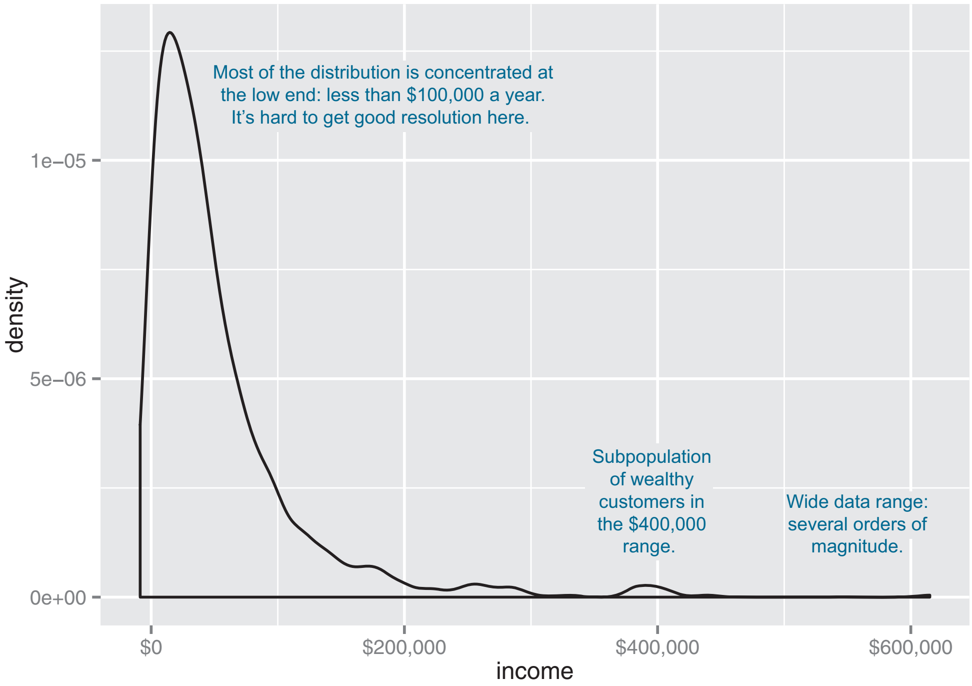
Load the ggplot2 library, if you haven't already done so.

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



Most of the distribution is concentrated at the low end: less than $100,000 a year. It's hard to get good resolution here.

Subpopulation of wealthy customers in the $400,000 range.

Wide data range: several orders of magnitude.

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)`.

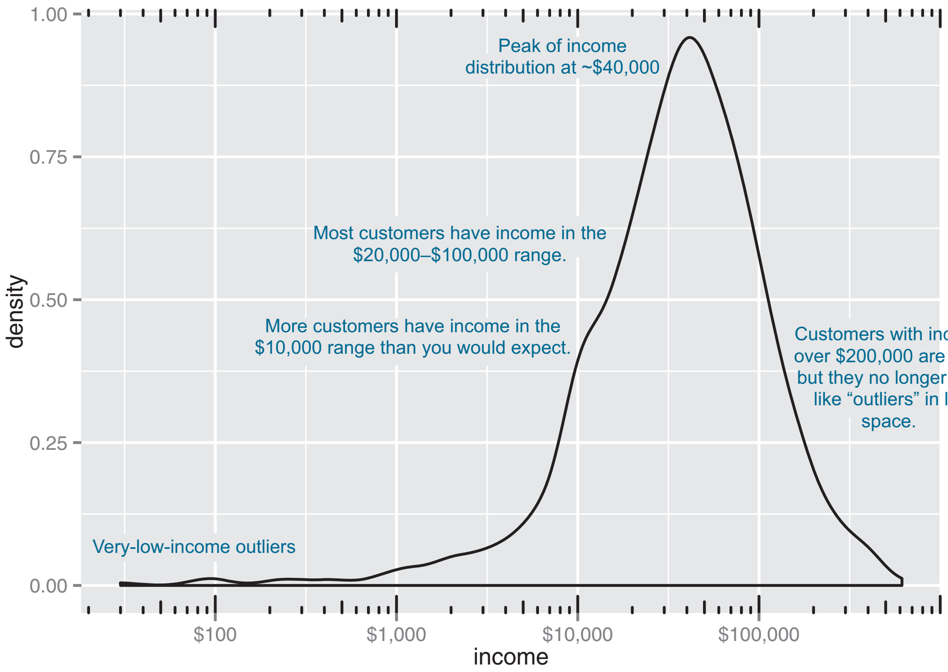


Figure 3.5 The density plot of income on a log10 scale highlights details of the income distribution that are harder to see in a regular density plot.

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

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

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

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) = 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.

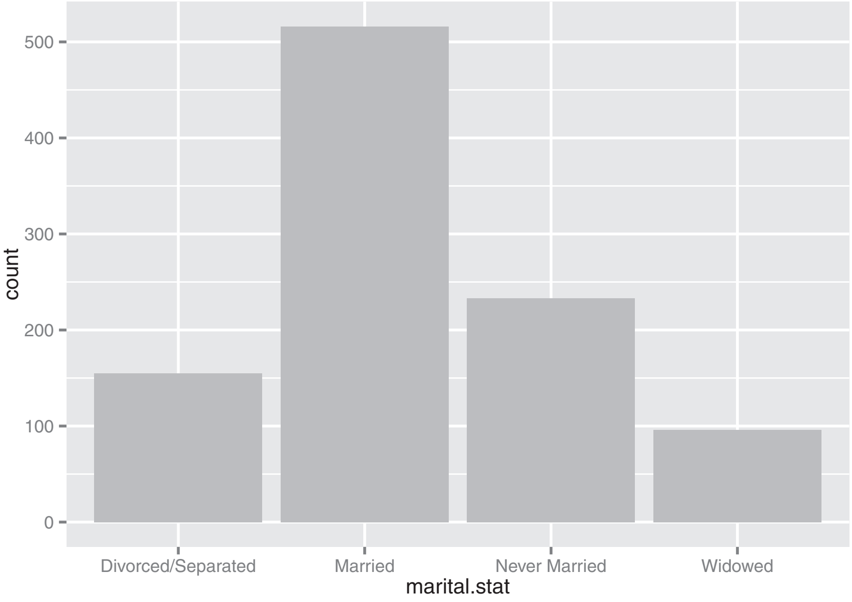


Figure 3.6 Bar charts show the distribution of categorical variables.

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

```
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.

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

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

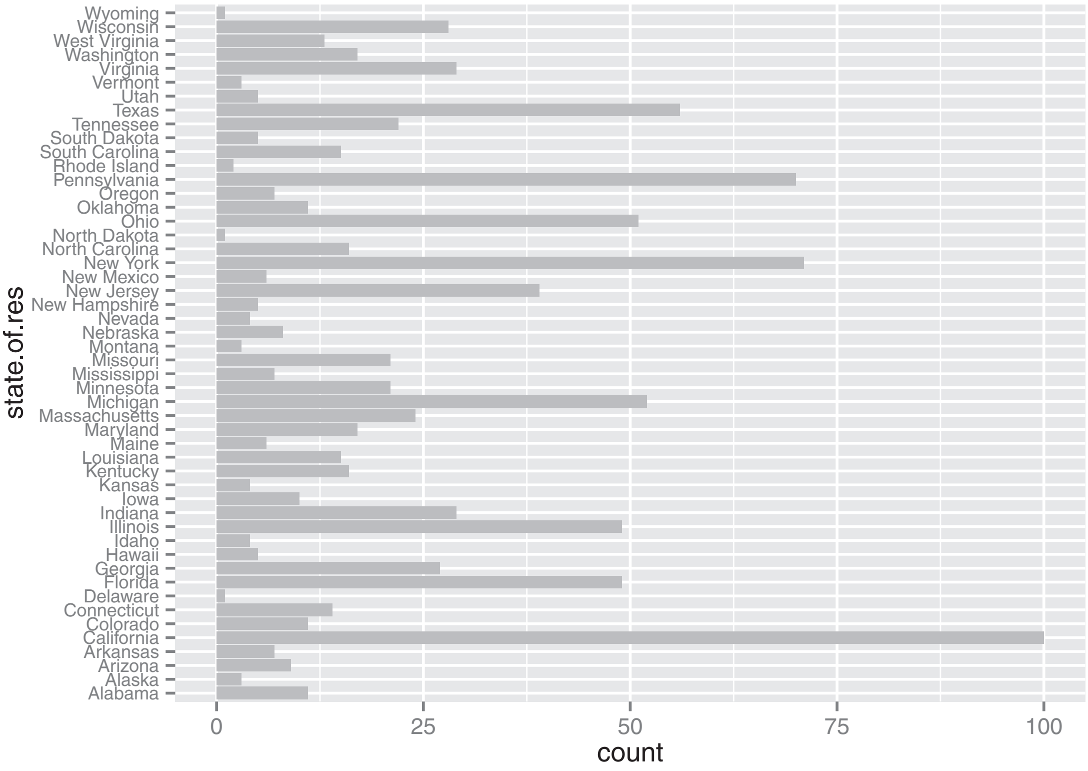


Figure 3.7 A horizontal bar chart can be easier to read when there are several categories with long names.

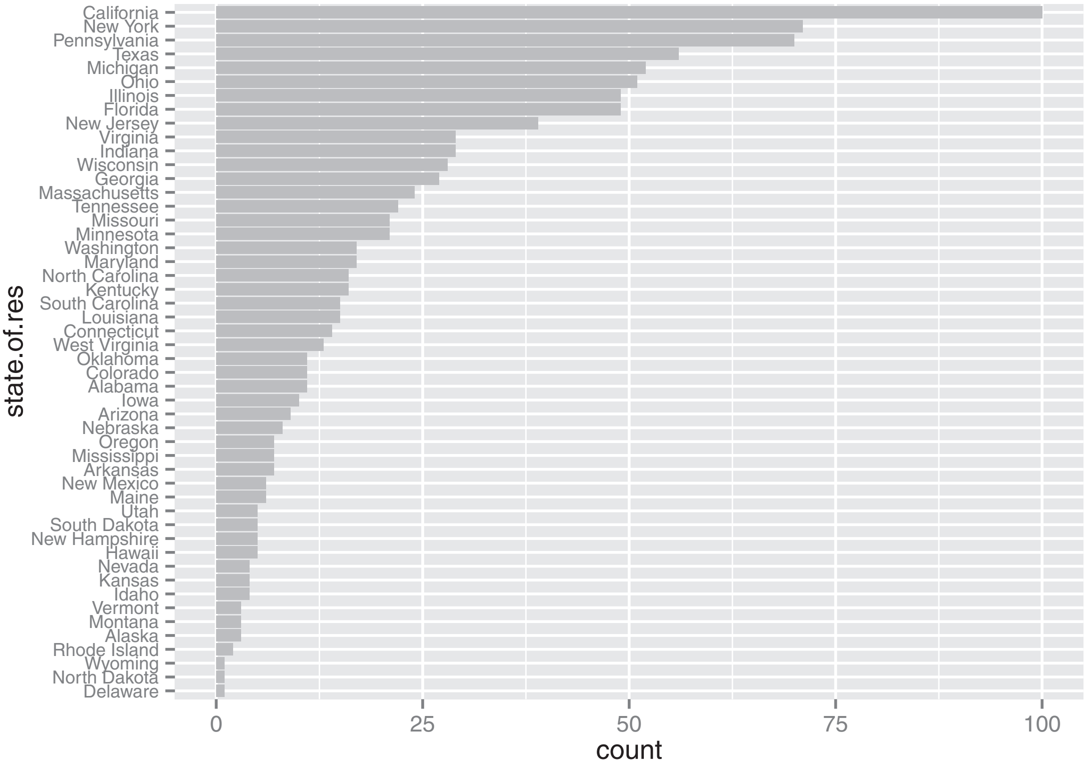


Figure 3.8 Sorting the bar chart by count makes it even easier to read.

Cleveland[3] 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)
```

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

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

Rename the columns for readability.

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

```
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.

Plot the line plot.

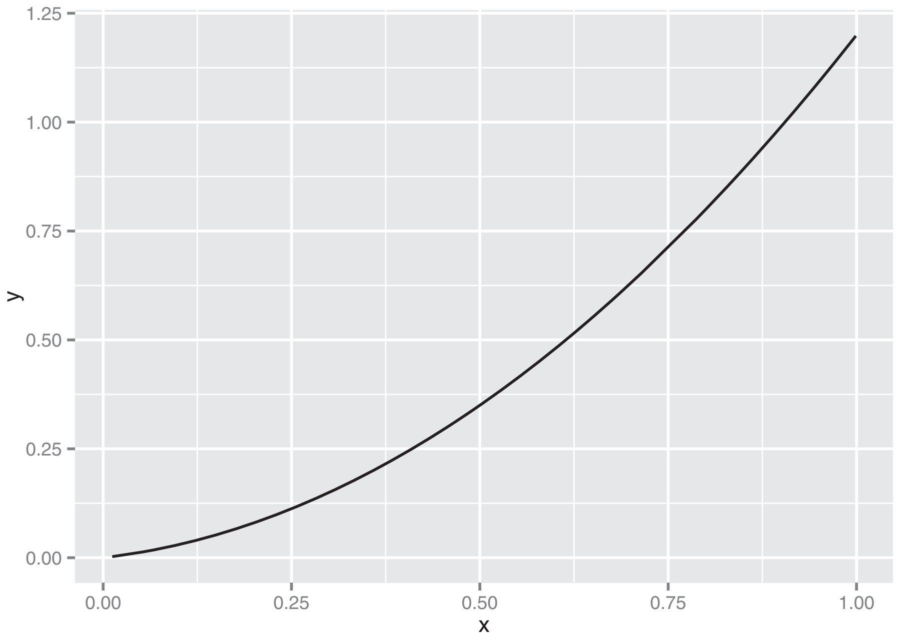


Figure 3.9 Example of a line plot

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))
```

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

```
cor(custdata2$age, custdata2$income)
```

Get correlation of age and income.

```
[1] -0.02240845
```

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)
```

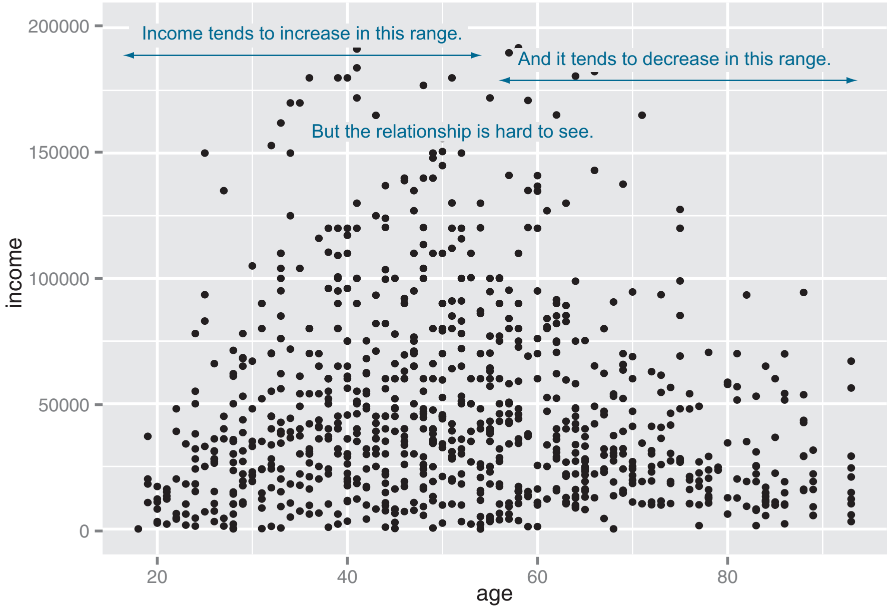


Figure 3.10 A scatter plot of income versus age

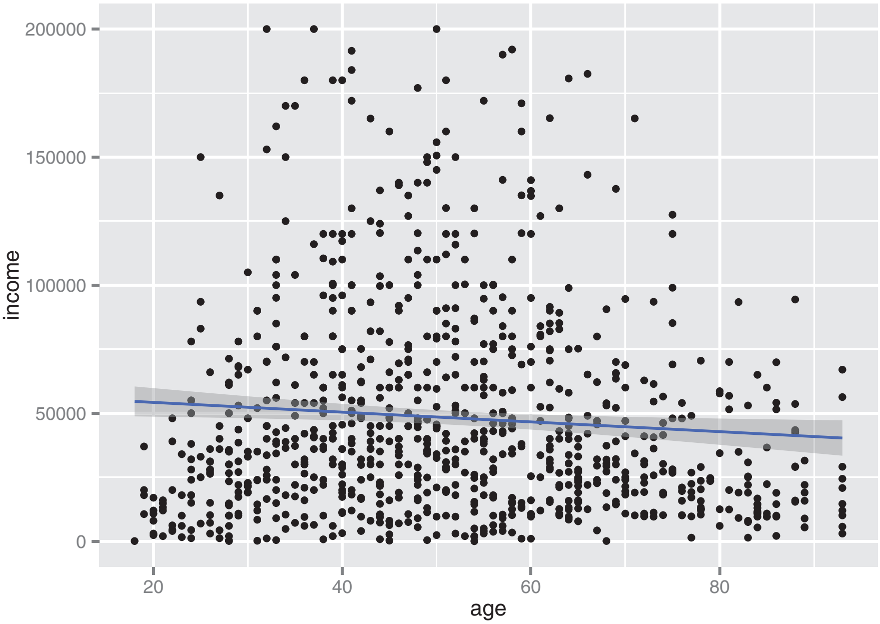


Figure 3.11 A scatter plot of income versus age, with a linear fit

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)
```

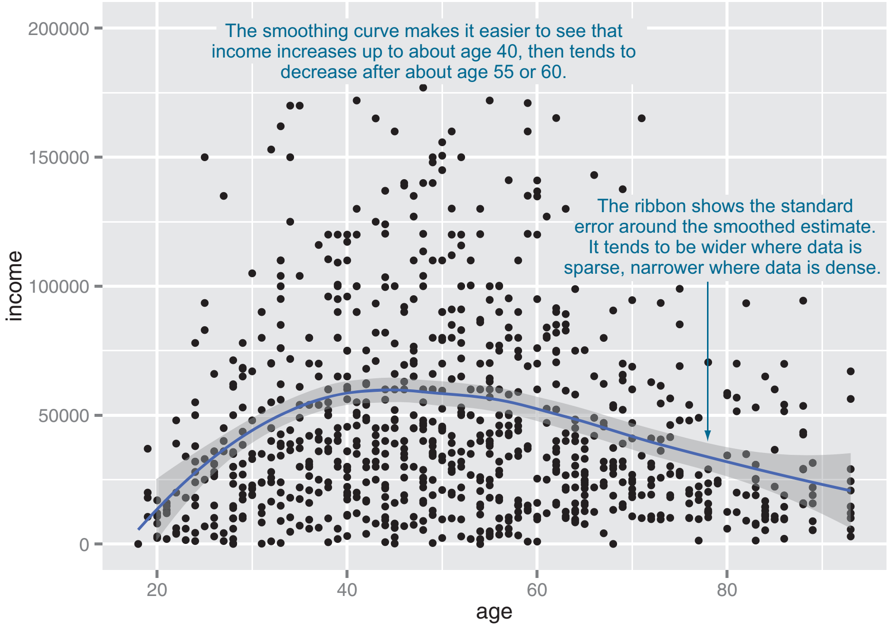
In this case, the linear fit doesn't really capture the shape of the data. You can better capture the shape by instead plotting a smoothing curve through the data, as shown in figure 3.12.

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

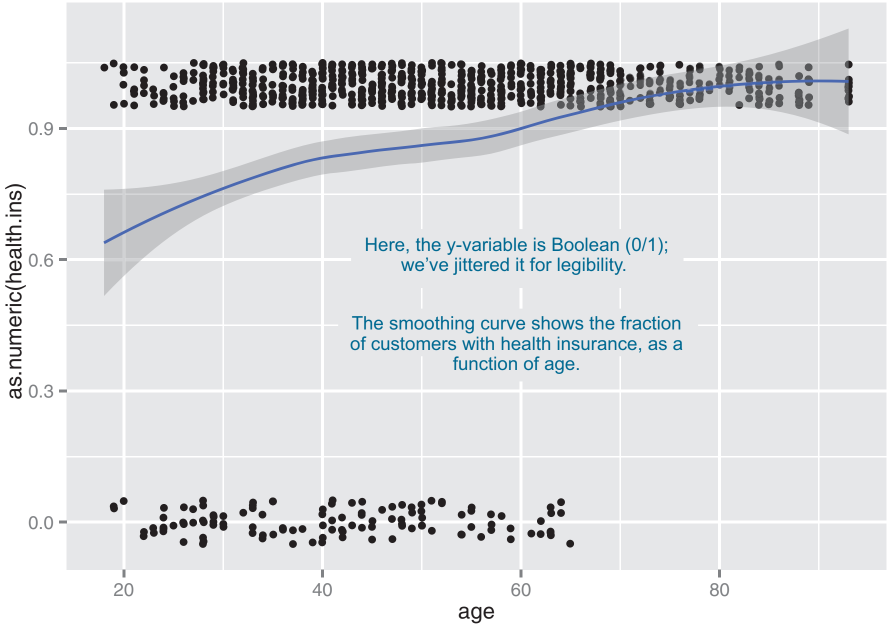
[4] The *stat* layers in `ggplot2` are the layers that perform transformations on the data. They're usually called under the covers by the *geom* layers. Sometimes you have to call them directly, to access parameters that aren't accessible from the geom layers. In this case, the default smoothing curve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



The smoothing curve makes it easier to see that income increases up to about age 40, then tends to decrease after about age 55 or 60.

The ribbon shows the standard error around the smoothed estimate. It tends to be wider where data is sparse, narrower where data is dense.

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



Here, the y-variable is Boolean (0/1); we've jittered it for legibility.

The smoothing curve shows the fraction of customers with health insurance, as a function of age.

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

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.

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

```
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.

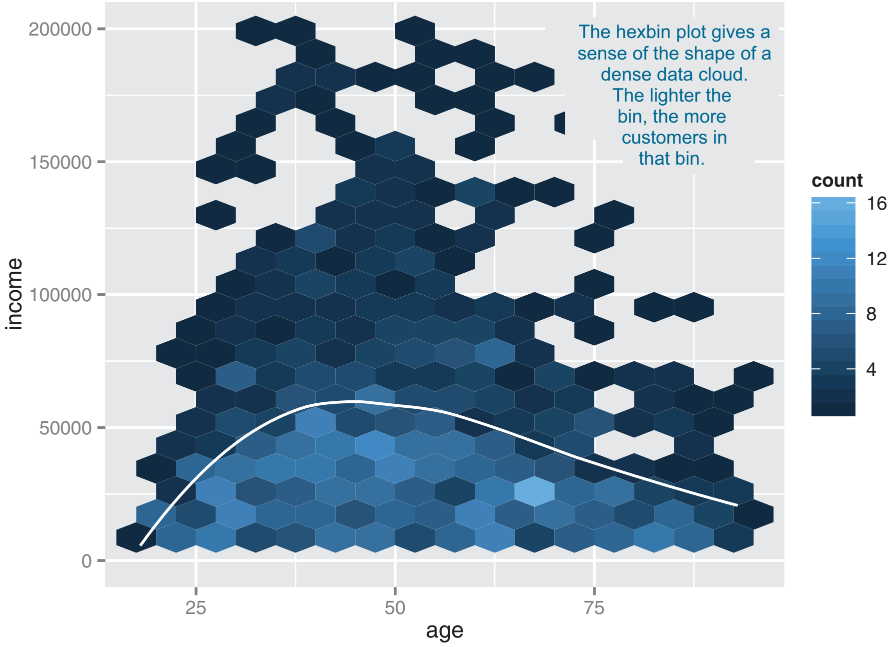
Since y values can only be 0 or 1, add a small jitter to get a sense of data density.

Add smoothing curve.

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.



The hexbin plot gives a sense of the shape of a dense data cloud. The lighter the bin, the more customers in that bin.

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)

ggplot(custdata2, aes(x=age, y=income)) +
   geom_hex(binwidth=c(5, 10000)) +
   geom_smooth(color="white", se=F) +
   ylim(0,200000)
```

Load hexbin library.

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

Add smoothing curve in white; suppress standard error ribbon (se=F).

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.

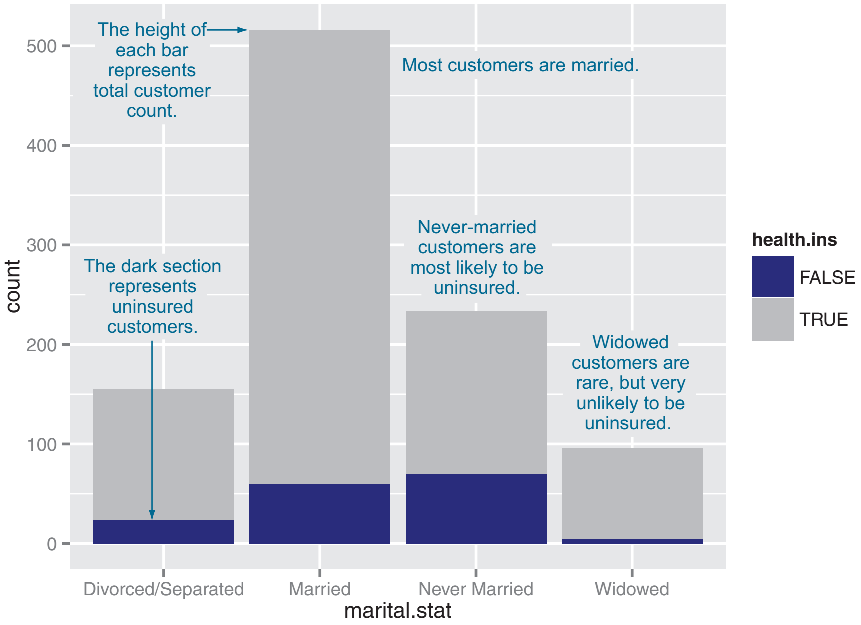


Figure 3.15 Health insurance versus marital status: stacked bar chart

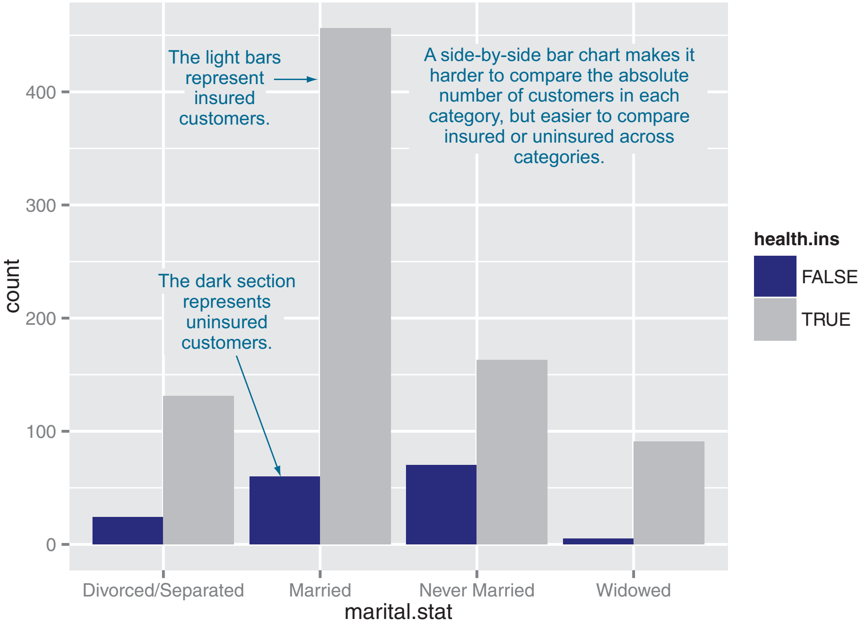


Figure 3.16 Health insurance versus marital status: side-by-side bar chart

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.

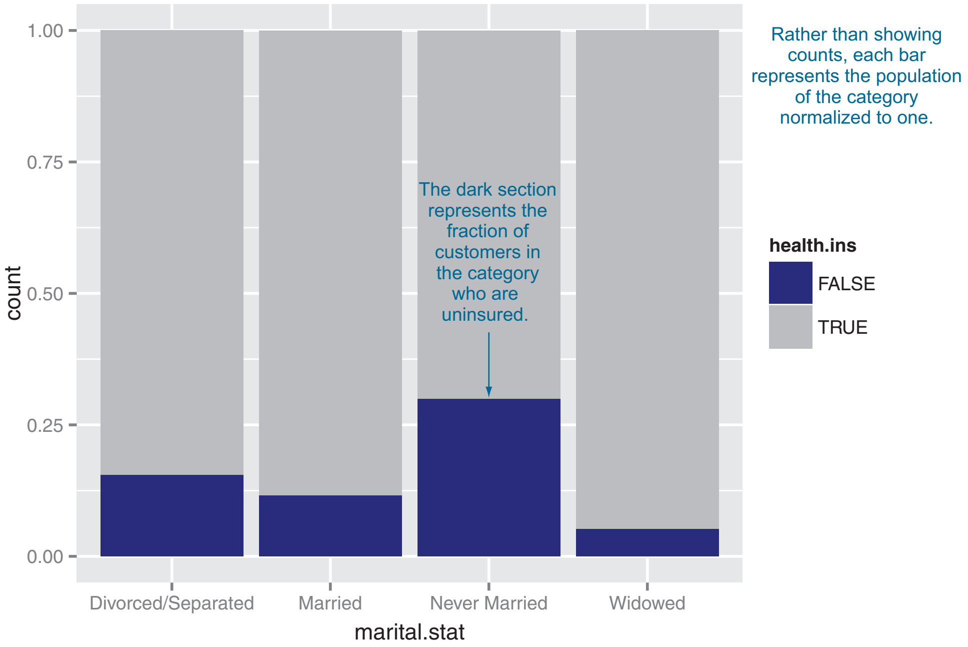
The main shortcoming of both the stacked and side-by-side bar charts is that you can't easily compare the ratios of insured to uninsured across categories, especially for rare categories like *Widowed*. You can use what `ggplot2` calls a *filled bar chart* to plot a visualization of the ratios directly,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



Rather than showing counts, each bar represents the population of the category normalized to one.

The dark section represents the fraction of customers in the category who are uninsured.

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))
```

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

Jitter the points slightly for legibility.

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.

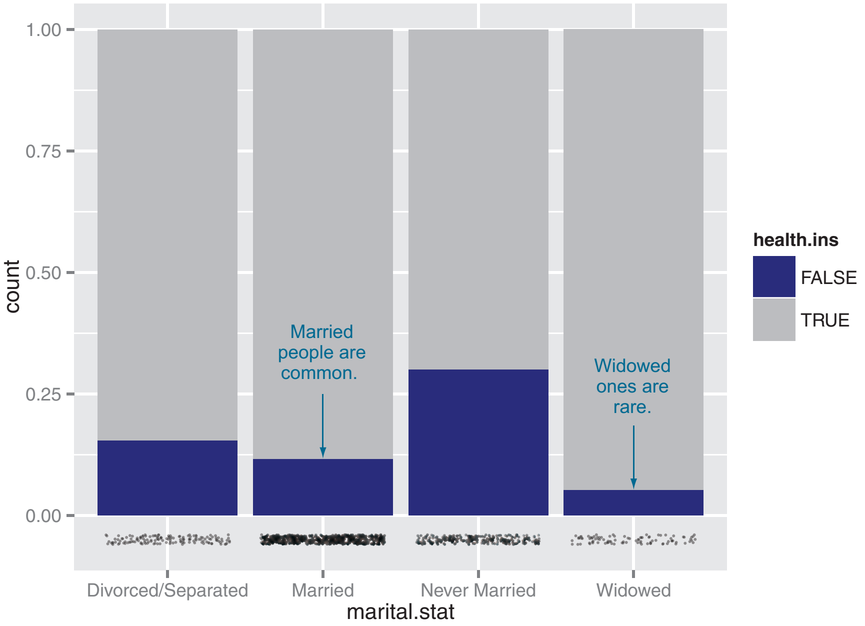


Figure 3.18 Health insurance versus marital status: filled bar chart with rug

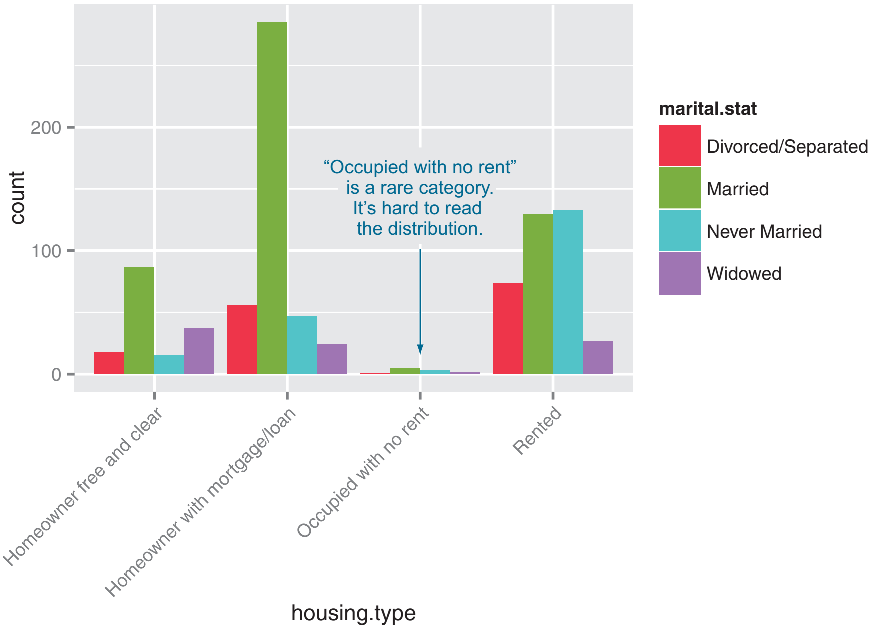
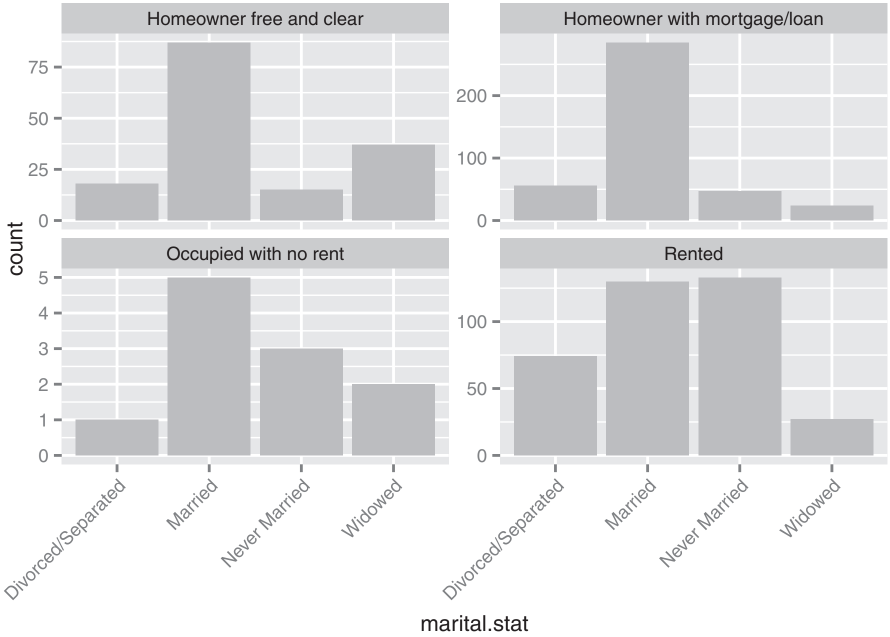


Figure 3.19 Distribution of marital status by housing type: side-by-side bar chart



Note that every facet has a different scale on the y-axis.

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 *true* 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 (`var1` and `var2`). Highlights the frequencies of each value of `var1`.
Side-by-side bar chart	Shows the relationship between two categorical variables (`var1` and `var2`). Good for comparing the frequencies of each value of `var2` across the values of `var1`. Works best when `var2` is binary.
Filled bar chart	Shows the relationship between two categorical variables (`var1` and `var2`). Good for comparing the relative frequencies of each value of `var2` within each value of `var1`. Works best when `var2` is binary.
Bar chart with faceting	Shows the relationship between two categorical variables (`var1` and `var2`). Best for comparing the relative frequencies of each value of `var2` within each value of `var1` when `var2` 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.

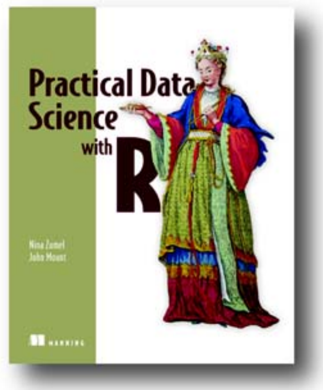
3.3 Summary

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. The data science process is made of loops within loops. The data exploration and data cleaning stages 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.

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.

Time series

Time series are how you organize data when time is an important factor. Examples include forecasting stock prices, modeling the environment, and predicting future product demand. In classic predictive modeling, learning a strong relation between presumed inputs and results, both sampled from the same time, is enough. By contrast, for time series you are asked to forecast one or more quantities for a series of times in the future, based only on measurements from the past. Time series models have a high risk of false fit, so using well-characterized techniques is important. The following chapter demonstrates the most common components of time series prediction using R: smoothing trends, estimating moving averages, identifying seasonal oscillations, and estimating autoregressive relations.

Chapter 15 from *R in Action, Second Edition*
by Robert I. Kabacoff

Time series

This chapter covers

- Creating a time series
- Decomposing a time series into components
- Developing predictive models
- Forecasting future values

How fast is global warming occurring, and what will the impact be in 10 years? With the exception of repeated measures ANOVA in section 9.6, each of the preceding chapters has focused on *cross-sectional* data. In a cross-sectional dataset, variables are measured at a single point in time. In contrast, *longitudinal* data involves measuring variables repeatedly over time. By following a phenomenon over time, it's possible to learn a great deal about it.

In this chapter, we'll examine observations that have been recorded at regularly spaced time intervals for a given span of time. We can arrange observations such as these into a *time series* of the form $Y_1, Y_2, Y_3, \ldots , Y_{t, \ldots,} Y_T$, where Y_t represents the value of Y at time t and T is the total number of observations in the series.

Consider two very different time series displayed in figure 15.1. The series on the left contains the quarterly earnings (dollars) per Johnson & Johnson share between 1960 and 1980. There are 84 observations: one for each quarter over 21

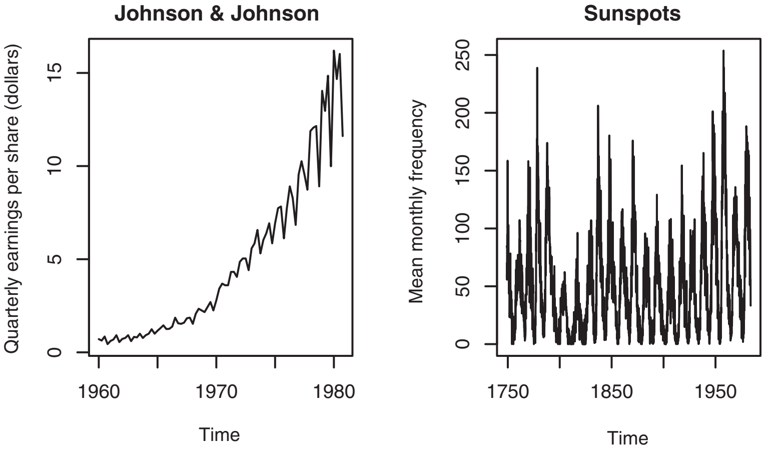


Figure 15.1 Time series plots for (a) Johnson & Johnson quarterly earnings per share (in dollars) from 1960 to 1980, and (b) the monthly mean relative sunspot numbers recorded from 1749 to 1983

years. The series on the right describes the monthly mean relative sunspot numbers from 1749 to 1983 recorded by the Swiss Federal Observatory and the Tokyo Astronomical Observatory. The sunspots time series is much longer, with 2,820 observations—1 per month for 235 years.

Studies of time-series data involve two fundamental questions: what happened (description), and what will happen next (forecasting)? For the Johnson & Johnson data, you might ask

- Is the price of Johnson & Johnson shares changing over time?
- Are there quarterly effects, with share prices rising and falling in a regular fashion throughout the year?
- Can you forecast what future share prices will be and, if so, to what degree of accuracy?

For the sunspot data, you might ask

- What statistical models best describe sunspot activity?
- Do some models fit the data better than others?
- Are the number of sunspots at a given time predictable and, if so, to what degree?

The ability to accurately predict stock prices has relevance for my (hopefully) early retirement to a tropical island, whereas the ability to predict sunspot activity has relevance for my cell phone reception on said island.

Predicting future values of a time series, or *forecasting*, is a fundamental human activity, and studies of time series data have important real-world applications. Economists use time-series data in an attempt to understand and predict what will happen in financial markets. City planners use time-series data to predict future transportation demands. Climate scientists use time-series data to study global climate change. Corporations use time series to predict product demand and future sales. Healthcare officials use time-series data to study the spread of disease and to predict the number of future cases in a given region. Seismologists study times-series data in order to predict earthquakes. In each case, the study of historical time series is an indispensable part of the process. Because different approaches may work best with different types of time series, we'll investigate many examples in this chapter.

There is a wide range of methods for describing time-series data and forecasting future values. If you work with time-series data, you'll find that R has some of the most comprehensive analytic capabilities available anywhere. This chapter explores some of the most common descriptive and forecasting approaches and the R functions used to perform them. The functions are listed in table 15.1 in their order of appearance in the chapter.

Table 15.1 Functions for time-series analysis

Function	Package	Use
`ts()`	`stats`	Creates a time-series object.
`plot()`	`graphics`	Plots a time series.
`start()`	`stats`	Returns the starting time of a time series.
`end()`	`stats`	Returns the ending time of a time series.
`frequency()`	`stats`	Returns the period of a time series.
`window()`	`stats`	Subsets a time-series object.
`ma()`	`forecast`	Fits a simple moving-average model.
`stl()`	`stats`	Decomposes a time series into seasonal, trend, and irregular components using loess.
`monthplot()`	`stats`	Plots the seasonal components of a time series.
`seasonplot()`	`forecast`	Generates a season plot.
`HoltWinters()`	`stats`	Fits an exponential smoothing model.
`forecast()`	`forecast`	Forecasts future values of a time series.
`accuracy()`	`forecast`	Reports fit measures for a time-series model.
`ets()`	`forecast`	Fits an exponential smoothing model. Includes the ability to automate the selection of a model.
`lag()`	`stats`	Returns a lagged version of a time series.
`Acf()`	`forecast`	Estimates the autocorrelation function.
`Pacf()`	`forecast`	Estimates the partial autocorrelation function.
`diff()`	`base`	Returns lagged and iterated differences.

Table 15.1 Functions for time-series analysis

Function	Package	Use
`ndiffs()`	`forecast`	Determines the level of differencing needed to remove trends in a time series.
`adf.test()`	`tseries`	Computes an Augmented Dickey–Fuller test that a time series is stationary.
`arima()`	`stats`	Fits autoregressive integrated moving-average models.
`Box.test()`	`stats`	Computes a Ljung–Box test that the residuals of a time series are independent.
`bds.test()`	`tseries`	Computes the BDS test that a series consists of independent, identically distributed random variables.
`auto.arima()`	`forecast`	Automates the selection of an ARIMA model.

Table 15.2 lists the time-series data that you'll analyze. They're available with the base installation of R. The datasets vary greatly in their characteristics and the models that fit them best.

Table 15.2 Datasets used in this chapter

Time series	Description
`AirPassengers`	Monthly airline passenger numbers from 1949–1960
`JohnsonJohnson`	Quarterly earnings per Johnson & Johnson share
`nhtemp`	Average yearly temperatures in New Haven, Connecticut, from 1912–1971
`Nile`	Flow of the river Nile
`sunspots`	Monthly sunspot numbers from 1749–1983

We'll start with methods for creating and manipulating time series, describing and plotting them, and decomposing them into level, trend, seasonal, and irregular (error) components. Then we'll turn to forecasting, starting with popular exponential modeling approaches that use weighted averages of time-series values to predict future values. Next we'll consider a set of forecasting techniques called *autoregressive integrated moving averages (ARIMA) models* that use correlations among recent data points and among recent prediction errors to make future forecasts. Throughout, we'll consider methods of evaluating the fit of models and the accuracy of their predictions. The chapter ends with a description of resources available for learning more about these topics.

15.1 Creating a time-series object in R

In order to work with a time series in R, you have to place it into a *time-series object*—an R structure that contains the observations, the starting and ending time of the series, and information about its periodicity (for example, monthly, quarterly, or annual data). Once the data are in a time-series object, you can use numerous functions to manipulate, model, and plot the data.

A vector of numbers, or a column in a data frame, can be saved as a time-series object using the `ts()` function. The format is

```
myseries <-  ts(data, start=, end=, frequency=)
```

where *`myseries`* is the time-series object, *`data`* is a numeric vector containing the observations, `start` specifies the series start time, `end` specifies the end time (optional), and `frequency` indicates the number of observations per unit time (for example, `frequency=1` for annual data, `frequency=12` for monthly data, and `frequency=4` for quarterly data).

An example is given in the following listing. The data consist of monthly sales figures for two years, starting in January 2003.

Listing 15.1 Creating a time-series object

```
> sales <- c(18, 33, 41,  7, 34, 35, 24, 25, 24, 21, 25, 20,
             22, 31, 40, 29, 25, 21, 22, 54, 31, 25, 26, 35)

> tsales <- ts(sales, start=c(2003, 1), frequency=12)
> tsales

     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2003  18  33  41   7  34  35  24  25  24  21  25  20
2004  22  31  40  29  25  21  22  54  31  25  26  35

> plot(tsales)
> start(tsales)

[1] 2003    1

> end(tsales)

[1] 2004   12

> frequency(tsales)

[1] 12

> tsales.subset <- window(tsales, start=c(2003, 5), end=c(2004, 6))
> tsales.subset

