

THIRD EDITION



R

IN ACTION

Robert I. Kabacoff

MEAP

 MANNING



MEAP Edition
Manning Early Access Program
R in Action
Third Edition
Version 7

Copyright 2020 Manning Publications

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

©Manning Publications Co. To comment go to liveBook

welcome

Thank you for purchasing the MEAP for *R in Action (3rd edition)*. If you picked up this book, you probably have some data that you need to collect, summarize, transform, explore, model, visualize, or present. If so, then R is for you! R has become the worldwide language for statistics, predictive analytics, and data visualization. It offers the widest range of methodologies for understanding data currently available, from the most basic to the most complex and bleeding edge.

This book should appeal to anyone who deals with data. No background in statistical programming or the R language is assumed. Although the book is accessible to novices, there should be enough new and practical material to satisfy even experienced R mavens.

There is a generally held notion that R is difficult to learn. What I hope to show you is that is doesn't have to be. R is broad and powerful, with so many analytic and graphic functions available (more than 80,000 at last count) that it easily intimidates both novice and experienced users alike. But there is rhyme and reason to the apparent madness. With guidelines and instructions, you can navigate the tremendous resources available, selecting the tools you need to accomplish your work with style, elegance, efficiency, and certain degree of coolness.

By the end of the book, you should be able to use R to

- Access data (importing data from multiple sources)
- Clean data (code missing data, fix or delete miscoded data, transform variables into more useful formats)
- Explore and summarize data (getting descriptive statistics to help characterize the data)
- Visualize data (using a wide range of attractive and meaningful graphs)
- Model data (uncovering relationships, testing hypotheses, and developing predictive models using both basic and advanced statistical techniques and cutting edge machine learning approaches)
- Prepare results for others (creating publication-quality tables, graphs, and reports)

If you have any questions, comments, or suggestions, please share them in Manning's [Author Online forum](#) for my book. Your comments are invaluable and will help me craft content that is easier to understand and use effectively.

—Rob Kabacoff

brief contents

PART 1: GETTING STARTED

- 1 Introduction to R*
- 2 Creating a dataset*
- 3 Basic data management*
- 4 Getting started with graphs*
- 5 Advanced data management*

PART 2: BASIC METHODS

- 6 Basic Graphs*
- 7 Basic Statistics*

PART 3: INTERMEDIATE METHODS

- 8 Regression*
- 9 Analysis of Variance*
- 10 Power Analysis*
- 11 Intermediate Graphs*
- 12 Resampling statistics and bootstrapping*

PART 4: ADVANCED METHODS

- 13 Generalized Linear Models*
- 14 Principal Components and Factor Analysis*
- 15 Time Series*
- 16 Clustering*

17 Classification

18 Advanced methods for missing data

PART 5: EXPANDING YOUR SKILLS

19 Advanced Graphics with ggplot2

20 Advanced R Programming

21 Creating a Package

APPENDIXES:

A Version control with git

B Customizing the startup environment

C Exporting data from R

D Matrix Algebra in R

E Packages used in this book

F Working with large datasets

G Updating an R installation

1

Introduction to R

This chapter covers

- **Installing R and RStudio**
- **Understanding the R language**
- **Running programs**

How we analyze data has changed dramatically in recent years. With the advent of personal computers and the internet, the sheer volume of data we have available has grown enormously. Companies have terabytes of data about the consumers they interact with, and governmental, academic, and private research institutions have extensive archival and survey data on every manner of research topic. Gleaning information (let alone wisdom) from these massive stores of data has become an industry in itself. At the same time, presenting the information in easily accessible and digestible ways has become increasingly challenging.

The science of data analysis (statistics, psychometrics, econometrics, and machine learning) has kept pace with this explosion of data. Before personal computers and the internet, new statistical methods were developed by academic researchers who published their results as theoretical papers in professional journals. It could take years for these methods to be adapted by programmers and incorporated into the statistical packages widely available to data analysts. Today, new methodologies appear *daily*. Statistical researchers publish new and improved methods, along with the code to produce them, on easily accessible websites.

The advent of personal computers had another effect on the way we analyze data. When data analysis was carried out on mainframe computers, computer time was precious and difficult to come by. Analysts would carefully set up a computer run with all the parameters and options thought to be needed. When the procedure ran, the resulting output could be dozens or hundreds of pages long. The analyst would sift through this output, extracting

useful material and discarding the rest. Many popular statistical packages (such as SAS and SPSS) were originally developed during this period and still follow this approach to some degree.

With the cheap and easy access afforded by personal computers, modern data analysis has shifted to a different paradigm. Rather than setting up a complete data analysis all at once, the process has become highly interactive, with the output from each stage serving as the input for the next stage. An example of a typical analysis is shown in figure 1.1. At any point, the cycles may include transforming the data, imputing missing values, adding or deleting variables, fitting statistical models, and looping back through the whole process again. The process stops when the analyst believes they understand the data intimately and have answered all the relevant questions that can be answered.

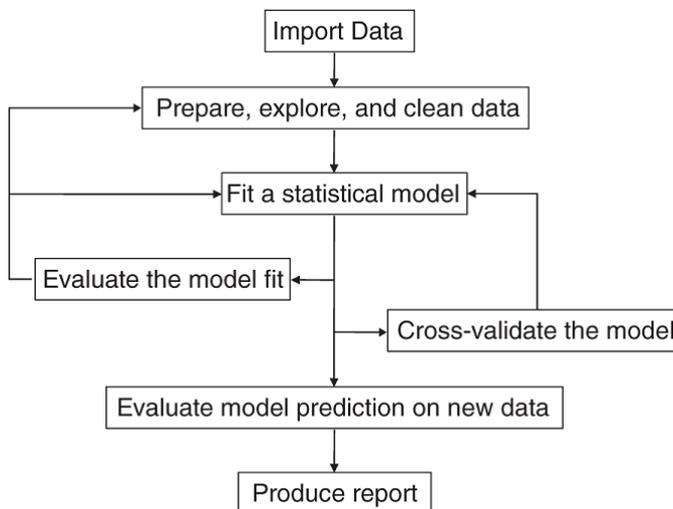


Figure 1.1 Steps in a typical data analysis

The advent of personal computers (and especially the availability of high-resolution monitors) has also had an impact on how results are understood and presented. A picture *really can* be worth a thousand words, and human beings are adept at extracting useful information from visual presentations. Modern data analysis increasingly relies on graphical presentations to uncover meaning and convey results.

Today's data analysts need to access data from a wide range of sources (database management systems, text files, statistical packages, spreadsheets, and web pages), merge the pieces of data together, clean and annotate them, analyze them with the latest methods, present the findings in meaningful and graphically appealing ways, and incorporate the results into attractive reports that can be distributed to stakeholders and the public. As you'll see in

the following pages, R is a comprehensive software package that's ideally suited to accomplish these goals.

1.1 Why use R?

R is a language and environment for statistical computing and graphics, similar to the S language originally developed at Bell Labs. It's an open source solution to data analysis that's supported by a large and active worldwide research community. But there are many popular statistical and graphing packages available (such as Microsoft Excel, SAS, IBM SPSS, Stata, and Minitab). Why turn to R?

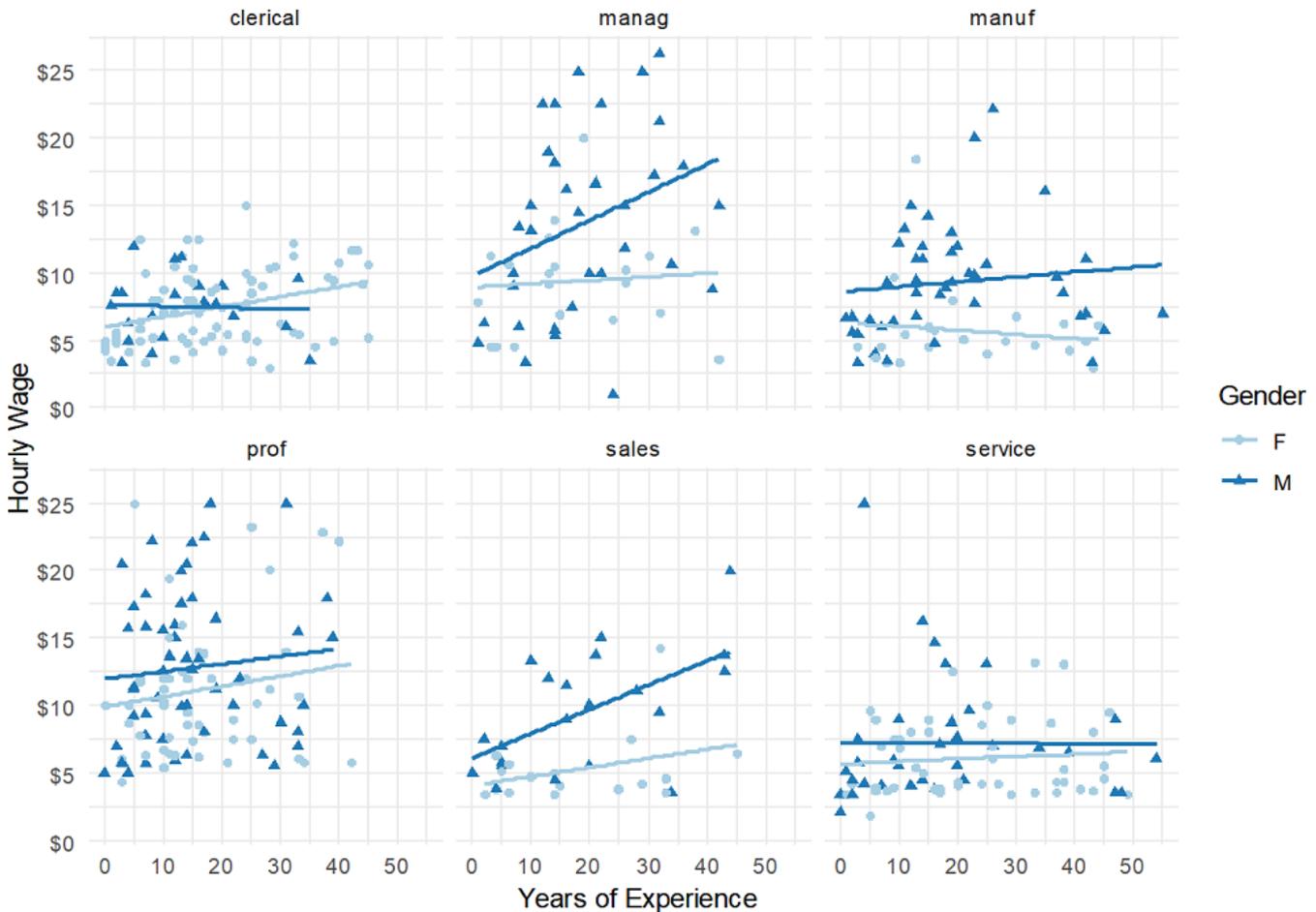
R has many features to recommend it:

- Most commercial statistical software platforms cost thousands, if not tens of thousands, of dollars. R is free! If you're a teacher or a student, the benefits are obvious.
- R is a comprehensive statistical platform, offering all manner of data-analytic techniques. Just about any type of data analysis can be done in R.
- R contains advanced statistical routines not yet available in other packages. In fact, new methods become available for download on a weekly basis. If you're a SAS user, imagine getting a new SAS PROC every few days.
- R has state-of-the-art graphics capabilities. If you want to visualize complex data, R has the most comprehensive and powerful feature set available.
- R is a powerful platform for interactive data analysis and exploration. From its inception, it was designed to support the approach outlined in figure 1.1. For example, the results of any analytic step can easily be saved, manipulated, and used as input for additional analyses.
- Getting data into a usable form from multiple sources can be a challenging proposition. R can easily import data from a wide variety of sources, including text files, database-management systems, statistical packages, and specialized data stores. It can write data out to these systems as well. R can also access data directly from web pages, social media sites, and a wide range of online data services.
- R provides an unparalleled platform for programming new statistical methods in an easy, straightforward manner. It's easily extensible and provides a natural language for quickly programming recently published methods.
- R functionality can be integrated into applications written in other languages, including C++, Java, Python, PHP, Pentaho, SAS, and SPSS. This allows you to continue working in a language that you may be familiar with, while adding R's capabilities to your applications.
- R runs on a wide array of platforms, including Windows, Unix, and Mac OS X. It's likely to run on any computer you may have. (I've even come across guides for installing R on an iPhone, which is impressive but probably not a good idea.)
- If you don't want to learn a new language, a variety of graphic user interfaces (GUIs) are available, offering the power of R through menus and dialogs.

You can see an example of R's graphic capabilities in figure 1.2. This graph describes the relationships between years of experience and wages for men in women in six industries, collected from the US Current Population Survey in 1985. Technically, it's a matrix of scatterplots with gender displayed by color and symbol. Trends are described using linear regression lines. If these terms scatterplot and regression lines are unfamiliar to you, don't worry. We'll cover them in later chapters.

Relationship between wages and experience

Current Population Survey



source: <http://mosaic-web.org/>

Figure 1.2 Relationships between wages and years of experience for men and women in six industries. Source: mosaicData package. Graphs like this can be created easily with a few lines of code in R.

Some of the more interesting findings from this graph:

- The relationship between experience and wages varies by both gender and industry.
- In the service industry, wages do not appear to go up with experience for either men or women.
- In management positions, wages tend to go up with experience for men, but not for women.

Are these differences real or can they be explained as chance sampling variation? We'll discuss this further in Chapter 8 Regression. The important point is that R allows you to create elegant, informative, highly customized graphs in a simple and straightforward fashion. Creating similar plots in other statistical languages would be difficult, time-consuming, or impossible.

Unfortunately, R can have a steep learning curve. Because it can do so much, the documentation and help files available are voluminous. Additionally, because much of the functionality comes from optional modules created by independent contributors, this documentation can be scattered and difficult to locate. In fact, getting a handle on all that R can do is a challenge.

The goal of this book is to make access to R quick and easy. We'll tour the many features of R, covering enough material to get you started on your data, with pointers on where to go when you need to learn more. Let's begin by installing the program.

1.2 Obtaining and installing R

R is freely available from the Comprehensive R Archive Network (CRAN) at <http://cran.r-project.org>. Precompiled binaries are available for Linux, Mac OS X, and Windows. Follow the directions for installing the base product on the platform of your choice. Later we'll talk about adding functionality through optional modules called *packages* (also available from CRAN). Appendix A describes how to install or update an existing R installation to a newer version.

1.3 Working with R

R is a case-sensitive, interpreted language. You can enter commands one at a time at the command prompt (`>`) or run a set of commands from a source file. There are a wide variety of data types, including vectors, matrices, data frames (similar to datasets), and lists (collections of objects). We'll discuss each of these data types in chapter 2.

Most functionality is provided through built-in and user-created functions and the creation and manipulation of objects. An *object* is basically anything that can be assigned a value. For R, that is just about everything (data, functions, graphs, analytic results, and more). Every object has a *class attribute* (basically one or more associated text descriptors) that tells R how to print, plot, summarize, or in some other way, manipulate the object.

All objects are kept in memory during an interactive session. Basic functions are available by default. Other functions are contained in packages that can be attached to a current session as needed.

Statements consist of functions and assignments. R uses the symbol `<-` for assignments, rather than the typical `=` sign. For example, the statement

```
x <- rnorm(5)
```

creates a vector object named `x` containing five random deviates from a standard normal distribution.

NOTE R allows the `=` sign to be used for object assignments. But you won't find many programs written that way, because it's not standard syntax, there are some situations in which it won't work, and R programmers will make fun of you. You can also reverse the assignment direction. For instance, `rnorm(5) -> x` is equivalent to the previous statement. Again, doing so is uncommon and isn't recommended in this book.

Comments are preceded by the `#` symbol. Any text appearing after the `#` is ignored by the R interpreter. An example program is given in the next section.

1.3.1 Getting started

The first step in using R is, of course, to install it. Instructions are provided in Appendix A. Once R is installed, start it up. If you're using Windows, launch R from the Start menu. On a Mac, double-click the R icon in the Applications folder. For Linux, type `R` at the command prompt of a terminal window. Any of these will start the R interface (see figure 1.3 for an example).

To get a feel for the interface, let's work through a simple, contrived example. Say that you're studying physical development and you've collected the ages and weights of 10 infants in their first year of life (see table 1.1). You're interested in the distribution of the weights and their relationship to age.

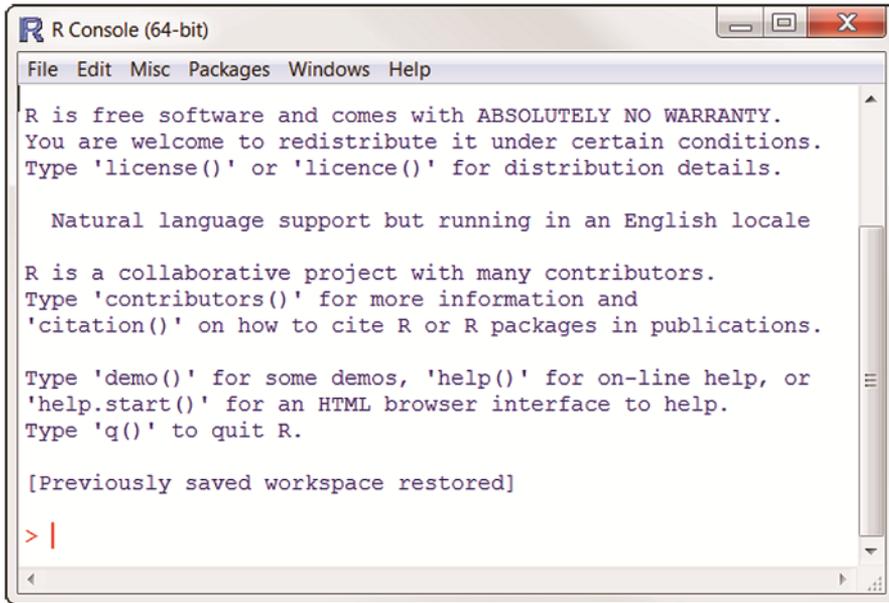


Figure 1.3 Example of the R interface on Windows

Table 1.1 The ages and weights of 10 infants

Age (mo.)	Weight (kg.)
01	4.4
03	5.3
05	7.2
02	5.2
11	8.5
09	7.3
03	6.0
09	10.4
12	10.2
03	6.1

Note: These are fictional data.