     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2003                  34  35  24  25  24  21  25  20
2004  22  31  40  29  25  21
```

❶ **Creates a time-series object**

❷ **Gets information about the object**

Subsets the object ❸

In this listing, the `ts()` function is used to create the time-series object ❶. Once it's created, you can print and plot it; the plot is given in figure 15.2. You can modify the plot using the techniques described in chapter 3. For example, `plot(tsales, type="o", pch=19)` would create a time-series plot with connected, solid-filled circles.

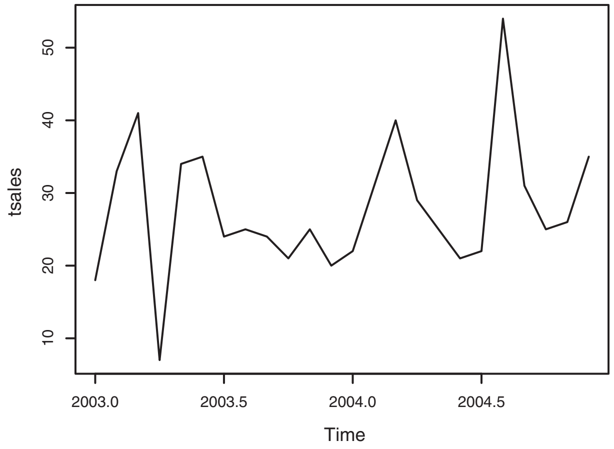


Figure 15.2 Time-series plot for the sales data in listing 15.1. The decimal notation on the time dimension is used to represent the portion of a year. For example, 2003.5 represents July 1 (halfway through 2003).

Once you've created the time-series object, you can use functions like `start()`, `end()`, and `frequency()` to return its properties ❷. You can also use the `window()` function to create a new time series that's a subset of the original ❸.

15.2 Smoothing and seasonal decomposition

Just as analysts explore a dataset with descriptive statistics and graphs before attempting to model the data, describing a time series numerically and visually should be the first step before attempting to build complex models. In this section, we'll look at smoothing a time series to clarify its general trend, and decomposing a time series in order to observe any seasonal effects.

15.2.1 Smoothing with simple moving averages

The first step when investigating a time series is to plot it, as in listing 15.1. Consider the `Nile` time series. It records the annual flow of the river Nile at Ashwan from 1871–1970. A plot of the series can be seen in the upper-left panel of figure 15.3. The time series appears to be decreasing, but there is a great deal of variation from year to year.

Time series typically have a significant irregular or error component. In order to discern any patterns in the data, you'll frequently want to plot a smoothed curve that damps down these fluctuations. One of the simplest methods of smoothing a time series is to use simple moving averages. For example, each data point can be replaced with the mean of that observation and one observation before and after it. This is called a *centered moving average*. A centered moving average is defined as

$$S_t = (Y_{t-q} + \ldots + Y_t + \ldots + Y_{t+q}) \;/\; (2q + 1)$$

where S_t is the smoothed value at time t and k = 2q + 1 is the number of observations that are averaged. The k value is usually chosen to be an odd number (3 in this example). By necessity, when using a centered moving average, you lose the (k – 1) / 2 observations at each end of the series.

Several functions in R can provide a simple moving average, including SMA() in the TTR package, rollmean() in the zoo package, and ma() in the forecast package. Here, you'll use the ma() function to smooth the Nile time series that comes with the base R installation.

The code in the next listing plots the raw time series and smoothed versions using k equal to 3, 7, and 15. The plots are given in figure 15.3.

Listing 15.2 Simple moving averages

```
library(forecast)
opar <- par(no.readonly=TRUE)
par(mfrow=c(2,2))
ylim <- c(min(Nile), max(Nile))
plot(Nile, main="Raw time series")
plot(ma(Nile, 3), main="Simple Moving Averages (k=3)", ylim=ylim)
plot(ma(Nile, 7), main="Simple Moving Averages (k=7)", ylim=ylim)
plot(ma(Nile, 15), main="Simple Moving Averages (k=15)", ylim=ylim)
par(opar)
```

As k increases, the plot becomes increasingly smoothed. The challenge is to find the value of k that highlights the major patterns in the data, without under- or oversmoothing. This is more art than science, and you'll probably want to try several values of k before settling on one. From the plots in figure 15.3, there certainly appears to have been a drop in river flow between 1892 and 1900. Other changes are open to interpretation. For example, there may have been a small increasing trend between 1941 and 1961, but this could also have been a random variation.

For time-series data with a periodicity greater than one (that is, with a seasonal component), you'll want to go beyond a description of the overall trend. Seasonal decomposition can be used to examine both seasonal and general trends.

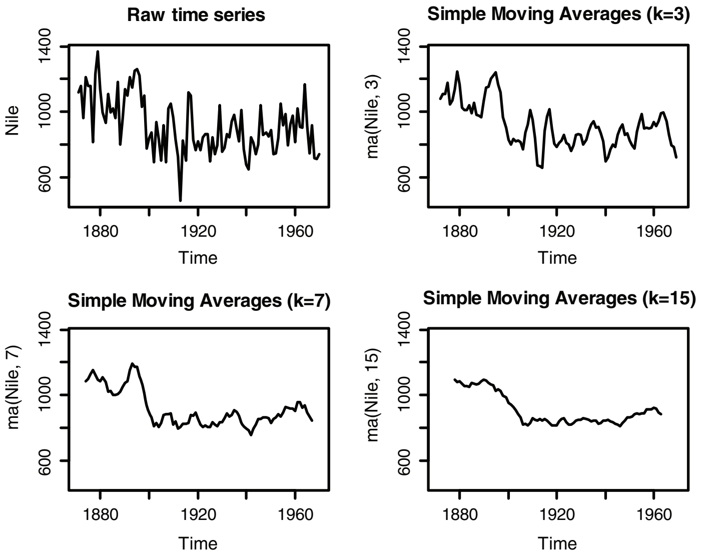


Figure 15.3 The Nile time series measuring annual river flow at Ashwan from 1871–1970 (upper left). The other plots are smoothed versions using simple moving averages at three smoothing levels (k=3, 7, and 15).

15.2.2 Seasonal decomposition

Time-series data that have a seasonal aspect (such as monthly or quarterly data) can be decomposed into a trend component, a seasonal component, and an irregular component. The *trend component* captures changes in level over time. The *seasonal component* captures cyclical effects due to the time of year. The *irregular* (or *error*) *component* captures those influences not described by the trend and seasonal effects.

The decomposition can be additive or multiplicative. In an additive model, the components sum to give the values of the time series. Specifically,

$$Y_t = \text{Trend}_t + \text{Seasonal}_t + \text{Irregular}_t$$

where the observation at time t is the sum of the contributions of the trend at time t, the seasonal effect at time t, and an irregular effect at time t.

In a multiplicative model, given by the equation

$$Y_t = \text{Trend}_t * \text{Seasonal}_t * \text{Irregular}_t$$

the trend, seasonal, and irregular influences are multiplied. Examples are given in figure 15.4.

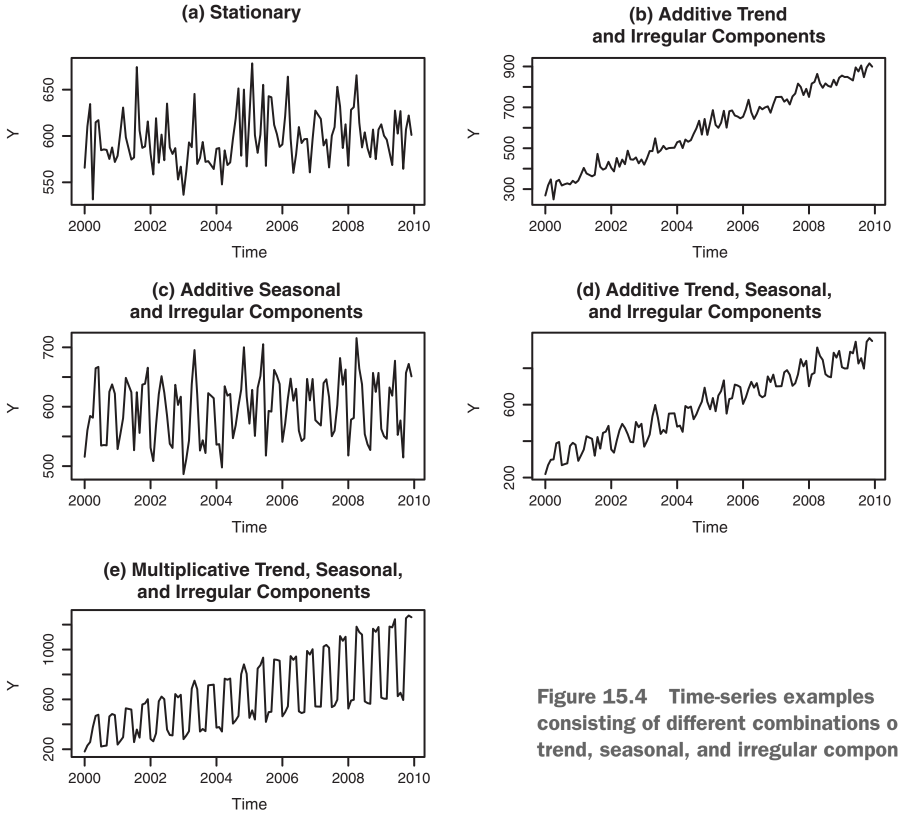


Figure 15.4 Time-series examples consisting of different combinations of trend, seasonal, and irregular components

In the first plot (a), there is neither a trend nor a seasonal component. The only influence is a random fluctuation around a given level. In the second plot (b), there is an upward trend over time, as well as random fluctuations. In the third plot (c), there are seasonal effects and random fluctuations, but no overall trend away from a horizontal line. In the fourth plot (d), all three components are present: an upward trend, seasonal effects, and random fluctuations. You also see all three components in the final plot (e), but here they combine in a multiplicative way. Notice how the variability is proportional to the level: as the level increases, so does the variability. This amplification (or possible damping) based on the current level of the series strongly suggests a multiplicative model.

An example may make the difference between additive and multiplicative models clearer. Consider a time series that records the monthly sales of motorcycles over a 10-year period. In a model with an additive seasonal effect, the number of motorcycles sold tends to increase by 500 in November and December (due to the Christmas rush) and decrease by 200 in January (when sales tend to be down). The seasonal increase or decrease is independent of the current sales volume.

In a model with a multiplicative seasonal effect, motorcycle sales in November and December tend to increase by 20% and decrease in January by 10%. In the multiplicative case, the impact of the seasonal effect is proportional to the current sales volume. This isn't the case in an additive model. In many instances, the multiplicative model is more realistic.

A popular method for decomposing a time series into trend, seasonal, and irregular components is seasonal decomposition by loess smoothing. In R, this can be accomplished with the `stl()` function. The format is

```
stl(ts, s.window=, t.window=)
```

where *ts* is the time series to be decomposed, `s.window` controls how fast the seasonal effects can change over time, and `t.window` controls how fast the trend can change over time. Smaller values allow more rapid change. Setting `s.window="periodic"` forces seasonal effects to be identical across years. Only the *ts* and `s.window` parameters are required. See `help(stl)` for details.

The `stl()` function can only handle additive models, but this isn't a serious limitation. Multiplicative models can be transformed into additive models using a log transformation:

$$\log(Y_t) = \log(\text{Trend}_t * \text{Seasonal}_t * \text{Irregular}_t)$$
$$= \log(\text{Trend}_t) + \log(\text{Seasonal}_t) + \log(\text{Irregular}_t)$$

After fitting the additive model to the log transformed series, the results can be back-transformed to the original scale. Let's look at an example.

The time series `AirPassengers` comes with a base R installation and describes the monthly totals (in thousands) of international airline passengers between 1949 and 1960. A plot of the data is given in the top of figure 15.5. From the graph, it appears that variability of the series increases with the level, suggesting a multiplicative model.

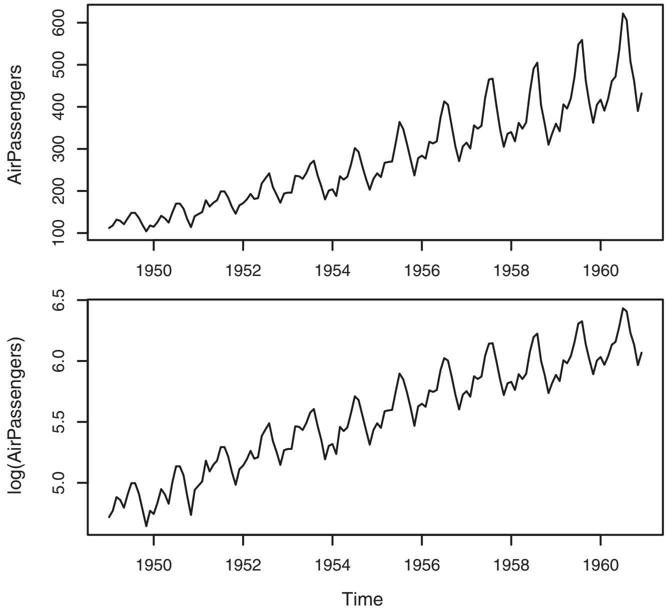


Figure 15.5 Plot of the `AirPassengers` time series (top). The time series contains the monthly totals (in thousands) of international airline passengers between 1949 and 1960. The log-transformed time series (bottom) stabilizes the variance and fits an additive seasonal decomposition model better.

The plot in the lower portion of figure 15.5 displays the time series created by taking the log of each observation. The variance has stabilized, and the logged series looks like an appropriate candidate for an additive decomposition. This is carried out using the `stl()` function in the following listing.

Listing 15.3 Seasonal decomposition using `stl()`

```
> plot(AirPassengers)
> lAirPassengers <- log(AirPassengers)
> plot(lAirPassengers, ylab="log(AirPassengers)")

> fit <- stl(lAirPassengers, s.window="period")
> plot(fit)

> fit$time.series

          seasonal trend  remainder
Jan 1949 -0.09164 4.829 -0.0192494
Feb 1949 -0.11403 4.830  0.0543448
Mar 1949  0.01587 4.831  0.0355884
Apr 1949 -0.01403 4.833  0.0404633
May 1949 -0.01502 4.835 -0.0245905
Jun 1949  0.10979 4.838 -0.0426814
Jul 1949  0.21640 4.841 -0.0601152
Aug 1949  0.20961 4.843 -0.0558625
Sep 1949  0.06747 4.846 -0.0008274
Oct 1949 -0.07025 4.851 -0.0015113
Nov 1949 -0.21353 4.856  0.0021631
```

1 Plots the time series

2 Decomposes the time series

3 Components for each observation

```
Dec 1949 -0.10064 4.865  0.0067347
... output omitted ...

> exp(fit$time.series)

         seasonal trend remainder
Jan 1949   0.9124 125.1    0.9809
Feb 1949   0.8922 125.3    1.0558
Mar 1949   1.0160 125.4    1.0362
Apr 1949   0.9861 125.6    1.0413
May 1949   0.9851 125.9    0.9757
Jun 1949   1.1160 126.2    0.9582
Jul 1949   1.2416 126.6    0.9417
Aug 1949   1.2332 126.9    0.9457
Sep 1949   1.0698 127.2    0.9992
Oct 1949   0.9322 127.9    0.9985
Nov 1949   0.8077 128.5    1.0022
Dec 1949   0.9043 129.6    1.0068
... output omitted ...
```

First, the time series is plotted and transformed ❶. A seasonal decomposition is performed and saved in an object called fit ❷. Plotting the results gives the graph in figure 15.6. The graph shows the time series, seasonal, trend, and irregular components from 1949 to 1960. Note that the seasonal components have been constrained to

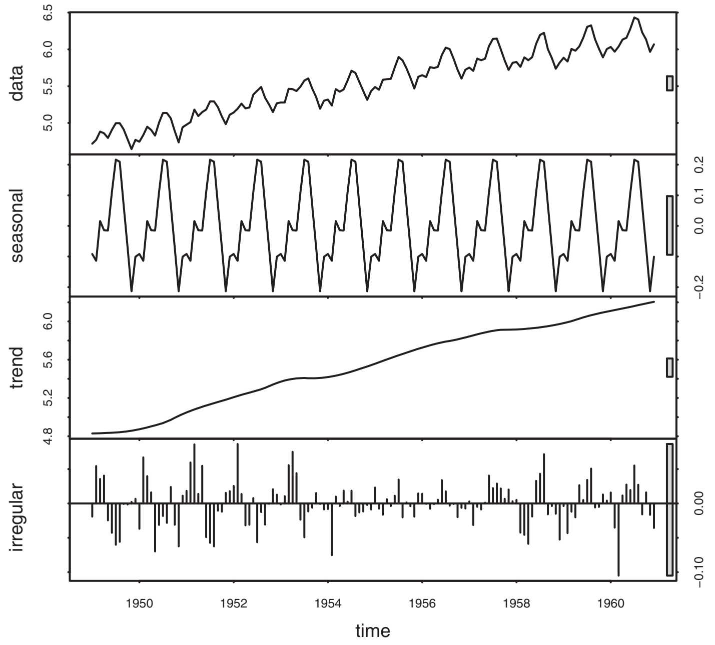


Figure 15.6 A seasonal decomposition of the logged `AirPassengers` time series using the `stl()` function. The time series (data) is decomposed into seasonal, trend, and irregular components.

remain the same across each year (using the `s.window="period"` option). The trend is monotonically increasing, and the seasonal effect suggests more passengers in the summer (perhaps during vacations). The grey bars on the right are magnitude guides—each bar represents the same magnitude. This is useful because the y-axes are different for each graph.

The object returned by the `stl()` function contains a component called `time.series` that contains the trend, season, and irregular portion of each observation ❸. In this case, `fit$time.series` is based on the logged time series. `exp(fit$time.series)` converts the decomposition back to the original metric. Examining the seasonal effects suggests that the number of passengers increased by 24% in July (a multiplier of 1.24) and decreased by 20% in November (with a multiplier of .80).

Two additional graphs can help to visualize a seasonal decomposition. They're created by the `monthplot()` function that comes with base R and the `seasonplot()` function provided in the `forecast` package. The code

```
par(mfrow=c(2,1))
library(forecast)
monthplot(AirPassengers, xlab="",  ylab="")
seasonplot(AirPassengers, year.labels="TRUE", main="")
```

produces the graphs in figure 15.7.

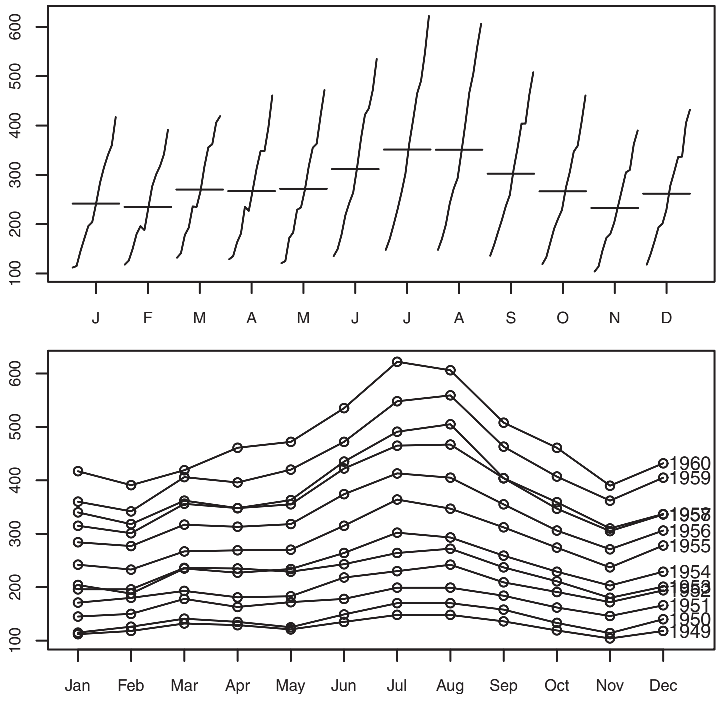


Figure 15.7 A month plot (top) and season plot (bottom) for the `AirPassengers` time series. Each shows an increasing trend and similar seasonal pattern year to year.

The month plot (top figure) displays the subseries for each month (all January values connected, all February values connected, and so on), along with the average of each subseries. From this graph, it appears that the trend is increasing for each month in a roughly uniform way. Additionally, the greatest number of passengers occurs in July and August. The season plot (lower figure) displays the subseries by year. Again you see a similar pattern, with increases in passengers each year, and the same seasonal pattern.

Note that although you've described the time series, you haven't predicted any future values. In the next section, we'll consider the use of exponential models for forecasting beyond the available data.

15.3 Exponential forecasting models

Exponential models are some of the most popular approaches to forecasting the future values of a time series. They're simpler than many other types of models, but they can yield good short-term predictions in a wide range of applications. They differ from each other in the components of the time series that are modeled. A simple exponential model (also called a *single exponential model*) fits a time series that has a constant level and an irregular component at time *i* but has neither a trend nor a seasonal component. A *double exponential model* (also called a *Holt exponential smoothing*) fits a time series with both a level and a trend. Finally, a *triple exponential model* (also called a *Holt-Winters exponential smoothing*) fits a time series with level, trend, and seasonal components.

Exponential models can be fit with either the `HoltWinters()` function in the base installation or the `ets()` function that comes with the `forecast` package. The `ets()` function has more options and is generally more powerful. We'll focus on the `ets()` function in this section.

The format of the `ets()` function is

```
ets(ts, model="ZZZ")
```

where *ts* is a time series and the model is specified by three letters. The first letter denotes the error type, the second letter denotes the trend type, and the third letter denotes the seasonal type. Allowable letters are *A* for additive, *M* for multiplicative, *N* for none, and *Z* for automatically selected. Examples of common models are given in table 15.3.

Table 15.3 Functions for fitting simple, double, and triple exponential forecasting models

Type	Parameters fit	Functions

`simple`	`level`	`ets(ts, model="ANN")`
		`ses(ts)`

`double`	`level, slope`	`ets(ts, model="AAN")`
		`holt(ts)`

`triple`	`level, slope, seasonal`	`ets(ts, model="AAA")`
		`hw(ts)`

The ses(), holt(), and hw() functions are convenience wrappers to the ets() function with prespecified defaults.

First we'll look at the most basic exponential model: simple exponential smoothing. Be sure to install the forecast package (install.packages("forecast")) before proceeding.

15.3.1 Simple exponential smoothing

Simple exponential smoothing uses a weighted average of existing time-series values to make a short-term prediction of future values. The weights are chosen so that observations have an exponentially decreasing impact on the average as you go back in time.

The simple exponential smoothing model assumes that an observation in the time series can be described by

$$\mathrm{Y}_{\mathrm{t}} = \text{level} + \text{irregular}_{t}$$

The prediction at time $\mathrm{Y}_{\mathrm{t+1}}$ (called the *1-step ahead forecast*) is written as

$$Y_{\mathrm{t+1}} = c_0 Y_{\mathrm{t}} + c_1 Y_{\mathrm{t-1}} + c_2 Y_{\mathrm{t-2}} + c_2 Y_{\mathrm{t-2}} + \ldots$$

where $c_{\mathrm{i}} = \alpha(1-\alpha)^{\mathrm{i}}$, i = 0, 1, 2, ... and $0 \leq \alpha \leq 1$. The c_{i} weights sum to one, and the 1-step ahead forecast can be seen to be a weighted average of the current value and all past values of the time series. The alpha (α) parameter controls the rate of decay for the weights. The closer alpha is to 1, the more weight is given to recent observations. The closer alpha is to 0, the more weight is given to past observations. The actual value of alpha is usually chosen by computer in order to optimize a fit criterion. A common fit criterion is the sum of squared errors between the actual and predicted values. An example will help clarify these ideas.

The nhtemp time series contains the mean annual temperature in degrees Fahrenheit in New Haven, Connecticut, from 1912 to 1971. A plot of the time series can be seen as the line in figure 15.8.

There is no obvious trend, and the yearly data lack a seasonal component, so the simple exponential model is a reasonable place to start. The code for making a 1-step ahead forecast using the ses() function is given next.

Listing 15.4 Simple exponential smoothing

```
> library(forecast)
> fit <- ets(nhtemp, model="ANN")          ❶ Fits the model
> fit

ETS(A,N,N)

Call:
 ets(y = nhtemp, model = "ANN")

  Smoothing parameters:
    alpha = 0.182

  Initial states:
    l = 50.2759
  sigma:  1.126

  AIC  AICc   BIC
263.9 264.1 268.1
                                                ❷ I-step ahead forecast
> forecast(fit, 1)                          ◁

     Point Forecast Lo 80 Hi 80 Lo 95 Hi 95
1972          51.87 50.43 53.31 49.66 54.08

> plot(forecast(fit, 1), xlab="Year",
    ylab=expression(paste("Temperature (", degree*F,")",)),
    main="New Haven Annual Mean Temperature")

> accuracy(fit)                                                 ◁
                                                               ❸ Prints accuracy measures
                 ME  RMSE    MAE    MPE  MAPE   MASE
Training set 0.146 1.126 0.8951 0.2419 1.749 0.9228
```

The `ets(mode="ANN")` statement fits the simple exponential model to the `nhtemp` time series ❶. The `A` indicates that the errors are additive, and the `NN` indicates that there is no trend and no seasonal component. The relatively low value of alpha (0.18) indicates that distant as well as recent observations are being considered in the forecast. This value is automatically chosen to maximize the fit of the model to the given dataset.

The `forecast()` function is used to predict the time series *k* steps into the future. The format is `forecast(`*`fit, k`*`)`. The 1-step ahead forecast for this series is 51.9°F with a 95% confidence interval (49.7°F to 54.1°F) ❷. The time series, the forecasted value, and the 80% and 95% confidence intervals are plotted in figure 15.8 ❸.

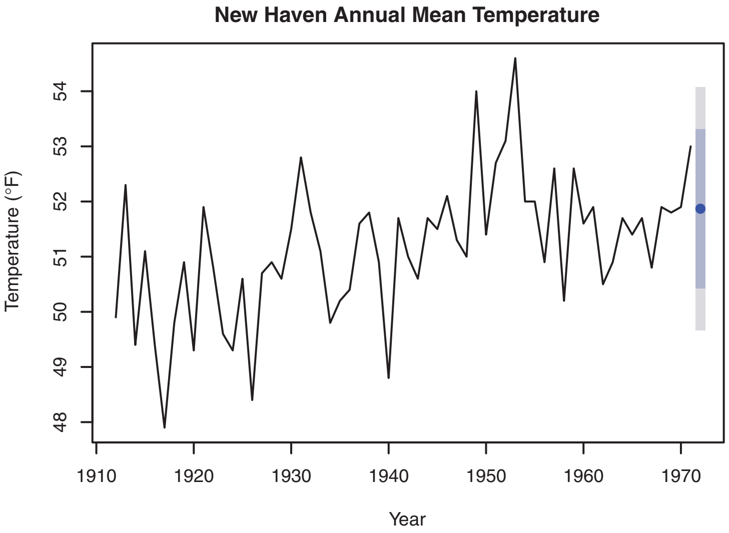


Figure 15.8 Average yearly temperatures in New Haven, Connecticut; and a 1-step ahead prediction from a simple exponential forecast using the `ets()` function

The `forecast` package also provides an `accuracy()` function that displays the most popular predictive accuracy measures for time-series forecasts ❸. A description of each is given in table 15.4. The e_t represent the error or irregular component of each observation $(Y_{t-} \hat{Y}_i)$.

Table 15.4 Predictive accuracy measures

Measure	Abbreviation	Definition
Mean error	ME	mean(e_t)
Root mean squared error	RMSE	sqrt(mean(e_t^2))
Mean absolute error	MAE	mean(| e_t |)
Mean percentage error	MPE	mean(100 * e_t / Y_t)
Mean absolute percentage error	MAPE	mean(| 100 * e_t / Y_t |)
Mean absolute scaled error	MASE	mean(| q_t |) where q_t = e_t / (1/(T-1) * sum(| $y_{t-}y_{t-1}$|)), T is the number of observations, and the sum goes from t=2 to t=T

The mean error and mean percentage error may not be that useful, because positive and negative errors can cancel out. The RMSE gives the square root of the mean square error, which in this case is 1.13°F. The mean absolute percentage error reports the error as a percentage of the time-series values. It's unit-less and can be used to compare prediction accuracy across time series. But it assumes a measurement scale with a true zero point (for example, number of passengers per day). Because the Fahrenheit scale has no true zero, you can't use it here. The mean absolute scaled error is the most recent accuracy measure and is used to compare the forecast accuracy across time series on different scales. There is no one best measure of predictive accuracy. The RMSE is certainly the best known and often cited.

Simple exponential smoothing assumes the absence of trend or seasonal components. The next section considers exponential models that can accommodate both.

15.3.2 Holt and Holt-Winters exponential smoothing

The Holt exponential smoothing approach can fit a time series that has an overall level and a trend (slope). The model for an observation at time *t* is

$$Y_t = \text{level} + \text{slope}*\text{t} + \text{irregular}_t$$

An alpha smoothing parameter controls the exponential decay for the level, and a beta smoothing parameter controls the exponential decay for the slope. Again, each parameter ranges from 0 to 1, with larger values giving more weight to recent observations.

The Holt-Winters exponential smoothing approach can be used to fit a time series that has an overall level, a trend, and a seasonal component. Here, the model is

$$Y_t = \text{level} + \text{slope}*\text{t} + s_t + \text{irregular}_t$$

where s_t represents the seasonal influence at time *t*. In addition to alpha and beta parameters, a gamma smoothing parameter controls the exponential decay of the seasonal component. Like the others, it ranges from 0 to 1, and larger values give more weight to recent observations in calculating the seasonal effect.

In section 15.2, you decomposed a time series describing the monthly totals (in log thousands) of international airline passengers into additive trend, seasonal, and irregular components. Let's use an exponential model to predict future travel. Again, you'll use log values so that an additive model fits the data. The code in the following listing applies the Holt-Winters exponential smoothing approach to predicting the next five values of the `AirPassengers` time series.

Listing 15.5 Exponential smoothing with level, slope, and seasonal components

```
> library(forecast)
> fit <- ets(log(AirPassengers), model="AAA")
> fit

ETS(A,A,A)

Call:
 ets(y = log(AirPassengers), model = "AAA")

  Smoothing parameters:
    alpha = 0.8528
    beta  = 4e-04
    gamma = 0.0121

  Initial states:
    l = 4.8362
    b = 0.0097
    s=-0.1137 -0.2251 -0.0756 0.0623 0.2079 0.2222
           0.1235 -0.009 0 0.0203 -0.1203 -0.0925

  sigma:  0.0367

   AIC   AICc    BIC
-204.1 -199.8 -156.5

>accuracy(fit)

                     ME    RMSE     MAE       MPE   MAPE    MASE
Training set -0.0003695 0.03672 0.02835 -0.007882 0.5206 0.07532

> pred <- forecast(fit, 5)
> pred
         Point Forecast Lo 80 Hi 80 Lo 95 Hi 95
Jan 1961          6.101 6.054 6.148 6.029 6.173
Feb 1961          6.084 6.022 6.146 5.989 6.179
Mar 1961          6.233 6.159 6.307 6.120 6.346
Apr 1961          6.222 6.138 6.306 6.093 6.350
May 1961          6.225 6.131 6.318 6.082 6.367

> plot(pred, main="Forecast for Air Travel",
    ylab="Log(AirPassengers)", xlab="Time")
```

1 Smoothing parameters

2 Future forecasts

```
> pred$mean <- exp(pred$mean)
> pred$lower <- exp(pred$lower)
> pred$upper <- exp(pred$upper)
> p <- cbind(pred$mean, pred$lower, pred$upper)
> dimnames(p)[[2]] <- c("mean", "Lo 80", "Lo 95", "Hi 80", "Hi 95")
> p
```

❸ Makes forecasts in the original scale

```
          mean  Lo 80 Lo 95 Hi 80 Hi 95
Jan 1961 446.3 425.8 415.3 467.8 479.6
Feb 1961 438.8 412.5 399.2 466.8 482.3
Mar 1961 509.2 473.0 454.9 548.2 570.0
Apr 1961 503.6 463.0 442.9 547.7 572.6
May 1961 505.0 460.1 437.9 554.3 582.3
```

The smoothing parameters for the level (.82), trend (.0004), and seasonal components (.012) are given in ❶. The low value for the trend (.0004) doesn't mean there is no slope; it indicates that the slope estimated from early observations didn't need to be updated.

The forecast() function produces forecasts for the next five months ❷ and is plotted in figure 15.9. Because the predictions are on a log scale, exponentiation is used to get the predictions in the original metric: numbers (in thousands) of passengers ❸. The matrix pred$mean contains the point forecasts, and the matrices pred$lower and pred$upper contain the 80% and 95% lower and upper confidence limits, respectively. The exp() function is used to return the predictions to the original scale, and cbind() creates a single table. Thus the model predicts 509,200 passengers in March, with a 95% confidence band ranging from 454,900 to 570,000.

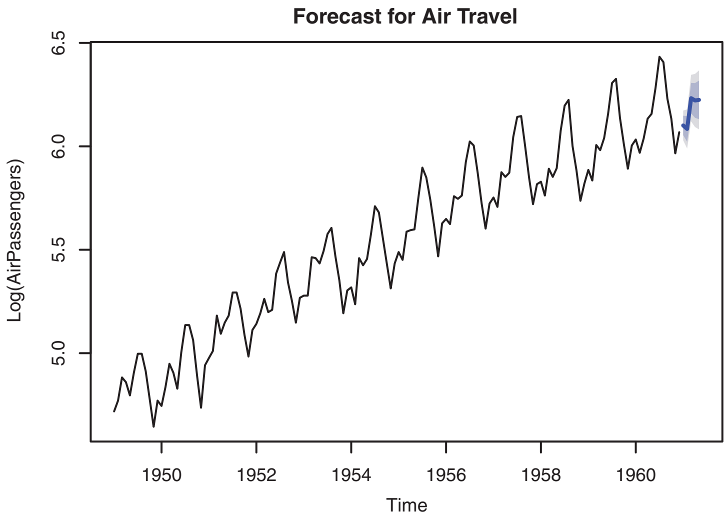


Figure 15.9 Five-year forecast of log(number of international airline passengers in thousands) based on a Holt-Winters exponential smoothing model. Data are from the AirPassengers time series.

15.3.3 The ets() function and automated forecasting

The `ets()` function has additional capabilities. You can use it to fit exponential models that have multiplicative components, add a dampening component, and perform automated forecasts. Let's consider each in turn.

In the previous section, you fit an additive exponential model to the log of the `AirPassengers` time series. Alternatively, you could fit a multiplicative model to the original data. The function call would be either `ets(AirPassengers, model="MAM")` or the equivalent `hw(AirPassengers, seasonal="multiplicative")`. The trend remains additive, but the seasonal and irregular components are assumed to be multiplicative. By using a multiplicative model in this case, the accuracy statistics and forecasted values are reported in the original metric (thousands of passengers)—a decided advantage.

The `ets()` function can also fit a damping component. Time-series predictions often assume that a trend will continue up forever (housing market, anyone?). A damping component forces the trend to a horizontal asymptote over a period of time. In many cases, a damped model makes more realistic predictions.

Finally, you can invoke the `ets()` function to automatically select a best-fitting model for the data. Let's fit an automated exponential model to the Johnson & Johnson data described in the introduction to this chapter. The following code allows the software to select a best-fitting model.

Listing 15.6 Automatic exponential forecasting with `ets()`

```
> library(forecast)
> fit <- ets(JohnsonJohnson)
> fit

ETS(M,M,M)

Call:
 ets(y = JohnsonJohnson)

  Smoothing parameters:
    alpha = 0.2328
    beta  = 0.0367
    gamma = 0.5261

  Initial states:
    l = 0.625
    b = 1.0286
    s=0.6916 1.2639 0.9724 1.0721

  sigma:  0.0863

     AIC     AICc      BIC
162.4737 164.3937 181.9203

> plot(forecast(fit), main="Johnson & Johnson Forecasts",
    ylab="Quarterly Earnings (Dollars)", xlab="Time", flty=2)
```

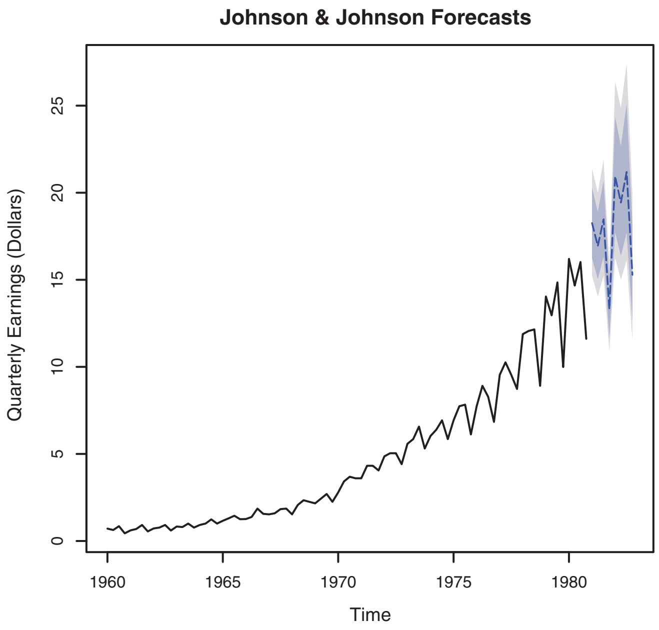


Figure 15.10 Multiplicative exponential smoothing forecast with trend and seasonal components. The forecasts are a dashed line, and the 80% and 95% confidence intervals are provided in light and dark gray, respectively.

Because no model is specified, the software performs a search over a wide array of models to find one that minimizes the fit criterion (log-likelihood by default). The selected model is one that has multiplicative trend, seasonal, and error components. The plot, along with forecasts for the next eight quarters (the default in this case), is given in figure 15.10. The `flty` parameter sets the line type for the forecast line (dashed in this case).

As stated earlier, exponential time-series modeling is popular because it can give good short-term forecasts in many situations. A second approach that is also popular is the Box-Jenkins methodology, commonly referred to as ARIMA models. These are described in the next section.

15.4 ARIMA forecasting models

In the *autoregressive integrated moving average (ARIMA)* approach to forecasting, predicted values are a linear function of recent actual values and recent errors of prediction (residuals). ARIMA is a complex approach to forecasting. In this section, we'll limit discussion to ARIMA models for non-seasonal time series.

Before describing ARIMA models, a number of terms need to be defined, including lags, autocorrelation, partial autocorrelation, differencing, and stationarity. Each is considered in the next section.

15.4.1 Prerequisite concepts

When you *lag* a time series, you shift it back by a given number of observations. Consider the first few observations from the `Nile` time series, displayed in table 15.5. Lag 0 is the unshifted time series. Lag 1 is the time series shifted one position to the left. Lag 2 shifts the time series two positions to the left, and so on. Time series can be lagged using the function `lag(ts,k)`, where *ts* is the time series and *k* is the number of lags.

Table 15.5 The `Nile` time series at various lags

Lag	1869	1870	1871	1872	1873	1874	1875	...
0			1120	1160	963	1210	1160	...
1		1120	1160	963	1210	1160	1160	...
2	1120	1160	963	1210	1160	1160	813	...

Autocorrelation measures the way observations in a time series relate to each other. AC_k is the correlation between a set of observations (Y_t) and observations k periods earlier (Y_{t-k}). So AC_1 is the correlation between the Lag 1 and Lag 0 time series, AC_2 is the correlation between the Lag 2 and Lag 0 time series, and so on. Plotting these correlations (AC_1, AC_2, …, AC_k) produces an *autocorrelation function (ACF) plot*. The ACF plot is used to select appropriate parameters for the ARIMA model and to assess the fit of the final model.

An ACF plot can be produced with the `acf()` function in the `stats` package or the `Acf()` function in the `forecast` package. Here, the `Acf()` function is used because it produces a plot that is somewhat easier to read. The format is `Acf(ts)`, where *ts* is the original time series. The ACF plot for the `Nile` time series, with k=1 to 18, is provided a little later, in the top half of figure 15.12.

A *partial autocorrelation* is the correlation between Y_t and Y_{t-k} with the effects of all Y values between the two (Y_{t-1}, Y_{t-2}, …, Y_{t-k+1}) removed. Partial autocorrelations can also be plotted for multiple values of k. The PACF plot can be generated with either the `pacf()` function in the `stats` package or the `Pacf()` function in the `forecast` package. Again, the `Pacf()` function is preferred due to its formatting. The function call is `Pacf(ts)`, where *ts* is the time series to be assessed. The PACF plot is also used to determine the most appropriate parameters for the ARIMA model. The results for the `Nile` time series are given in the bottom half of figure 15.12.

ARIMA models are designed to fit *stationary* time series (or time series that can be made stationary). In a stationary time series, the statistical properties of the series don't change over time. For example, the mean and variance of Y_t are constant. Additionally, the autocorrelations for any lag k don't change with time.

It may be necessary to transform the values of a time series in order to achieve constant variance before proceeding to fitting an ARIMA model. The log transformation is often useful here, as you saw in section 15.1.3. Other transformations, such as the Box-Cox transformation described in section 8.5.2, may also be helpful.

Because stationary time series are assumed to have constant means, they can't have a trend component. Many non-stationary time series can be made stationary through *differencing*. In differencing, each value of a time series Y_t is replaced with $Y_{t-1} – Y_t$. Differencing a time series once removes a linear trend. Differencing it a second time removes a quadratic trend. A third time removes a cubic trend. It's rarely necessary to difference more than twice.

You can difference a time series with the `diff()` function. The format is `diff(`*`ts`*`, differences=`*`d`*`)`, where d indicates the number of times the time series *`ts`* is differenced. The default is d=1. The `ndiffs()` function in the `forecast` package can be used to help determine the best value of d. The format is `ndiffs(`*`ts`*`)`.

Stationarity is often evaluated with a visual inspection of a time-series plot. If the variance isn't constant, the data are transformed. If there are trends, the data are differenced. You can also use a statistical procedure called the *Augmented Dickey-Fuller (ADF) test* to evaluate the assumption of stationarity. In R, the function `adf.test()` in the `tseries` package performs the test. The format is `adf.test(`*`ts`*`)`, where *`ts`* is the time series to be evaluated. A significant result suggests stationarity.

To summarize, ACF and PCF plots are used to determine the parameters of ARIMA models. Stationarity is an important assumption, and transformations and differencing are used to help achieve stationarity. With these concepts in hand, we can now turn to fitting models with an autoregressive (AR) component, a moving averages (MA) component, or both components (ARMA). Finally, we'll examine ARIMA models that include ARMA components and differencing to achieve stationarity (Integration).

15.4.2 ARMA and ARIMA models

In an *autoregressive* model of order p, each value in a time series is predicted from a linear combination of the previous p values

$$AR(p): Y_t = \mu + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \ldots + \beta_p Y_{t-p} + \varepsilon_t$$

where Y_t is a given value of the series, μ is the mean of the series, the βs are the weights, and ε_t is the irregular component. In a *moving average* model of order q, each value in the time series is predicted from a linear combination of q previous errors. In this case

$$MA(q): Y_t = \mu - \theta_1 \varepsilon_{t-1} - \theta_2 \varepsilon_{t-2} \ldots - \theta_q \varepsilon_{t-q} + \varepsilon_t$$

where the εs are the errors of prediction and the θs are the weights. (It's important to note that the moving averages described here aren't the simple moving averages described in section 15.1.2.)

Combining the two approaches yields an ARMA(p, q) model of the form

$$Y_t = \mu + \beta_1 Y_{t-1} + \beta_2 Y_{t-2} + \ldots + \beta_p Y_{t-p} - \theta_1 \varepsilon_{t-1} - \theta_2 \varepsilon_{t-2} \ldots - \theta_q \varepsilon_{t-q} + \varepsilon_t$$

that predicts each value of the time series from the past p values and q residuals.

An ARIMA(p, d, q) model is a model in which the time series has been differenced d times, and the resulting values are predicted from the previous p actual values and q previous errors. The predictions are "un-differenced" or *integrated* to achieve the final prediction.

The steps in ARIMA modeling are as follows:

1. Ensure that the time series is stationary.
2. Identify a reasonable model or models (possible values of p and q).
3. Fit the model.
4. Evaluate the model's fit, including statistical assumptions and predictive accuracy.
5. Make forecasts.

Let's apply each step in turn to fit an ARIMA model to the `Nile` time series.

ENSURING THAT THE TIME SERIES IS STATIONARY

First you plot the time series and assess its stationarity (see listing 15.7 and the top half of figure 15.11). The variance appears to be stable across the years observed, so there's no need for a transformation. There may be a trend, which is supported by the results of the `ndiffs()` function.

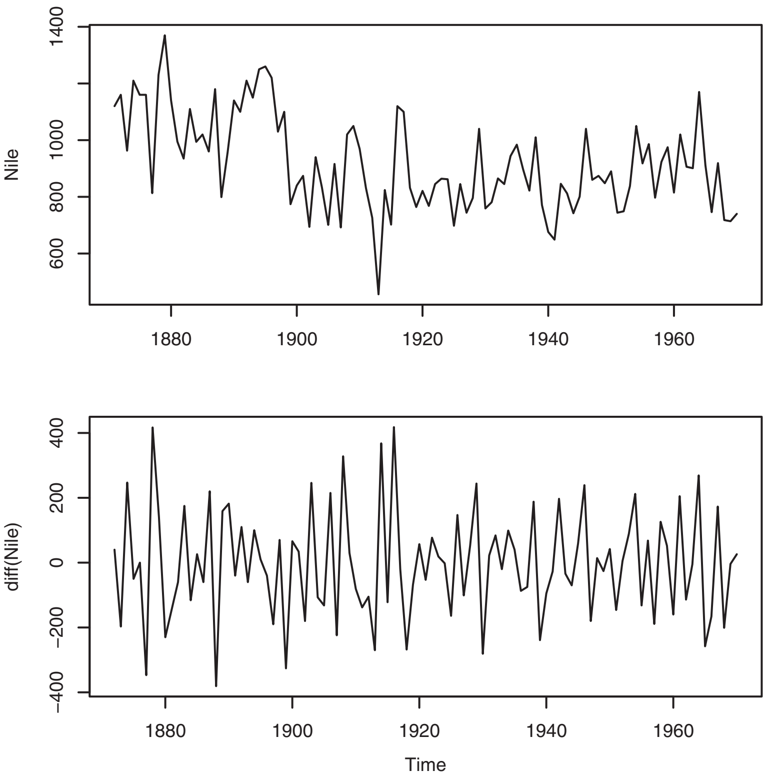


Figure 15.11 Time series displaying the annual flow of the river Nile at Ashwan from 1871 to 1970 (top) along with the times series differenced once (bottom). The differencing removes the decreasing trend evident in the original plot.

Listing 15.7 Transforming the time series and assessing stationarity

```
> library(forecast)
> library(tseries)
> plot(Nile)
> ndiffs(Nile)

[1] 1

> dNile <- diff(Nile)
> plot(dNile)
> adf.test(dNile)

    Augmented Dickey-Fuller Test

data:  dNile
Dickey-Fuller = -6.5924, Lag order = 4, p-value = 0.01
alternative hypothesis: stationary
```

The series is differenced once (lag=1 is the default) and saved as dNile. The differenced time series is plotted in the bottom half of figure 15.11 and certainly looks more stationary. Applying the ADF test to the differenced series suggest that it's now stationary, so you can proceed to the next step.

IDENTIFYING ONE OR MORE REASONABLE MODELS

Possible models are selected based on the ACF and PACF plots:

```
Acf(dNile)
Pacf(dNile)
```

The resulting plots are given in figure 15.12.

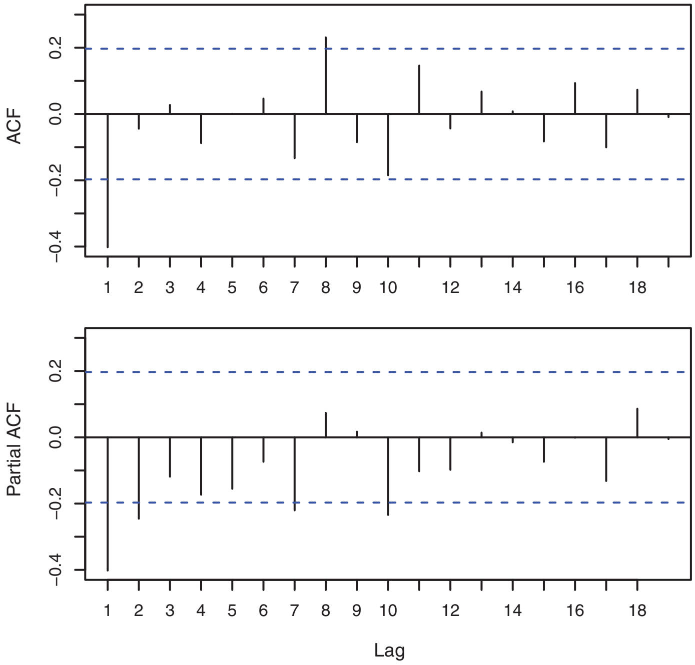


Figure 15.12 Autocorrelation and partial autocorrelation plots for the differenced Nile time series

The goal is to identify the parameters p, d, and q. You already know that d=1 from the previous section. You get p and q by comparing the ACF and PACF plots with the guidelines given in table 15.6.

Table 15.6 Guidelines for selecting an ARIMA model

Model	ACF	PACF
ARIMA(p, d, 0)	Trails off to zero	Zero after lag p
ARIMA(0, d, q)	Zero after lag q	Trails off to zero
ARIMA(p, d, q)	Trails off to zero	Trails off to zero

The results in table 15.6 are theoretical, and the actual ACF and PACF may not match this exactly. But they can be used to give a rough guide of reasonable models to try. For the `Nile` time series in figure 15.12, there appears to be one large autocorrelation at lag 1, and the partial autocorrelations trail off to zero as the lags get bigger. This suggests trying an ARIMA(0, 1, 1) model.

FITTING THE MODEL(S)

The ARIMA model is fit with the `arima()` function. The format is `arima(`*`ts`*`, order=c(q, d, q))`. The result of fitting an ARIMA(0, 1, 1) model to the `Nile` time series is given in the following listing.

Listing 15.8 Fitting an ARIMA model

```
> library(forecast)
> fit <- arima(Nile, order=c(0,1,1))
> fit

Series: Nile
ARIMA(0,1,1)

Coefficients:
          ma1
      -0.7329
s.e.   0.1143

sigma^2 estimated as 20600:  log likelihood=-632.55
AIC=1269.09   AICc=1269.22   BIC=1274.28

> accuracy(fit)

                 ME  RMSE   MAE    MPE  MAPE   MASE
Training set -11.94 142.8 112.2 -3.575 12.94 0.8089
```

Note that you apply the model to the original time series. By specifying d=1, it calculates first differences for you. The coefficient for the moving averages (-0.73) is provided along with the AIC. If you fit other models, the AIC can help you choose which one is most reasonable. Smaller AIC values suggest better models. The accuracy measures can help you determine whether the model fits with sufficient accuracy. Here the mean absolute percent error is 13% of the river level.

EVALUATING MODEL FIT

If the model is appropriate, the residuals should be normally distributed with mean zero, and the autocorrelations should be zero for every possible lag. In other words, the residuals should be normally and independently distributed (no relationship between them). The assumptions can be evaluated with the following code.

Listing 15.9 Evaluating the model fit

```
> qqnorm(fit$residuals)
> qqline(fit$residuals)
> Box.test(fit$residuals, type="Ljung-Box")

    Box-Ljung test

data:  fit$residuals
X-squared = 1.3711, df = 1, p-value = 0.2416
```

The qqnorm() and qqline() functions produce the plot in figure 15.13. Normally distributed data should fall along the line. In this case, the results look good.

The Box.test() function provides a test that the autocorrelations are all zero. The results aren't significant, suggesting that the autocorrelations don't differ from zero. This ARIMA model appears to fit the data well.

MAKING FORECASTS

If the model hadn't met the assumptions of normal residuals and zero autocorrelations, it would have been necessary to alter the model, add parameters, or try a different approach. Once a final model has been chosen, it can be used to make predictions of future values. In the next listing, the forecast() function from the forecast package is used to predict three years ahead.

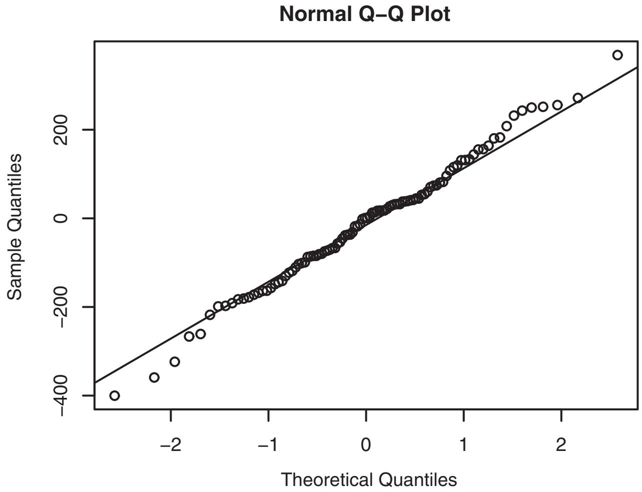


Figure 15.13 Normal Q-Q plot for determining the normality of the time-series residuals

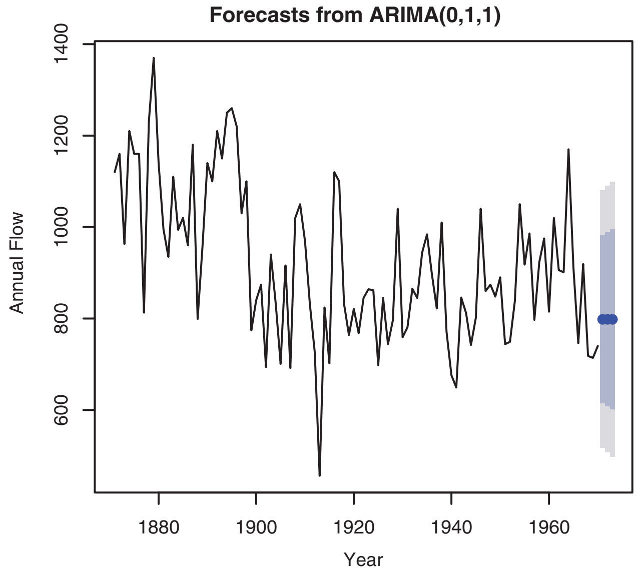


Figure 15.14 Three-year forecast for the Nile time series from a fitted ARIMA(0,1,1) model. Blue dots represent point estimates, and the light and dark gray bands represent the 80% and 95% confidence bands limits, respectively.

Listing 15.10 Forecasting with an ARIMA model

```
> forecast(fit, 3)

     Point Forecast    Lo 80    Hi 80    Lo 95    Hi 95
1971       798.3673 614.4307 982.3040 517.0605 1079.674
1972       798.3673 607.9845 988.7502 507.2019 1089.533
1973       798.3673 601.7495 994.9851 497.6663 1099.068

> plot(forecast(fit, 3), xlab="Year", ylab="Annual Flow")
```

The `plot()` function is used to plot the forecast in figure 15.14. Point estimates are given by the blue dots, and 80% and 95% confidence bands are represented by dark and light bands, respectively.

15.4.3 Automated ARIMA forecasting

In section 15.2.3, you used the `ets()` function in the `forecast` package to automate the selection of a best exponential model. The package also provides an `auto.arima()` function to select a best ARIMA model. The next listing applies this approach to the `sunspots` time series described in the chapter introduction.

Listing 15.11 Automated ARIMA forecasting

```
> library(forecast)
> fit <- auto.arima(sunspots)
> fit
Series: sunspots
ARIMA(2,1,2)
```

```
Coefficients:
         ar1      ar2     ma1     ma2
        1.35   -0.396   -1.77   0.810
s.e.    0.03    0.029    0.02   0.019

sigma^2 estimated as 243:  log likelihood=-11746
AIC=23501   AICc=23501   BIC=23531

> forecast(fit, 3)


         Point Forecast        Lo 80    Hi 80      Lo 95    Hi 95
Jan 1984      40.437722   20.4412613 60.43418   9.855774 71.01967
Feb 1984      41.352897   18.2795867 64.42621   6.065314 76.64048
Mar 1984      39.796425   15.2537785 64.33907   2.261686 77.33116

> accuracy(fit)
                   ME RMSE   MAE MPE MAPE MASE
Training set -0.02673 15.6 11.03 NaN  Inf 0.32
```

The function selects an ARIMA model with p=2, d=1, and q=2. These are values that minimize the AIC criterion over a large number of possible models. The MPE and MAPE accuracy blow up because there are zero values in the series (a drawback of these two statistics). Plotting the results and evaluating the fit are left for you as an exercise.

15.5 Going further

There are many good books on time-series analysis and forecasting. If you're new to the subject, I suggest starting with the book *Time Series* (Open University, 2006). Although it doesn't include R code, it provides a very understandable and intuitive introduction. *A Little Book of R for Time Series* by Avril Coghlan (http://mng.bz/8fz0, 2010) pairs well with the Open University text and includes R code and examples.

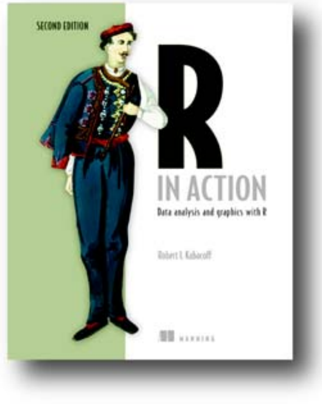
Forecasting: Principles and Practice (http://otexts.com/fpp, 2013) is a clear and concise online textbook written by Rob Hyndman and George Athanasopoulos; it includes R code throughout. I highly recommend it. Additionally, Cowpertwait & Metcalfe (2009) have written an excellent text on analyzing time series with R. A more advanced treatment that also includes R code can be found in Shumway & Stoffer (2010).

Finally, you can consult the CRAN Task View on Time Series Analysis (http://cran.r-project.org/web/views/TimeSeries.html). It contains a comprehensive summary of all of R's time-series capabilities.

15.6 Summary

Forecasting has a long and varied history, from early shamans predicting the weather to modern data scientists predicting the results of recent elections. Prediction is fundamental to both science and human nature. In this chapter, we've looked at how to create time series in R, assess trends, and examine seasonal effects. Then we considered two of the most popular approaches to forecasting: exponential models and ARIMA models.

Although these methodologies can be crucial in understanding and predicting a wide variety of phenomena, it's important to remember that they each entail extrapolation—going beyond the data. They assume that future conditions mirror current conditions. Financial predictions made in 2007 assumed continued economic growth in 2008 and beyond. As we all know now, that isn't exactly how things turned out. Significant events can change the trend and pattern in a time series, and the farther out you try to predict, the greater the uncertainty.



Business pros and researchers thrive on data, and R speaks the language of data analysis. R is a powerful programming language for statistical computing. Unlike general-purpose tools, R provides thousands of modules for solving just about any data-crunching or presentation challenge you're likely to face. R runs on all important platforms and is used by thousands of major corporations and institutions worldwide.

R in Action, Second Edition teaches you how to use the R language by presenting examples relevant to scientific, technical, and business developers. Focusing on practical solutions, the book offers a crash course in statistics, including elegant methods for dealing with messy and incomplete data. You'll also master R's extensive graphical capabilities for exploring and presenting data visually. And this expanded second edition includes new chapters on forecasting, data mining, and dynamic report writing.

What's inside

- Complete R language tutorial
- Using R to manage, analyze, and visualize data
- Techniques for debugging programs and creating packages
- OOP in R
- Over 160 graphs

This book is designed for readers who need to solve practical data analysis problems using the R language and tools. Some background in mathematics and statistics is helpful, but no prior experience with R or computer programming is required.

Deep learning and neural networks

Neural networks have long been a supervised machine learning tool of choice when data is numeric and represents a physical situation. This includes working with images, recorded sound, and scientific measurements. The following chapter introduces the basic concepts behind classic neural net applications. The examples are implemented in Python and scikit-learn, which, together with the pandas package, offer a powerful programming platform for machine learning and data science. The chapter ends with a description of so-called restricted Boltzmann machines and the move to unsupervised training procedures that anticipate current advanced methods such as deep learning and word2vec.

Chapter 6 from *Algorithms of the Intelligent Web, Second Edition* by Douglas G. McIlwraith, Haralambos Marmanis, and Dmitry Babenko

Deep learning and neural networks

This chapter covers

- Neural network basics
- An introduction to deep learning
- Digit recognition using restricted Boltzmann machines

There is much discussion about deep learning at the moment, and it's widely seen to be the next big advance in machine learning and artificial intelligence. In this chapter we'd like to cut through the rhetoric to provide you with the facts. At the end of this chapter you should understand the basic building block of any deep learning network, the perceptron, and understand how these fit together in a deep network. Neural networks heralded the introduction of the perceptron, so we'll discuss these before exploring deeper, more expressive networks. These deeper networks come with significant challenges in representation and training, so we need to ensure a good foundational knowledge before leaping in.

Before we do all of this, we'll discuss the nature of deep learning and the kinds of problems deep learning has been applied to and what makes these successful. This should give you a foundational motivation for deep learning and a frame on which to hang some of the more complicated theoretical concepts later in the chapter. Remember that this is a still a vibrant and active area of research in the community, and so I recommend that you keep abreast of the latest advances by following the literature. The following resources can provide an up-to-date summary of what's happening in the community: Startup[1] and KDNuggets.[2] But I urge you to do your own research and come up with your own conclusions!

6.1 An intuitive approach to deep learning

In order to understand deep learning, let's choose the application of image recognition; namely, given a picture or a video, how do we build classifiers that will recognize objects? Such an application has potentially wide-reaching applications. With the advent of the quantified self[3,4] and Google Glass, we can imagine applications for this device that recognize objects and provide information to the user.

Let's take the example of recognizing a car. Deep learning builds up layers of understanding, with each layer utilizing the previous one. Figure 6.1 shows some of the possible layers of understanding for a deep network trained to recognize cars. Both this example and some of the images that follow have been reproduced from Andrew Ng's lecture on the subject.[5]

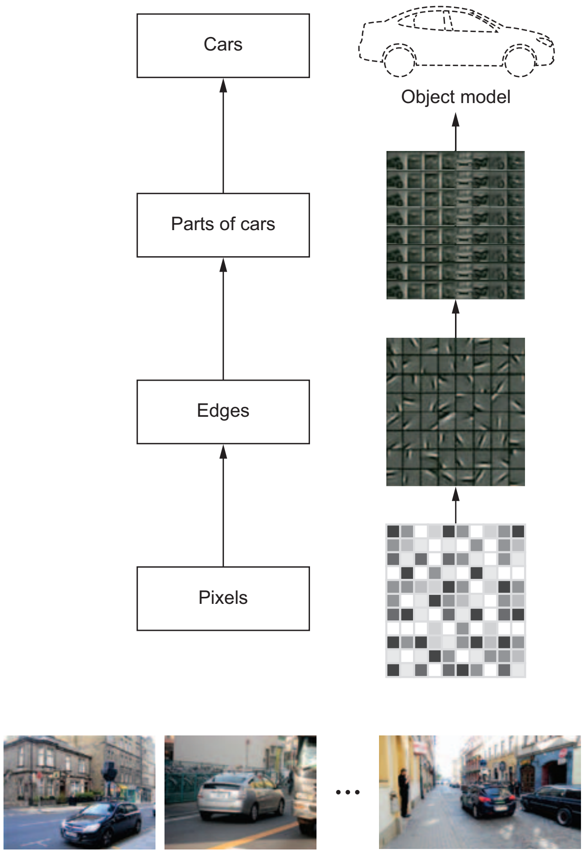


Figure 6.1 Visualizing a deep network for recognizing cars. Some graphical content reproduced from Andrew Ng's talk on the subject, cited previously. A base training set of pictures is used to create a basis of edges. These edges can be combined to detect parts of cars, and these parts of cars can be combined to detect an object type, which is in this case a car.

At the bottom of figure 6.1 you can see a number of stock images of cars. We'll consider

[1] Startup.ML, "Deep Learning News," June 30, 2015, http://news.startup.ml/ (accessed December 21, 2015).

[2] KDNuggets, "Deep Learning," http://www.kdnuggets.com/tag/deep-learning (accessed December 21, 2015).

[3] Gina Neff and Dawn Nafus, *The Quantified Self* (Boston: MIT Press, 2016).

[4] Deborah Lupton, *The Quantified Self* (Cambridge: Polity Press, 2016).

[5] Andrew Ng, "Bay Area Vision Meeting: Unsupervised Feature Learning and Deep Learning," YouTube, March 7, 2011, https://www.youtube.com/watch?v=ZmNOAtZIgIk (accessed December 21, 2015).

these our training set. The question is now how do we use deep learning to recognize the similarities between these images, that is, that they all contain a car, possibly without any hand-labeled ground truth? The algorithm isn't told that the scene contains a car.

As you'll see, deep learning relies on progressively higher-concept abstractions built directly from lower-level abstractions. In the case of our image-recognition problem, we start out with the smallest element of information in our pictures, the pixel. The entire image set is used to construct a basis of features—think back to chapter 3 where we discussed extracting structure from data—that can be used in composite to detect a slightly higher level of abstraction such as lines and curves. In the next-highest level, these lines are curves that are combined to create parts of cars that have been seen in the training set, and these parts are further combined to create object detectors for a whole car.

There are two important concepts to note here. First, no explicit feature engineering has been performed. If you remember, at the end of the last chapter we talked about the importance of creating a good representation of your data. We discussed this in the context of click prediction for advertising and noted that experts in the space typically perform this manually. But in this example, unsupervised feature learning has been performed; that is, representations of the data have been learned without any explicit interaction from the user. This may parallel how we as humans may perform recognition—and we are very good at pattern recognition indeed!

The second important fact to note is that the concept of a car has not been made explicit. Provided sufficient variance in the input set of pictures, the highest-level car detectors should do sufficiently well on any car presented. Before we get ahead of ourselves, though, let's clear up some of the basics around neural networks.

6.2 Neural networks

Neural networks aren't a new technology by any means and have been around since the 1940s. They are a biologically inspired concept whereby an output neuron is activated based on the input from several connected input neurons. Neural networks are sometimes known as artificial neural networks, because they achieve artificially a similar functionality to a human neuron. Jeubin Huang[1] provides an introduction to the biology of the human brain. Although many aspects about the functionality of the human brain are still a mystery, we're able to understand the basic building blocks of operation—but how this gives rise to consciousness is another matter.

Neurons in the brain use a number of dendrites to collect both positive (excitative) and negative (inhibitory) output information from other neurons and encode this electrically, sending this down an axon. This axon splits and reaches hundreds or thousands of dendrites attached to other neurons. A small gap exists between the axon and the input dendrites of the next neuron, and this gap is known as a synapse.

[1] Jeubin Huang, "Overview of Cerebral Function," Merck Manual, September 1, 2015, http://www.merckmanuals.com/professional/neurologic-disorders/function-and-dysfunction-of-the-cerebral-lobes/overview-of-cerebral-function (accessed December 21, 2015).

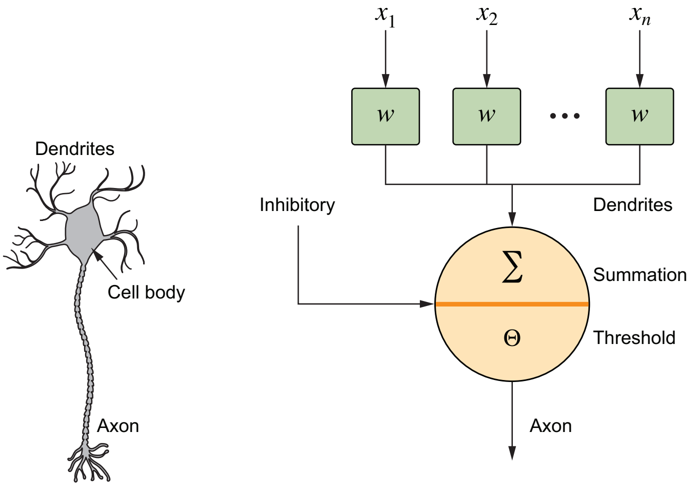


Figure 6.2 On the left we provide a schematic of a human biological neuron. To the right, we show a human-inspired neural network implemented using weighted summation, an inhibitory input, and a threshold.

Electrical information is converted into chemical output that then excites the dendrite of the next neuron. In this scenario, learning is encoded by the neuron itself. Neurons send messages down their axon only if their overall excitation is large enough.

Figure 6.2 shows the schematic of a human biological neuron and an artificial neuron developed by McCulloch and Pitts, the so-called MCP model.[1] Our artificial neuron is built using a simple summation and threshold value and works as follows. Logic inputs, both positive and negative, are received from the dendrite equivalents and a weighted summation is performed. If this output exceeds a certain threshold and no inhibitory input is observed, a positive value is emitted. This output may then be fed onward to the input of other such neurons through their dendrites' equivalent inputs. A little thought will reveal that this is—ignoring the inhibitory input—a linear model in n dimensional space, with linked coefficients, where n is the number of inputs to the neuron. Figure 6.3 illustrates the behavior of this model for $n = 1$.

In this illustration we use a simple hand-built neuron with a unit weight ($w = 1$). The input to the neuron is allowed to vary from -10 to 10, and the summation of the weighted input values are provided on the Y-axis. Choosing a threshold of 0, the neuron would fire if the input is greater than 0 but not otherwise.

[1] Warren S. McCulloch and Walter H. Pitts, "A Logical Calculus of the Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics* 5 (1943): 115–33.

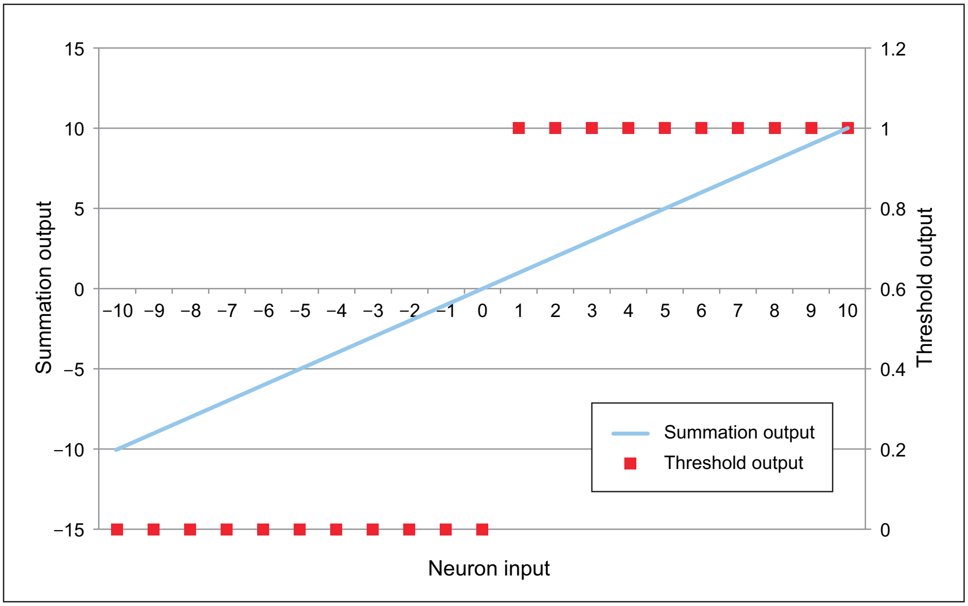


Figure 6.3 MCP as a 2-D linear model without inhibitory input. The weights of the model correspond to the coefficients of a linear model. In this case our neuron supports only a single input and the weight has been set to 1 for illustration. Given a threshold of 0, all inputs with a value less than or equal to 0 would inhibit the neuron from firing, whereas all inputs with a value greater than 0 would cause the neuron to fire.

6.3 The perceptron

In the previous section we introduced the MCP neuron. With this basic approach, it turns out that it's possible to learn and to generalize training data but in a very limited fashion. But we can do better, and thus the perceptron was born. The perceptron builds on the MCP model in three important ways:[1,2]

- A threshold bias was added as an input to the summation. This serves several equivalent purposes. First, it allows bias to be captured from the input neurons. Second, it means that output thresholds can be standardized around a single value, such as zero, without loss of generality.
- The perceptron allows input weights to be independent, so a neuron doesn't need to be connected to an input multiple times to have a greater impact.
- The development of the perceptron heralded the development of an algorithm to learn the best weights given a set of input and output data.

[1] Frank Rosenblatt, *The Perceptron—a perceiving and recognizing automaton* (New York: Cornell Aeronautical Laboratory, 1957).

[2] Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review* 65, no. 6 (November 1958): 386–408.

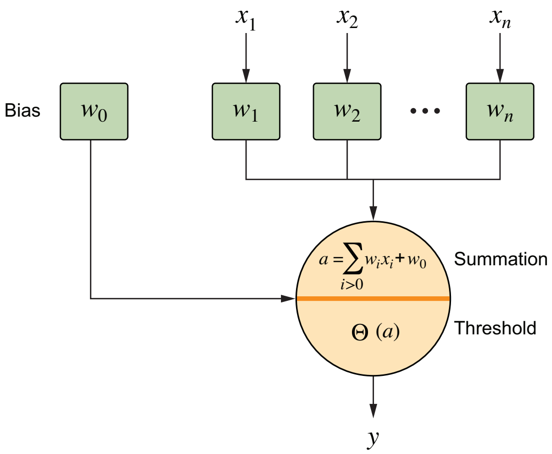


Figure 6.4 The perceptron. Inputs x_1 through x_n are received and multiplied by their associated weight, with perceptron bias, w_0, being added in afterward. This output, given by a, is then passed through a threshold function to obtain the output.

Figure 6.4 provides a graphical overview of this new extended model. As before, an intermediate value is created using the weighted summation of the inputs, but we now notice the inclusion of a bias value, w_0. This is learned along with the input weights during the training step; more about this in the following sections. The intermediate value, denoted by a here, is then passed through a threshold function to obtain the final result, y.

6.3.1 Training

Now that you know that a neural network consists of many more basic elements called perceptrons, let's look at how we train a perceptron in isolation. So what does it mean to train a perceptron? Let's take a more concrete example using the logical AND function. We'll consider a perceptron of two binary inputs with a binary threshold activation function around 0. How do we learn the weights, such that the output of the perceptron is 1, if and only if the two inputs are both equal to 1? Put another way, can we choose continuous valued weights such that the weighted sum of the inputs is greater than 0 when the two inputs are both 1 with the output being less than 0 otherwise? Let's formalize this problem. We give X as our vector of binary input values and W as our vector of continuous input weights.

$$X = (x_1, x_2), \ W = (w_1, w_2)$$

Thus, we need to learn weights such that the following restrictions hold true for combinations of binary inputs x_1, x_2:

$$X \cdot W > 0, \quad x_1 = x_2 = 1$$
$$\leq 0 \ otherwise$$

Unfortunately for us, there are no solutions if we pose the problem like this! There are two options to make this problem tractable. We either allow the threshold to move, defining it as a value not equal to 0, or we introduce an offset; both are equivalent. We'll opt for the latter in this text, providing us with the new vectors

$$X = (x_1, x_2, 1),\ W = (w_1, w_2, \theta)$$

and our existing equalities remain the same. You can now see that with careful weight selection we can create the AND function. Consider the case where

$$w_1 = 1,\ w_2 = 1\ and\ \theta = -1.5$$

Table 6.1 provides the output from our perceptron and the output from the AND function.

Table 6.1 Comparing the output of our perceptron with the output of the logical AND function, which returns 1 if both inputs are equal to 1. Results provided are for the case where $w_1 - 1, w_2 - 1\ and\ \theta = -1.5$.

x_1	x_2	theta	weighted sum	sign of weighted sum	x_1 AND x_2
1	0	-1.5	-0.5	negative	0
0	1	-1.5	-0.5	negative	0
0	0	-1.5	-1.5	negative	0
1	1	-1.5	0.5	positive	1

Now that we understand that it's indeed possible to represent a logical AND using a perceptron, we must develop a systematic way to learn our weights in a supervised manner. Put another way, given a data set consisting of inputs and outputs, related linearly in this case, how do we learn the weights of our perceptron? We can achieve this using the perceptron algorithm developed by Rosenblatt.[1,2] In the following listing we present the pseudo code used for learning.

Listing 6.1 The perceptron learning algorithm

```
Initialize W to contain random small numbers

For each item in training set:
    Calculate the current output of the perceptron for the item.
    For each weight, update, depending upon output correctness.
```

So far, so good. Looks easy, right? We start with some small random values for our weights and then we iterate over our data points and update the weights depending on the correctness of our perceptron. In actual fact, we update the weights only if we get the output wrong; otherwise, we leave the weights alone. Furthermore, we update

[1] Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain."

[2] Rosenblatt, *Principles of neurodynamics; perceptrons and the theory of brain mechanisms* (Washington: Spartan Books, 1962).

the weights such that they become more like their input vectors in magnitude but with the corresponding sign of the output value. Let's write this down more formally, as shown in the next listing.

Listing 6.2 The perceptron learning algorithm (2)

```
Initialize $W_0$ to contain random small numbers
For each example j:
    $y_j(t) = \sum_k w_k(t) \cdot x_{j,k}$

    For each feature weight k:
    $w_k(t+1) = w_k(t) + n(d_j - y_j(t)) \cdot x_{j,k}$
```

Calculate the output of the perceptron given the current weights (time t).

Features are updated only if the expected output *d*, and the actual output differ. If they do, we move the weight in the sign of the correct answer but a magnitude given by the corresponding input.

Update the features. Note this is often done as a vector operation rather than a for loop.

Provided the input data is linearly separable, such an algorithm is guaranteed to converge to a solution.[1]

6.3.2 Training a perceptron in scikit-learn

Previously we presented the simplest form of a neural network, the perceptron; we also discussed how this algorithm is trained. Let's now move to scikit-learn and explore how we can train a perceptron using some data. The next listing provides the code to perform the necessary imports and create a NumPy array of data points.

Listing 6.3 Creating data for perceptron learning

```
import numpy as np
import matplotlib.pyplot as plt
import random
from sklearn.linear_model import perceptron

#Let's set up our data and our target
data = np.array([[0,1],[0,0],[1,0],[1,1]])
target = np.array([0,0,0,1])
```

Create four data points in a NumPy array.

Specify the classes of these data points. In this example only data point x=1,y=1 has a target class of 1; other data points have been assigned a class of 0.

In listing 6.3, we perform the necessary imports for our single perceptron example and create a very small dataset with only four data points. This dataset is contained within a NumPy array called `data`. Each data point is assigned to either the class `0` or the class `1`, with these classes being stored within the `target` array. Figure 6.5 provides a graphical overview of this data.

In figure 6.5 the only data point with a positive class (with a label of 1) is found at coordinate (1,1) and represented in red. All other data points are associated with the

[1] Brian Ripley, *Pattern Recognition and Neural Networks* (Cambridge: Cambridge University Press, 1996).

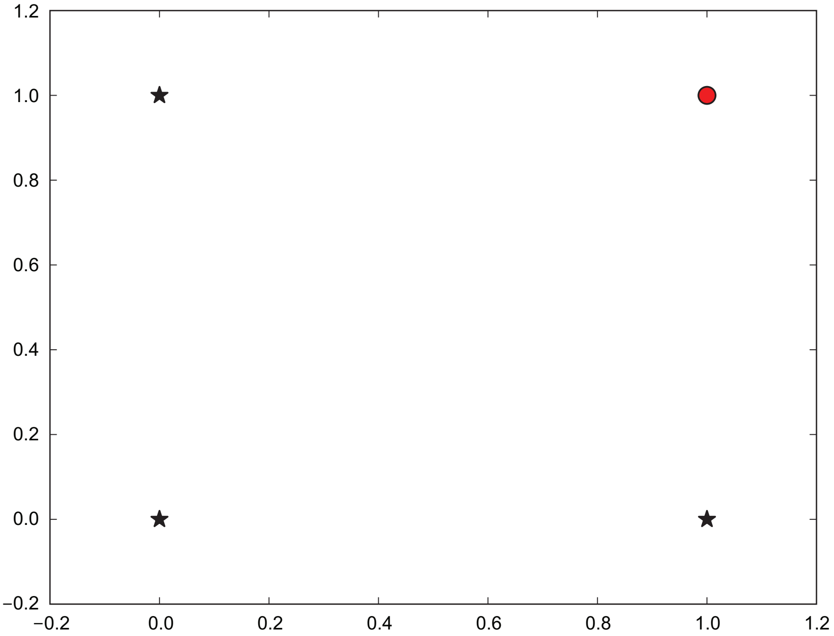


Figure 6.5 Graphical overview of our data for a single perceptron. Data with class label 1 is represented by a round dot, whereas data with class label 0 is represented by a star. It is the aim of the perceptron to separate these points.

negative class. The following listing provides the sample code to train our simple perceptron and to return the coefficients (w_1,w_2 relating to x_1 and x_2, respectively) along with the bias w_0.

Listing 6.4 Training a single perceptron

```
p = perceptron.Perceptron(n_iter=100)
p_out = p.fit(data,target)
print p_out
msg = ("Coefficients: %s, Intercept: %s")
print msg % (str(p.coef_),str(p.intercept_))
```

Create a single perceptron. n_iter specifies the number of times the training data should be iterated through when training.

Train the perceptron using the data and the associated target classes.

Print out the coefficients and the bias of the perceptron.

Listing 6.4 provides output similar to the following:

```
Perceptron(alpha=0.0001, class_weight=None, eta0=1.0, fit_intercept=True,
      n_iter=100, n_jobs=1, penalty=None, random_state=0, shuffle=False,
      verbose=0, warm_start=False)
Coefficients: [[ 3.  2.]] ,Intercept: [-4.]
```

The first line provides the parameters under which the perceptron was trained; the second provides the output weights and bias of the perceptron. Don't worry if the coefficients and the intercept are slightly different when you run this. There are many solutions to this problem and the learning algorithm can return any of them. For a greater understanding of the parameters, I encourage you to read the associated scikit-learn documentation.[1]

6.3.3 A geometric interpretation of the perceptron for two inputs

In this example, we've successfully trained a single perceptron and returned the weights and bias of the final perceptron. Great! But how do we interpret these weights in an intuitive way? Luckily, this is easily possible in 2-D space, and we can extend this intuition into higher-dimensional spaces also.

Let's consider the perceptron for two input values only. From figure 6.3 we have

$$y = w_0 + w_1 x_1 + w_2 x_2$$

This should look familiar to you as the equation of a plane (three dimensions) in x_1, x_2, and y. If they're not equivalent, this plane intersects with the viewing plane of figure 6.5 and you're left with a line. When viewed from the point of reference of figure 6.5, points to one side of the line correspond to values of $x \cdot W > 0$, whereas points on the other side of the line correspond to values of $x \cdot W < 0$. Points on the line correspond to $x \cdot W = 0$. Let's take the concrete example from earlier and visualize this. Using the coefficients just provided, we have the equation of a plane given by

$$y = -4 + 3x_1 + 2x_2$$

The value of y is at 90 degrees to the (x_1,x_2) plane and thus can be thought of as along a line following the eyes of the viewer, straight through the viewing plane. The value of y on the viewing plane is given by 0, and so we can find the line of intersection by substituting the value of y=0 in the previous equation:

$$0 = 4 + 3x_1 + 2x_2$$

$$4 - 3x_1 = 2x_2$$

$$x_2 = \frac{4}{2} - \frac{3}{2}x_1$$

This last line follows the standard form of a straight line; now all we need to do is plot this to see how the perceptron has separated our training data. The next listing provides the associated code, and figure 6.6 provides the output.

[1] Scikit-learn, "Perceptron," January 01, 2014, http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html (accessed February 25, 2016).

Listing 6.5 Plotting the output of the perceptron

```
colors = np.array(['k','r'])
markers = np.array(['*','o'])

for data,target in zip(data,target):
plt.scatter(data[0],data[1],s=100,
    c=colors[target],marker=markers[target])

grad = -p.coef_[0][0]/p.coef_[0][1]
intercept = -p.intercept_/p.coef_[0][1]

x_vals = np.linspace(0,1)
y_vals = grad*x_vals + intercept
plt.plot(x_vals,y_vals)
plt.show()
```

Set up a color array and plot the data points: black star for class zero, red dot for class 1.

Calculate the parameters of the straight line (intersection of two planes).

Create the data points and plot the line of intersection between the two planes.

In listing 6.5 we plot our four data points along with the projection of the separating plane learned by the perceptron onto the viewing plane. This provides us with a separation as per figure 6.6. In general, for a larger number of input variables, you can think of these as existing in n dimensional space, with the perceptron separating these using a hyperplane in n+1 dimensions. You should now be able to see that the basic linear form of the perceptron, that is, with a threshold activation, is equivalent to separation using a hyperplane. Consequently, such models are of use only where data is linearly separable and will be unable to separate positive and negative classes otherwise.

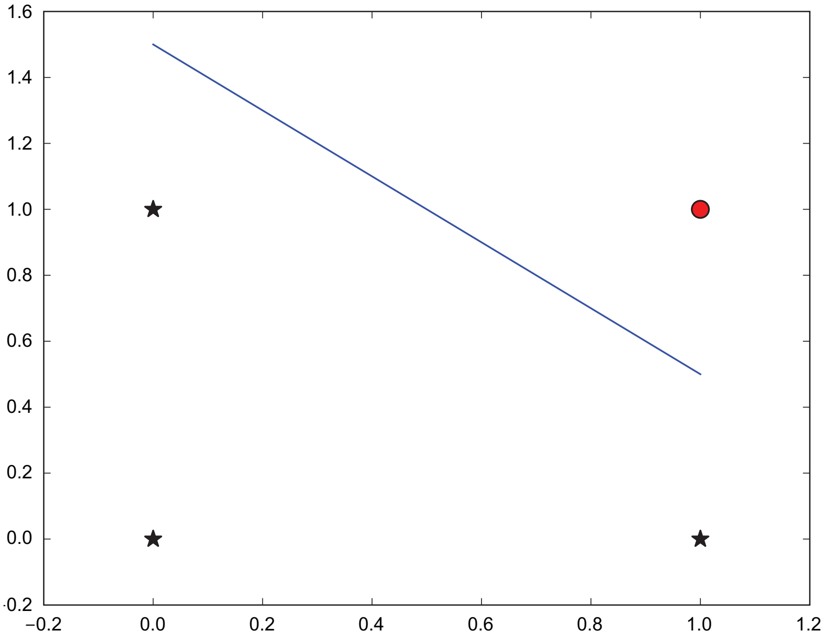


Figure 6.6 The projection of our separating plane onto the viewing plane (*y*=0). All points to the top right of the figure satisfy the constraint $W \cdot x > 0$, whereas points to the bottom left of the line satisfy the constraint $W \cdot x < 0$.

6.4 Multilayer perceptrons

In the previous sections we looked at deep learning from a very high level, and you started to understand the basics around neural networks, specifically, a single unit of a neural network known as a perceptron. We also showed that the basic form of the perceptron is equivalent to a linear model.

In order to perform non-linear separation, we can keep the simple threshold activation function and increase the complexity of the network architecture to create so-called multi-layer feed-forward networks. These are networks where perceptrons are organized in layers, with the input of a layer being provided by a previous layer and the output of this layer acting as an input to the next. Feed forward comes from the fact that data flows only from the inputs to the outputs of the network and not in the opposite direction, that is, no cycles. Figure 6.7 provides a graphical summary of this concept, extending the notation that we used in figure 6.3.

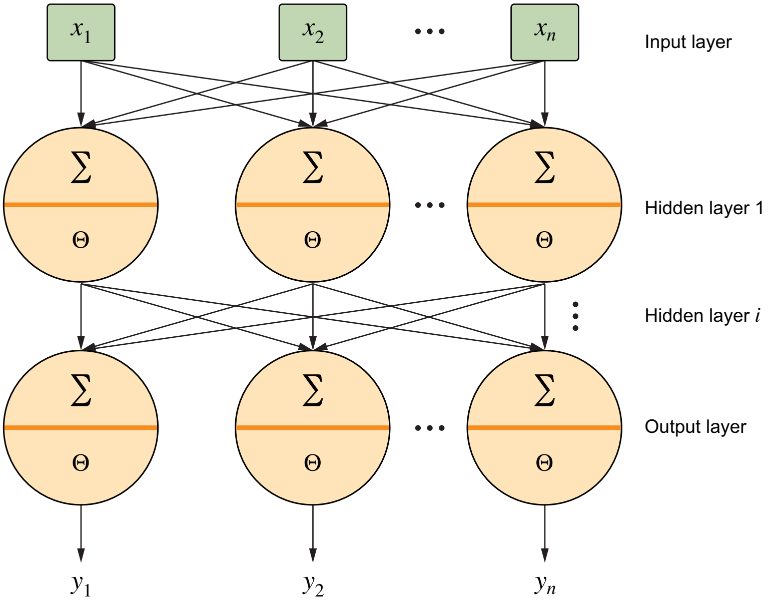


Figure 6.7 A multilayer perceptron. Read from top to bottom, it consists of an input layer (vector of values *X*), a number of hidden layers, and an output layer that returns the vector *Y*.

In the spirit of demonstrating non-linearity, let's consider a very small example that would fail if we presented it to a perceptron, namely, the XOR function. This example is taken from Minsky and Papert's 1969 book, *Perceptrons: An Introduction to Computational Geometry*.[1] We'll then consider how a two-layer perceptron can be used to

[1] Marvin Minsky and Seymour Papert, *Perceptrons: An Introduction to Computational Geometry* (Boston, MA: MIT Press, 1969).

approximate this function and discuss the back propagation algorithm used to train such a network. Recall that the XOR function functions as provided by table 6.2.

Table 6.2 Input and output values for the XOR function. Outputs a 1 if either x_1 or x_2 is set to 1 but a 0 of they are both set to 1 (or both to set to 0).

x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

If we consider the XOR function graphically, using the same conventions as in figure 6.5, we obtain figure 6.8.

As you can see from figure 6.8, the output from the XOR function is not linearly separable in two-dimensional space; there exists no single hyperplane that can separate positive and negative classes perfectly. Try to draw a line anywhere on this graph

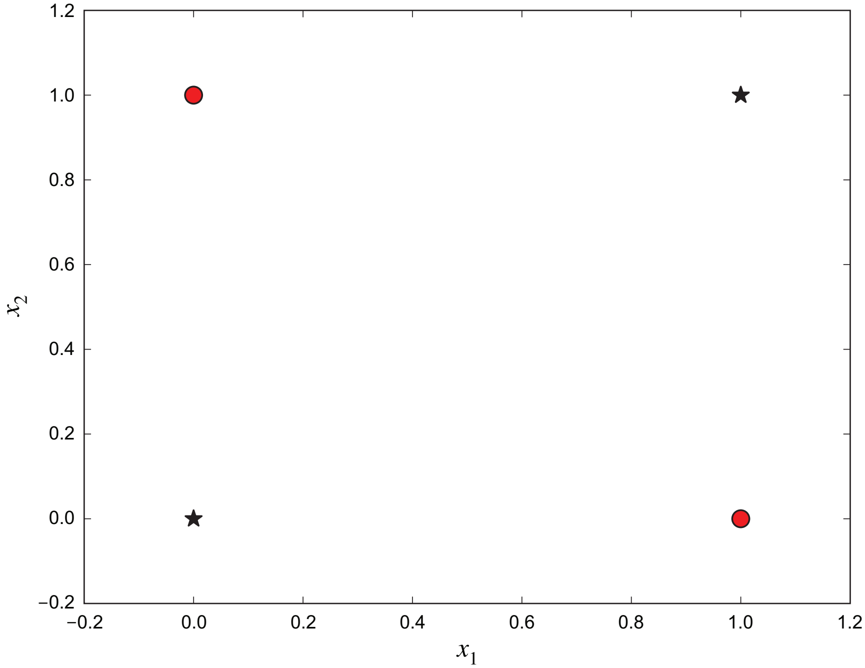


Figure 6.8 Graphical representation of the XOR function. Positive classes are specified with red circles, whereas negative classes are specified with black stars. No single hyperplane can separate these data points into two sets, and so we say that the data set is not linearly separable.

with all positive classes on one side of the line and all negative classes on the opposite side of the line and you will fail! We could, however, separate these data points if we had more than one hyperplane and a way to combine them. So let's do that! This is equivalent to creating a network with a single hidden layer and a final, combined-output layer. Consider figure 6.9, which shows such a network graphically.

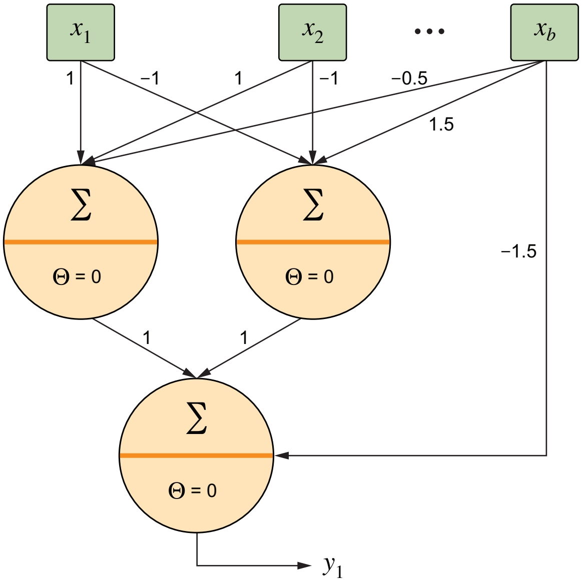


Figure 6.9 A two-layer perceptron can separate a non-linearly separable function (XOR). Values on the connections demonstrate the weight of those connections. The introduction of the bias term implies that the activation threshold for our neurons is at 0. Conceptually, the two hidden neurons correspond to two hyperplanes. The final combining perceptron is equivalent to a logical AND on the output of the two hidden neurons. This can be thought of as picking out the areas of the (x1,x2) plane for which the two hidden neurons activate together.

What you see here is a two-layer hidden network, with two inputs and a single output. Connected to each hidden node and the output node is a bias term. Remember from earlier that the bias itself will always equal 1; only the weights will change. This allows the both the activation profiles and the offsets of the nodes to be learned during training. Spend a little time convincing yourself that this does indeed work.

Each hidden node creates a single hyperplane, as with our single perceptron case from figure 6.6, and these are brought together with a final perceptron. This final perceptron is simply an AND gate when two conditions are observed. The first is that the input in x1,x2 space is above the blue line given in figure 6.10. The second is that the input is below the red line, also given in figure 6.10. As such, this two-layer perceptron carves out the diagonal across the space that includes both positive examples from the training data but none of the negative examples. It has successfully separated a non-linearly separable dataset.

In this section we investigated the application of neural networks to non-linearly separable datasets. Using the XOR function as an example, we showed that it's possible to create a neural network by hand that separates a non-linearly separable set and provide intuition as to how this works geometrically. We're missing an important step, however! It's important to be able to automatically learn the weights for a network

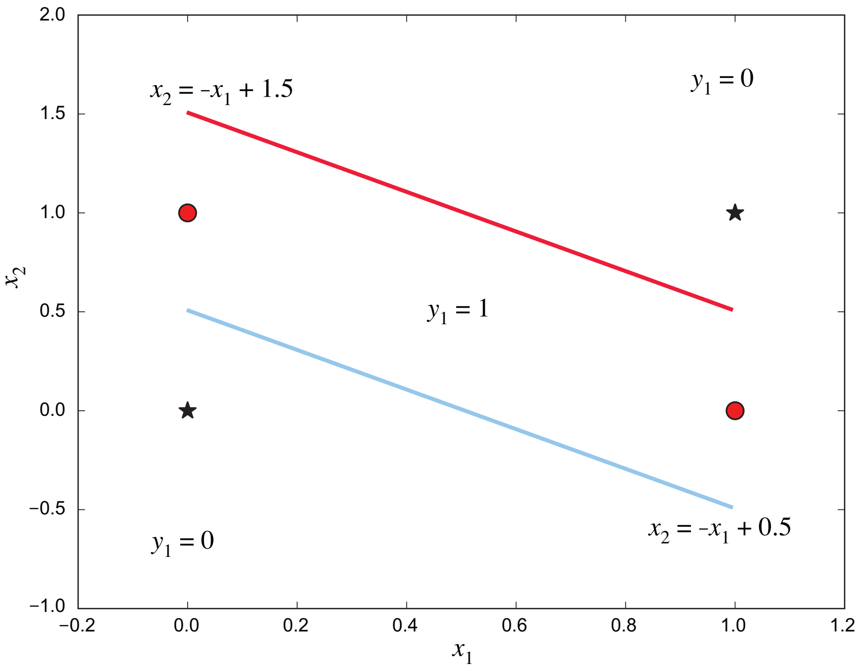


Figure 6.10 **Separating a non-linearly separable dataset using the neural network given in figure 6.9. You should notice that the neuron to the left of figure 6.9 when intersecting with the viewing plane creates the bottom line, whereas the right-most neuron creates the top line. The left-most neuron fires an output 1 only when the input is above the bottom line, whereas the right-most neuron fires an output 1 only when the input is below the top line. The final combining neuron fires y_1=1 only when both of these constraints are satisfied. Thus, the network outputs 1 only when the data points are in the narrow corridor between the bottom and top lines.**

given a training dataset. These weights can then be used to classify and predict data beyond the original inputs. This is in the subject of the next section.

6.4.1 Training using backpropagation

In the previous examples we used the step function as our neuron activation function. Unfortunately, coming up with an automated method to train such a network is difficult. This is because the step function doesn't allow us to encode uncertainly in any form—the thresholds are hard.

It would be more appropriate if we could use a function that approximates the step function but is more gradual. In this way, a small change to one of the weights within the network will make a small change to the operation of the entire network. This is indeed what we'll do. Recall the logistic function from chapter 4. Instead of using the step function, we'll now replace this with a more general activation function. In the next section we'll briefly introduce common activation functions before explaining how the choice of activation function will help us derive a training algorithm.

6.4.2 Activation functions

Let's take a few moments to look at some possible activation functions for our perceptron. We've already seen the simplest case—a hard threshold around 0. With an offset of zero, this yields the output profile as given by figure 6.3. But what else could we do? Figure 6.11 provides the activation profiles of several other functions; their definitions follow.

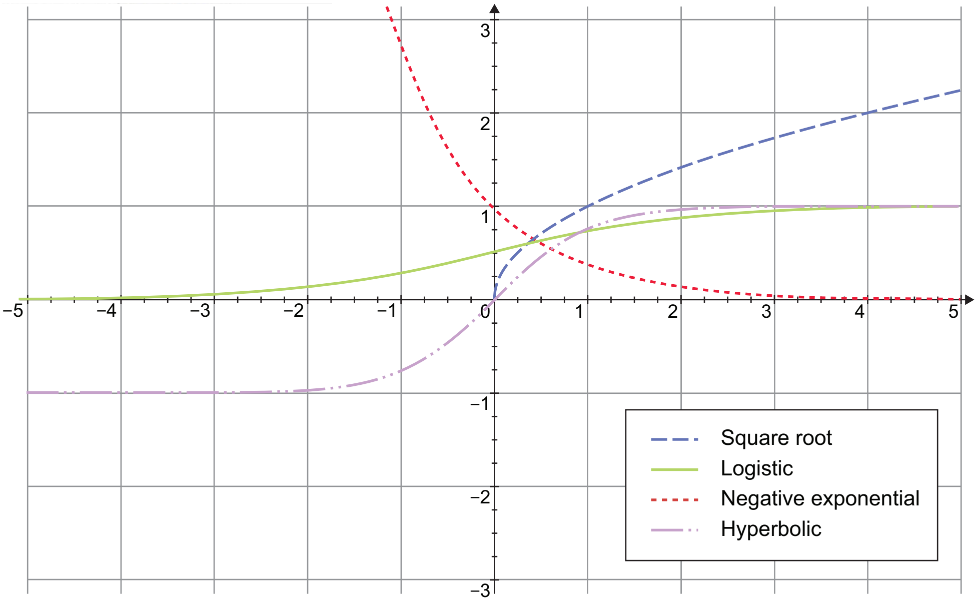


Figure 6.11 Output profiles for several activation functions over the same range as provided in figure 6.3. Activation profiles demonstrated are square root, logistic, negative exponential, and hyperbolic.

- *Square root*—Defined as $y = \sqrt{x}$, domain [0,∞], range [0,∞].
- *Logistic*—Defined as $1/(1 + e^{-x})$, domain defined for [−∞ ,∞], range [0,1].
- *Negative exponential*—Given by e^{-x}, domain [−∞,∞] range [0,∞].
- *Hyperbolic (tanh)*—Defined as $(e^x - e^{-x})/(e^x + e^{-x})$. Note that this is equivalent to logistic with its output transformed to a different range, domain [−∞,∞] range [−1,1].

In general, the use of such activation functions enables us to create and train multilayer neural networks that approximate a much larger class of functions. The most important property of the activation function is that it is differentiable. You'll see why in the following section.

Logistic regression, the perceptron, and the Generalized Linear Model

Think back to chapter 4 where we introduced the concept of logistic regression. There we identified that a linear response model would be unsuitable for probability estimation and instead modified the logistic response to curve to create a more appropriate response. From this starting point we derived that the log-odds are linear in the combination of weights and input variables and applied this to a classification problem.

In this chapter, we started from a basic biological concept and built a computational formalism that captures this. We've not discussed probability at all but started from a standpoint of activation within a neuron. Intuitively we've extended this into a more general concept and reached the same equation, namely,

$$y = \frac{1}{1 + e^{-(w_0 + w_1 \cdot x_1 + \ldots + w_n \cdot x_n)}}$$

In fact, what we've encountered here is a more general class of problem known as generalized linear models (GLM).[a] In this class of models, a linear model (w_0+w_1*x_1+…+w_n*x_n) is related to the output variable, y, by a link function, which in this case is the logistic function.

$$1/1 + e^{-x}$$

This equivalence of algorithms and concepts is common in machine learning, and you've seen it already within the pages of this book. Just think back to section 2.5 where we discussed the equivalence of expectation maximization (EM) over a Gaussian mixture model with tied and coupled covariance, and the vanilla k-means algorithm. The usual reason for this is that multiple researchers have started from different points within the research and discovered equivalence through the extension of basic building blocks.

[a] Peter McCullagh and John A. Nelder, *Generalized Linear Models* (London: Chapman and Hall/CRC) 1989.

6.4.3 *Intuition behind backpropagation*

To provide the intuition behind backpropagation we're going to work with the same example as previously, the XOR function, but we're now going to try to learn the weights rather than specify them by hand. Note also that from now on, the activation function will be assumed to be sigmoid (logistic). Figure 6.12 provides the graphical overview of our new network.

Our job is to learn $W(a, b), \forall a, b$ using a specified training dataset. More specifically, can we come up with an algorithm that minimizes the error (squared difference between the expected and actual values of y_1) over that dataset, when that dataset's inputs are applied to the network?

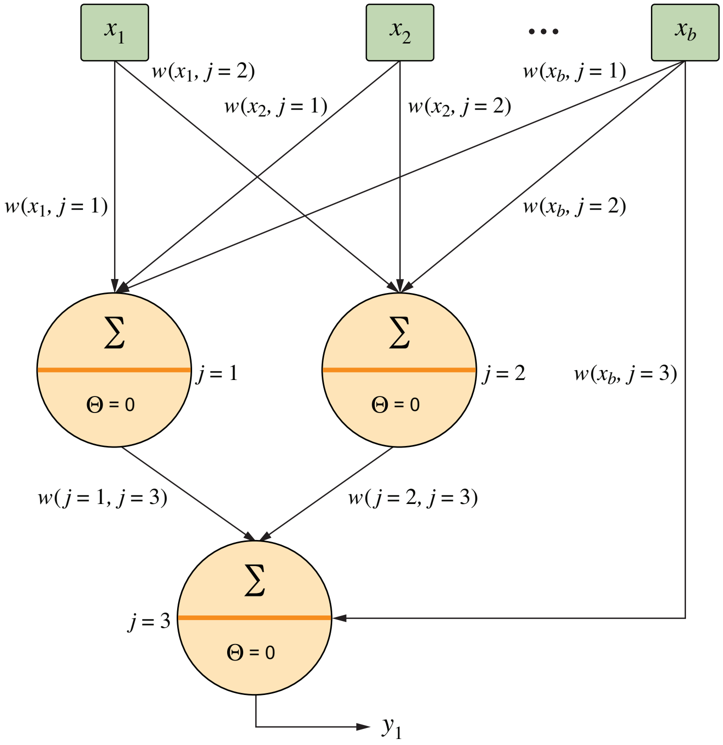


Figure 6.12 Overview of our backpropagation example. Given a set of inputs (x_1,x_2) and target variables (y_1) that follow the XOR function over x_1 and x_2, can we learn the values of W that minimize the squared difference between the training values and the network output? In this example we use the logistic activation function, namely, $\theta = 1/\left(1 + e^{-w\left(_0 + \sum w_i x_i\right)}\right)$.

One way to do this is through an algorithm known as backpropagation. This algorithm operates broadly as follows. First, we initialize the all weights to random values, and then we pass a single training data item throughout the network. We calculate the error at the output and *backpropagate* this error through the network, hence the name! Each weight in the network is changed in a direction that corresponds to that which minimizes the error of network. This continues until a termination condition is met, for example, a number of iterations are hit or the network has converged.

6.4.4 Backpropagation theory

To ease understanding, the update rule for backpropagation can be considered in two parts: updating the weights leading to output neurons and updating the weights leading to hidden neurons. Both are logically identical, but the mathematics for the latter is a little trickier. Because of this, we'll discuss only the former here to give you a taste of backpropagation. See the seminal *Nature* paper[1] if you want to understand the full form.

[1] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J Williams, "Learning representations by back-propagating errors," *Nature* 323 (October 1986): 533–36.

The first thing to note about training is that we're interested in how the error at the output changes with respect to the change in a single weight value. Only through this value can we move the weight in a direction that minimizes the output error. Let's start by working out the *partial* derivative of the error with respect to a particular input weight in the layer below. That is, we assume that all other weights remain constant. To do this, we'll use the chain rule:

$$\frac{\delta E}{\delta w(i,j)} =$$

❶ $\dfrac{\delta E}{\delta o(j)} \times$

❷ $\dfrac{\delta o(j)}{\delta n(j)} \times$

❸ $\dfrac{\delta n(j)}{\delta w(i,j)}$

In plain terms, the rate of change of output error is linked to the weight, through the rate of change of the following:

❶ The error given the output of the activation function
❷ The output of the activation function given the weighted sum of its inputs
❸ The weighted sum of its inputs and an individual weight

If you remember from earlier, the logistic activation function is used because it makes training tractable. The main reason for this is that the function is differentiable. You should now understand why this is a requirement. Term ❷ of the equation is equivalent to the derivative of the activation function. This can be written as

$$\frac{\delta o(j)}{\delta n(j)} = \frac{\delta}{\delta n(j)}\theta(n(j)) = \theta(n(j))(1-\theta(n_j))$$

that is, the rate of change of output of our activation function can be written in terms of the activation function itself! If we can compute ❶ and ❸, we'll know the direction in which to move any particular weight in order to minimize the error in the output. It turns out that this is indeed possible. Term ❸ can be differentiated directly as

$$\frac{\delta n(j)}{\delta w(i,j)} = x_i$$

That is, the rate of change of the input to the activation function with respect to a particular weight linking I and J is given only by the value of x_I itself. Because we looked only at the output layer, determining the differential of the error given the output ❶ is easy if we can just draw on the concept of error directly:

$$\frac{\delta E}{\delta o(j)} = \frac{\delta}{\delta o(j)}(o_{expected} - o(j))^2 = 2(o(j) - o_{expected})$$

We can now express a complete weight update rule for a weight leading to an output node:

$$-\alpha x_i 2(o(j) - o_{expected})\theta(n(j))(1-\theta(n_j))$$

Thus, we update the weight depending on the entering value corresponding to that weight, the difference in the output and the expected value, and output of the derivative of the activation function given all input and weights. Note that we add a negative sign and an alpha term. The former ensures that we move in the direction of negative error, and the latter specifies how fast we move in that direction.

This should give you a feeling for how the weights are updated feeding into the output layer, and the inner layer update functions follow much the same logic. But we must use the chain rule to find out the contribution of the output value at that inner node to the overall error of the network, that is, we must know the rate of change of inputs/outputs on the path leading from the node in question to an output node. Only then can the rate of change of output error be assessed for a delta change in the weight of an inner node. This gives rise to the full form of backpropagation.[1]

6.4.5 MLNN in scikit-learn

Given that you now understand the multilayer perceptron and the theory behind training using backpropagation, let's return to Python to code up an example. Because there's no implementation of MLPs in scikit-learn, we're going to use PyBrain.[2] PyBrain focuses specifically on building and training neural networks. The following listing provides you with the first snippet of code required build a neural network equivalent to the one presented in figure 6.12. Please refer to the full code that is distributed with this book for the associated imports required to run this code.

Listing 6.6 Building a multilayer perceptron using PyBrain

```
#Create network modules
net = FeedForwardNetwork()
inl = LinearLayer(2)
hidl = SigmoidLayer(2)
outl = LinearLayer(1)
b = BiasUnit()
```

In listing 6.6 we first create a `FeedForwardNetwork` object. We also create an input layer (`inl`), and output layer (`outl`), and a hidden layer (`hidl`) of neurons. Note that our input and output layers use the vanilla activation function (threshold at 0), whereas our hidden layer uses the sigmoid activation function for reasons of training, as we discussed earlier. Finally, we create a bias unit. We don't quite have a neural network yet, because we haven't connected the layers. That's what we do in the next listing.

[1] Rumelhart et al., "Learning representations by back-propagating errors," 533–36.

[2] Tom Schaul, et al., "PyBrain," *Journal of Machine Learning Research* 11 (2010): 743–46.

Listing 6.7 Building a multilayer perceptron using PyBrain (2)

```
#Create connections
in_to_h = FullConnection(inl, hidl)
h_to_out = FullConnection(hidl, outl)
bias_to_h = FullConnection(b,hidl)
bias_to_out = FullConnection(b,outl)

#Add modules to net
net.addInputModule(inl)
net.addModule(hidl);
net.addModule(b)
net.addOutputModule(outl)

#Add connections to net and sort
net.addConnection(in_to_h)
net.addConnection(h_to_out)
net.addConnection(bias_to_h)
net.addConnection(bias_to_out)
net.sortModules()
```

In listing 6.7 we now create connection objects and add our previously created neurons (modules) and their connections to the `FeedForwardNetwork` object. Calling `sortModules()` completes the instantiation of the network.

Before continuing, let's take a moment to delve into the `FullConnection` object. Here we create four instances of the object to pass to the network object. The signature of these constructors takes two layers, and internally the object creates a connection between every neuron in the layer of the first parameters and every neuron in the layer of the second. The final method sorts the modules within the `FeedForwardNetwork` object and performs some internal initialization.

Now that we have a neural network equivalent to figure 6.11, we need to learn its weights! To do this we need some data. The next listing provides the code to do this, and much of it is reproduced from the PyBrain documentation.[1]

Listing 6.8 Training our neural network

```
d = [(0,0),
     (0,1),
     (1,0),
     (1,1)]

#target class
c = [0,1,1,0]

data_set = SupervisedDataSet(2, 1) # 2 inputs, 1 output

random.seed()
for i in xrange(1000):
    r = random.randint(0,3)
    data_set.addSample(d[r],c[r])
```

Create a data set and associated targets that reflect the XOR function.

Create empty PyBrain SupervisedDataSet object.

Randomly sample the four data points 1000 times and add to the training set.

[1] PyBrain Quickstart, November 12, 2009, http://pybrain.org/docs/index.html#quickstart (accessed December 22, 2015).

```
backprop_trainer = \
BackpropTrainer(net, data_set, learningrate=0.1)

for i in xrange(50):
    err = backprop_trainer.train()
    print "Iter. %d, err.: %.5f" % (i, err)
```

Create a new backpropagation trainer object with the network, dataset, and learning rate.

Perform backpropagation 50 times using the same 1000 data points. Print the error after every iteration.

As you now know from section 6.4.4, backpropagation traverses the weight space in order to reach a minima in the error between the output terms and the expected output. Every call to `train()` causes the weights to be updated so that the neural network better represents the function generating the data. This means we're probably going to need a reasonable amount of data (for our XOR example, four data points is not going to cut it!) for each call to `train()`. To address this problem we'll generate many data points drawn from the XOR distribution and use these to train our network using backpropagation. As you'll see, subsequent calls to `train()` successfully decrease the error between the network output and the specified target. The exact number of iterations required to find the global minima will depend on many factors, one of which is the learning rate. This controls how fast the weights are updated at each training interval. Smaller rates will take longer to converge, that is, find the global minima, but they're less likely to result in suboptimal models. Let's take a quick look at the output generated by listing 6.8 and use this to illustrate this concept:

```
Iteration 0, error: 0.1824
Iteration 1, error: 0.1399
Iteration 2, error: 0.1384
Iteration 3, error: 0.1406
Iteration 4, error: 0.1264
Iteration 5, error: 0.1333
Iteration 6, error: 0.1398
Iteration 7, error: 0.1374
Iteration 8, error: 0.1317
Iteration 9, error: 0.1332
…
```

As you'll see, successive calls reduce the error of the network. We know that at least one solution does exist, but backpropagation is not guaranteed to find this. Under certain circumstances, the error will decrease and won't be able to improve any further, or it may bounce between suboptimal (or local) solutions. The algorithm may get stuck and not be able to find the global minima, that is, the lowest possible error.

Because this outcome depends on the starting values of the weights, we're not able to say if your example will converge quickly, so try running this a few times. Also try experimenting with the learning rate from listing 6.8. How big can you make the rate before the algorithm gets caught in local solutions most of the time? In practice, the choice of training rate is always a trade-off between finding suboptimal solutions and speed, so you want to choose the largest rate that gives you the correct answer. Experiment with this until you're left with a network that has converged with an error of zero.

6.4.6 A learned MLP

In the previous example we created an MLP using the PyBrain package and trained a multi-layer perceptron to mimic the XOR function. Provided your error in the previous output reached zero, you should be able to follow this section with your own model! First, let's interrogate our model to obtain the weights of the network, corresponding to figure 6.11. The following code shows you how.

Listing 6.9 Obtaining the weights of your trained neural network

```
#print net.params
print "[w(x_1,j=1),w(x_2,j=1),w(x_1,j=2),w(x_2,j=2)]: " + str(in_to_h.params)
print "[w(j=1,j=3),w(j=2,j=3)]: "+str(h_to_out.params)
print "[w(x_b,j=1),w(x_b,j=2)]: "+str(bias_to_h.params)
print "[w(x_b,j=3)]:" +str(bias_to_out.params)

> [w(x_1,j=1),w(x_2,j=1),w(x_1,j=2),w(x_2,j=2)]: [-2.32590226  2.25416963 -
     2.74926055  2.64570441]
> [w(j=1,j=3),w(j=2,j=3)]: [-2.57370943  2.66864851]
> [w(x_b,j=1),w(x_b,j=2)]: [ 1.29021983 -1.82249033]
> [w(x_b,j=3)]:[ 1.6469595]
```

The output resulting from executing listing 6.9 provides my trained output for a neuron. Your results may vary, however. The important thing is that the behavior of the network is correct. You can check this by activating the network with input and checking that the output is as expected. Observe the next listing.

Listing 6.10 Activating your neural network

```
print "Activating 0,0. Output: " + str(net.activate([0,0]))
print "Activating 0,1. Output: " + str(net.activate([0,1]))
print "Activating 1,0. Output: " + str(net.activate([1,0]))
print "Activating 1,1. Output: " + str(net.activate([1,1]))

> Activating 0,0. Output: [ -1.33226763e-15]
> Activating 0,1. Output: [ 1.]
> Activating 1,0. Output: [ 1.]
> Activating 1,1. Output: [  1.55431223e-15]
```

In listing 6.10 you can see that the output of our trained network is very close to 1 for those patterns that should result in a positive value. Conversely, the output is very close to 0 for those patterns that should result in a negative output. In general, positive testing samples should have outputs greater than 0.5, and negative testing samples should provide outputs less than 0.5. In order to ensure that you fully understand this network, try modifying the input values and tracing them through the network in the supporting content spreadsheet distributed with this book.

6.5 Going deeper: from multilayer neural networks to deep learning

In many areas of research, progress is made in fits and starts. Areas can go stale for a period and then experience a rapid rush, usually sparked by a particular advance or discovery. This pattern is no different in the field of neural networks, and we're lucky to be right in middle of some really exciting advances, most of which have been grouped under the umbrella of deep learning. I'd like to share a few of these with you now before delving into the simplest example of a deep network that we can. So why did neural networks become hot again? Well, it is a bit of a perfect storm.

First, there's more data available than ever before. The big internet giants have access to a huge repository of image data that can be leveraged to do interesting things. One example you may have heard of is Google's 2012 paper where they trained a nine-layer network with 10 million images downloaded from the internet[1] to recognize intermediate representations without labeling, the most publicized being a cat face! This lends some weight to the hypothesis that more data beats a cleverer algorithm.[2] Such an achievement would not have been possible only a few years before.

The second advance is a leap in theoretical knowledge. It wasn't until recent advances by Geoffrey Hinton and collaborators that the community understood that deep networks could be trained effectively by treating each layer as a Restricted Boltzmann Machine (RBM).[3,4] Indeed, many deep learning networks are now constructed by stacking RBMs—more on these in a moment. Yann Le Cun, Yoshua Bengio, and others have made many further theoretical advances in this field, and I refer you to a review of their work to gain better insight.[5]

6.5.1 Restricted Boltzmann Machines

In this section we're going to look at Restricted Boltzmann Machines (RBM). More specifically, we'll look at a specific flavor of RBM called Bernoulli RBM (BRBM). We'll get to why these are a special case of the RBM in a moment. In general, RBMs are mentioned in the context of deep learning, because they're good feature learners. Because of this, they can be used in a deeper network to learn feature representations, with their output used as input to the other RBMs or a multilayer perceptron (MLP). Think back to section 6.1 and our example of car recommendation. So far, we've spent quite some time covering MLPs in general; we must now uncover the automatic feature-extraction aspect of deep learning!

[1] Quoc V. Le, et al., "Building high-level features using large scale unsupervised learning," ICML 2012: 29th International Conference on Machine Learning (Edinburgh: ICML, 2012): 1.

[2] Pedro Domingos, "A Few Useful Things to Know about Machine Learning," *Communications of the ACM*, October 1, 2012: 78–87.

[3] Miguel A. Carreira-Perpiñán and Geoffrey Hinton, "On Contrastive Divergence Learning," (NJ: Society for Artificial Intelligence and Statistics, 2005), 33–40.

[4] G. E. Hinton and R. R. Salakhutdinov, "Reducing the Dimensionality of Data with Neural Networks," *Science*, July 28, 2006: 504–507.

[5] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, "Deep learning," *Nature* 521 (May 2015): 436–44.

To do this, we're also going to use an example from the scikit-learn documentation.[1] This example uses a BRBM to extract features from the scikit-learn digits dataset and then uses logistic regression to classify the data items with the learned features. After working through this example, we'll touch on how you might go about making deeper networks and delving further into the burgeoning area of deep learning. Before we get started on this, let's make sure that you understand the basics—so what is a BRBM?

6.5.2 The Bernoulli Restricted Boltzmann Machine

In general, an RBM is a bipartite graph where nodes within each partition are fully connected to nodes within the other partition. The restricted aspect comes from the fact that the visible nodes may only be connected to hidden nodes, and vice versa. The Bernoulli RBM restricts each node further to be binary. Figure 6.13 provides a graphical overview of an RBM.

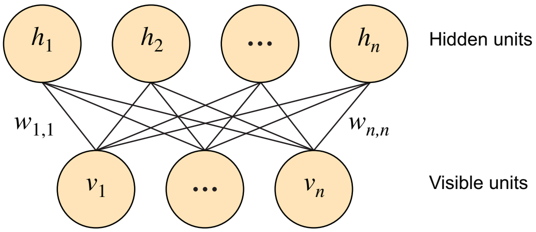


Figure 6.13 Graphical overview of a Restricted Boltzmann Machine. An RBM is a bipartite graph between hidden and visible units, with each element of each partition fully connected to other units in the opposing partition. We'll use H to refer to the vector composed of the hidden units and V to refer to the vector composed of the visible units. Note that each node may have an associated bias weight, which are not represented here for simplicity.

In general, the number of visible units is defined by the problem; for a binary classification problem you may have two. By increasing the number of hidden values, you increase the ability of the RBM to model complex functions, but this comes at the price of overfitting. Hinton[2] provides a recipe for choosing the number of hidden units dependent on the complexity of your data and the number of training samples.

Similarly to MLPs with a logistic activation function, the probability of a particular visible unit firing given the value of the hidden variables is

$$P(v_i = 1|H) = \theta(\sum^j w_{ij} h_j + b_i)$$

[1] scikit-learn, Restricted Boltzmann Machine features for digit classification, January 01, 2014, http://scikit-learn.org/stable/auto_examples/neural_networks/plot_rbm_logistic_classification.html#example-neural-networks-plot-rbm-logistic-classification-py (accessed December 22, 2015).

[2] Geoffrey Hinton, "A Practical Guide to Training Restricted Boltzmann Machines" in *Neural Networks: Tricks of the Trade*, edited by Grégoire Montavon, G. B. Orr, and K. R. Muller (NY: University of Toronto, 2012) 599–619.

where θ is the logistic function. The probabilities of the hidden values are given similarly. RBMs are trained as energy-based models that measure the agreement between the hidden states and the visible ones.

$$E(V, H) = -\left(\sum^{i} \sum^{j} (w_i, j, v_i, h_j) + \sum^{i} b_i\, v_i + \sum^{j}, c_j\, h_j\right)$$

This value decreases where there is more agreement between the hidden nodes and the visible ones; thus, decreasing the energy results in more acceptable configurations given the training data. We can relate this energy function to a probability as follows

$$P(V, H) = e^{-E(V, H)} / Z$$

where Z is a normalization factor equal to $(\sum v, h)\ e^{-E(v, h)}$. That is, we evaluate the current weight selection by normalizing over all permutations of hidden and visible state possibilities. In order to train the RBM, we need to maximize the likelihood $P(V)$ or similarly the log likelihood. Taking the log of $P(V,H)$ we can write

$$\mathrm{Log}(P(V, H)) = \mathrm{Log}(e^{-(V, H)} - \mathrm{Log}\, Z(W))$$
$$\mathrm{Log}(P(V)) = \sum^{H} \mathrm{Log}(e^{-(V, H)} - \mathrm{Log}\, Z(W))$$

Thus, for a given visible datum we can maximize its likelihood by choosing weights that maximize agreement over possible hidden variables (first term) and minimizing the normalization factor $\sum v, h\ \ e^{-E(w, v, h)}$, which evaluates the weight selection over all possible visible and hidden values. In a sense you can think of the former as trying to get the model to fire when it should and the latter providing this optimization with the context of the training data. Maximizing the first selects weights that over all possible hidden values should be maximally likely to generate the visible value, whereas minimizing the second ensures that the model remains representative of the data.

If you remember from earlier, we need to differentiate the previous function with respect to the weight in order to find the direction in which we should move the weight to maximize the likelihood. If you follow the theory,[1] you'll see that it's possible to compute the derivative of the first term exactly, but for the second we need to rely on an iterative algorithm to estimate it. In short, it's trickier to perform the training, but in practice it's possible. You'll see how in the next section!

6.5.3 RBMS in action

In this section we'll use a modified version of the logistic classification problem presented in the scikit-learn documentation.[2] Full credit must be given to Dauphin,

[1] Kevin Swersky, Bo Chen, Ben Marlin, and N de Freitas, "A Tutorial on Stochastic Approximation Algorithms for Training Restricted Boltzmann Machines and Deep Belief Nets," Information Theory and Applications Workshop (ITA), 2010 (San Diego: ITA, 2010) 1–10.

[2] scikit-learn, "Restricted Boltzmann Machine features for digit classification," http://scikit-learn.org/stable/auto_examples/neural_networks/plot_rbm_logistic_classification.html#example-neural-networks-plot-rbm-logistic-classification-py (accessed December 22, 2015).

Niculae, and Synnaeve for this illustrative example, and we won't stray far from their material here. As in previous examples, we'll omit the import block and concentrate on the code. The full listing can be found in the supporting content.

Listing 6.11 Creating your dataset

```
digits = datasets.load_digits()
X = np.asarray(digits.data, 'float32')
X, Y = nudge_dataset(X, digits.target)
X = (X - np.min(X, 0)) / (np.max(X, 0) + 0.0001)  # 0-1 scaling
X_train, X_test, Y_train, Y_test = train_test_split(X,
     Y,test_size=0.2,random_state=0)
```

The first thing we do is to load in the dataset, but we actually do more than this. From the original dataset, we generate further artificial samples by nudging the dataset with linear shifts of one pixel, normalizing each so that each pixel value is between 0 and 1. So, for every labeled image, a further four images are generated—shifted up, down, right, and left respectively—each with the same label, that is, which number the image represents. This allows training to learn better representations of the data using such a small data set, specifically representations that are less dependent on the script being centralized within the image. This is achieved using the nudge_dataset function, defined in the following listing.

Listing 6.12 Generating artificial data

```
def nudge_dataset(X, Y):
    """
    This produces a dataset 5 times bigger than the original one,
    by moving the 8x8 images in X around by 1px to left, right, down, up
    """
    direction_vectors = [[[0, 1, 0],[0, 0, 0],[0, 0, 0]],
                         [[0, 0, 0],[1, 0, 0],[0, 0, 0]],
                         [[0, 0, 0],[0, 0, 1],[0, 0, 0]],
                         [[0, 0, 0],[0, 0, 0],[0, 1, 0]]]
    shift = \
        lambda x, w: convolve(x.reshape((8, 8)), mode='constant',\
        weights=w).ravel()
    X = np.concatenate([X] + \
        [np.apply_along_axis(shift, 1, X, vector) for vector in \
        direction_vectors])
    Y = np.concatenate([Y for _ in range(5)], axis=0)
    return X, Y
```

Given this data, it's now simple to create a decision pipeline consisting of an RBM, followed by logistic regression. The next listing presents the code to both set up this pipeline and train the model.

Listing 6.13 Setting up and training an RBM/LR pipeline

```
# Models we will use
logistic = linear_model.LogisticRegression()
rbm = BernoulliRBM(random_state=0, verbose=True)

classifier = Pipeline(steps=[('rbm', rbm), ('logistic', logistic)])

###############################################################################
# Training

# Hyper-parameters. These were set by cross-validation,
# using a GridSearchCV. Here we are not performing cross-validation to
# save time.
rbm.learning_rate = 0.06
rbm.n_iter = 20
# More components tend to give better prediction performance, but larger
# fitting time
rbm.n_components = 100
logistic.C = 6000.0

# Training RBM-Logistic Pipeline
classifier.fit(X_train, Y_train)

# Training Logistic regression
logistic_classifier = linear_model.LogisticRegression(C=100.0)
logistic_classifier.fit(X_train, Y_train)
```

This code is taken directly from Scikit-learn,[1] and there are some important things to note. The hyper-parameters, that is, the parameters of the RBM, have been selected specially for the dataset in order to provide an illustrative example that we can discuss. More details can be found in the original documentation.

You'll see that beyond this, the code does very little. A classifier pipeline is set up consisting of RBM followed by a logistic regression classifier, as well as a standalone logistic regression classifier for comparison. In a minute you'll see how these two approaches perform. The following listing provides the code to do this.

Listing 6.14 Evaluating the RBM/LR pipeline

```
print("Logistic regression using RBM features:\n%s\n" % (
    metrics.classification_report(
        Y_test,
        classifier.predict(X_test))))

print("Logistic regression using raw pixel features:\n%s\n" % (
    metrics.classification_report(
        Y_test,
        logistic_classifier.predict(X_test))))
```

The output of this provides a detailed summary of the two approaches, and you should see that the RBM/LR pipeline far outstrips the basic LR approach in precision,

[1] Scikit-learn, "Bernoulli RBM."

recall, and f1 score. But why is this? If we plot the hidden components of the RBM, we should start to understand why. The next listing provides the code to do this, and figure 6.14 provides a graphical overview of the hidden components of our RBM.

Listing 6.15 Representing the hidden units graphically

```
plt.figure(figsize=(4.2, 4))
for i, comp in enumerate(rbm.components_):
    #print(i)
    #print(comp)
    plt.subplot(10, 10, i + 1)
    plt.imshow(comp.reshape((8, 8)),
     cmap=plt.cm.gray_r,interpolation='nearest')
    plt.xticks(())
    plt.yticks(())

plt.suptitle('100 components extracted by RBM', fontsize=16)
plt.subplots_adjust(0.08, 0.02, 0.92, 0.85, 0.08, 0.23)
plt.show()
```

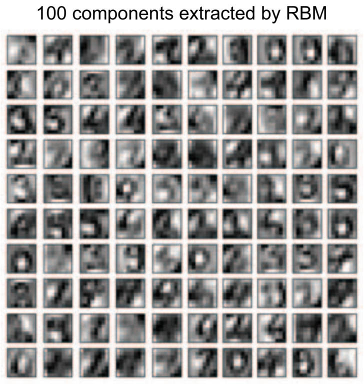


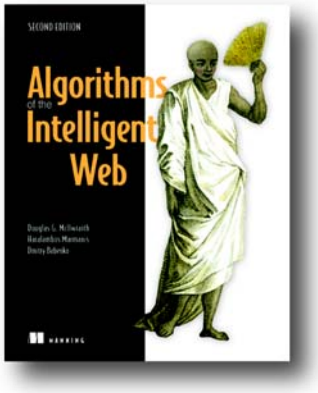
Figure 6.14 A graphical representation of the weights between the hidden and visible units in our RBM. Each square represents a single hidden unit, and the 64 grayscale values within represent the weights from that hidden value to all the visible units. In a sense this dictates how well that hidden variable is able to recognize images like the one presented.

In our RBM, we have 100 hidden nodes and 64 visible units, because this is the size of the images being used. Each square in figure 6.14 is a grayscale interpretation of the weights between that hidden component and each visible unit. In a sense, each hidden component can be thought of as recognizing the image as given previously. In the pipeline, the logistic regression model then uses the 100 activation probabilities ($P(h_i = 1|v = image)$ for each i) as its input; thus, instead of doing logistic regression over 64 raw pixels, it's performed over 100 inputs, each having a high value when the input looks close to the ones provided in figure 6.14. Going back to the first section in this chapter, you should now be able to see that we've created a network that has automatically learned some intermediate representation of the numbers, using an RBM. In essence, we've created a single layer of a deep network! Imagine now what could be achieved with deeper networks and multiple layers of RBMs to create more intermediate representations!

6.6 Conclusions

In this chapter

- We provided you a whistle-stop tour of neural networks and their relationship to deep learning. Starting with the simplest neural network model, the MCP model, we moved on to the perceptron and discussed its relationship with logistic regression.
- We found that it's not possible to represent non-linear functions using a single perceptron, but that it is possible if we create multilayer perceptrons (MLP).
- We discussed how MLPs are trained through backpropagation—and the adoption of differentiable activation functions—and provided you with an example whereby a non-linear function is learned using backpropagation in PyBrain.
- We discussed the recent advances in deep learning, specifically, building multiple layers of networks that can learn intermediate representations of the data.
- We concentrated on one such network known as a Restricted Boltzmann Machine, and we showed how you can construct the simplest deep network over the digits dataset, using a single RBM and a logistic regression classifier.



There's priceless insight trapped in the flood of data web users leave behind as they interact with your pages and applications. You can unlock those insights by using intelligent algorithms like the ones that have earned Facebook, Google, Twitter, and Microsoft a place among the giants of web data pattern extraction. Improved search, data classification, and other smart pattern matching techniques can give you an enormous advantage when you need to understand and interact with your users.

Algorithms of the Intelligent Web, Second Edition teaches the most important approaches to algorithmic web data analysis, enabling you to create your own machine learning applications that crunch, munge, and wrangle data collected from users, web applications, sensors, and website logs. In this totally-revised edition, you'll look at intelligent algorithms through the lens of machine learning, with code examples that show you how to extract value from their data. Key machine learning concepts are explained and introduced with code examples in Python's scikit-learn. This book guides you through the underlying machinery and intelligent algorithms to capture, store, and structure data streams coming from the web. You'll explore recommendation engines from the example of Netflix movie recommendations and dive into classification via statistical algorithms, neural networks, and deep learning. You'll also consider the ins and outs of ranking and how to test applications based on intelligent algorithms.

What's inside

- Machine learning for newbies
- Algorithms for IoT, public health, and other areas
- An introduction to deep learning and neural networks
- Clarifies how recommendation engines really work

Text mining and text analytics

Text mining is the science of extracting machine-usable information from text. Classic machine learning concentrates on the analysis of orderly or structured data, but such data represent only a small percentage of the information around us. The following chapter uses Python and the Natural Language Toolkit (NLTK) to demonstrate the classic transformations used to convert free text into structured data. The example project builds a system to classify Reddit posts into different categories. This categorization can then be used as structured information in additional projects.

Chapter 8 from *Introducing Data Science*
by Davy Cielen, Arno D. B. Meysman, and Mohamed Ali

Text mining and text analytics

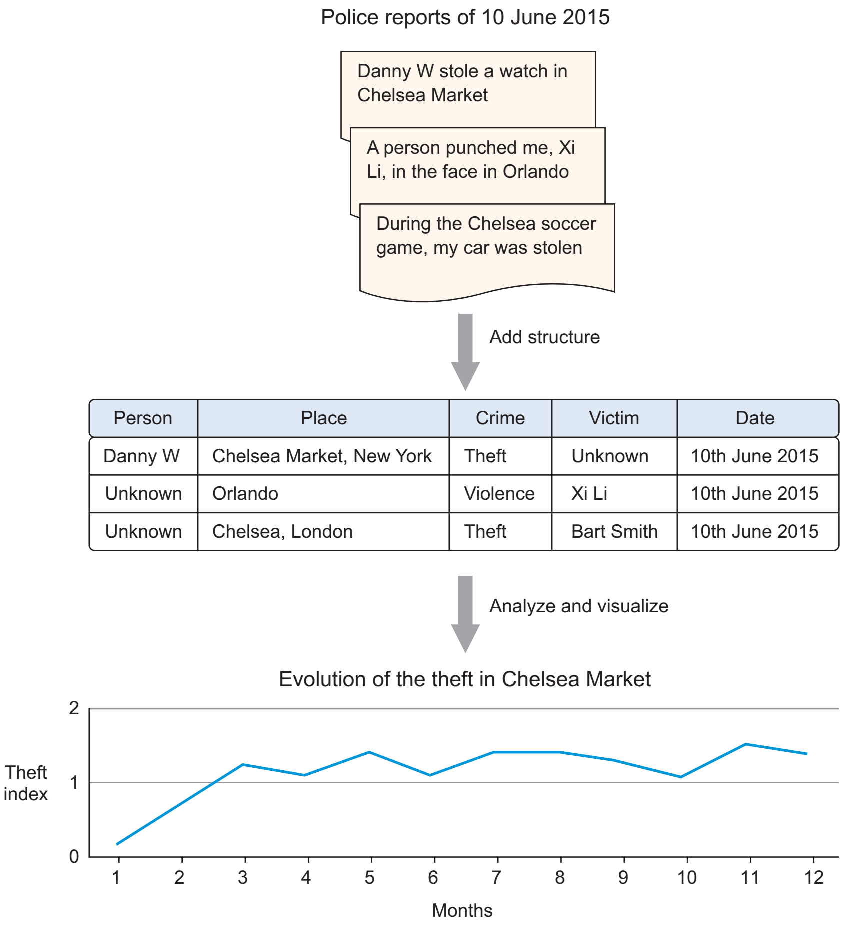
This chapter covers

- Understanding the importance of text mining
- Introducing the most important concepts in text mining
- Working through a text mining project

Most of the human recorded information in the world is in the form of written text. We all learn to read and write from infancy so we can express ourselves through writing and learn what others know, think, and feel. We use this skill all the time when reading or writing an email, a blog, text messages, or this book, so it's no wonder written language comes naturally to most of us. Businesses are convinced that much value can be found in the texts that people produce, and rightly so because they contain information on what those people like, dislike, what they know or would like to know, crave and desire, their current health or mood, and so much more. Many of these things can be relevant for companies or researchers, but no single person can read and interpret this tsunami of written material by themself. Once again, we need to turn to computers to do the job for us.

Sadly, however, the natural language doesn't come as "natural" to computers as it does to humans. Deriving meaning and filtering out the unimportant from the important is still something a human is better at than any machine. Luckily, data scientists can apply specific text mining and text analytics techniques to find the relevant information in heaps of text that would otherwise take them centuries to read themselves.

Text mining or *text analytics* is a discipline that combines language science and computer science with statistical and machine learning techniques. Text mining is used for analyzing texts and turning them into a more structured form. Then it takes this structured form and tries to derive insights from it. When analyzing crime from police reports, for example, text mining helps you recognize persons, places, and types of crimes from the reports. Then this new structure is used to gain insight into the evolution of crimes. See figure 8.1.



Police reports of 10 June 2015

Danny W stole a watch in Chelsea Market

A person punched me, Xi Li, in the face in Orlando

During the Chelsea soccer game, my car was stolen

Person	Place	Crime	Victim	Date
Danny W	Chelsea Market, New York	Theft	Unknown	10th June 2015
Unknown	Orlando	Violence	Xi Li	10th June 2015
Unknown	Chelsea, London	Theft	Bart Smith	10th June 2015

Figure 8.1 In text analytics, (usually) the first challenge is to structure the input text; then it can be thoroughly analyzed.

While language isn't limited to the natural language, the focus of this chapter will be on *Natural Language Processing (NLP)*. Examples of non-natural languages would be machine logs, mathematics, and Morse code. Technically even Esperanto, Klingon, and Dragon language aren't in the field of natural languages because they were invented deliberately instead of evolving over time; they didn't come "natural" to us. These last languages are nevertheless fit for natural communication (speech, writing); they have a grammar and a vocabulary as all natural languages do, and the same text mining techniques could apply to them.

8.1 Text mining in the real world

In your day-to-day life you've already come across text mining and natural language applications. Autocomplete and spelling correctors are constantly analyzing the text you type before sending an email or text message. When Facebook autocompletes your status with the name of a friend, it does this with the help of a technique called *named entity recognition*, although this would be only one component of their repertoire. The goal isn't only to detect that you're typing a noun, but also to guess you're referring to a person and recognize who it might be. Another example of named entity recognition is shown in figure 8.2. Google knows Chelsea is a football club but responds differently when asked for a person.

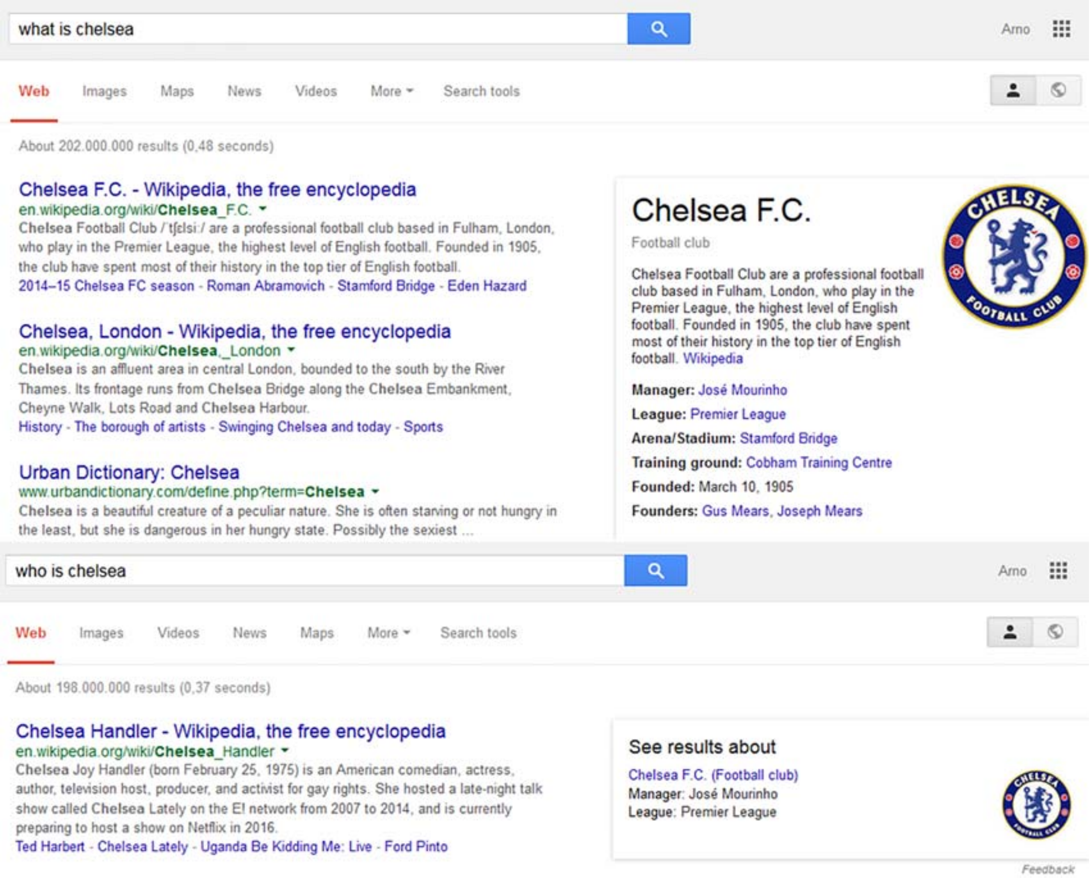
Google uses many types of text mining when presenting you with the results of a query. What pops up in your own mind when someone says "Chelsea"? Chelsea could be many things: a person; a soccer club; a neighborhood in Manhattan, New York or London; a food market; a flower show; and so on. Google knows this and returns different answers to the question "Who is Chelsea?" versus "What is Chelsea?" To provide the most relevant answer, Google must do (among other things) all of the following:

- Preprocess all the documents it collects for named entities
- Perform language identification
- Detect what type of entity you're referring to
- Match a query to a result
- Detect the type of content to return (PDF, adult-sensitive)

This example shows that text mining isn't only about the direct meaning of text itself but also involves meta-attributes such as language and document type.

Google uses text mining for much more than answering queries. Next to shielding its Gmail users from spam, it also divides the emails into different categories such as social, updates, and forums, as shown in figure 8.3.

It's possible to go much further than answering simple questions when you combine text with other logic and mathematics.



About 202.000.000 results (0,48 seconds)

Chelsea F.C. - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Chelsea_F.C.
Chelsea Football Club /ˈtʃɛlsiː/ are a professional football club based in Fulham, London, who play in the Premier League, the highest level of English football. Founded in 1905, the club have spent most of their history in the top tier of English football.
2014–15 Chelsea FC season - Roman Abramovich - Stamford Bridge - Eden Hazard

Chelsea, London - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Chelsea,_London
Chelsea is an affluent area in central London, bounded to the south by the River Thames. Its frontage runs from Chelsea Bridge along the Chelsea Embankment, Cheyne Walk, Lots Road and Chelsea Harbour.
History - The borough of artists - Swinging Chelsea and today - Sports

Urban Dictionary: Chelsea
www.urbandictionary.com/define.php?term=Chelsea
Chelsea is a beautiful creature of a peculiar nature. She is often starving or not hungry in the least, but she is dangerous in her hungry state. Possibly the sexiest ...

Chelsea F.C.
Football club
Chelsea Football Club are a professional football club based in Fulham, London, who play in the Premier League, the highest level of English football. Founded in 1905, the club have spent most of their history in the top tier of English football. Wikipedia

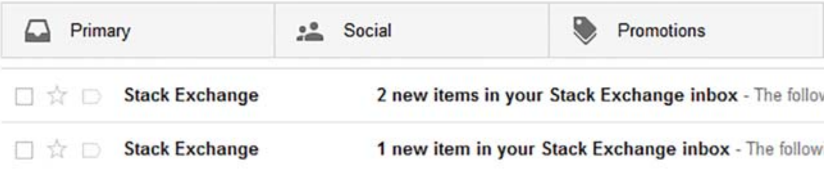
Manager: José Mourinho
League: Premier League
Arena/Stadium: Stamford Bridge
Training ground: Cobham Training Centre
Founded: March 10, 1905
Founders: Gus Mears, Joseph Mears

About 198.000.000 results (0,37 seconds)

Chelsea Handler - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Chelsea_Handler
Chelsea Joy Handler (born February 25, 1975) is an American comedian, actress, author, television host, producer, and activist for gay rights. She hosted a late-night talk show called Chelsea Lately on the E! network from 2007 to 2014, and is currently preparing to host a show on Netflix in 2016.
Ted Harbert - Chelsea Lately - Uganda Be Kidding Me: Live - Ford Pinto

See results about
Chelsea F.C. (Football club)
Manager: José Mourinho
League: Premier League

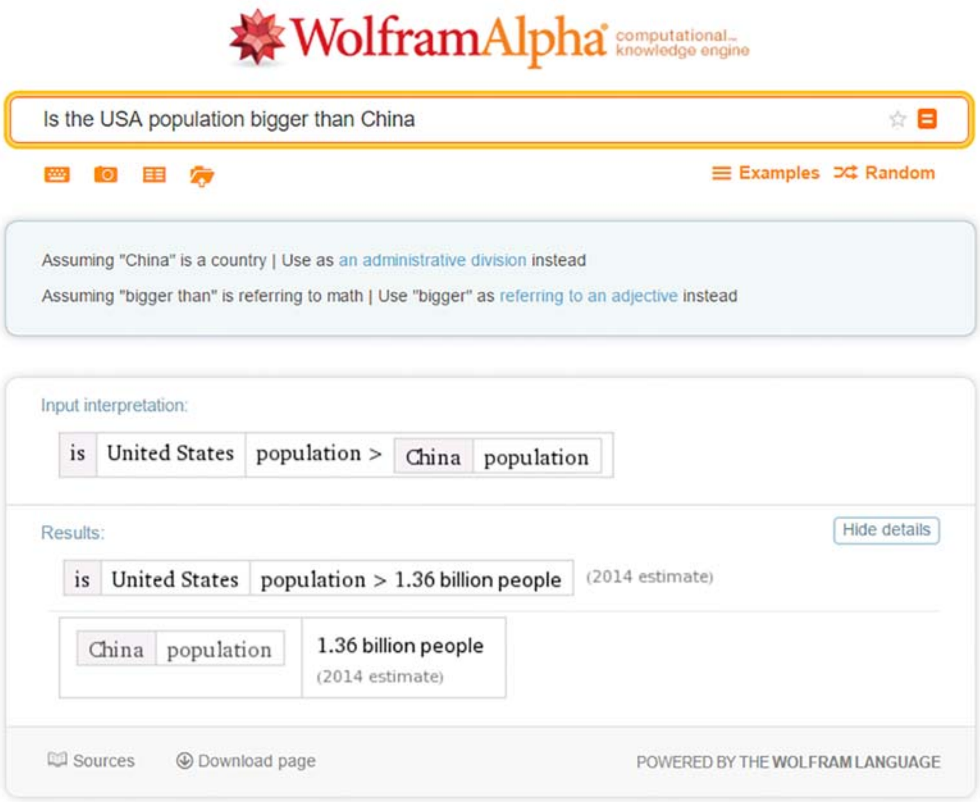
Figure 8.2 The different answers to the queries "Who is Chelsea?" and "What is Chelsea?" imply that Google uses text mining techniques to answer these queries.



□ ☆ Stack Exchange	2 new items in your Stack Exchange inbox - The follov
□ ☆ Stack Exchange	1 new item in your Stack Exchange inbox - The follow

Figure 8.3 Emails can be automatically divided by category based on content and origin.

This allows for the creation of *automatic reasoning engines* driven by natural language queries. Figure 8.4 shows how "Wolfram Alpha," a computational knowledge engine, uses text mining and automatic reasoning to answer the question "Is the USA population bigger than China?"



Is the USA population bigger than China

Assuming "China" is a country | Use as an administrative division instead

Assuming "bigger than" is referring to math | Use "bigger" as referring to an adjective instead

Input interpretation:

is | United States | population > | China | population

Results:

is | United States | population > 1.36 billion people (2014 estimate)

China | population | 1.36 billion people (2014 estimate)

Figure 8.4 The Wolfram Alpha engine uses text mining and logical reasoning to answer a question.

If this isn't impressive enough, the IBM Watson astonished many in 2011 when the machine was set up against two human players in a game of *Jeopardy*. *Jeopardy* is an American quiz show where people receive the answer to a question and points are scored for guessing the correct question for that answer. See figure 8.5.

It's safe to say this round goes to artificial intelligence. IBM Watson is a cognitive engine that can interpret natural language and answer questions based on an extensive knowledge base.

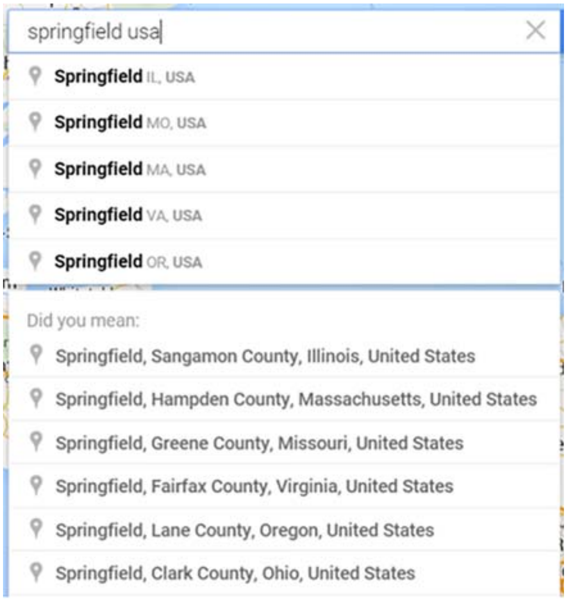
Figure 8.5 IBM Watson wins *Jeopardy* against human players.

Text mining has many applications, including, but not limited to, the following:

- Entity identification
- Plagiarism detection
- Topic identification
- Text clustering
- Translation
- Automatic text summarization
- Fraud detection
- Spam filtering
- Sentiment analysis

Text mining is useful, but is it difficult? Sorry to disappoint: Yes, it is.

When looking at the examples of Wolfram Alpha and IBM Watson, you might have gotten the impression that text mining is easy. Sadly, no. In reality text mining is a complicated task and even many seemingly simple things can't be done satisfactorily. For instance, take the task of guessing the correct address. Figure 8.6 shows how difficult it is to return the exact result with certitude and how Google Maps prompts you for more information when looking for "Springfield." In this case a human wouldn't have done any better without additional context, but this ambiguity is one of the many problems you face in a text mining application.



springfield usa

Springfield IL, USA

Springfield MO, USA

Springfield MA, USA

Springfield VA, USA

Springfield OR, USA

Did you mean:

Springfield, Sangamon County, Illinois, United States

Springfield, Hampden County, Massachusetts, United States

Springfield, Greene County, Missouri, United States

Springfield, Fairfax County, Virginia, United States

Springfield, Lane County, Oregon, United States

Springfield, Clark County, Ohio, United States

Figure 8.6 Google Maps asks you for more context due to the ambiguity of the query "Springfield."

Another problem is *spelling mistakes* and *different (correct) spelling* forms of a word. Take the following three references to New York: "NY," "Neww York," and "New York." For a human, it's easy to see they all refer to the city of New York. Because of the way our brain interprets text, understanding text with spelling mistakes comes naturally to us; people may not even notice them. But for a computer these are unrelated strings unless we use algorithms to tell it that they're referring to the same entity. Related problems are synonyms and the use of pronouns. Try assigning the right person to the pronoun "she" in the next sentences: "John gave flowers to Marleen's parents when he met her parents for the first time. She was so happy with this gesture." Easy enough, right? Not for a computer.

We can solve many similar problems with ease, but they often prove hard for a machine. We can train algorithms that work well on a specific problem in a well-defined scope, but more general algorithms that work in all cases are another beast altogether. For instance, we can teach a computer to recognize and retrieve US account numbers from text, but this doesn't generalize well to account numbers from other countries.

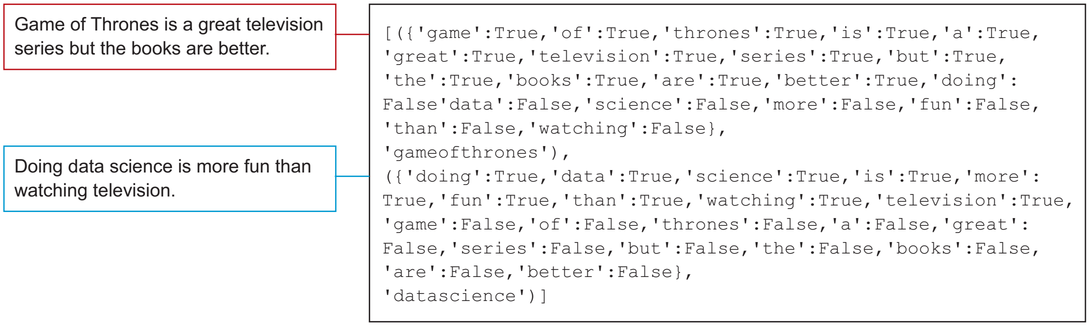
Language algorithms are also sensitive to the context the language is used in, even if the language itself remains the same. English models won't work for Arabic and vice versa, but even if we keep to English—an algorithm trained for Twitter data isn't likely to perform well on legal texts. Let's keep this in mind when we move on to the chapter case study: there's no perfect, one-size-fits-all solution in text mining.

8.2 Text mining techniques

During our upcoming case study we'll tackle the problem of *text classification*: automatically classifying uncategorized texts into specific categories. To get from raw textual data to our final destination we'll need a few data mining techniques that require background information for us to use them effectively. The first important concept in text mining is the "bag of words."

8.2.1 Bag of words

To build our classification model we'll go with the bag of words approach. *Bag of words* is the simplest way of structuring textual data: every document is turned into a word vector. If a certain word is present in the vector it's labeled "True"; the others are labeled "False". Figure 8.7 shows a simplified example of this, in case there are only two documents: one about the television show *Game of Thrones* and one about data science. The two word vectors together form the *document-term matrix*. The document-term matrix holds a column for every term and a row for every document. The values are yours to decide upon. In this chapter we'll use binary: term is present? True or False.



Game of Thrones is a great television series but the books are better.

Doing data science is more fun than watching television.