The analysis is given in listing 1.1. Age and weight data are entered as vectors using the function `c()`, which combines its arguments into a vector or list. The mean and standard

deviation of the weights, along with the correlation between age and weight, are provided by the functions `mean()`, `sd()`, and `cor()`, respectively. Finally, age is plotted against weight using the `plot()` function, allowing you to visually inspect the trend. The `q()` function ends the session and lets you quit.

Listing 1.1 A sample R session

```
> age <- c(1,3,5,2,11,9,3,9,12,3)
> weight <- c(4.4,5.3,7.2,5.2,8.5,7.3,6.0,10.4,10.2,6.1)
> mean(weight)
[1] 7.06
> sd(weight)
[1] 2.077498
> cor(age,weight)
[1] 0.9075655
> plot(age,weight)
> q()
```

You can see from listing 1.1 that the mean weight for these 10 infants is 7.06 kilograms, that the standard deviation is 2.08 kilograms, and that there is strong linear relationship between age in months and weight in kilograms (correlation = 0.91). The relationship can also be seen in the scatter plot in figure 1.4. Not surprisingly, as infants get older, they tend to weigh more.

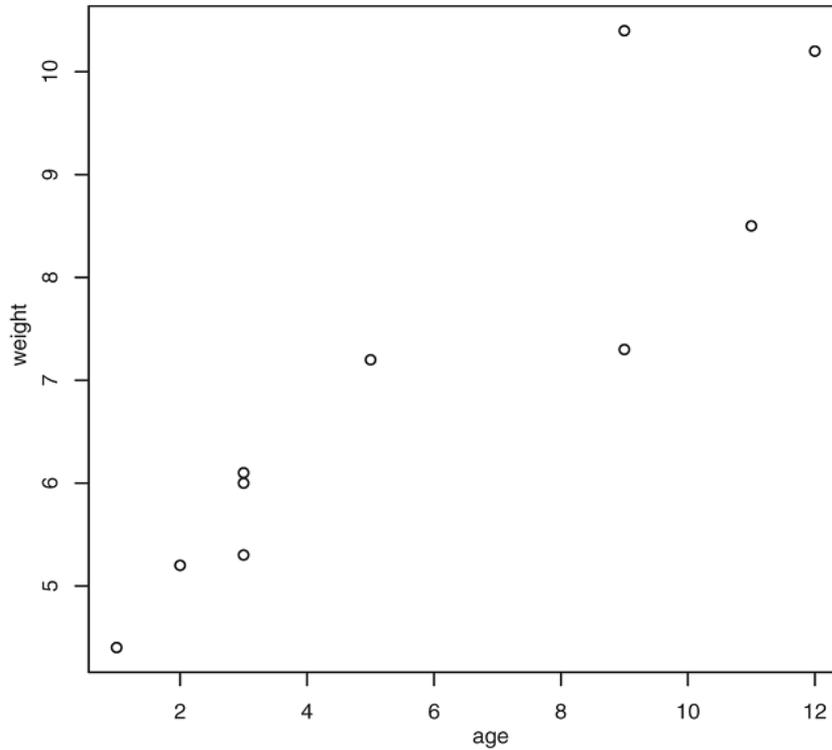


Figure 1.4 Scatter plot of infant weight (kg) by age (mo)

The scatter plot in figure 1.4 is informative but somewhat utilitarian and unattractive. In later chapters, you'll see how to create more attractive and sophisticated graphs.

TIP To get a sense of what R can do graphically, take a look at the graphs described in *Data Visualization with R* (<http://rkabacoff.github.io/datavis>) and *The Top 50 ggplot2 Visualizations – The Master List* (<http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html>).

1.3.2 Using RStudio

The standard interface to R is very basic, offering little more than a command prompt for entering lines of code. For real-life projects, you'll want a more comprehensive tool for writing code and viewing output. Several such tools, called Integrated Development Environments (IDEs) have been developed for R, including Eclipse with StatET, Visual Studio for R, and RStudio Desktop.

RStudio Desktop (<http://www.rstudio.com>) is by far the most popular choice. It provides a multi-window, multi-tabbed environment, with tools for importing data, writing clean code, debugging errors, visualizing output, and writing reports.

RStudio is freely available as an open source product, and is easily installed on Windows, Mac, and Linux. Since RStudio is an interface to R, be sure to install R before installing RStudio Desktop.

TIP You can customize the RStudio interface by selecting the **Tools > Global Options...** from the menu bar. On the **General** tab, I recommend unchecking **Restore .RData into workspace at startup**, and selecting **Never** for **Save workspace to .Rdata on exit**. This will ensure a clean startup each time you run RStudio.

Let's rerun the code from Listing 1.1 using RStudio. If you're using Windows, launch RStudio from the Start menu. On a Mac, double-click the RStudio icon in the Applications folder. For Linux, type `rstudio` at the command prompt of a terminal window. The same interface will appear on all three platforms (see Figure 1.5).

The screenshot displays the RStudio Desktop interface with four callout boxes identifying key components:

- A. script window - your program**: Points to the editor window containing R code for creating vectors, calculating statistics, and plotting.
- B. console window - where your program runs**: Points to the terminal window showing the execution output of the R code.
- C. Environment window - lists objects that have been created**: Points to the Environment pane showing the objects `age` and `weight` with their respective data types and values.
- D. Plot window**: Points to the plot window showing a scatter plot of `weight` versus `age`.

The script window contains the following R code:

```
1 age <- c(1,3,5,2,11,9,3,9,12,3)
2 weight <- c(4.4,5.3,7.2,5.2,8.5,7.3,6.0,10.4,10.2,6.1)
3 mean(weight)
4 sd(weight)
5 cor(age,weight)
6 plot(age,weight)
7
```

The console window shows the following output:

```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> age <- c(1,3,5,2,11,9,3,9,12,3)
> weight <- c(4.4,5.3,7.2,5.2,8.5,7.3,6.0,10.4,10.2,6.1)
> mean(weight)
[1] 7.06
> sd(weight)
[1] 2.077498
> cor(age,weight)
[1] 0.9075655
> plot(age,weight)
>
```

The Environment window shows the following values:

Object	Type	Value
age	num [1:10]	1 3 5 2 11 9 3 9 12 3
weight	num [1:10]	4.4 5.3 7.2 5.2 8.5 7.3 6 10.4 10.2 6.1

The plot window shows a scatter plot of `weight` versus `age` with the following data points:

age	weight
1	4.4
3	5.3
5	7.2
2	5.2
11	8.5
9	7.3
3	6.0
9	10.4
12	10.2
3	6.1

Figure 1.5 RStudio Desktop

SCRIPT WINDOW

From the **File** menu, select **New File > R Script**. A new script window will open in the upper right hand corner of the screen (Figure 1.5 A). Type the code from Listing 1.1. into this window.

As you type, the editor offers syntax highlighting and code completion (see figure 1.6). For example, as you type `plot` a pop-up window will appear with all functions that start with the letters that you've typed so far. Use can use the UP and DOWN arrow keys to select a function from the list and press TAB to select it. Within functions (parentheses) press TAB to see function options. Within quote marks, press TAB to complete file paths.

To execute code, highlight/select it and click the Run button or press Cntr+Enter. Pressing Cntrl+Shift+Enter will run the entire script.

To save the script, press the Save icon or select **File > Save** from the menu bar. Select a name and location from the dialog box that opens. By convention, script files end with a `.R` extension. The name of the script file will appear in the window tab in a red starred format if the current version has not been saved.

CONSOLE WINDOW

Code runs in the Console window (Figure 1.5 B). This is basically the same console you would see if you were using the basic R interface. You can submit code from a script window with a Run command, or enter interactive commands directly in this window at the command prompt (`>`).

If the command prompt changes to a plus (+) sign, the interpreter is waiting for a complete statement. This will often happen if the statement is too long for one line or if there are mismatched parentheses in the code. You can get back to the command prompt by pressing `ESC`.

Additionally, pressing the UP and DOWN arrow keys will cycle through past commands. You can edit a command and resubmit it with the ENTER key. Clicking on the broom icon clears text from the window.

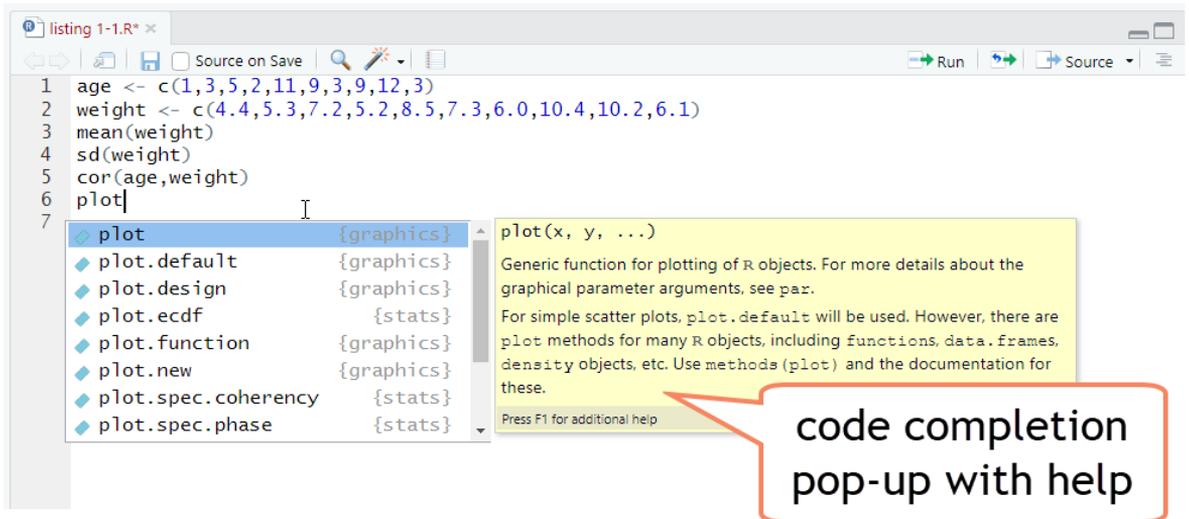


Figure 1.6 Script window

ENVIRONMENT AND HISTORY WINDOWS

Any objects that were created (`age` and `weight` in this example) will appear in the Environment window (see figure 1.5 C). A record of executed commands will be saved in the History window (the tab the right of Environment).

PLOT WINDOW

Any plots that are created from the script will appear in the plot window (Figure 1.5 D). The toolbar for this window allows you to cycle through the graphs that have been created. In addition, you can open a zoom window to see the graph at different sizes, export the graphs in several formats, and delete one or all the graphs created so far.

1.3.3 Getting help

R provides extensive help facilities, and learning to navigate them will help you significantly in your programming efforts. The built-in help system provides details, references, and examples of any function contained in a currently installed package. You can obtain help by executing any of functions listed in table 1.2.

Help is also available through the RStudio interface. In the Script window, place the cursor on a function name and press F1 to bring up the help window.

Table 1.2 R help functions

Function	Action
<code>help.start()</code>	General help
<code>help("foo")</code> or <code>?foo</code>	Help on function <code>foo</code>
<code>help(package = "foo")</code>	Help on a package named <code>foo</code>
<code>help.search("foo")</code> or <code>??foo</code>	Searches the help system for instances of the string <code>foo</code>
<code>example("foo")</code>	Examples of function <code>foo</code> (quotation marks optional)
<code>data()</code>	Lists all available example datasets contained in currently loaded packages
<code>vignette()</code>	Lists all available vignettes for currently installed packages
<code>vignette("foo")</code>	Displays specific vignettes for topic <code>foo</code>
<code>help.start()</code>	General help

The function `help.start()` opens a browser window with access to introductory and advanced manuals, FAQs, and reference materials. Alternatively, choose **Help > R Help** from the menu. The vignettes returned by the `vignette()` function are practical introductory articles provided in PDF or HTML format. Not all packages have vignettes.

All help files have a similar format (see figure 1.7). The help page has a title and brief description, followed by the function's syntax and options. Computational details are provided in the Details section. The See Also section describes and links to related functions. The help page almost always ends with examples illustrating typical uses of the function.

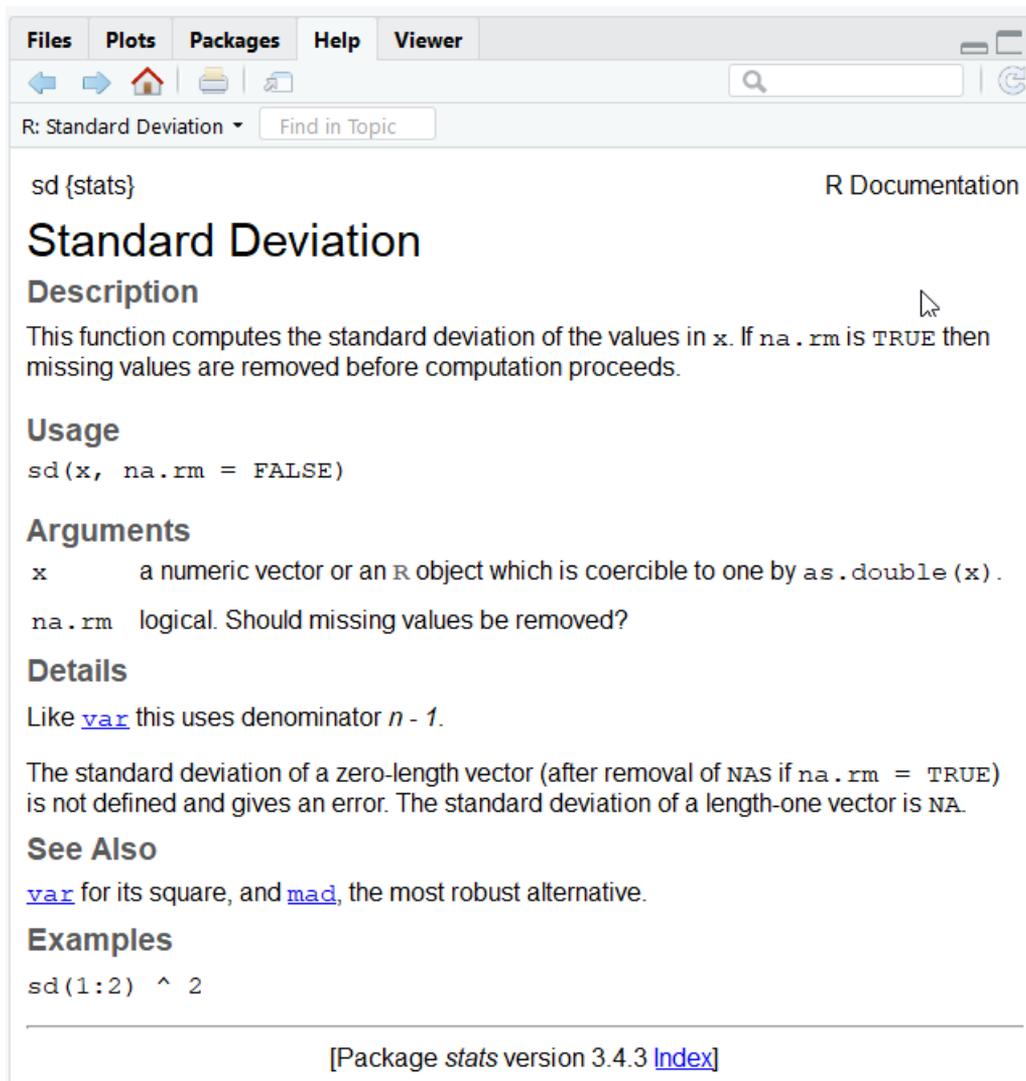


Figure 1.7 Help window

As you can see, R provides extensive help facilities, and learning to navigate them will definitely aid your programming efforts. It's a rare session that I don't use `?` to look up the features (such as options or return values) of some function.

1.3.4 The workspace

The workspace is your current R working environment and includes any user-defined objects (vectors, matrices, functions, data frames, and lists). The current working directory is the directory from which R will read files and to which it will save results by default. You can find out what the current working directory is by using the `getwd()` function. You can set the current working directory by using the `setwd()` function. If you need to input a file that isn't in the current working directory, use the full pathname in the call. Always enclose the names of files and directories from the operating system in quotation marks. Some standard commands for managing your workspace are listed in table 1.3.

Table 1.3 Functions for managing the R workspace

Function	Action
<code>getwd()</code>	Lists the current working directory.
<code>setwd("mydirectory")</code>	Changes the current working directory to <i>mydirectory</i> .
<code>ls()</code>	Lists the objects in the current workspace.
<code>rm(objectlist)</code>	Removes (deletes) one or more objects.
<code>help(options)</code>	Provides information about available options.
<code>options()</code>	Lets you view or set current options.
<code>save.image("myfile")</code>	Saves the workspace to <i>myfile</i> (default = <i>.RData</i>).
<code>save(objectlist, file="myfile")</code>	Saves specific objects to a file.
<code>load("myfile")</code>	Loads a workspace into the current session.

To see these commands in action, look at the following listing.

Listing 1.2 An example of commands used to manage the R workspace

```
setwd("C:/myprojects/project1")
options()
options(digits=3)
```

First, the current working directory is set to `C:/myprojects/project1`. The current option settings are then displayed, and numbers are formatted to print with three digits after the decimal place.

Note the forward slashes in the pathname of the `setwd()` command. R treats the backslash (`\`) as an escape character. Even when you're using R on a Windows platform, use forward slashes in pathnames. Also note that the `setwd()` function won't create a directory that doesn't exist. If necessary, you can use the `dir.create()` function to create a directory and then use `setwd()` to change to its location.

1.3.5 Projects

It's a good idea to keep your projects in separate directories. RStudio provides a simple mechanism for this. Choose **File > New Project ...** and specify either **New Directory** to start a project in a brand new working directory, or **Existing Directory** to associate a project with an existing working directory. All your program files, command history, report output, graphs and data will be saved in the project directory. You can easily switch between projects using the **Project** dropdown menu in the upper right portion of the RStudio application.