```
[({'game':True,'of':True,'thrones':True,'is':True,'a':True,
'great':True,'television':True,'series':True,'but':True,
'the':True,'books':True,'are':True,'better':True,'doing':
False'data':False,'science':False,'more':False,'fun':False,
'than':False,'watching':False},
'gameofthrones'),
({'doing':True,'data':True,'science':True,'is':True,'more':
True,'fun':True,'than':True,'watching':True,'television':True,
'game':False,'of':False,'thrones':False,'a':False,'great':
False,'series':False,'but':False,'the':False,'books':False,
'are':False,'better':False},
'datascience')]
```

Figure 8.7 A text is transformed into a bag of words by labeling each word (term) with "True" if it is present in the document and "False" if not.

The example from figure 8.7 does give you an idea of the structured data we'll need to start text analysis, but it's severely simplified: not a single word was filtered out and no stemming (we'll go into this later) was applied. A big corpus can have thousands of unique words. If all have to be labeled like this without any filtering, it's easy to see we might end up with a large volume of data. *Binary coded bag of words* as shown in figure 8.7 is but one way to structure the data; other techniques exist.

Term Frequency—Inverse Document Frequency (TF-IDF)

A well-known formula to fill up the document-term matrix is *TF-IDF* or Term Frequency multiplied by Inverse Document Frequency. *Binary bag of words* assigns True or False (term is there or not), while *simple frequencies* count the number of times the term occurred. TF-IDF is a bit more complicated and takes into account how many times a term occurred in the document (TF). TF can be a simple term count, a binary count (True or False), or a logarithmically scaled term count. It depends on what works best for you. In case TF is a term frequency, the formula of TF is the following:

$$TF = f_{t,d}$$

TF is the frequency (f) of the term (t) in the document (d).

But TF-IDF also takes into account all the other documents because of the Inverse Document Frequency. IDF gives an idea of how common the word is in the entire corpus: the higher the document frequency the more common, and more common words are less informative. For example the words "a" or "the" aren't likely to provide specific information on a text. The formula of IDF with logarithmic scaling is the most commonly used form of IDF:

$$IDF = \log(N/|\{d \in D:t \in d\}|)$$

with N being the total number of documents in the corpus, and the $|\{d \in D:t \in d\}|$ being the number of documents (d) in which the term (t) appears.

The TF-IDF score says this about a term: how important is this word to distinguish this document from the others in the corpus? The formula of TF-IDF is thus

$$\frac{1F}{IDF} = f_{t,d}/\log(N/|\{d \in D:t \in d\}|)$$

We won't use TF-IDF, but when setting your next steps in text mining, this should be one of the first things you'll encounter. TF-IDF is also what was used by Elasticsearch behind the scenes in chapter 6. It's a good way to go if you want to use TF-IDF for text analytics; leave the text mining to specialized software such as SOLR or Elasticsearch and take the document/term matrix for text analytics from there.

Before getting to the actual bag of words, many other data manipulation steps take place:

- *Tokenization*—The text is cut into pieces called "tokens" or "terms." These tokens are the most basic unit of information you'll use for your model. The terms are often words but this isn't a necessity. Entire sentences can be used for analysis. We'll use *unigrams*: terms consisting of one word. Often, however, it's useful to include *bigrams* (two words per token) or *trigrams* (three words per token) to capture extra meaning and increase the performance of your models.

This does come at a cost, though, because you're building bigger term-vectors by including bigrams and/or trigrams in the equation.

- *Stop word filtering*—Every language comes with words that have little value in text analytics because they're used so often. NLTK comes with a short list of English stop words we can filter. If the text is tokenized into words, it often makes sense to rid the word vector of these low-information stop words.
- *Lowercasing*—Words with capital letters appear at the beginning of a sentence, others because they're proper nouns or adjectives. We gain no added value making that distinction in our term matrix, so all terms will be set to lowercase.

Another data preparation technique is *stemming*. This one requires more elaboration.

8.2.2 Stemming and lemmatization

Stemming is the process of bringing words back to their root form; this way you end up with less variance in the data. This makes sense if words have similar meanings but are written differently because, for example, one is in its plural form. Stemming attempts to unify by cutting off parts of the word. For example "planes" and "plane" both become "plane."

Another technique, called *lemmatization*, has this same goal but does so in a more grammatically sensitive way. For example, while both stemming and lemmatization would reduce "cars" to "car," lemmatization can also bring back conjugated verbs to their unconjugated forms such as "are" to "be." Which one you use depends on your case, and lemmatization profits heavily from POS Tagging (Part of Speech Tagging). *POS Tagging* is the process of attributing a grammatical label to every part of a sentence. You probably did this manually in school as a language exercise. Take the sentence "*Game of Thrones* is a television series." If we apply POS Tagging on it we get

({"game":"NN"},{"of":"IN},{"thrones":"NNS},{"is":"VBZ},{"a":"DT},{"television":"NN}, {"series":"NN})

NN is a noun, IN is a preposition, NNS is a noun in its plural form, VBZ is a third-person singular verb, and DT is a determiner. Table 8.1 has the full list.

Table 8.1 A list of all POS tags

Tag	Meaning	Tag	Meaning
CC	Coordinating conjunction	CD	Cardinal number
DT	Determiner	EX	Existential
FW	Foreign word	IN	Preposition or subordinating conjunction
JJ	Adjective	JJR	Adjective, comparative
JJS	Adjective, superlative	LS	List item marker
MD	Modal	NN	Noun, singular or mass

Table 8.1 A list of all POS tags *(continued)*

Tag	Meaning	Tag	Meaning
NNS	Noun, plural	NNP	Proper noun, singular
NNPS	Proper noun, plural	PDT	Predeterminer
POS	Possessive ending	PRP	Personal pronoun
PRP$	Possessive pronoun	RB	Adverb
RBR	Adverb, comparative	RBS	Adverb, superlative
RP	Particle	SYM	Symbol
UH	Interjection	VB	Verb, base form
VBD	Verb, past tense	VBG	Verb, gerund or present participle
VBN	Verb, past participle	VBP	Verb, non-3rd person singular present
VBZ	Verb, 3rd person singular present	WDT	Wh-determiner
WP	Wh-pronoun	WP$	Possessive wh-pronoun
WRB	Wh-adverb		

POS Tagging is a use case of sentence-tokenization rather than word-tokenization. After the POS Tagging is complete you can still proceed to word tokenization, but a POS Tagger requires whole sentences. Combining POS Tagging and lemmatization is likely to give cleaner data than using only a stemmer. For the sake of simplicity we'll stick to stemming in the case study, but consider this an opportunity to elaborate on the exercise.

We now know the most important things we'll use to do the data cleansing and manipulation (text mining). For our text analytics, let's add the decision tree classifier to our repertoire.

8.2.3 Decision tree classifier

The data analysis part of our case study will be kept simple as well. We'll test a Naïve Bayes classifier and a decision tree classifier. As seen in chapter 3 the Naïve Bayes classifier is called that because it considers each input variable to be independent of all the others, which is naïve, especially in text mining. Take the simple examples of "data science," "data analysis," or "game of thrones." If we cut our data in unigrams we get the following separate variables (if we ignore stemming and such): "data," "science," "analysis," "game," "of," and "thrones." Obviously links will be lost. This can, in turn, be overcome by creating bigrams (data science, data analysis) and trigrams (game of thrones).

The decision tree classifier, however, doesn't consider the variables to be independent of one another and actively creates *interaction variables* and *buckets*. An *interaction* *variable* is a variable that combines other variables. For instance "data" and "science" might be good predictors in their own right but probably the two of them co-occurring in the same text might have its own value. A bucket is somewhat the opposite. Instead of combining two variables, a variable is split into multiple new ones. This makes sense for numerical variables. Figure 8.8 shows what a decision tree might look like and where you can find interaction and bucketing.

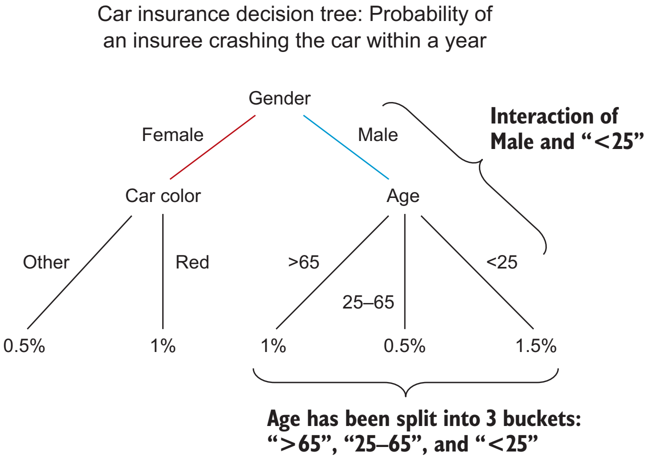

Car insurance decision tree: Probability of an insuree crashing the car within a year
Gender
Female
Male
Interaction of Male and "<25"
Car color
Age
Other
Red
>65
25–65
<25
0.5%
1%
1%
0.5%
1.5%
Age has been split into 3 buckets: ">65", "25–65", and "<25"

Figure 8.8 Fictitious decision tree model. A decision tree automatically creates buckets and supposes interactions between input variables.

Whereas Naïve Bayes supposes independence of all the input variables, a decision tree is built upon the assumption of interdependence. But how does it build this structure? A decision tree has a few possible criteria it can use to split into branches and decide which variables are more important (are closer to the root of the tree) than others. The one we'll use in the NLTK decision tree classifier is "information gain." To understand information gain, we first need to look at entropy. *Entropy* is a measure of unpredictability or chaos. A simple example would be the gender of a baby. When a woman is pregnant, the gender of the fetus can be male or female, but we don't know which one it is. If you were to guess, you have a 50% chance to guess correctly (give or take, because gender distribution isn't 100% uniform). However, during the pregnancy you have the opportunity to do an ultrasound to determine the gender of the fetus. An ultrasound is never 100% conclusive, but the farther along in fetal development, the more accurate it becomes. This accuracy gain, or *information gain*, is there because uncertainty or entropy drops. Let's say an ultrasound at 12 weeks pregnancy has a 90% accuracy in determining the gender of the baby. A 10% uncertainty still exists, but the ultrasound did reduce the uncertainty

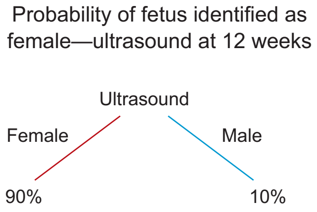

Probability of fetus identified as female—ultrasound at 12 weeks
Ultrasound
Female
Male
90%
10%

Figure 8.9 Decision tree with one variable: the doctor's conclusion from watching an ultrasound during a pregnancy. What is the probability of the fetus being female?

from 50% to 10%. That's a pretty good discriminator. A decision tree follows this same principle, as shown in figure 8.9.

If another gender test has more predictive power, it could become the root of the tree with the ultrasound test being in the branches, and this can go on until we run out of variables or observations. We can run out of observations, because at every branch split we also split the input data. This is a big weakness of the decision tree, because at the leaf level of the tree robustness breaks down if too few observations are left; the decision trees starts to overfit the data. *Overfitting* allows the model to mistake randomness for real correlations. To counteract this, a decision tree is *pruned*: its meaningless branches are left out of the final model.

Now that we've looked at the most important new techniques, let's dive into the case study.

8.3 Case study: Classifying Reddit posts

While text mining has many applications, in this chapter's case study we focus on *document classification*. As pointed out earlier in this chapter, this is exactly what Google does when it arranges your emails in categories or attempts to distinguish spam from regular emails. It's also extensively used by contact centers that process incoming customer questions or complaints: written complaints first pass through a topic detection filter so they can be assigned to the correct people for handling. Document classification is also one of the mandatory features of social media monitoring systems. The monitored tweets, forum or Facebook posts, newspaper articles, and many other internet resources are assigned topic labels. This way they can be reused in reports. *Sentiment analysis* is a specific type of text classification: is the author of a post negative, positive, or neutral on something? That "something" can be recognized with entity recognition.

In this case study we'll draw on posts from Reddit, a website also known as the self-proclaimed "front page of the internet," and attempt to train a model capable of distinguishing whether someone is talking about "data science" or "game of thrones."

The end result can be a presentation of our model or a full-blown interactive application. In chapter 9 we'll focus on application building for the end user, so for now we'll stick to presenting our classification model.

To achieve our goal we'll need all the help and tools we can get, and it happens Python is once again ready to provide them.

8.3.1 Meet the Natural Language Toolkit

Python might not be the most execution efficient language on earth, but it has a mature package for text mining and language processing: the *Natural Language Toolkit (NLTK)*. NLTK is a collection of algorithms, functions, and annotated works that will guide you in taking your first steps in text mining and natural language processing. NLTK is also excellently documented on nltk.org. NLTK is, however, not often used for production-grade work, like other libraries such as scikit-learn.

Installing NLTK and its corpora

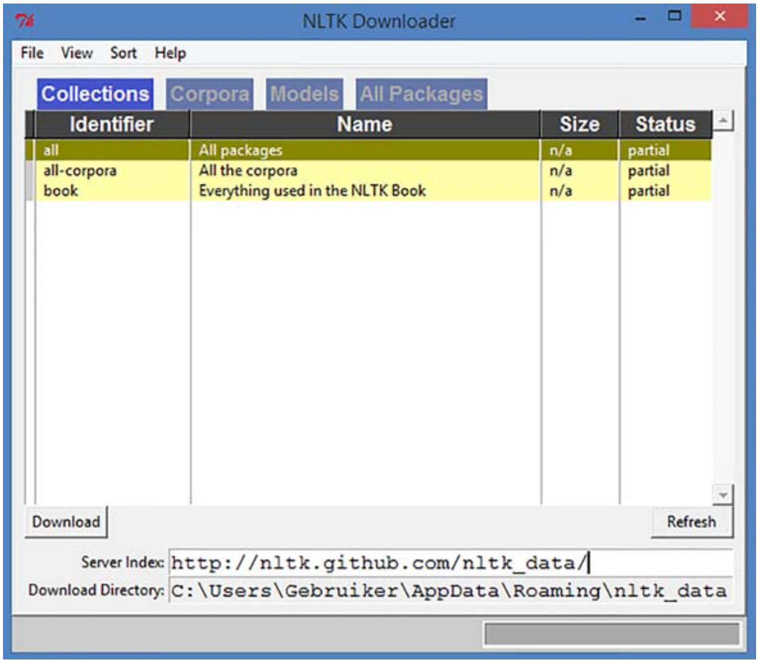
Install NLTK with your favorite package installer. In case you're using Anaconda, it comes installed with the default Anaconda setup. Otherwise you can go for "pip" or "easy_install". When this is done you still need to install the models and corpora included to have it be fully functional. For this, run the following Python code:

- import nltk
- nltk.download()

Depending on your installation this will give you a pop-up or more command-line options.

Figure 8.10 shows the pop-up box you get when issuing the nltk.download() command.

You can download all the corpora if you like, but for this chapter we'll only make use of "punkt" and "stopwords". This download will be explicitly mentioned in the code that comes with this book.



Identifier	Name	Size	Status
all	All packages	n/a	partial
all-corpora	All the corpora	n/a	partial
book	Everything used in the NLTK Book	n/a	partial

Server Index: http://nltk.github.com/nltk_data/

Download Directory: C:\Users\Gebruiker\AppData\Roaming\nltk_data

Figure 8.10 Choose All Packages to fully complete the NLTK installation.

Two IPython notebook files are available for this chapter:

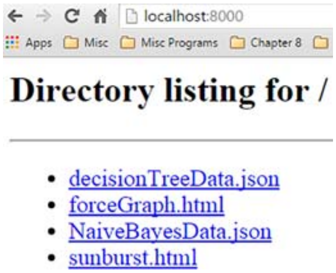
- *Data collection*—Will contain the data collection part of this chapter's case study.
- *Data preparation and analysis*—The stored data is put through data preparation and then subjected to analysis.

All code in the upcoming case study can be found in these two files in the same sequence and can also be run as such. In addition, two interactive graphs are available for download:

- *forceGraph.html*—Represents the top 20 features of our Naïve Bayes model
- *Sunburst.html*—Represents the top four branches of our decision tree model

To open these two HTML pages, an HTTP server is necessary, which you can get using Python and a command window:

- Open a command window (Linux, Windows, whatever you fancy).
- Move to the folder containing the HTML files and their JSON data files: decisionTreeData.json for the sunburst diagram and NaiveBayesData.json for the force graph. It's important the HTML files remain in the same location as their data files or you'll have to change the JavaScript in the HTML file.
- Create a Python HTTP server with the following command: `python -m SimpleHTTPServer 8000`
- Open a browser and go to localhost:8000; here you can select the HTML files, as shown in figure 8.11.



Directory listing for /

- decisionTreeData.json
- forceGraph.html
- NaiveBayesData.json
- sunburst.html

Figure 8.11 Python HTTP server serving this chapter's output

The Python packages we'll use in this chapter:

- *NLTK*—For text mining
- *PRAW*—Allows downloading posts from Reddit
- *SQLite3*—Enables us to store data in the SQLite format
- *Matplotlib*—A plotting library for visualizing data

Make sure to install all the necessary libraries and corpora before moving on. Before we dive into the action, however, let's look at the steps we'll take to get to our goal of creating a topic classification model.

8.3.2 Data science process overview and step 1: The research goal

To solve this text mining exercise, we'll once again make use of the data science process. Figure 8.12 shows the data science process applied to our Reddit classification case.

Not all the elements depicted in figure 8.12 might make sense at this point, and the rest of the chapter is dedicated to working this out in practice as we work toward our research goal: creating a classification model capable of distinguishing posts about "data science" from posts about "Game of Thrones." Without further ado, let's go get our data.

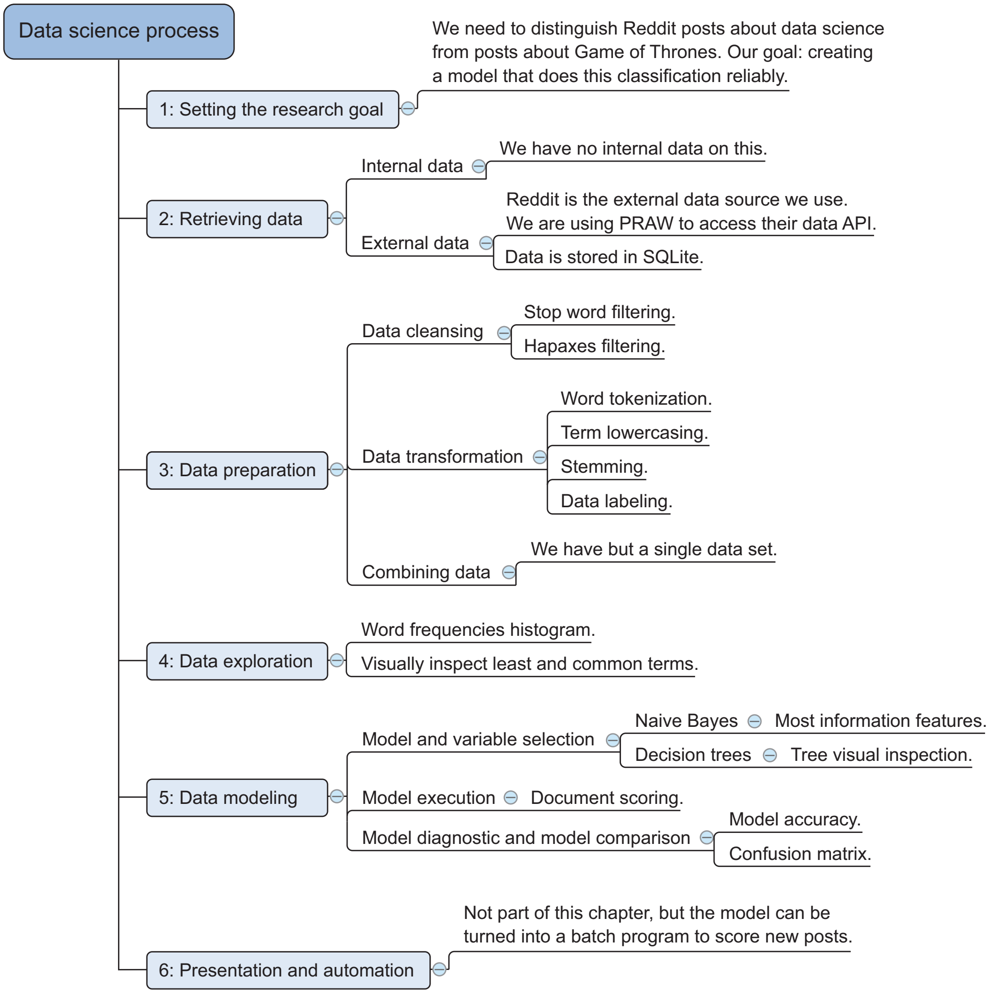

Data science process
We need to distinguish Reddit posts about data science from posts about Game of Thrones. Our goal: creating a model that does this classification reliably.
1: Setting the research goal
Internal data
We have no internal data on this.
2: Retrieving data
Reddit is the external data source we use.
We are using PRAW to access their data API.
External data
Data is stored in SQLite.
Data cleansing
Stop word filtering.
Hapaxes filtering.
Word tokenization.
Term lowercasing.
3: Data preparation
Data transformation
Stemming.
Data labeling.
We have but a single data set.
Combining data
Word frequencies histogram.
4: Data exploration
Visually inspect least and common terms.
Naive Bayes
Most information features.
Model and variable selection
Decision trees
Tree visual inspection.
5: Data modeling
Model execution
Document scoring.
Model accuracy.
Model diagnostic and model comparison
Confusion matrix.
Not part of this chapter, but the model can be turned into a batch program to score new posts.
6: Presentation and automation

Figure 8.12 Data science process overview applied to Reddit topic classification case study

8.3.3 Step 2: Data retrieval

We'll use Reddit data for this case, and for those unfamiliar with Reddit, take the time to familiarize yourself with its concepts at www.reddit.com.

Reddit calls itself "the front page of the internet" because users can post things they find interesting and/or found somewhere on the internet, and only those things deemed interesting by many people are featured as "popular" on its homepage. You could say Reddit gives an overview of the trending things on the internet. Any user can post within a predefined category called a "subreddit." When a post is made, other users get to comment on it and can up-vote it if they like the content or down-vote it if they dislike it. Because a post is always part of a subreddit, we have this metadata at our disposal when we hook up to the Reddit API to get our data. We're effectively fetching labeled data because we'll assume that a post in the subreddit "gameofthrones" has something to do with "gameofthrones."

To get to our data we make use of the official Reddit Python API library called PRAW. Once we get the data we need, we'll store it in a lightweight database-like file called SQLite. SQLite is ideal for storing small amounts of data because it doesn't require any setup to use and will respond to SQL queries like any regular relational database does. Any other data storage medium will do; if you prefer Oracle or Postgres databases, Python has an excellent library to interact with these without the need to write SQL. SQLAlchemy will work for SQLite files as well. Figure 8.13 shows the data retrieval step within the data science process.

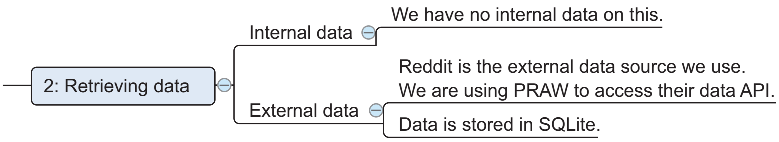

2: Retrieving data
Internal data
We have no internal data on this.
External data
Reddit is the external data source we use.
We are using PRAW to access their data API.
Data is stored in SQLite.

Figure 8.13 The data science process data retrieval step for a Reddit topic classification case

Open your favorite Python interpreter; it's time for action, as shown in listing 8.1. First we need to collect our data from the Reddit website. If you haven't already, use `pip install praw` or `conda install praw` (Anaconda) before running the following script.

NOTE The code for step 2 can also be found in the IPython file "Chapter 8 data collection." It's available in this book's download section.

Listing 8.1 Setting up SQLLite database and Reddit API client

```
import praw
import sqlite3

conn = sqlite3.connect('reddit.db')
c = conn.cursor()

c.execute('''DROP TABLE IF EXISTS topics''')
c.execute('''DROP TABLE IF EXISTS comments''')
c.execute('''CREATE TABLE topics
             (topicTitle text, topicText text, topicID text,
topicCategory text)''')
c.execute('''CREATE TABLE comments
             (commentText text, commentID text ,
topicTitle text, topicText text, topicID text ,
 topicCategory text)''')

user_agent = "Introducing Data Science Book"
r = praw.Reddit(user_agent=user_agent)

subreddits = ['datascience','gameofthrones']

limit = 1000
```

Import PRAW and SQLite3 libraries.

Set up connection to SQLite database.

Execute SQL statements to create topics and comments table.

Create PRAW user agent so we can use Reddit API.

Our list of subreddits we'll draw into our SQLite database.

Maximum number of posts we'll fetch from Reddit per category. Maximum Reddit allows at any single time is also 1,000.

Let's first import the necessary libraries.

Now that we have access to the SQLite3 and PRAW capabilities, we need to prepare our little local database for the data it's about to receive. By defining a connection to a SQLite file we automatically create it if it doesn't already exist. We then define a data cursor that's capable of executing any SQL statement, so we use it to predefine the structure of our database. The database will contain two tables: the topics table contains Reddit topics, which is similar to someone starting a new post on a forum, and the second table contains the comments and is linked to the topic table via the "topicID" column. The two tables have a one (topic table) to many (comment table) relationship. For the case study, we'll limit ourselves to using the topics table, but the data collection will incorporate both because this allows you to experiment with this extra data if you feel like it. To hone your text-mining skills you could perform sentiment analysis on the topic comments and find out what topics receive negative or positive comments. You could then correlate this to the model features we'll produce by the end of this chapter.

We need to create a PRAW client to get access to the data. Every subreddit can be identified by its name, and we're interested in "datascience" and "gameofthrones." The limit represents the maximum number of topics (posts, not comments) we'll draw in from Reddit. A thousand is also the maximum number the API allows us to fetch at any given request, though we could request more later on when people have posted new things. In fact we can run the API request periodically and gather data over time. While at any given time you're limited to a thousand posts, nothing stops you from growing your own database over the course of months. It's worth noting the following script might take about an hour to complete. If you don't feel like waiting, feel free to proceed and use the downloadable SQLite file. Also, if you run it now you are not likely to get the exact same output as when it was first run to create the output shown in this chapter.

Let's look at our data retrieval function, as shown in the following listing.

Listing 8.2 Reddit data retrieval and storage in SQLite

Specific fields of the topic are appended to the list. We only use the title and text throughout the exercise but the topic ID would be useful for building your own (bigger) database of topics.

From subreddits, get hottest 1,000 (in our case) topics.

This part is an informative print and not necessary for code to work. It only informs you about the download progress.

```
def prawGetData(limit,subredditName):
    topics = r.get_subreddit(subredditName).get_hot(limit=limit)
    commentInsert = []
    topicInsert = []
    topicNBR = 1
    for topic in topics:
        if (float(topicNBR)/limit)*100 in xrange(1,100):
            print '*********** TOPIC:' + str(topic.id)
+ ' *********COMPLETE: ' + str((float(topicNBR)/limit)*100)
+ ' % ****'
        topicNBR += 1
        try:
            topicInsert.append((topic.title,topic.selftext,topic.id,
            subredditName))
        except:
            pass
        try:
            for comment in topic.comments:
                commentInsert.append((comment.body,comment.id,
topic.title,topic.selftext,topic.id,subredditName))
        except:
            pass
    print  '********************************'
    print  'INSERTING DATA INTO SQLITE'
    c.executemany('INSERT INTO topics VALUES (?,?,?,?)', topicInsert)
    print  'INSERTED TOPICS'
    c.executemany('INSERT INTO comments VALUES (?,?,?,?,?,?)', commentInsert)
    print  'INSERTED COMMENTS'
    conn.commit()

for subject in subreddits:
    prawGetData(limit=limit,subredditName=subject)
```

Append comments to a list. These are not used in the exercise but now you have them for experimentation.

Insert all topics into SQLite database.

Insert all comments into SQLite database.

Commit changes (data insertions) to database. Without the commit, no data will be inserted.

The function is executed for all subreddits we specified earlier.

The `prawGetData()` function retrieves the "hottest" topics in its subreddit, appends this to an array, and then gets all its related comments. This goes on until a thousand topics are reached or no more topics exist to fetch and everything is stored in the SQLite database. The print statements are there to inform you on its progress toward gathering a thousand topics. All that's left for us to do is execute the function for each subreddit.

If you'd like this analysis to incorporate more than two subreddits, this is a matter of adding an extra category to the subreddits array.

With the data collected, we're ready to move on to data preparation.

8.3.4 Step 3: Data preparation

As always, data preparation is the most crucial step to get correct results. For text mining this is even truer since we don't even start off with structured data.

The upcoming code is available online as IPython file "Chapter 8 data preparation and analysis." Let's start by importing the required libraries and preparing the SQLite database, as shown in the following listing.

Listing 8.3 Text mining, libraries, corpora dependencies, and SQLite database connection

```
import sqlite3
import nltk
import matplotlib.pyplot as plt
from collections import OrderedDict
import random
```

Import all required libraries

```
nltk.download('punkt')
nltk.download('stopwords')
```

Download corpora we make use of

```
conn = sqlite3.connect('reddit.db')
c = conn.cursor()
```

Make a connection to SQLite database that contains our Reddit data

In case you haven't already downloaded the full NLTK corpus, we'll now download the part of it we'll use. Don't worry if you already downloaded it, the script will detect if your corpora is up to date.

Our data is still stored in the Reddit SQLite file so let's create a connection to it.

Even before exploring our data we know of at least two things we have to do to clean the data: stop word filtering and lowercasing.

A general word filter function will help us filter out the unclean parts. Let's create one in the following listing.

Listing 8.4 Word filtering and lowercasing functions

```
def wordFilter(excluded,wordrow):
    filtered = [word for word in wordrow if word not in excluded]
    return filtered
stopwords = nltk.corpus.stopwords.words('english')
def lowerCaseArray(wordrow):
    lowercased = [word.lower() for word in wordrow]
    return lowercased
```

wordFilter() function will remove a term from an array of terms

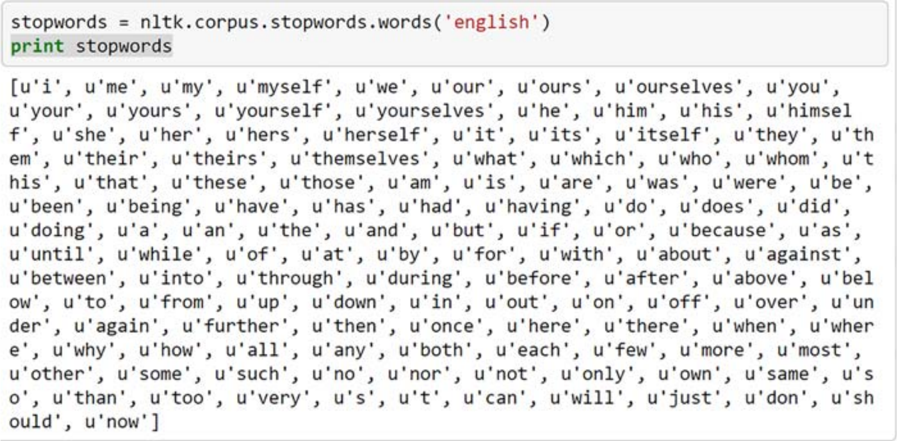
Stop word variable contains English stop words per default present in NLTK

lowerCaseArray() function transforms any term to its lowercased version

The English stop words will be the first to leave our data. The following code will provide us these stop words:

```
stopwords = nltk.corpus.stopwords.words('english')
print stopwords
```

Figure 8.14 shows the list of English stop words in NLTK.



```
stopwords = nltk.corpus.stopwords.words('english')
print stopwords
```

```
[u'i', u'me', u'my', u'myself', u'we', u'our', u'ours', u'ourselves', u'you',
u'your', u'yours', u'yourself', u'yourselves', u'he', u'him', u'his', u'himsel
f', u'she', u'her', u'hers', u'herself', u'it', u'its', u'itself', u'they', u'th
em', u'their', u'theirs', u'themselves', u'what', u'which', u'who', u'whom', u't
his', u'that', u'these', u'those', u'am', u'is', u'are', u'was', u'were', u'be',
u'been', u'being', u'have', u'has', u'had', u'having', u'do', u'does', u'did',
u'doing', u'a', u'an', u'the', u'and', u'but', u'if', u'or', u'because', u'as',
u'until', u'while', u'of', u'at', u'by', u'for', u'with', u'about', u'against',
u'between', u'into', u'through', u'during', u'before', u'after', u'above', u'bel
ow', u'to', u'from', u'up', u'down', u'in', u'out', u'on', u'off', u'over', u'un
der', u'again', u'further', u'then', u'once', u'here', u'there', u'when', u'wher
e', u'why', u'how', u'all', u'any', u'both', u'each', u'few', u'more', u'most',
u'other', u'some', u'such', u'no', u'nor', u'not', u'only', u'own', u'same', u's
o', u'than', u'too', u'very', u's', u't', u'can', u'will', u'just', u'don', u'sh
ould', u'now']
```

Figure 8.14 English stop words list in NLTK

With all the necessary components in place, let's have a look at our first data processing function in the following listing.

Listing 8.5 First data preparation function and execution

We'll use data['all_words'] for data exploration.

Create pointer to AWLite data.

Fetch data row by row.

row[0] is title, row[1] is topic text; we turn them into a single text blob.

```
def data_processing(sql):
    c.execute(sql)
    data = {'wordMatrix':[],'all_words':[]}
    row = c.fetchone()
    while row is not None:
        wordrow = nltk.tokenize.word_tokenize(row[0]+" "+row[1])
        wordrow_lowercased = lowerCaseArray(wordrow)
        wordrow_nostopwords = wordFilter(stopwords,wordrow_lowercased)
        data['all_words'].extend(wordrow_nostopwords)
        data['wordMatrix'].append(wordrow_nostopwords)
        row = c.fetchone()
    return data

subreddits = ['datascience','gameofthrones']
data = {}
for subject in subreddits:
    data[subject] = data_processing(sql='''SELECT
     topicTitle,topicText,topicCategory FROM topics
WHERE topicCategory = '''+"'"+subject+"'")
```

Get new document from SQLite database.

Our subreddits as defined earlier.

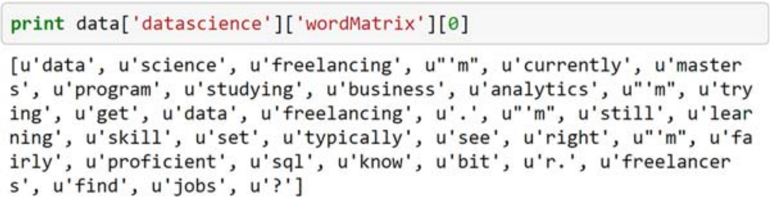
Call data processing function for every subreddit.

data['wordMatrix'] is a matrix comprised of word vectors; 1 vector per document.

Our data_processing() function takes in a SQL statement and returns the document-term matrix. It does this by looping through the data one entry (Reddit topic) at a time and combines the topic title and topic body text into a single word vector with the use of word tokenization. A *tokenizer* is a text handling script that cuts the text into pieces. You have many different ways to tokenize a text: you can divide it into sentences or words, you can split by space and punctuations, or you can take other characters into account, and so on. Here we opted for the standard NLTK word tokenizer. This word tokenizer is simple; all it does is split the text into terms if there's a space between the words. We then lowercase the vector and filter out the stop words. Note how the order is important here; a stop word in the beginning of a sentence wouldn't be filtered if we first filter the stop words before lowercasing. For instance in "I like Game of Thrones," the "I" would not be lowercased and thus would not be filtered out. We then create a word matrix (term-document matrix) and a list containing all the words. Notice how we extend the list without filtering for doubles; this way we can create a histogram on word occurrences during data exploration. Let's execute the function for our two topic categories.

Figure 8.15 shows the first word vector of the "datascience" category.