It is easy to become overwhelmed with project files. I recommend creating several subfolders within the main project folder. I usually create a *data* folder to contain raw data files, an *img* folder for image files and graphical output, a *docs* folder for project documentation, and a *reports* folder for reports. I keep the R scripts and a README file in the main directory. If there is an order to the R scripts I number them (e.g., 01_import_data.R, 02_clean_data.R, etc.). The README is a text file containing information such as author, date, stakeholders and their contact information, and the purpose of the project. Six months from now, this will remind me what I did and why I did it.

1.4 Packages

R comes with extensive capabilities right out of the box. But some of its most exciting features are available as optional modules that you can download and install. There are more than 10,000 user-contributed modules called *packages* that you can download from <http://cran.r-project.org/web/packages>. They provide a tremendous range of new capabilities, from the analysis of geospatial data to protein mass spectra processing to the analysis of psychological tests! You'll use many of these optional packages in this book.

One set of packages, collectively called the *tidyverse*, deserves particular attention. This is a relatively new collection of packages that offers a streamlined, consistent, and intuitive approach to data manipulation and analysis. The advantages offered by tidyverse packages (with names like *tidyr*, *dplyr*, *lubridate*, *stringr* and *ggplot2*) is changing the way data scientists write code in R, and we will be employing these packages often. In fact, the opportunity to describe how to use these packages for data analysis and visualization was a major motivation for writing a third edition of this book!

1.4.1 What are packages?

Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored on your computer is called the *library*. The function `.libPaths()` shows you where your library is located, and the function `library()` shows you what packages you've saved in your library.

R comes with a standard set of packages (including *base*, *datasets*, *utils*, *grDevices*, *graphics*, *stats*, and *methods*). They provide a wide range of functions and datasets that are available by default. Other packages are available for download and installation. Once

installed, they must be loaded into the session in order to be used. The command `search()` tells you which packages are loaded and ready to use.

1.4.2 Installing a package

A number of R functions let you manipulate packages. To install a package for the first time, use the `install.packages()` command. For example, the `gclus` package contains functions for creating enhanced scatter plots. You can download and install the package with the command `install.packages("gclus")`.

You only need to install a package once. But like any software, packages are often updated by their authors. Use the command `update.packages()` to update any packages that you've installed. To see details on your packages, you can use the `installed.packages()` command. It lists the packages you have, along with their version numbers, dependencies, and other information.

You can also install and update packages using the RStudio interface. Select the Packages tab (from the window on the lower right). Enter the name (or partial name) in the search box in the upper right of that tabbed window. Place a check mark next to the package(s) you want to install and click the install button. Alternatively, click the update button to update a package already installed.

1.4.3 Loading a package

Installing a package downloads it from a CRAN mirror site and places it in your library. To use it in an R session, you need to load the package using the `library()` command. For example, to use the package `gclus`, issue the command `library(gclus)`.

Of course, you must have installed a package before you can load it. You'll only have to load the package once in a given session. If desired, you can customize your startup environment to automatically load the packages you use most often. Customizing your startup is covered in appendix B.

1.4.4 Learning about a package

When you load a package, a new set of functions and datasets becomes available. Small illustrative datasets are provided along with sample code, allowing you to try out the new functionalities. The help system contains a description of each function (along with examples) and information about each dataset included. Entering `help(package="package_name")` provides a brief description of the package and an index of the functions and datasets included. Using `help()` with any of these function or dataset names provides further details. The same information can be downloaded as a PDF manual from CRAN. To get help on a package using the RStudio interface, click on the Packages tab (lower right window), enter the name of the package in the search window, and click on the name of the package.

Common mistakes in R programming

Some common mistakes are made frequently by both beginning and experienced R programmers. If your program generates an error, be sure to check for the following:

- **Using the wrong case**—`help()`, `Help()`, and `HELP()` are three different functions (only the first will work).
- **Forgetting to use quotation marks when they're needed**—`install.packages-"gclus")` works, whereas `install.packages(gclus)` generates an error.
- **Forgetting to include the parentheses in a function call**—For example, `help()` works, but `help` doesn't. Even if there are no options, you still need the `()`.
- **Using the `\` in a pathname on Windows**—R sees the backslash character as an escape character. `setwd("c:\mydata")` generates an error. Use `setwd("c:/mydata")` or `setwd("c:\\mydata")` instead.
- **Using a function from a package that's not loaded**—The function `order.clusters()` is contained in the `gclus` package. If you try to use it before loading the package, you'll get an error.

The error messages in R can be cryptic, but if you're careful to follow these points, you should avoid seeing many of them.

1.5 Using output as input: reusing results

One of the most useful design features of R is that the output of analyses can easily be saved and used as input to additional analyses. Let's walk through an example, using one of the datasets that comes preinstalled with R. If you don't understand the statistics involved, don't worry. We're focusing on the general principle here.

R comes with many built in datasets that can be used to practice data analyses. One such dataset, called *mtcars*, contains information 32 automobiles collected from *Motor Trend* magazine road tests. Suppose we're interested in describing the relationship between a car's fuel efficiency and weight.

First, we could run a simple linear regression predicting miles per gallon (`mpg`) from car weight (`wt`). This is accomplished with the following function call:

```
lm(mpg~wt, data=mtcars)
```

The results are displayed on the screen, and no information is saved.

Alternatively, run the regression, but store the results in an object:

```
lmfit <- lm(mpg~wt, data=mtcars)
```

The assignment creates a list object called `lmfit` that contains extensive information from the analysis (including the predicted values, residuals, regression coefficients, and more). Although no output is sent to the screen, the results can be both displayed and manipulated further.

Typing `summary(lmfit)` displays a summary of the results, and `plot(lmfit)` produces diagnostic plots. The statement `cook<-cooks.distance(lmfit)` generates and stores

influence statistics, and `plot(cook)` graphs them. To predict miles per gallon from car weight in a new set of data, you'd use `predict(lmfit, mynewdata)`.

To see what a function returns, look at the *Value* section of the R help page for that function. Here you'd look at `help(lm)` or `?lm`. This tells you what's saved when you assign the results of that function to an object.

1.6 Working with large datasets

Programmers frequently ask me if R can handle large data problems. Typically, they work with massive amounts of data gathered from web research, climatology, or genetics. Because R holds objects in memory, you're generally limited by the amount of RAM available. For example, on my 3-year-old Windows PC with 8 GB of RAM, I can easily handle datasets with 10 million elements (100 variables by 100,000 observations). On an iMac with 16 GB of RAM, I can usually handle 100 million elements without difficulty.

But there are two issues to consider: the size of the dataset and the statistical methods that will be applied. R can handle data analysis problems in the gigabyte to terabyte range, but specialized procedures are required. The management and analysis of very large datasets is discussed in appendix F.

1.7 Working through an example

We'll finish this chapter with an example that ties together many of these ideas. Here's the task:

1. Open the general help, and look at the "Introduction to R" section.
2. Install the `vcd` package (a package for visualizing categorical data that you'll be using in chapter 11).
3. List the functions and datasets available in this package.
4. Load the package, and read the description of the dataset `Arthritis`.
5. Print out the `Arthritis` dataset (entering the name of an object will list it).
6. Run the example that comes with the `Arthritis` dataset. Don't worry if you don't understand the results; it basically shows that arthritis patients receiving treatment improved much more than patients receiving a placebo.