```
print data['datascience']['wordMatrix'][0]
```



```
print data['datascience']['wordMatrix'][0]
```

```
[u'data', u'science', u'freelancing', u"'m", u'currently', u'master
s', u'program', u'studying', u'business', u'analytics', u"'m", u'try
ing', u'get', u'data', u'freelancing', u'.', u"'m", u'still', u'lear
ning', u'skill', u'set', u'typically', u'see', u'right', u"'m", u'fa
irly', u'proficient', u'sql', u'know', u'bit', u'r.', u'freelancer
s', u'find', u'jobs', u'?']
```

Figure 8.15 The first word vector of the "datascience" category after first data processing attempt

This sure looks polluted: punctuations are kept as separate terms and several words haven't even been split. Further data exploration should clarify a few things for us.

8.3.5 Step 4: Data exploration

We now have all our terms separated, but the sheer size of the data hinders us from getting a good grip on whether it's clean enough for actual use. By looking at a single vector, we already spot a few problems though: several words haven't been split correctly and the vector contains many single-character terms. Single character terms might be good topic differentiators in certain cases. For example, an economic text will contain more $, £, and ¤ signs than a medical text. But in most cases these one-character terms are useless. First, let's have a look at the frequency distribution of our terms.

```
wordfreqs_cat1 = nltk.FreqDist(data['datascience']['all_words'])
plt.hist(wordfreqs_cat1.values(), bins = range(10))
plt.show()
wordfreqs_cat2 = nltk.FreqDist(data['gameofthrones']['all_words'])
plt.hist(wordfreqs_cat2.values(), bins = range(20))
plt.show()
```

By drawing a histogram of the frequency distribution (figure 8.16) we quickly notice that the bulk of our terms only occur in a single document.

Single-occurrence terms such as these are called *hapaxes*, and model-wise they're useless because a single occurrence of a feature is never enough to build a reliable model. This is good news for us; cutting these hapaxes out will significantly shrink our data without harming our eventual model. Let's look at a few of these single-occurrence terms.

```
print wordfreqs_cat1.hapaxes()
print wordfreqs_cat2.hapaxes()
```

Terms we see in figure 8.17 make sense, and if we had more data they'd likely occur more often.

```
print wordfreqs_cat1.hapaxes()
print wordfreqs_cat2.hapaxes()
```

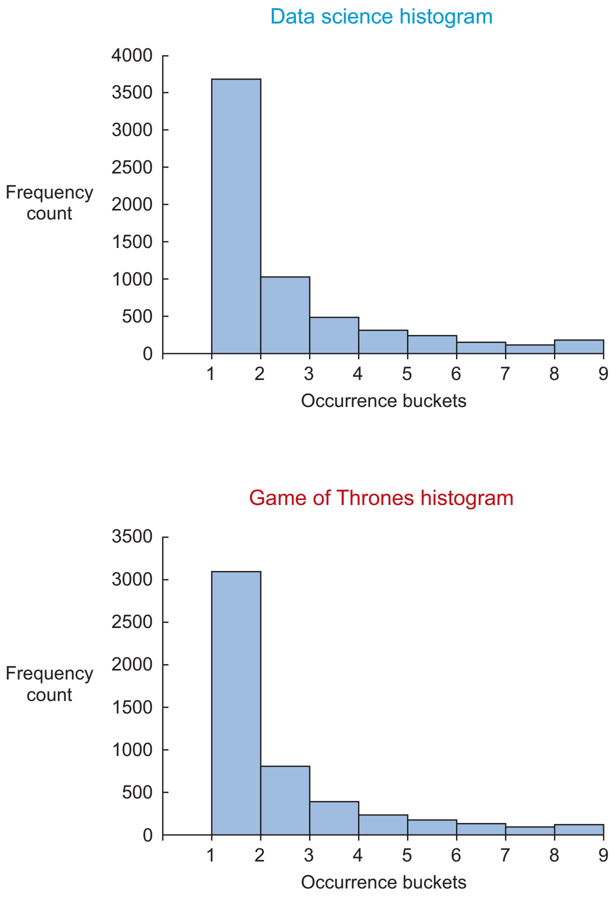
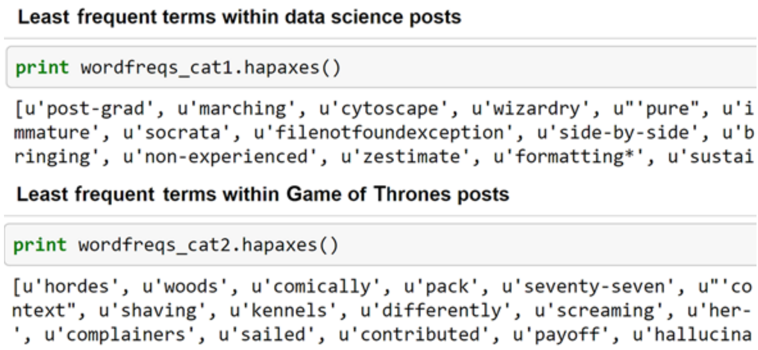


Figure 8.16 This histogram of term frequencies shows both the "data science" and "game of thrones" term matrices have more than 3,000 terms that occur once.



Least frequent terms within data science posts

```
print wordfreqs_cat1.hapaxes()
```

```
[u'post-grad', u'marching', u'cytoscape', u'wizardry', u"'pure", u'i
mmature', u'socrata', u'filenotfoundexception', u'side-by-side', u'b
ringing', u'non-experienced', u'zestimate', u'formatting*', u'sustai
```

Least frequent terms within Game of Thrones posts

```
print wordfreqs_cat2.hapaxes()
```

```
[u'hordes', u'woods', u'comically', u'pack', u'seventy-seven', u"'co
ntext", u'shaving', u'kennels', u'differently', u'screaming', u'her-
', u'complainers', u'sailed', u'contributed', u'payoff', u'hallucina
```

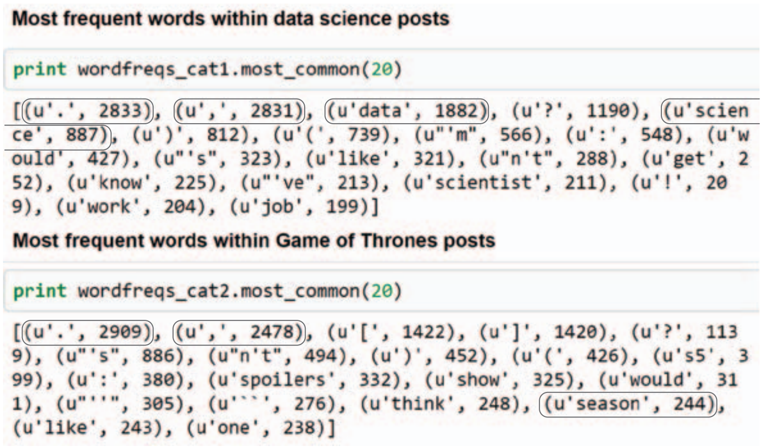
Figure 8.17 "Data science" and "game of thrones" single occurrence terms (hapaxes)

Many of these terms are incorrect spellings of otherwise useful ones, such as: Jaimie is Jaime (Lannister), Milisandre would be Melisandre, and so on. A decent *Game of Thrones*-specific thesaurus could help us find and replace these misspellings with a fuzzy search algorithm. This proves data cleaning in text mining can go on indefinitely if you so desire; keeping effort and payoff in balance is crucial here.

Let's now have a look at the most frequent words.

```
print wordfreqs_cat1.most_common(20)
print wordfreqs_cat2.most_common(20)
```

Figure 8.18 shows the output of asking for the top 20 most common words for each category.



Most frequent words within data science posts

```
print wordfreqs_cat1.most_common(20)
```

```
[(u'.', 2833), (u',', 2831), (u'data', 1882), (u'?', 1190), (u'scien
ce', 887), (u')', 812), (u'(', 739), (u"'m", 566), (u':', 548), (u'w
ould', 427), (u"'s", 323), (u'like', 321), (u"n't", 288), (u'get', 2
52), (u'know', 225), (u"'ve", 213), (u'scientist', 211), (u'!', 20
9), (u'work', 204), (u'job', 199)]
```

Most frequent words within Game of Thrones posts

```
print wordfreqs_cat2.most_common(20)
```

```
[(u'.', 2909), (u',', 2478), (u'[', 1422), (u']', 1420), (u'?', 113
9), (u"'s", 886), (u"n't", 494), (u')', 452), (u'(', 426), (u's5', 3
99), (u':', 380), (u'spoilers', 332), (u'show', 325), (u'would', 31
1), (u"''", 305), (u'``', 276), (u'think', 248), (u'season', 244),
(u'like', 243), (u'one', 238)]
```

Figure 8.18 Top 20 most frequent words for the "data science" and "game of thrones" posts

Now this looks encouraging: several common words do seem specific to their topics. Words such as "data," "science," and "season" are likely to become good differentiators. Another important thing to notice is the abundance of the single character terms such as "." and ","; we'll get rid of these.

With this extra knowledge, let's revise our data preparation script.

8.3.6 Step 3 revisited: Data preparation adapted

This short data exploration has already drawn our attention to a few obvious tweaks we can make to improve our text. Another important one is stemming the terms.

The following listing shows a simple stemming algorithm called "snowball stemming." These snowball stemmers can be language-specific, so we'll use the English one; however, it does support many languages.

Listing 8.6 The Reddit data processing revised after data exploration

```
stemmer = nltk.SnowballStemmer("english")
def wordStemmer(wordrow):
    stemmed = [stemmer.stem(word) for word in wordrow]
    return stemmed

manual_stopwords = [',','.',')',',','(','m',"'m","n't",'e.g',"'ve",'s','#','/
     ','``',"'s","''",'!','r',']','=','[','s','&','%','*','...','1','2','3','
     4','5','6','7','8','9','10','--',"''",';','-',':']

def data_processing(sql,manual_stopwords):
    #create pointer to the sqlite data
    c.execute(sql)
    data = {'wordMatrix':[],'all_words':[]}
    interWordMatrix = []
    interWordList = []

    row = c.fetchone()
    while row is not None:
        tokenizer = nltk.tokenize.RegexpTokenizer(r'\w+|[^\w\s]+')

        wordrow = tokenizer.tokenize(row[0]+" "+row[1])
        wordrow_lowercased = lowerCaseArray(wordrow)
        wordrow_nostopwords = wordFilter(stopwords,wordrow_lowercased)

        wordrow_nostopwords =
           wordFilter(manual_stopwords,wordrow_nostopwords)
        wordrow_stemmed = wordStemmer(wordrow_nostopwords)

        interWordList.extend(wordrow_stemmed)
        interWordMatrix.append(wordrow_stemmed)

        row = c.fetchone()
    wordfreqs = nltk.FreqDist(interWordList) s
    hapaxes = wordfreqs.hapaxes()
    for wordvector in interWordMatrix:
        wordvector_nohapexes = wordFilter(hapaxes,wordvector)
        data['wordMatrix'].append(wordvector_nohapexes)
        data['all_words'].extend(wordvector_nohapexes)

    return data

for subject in subreddits:
    data[subject] = data_processing(sql='''SELECT
     topicTitle,topicText,topicCategory FROM topics
WHERE topicCategory = '''+"'"+subject+"'",
manual_stopwords=manual_stopwords)
```

Initializes stemmer from NLTK library.

Stop words array defines terms to remove/ignore.

Now we define our revised data preparation.

row[0] and row[1] contain the title and text of the post, respectively. We combine them into a single text blob.

Fetch data (reddit posts) one by one from SQLite database.

Remove manually added stop words from text blob.

Temporary word list used to remove hapaxes later on.

Temporary word matrix; will become final word matrix after hapaxes removal.

Get new topic.

Make frequency distribution of all terms.

Get list of hapaxes.

Loop through temporary word matrix.

Remove hapaxes in each word vector.

Extend list of all terms with corrected word vector.

Append correct word vector to final word matrix.

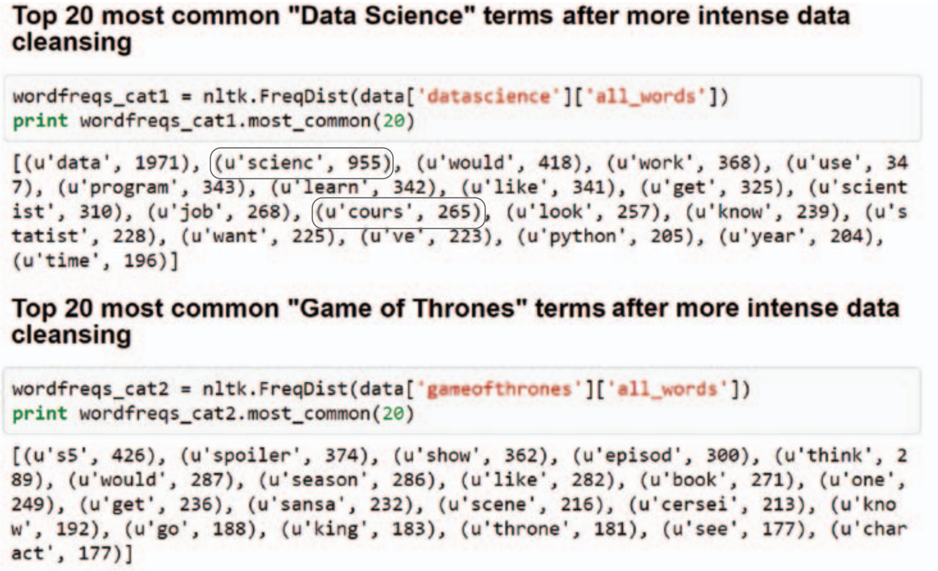
Run new data processing function for both subreddits.

Notice the changes since the last `data_processing()` function. Our tokenizer is now a regular expression tokenizer. Regular expressions are not part of this book and are often considered challenging to master, but all this simple one does is cut the text into words. For words, any alphanumeric combination is allowed (\w), so there are no more special characters or punctuations. We also applied the word stemmer and removed a list of extra stop words. And, all the hapaxes are removed at the end because everything needs to be stemmed first. Let's run our data preparation again.

If we did the same exploratory analysis as before, we'd see it makes more sense, and we have no more hapaxes.

```
print wordfreqs_cat1.hapaxes()
print wordfreqs_cat2.hapaxes()
```

Let's take the top 20 words of each category again (see figure 8.19).



Top 20 most common "Data Science" terms after more intense data cleansing

```
wordfreqs_cat1 = nltk.FreqDist(data['datascience']['all_words'])
print wordfreqs_cat1.most_common(20)
```

```
[(u'data', 1971), (u'scienc', 955), (u'would', 418), (u'work', 368), (u'use', 34
7), (u'program', 343), (u'learn', 342), (u'like', 341), (u'get', 325), (u'scient
ist', 310), (u'job', 268), (u'cours', 265), (u'look', 257), (u'know', 239), (u's
tatist', 228), (u'want', 225), (u've', 223), (u'python', 205), (u'year', 204),
(u'time', 196)]
```

Top 20 most common "Game of Thrones" terms after more intense data cleansing

```
wordfreqs_cat2 = nltk.FreqDist(data['gameofthrones']['all_words'])
print wordfreqs_cat2.most_common(20)
```

```
[(u's5', 426), (u'spoiler', 374), (u'show', 362), (u'episod', 300), (u'think', 2
89), (u'would', 287), (u'season', 286), (u'like', 282), (u'book', 271), (u'one',
249), (u'get', 236), (u'sansa', 232), (u'scene', 216), (u'cersei', 213), (u'kno
w', 192), (u'go', 188), (u'king', 183), (u'throne', 181), (u'see', 177), (u'char
act', 177)]
```

Figure 8.19 Top 20 most frequent words in "data science" and "game of thrones" Reddit posts after data preparation

We can see in figure 8.19 how the data quality has improved remarkably. Also, notice how certain words are shortened because of the stemming we applied. For instance, "science" and "sciences" have become "scienc;" "courses" and "course" have become "cours," and so on. The resulting terms are not actual words but still interpretable. If you insist on your terms remaining actual words, lemmatization would be the way to go.

With the data cleaning process "completed" (remark: a text mining cleansing exercise can almost never be fully completed), all that remains is a few data transformations to get the data in the bag of words format.

First, let's label all our data and also create a holdout sample of 100 observations per category, as shown in the following listing.

Listing 8.7 Final data transformation and data splitting before modeling

Holdout sample is comprised of unlabeled data from the two subreddits: 100 observations from each data set. The labels are kept in a separate data set.

Holdout sample will be used to determine the model's flaws by constructing a confusion matrix.

```
holdoutLength  = 100
```

We create a single data set with every word vector tagged as being either 'datascience' or 'gameofthrones.' We keep part of the data aside for holdout sample.

```
labeled_data1 = [(word,'datascience') for word in
     data['datascience']['wordMatrix'][holdoutLength:]]
labeled_data2 = [(word,'gameofthrones') for word in
     data['gameofthrones']['wordMatrix'][holdoutLength:]]
labeled_data = []
labeled_data.extend(labeled_data1)
labeled_data.extend(labeled_data2)

holdout_data = data['datascience']['wordMatrix'][:holdoutLength]
holdout_data.extend(data['gameofthrones']['wordMatrix'][:holdoutLength])
holdout_data_labels = ([('datascience')
for _ in xrange(holdoutLength)] + [('gameofthrones') for _ in
     xrange(holdoutLength)])
```

A list of all unique terms is created to build the bag of words data we need for training or scoring a model.

```
data['datascience']['all_words_dedup'] =
list(OrderedDict.fromkeys(
data['datascience']['all_words']))
data['gameofthrones']['all_words_dedup'] =
list(OrderedDict.fromkeys(
data['gameofthrones']['all_words']))
all_words = []
all_words.extend(data['datascience']['all_words_dedup'])
all_words.extend(data['gameofthrones']['all_words_dedup'])
all_words_dedup = list(OrderedDict.fromkeys(all_words))
```

Data is turned into a binary bag of words format.

```
prepared_data = [({word: (word in x[0]) for word
in all_words_dedup}, x[1]) for x in labeled_data]
prepared_holdout_data = [({word: (word in x[0])
for word in all_words_dedup})
for x in holdout_data]
```

Data for model training and testing is first shuffled.

Size of training data will be 75% of total and remaining 25% will be used for testing model performance.

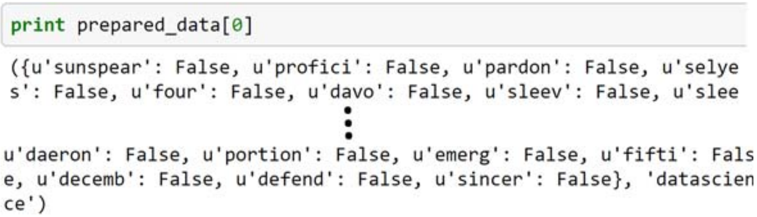
```
random.shuffle(prepared_data)
train_size = int(len(prepared_data) * 0.75)
train = prepared_data[:train_size]
test = prepared_data[train_size:]
```

The holdout sample will be used for our final test of the model and the creation of a confusion matrix. A *confusion matrix* is a way of checking how well a model did on previously unseen data. The matrix shows how many observations were correctly and incorrectly classified.

Before creating or training and testing data we need to take one last step: pouring the data into a bag of words format where every term is given either a “True” or “False” label depending on its presence in that particular post. We also need to do this for the unlabeled holdout sample.

Our prepared data now contains every term for each vector, as shown in figure 8.20.

```
print prepared_data[0]
```



```
print prepared_data[0]
```

```
({u'sunspear': False, u'profici': False, u'pardon': False, u'selye
s': False, u'four': False, u'davo': False, u'sleev': False, u'slee
                              ⋮
u'daeron': False, u'portion': False, u'emerg': False, u'fifti': Fals
e, u'decemb': False, u'defend': False, u'sincer': False}, 'datascien
ce')
```

Figure 8.20 A binary bag of words ready for modeling is very sparse data.

We created a big but sparse matrix, allowing us to apply techniques from chapter 5 if it was too big to handle on our machine. With such a small table, however, there’s no need for that now and we can proceed to shuffle and split the data into a training and test set.

While the biggest part of your data should always go to the model training, an optimal split ratio exists. Here we opted for a 3-1 split, but feel free to play with this. The more observations you have, the more freedom you have here. If you have few observations you’ll need to allocate relatively more to training the model. We’re now ready to move on to the most rewarding part: data analysis.

8.3.7 Step 5: Data analysis

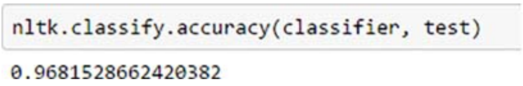
For our analysis we’ll fit two classification algorithms to our data: Naïve Bayes and decision trees. Naïve Bayes was explained in chapter 3 and decision tree earlier in this chapter.

Let’s first test the performance of our Naïve Bayes classifier. NLTK comes with a classifier, but feel free to use algorithms from other packages such as SciPy.

```
classifier  = nltk.NaiveBayesClassifier.train(train)
```

With the classifier trained we can use the test data to get a measure on overall accuracy.

```
nltk.classify.accuracy(classifier, test)
```



```
nltk.classify.accuracy(classifier, test)
```

```
0.9681528662420382
```

Figure 8.21 Classification accuracy is a measure representing what percentage of observations was correctly classified on the test data.

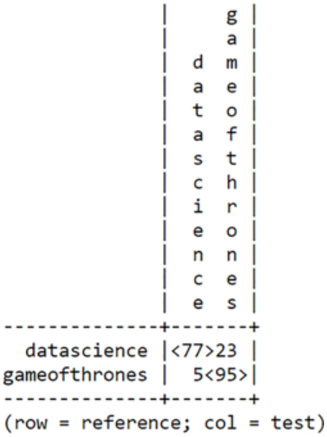
The accuracy on the test data is estimated to be greater than 90%, as seen in figure 8.21. *Classification accuracy* is the number of correctly classified observations as a percentage of the total number of observations. Be advised, though, that this can be different in your case if you used different data.

```
nltk.classify.accuracy(classifier, test)
```

That's a good number. We can now lean back and relax, right? No, not really. Let's test it again on the 200 observations holdout sample and this time create a confusion matrix.

```
classified_data = classifier.classify_many(prepared_holdout_data)
cm = nltk.ConfusionMatrix(holdout_data_labels, classified_data)
print cm
```

The confusion matrix in figure 8.22 shows us the 97% is probably over the top because we have 28 (23 + 5) misclassified cases. Again, this can be different with your data if you filled the SQLite file yourself.



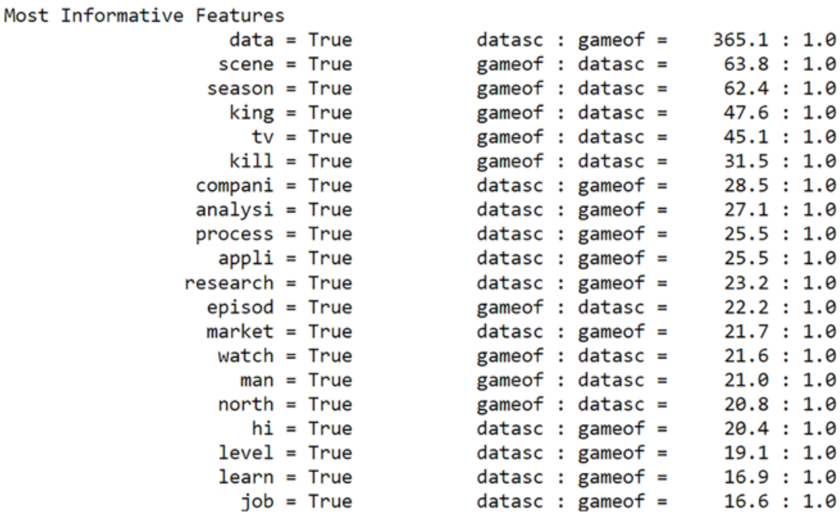
```
              |     g |
              |     a |
              | d   m |
              | a   e |
              | t   o |
              | a   f |
              | s   t |
              | c   h |
              | i   r |
              | e   o |
              | n   n |
              | c   e |
              | e   s |
--------------+-------+
  datascience |<77>23 |
gameofthrones |  5<95>|
--------------+-------+
(row = reference; col = test)
```

Figure 8.22 Naïve Bayes model confusion matrix shows 28 (23 + 5) observations out of 200 were misclassified

Twenty-eight misclassifications means we have an 86% accuracy on the holdout sample. This needs to be compared to randomly assigning a new post to either the "datascience" or "gameofthrones" group. If we'd randomly assigned them, we could expect an accuracy of 50%, and our model seems to perform better than that. Let's look at what it uses to determine the categories by digging into the most informative model features.

```
print(classifier.show_most_informative_features(20))
```

Figure 8.23 shows the top 20 terms capable of distinguishing between the two categories.



```
Most Informative Features
                    data = True           datasc : gameof =    365.1 : 1.0
                   scene = True           gameof : datasc =     63.8 : 1.0
                  season = True           gameof : datasc =     62.4 : 1.0
                    king = True           gameof : datasc =     47.6 : 1.0
                      tv = True           gameof : datasc =     45.1 : 1.0
                    kill = True           gameof : datasc =     31.5 : 1.0
                  compani = True           datasc : gameof =     28.5 : 1.0
                 analysi = True           datasc : gameof =     27.1 : 1.0
                 process = True           datasc : gameof =     25.5 : 1.0
                   appli = True           datasc : gameof =     25.5 : 1.0
                research = True           datasc : gameof =     23.2 : 1.0
                  episod = True           gameof : datasc =     22.2 : 1.0
                  market = True           datasc : gameof =     21.7 : 1.0
                   watch = True           gameof : datasc =     21.6 : 1.0
                     man = True           gameof : datasc =     21.0 : 1.0
                   north = True           gameof : datasc =     20.8 : 1.0
                      hi = True           datasc : gameof =     20.4 : 1.0
                   level = True           datasc : gameof =     19.1 : 1.0
                   learn = True           datasc : gameof =     16.9 : 1.0
                     job = True           datasc : gameof =     16.6 : 1.0
```

Figure 8.23 The most important terms in the Naïve Bayes classification model

The term "data" is given heavy weight and seems to be the most important indicator of whether a topic belongs in the data science category. Terms such as "scene," "season," "king," "tv," and "kill" are good indications the topic is *Game of Thrones* rather than data science. All these things make perfect sense, so the model passed both the accuracy and the sanity check.

The Naïve Bayes does well, so let's have a look at the decision tree in the following listing.

Listing 8.8 Decision tree model training and evaluation

```
classifier2 = nltk.DecisionTreeClassifier.train(train)
nltk.classify.accuracy(classifier2, test)
classified_data2 = classifier2.classify_many(prepared_holdout_data)
cm = nltk.ConfusionMatrix(holdout_data_labels, classified_data2)
print cm
```

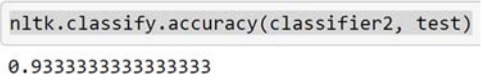
Train decision tree classifier

Test classifier accuracy

Attempt to classify holdout data (scoring)

Create confusion matrix based on classification results and actual labels

Show confusion matrix



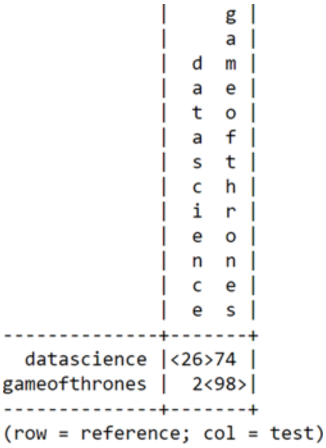
```
nltk.classify.accuracy(classifier2, test)

0.9333333333333333
```

Figure 8.24 Decision tree model accuracy

As shown in figure 8.24, the promised accuracy is 93%.

We now know better than to rely solely on this single test, so once again we turn to a confusion matrix on a second set of data, as shown in figure 8.25.

Figure 8.25 shows a different story. On these 200 observations of the holdout sample the decision tree model tends to classify well when the post is about *Game of Thrones* but fails miserably when confronted with the data science posts. It seems the model has a preference for *Game of Thrones*, and can you blame it? Let's have a look at the actual model, even though in this case we'll use the Naïve Bayes as our final model.



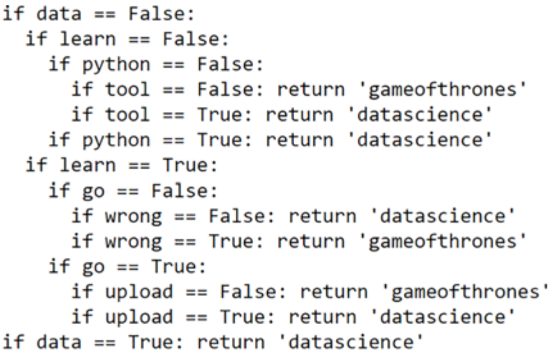
```
              |      g |
              |      a |
              |  d   m |
              |  a   e |
              |  t   o |
              |  a   f |
              |  s   t |
              |  c   h |
              |  i   r |
              |  e   o |
              |  n   n |
              |  c   e |
              |  e   s |
--------------+--------+
  datascience |<26>74  |
gameofthrones |  2<98> |
--------------+--------+
(row = reference; col = test)
```

Figure 8.25 Confusion matrix on decision tree model

```
print(classifier2.pseudocode(depth=4))
```

The decision tree has, as the name suggests, a tree-like model, as shown in figure 8.26.

The Naïve Bayes considers all the terms and has weights attributed, but the decision tree model goes through them sequentially, following the path from the root to the outer branches and leaves. Figure 8.26 only shows the top four layers, starting with the term "data." If "data" is present in the post, it's always data science. If "data" can't be found, it checks for the term "learn," and so it continues. A possible reason why this decision tree isn't performing well is the lack of pruning. When a decision tree is built it has many leaves, often too many. A tree is then pruned to a certain level to minimize overfitting. A big advantage of decision trees is the implicit interaction effects between words it



```
if data == False:
  if learn == False:
    if python == False:
      if tool == False: return 'gameofthrones'
      if tool == True: return 'datascience'
    if python == True: return 'datascience'
  if learn == True:
    if go == False:
      if wrong == False: return 'datascience'
      if wrong == True: return 'gameofthrones'
    if go == True:
      if upload == False: return 'gameofthrones'
      if upload == True: return 'datascience'
if data == True: return 'datascience'
```

Figure 8.26 Decision tree model tree structure representation

takes into account when constructing the branches. When multiple terms together create a stronger classification than single terms, the decision tree will actually outperform the Naïve Bayes. We won't go into the details of that here, but consider this one of the next steps you could take to improve the model.

We now have two classification models that give us insight into how the two contents of the subreddits differ. The last step would be to share this newfound information with other people.

8.3.8 Step 6: Presentation and automation

As a last step we need to use what we learned and either turn it into a useful application or present our results to others. The last chapter of this book discusses building an interactive application, as this is a project in itself. For now we'll content ourselves with a nice way to convey our findings. A nice graph or, better yet, an interactive graph, can catch the eye; it's the icing on the presentation cake. While it's easy and tempting to represent the numbers as such or a bar chart at most, it could be nice to go one step further.

For instance, to represent the Naïve Bayes model, we could use a force graph (figure 8.27), where the bubble and link size represent how strongly related a word is to the "game of thrones" or "data science" subreddits. Notice how the words on the bubbles are often cut off; remember this is because of the stemming we applied.

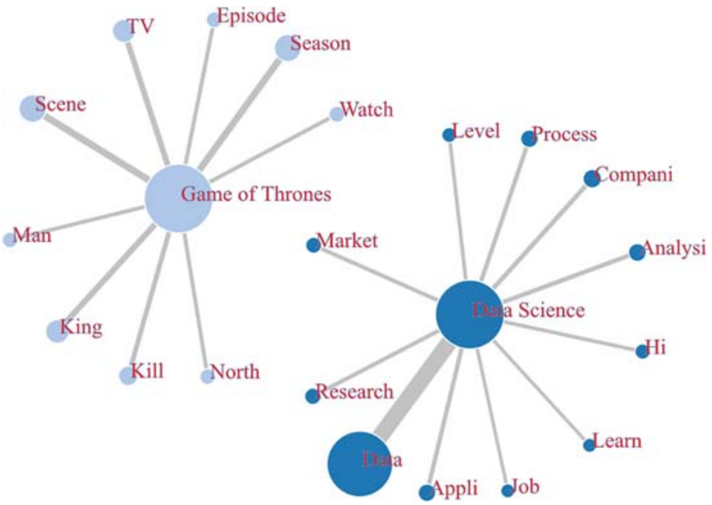


Figure 8.27 Interactive force graph with the top 20 Naïve Bayes significant terms and their weights

While figure 8.27 in itself is static, you can open the HTML file "forceGraph.html" to enjoy the d3.js force graph effect as explained earlier in this chapter. d3.js is outside of this book's scope but you don't need an elaborate knowledge of d3.js to use it. An extensive set of examples can be used with minimal adjustments to the code provided at https://github.com/mbostock/d3/wiki/Gallery. All you need is common sense and a minor knowledge of JavaScript. The code for the force graph example can found at http://bl.ocks.org/mbostock/4062045.

We can also represent our decision tree in a rather original way. We could go for a fancy version of an actual tree diagram, but the following sunburst diagram is more original and equally fun to use.

Figure 8.28 shows the top layer of the sunburst diagram. It's possible to zoom in by clicking a circle segment. You can zoom back out by clicking the center circle. The code for this example can be found at http://bl.ocks.org/metmajer/5480307.

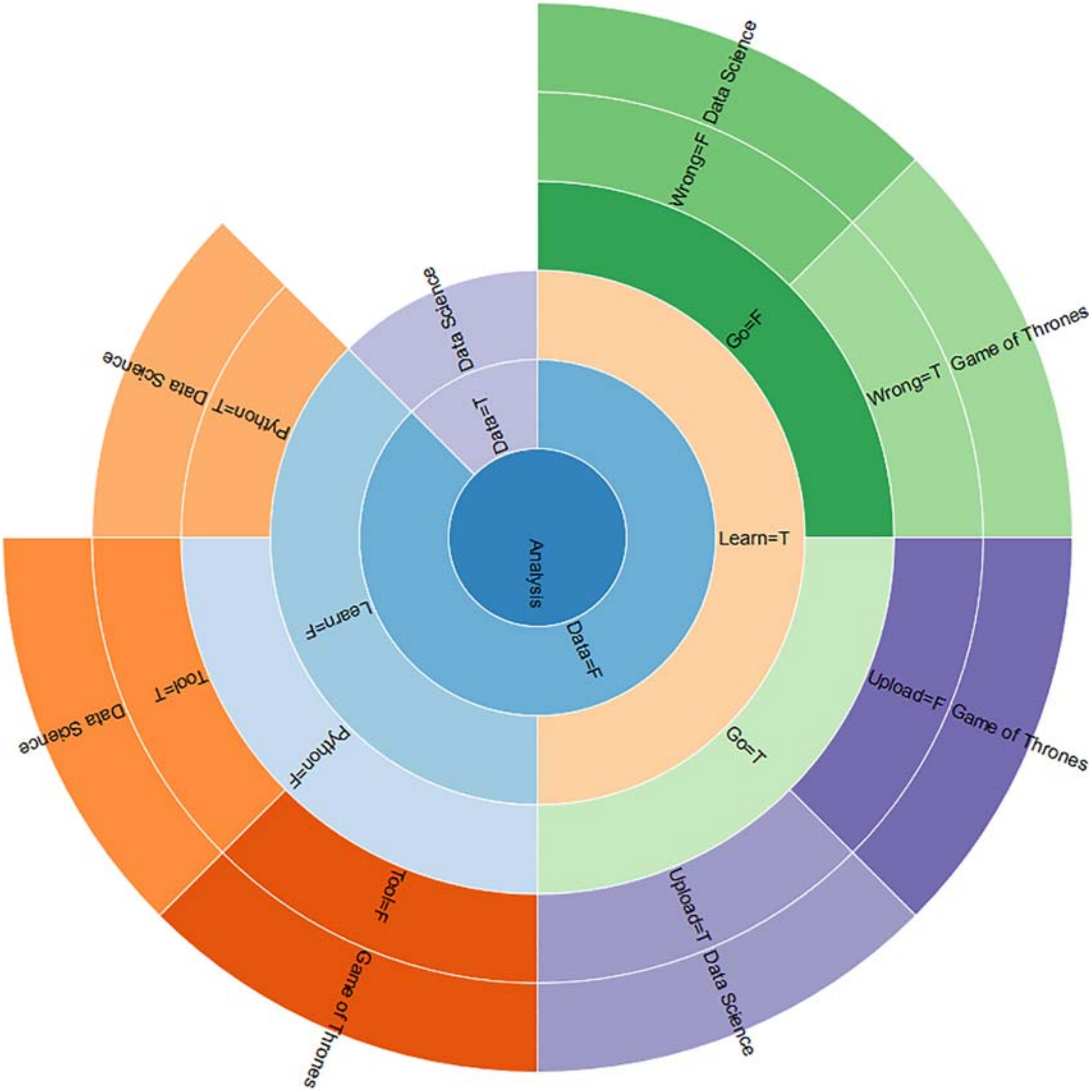


Figure 8.28 Sunburst diagram created from the top four branches of the decision tree model

Showing your results in an original way can be key to a successful project. People never appreciate the effort you've put into achieving your results if you can't communicate them and they're meaningful to them. An original data visualization here and there certainly helps with this.

8.4 Summary

- Text mining is widely used for things such as entity identification, plagiarism detection, topic identification, translation, fraud detection, spam filtering, and more.
- Python has a mature toolkit for text mining called NLTK, or the natural language toolkit. NLTK is good for playing around and learning the ropes; for real-life applications, however, Scikit-learn is usually considered more "production-ready." Scikit-learn is extensively used in previous chapters.
- The data preparation of textual data is more intensive than numerical data preparation and involves extra techniques, such as
  - *Stemming*—Cutting the end of a word in a smart way so it can be matched with some conjugated or plural versions of this word.
  - *Lemmatization*—Like stemming, it's meant to remove doubles, but unlike stemming, it looks at the meaning of the word.
  - *Stop word filtering*—Certain words occur too often to be useful and filtering them out can significantly improve models. Stop words are often corpus-specific.
  - *Tokenization*—Cutting text into pieces. Tokens can be single words, combinations of words (n-grams), or even whole sentences.
  - *POS Tagging*—Part-of-speech tagging. Sometimes it can be useful to know what the function of a certain word within a sentence is to understand it better.
- In our case study we attempted to distinguish Reddit posts on "Game of Thrones" versus posts on "data science." In this endeavor we tried both the Naïve Bayes and decision tree classifiers. Naïve Bayes assumes all features to be independent of one another; the decision tree classifier assumes dependency, allowing for different models.
- In our example, Naïve Bayes yielded the better model, but very often the decision tree classifier does a better job, usually when more data is available.
- We determined the performance difference using a confusion matrix we calculated after applying both models on new (but labeled) data.
- When presenting findings to other people, it can help to include an interesting data visualization capable of conveying your results in a memorable way.

Data science has become one of the hottest fields in technology. Firms worldwide are scrambling to find developers with data science skills to work on projects ranging from social media marketing to machine learning, but the prerequisite knowledge and experience for this career can seem bewildering. This book is designed to help you get started.

At its core, data science is a set of concepts and techniques for extracting meaning and clarity from enormous stored data sets or fast-moving data streams. Data scientists write programs to interpret these data. The Python programming language is a favorite tool of data scientists because it's easy to read and write, and it provides several high-value libraries that simplify core tasks like statistics, machine learning algorithms, and mathematics.

What's inside

- Get familiar with the most important data science concepts
- Use Python to work with data in common—and not-so-common—storage formats
- Write algorithms in Python
- Use Python tools such as iPython to make sense of big data
- Get hands on experience with the most common Python data science libraries such as Scikit-learn and StatsModels
- Use data science in a big data world

This book assumes you're comfortable reading code in Python or a similar language, such as C, Ruby, or JavaScript. No prior experience with data science is required.

Modeling dependencies with Bayesian and Markov networks

Probabilistic network models are good for tasks where you must infer multiple internal (or unobserved) states of the world from external observations, or when you wish to simulate processes for "what-if" scenarios. The following chapter provides an excellent introduction to network models and to the probabilistic reasoning that underlies them. The implementation details may be difficult to follow if you're not familiar with the Figaro probabilistic programming language, but the motivating examples clearly explain the underlying concepts. You'll also get a good idea of the types of decision processes that Figaro can represent.

Chapter 5 from *Practical Probabilistic Programming*
by Avi Pfeffer

Modeling dependencies with Bayesian and Markov networks

This chapter covers

- Types of relationships among variables in a probabilistic model and how these relationships translate into dependencies
- How to express these various types of dependencies in Figaro
- Bayesian networks: models that encode directed dependencies among variables
- Markov networks: models that encode undirected dependencies among variables
- Practical examples of Bayesian and Markov networks

In chapter 4, you learned about the relationships between probabilistic models and probabilistic programs, and you also saw the ingredients of a probabilistic model, which are variables, dependencies, functional forms, and numerical parameters. This chapter focuses on two modeling frameworks: Bayesian networks and Markov networks. Each framework is based on a different way of encoding dependencies.

Dependencies capture relationships between variables. Understanding the kinds of relationships and how they translate into dependencies in a probabilistic model is one of the most important skills you can acquire for building models. Accordingly, you'll learn all about various kinds of relationships and dependencies. In general, there are two kinds of dependencies between variables: directed dependencies, which express asymmetric relationships, and undirected dependencies, which turn into symmetric relationships. Bayesian networks encode directed dependencies, whereas Markov networks encode undirected dependencies. You'll also learn how to extend traditional Bayesian networks with programming language capabilities to benefit from the power of probabilistic programming.

After you understand the material in this chapter, you'll have a solid knowledge of the essentials of probabilistic programming. All probabilistic models boil down to a collection of directed and undirected dependencies. You'll know when to introduce a dependency between variables, whether to make it directed or undirected, and, if it's directed, what direction it should take. Chapter 6 builds on this knowledge to create more-complex models using data structures, and chapter 7 will further extend your skills with object-oriented modeling.

This chapter assumes you have a basic knowledge of Figaro, as appears in chapter 2. In particular, you should have familiarity with `Chain`, which underlies directed dependencies, and conditions and constraints, which are the basis for undirected dependencies. You'll also use the `CPD` and `RichCPD` elements that appear in chapter 4. Don't worry if you don't remember these concepts; I'll remind you of them here when you see them.

5.1 Modeling dependencies

Probabilistic reasoning is all about using dependencies between variables. Two variables are *dependent* if knowledge of one variable provides information about the other variable. Conversely, if knowing something about one variable tells you nothing about the other variable, the variables are *independent.*

Consider a computer system diagnosis application in which you're trying to reason about faults in a printing process. Suppose you have two variables, Printer Power Button On and Printer State. If you observe that the power button is off, you can infer that the printer state is down. Conversely, if you observe that the printer state is down, you can infer that the power button might be off. These two variables are clearly dependent.

Dependencies are used to model variables that are related in some way. Many kinds of relationships between variables exist, but only two kinds of dependencies:

- *Directed dependencies* go from one variable to the other. Typically, these model a cause-effect relationship between the variables.
- *Undirected dependencies* model relationships between variables where there's no obvious direction of influence between them.

The next two subsections describe both kinds of dependencies in detail and give plenty of examples.

5.1.1 Directed dependencies

Directed dependencies lead from one variable to another and are typically used to represent cause-effect relationships. For example, the printer's power button being off causes the printer to be down, so a direct dependency exists between Printer Power Button On and Printer State. Figure 5.1 illustrates this dependency, with a directed edge between the two variables. (*Edge* is the usual term for an arrow between two nodes in a graph.) The first variable (in this case, Printer Power Button On) is called the *parent*, and the second variable (Printer State) is called the *child*.

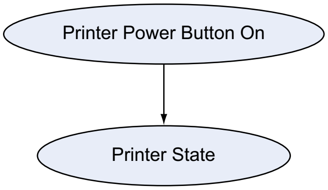


Figure 5.1 Directed dependency expressing cause-effect relationship

Why does the arrow go from cause to effect? A simple reason is that causes tend to happen before effects. More deeply, the answer is closely related to the concept of the generative model explored in chapter 4. Remember, a generative model describes a process for generating values of all of the variables in your model. Typically, the generative process simulates a real-world process. If a cause leads to an effect, you want to generate the value of the cause first, and use that value when you generate the effect. In our example, if you create a model of a printer, and imagine generating values for all of the variables in the model, you'll first generate a value for Printer Power Button On and then use this value to generate Printer State.

Now, it bears repeating that *the direction of a dependency isn't necessarily the direction of reasoning*. You can reason from the printer power button being off to the printer state being down, but you can also reason in the opposite direction: if the printer state is up, you know for sure that the power button isn't off. Many people make the mistake of constructing their models in the direction they intend to reason. In a diagnosis application, you might observe that the printer is down and try to determine its causes, so you'd reason from Printer State to Printer Power Button On. You might be tempted to make the arrow go from Printer State to Printer Power Button On. This is incorrect. The arrow should express the generative process, which follows the cause-effect direction.

I've said that directed dependencies typically model cause-effect relationships. In fact, cause-effect is just one example of a general class of asymmetric relationships between variables. Let's have a closer look at various kinds of asymmetric relationships—first, cause-effect relationships, and then other kinds.

Varieties of cause-effect relationships

Here are some kinds of cause-effect relationships:

- *What happens first to what happens next*—The most obvious kind of cause-effect relationship is between one thing that leads to another thing at a later time. For example, if someone turns the printer power off, then after that, the printer will be down. This temporal relationship is such a common characteristic of cause-effect relationships that you might think all cause-effect relationships involve time, but I don't agree with this.
- *Cause-effect of states*—Sometimes you can have two variables that represent different aspects of the state of the situation at a given point in time. For example, you might have one variable representing whether the printer power button is off and another representing whether the printer is down. Both of these are states that hold at the same moment in time. In this example, the printer power button being off causes the printer to be down, because it makes the printer have no power.
- *True value to measurement*—Whenever one variable is a *measurement* of the value of another variable, you say that the true value is a cause of the measurement. For example, suppose you have a Power Indicator Lit variable that represents whether the printer's power LED is lit. An asymmetric relationship exists from Printer Power Button On to Power Indicator Lit. Typically, measurements are produced by sensors, and there may be more than one measurement of the same value. Also, measurements are usually observed, and you want to reason from the measurements to the true values, so this is another example of the direction of the dependencies being different from the direction of reasoning.
- *Parameter to variable that uses the parameter*—For example, consider the bias of a coin, representing the probability that a toss will come out heads, and a toss of that coin. The toss uses the bias to determine the outcome. It's clear that the bias is generated first, and only then the individual toss. And when there are many tosses of the same coin, they're all generated after the bias.

Additional asymmetric relationships

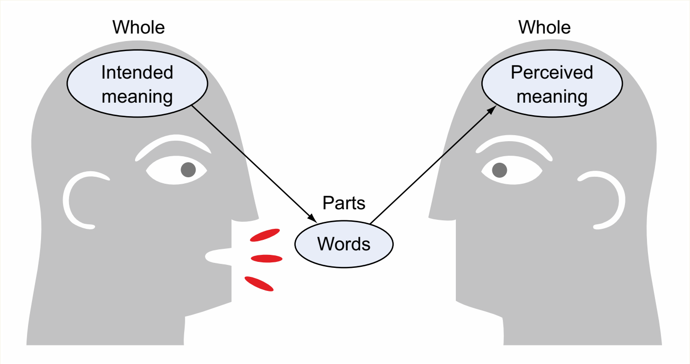
The preceding cases are by far the most important and least ambiguous. If you understand these cases, you'll be 95% of the way to determining the correct direction of dependencies. Now let's go deeper by considering a variety of other relationships that, although obviously asymmetric, are ambiguous about the direction of dependency. I'll list these relationships and then describe a rule of thumb that can help you resolve the ambiguity.

- *Part to whole*—Often, the properties of part of an object lead to properties of the object as a whole. For example, consider a printer with toner and a paper feeder. Faults with either the toner or paper feeder, which are parts of the printer, can lead to faults with the printer as a whole. Other times, properties of the whole can determine properties of the part. For example, if the printer as a whole is badly made, both the paper feeder and the toner will likely be badly made.

- *Specific to general*—This one can also go both ways. A user can experience a printer problem in lots of ways, such as a paper jam or poor print quality. If the user experiences any of these specific problems, that user will experience the more general problem of a poor printing experience. In this case, the specific causes the general. On the other hand, imagine the process of generating an object. Typically, you generate the general properties of the object before refining them with specifics. For example, when generating a printer, you might first decide whether it's a laser or inkjet printer before generating its individual properties. Indeed, it doesn't make sense to generate the specific properties, which might be relevant for only a particular kind of printer, before you know what kind of printer it is.
- *Concrete/detailed to abstract/summary*—An example of a concrete-abstract relationship is between a score on a test and a letter grade. Many scores correspond to the same letter grade. Clearly, the teacher bases the letter grade on the test score, not the other way around, so the test score causes the letter grade. On the other hand, consider the process of generating a student and test results. You might first generate the abstract kind of student (for example, an A student or B student), and then generate the concrete test score.

Disambiguating cause-effect relationships

As you can see from the preceding examples, although it's clear that an asymmetric dependency exists in these cases, teasing out the direction of the dependency can be tricky. Here's an idea that can help. Imagine that someone is speaking a sentence in English to another person, as shown in the following illustration. The speaker has a certain meaning in mind for the whole sentence. From this meaning, the speaker generates words. This is a whole-part relationship, from the sentence to its words. Then the sentence is heard by somebody else. This person puts together the perceived meaning of the sentence from the words, creating a part-whole relationship.



Communication of a sentence: the speaker's intended meaning of the whole sentence generates the words (parts), which in turn generate the hearer's perceived meaning of the whole sentence.

(continued)

If you look at this example closely, you can see that in the process of making the sentence, the meaning of the whole sentence is made before the individual words. But in the process of perceiving the sentence, the meaning of the sentence is perceived after the individual words. This isn't an ironclad rule, but often when you make something, first you make the general/abstract/whole thing and then refine it to produce the specific/concrete/detailed/many-parted thing. When you generate a printer, you first generate the general class of the whole printer. Then you generate the specific type of printer, and also detailed information about the components of the printer. Similarly, when generating the student, you first generate the abstract class of the student and then fill in the concrete test score. On the other hand, when you perceive and report something, you first perceive specific/concrete/detailed information about its parts, and then derive and summarize general/abstract properties about its whole. For example, the user who experiences a specific printer problem may summarize it in a general way, or a teacher grading a student will first observe the test score before reporting the letter grade. So here's the rule of thumb:

> *If you're modeling the making or definition of properties, dependencies go from the general, abstract, or whole concept to the specific, concrete, or parts. If you're modeling the perception and reporting of properties, dependencies go from the specific, concrete, or parts to the general, abstract, or whole.*

As I said earlier, if you understand the main cause-effect relationships, you'll get the directions right almost all of the time. In the exceptions to this rule, even experienced modelers can disagree on the appropriate direction of the dependencies. I hope that the rule of thumb I gave you will provide some guidance to help you build models.

DIRECTED DEPENDENCIES IN FIGARO

Remember the four ingredients of a probabilistic model? They're variables, dependencies, functional forms, and numerical parameters. Until now, you've assumed that the variables are given and focused on the dependencies. When you want to express these dependencies in Figaro, you need to provide a functional form and specify numerical parameters.

You can express directed dependencies in Figaro in a variety of ways. The general principle is to use a `Chain` of some sort as the functional form. Remember that `Chain` has two arguments:

- The parent element
- A chain function from a value of the parent element to a result element

`Chain(parentElement, chainFunction)` defines the following generative process: get a value for the parent from `parentElement`, then apply `chainFunction` to obtain `resultElement`, and finally get a value from `resultElement`.

When expressing a directed dependency by using `Chain`, the `parentElement` is, naturally, the parent. The chain function specifies a probability distribution over the child for each value of the parent. Figaro has a lot of constructs that use `Chain`, so in many cases you're using `Chain` even if it doesn't look like it.

Here are some equivalent examples, all expressing that if the printer power button is off, the printer will definitely be down, but if the power button is on, the printer could be up or down:

```
val printerPowerButtonOn = Flip(0.95)

val printerState =
  Chain(printerPowerButtonOn,
        (on: Boolean) =>
          if (on) Select(0.2 -> 'down, 0.8 -> 'up)
          else Constant('down)
```

Using Chain

Parent element

Argument to chain function

Body of chain function

```
val printerState =
  If(printerPowerButtonOn,
     Select(0.2 -> 'down, 0.8 -> 'up),
     Constant('down))
```

Using If

Parent element

Outcome if parent is true

Outcome if parent is false

```
val printerState =
  CPD(printerPowerButtonOn,
      false -> Constant('down),
      true -> Select(0.2 -> 'down, 0.8 -> 'up))
```

Using CPD

Parent element

Clause specifying outcome if parent is false

Clause specifying outcome if parent is true

```
val printerState =
  RichCPD(printerPowerButtonOn,
          OneOf(false) -> Constant('down),
          * -> Select(0.2 -> 'down, 0.8 -> 'up))
```

Using RichCPD

Parent element

Clause specifying outcome if the parent value is in the set { false }

Clause specifying outcome if parent has any other value

As mentioned earlier, one kind of asymmetric relationship is between a parameter and a variable that depends on the parameter, such as the bias of a coin and a toss of that coin. Again, this can be expressed using a `Chain` or using a compound version of an atomic element. Here are a couple of equivalent examples:

```
val toss = Chain(bias, (d: Double) => Flip(d))

val toss = Flip(bias)
```

5.1.2 Undirected dependencies

You've seen that directed dependencies can represent a variety of asymmetric relationships. Undirected dependencies model relationships between variables where there's no obvious direction between them. These kinds of relationships are called *symmetric relationships*. If you have variables that are correlated, but there's no obvious generative process whereby one variable is generated before the other, an undirected dependency might be appropriate. Symmetric relationships can arise in two ways:

- *Two effects of the same cause, where the cause isn't modeled explicitly*—For example, two measurements of the same value, when you don't have a variable for the value, or two consequences of the same event, when you don't have a variable for the event. Clearly, if you don't know the underlying value of an event, the two measurements or consequences are related. Imagine, in the printer scenario, that you have separate variables for the print quality and speed of printing. If you didn't have a variable representing the state of the printer, these two variables would be related, because they're two aspects of printing that might have the same underlying cause.

  You might ask, why don't we include a variable for the cause in our model? One possible answer is that the cause is much more complex than the effects and would be difficult to model accurately. In this chapter, you'll see an example of image reconstruction. The image is a two-dimensional effect of a complex three-dimensional scene. It might be harder to create a correct probabilistic model of three-dimensional scenes than to model the relationships between pixels in the image.
- *Two causes of the same known effect*—This one's interesting. Usually, there's no relationship between two causes of the same effect. For example, the paper feeder status of a printer and the toner level both influence the status of the printer as a whole, but the paper feeder status and toner level are independent. But if you learn that the printer isn't printing well, suddenly the paper feeder status and toner level become dependent. If you learn that the toner is low, that might lead you to believe that the reason the print quality is poor is due to low toner rather than obstructed paper flow. In this example, the overall printer status is the effect, and the toner level and paper feeder status are the possible causes, and the two causes become dependent when the effect is known. This is an example of an *induced dependency*, which you'll learn about in more detail in section 5.2.3. If you don't have a variable for the effect, this becomes a symmetric relationship between the two causes. But it's less usual to leave the common effect of the two causes out of the model.

EXPRESSING UNDIRECTED DEPENDENCIES IN FIGARO

You can express asymmetric relationships in Figaro in two ways: using constraints and using conditions. Each has an advantage and a disadvantage. The advantage of the constraints method is that it's conceptually simpler. But the numbers that go in the constraints are hardcoded and can't be learned in Figaro because they aren't accessible to a learning algorithm. The advantage of the method that uses conditions is that the numbers can be learned.

The basic principle behind the two approaches is similar. Section 5.5 describes how undirected dependencies are encoded in detail, but here's the short version.

When an undirected dependency exists between two variables, some joint values of the two variables are preferred to others. This can be achieved by assigning weights to the different joint values. A constraint encodes the weights by specifying a function from a joint value of the two variables to a real number representing the weight of that value. In the conditions approach, essentially the same information is encoded but in a more complex way.

For example, let's encode a relationship between two adjacent pixels in an image. These relationships have an "all else being equal" nature. For example, you might believe that, all else being equal, the two pixels are three times as likely to have the same color as they are to have different colors. In actuality, many relationships may affect the colors of the two pixels, so it's not actually three times as likely that they have the same color.

Here's how to express this relationship in Figaro using the constraints approach. Let's call the two pixel colors `color1` and `color2`. To keep the example simple, you'll assume that colors are Boolean. Remember that a constraint is a function from the value of an element to a `Double`. You need to create an element that represents the pair of `color1` and `color2` so you can specify a constraint on the pair. You can do this as follows:

```
import com.cra.figaro.library.compound.^^
val pair = ^^(color1, color2)
```

^^ is the Figaro pair constructor.

Now you have to define the function that implements the constraint:

```
def sameColorConstraint(pair: (Boolean, Boolean)) =
  if (pair._1 == pair._2) 0.3; else 0.1
```

The test checks whether the first component of the pair equals the second component.

Finally, you apply the constraint to the pair of colors:

```
pair.setConstraint(sameColorConstraint _)
```

The underscore indicates that you want the function itself and not to apply it.

Figaro interprets the constraint exactly how you'd expect. All else being equal, a state in which the two components of the pair are equal (in other words, both colors are equal) will be three times as likely as one where they're unequal. These constraints come together in defining the probability distribution in the correct way, as defined in section 5.5. Unfortunately, however, the numbers 0.3 and 0.1 are buried inside the constraint function, so they can't be learned from data.

For the conditions approach, I'll show you how it works in code and then explain why it's correct. First, you define an auxiliary Boolean element, which you'll call `sameColorConstraintValue`. You then define it so that the probability that it comes out true is equal to the value of the constraint. Finally, you'll add a condition that says that this auxiliary element must be true. This can be achieved using an observation:

```
val sameColorConstraintValue =
  Chain(color1, color2,
        (b1: Boolean, b2: Boolean) =>
          if (b1 == b2) Flip(0.3); else Flip(0.1))
sameColorConstraintValue.observe(true)
```

This code is equivalent to the constraints version. To see this, realize that a condition causes any value that violates the condition to have probability 0; otherwise, it has probability 1. The overall probability of any state is obtained by combining the probabilities resulting from the definition of elements and from conditions or constraints. For example, suppose that the two colors are equal. Using the definition of Chain, sameColorConstraintValue will come out true with probability 0.3 and false with probability 0.7. If sameColorConstraintValue comes out true, the condition will have probability 1, whereas if it's false, the condition will have probability 0. Therefore, the combined probability of the condition is (0.3 × 1) + (0.7 × 0) = 0.3. Similarly, if the two colors are unequal, the combined probability is (0.1 × 1) + (0.9 × 0) = 0.1. So you see that the colors being the same is three times as likely, all else being equal, as the colors being different. This is exactly the same result as the one you got with the constraint.

This construction using conditions is general enough to cover all asymmetric relationships. The advantage of this approach, as you can see, is that the numbers 0.3 and 0.1 are inside Flip elements, so you could make them depend on other aspects of the model. For example, suppose you wanted to learn the degree to which two adjacent pixels are more likely to be the same color than different. You can create an element, perhaps defined using a beta distribution, to represent the weight. You can then use this element inside the Flip instead of 0.3. Then, given data, you can learn a distribution over the value of the weight.

5.1.3 Direct and indirect dependencies

Before you move on to look at Bayesian and Markov networks, I want to make an important point. A typical probabilistic model has many pairs of variables, and so knowledge of one variable changes your beliefs about the other. By definition, these are all cases of a dependency between variables. But most of these dependencies are *indirect*: they don't go directly between two variables but instead go through some intermediary variables. To be precise, the reason knowledge about the first variable changes your beliefs about the second is because knowledge about the first variable changes your beliefs about the intermediary variables, which in turn changes your beliefs about the second variable.

For example, look at figure 5.2, which has three variables: Printer Power Button On, Printer State, and Number of Printed Pages. Clearly, Printer Power Button On and Number of Printed Pages have a dependency. If you know that the power button

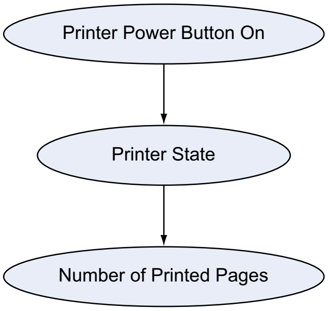


Figure 5.2 Printer Power Button On has an indirect relationship with Number of Printed Pages that goes through the intermediary variable Printer State.

is off, you'll believe that the number of printed pages is zero. But in the figure, the dependency from Printer Power Button On to Number of Printed Pages passes through an intermediary variable, which is Printer State. This means that the reason knowledge about Printer Power Button On changes your beliefs about Number of Printed Pages is that it first changes your beliefs about Printer State, and the changed beliefs about Printer State in turn change your beliefs about Number of Printed Pages. If you know the power button is off, that tells you that the printer is down, which leads you to believe that no pages will be printed.

> **TERMINOLOGY ALERT** Earlier I talked about directed and undirected dependencies, and now I'm talking about direct and indirect dependencies. Although the names are similar, they have different meanings. A *direct* dependency goes directly between two variables; its antonym is indirect, which goes through intermediary variables. A *directed* dependency has a direction from one variable to another, as opposed to an undirected dependency that has no direction. You can have a direct undirected dependency and an indirect directed dependency.

Let's look at another example, this time involving undirected dependencies. Earlier, I gave an example of adjacent pixels in an image that have an undirected dependency between them. What about nonadjacent pixels? Consider the example in figure 5.3. If you know that pixel 11 is red, that will lead you to believe that pixel 12 is likely to be red, which will in turn lead you to believe that pixel 13 is likely to be red. This is an obvious example of an indirect dependency, because knowledge about pixel 11 influences beliefs about pixel 13 only through the intermediary variable pixel 12.

It's important to recognize which dependencies in your domain are direct and which are indirect. In both Bayesian and Markov networks, you create a graph with

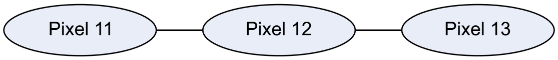


Figure 5.3 Pixel 11 has an indirect relationship with pixel 13 that goes through the intermediary variable pixel 12.

edges between variables to represent the dependencies. In a Bayesian network, these are directed edges, whereas in a Markov network, they're undirected. You draw an edge only for direct dependencies. If two variables have only an indirect dependency, you don't draw an edge between them.

Next, you'll look at Bayesian networks, which are models that represent directed dependencies, and then at Markov networks, which are models that represent undirected dependencies. I do want to point out, however, that just because there are separate modeling frameworks for directed and undirected dependencies doesn't mean you have to choose one or the other for your model. Other frameworks combine both kinds of dependencies in a single model. It's easy to do that in a probabilistic program: you use the generative definition of elements to encode directed dependencies, and add any constraints you want to express undirected dependencies.

5.2 Using Bayesian networks

You've seen that encoding relationships between variables is essential to probabilistic modeling. In this section, you'll learn about Bayesian networks, which are the standard framework for encoding asymmetric relationships using directed dependencies. You've already seen Bayesian networks in chapter 4, in the context of the Rembrandt example. This section provides a more thorough treatment, including a full definition and an explanation of the reasoning patterns you can use.

5.2.1 Bayesian networks defined

A Bayesian network is a representation of a probabilistic model consisting of three components:

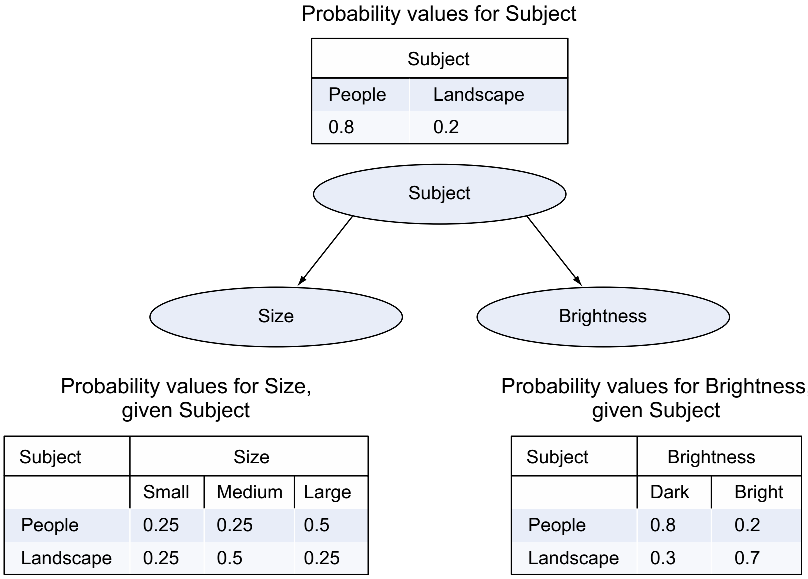
- A set of variables with their corresponding domains
- A directed acyclic graph in which each variable is a node
- For each variable, a conditional probability distribution (CPD) over the variable, given its parents

SET OF VARIABLES WITH CORRESPONDING DOMAINS

The example in figure 5.4 shows three variables: Subject, Size, and Brightness. The domain of a variable specifies which values are possible for that variable. The domain of Subject is {People, Landscape}, the domain of Size is {Small, Medium, Large}, and the domain of Brightness is {Dark, Bright}.

DIRECTED ACYCLIC GRAPH

Directed means that each edge in the graph has a direction; it goes from one variable to another. The first variable is called the *parent*, and the second variable is called the *child*. In figure 5.4 Subject is a parent of both Size and Brightness. The word acyclic means that there are no cycles in the graph: there are no *directed cycles* that follow the direction of the arrows; you can't start at a node, follow the arrows, and end up at the same node. But you can have an *undirected cycle* that would be a cycle if you ignored



Probability values for Subject

Subject	
People	Landscape
0.8	0.2

Probability values for Size, given Subject

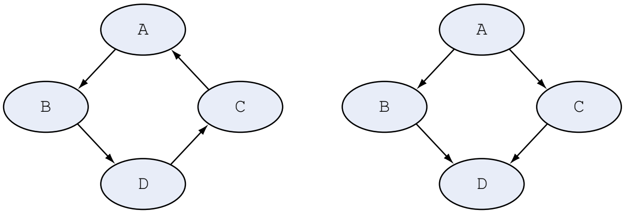
Subject	Size		
	Small	Medium	Large
People	0.25	0.25	0.5
Landscape	0.25	0.5	0.25

Probability values for Brightness, given Subject

Subject	Brightness	
	Dark	Bright
People	0.8	0.2
Landscape	0.3	0.7

Figure 5.4 A three-node Bayesian network

the directions of the edges. This concept is illustrated in figure 5.5. The graph on the left has a directed cycle A-B-D-C-A. In the graph on the right, the cycle A-B-D-C-A sometimes runs counter to the direction of the arrows, so it's an undirected cycle. Therefore, the graph on the left isn't allowed, but the graph on the right is allowed. This point is important, because later, when you allow undirected edges to express symmetric dependencies, you'll be allowed to have undirected cycles.



Directed cycle

No directed cycle

Figure 5.5 A directed cycle is a cycle that follows the arrows and ends up where it started.

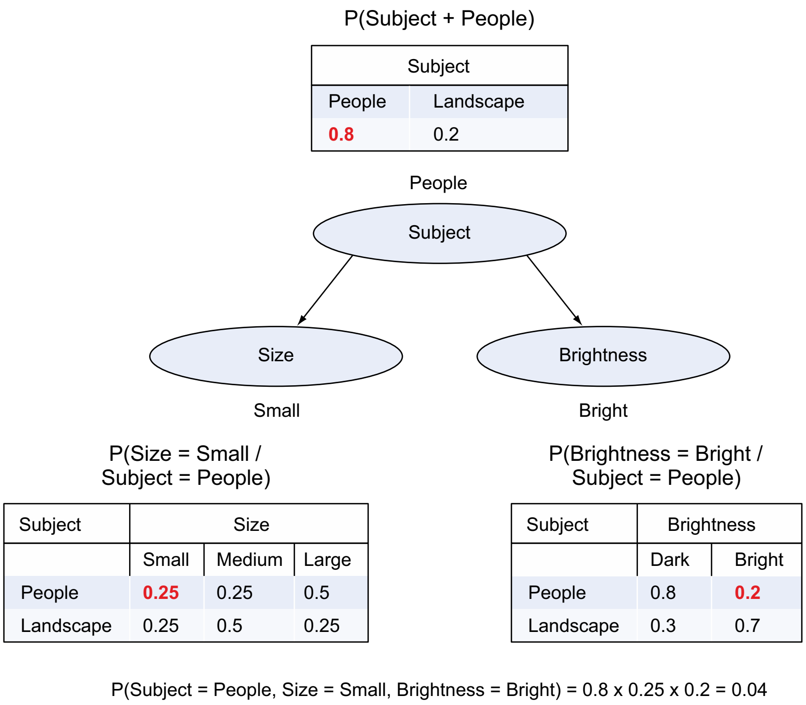
Conditional probability distribution over the variable

A CPD specifies a probability distribution over the child variable, given the values of its parents. This CPD considers every possible assignment of values to the parents, when the value of a parent can be any value in its domain. For each assignment, it defines a probability distribution over the child. In figure 5.6, each variable has a CPD. Subject is a root of the network, so it has no parents. When a variable has no parents, the CPD specifies a single probability distribution over the variable. In this example, Subject takes the value People with probability 0.8 and Landscape with probability 0.2. Size has a parent, which is Subject, so its CPD has a row for each value of Subject. The CPD says that when Subject has value People, the distribution over Size makes Size have value Small with probability 0.25, Medium with probability 0.25, and Large with probability 0.5. When Subject has value Landscape, Size has a different distribution. Finally, Brightness also has parent Subject, and its CPD also has a row for each value of Subject.

5.2.2 How a Bayesian network defines a probability distribution

That's all there is to the definition of Bayesian networks. Now, let's see how a Bayesian network defines a probability distribution. The first thing you need to do is define the possible worlds. For a Bayesian network, a possible world consists of an assignment of values to each of the variables, making sure the value of each variable is in its domain. For example, <Subject = People, Size = Small, Brightness = Bright> is a possible world.

Next, you define the probability of a possible world. This is simple. All you have to do is identify the entry in the CPD of each variable that matches the values of the parents and child in the possible world. The process is illustrated in figure 5.6. For


P(Subject + People)
People
Subject
Size
Brightness
Small
Bright
P(Size = Small / Subject = People)
P(Brightness = Bright / Subject = People)

Subject	
People	Landscape
0.8	0.2

Subject	Size		
	Small	Medium	Large
People	0.25	0.25	0.5
Landscape	0.25	0.5	0.25

Subject	Brightness	
	Dark	Bright
People	0.8	0.2
Landscape	0.3	0.7

P(Subject = People, Size = Small, Brightness = Bright) = 0.8 x 0.25 x 0.2 = 0.04

Figure 5.6 Computing the probability of a possible world by multiplying the appropriate entries in each CPD

example, for the possible world <Subject = People, Size = Small, Brightness = Bright>, the entry for Subject is 0.8, which is from the column labeled Subject. For Size, you look at the row corresponding to Subject = People and the column corresponding to Size = Small and get the entry 0.25. Finally, for Brightness, you again look at the row corresponding to Subject = People and this time take the column labeled Bright to get the entry 0.2. Finally, you multiply all these entries together to get the probability of the possible world, which is 0.8 × 0.25 × 0.2 = 0.04.

If you go through this process for every possible world, the probabilities will add up to 1, just as they're supposed to. This is always the case for a Bayesian network. So you've seen how a Bayesian network defines a valid probability distribution. Now that you understand exactly what a Bayesian network consists of and what it means, let's see how to use one to derive beliefs about some variables, given knowledge of other variables.

5.2.3 Reasoning with Bayesian networks

A Bayesian network encodes a lot of independencies that hold between variables. Recall that independence between two variables means that learning something about one variable doesn't tell you anything new about the other variable. From the preceding example, you can see that Number of Printed Pages and Printer Power Button On *aren't* independent. When you learn that no pages were printed, that reduces the probability that the power button is on.

Conditional independence is similar. Two variables are conditionally independent given a third variable if, *after the third variable is known*, learning something about the first variable doesn't tell you anything new about the second. A criterion called *d-separation* determines when two variables in a Bayesian network are conditionally independent of a third set of variables. The criterion is a little involved, so I won't provide a formal definition. Instead, I'll describe the basic principles and show you a few examples.

The basic idea is that reasoning *flows along a path* from one variable to another. In the example of figure 5.4, reasoning can flow from Size to Brightness along the path Size-Subject-Brightness. You saw a glimpse of this idea in section 5.1.3 on direct and indirect dependencies. In an indirect dependency, reasoning flows from one variable to another variable via other intermediary variables. In this example, Subject is the intermediary variable between Size and Brightness. In a Bayesian network, reasoning can flow along a path as long as the path isn't *blocked* at some variable.

In most cases, a path is blocked at a variable if the variable is observed. So if Subject is observed, the path Size-Subject-Brightness is blocked. This means that if you observe Size, it won't change your beliefs about Brightness if Subject is also observed. Another way of saying this is that Size is conditionally independent of Brightness, given Subject. In our model, the painter's choice of subject determines the size and the brightness, but after choosing the subject, the size and brightness are generated independently.

Converging arrows and induced dependencies

One other case may seem counterintuitive at first. In this case, a path is blocked at a variable if the variable is unobserved, and becomes unblocked if the variable is observed. I'll illustrate this situation by extending our example, as shown in figure 5.7. You have a new variable called Material, which could be oil or watercolor or something else. Naturally, Material is a cause of Brightness (perhaps oil paintings are brighter than watercolors), so the network has a directed edge from Material to Brightness. Here, you have two parents of the same child. This is called a *converging arrows* pattern, because edges from Subject and Material converge at Brightness.

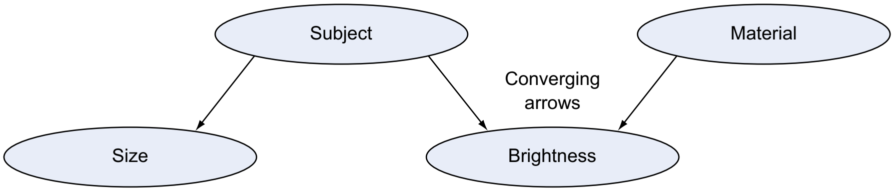


Figure 5.7 Extended painting example, including converging arrows between two parents of the same child

Now, let's think about reasoning between Subject and Material. According to the model, Subject and Material are generated independently. So this is true:

(1) Subject and Material are independent when nothing is observed.

But what happens when you observe that the painting is bright? According to our model, landscapes tend to be brighter than people paintings. After observing that the painting is bright, you'll infer that the painting is more likely to be a landscape. Let's say you then observe that the painting is an oil painting, which paintings also tend to be bright. This observation provides an alternative explanation of the brightness of our painting. Therefore, the probability that the painting is a landscape is discounted somewhat compared to what it was after you observed that the painting is bright but before you observed that it's an oil painting. You can see that reasoning is flowing from Material to Subject along the path Material-Brightness-Subject. So you get the following statement:

(2) Subject and Material aren't conditionally independent, given Brightness.

What you have here is the opposite pattern from the usual. You have a path that's blocked when the intermediary variable is unobserved, and becomes unblocked when the variable is observed. This kind of situation is called an *induced dependency*—a dependency between two variables that's induced by observing a third variable. Any converging arrows pattern in a Bayesian network can lead to an induced dependency.

A path between two variables can include both ordinary patterns and converging arrows. Reasoning flows along the path only if it's not blocked at any node on the

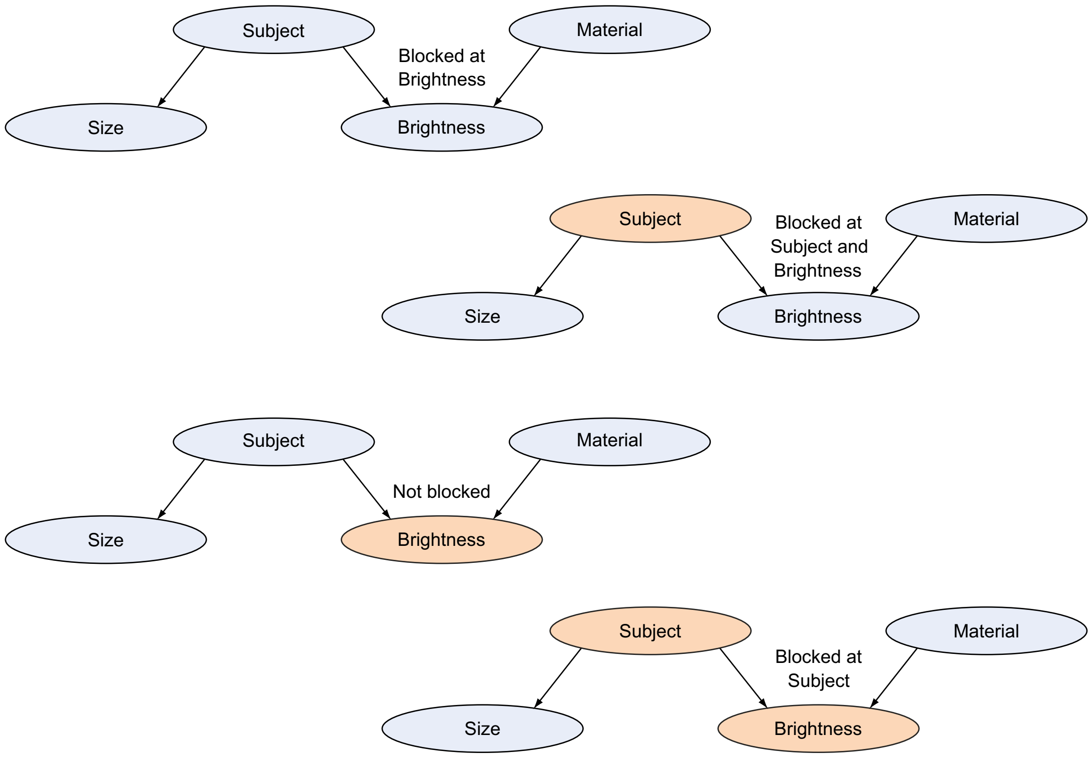


Figure 5.8 Examples of blocked and unblocked paths that combine an ordinary pattern with a converging arrows pattern. Each figure shows the path from Size to Material, with Subject and Brightness either unobserved or observed.

path. Figure 5.8 shows four examples for the path Size-Subject-Brightness-Material. In the top-left example, neither Subject nor Brightness is observed, and the path is blocked at Brightness because it has converging arrows. In the next example, on the right, Subject is now observed, so the path is blocked at both Subject and Brightness. In the next example, on the left, Subject is unobserved and Brightness is observed, which is precisely the condition required for the path not to be blocked at either Subject or Brightness. Finally, in the bottom right, Subject is observed in addition to Brightness, so the path is blocked there.

5.3 Exploring a Bayesian network example

Now that you've learned the basic concepts of Bayesian networks, let's look at an example of troubleshooting a printer problem. I'll first show you how to design the network and then show you all of the ways of reasoning with the network. I'll save a discussion of learning the parameters of the network for chapter 12, where you'll explore a useful design pattern for parameter learning.

5.3.1 Designing a computer system diagnosis model

Imagine that you're designing a help desk application for technical support. You want to help the tech support person identify the causes of faults as quickly as possible. You can use a probabilistic reasoning system for this application—given evidence, consisting of reports from the user and diagnostic tests, you want to determine the internal state of the system. For this application, it's natural to use a Bayesian network to represent the probabilistic model.

When you design a Bayesian network, you typically go through three steps: choosing the variables and their corresponding domains, specifying the network structure, and encoding the CPDs. You'll see how to do this in Figaro.

In practice, you don't usually go through all steps in a linear fashion, choosing all of the variables, building the entire network structure, and then writing down the CPDs. Usually, you'll build up the network a bit at a time, refining it as you go. You'll take that approach here. First, you'll build a network for a general print fault model and then drill down into a more detailed model of the printer.

GENERAL PRINT FAULT MODEL: VARIABLES

You want to model possible reports from the user, the faults that might be involved, and system factors that might lead to those faults. You'll introduce variables for all these things.

You start with a Print Result Summary. This represents the user's overall experience of the print result at a high level of abstraction. When the user first calls the help desk, that user may provide only a high-level summary like this. Three results are possible: (1) printing happens perfectly (you'll label this excellent); (2) printing happens, but isn't quite right (poor); (3) no printing happens at all (none).

Next, you consider various concrete aspects of the print result. These are Number of Printed Pages, which could be zero, some of the pages, or all of the pages; Prints Quickly, a Boolean variable indicating whether printing happens in a reasonable time; and Good Print Quality, another Boolean variable. One reason for modeling each of these aspects individually is that they differentiate between different faults. For example, if no pages print, it might be because the network is down. But if some but not all of the pages print, the problem is less likely to be the network and more likely to be user error.

So you consider all elements of the system that might influence the printing result. These include the Printer State, which could be good, poor, or out; the Software State, which could be correct, glitchy, or crashed; the Network State, which could be up, intermittent, or down; and User Command Correct, which is a Boolean variable.

GENERAL PRINT FAULT MODEL: NETWORK STRUCTURE

Considering the variables you've defined, there are three groups: the abstract Print Result Summary, the various concrete aspects of the print result, and the system states that influence the print result. Accordingly, it makes sense to design the network in three layers. What should be the order of the layers?

Cause-effect relationships exist between the system state variables and concrete print result variables. For example, the Network State being down is a cause of a failure to Print Quickly. In addition, concrete-abstract relationships exist between the individual print result variables and the overall Print Result Summary. In section 5.1.1, I said that these relationships could go in either direction. In our application, you're modeling the user's experience and reporting of the print result, so according to the rule of thumb I introduced there, the right direction is to go from the concrete print result variables to the abstract summary. So the order of layers in our network is (1) system state variables, (2) concrete print result variables, (3) Print Result Summary.

The network structure is shown in figure 5.9. You can see the three layers, but there's not always an edge from every variable in one layer to variables in the next layer. This is because some of the print result variables depend on the state of only some of the system components. For example, whether the print quality is good depends on the state of the printer but not on the network. Likewise, the speed of printing according to our model depends only on the network and the software. Whether these statements are correct is up for debate; the main point is that in any application, arguments can be made to remove some of the edges. The benefit of removing edges is smaller CPDs.

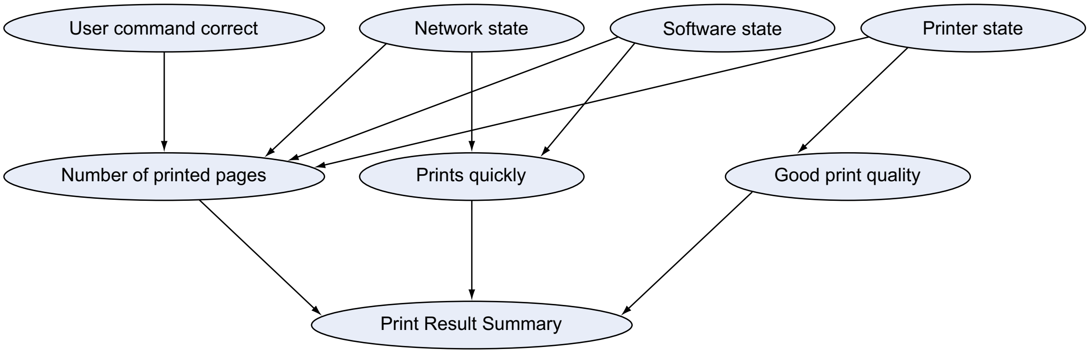


Figure 5.9 Network structure for the general print fault part of our computer system diagnosis model

General print fault model: CPDs

I'll show you the CPD design through Figaro code, making sure to explain the most interesting items. Section 5.1.1 showed you various ways CPDs can be defined in Figaro. You'll use a variety of them here.

Listing 5.1 Implementing the general print fault model in Figaro

This comes from the detailed printer model coming next.

```
val printerState = …

val softwareState =
  Select(0.8 -> 'correct, 0.15 -> 'glitchy, 0.05 -> 'crashed)
val networkState =
  Select(0.7 -> 'up, 0.2 -> 'intermittent, 0.1 -> 'down)
val userCommandCorrect =
  Flip(0.65)
```

For root variables, we have atomic CPDs like Select or Flip.

```
val numPrintedPages =
  RichCPD(userCommandCorrect, networkState,
          softwareState, printerState,
          (*, *, *, OneOf('out)) -> Constant('zero),
          (*, *, OneOf('crashed), *) -> Constant('zero),
          (*, OneOf('down), *, *) -> Constant('zero),
          (OneOf(false), *, *, *) ->
            Select(0.3 -> 'zero, 0.6 -> 'some, 0.1 -> 'all),
          (OneOf(true), *, *, *) ->
            Select(0.01 -> 'zero, 0.01 -> 'some, 0.98 -> 'all))
```

numPrintedPages uses RichCPD to represent the dependency on the user command and the network, software, and printer states.

```
val printsQuickly =
  Chain(networkState, softwareState,
        (network: Symbol, software: Symbol) =>
           if (network == 'down || software == 'crashed)
                  Constant(false)
           else if (network == 'intermittent || software == 'glitchy)
                  Flip(0.5)
           else Flip(0.9))
```

printsQuickly uses Chain to represent the dependency on network and software states.

```
val goodPrintQuality =
  CPD(printerState,
      'good -> Flip(0.95),
      'poor -> Flip(0.3),
      'out -> Constant(false))
```

goodPrintQuality uses a simpleCPD to represent the dependency on the printer state.

```
val printResultSummary =
  Apply(numPrintedPages, printsQuickly, goodPrintQuality,
        (pages: Symbol, quickly: Boolean, quality: Boolean) =>
          if (pages == 'zero) 'none
          else if (pages == 'some || !quickly || !quality) 'poor
          else 'excellent)
```

Because it's fully determined by its parents, printResultSummary uses Apply.

This code uses several techniques to represent the probabilistic dependence of a child on its parents:

- numPrintedPages uses RichCPD, which implements the following logic: If the printer state is out, or the network state is down, or the software is crashed, zero pages will be printed, regardless of the state of other parents. Otherwise, if the user issues the wrong command, it's unlikely all of the pages will be printed, but if the user issues the right command, it's highly likely that all pages will be printed.

- printsQuickly uses Chain to implement the following logic: If the network is down or software is crashed, it definitely won't print quickly. Otherwise, if either the network is intermittent or the software is glitchy, printing quickly is a toss-up. If both the network and software are in good states, it will usually print quickly (but not guaranteed).
- goodPrintQuality uses a simple CPD. If the printer is out, there definitely won't be good-quality printing. If the printer is in a poor state, there probably won't be good-quality printing. Even if the printer is in a good state, good-quality printing isn't guaranteed (because that's the way printers are).
- printSummary is a deterministic variable: It's fully determined by its parents without any uncertainty. You can use Apply instead of Chain for a deterministic variable.

Detailed printer model

This section goes through the detailed printer model more quickly, because many of the principles are the same. The network structure is shown in figure 5.10. Three factors influence the printer state: Paper Flow, Toner Level, and Printer Power Button On. The model adds a new kind of variable, which you haven't seen, an indicator or measurement. Paper Jam Indicator On is a measurement of the Paper Flow, and Toner Low Indicator On is a measurement of the Toner Level. As discussed in section 5.1.1, the relationship between a true value and its measurement is a kind of cause-effect relationship, so you have an edge from Paper Flow to Paper Jam Indicator On and from Toner Level to Toner Low Indicator On.

Here's the code defining the CPDs, which is mostly straightforward.

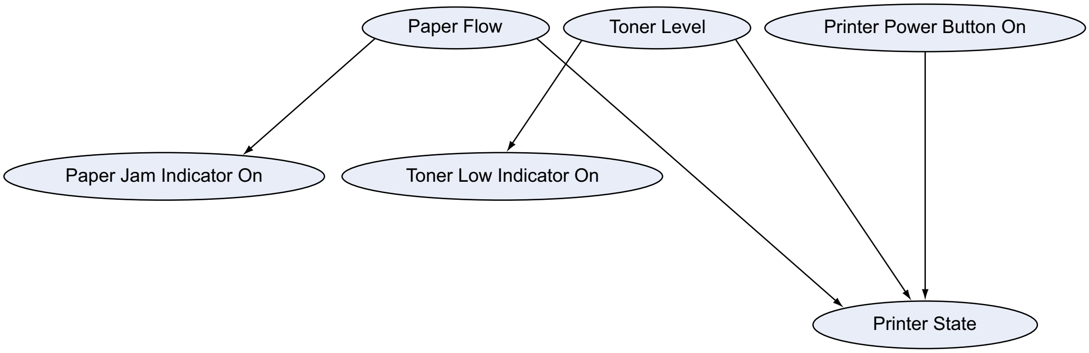


Figure 5.10 Network structure for detailed printer model

Listing 5.2 Detailed printer model in Figaro

```
val printerPowerButtonOn = Flip(0.95)
val tonerLevel = Select(0.7 -> 'high, 0.2 -> 'low, 0.1 -> 'out)
val tonerLowIndicatorOn =
  If(printerPowerButtonOn,
     CPD(paperFlow,
         'high -> Flip(0.2),
         'low -> Flip(0.6),
         'out -> Flip(0.99)),
     Constant(false))
val paperFlow = Select(0.6 -> 'smooth, 0.2 -> 'uneven, 0.2 -> 'jammed)

val paperJamIndicatorOn =
  If(printerPowerButtonOn,
     CPD(tonerLevel,
         'high -> Flip(0.1),
         'low -> Flip(0.3),
         'out -> Flip(0.99)),
     Constant(false))

val printerState =
  Apply(printerPowerButtonOn, tonerLevel, paperFlow,
        (power: Boolean, toner: Symbol, paper: Symbol) => {
          if (power) {
            if (toner == 'high && paper == 'smooth) 'good
            else if (toner == 'out || paper == 'out) 'out
            else 'poor
          } else 'out
        })
```

Recall that a single-quote character ' indicates the Scala Symbol type.

A CPD nested inside an If. If printer power button is on, toner low indicator depends on toner level. If power button is off, toner indicator will be off because there's no power, regardless of the toner level.

Printer state is a deterministic summary of the factors that make it up.

To summarize the example, the full Bayesian network structure is shown in figure 5.11. You can also see the entire program in the book's code under chap05/PrinterProblem.scala.

5.3.2 Reasoning with the computer system diagnosis model

This section shows how to reason with the computer system diagnosis model you just built. The Figaro mechanics of reasoning are simple, but the reasoning patterns you get out of it are interesting and illustrate the concepts introduced in section 5.2.3. All of the reasoning patterns in this section are divided into separate steps in the code for the chapter. Comment out the steps you don't want to execute, to highlight each step as it's presented.

QUERYING A PRIOR PROBABILITY

First, let's query the probability that the printer power button is on, without any evidence. This is the prior probability. You can use the following code:

```
val answerWithNoEvidence =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Prior probability the printer power button is on = " +
  answerWithNoEvidence)
```

Compute P(Printer Power Button On = true)

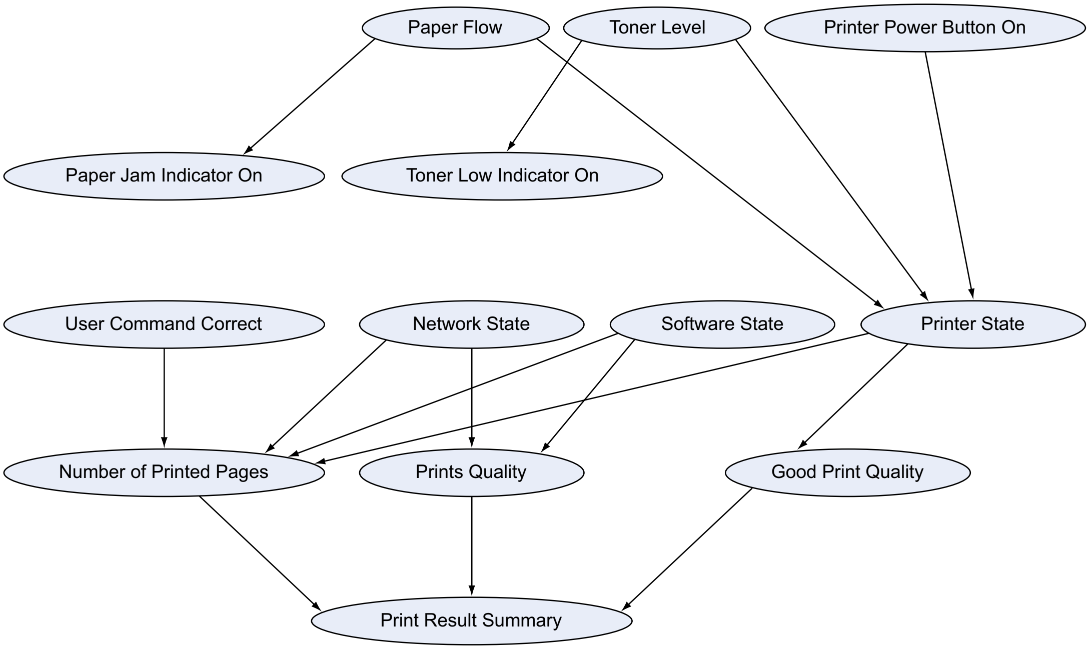


Figure 5.11 The full network for the computer system diagnosis example

This prints the following result:

```
Prior probability the printer power button is on = 0.95
```

If you look back at the model, you'll see that `printerPowerButtonOn` is defined using the following line:

```
val printerPowerButtonOn = Flip(0.95)
```

You can see that the answer to the query is exactly what you'd get if you ignored the entire model except for this definition. This is an example of a general rule: the network downstream from a variable is relevant to the variable only if it has evidence. In particular, for prior probabilities, there's no evidence, so you don't care about the downstream network.

QUERYING WITH EVIDENCE

What happens if you introduce evidence? Let's query the model for the probability that the printer power button is on, given that the print result is poor, implying there was some result, but it wasn't what the user wanted. You can use the following code:

```
printResultSummary.observe('poor)
val answerIfPrintResultPoor =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Probability the printer power button is on given a poor "
  + "result = " + answerIfPrintResultPoor)
```

Compute P(Printer Power Button On = true | Print Result = poor)

This prints the following result:

```
Probability the printer power button is on given a poor result = 1.0
```

This might be surprising! The probability is higher than if you didn't have any evidence about printing. If you think about the model, you can understand why. A poor printing result can happen only if at least some of the pages are printed, which won't be the case if the power is off. Therefore, the probability the power is on is 1.

Now let's query with the evidence that nothing is printed:

```
printResultSummary.observe('none')
val answerIfPrintResultNone =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Probability the printer power button is on given empty "
  + "result = " + answerIfPrintResultNone)
```

Compute P(Printer Power Button On = true | Printer Result = none)

Now the result is as follows:

```
Probability the printer power button is on given empty result =
0.8573402523786461
```

This is as you expect. The power button being off is a good explanation for an empty print result, so its probability increases based on the evidence.

INDEPENDENCE AND BLOCKING

Section 5.2.3 introduced the concept of a blocked path and the relationship of this concept to conditional independence. This concept can be illustrated with three variables: Print Result Summary, Printer Power Button On, and Printer State. In figure 5.11, you see that the path from Printer Power Button On to Print Result Summary goes through Printer State. Because this isn't a converging arrows pattern, the path is blocked if Printer State is observed. Indeed, this is the case, as you'll see now:

```
printResultSummary.unobserve()
printerState.observe('out)
val answerIfPrinterStateOut =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Probability the printer power button is on given " +
        "out printer state = " + answerIfPrinterStateOut)
```

Compute P(Printer Power Button On = true | Printer State = out)

```
printResultSummary.observe('none)
val answerIfPrinterStateOutAndResultNone =
  VariableElimination.probability(printerPowerButtonOn, true)
println("Probability the printer power button is on given " +
        "out printer state and empty result = " +
        answerIfPrinterStateOutAndResultNone)
```

Compute P(Printer Power Button On = true | Printer State = out, Print Result Summary = none)

This prints the following:

```
Probability the printer power button is on given out printer state =
0.6551724137931032
Probability the printer power button is on given out printer state and empty
result = 0.6551724137931033
```

You can see that learning that the print result is empty doesn't change the probability that the printer power button is on, after you already know that the printer state is out. This is because Print Result Summary is conditionally independent from Printer Power Button On, given Printer State.

Reasoning between different effects of the same cause

All of the reasoning paths you've seen so far have gone straight up the network. You can also combine both directions when reasoning. It's easy to see this when you consider what a measurement tells you about the value it's measuring, and what that can inform in turn. In our example, Toner Low Indicator On is a child of Toner Level, and Toner Level is a parent of Printer State. If the toner level is low, it's less likely that the printer state is good. Meanwhile, the toner low indicator being on is a sign that the toner level is low. It stands to reason that if you observe that the toner low indicator is on, it should reduce the probability that the printer state is good. You can see that this is the case with the following code:

```
printResultSummary.unobserve()
printerState.unobserve()
val printerStateGoodPrior =
  VariableElimination.probability(printerState, 'good)
println("Prior probability the printer state is good = "
        + printerStateGoodPrior)
```

Compute P(Printer State = good)

```
tonerLowIndicatorOn.observe(true)
val printerStateGoodGivenTonerLowIndicatorOn =
  VariableElimination.probability(printerState, 'good)
println("Probability printer state is good given low toner "
        + "indicator = " + printerStateGoodGivenTonerLowIndicatorOn)
```

Compute P(Printer State = good | Toner Low Indicator On = true)

This prints the following:

```
Prior probability the printer state is good = 0.3989999999999997
Probability the printer state is good given low toner indicator =
0.23398328690807796
```

You can see that the probability that the printer state is good decreases when you observe the low toner indicator, as expected.

Reasoning between different causes of the same effect: induced dependencies

As discussed in section 5.2.3, reasoning between different causes of the same effect is different from other kinds of reasoning, because it involves converging arrows, which lead to an induced dependency. For an example, let's use Software State and Network State, which are both parents of Prints Quickly. First, you'll get the prior probability that the software state is correct:

```
tonerLowIndicatorOn.unobserve()
val softwareStateCorrectPrior =
  VariableElimination.probability(softwareState, 'correct)
println("Prior probability the software state is correct = " +
        softwareStateCorrectPrior)
```

Compute P(Software State = correct)

This prints the following:

```
Prior probability the software state is correct = 0.8
```

Next, you'll observe that the network is up and query the software state again:

```
networkState.observe('up')
val softwareStateCorrectGivenNetworkUp =
  VariableElimination.probability(softwareState, 'correct)
println("Probability software state is correct given network up = " +
        softwareStateCorrectGivenNetworkUp)
```

Compute P(Software State = correct | Network State = up)

This prints the following:

```
Probability software state is correct given network up = 0.8
```

The probability hasn't changed, even though there's a clear path from Network State to Software State via Prints Quickly! This shows that in general, two causes of the same effect are independent. This is intuitively correct: the network being up has no bearing on whether the software state is correct.

Now, if you know that the printer isn't printing quickly, that's different. If you see the printer printing slowly, our model provides two possible explanations: a network problem or a software problem. If you observe that the network is up, it must be a software problem. You can see this with the following code:

Compute P(Software State = correct | Prints Quickly = false)

```
networkState.unobserve()
printsQuickly.observe(false)
val softwareStateCorrectGivenPrintsSlowly =
  VariableElimination.probability(softwareState, 'correct)
println("Probability software state is correct given prints "
        + "slowly = " + softwareStateCorrectGivenPrintsSlowly)
```

Compute P(Software State = correct | Prints Quickly = false, Network State = up)

```
networkState.observe('up)
val softwareStateCorrectGivenPrintsSlowlyAndNetworkUp =
  VariableElimination.probability(softwareState, 'correct)
println("Probability software state is correct given prints "
        + "slowly and network up = "
        + softwareStateCorrectGivenPrintsSlowlyAndNetworkUp)
```

Running this code prints the following:

```
Probability software state is correct given prints slowly =
0.6197991391678623
Probability software state is correct given prints slowly and network up =
 0.39024390243902435
```

Learning that the network is up significantly reduces the probability that the software is correct. So Software State and Network State are independent, but they aren't conditionally independent given Prints Quickly. This is an example of an induced dependency.

To summarize:

- When reasoning from an effect X to its indirect cause Y, X isn't independent of Y, but becomes conditionally independent, given Z, if Z blocks the path from X to Y.
- The same holds when reasoning from a cause to an indirect effect or between two effects of the same cause.
- For two causes X and Y of the same effect Z, the opposite is true. X and Y are independent, but not conditionally independent, given Z, as a result of the induced dependency.

That wraps up the computer system diagnosis example. You've seen a fairly substantial network with some interesting reasoning patterns. In the next section, you'll move beyond traditional Bayesian networks.

5.4 Using probabilistic programming to extend Bayesian networks: predicting product success

This section shows how to extend the basic Bayesian network model to predict the success of a marketing campaign and product placement. The purpose of this example is twofold: first, to show the power of using programming languages to extend Bayesian networks, and second, to illustrate how Bayesian networks can be used not only to infer causes of observed events but also to predict future events. As with the computer system diagnosis example, I'll first show how to design the model and express it in Figaro and then how to reason with the model.

5.4.1 Designing a product success prediction model

Imagine that you have a new product, and you want to make the product as successful as possible. You could achieve this in various ways. You could invest in the product's packaging and other things that might make it appealing to customers. You could try to price it in such a way that people would be more likely to buy it. Or you could promote the product by giving away free versions in the hope that people will share them with their friends. Before you choose a strategy, you want to know the relative importance of each factor.

This section describes the framework of a model you could use for this application. I say *framework*, because this is just a skeleton of the model you would build if you were doing this for real, but it's enough to make the point. The model presented here is a simple Bayesian network with just four nodes, but the types of the nodes and the CPDs are rich and interesting.

The model has four variables, as shown in figure 5.12:

- Target Social Network is a variable whose type is a social network. This is one way a programming language lets you go beyond ordinary Bayesian networks, where variables are just Booleans, enumerations, integers, or real numbers.

The CPD of this variable randomly generates the network based on the popularity of the target. As for the popularity itself, because it's a control variable, you model it as a known constant rather than a variable. But if you wanted to, you could introduce uncertainty about the popularity and make it a variable itself.

- Target Likes is a Boolean variable indicating whether the target likes the product, which is a function of the product quality. Again, the product quality is a known constant, but you could have made it a variable that depends on investment.
- Number Friends Like is an integer variable, but its CPD traverses the Target Social Network to determine the number of friends who are shown the product and who like it. This is a much richer CPD than has traditionally been available in Bayesian networks.
- Number Buy is an integer variable. Its model is simply defined by considering that each person who likes the product will buy it with a probability that depends on its affordability, a control variable whose value is a known constant. Therefore, the number of people who buy the product is a binomial.

Here's the Figaro code for this model. I'll describe how the code works on a high level and then present details of the model.

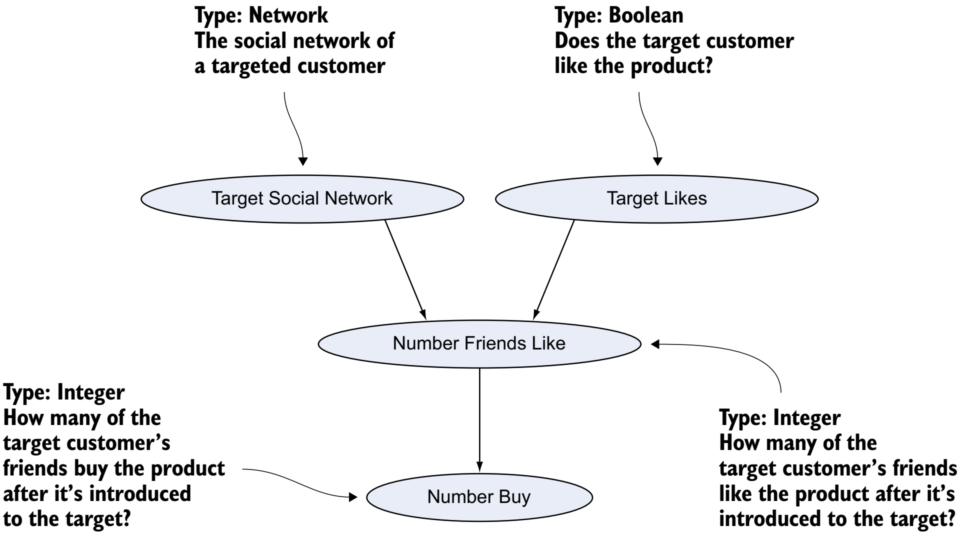

Type: Network
The social network of a targeted customer
Type: Boolean
Does the target customer like the product?
Target Social Network
Target Likes
Number Friends Like
Type: Integer
How many of the target customer's friends buy the product after it's introduced to the target?
Number Buy
Type: Integer
How many of the target customer's friends like the product after it's introduced to the target?

Figure 5.12 Product success prediction network

Listing 5.3 Product success prediction model in Figaro

```
class Network(popularity: Double) {
  val numNodes = Poisson(popularity)
}

class Model(targetPopularity: Double, productQuality: Double,
            affordability: Double) {

  def generateLikes(numFriends: Int,
                    productQuality: Double): Element[Int] = {

    def helper(friendsVisited: Int, totalLikes: Int,
               unprocessedLikes: Int): Element[Int] = {
      if (unprocessedLikes == 0) Constant(totalLikes)
      else {
        val unvisitedFraction =
          1.0 - (friendsVisited.toDouble - 1) / (numFriends - 1)
        val newlyVisited = Binomial(2, unvisitedFraction)
        val newlyLikes =
          Binomial(newlyVisited, Constant(productQuality))
        Chain(newlyVisited, newlyLikes,
              (visited: Int, likes: Int) =>
                helper(friendsVisited + unvisited,
                       totalLikes + likes,
                       unprocessedLikes + likes - 1))
      }
    }

    helper(1, 1, 1)
  }

  val targetSocialNetwork = new Network(targetPopularity)

  val targetLikes = Flip(productQuality)

  val numberFriendsLike =
    Chain(targetLikes, targetSocialNetwork.numNodes,
          (l: Boolean, n: Int) =>
            if (l) generateLikes(n, productQuality)
            else Constant(0))

  val numberBuy =
    Binomial(numberFriendsLike, Constant(affordability))
}
```

Define a Network class with a single attribute defined by a Poisson element (see text)

Create a Model class that takes the known control parameters as arguments

Define a recursive process for generating the number of people who like the product (see text)

The target social network is defined to be a random network, based on the target's popularity.

Whether the target likes the product is a Boolean element based on the product quality.

If the target likes the product, calculate the number of friends using generateLikes. If she doesn't, she doesn't tell friends about it, so the number is 0.

The number of friends who buy the product is a binomial (see text).

Three details of the code need additional explanation: the `Poisson` element used in the `Network` class, the `generateLikes` process, and the definition of `numberBuy`. Let's first talk about the `Poisson` element and the `numberBuy` logic and then get to the `generateLikes` process, which is the most interesting part of the model.

- A *Poisson element* is an integer element that uses what is known as the *Poisson distribution*. The Poisson distribution is typically used to model the number of occurrences of an event in a period of time, such as the number of network failures in a month or the number of corner kicks in a game of soccer. With a little creativity, the Poisson distribution can be used to model any situation where you want to know the number of things in a region. Here, you use it to model the number of people in someone's social network, which is different from the usual usage but still a reasonable choice.

The Poisson element takes as an argument the average number of occurrences you'd expect in that period of time, but allows for the number to be more or less than the average. In this model, the argument is the popularity of the target; the popularity should be an estimate of the average number of people you expect to be in the target's social network.

- Here's the logic for the number of people who buy the product. Each person who likes the product will buy it with a probability equal to the value of the affordability parameter. So the total number of people who buy is given by a binomial, in which the number of trials is the number of friends who like the product, and the probability of buying depends on the affordability of the product. Because the number of people who like the product is itself an element, you need to use the compound binomial that takes elements as its arguments. The compound binomial element requires that the probability of success of a trial also be an element, which is why the affordability is wrapped in a `Constant`. A `Constant` element takes an ordinary Scala value and produces the Figaro element that always has that value.
- The purpose of the `generateLikes` function is to determine the number of people who like the product after giving it to a target whose social network contains the given number of people. This function assumes that the target herself likes the product; otherwise, the function wouldn't be called at all. The function simulates a random process of people promoting the product to their friends if they like the product. The `generateLikes` function takes two arguments: (1) the number of people in the target's social network, which is an `Integer`, and (2) the quality of the product, which is a `Double` between 0 and 1.

  The precise logic of the `generateLikes` function isn't critical, because the main point is that you can use an interesting recursive function like this as a CPD. But I'll explain the logic, so you can see an example. Most of the work of `generateLikes` is done by a helper function. This function keeps track of three values:

  – `friendsVisited` holds the number of people in the target's social network who have already been informed about the product. This starts at 1, because initially the target has been informed about the product.
  – `totalLikes` represents the number of people, out of those who have been visited so far, who like the product. This also starts at 1, because you assume that the target likes the product for `generateLikes` to be called.
  – `unprocessedLikes` represents the number of people who like the product for whom you've not yet simulated promoting the product to their friends.

I'll explain the logic of the helper function through the following code.

Listing 5.4 Helper function for traversing the social network

```
def helper(friendsVisited: Int, totalLikes: Int,
      unprocessedLikes: Int): Element[Int] = {
 if (unprocessedLikes == 0) Constant(totalLikes)
 else {
  val unvisitedFraction =
      1.0 - (friendsVisited.toDouble - 1) / (numFriends - 1)
     val newlyVisited = Binomial(2, unvisitedFraction)
     val newlyLikes =
      Binomial(newlyVisited, Constant(productQuality))
     Chain(newlyVisited, newlyLikes,
        (visited: Int, likes: Int) =>
         helper(friendsVisited + visited,
            totalLikes + likes,
            unprocessedLikes + likes - 1))
    }
   }
```

The helper function takes ordinary Scala values as arguments but returns an element. It can be used inside a chain.

Termination criterion: if there are no people left to process, the number of people who like the product is equal to the number found so far.

Computes the probability that a random person in the social network (excluding the person being processed) won't yet have been visited

The person being processed promotes the product to two friends. The probability that each friend hasn't yet been visited is given by unvisitedFraction.

The probability that a given person likes the product is given by productQuality, which is wrapped in a Constant to conform to the interface of the compound binomial.

It's typical to write a recursive process using Chain.

The new number of visited people is equal to the old number plus those who are newly visited.

Everyone who newly likes the products is an unprocessed person, but you subtract one for the person you just processed.

The new number of people who like the product is equal to the old number plus those who newly like it.

This example uses more programming and Scala skills, but the essential modeling techniques are similar to Bayesian networks. The main point is to imagine a process by which a possible world is generated. In this example, you saw one relatively simple process of propagating a product through a social network; a richer process could easily be encoded.

5.4.2 Reasoning with the product success prediction model

Because you've designed the model for predicting success, the typical usage would be to set values for the control constants and predict the number of people who buy the product. Because the number of people is an Integer variable that could have a wide range, you're not really interested in predicting the probability of a particular value. Rather, you want to know the average value you can expect. This is known as the *expectation* of the value.

In probability theory, expectation is a general concept. The expectation takes a function defined on a variable and returns the average value of that function. An example is shown in figure 5.13. You start with a probability distribution over values of any type; in this example, the values are Integers. You then apply a function to each value to produce a Double value. In this example, the function converts each Integer to a Double representation of the Integer. Next, you take the weighted average of these Double values, where each Double value is weighted by its probability. This means that you multiply each Double value by its probability and add the results.

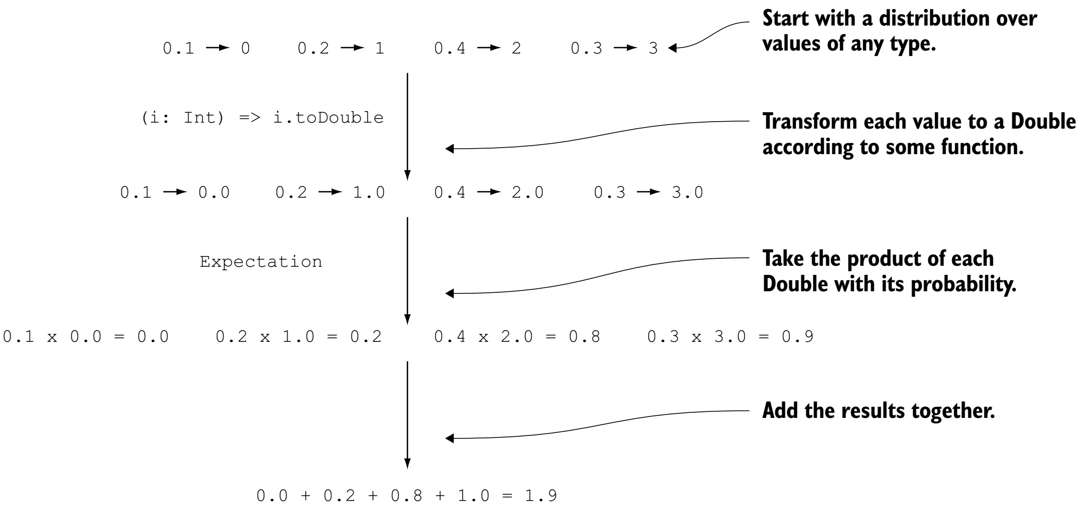

0.1 → 0 0.2 → 1 0.4 → 2 0.3 → 3
Start with a distribution over values of any type.
(i: Int) => i.toDouble
Transform each value to a Double according to some function.
0.1 → 0.0 0.2 → 1.0 0.4 → 2.0 0.3 → 3.0
Expectation
Take the product of each Double with its probability.
0.1 x 0.0 = 0.0 0.2 x 1.0 = 0.2 0.4 x 2.0 = 0.8 0.3 x 3.0 = 0.9
Add the results together.
0.0 + 0.2 + 0.8 + 1.0 = 1.9

Figure 5.13 Computing the expectation of a distribution over an Integer-valued element. The values of the element are first converted to Doubles. Then you take the average of these Doubles, where each Double is weighted by its probability.

In our example of predicting product success, you want to compute the expectation of the number of people who buy the product, which is an Integer variable. You can use the following line in Figaro to do this:

```
algorithm.expectation(model.numberBuy, (i: Int) => i.toDouble)
```

Here, `algorithm` is a handle on the inference algorithm you're using. This application uses an importance sampling algorithm, which is a particularly good algorithm for predicting the outcome of a complex generative process. Because you need a handle on the algorithm, you need to use slightly more-complex code than before to run inference, as is explained in the following code snippet. The entire process of taking in the control constants and computing the expected number of people who buy the product is accomplished by a function called `predict`:

```
def predict(targetPopularity: Double, productQuality: Double,
            affordability: Double): Double = {

  val model =
    new Model(targetPopularity, productQuality, affordability)

  val algorithm = Importance(1000, model.numberBuy)

  algorithm.start()

  val result =
    algorithm.expectation(model.numberBuy, (i: Int) => i.toDouble)

  algorithm.kill()

  result
}
```

Create a new instance of our product prediction model, using the given control constants

Create an instance of the importance sampling algorithm. 1000 is the number of samples to use, while model.numberBuy indicates what we want to predict.

Run the algorithm

Compute the expectation of the number of people who buy the product

Clean up and free resources taken by the algorithm

If you're trying to understand the effect of various controls on the number of people who buy the product, you'll want to run this `predict` function many times with different inputs. You might produce a result like this:

```
Popularity  Product quality Affordability Predicted number of buyers
100         0.5             0.5           2.0169999999999986
100         0.5             0.9           3.7759999999999962
100         0.9             0.5           29.21499999999997
100         0.9             0.9           53.13799999999996
10          0.5             0.5           0.7869999999999979
10          0.5             0.9           1.4769999999999976
10          0.9             0.5           3.3419999999999885
10          0.9             0.9           6.066999999999985
```

You can conclude a couple of things from this table. The number of buyers appears to be roughly proportional to the affordability of the product. But there's a disproportionate dependence on the product quality: for every case of popularity and affordability, the number of buyers when the product quality is 0.9 is at least several times as high as when the quality is 0.5. When the popularity is 100, it's about 15 times as high. There appears to be an interaction between the popularity and product quality, where the popularity places a limit on the number of people who will be reached when the quality is high. When the quality is low, the popularity doesn't matter as much.

In this example, you've seen Figaro used as a simulation language. Predicting what will happen in the future is typically what simulations do, and this is the use case described here. You could just as easily use the model for reasoning backward. For example, after you hand out the product to some people and observe how many other people they influenced to buy the product, you could estimate the quality of the product. This could be a valuable alternative use of the model.

5.5 Using Markov networks

The preceding sections have been concerned with Bayesian networks, which encode directed dependencies. It's time to turn your attention to undirected dependencies. The counterpart to Bayesian networks for undirected dependencies is Markov networks. I'll explain the principles behind Markov networks using a typical image-recovery application. I'll then show you how to represent and reason with the image-recovery model in Figaro.

5.5.1 Markov networks defined

A Markov network is a representation of a probabilistic model consisting of three things:

- *A set of variables*—Each variable has a domain, which is the set of possible values of the variable.
- *An undirected graph in which the nodes are variables*—The edges between nodes are undirected, meaning they have no arrow from one variable to the other. This graph is allowed to have cycles.
- *A set of potentials*—These potentials provide the numerical parameters of the model. I'll explain what potentials are in detail in a moment.

Figure 5.14 shows a Markov network for an image-recovery application. There's a variable for every pixel in the image. This figure shows a 4×4 array of pixels, but it's easy

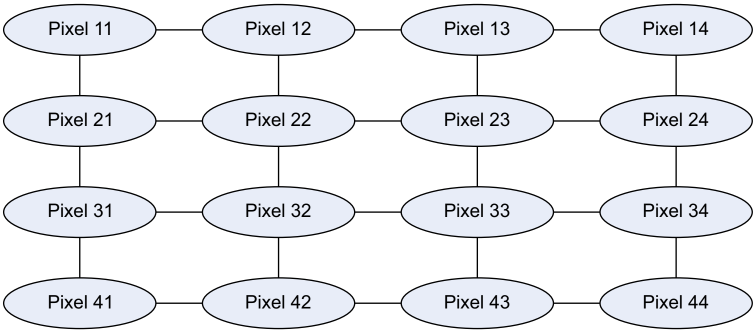


Figure 5.14 A Markov network for a pixel image

to see how it can be generalized to any size image. In principle, the value of a pixel could be any color, but for the sake of example, let's say it's a Boolean representing whether the pixel is bright or dark. There's an edge between any pair of pixels that are adjacent either horizontally or vertically. Intuitively, these edges encode the fact that, all else being equal, two adjacent pixels are more likely to have the same value than different values.

This "all else being equal" qualifier is important to understand the meaning of this model. If you ignore the edge between two pixels for a moment and consider the individual probability distribution over each of the pixels, it might be likely that they are, in fact, different. For example, based on everything else you know, you might believe that pixel 11 is bright with probability 90% and that pixel 12 is bright with probability 10%. In this case, it's highly likely that pixel 11 and pixel 12 are different. But the edge between pixel 11 and pixel 12 makes them *more likely to be the same than they would otherwise have been*. It adds a piece of knowledge that they're likely to be the same. This knowledge counterweighs the other knowledge that they're likely to be different, but it might not completely change the overall conclusion. The specific knowledge expressed by the edge between pixel 11 and pixel 12 is represented by the potential on that edge. Now let's see exactly how potentials are defined.

POTENTIALS

How are the numerical parameters of a Markov network defined? In a Bayesian network, each variable has a CPD. In a Markov network, it's not as simple. Variables don't own their numerical parameters. Instead, functions called *potentials* are defined on sets of variables. When there's a symmetric dependency, some joint states of the variables that are dependent on each other are more likely than others, all else being equal. The potential specifies a weight for each such joint state. Joint states with high weights are more likely than joint states with low weights, all else being equal. The relative probability of the two joint states is equal to the ratio between their weights, again, all else being equal.

Mathematically, a potential is simply a function from the values of variables to real numbers. Only positive real numbers or zero are allowed as the values of a potential. Table 5.1 shows an example of a unary potential over a single pixel for the image-recovery application, and table 5.2 shows a binary potential over two pixels.

Table 5.1 A unary potential over a single pixel. This potential encodes the fact that, all else being equal, a pixel is lit with probability 0.4.

Pixel 31	Potential value
F	0.6
T	0.4

Table 5.2 A binary potential over two adjacent pixels. This potential encodes the fact that, all else being equal, the two pixels are nine times as likely to have the same value as different values.

Pixel 31	Pixel 32	Potential value
F	F	0.9
F	T	0.1
T	F	0.1
T	T	0.9

How do potential functions interact with the graph structure? There are two rules:

- A potential function can mention only variables that are connected in the graph.
- If two variables are connected in the graph, they must be mentioned together by some potential function.

In our image-recovery example, every variable will have a copy of the unary potential in table 5.2, and every pair of adjacent pixels, either horizontally or vertically, will have a copy of the binary potential in table 5.2. You can see that the two rules are respected by this assignment of potentials.

HOW A MARKOV NETWORK DEFINES A PROBABILITY DISTRIBUTION

You've seen how a Markov network is defined. How does it define a probability distribution? How does it assign a probability to every possible world so that the probabilities of all possible worlds add up to 1? The answer isn't quite as simple as for Bayesian networks but also isn't too complicated.

Just as in a Bayesian network, a possible world in a Markov network consists of an assignment of values to all of the variables, making sure that the value of each variable is in its domain. What's the probability of such a possible world? Let's build it up piece by piece by using an example.

To keep things simple, let's consider a 2 × 2 array of pixels with the following assignment of values: pixel 11 = true, pixel 12 = true, pixel 21 = true, pixel 22 = false. You'll look at all potentials in the model and their potential values for this possible world. For the unary potentials, you have the values in table 5.3. Pixels that are true

Table 5.3 Potential values for unary potentials for example possible world

Variable	Potential value
Pixel 11	0.4
Pixel 12	0.4
Pixel 21	0.4
Pixel 22	0.6

have potential value 0.4, while the one pixel that's false has potential value 0.6. Table 5.4 shows the potential values from the four binary pixels. The cases where the two pixels have the same value have potential value 0.9, whereas the other two cases have potential value 0.1. This covers all potentials in the model.

Table 5.4 Potential values for binary potentials for example possible world

Variable 1	Variable 2	Potential value
Pixel 11	Pixel 12	0.9
Pixel 21	Pixel 22	0.1
Pixel 11	Pixel 21	0.9
Pixel 12	Pixel 22	0.1

Next, you multiply the potential values from all of the potentials. In our example, you get 0.4 × 0.4 × 0.4 × 0.6 × 0.9 × 0.1 × 0.9 × 0.1 = 0.00031104. Why do you multiply? Think about the "all else being equal" principle. If two worlds have the same probability except for one potential, then the probabilities of the worlds are proportional to their potential value according to that potential. This is exactly the effect you get when you multiply the probabilities by the value of this potential. Continuing this reasoning, you multiply the potential values of all of the potentials to get the "probability" of a possible world.

I put "probability" in quotes because it's not actually a probability. When you multiply the potential values in this way, you'll find that the "probabilities" don't sum to 1. This is easily fixed. To get the probability of any possible world, you normalize the "probabilities" computed by multiplying the potential values. You call these the *unnormalized probabilities*. The sum of these unnormalized probabilities is called the *normalizing factor* and is usually denoted by the letter Z. So you take the unnormalized probabilities and divide them by Z to get the probabilities. Don't worry if this process sounds cumbersome to you; Figaro takes care of all of it.

A surprising point comes out of this discussion. In a Bayesian network, you could compute the probability of a possible world by multiplying the relevant CPD entries. In a Markov network, you can't determine the probability of any possible world without considering all possible worlds. You need to compute the unnormalized probability of every possible world to calculate the normalizing factor. For this reason, some people find that representing Markov networks is harder than Bayesian networks, because it's harder to interpret the numbers as defining a probability. I say that if you keep in mind the "all else being equal" principle, you'll be able to define the parameters of a Markov network with confidence. You can leave computing the normalizing factor to Figaro. Of course, the parameters of both Bayesian networks and Markov networks can be learned from data. Use whichever structure seems more appropriate to you, based on the kinds of relationships in your application.

5.5.2 Representing and reasoning with Markov networks

There's one way Markov networks are definitely simpler than Bayesian networks: in the reasoning patterns. A Markov network has no notion of induced dependencies. You can reason from one variable to another variable along any path, as long as that path isn't blocked by a variable that has been observed. Two variables are dependent if there's a path between them, and they become conditionally independent given a set of variables if those variables block all paths between the two variables. That's all there is to it.

Also, because all edges in a Markov network are undirected, there's no notion of cause and effect or past and future. You don't usually think of tasks such as predicting future outcomes or inferring past causes of current observations. Instead, you simply infer the values of some variables, given other variables.

REPRESENTING THE IMAGE-RECOVERY MODEL IN FIGARO

In the image-recovery application, you'll assume that some of the pixels are observed and the rest are unobserved. You want to recover the unobserved pixels. You'll use the model described in the previous section, which specifies both the potential value for each pixel being on and the potential value for adjacent pixels having the same value. Here's the Figaro code for representing the model. Remember that in section 5.1.2, I said there are two methods for specifying symmetric relationships, a constraints method and a conditions method. This code uses the constraints method:

```
val pixels = Array.fill(10, 10)(Flip(0.4))

def setConstraint(i1: Int, j1: Int, i2: Int, j2: Int) {
  val pixel1 = pixels(i1)(j1)
  val pixel2 = pixels(i2)(j2)
  val pair = ^^(pixel1, pixel2)
  pair.addConstraint(bb => if (bb._1 == bb._2) 0.9; else 0.1)
}

for {
  i <- 0 until 10
  j <- 0 until 10
} {
  if (i <= 8) setConstraint(i, j, i+1, j)
  if (j <= 8) setConstraint(i, j, i, j+1)
}
```

Set the unary constraint on each variable

Set the binary constraint on a pair of variables given their coordinates

Apply the binary constraint to all pairs of adjacent variables

A few notes on this code are in order:

- In the definition of pixels, `Array.fill(10, 10)(Flip(0.4))` creates a 10 × 10 array and fills every element of the array with a different instance of `Flip(0.4)`. All of the different pixels are defined by different `Flip` elements, which is important because they can all have different values.
- You might be wondering why a `Flip` element is used at all for the unary potentials rather than a constraint. For a unary potential, defining it in the usual

Figaro way or using a constraint has the same effect. In this case, the Flip will come out true with probability 0.4 and false with probability 0.6. These probabilities will be multiplied into the unnormalized probability of a possible world, just as if they had been specified through a constraint.

In fact, every Figaro element has to be defined in the usual way, using some type of element constructor, even if you're using Figaro to encode a Markov network. If your element doesn't have a unary constraint, this ordinary Figaro constructor should be neutral and not favor any possible world over another. You could use Flip(0.5) or a Uniform element to achieve this.

- In the definition of setConstraint, ^^ is the Figaro pair constructor. So ^^(pixel1, pixel2) creates an element whose value is the pair of values of the elements pixel1 and pixel2.
- In the for comprehension, 0 until 10 is Scala for the integers 0 to 10, exclusive; in other words, the integers 0, 1, …, 9. If you wanted to say 0 to 10 inclusive, you would use 0 to 10.
- This for comprehension also shows an example of a nested loop. In other languages, this would have been written using one for loop inside another. In Scala, you can put both loops in the same for header.

Reasoning with the image-recovery model

You want to use the image-recovery model to infer the values of unobserved pixels, given the observed pixels. You need three things: a way to ingest and process the evidence, a way to compute the most likely states of pixels, and a way to view the results. Let's look at these one at a time.

Process evidence: If you have a 10 × 10 array of pixels, you might have data that's a 10 × 10 array of characters, where each character is 0 for off, 1 for on, or ? for unknown. You can process this data by using the following simple setEvidence function:

```
def setEvidence(data: String) = {
  for { n <- 0 until data.length } {
    val i = n / 10
    val j = n % 10
    data(n) match {
      case '0' => pixels(i)(j).observe(false)
      case '1' => pixels(i)(j).observe(true)
      case _ => ()
    }
  }
}
```

Uses Scala pattern matching, which in this case is a lot like a switch statement in other languages. The _ indicates a default case. So no observation will be made on a '?'.

Compute the most likely state of pixels: This example introduces a new kind of query you haven't considered before. In the past, you've wanted to estimate the posterior probabilities of elements. This time, you're interested in the most likely values of elements (the values with highest probability). But you're not interested in the most likely values of elements individually, without regard to the other elements; rather, you want the most likely joint assignment of values to all of the variables. You want to know what the most likely possible world is.

This query is known in Figaro as a *most probable explanation* (MPE) query, because you want to know the world that's the most probable explanation of the data. The algorithms for MPE queries are different from the probability computation algorithms you've seen so far, although they're related. In this example, you'll use a version of the belief propagation designed for computing the MPE. This algorithm is known as `MPE-BeliefPropagation`. Belief propagation is an iterative algorithm, and you can control the number of iterations with a parameter. In this case, you'll use 10 iterations. You can create an instance of the `MPEBeliefPropagation` algorithm and tell it to run in this way:

```
val algorithm = MPEBeliefPropagation(10)
algorithm.start()
```

Viewing the results: This is simply a matter of going through all of the pixels, getting their most likely values, and printing them accordingly. You can obtain the most likely value of an element by using the `mostLikelyValue` method of `MPEBeliefPropagation`. Here's the code:

```
for {
  i <- 0 until 10
} {
  for { j <- 0 until 10 } {
    val mlv = algorithm.mostLikelyValue(pixels(i)(j))
    if (mlv) print('1') else print('0')
  }
  println()
}
```

To run the model, you need to provide it with some input. Ordinarily, this would be read from a file or provided programmatically by another module. To keep this example simple, you can define the input directly in the program, as follows:

```
val data =
  """00?000?000
     0?010?0010
     110?010011
     11??000111
     11011000?1
     1?0?100?10
     00001?0?00
     0010??0100
     01?01001?0
     0??000110?""".filterNot(_.isWhitespace)

  setEvidence(data)
```

Scala's """ constructor allows you to create long strings that span multiple lines. You filter out all whitespace characters to produce a string of 100 characters.

When run on this data, the program outputs the following:

```
0000000000
0001000010
1100010011
1100000111
1101100011
1000100010
0000100000
0010000100
0100100100
0000001100
```

That's all there is to Markov networks. This has been a long chapter, but you've learned a lot. Now you know all of the main principles of probabilistic models and can write probabilistic programs for a variety of applications.

5.6 Summary

- Probabilistic modeling is all about encoding relationships between variables. Symmetric relationships produce directed dependencies, whereas asymmetric relationships produce undirected dependencies.
- Directed dependencies lead from a cause to an effect. There are a variety of cause-and-effect relationships.
- Bayesian networks encode directed dependencies by using a directed acyclic graph.
- The direction of the arrows in a Bayesian network isn't necessarily the direction of reasoning. Bayesian networks can be used to reason in all directions in the network.
- Markov networks encode undirected dependencies by using an undirected graph.
- If you can identify the types of relationships between variables in your model and use them to write your programs, you won't go far wrong.

5.7 Exercises

Solutions to selected exercises are available online at www.manning.com/books/practical-probabilistic-programming.

1. For each of the following pairs of variables, decide whether the dependency between them should be directed or undirected, and if directed, which way the arrow should go.
   a. A player's poker cards and the player's bet
   b. Player 1's poker cards and player 2's poker cards
   c. My mood and today's weather
   d. My mood and whether I've had breakfast
   e. The temperature in my living room and the house thermostat setting

- f The temperature in my living room and the thermometer reading in the living room
- g The temperature in my living room and the temperature in the kitchen
- h The topic of a news article and the contents of the article
- i The summary of a news article and the contents of the article

2 For each of the following sets of variables, draw a Bayesian network over the variables.

- a Player 1's poker cards, player 2's poker cards, player 1's bet, and player 2's subsequent bet
- b My mood on waking up, my mood at 10 a.m., today's weather, and whether I've had breakfast
- c The temperature in my living room, the temperature in the kitchen, the house thermostat setting, the thermometer reading in the living room, and the thermometer reading in the kitchen
- d The topic of a news article, the summary of the article, the contents of the article, and the comments on the article

3 Your task is to design a Bayesian network to model the process of cooking soup. The goal of this network is to help you decide what ingredients to use and how much of each ingredient, as well as cooking variables such as the amount of heat and time, in order to optimize various food qualities such as spiciness and creaminess.

- a What are the variables in your model?
- b Draw a Bayesian network structure over these variables.
- c Choose Figaro functional forms for each of your variables.
- d Populate your functional forms in Figaro with numerical parameters.
- e Use your Figaro model to answer queries such as how long you should cook the soup with a given set of ingredients to ensure optimal creaminess.

4 A match of tennis consists of a number of sets, each of which consists of a number of games. The first player to win two sets wins the match. The first player to win six games wins a set. (Let's ignore tie breakers, but after doing exercise 5, you'll be able to model them.) Players alternate serving for a game at a time. Write a Figaro program that takes two arguments—the probability that each player wins a game in which he's the server—and predicts the winner of the match.

5 Now elaborate your tennis model to model individual points. Within a game, the players try to get points. The first player to get four points wins the game, unless both players get to three points. In that case, the first player to move two points ahead of his opponent wins the game. Write a Figaro program that takes two arguments as before, except that now they're the probability that each

player wins a point in which he's the server. Again, your program should predict the winner of the match.

6 Now you can further elaborate the tennis model to model individual points as rallies. Players could have variables such as serving ability, speed, and error proneness. Your model can be as detailed as you like, including the positions of the players and the ball at each shot in the rally.

7 My house has central air conditioning controlled by a thermostat downstairs. The top floor is usually hotter than the ground floor. Create a Markov network representing the temperature throughout the house (ignoring the thermostat for now). Write a Figaro model to represent the network. Use the model to compute the probability that the top floor is at least 80 degrees Fahrenheit, given that the ground floor is 72 degrees Fahrenheit.

8 Now add the thermostat to the model as well as the outside temperature and whether any windows are open. This model will combine directed and undirected dependencies, so it won't be a pure Markov network, but it's easy enough to make this combination in Figaro. Write the Figaro program and use it to help decide whether I should open the window on the top floor.

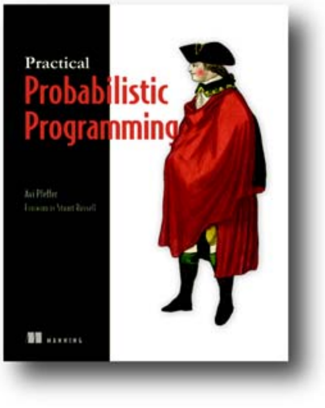
9 Consider the spam filter application from chapter 3. In that application, every email was treated as independent. Now suppose you have multiple emails from the same sender. Their spam status will be highly correlated.

   a Create a Markov network to capture these correlations.

   b The Bayesian network for each spam email is shown in figure 3.8. Replicate this network in the Markov network from exercise 9a. Again, this is a combined directed and undirected network.

Note: Although I've said that the class of an object determines its properties causally, which is the approach used in the spam filter, in some cases it makes sense to go in the opposite direction. This is the case when all of the features are always observed, as is the case with the spam filter. In this case, it can be wasteful to model the probability distribution over the observed features explicitly. Instead, a model can be created where the features of each email determine the class of an email. The classes of the email are then related by the Markov network from exercise 9a. This kind of model, known as a *conditional random field*, is widely used in natural language understanding and computer vision applications.

I'm not going to go into details of conditional random fields here, but I'll note, for those who are familiar with them, that they can easily be represented in Figaro. The trick is to make sure that the observed email features aren't Figaro elements but rather Scala variables that help determine the distribution over the element representing whether the email is spam. Any learnable parameters of the model will, of course, be Figaro elements, and they can interact with the Scala variables representing the features to determine the spam probability.



The data you accumulate about your customers, products, and website users can help you not only to interpret your past, but also to predict your future! Probabilistic programming uses code to draw probabilistic inferences from data. By applying specialized algorithms, your programs assign degrees of probability to conclusions. This means you can forecast future events like sales trends, computer system failures, experimental outcomes, and many other critical concerns.

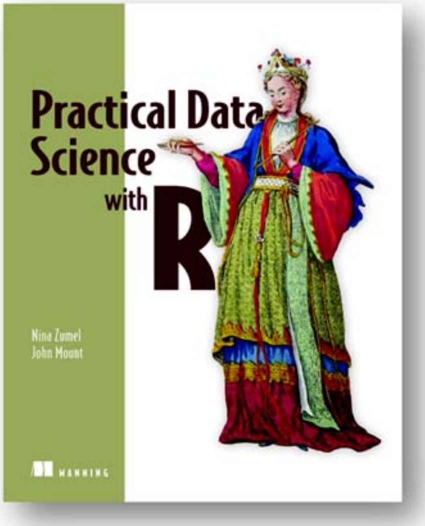
Practical Probabilistic Programming introduces the working programmer to probabilistic programming. In this book, you'll immediately work on practical examples like building a spam filter, diagnosing computer system data problems, and recovering digital images. You'll discover probabilistic inference, where algorithms help make extended predictions about issues like social media usage. Along the way, you'll learn to use functional-style programming for text analysis, object-oriented models to predict social phenomena like the spread of tweets, and open universe models to gauge real-life social media usage. The book also has chapters on how probabilistic models can help in decision-making and modeling of dynamic systems.

What's inside

- Introduction to probabilistic modeling
- Writing probabilistic programs in Figaro
- Building Bayesian networks
- Predicting product lifecycles
- Decision-making algorithms

This book assumes no prior exposure to probabilistic programming. Knowledge of Scala is helpful.

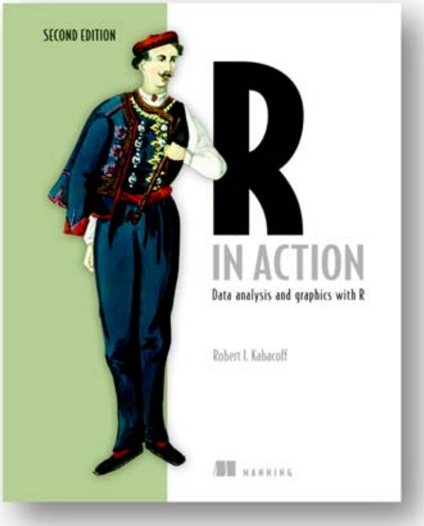
index



Practical Data Science with R

by Nina Zumel and John Mount

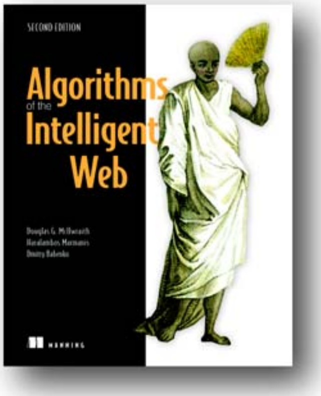
ISBN: 9781617291562
416 pages
$49.99
March 2014



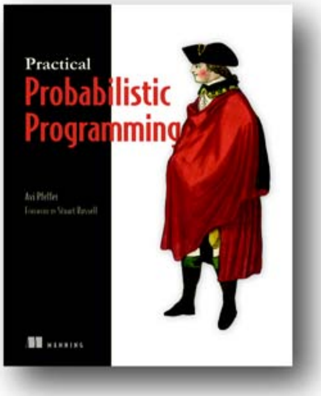
R in Action, Second Edition
Data analysis and graphics with R

by Robert I. Kabacoff

ISBN: 9781617291388
608 pages
$59.99
May 2015


SECOND EDITION
Algorithms of the Intelligent Web
Douglas G. McIlwraith
Haralambos Marmanis
Dmitry Babenko
MANNING

Big data, machine learning, and more, using Python tools
Introducing Data Science
Davy Cielen
Arno D. B. Meysman
Mohamed Ali
MANNING


Practical Probabilistic Programming
Avi Pfeffer
Foreword by Stuart Russell
MANNING