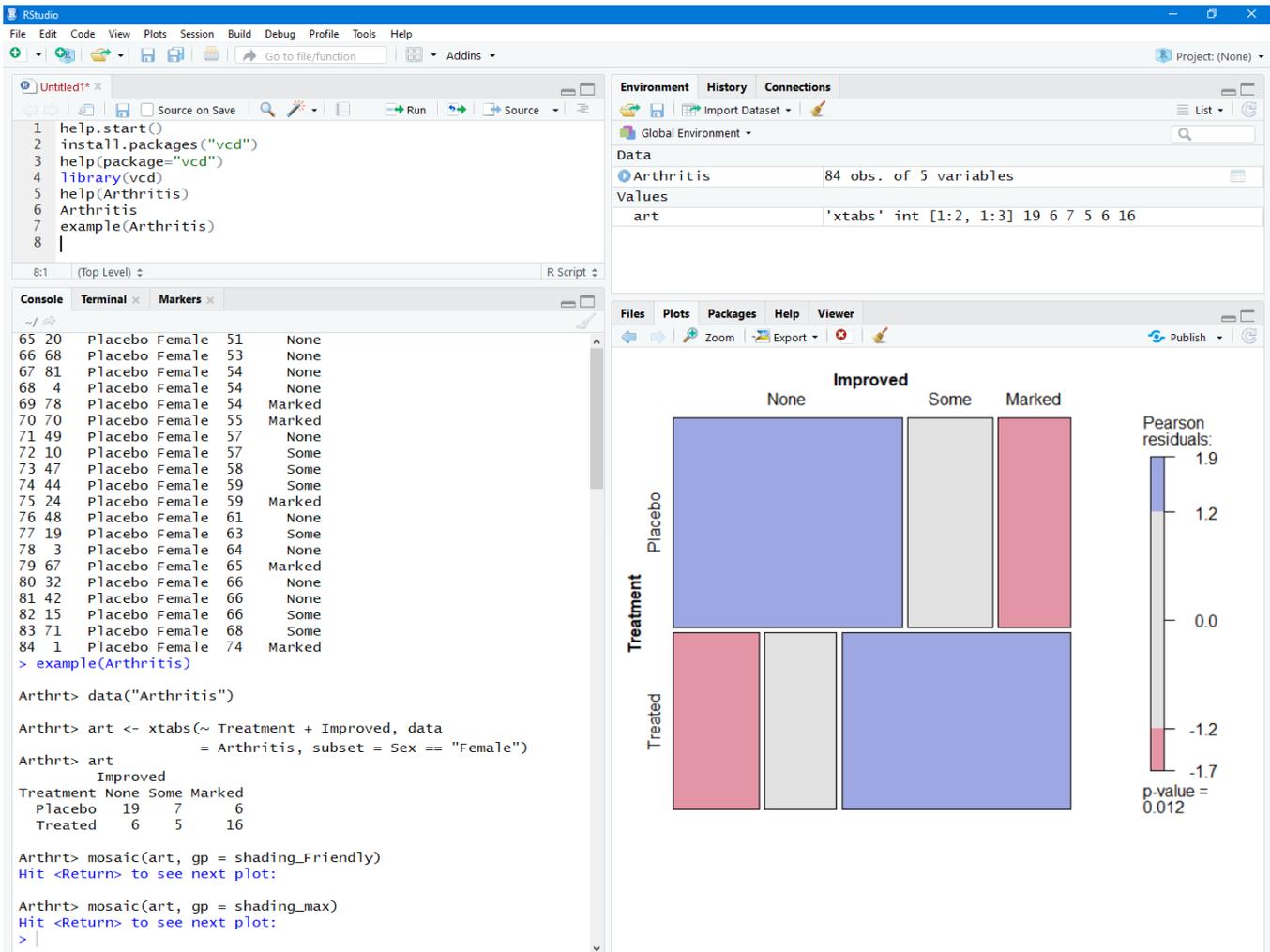


Figure 1.8 RStudio window when executing the code in Listing 1.3.

The code required is provided in the following listing, with a sample of the results displayed in figure 1.8. As this short exercise demonstrates, you can accomplish a great deal with a small amount of code.

Listing 1.3 Working with a new package

```
help.start()
install.packages("vcd")
help(package="vcd")
library(vcd)
```

```
help(Arthritis)
Arthritis
example(Arthritis)
```

1.8 Summary

- R provides a comprehensive, highly interactive environment for analyzing and visualizing data.
- RStudio is an integrated development environment that makes programming in R easier and more productive.
- Packages are freely available add-on modules that greatly extend the power of the R platform.
- R has an extensive help system and learning to use it will greatly facilitate your ability to program effectively.

In this chapter, we looked at some of the strengths that make R an attractive option for students, researchers, statisticians, and data analysts trying to understand the meaning of their data. We walked through the program's installation and talked about how to enhance R's capabilities by downloading additional packages. We explored the basic interface and produced a few simple graphs. Because R can be a complex program, we spent some time looking at how to access the extensive help that's available. Hopefully you're getting a sense of how powerful this freely available software can be.

Now that you have R and RStudio up and running, it's time to get your data into the mix. In the next chapter, we'll look at the types of data R can handle and how to import them into R from text files, other programs, and database management systems.

2

Creating a dataset

This chapter covers

- Exploring R data structures
- Using data entry
- Importing data
- Annotating datasets

The first step in any data analysis is the creation of a dataset containing the information to be studied, in a format that meets your needs. In R, this task involves the following:

- Selecting a data structure to hold your data
- Entering or importing your data into the data structure

The first part of this chapter (sections 2.1–2.2) describes the wealth of structures that R can use to hold data. In particular, section 2.2 describes vectors, factors, matrices, data frames, and lists. Familiarizing yourself with these structures (and the notation used to access elements within them) will help you tremendously in understanding how R works. You might want to take your time working through this section.

The second part of this chapter (section 2.3) covers the many methods available for importing data into R. Data can be entered manually or imported from an external source. These data sources can include text files, spreadsheets, statistical packages, and database-management systems. For example, the data that I work with typically comes as comma delimited text files or EXCEL spreadsheets. On occasion, though, I receive data as SAS and SPSS datasets or through connections to SQL databases. It's likely that you'll only have to use one or two of the methods described in this section, so feel free to choose those that fit your situation.

Once a dataset is created, you'll typically annotate it, adding descriptive labels for variables and variable codes. The third portion of this chapter (section 2.4) looks at annotating datasets and reviews some useful functions for working with datasets (section 2.5). Let's start with the basics.

2.1 Understanding datasets

A dataset is usually a rectangular array of data with rows representing observations and columns representing variables. Table 2.1 provides an example of a hypothetical patient dataset.

Table 2.1 A patient dataset

PatientID	AdmDate	Age	Diabetes	Status
1	10/15/2018	25	Type1	Poor
2	11/01/2018	34	Type2	Improved
3	10/21/2018	28	Type1	Excellent
4	10/28/2018	52	Type1	Poor

Different traditions have different names for the rows and columns of a dataset. Statisticians refer to them as observations and variables, database analysts call them records and fields, and those from the data-mining and machine-learning disciplines call them examples and attributes. We'll use the terms *observations* and *variables* throughout this book.

You can distinguish between the structure of the dataset (in this case, a rectangular array) and the contents or data types included. In the dataset shown in table 2.1, `PatientID` is a row or case identifier, `AdmDate` is a date variable, `Age` is a continuous (quantitative) variable, `Diabetes` is a nominal variable, and `Status` is an ordinal variable. Both nominal and ordinal variables are categorical, but the categories in an ordinal variable have a natural ordering.

R contains a wide variety of structures for holding data, including scalars, vectors, arrays, data frames, and lists. Table 2.1 corresponds to a data frame in R. This diversity of structures provides the R language with a great deal of flexibility in dealing with data.

The data types that R can handle include numeric, character, logical (TRUE/FALSE), complex (imaginary numbers), and raw (bytes). In R, `PatientID`, `AdmDate`, and `Age` are numeric variables, whereas `Diabetes` and `Status` are character variables. Additionally, you need to tell R that `PatientID` is a case identifier, that `AdmDate` contains dates, and that `Diabetes` and `Status` are nominal and ordinal variables, respectively. R refers to case identifiers as `rownames` and categorical variables (nominal, ordinal) as `factors`. We'll cover each of these in the next section. You'll learn about dates in chapter 3.

2.2 Data structures

R has a wide variety of objects for holding data, including scalars, vectors, matrices, arrays, data frames, and lists. They differ in terms of the type of data they can hold, how they're created, their structural complexity, and the notation used to identify and access individual elements. Figure 2.1 shows a diagram of these data structures. Let's look at each structure in turn, starting with vectors.

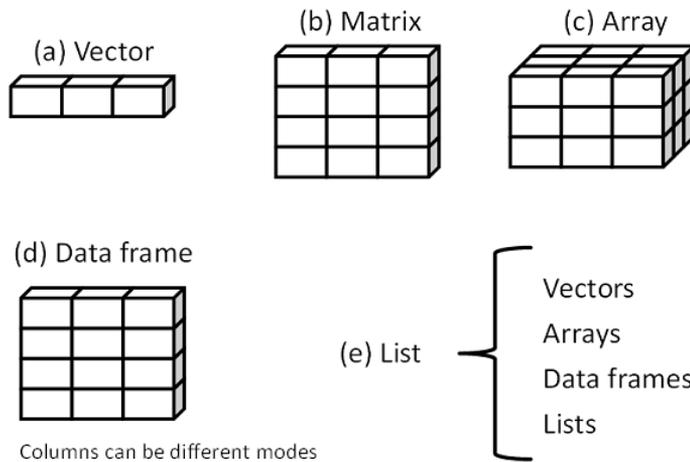


Figure 2.1 R data structures

Some definitions

Several terms are idiosyncratic to R and thus confusing to new users.

In R, an *object* is anything that can be assigned to a variable. This includes constants, data structures, functions, and even graphs. An object has a *mode* (which describes how the object is stored) and a *class* (which tells generic functions like `print` how to handle it).

A *data frame* is a structure in R that holds data and is similar to the datasets found in standard statistical packages (for example, SAS, SPSS, and Stata). The columns are variables, and the rows are observations. You can have variables of different types (for example, numeric or character) in the same data frame. Data frames are the main structures you use to store datasets.

Factors are nominal or ordinal variables. They're stored and treated specially in R. You'll learn about factors in section 2.2.5.

Most other terms used in R should be familiar to you and follow the terminology used in statistics and computing in general.

2.2.1 Vectors

Vectors are one-dimensional arrays that can hold numeric data, character data, or logical data. The combine function `c()` is used to form the vector. Here are examples of each type of vector:

```
a <- c(1, 2, 5, 3, 6, -2, 4)
b <- c("one", "two", "three")
c <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)
```

Here, `a` is a numeric vector, `b` is a character vector, and `c` is a logical vector. Note that the data in a vector must be only one type or mode (numeric, character, or logical). You can't mix modes in the same vector.

NOTE *Scalars* are one-element vectors. Examples include `f <- 3`, `g <- "US"`, and `h <- TRUE`. They're used to hold constants.

You can refer to elements of a vector using a numeric vector of positions within brackets. For example, `a[c(2, 4)]` refers to the second and fourth elements of vector `a`. Here are additional examples:

```
> a <- c("k", "j", "h", "a", "c", "m")
> a[3]
[1] "h"
> a[c(1, 3, 5)]
[1] "k" "h" "c"
> a[2:6]
[1] "j" "h" "a" "c" "m"
```

The colon operator used in the last statement generates a sequence of numbers. For example, `a <- c(2:6)` is equivalent to `a <- c(2, 3, 4, 5, 6)`.

2.2.2 Matrices

A *matrix* is a two-dimensional array in which each element has the same mode (numeric, character, or logical). Matrices are created with the `matrix` function. The general format is

```
myymatrix <- matrix(vector, nrow=number_of_rows, ncol=number_of_columns,
  byrow=logical_value, dimnames=list(
  char_vector_rownames, char_vector_colnames))
```

where `vector` contains the elements for the matrix, `nrow` and `ncol` specify the row and column dimensions, and `dimnames` contains optional row and column labels stored in character vectors. The option `byrow` indicates whether the matrix should be filled in by row (`byrow=TRUE`) or by column (`byrow=FALSE`). The default is by column. The following listing demonstrates the `matrix` function.

Listing 2.1 Creating matrices

```
> y <- matrix(1:20, nrow=5, ncol=4) #1
```

```

> y
  [,1] [,2] [,3] [,4]
[1,]  1  6 11 16
[2,]  2  7 12 17
[3,]  3  8 13 18
[4,]  4  9 14 19
[5,]  5 10 15 20
> cells <- c(1,26,24,68)
> rnames <- c("R1", "R2")
> cnames <- c("C1", "C2")           #2
> mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=TRUE,
  dimnames=list(rnames, cnames))
> mymatrix
  C1 C2
R1 1 26
R2 24 68
> mymatrix <- matrix(cells, nrow=2, ncol=2, byrow=FALSE,
  dimnames=list(rnames, cnames))
> mymatrix           #3
  C1 C2
R1 1 24
R2 26 68

```

#1 Creates a 5×4 matrix
#2 2×2 matrix filled by rows
#3 2×2 matrix filled by columns

First you create a 5×4 matrix #1. Then you create a 2×2 matrix with labels and fill the matrix by rows #2. Finally, you create a 2×2 matrix and fill the matrix by columns #3.

You can identify rows, columns, or elements of a matrix by using subscripts and brackets. $x[i,]$ refers to the i th row of matrix x , $x[,j]$ refers to the j th column, and $x[i, j]$ refers to the ij th element, respectively. The subscripts i and j can be numeric vectors in order to select multiple rows or columns, as shown in the following listing.

Listing 2.2 Using matrix subscripts

```

> x <- matrix(1:10, nrow=2)
> x
  [,1] [,2] [,3] [,4] [,5]
[1,]  1  3  5  7  9
[2,]  2  4  6  8 10
> x[2,]
 [1] 2 4 6 8 10
> x[,2]
 [1] 3 4
> x[1,4]
 [1] 7
> x[1, c(4,5)]
 [1] 7 9

```

First a 2×5 matrix is created containing the numbers 1 to 10. By default, the matrix is filled by column. Then the elements in the second row are selected, followed by the elements in the

second column. Next, the element in the first row and fourth column is selected. Finally, the elements in the first row and the fourth and fifth columns are selected.

Matrices are two-dimensional and, like vectors, can contain only one data type. When there are more than two dimensions, you use arrays (section 2.2.3). When there are multiple modes of data, you use data frames (section 2.2.4).

2.2.3 Arrays

Arrays are similar to matrices but can have more than two dimensions. They're created with an array function of the following form

```
myarray <- array(vector, dimensions, dimnames)
```

where `vector` contains the data for the array, `dimensions` is a numeric vector giving the maximal index for each dimension, and `dimnames` is an optional list of dimension labels. The following listing gives an example of creating a three-dimensional ($2 \times 3 \times 4$) array of numbers.

Listing 2.3 Creating an array

```
> dim1 <- c("A1", "A2")
> dim2 <- c("B1", "B2", "B3")
> dim3 <- c("C1", "C2", "C3", "C4")
> z <- array(1:24, c(2, 3, 4), dimnames=list(dim1, dim2, dim3))
> z
,, C1
  B1 B2 B3
A1 1 3 5
A2 2 4 6

,, C2
  B1 B2 B3
A1 7 9 11
A2 8 10 12

,, C3
  B1 B2 B3
A1 13 15 17
A2 14 16 18

,, C4
  B1 B2 B3
A1 19 21 23
A2 20 22 24
```

As you can see, arrays are a natural extension of matrices. They can be useful in creating functions that perform statistical calculations. Like matrices, they must be a single mode. Identifying elements follows what you've seen for matrices. In the previous example, the `z[1, 2, 3]` element is 15.

2.2.4 Data frames

A *data frame* is more general than a matrix in that different columns can contain different modes of data (numeric, character, and so on). It's similar to the dataset you'd typically see in SAS, SPSS, and Stata. Data frames are the most common data structure you'll deal with in R.

The patient dataset in table 2.1 consists of numeric and character data. Because there are multiple modes of data, you can't contain the data in a matrix. In this case, a data frame is the structure of choice.

A data frame is created with the `data.frame()` function

```
mydata <- data.frame(col1, col2, col3,...)
```

where `col1`, `col2`, `col3`, and so on are column vectors of any type (such as character, numeric, or logical). Names for each column can be provided with the `names` function. The following listing makes this clear.

Listing 2.4 Creating a data frame

```
> patientID <- c(1, 2, 3, 4)
> age <- c(25, 34, 28, 52)
> diabetes <- c("Type1", "Type2", "Type1", "Type1")
> status <- c("Poor", "Improved", "Excellent", "Poor")
> patientdata <- data.frame(patientID, age, diabetes, status)
> patientdata
  patientID age diabetes  status
1         1  25  Type1   Poor
2         2  34  Type2 Improved
3         3  28  Type1 Excellent
4         4  52  Type1   Poor
```

Each column must have only one mode, but you can put columns of different modes together to form the data frame. Because data frames are close to what analysts typically think of as datasets, we'll use the terms *columns* and *variables* interchangeably when discussing data frames.

There are several ways to identify the elements of a data frame. You can use the subscript notation you used before (for example, with matrices), or you can specify column names. Using the `patientdata` data frame created earlier, the following listing demonstrates these approaches.

Listing 2.5 Specifying elements of a data frame

```
> patientdata[1:2]
  patientID age
1         1  25
2         2  34
3         3  28
4         4  52
> patientdata[c("diabetes", "status")]
  diabetes  status
1  Type1   Poor
```

```

2 Type2 Improved
3 Type1 Excellent
4 Type1 Poor
> patientdata$age #1
[1] 25 34 28 52

```

#1 Indicates the age variable in the patient data frame

The `$` notation in the third example is new **#1**. It's used to indicate a particular variable from a given data frame. For example, if you want to cross-tabulate diabetes type by status, you can use the following code:

```

> table(patientdata$diabetes, patientdata$status)

      Excellent Improved Poor
Type1      1      0      2
Type2      0      1      0

```

It can get tiresome typing `patientdata$` at the beginning of every variable name, but shortcuts are available. For example, the `with()` function can to simplify your code.

USING WITH

Consider the following code.

```

summary(mtcars$mpg)
plot(mtcars$mpg, mtcars$disp)
plot(mtcars$mpg, mtcars$wt)

```

You can write this code more concisely as

```

with(mtcars, {
  summary(mpg)
  plot(mpg, disp)
  plot(mpg, wt)
})

```

The statements within the `{}` brackets are evaluated with reference to the `mtcars` data frame. If there's only one statement (for example, `summary(mpg)`), the `{}` brackets are optional.

The limitation of the `with()` function is that assignments exist only within the function brackets. Consider the following:

```

> with(mtcars, {
  stats <- summary(mpg)
  stats
})
  Min. 1st Qu.  Median   Mean 3rd Qu.  Max.
10.40 15.43 19.20 20.09 22.80 33.90
> stats
Error: object 'stats' not found

```

If you need to create objects that will exist outside of the `with()` construct, use the special assignment operator `<<-` instead of the standard one (`<-`). It saves the object to the global environment outside of the `with()` call. This can be demonstrated with the following code:

```
> with(mtcars, {
  nokeepstats <- summary(mpg)
  keepstats <<- summary(mpg)
})
> nokeepstats
Error: object 'nokeepstats' not found
> keepstats
  Min. 1st Qu.  Median    Mean 3rd Qu.   Max.
10.40  15.43  19.20  20.09  22.80  33.90
```

CASE IDENTIFIERS

In the patient data example, `patientID` is used to identify individuals in the dataset. In R, case identifiers can be specified with a `rowname` option in the data-frame function. For example, the statement

```
patientdata <- data.frame(patientID, age, diabetes,
  status, row.names=patientID)
```

specifies `patientID` as the variable to use in labeling cases on various printouts and graphs produced by R.

2.2.5 Factors

As you've seen, variables can be described as nominal, ordinal, or continuous. Nominal variables are categorical, without an implied order. `Diabetes (Type1, Type2)` is an example of a nominal variable. Even if `Type1` is coded as a 1 and `Type2` is coded as a 2 in the data, no order is implied. Ordinal variables imply order but not amount. `Status (poor, improved, excellent)` is a good example of an ordinal variable. You know that a patient with a poor status isn't doing as well as a patient with an improved status, but not by how much. Continuous variables can take on any value within some range, and both order and amount are implied. `Age` in years is a continuous variable and can take on values such as 14.5 or 22.8 and any value in between. You know that someone who is 15 is one year older than someone who is 14.

Categorical (nominal) and ordered categorical (ordinal) variables in R are called *factors*. Factors are crucial in R because they determine how data is analyzed and presented visually. You'll see examples of this throughout the book.

The function `factor()` stores the categorical values as a vector of integers in the range `[1... k]`, (where `k` is the number of unique values in the nominal variable) and an internal vector of character strings (the original values) mapped to these integers.

For example, assume that you have this vector:

```
diabetes <- c("Type1", "Type2", "Type1", "Type1")
```

The statement `diabetes <- factor(diabetes)` stores this vector as (1, 2, 1, 1) and associates it with 1 = Type1 and 2 = Type2 internally (the assignment is alphabetical). Any analyses performed on the vector `diabetes` will treat the variable as nominal and select the statistical methods appropriate for this level of measurement.

For vectors representing ordinal variables, you add the parameter `ordered=TRUE` to the `factor()` function. Given the vector

```
status <- c("Poor", "Improved", "Excellent", "Poor")
```

the statement `status <- factor(status, ordered=TRUE)` will encode the vector as (3, 2, 1, 3) and associate these values internally as 1 = Excellent, 2 = Improved, and 3 = Poor. Additionally, any analyses performed on this vector will treat the variable as ordinal and select the statistical methods appropriately.

By default, factor levels for character vectors are created in alphabetical order. This worked for the `status` factor, because the order "Excellent," "Improved," "Poor" made sense. There would have been a problem if "Poor" had been coded as "Ailing" instead, because the order would have been "Ailing," "Excellent," "Improved." A similar problem would exist if the desired order was "Poor," "Improved," "Excellent." For ordered factors, the alphabetical default is rarely sufficient.

You can override the default by specifying a `levels` option. For example,

```
status <- factor(status, order=TRUE,
  levels=c("Poor", "Improved", "Excellent"))
```

assigns the levels as 1 = Poor, 2 = Improved, 3 = Excellent. Be sure the specified levels match your actual data values. Any data values not in the list will be set to `missing`.

Numeric variables can be coded as factors using the `levels` and `labels` options. If `sex` was coded as 1 for male and 2 for female in the original data, then

```
sex <- factor(sex, levels=c(1, 2), labels=c("Male", "Female"))
```

would convert the variable to an unordered factor. Note that the order of the labels must match the order of the levels. In this example, `sex` would be treated as categorical, the labels "Male" and "Female" would appear in the output instead of 1 and 2, and any `sex` value that wasn't initially coded as a 1 or 2 would be set to `missing`.

The following listing demonstrates how specifying factors and ordered factors impacts data analyses.

Listing 2.6 Using factors

```
> patientID <- c(1, 2, 3, 4)           #1
> age <- c(25, 34, 28, 52)
> diabetes <- c("Type1", "Type2", "Type1", "Type1")
> status <- c("Poor", "Improved", "Excellent", "Poor")
> diabetes <- factor(diabetes)
> status <- factor(status, order=TRUE)
> patientdata <- data.frame(patientID, age, diabetes, status) #2
> str(patientdata)
```

```
'data.frame': 4 obs. of 4 variables:
 $ patientID: num 1 2 3 4
 $ age      : num 25 34 28 52
 $ diabetes : Factor w/ 2 levels "Type1","Type2": 1 2 1 1
 $ status   : Ord.factor w/ 3 levels "Excellent"<"Improved"<...: 3 2 1 3
> summary(patientdata)      #3
 patientID  age  diabetes  status
Min. :1.00 Min. :25.00 Type1:3 Excellent:1
1st Qu.:1.75 1st Qu.:27.25 Type2:1 Improved :1
Median :2.50 Median :31.00   Poor   :2
Mean   :2.50 Mean   :34.75
3rd Qu.:3.25 3rd Qu.:38.50
Max.   :4.00 Max.   :52.00
```

#1 Enter data as vectors.

#2 Displays the object structure

#3 Displays the object summary

First you enter the data as vectors #1. Then you specify that `diabetes` is a factor and `status` is an ordered factor. Finally, you combine the data into a data frame. The function `str(object)` provides information about an object in R (the data frame, in this case) #2. The output indicates that `diabetes` is a factor and `status` is an ordered factor, along with how they're coded internally. Note that the `summary()` function treats the variables differently #3. It provides the minimum, maximum, mean, and quartiles for the continuous variable `age`, and frequency counts for the categorical variables `diabetes` and `status`.

2.2.6 Lists

Lists are the most complex of the R data types. Basically, a *list* is an ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name. For example, a list may contain a combination of vectors, matrices, data frames, and even other lists. You create a list using the `list()` function

```
mylist <- list(object1, object2, ...)
```

where the objects are any of the structures seen so far. Optionally, you can name the objects in a list:

```
mylist <- list(name1=object1, name2=object2, ...)
```

The following listing shows an example.

Listing 2.7 Creating a list

```
> g <- "My First List"
> h <- c(25, 26, 18, 39)
> j <- matrix(1:10, nrow=5)
> k <- c("one", "two", "three")
> mylist <- list(title=g, ages=h, j, k) #A
> mylist
#B
$title
[1] "My First List"
```

```

$ages
[1] 25 26 18 39

[[3]]
  [,1] [,2]
[1,]  1  6
[2,]  2  7
[3,]  3  8
[4,]  4  9
[5,]  5 10

[[4]]
[1] "one" "two" "three"

> mylist[[2]]           #C
[1] 25 26 18 39
> mylist[["ages"]]
[[1] 25 26 18 39

```

#A Creates a list

#B Prints the entire list

#C Prints the second component

In this example, you create a list with four components: a string, a numeric vector, a matrix, and a character vector. You can combine any number of objects and save them as a list.

You can also specify elements of the list by indicating a component number or a name within double brackets. In this example, `mylist[[2]]` and `mylist[["ages"]]` both refer to the same four-element numeric vector. For named components, `mylist$ages` would also work. Lists are important R structures for two reasons. First, they allow you to organize and recall disparate information in a simple way. Second, the results of many R functions return lists. It's up to the analyst to pull out the components that are needed. You'll see numerous examples of functions that return lists in later chapters.

2.2.7 Tibbles

Before moving on, it is worth mentioning tibbles. Tibbles are data frames that have specialized behaviors that are designed to make them more useful. They're created using either the `tibble()` or `as_tibble()` function from the `tibble` package. To install the `tibble` package, use `install.packages("tibble")`. Some of their attractive features are described below.

Tibbles print in a more compact format than standard data frames. Additionally, variable labels describe the data type of each column.

```

library(tibble)
mtcars <- as_tibble(mtcars)
mtcars

```

```

# A tibble: 32 x 11
  mpg  cyl disp  hp drat   wt  qsec    vs  am gear carb

```

```
* <dbl> <dbl>
1 21 6 160 110 3.9 2.62 16.5 0 1 4 4
2 21 6 160 110 3.9 2.88 17.0 0 1 4 4
3 22.8 4 108 93 3.85 2.32 18.6 1 1 4 1
4 21.4 6 258 110 3.08 3.22 19.4 1 0 3 1
5 18.7 8 360 175 3.15 3.44 17.0 0 0 3 2
6 18.1 6 225 105 2.76 3.46 20.2 1 0 3 1
7 14.3 8 360 245 3.21 3.57 15.8 0 0 3 4
8 24.4 4 147. 62 3.69 3.19 20 1 0 4 2
9 22.8 4 141. 95 3.92 3.15 22.9 1 0 4 2
10 19.2 6 168. 123 3.92 3.44 18.3 1 0 4 4
# ... with 22 more rows
```

Tibbles never convert character variables to factors. Base R functions such as `read.table()`, `data.frame()` and `as.data.frame()` convert character data to factors by default. You would have to add the option `stringsAsFactors = FALSE` to these functions to suppress this behavior.

Tibbles never change the names of variables. If the dataset being imported has a variable called "Last Address", base R functions would convert the name to "Last.Address", since R variable names don't use spaces. Tibbles would keep the name as is and use back ticks (e.g., ``Last Address``) to make the variable name syntactically correct.

Subsetting a tibble always returns a tibble. For example, subsetting the `mtcars` data frame using `mtcars[, "mpg"]`, would return a vector, rather than a one column data frame. R automatically simplifies the results. To get a one column data frame, you would have to include the `drop = FALSE` option (`mtcars[, "mpg", drop = FALSE]`). In contrast, if `mtcars` is a tibble, then `mtcars[, "mpg"]` would return a one column tibble. The results are not simplified, allowing you to easily predict what the results of a subsetting operation will return.

Finally, tibbles don't support row names. The function `rownames_to_column()` can be used to convert the row names in a data frame to a variable in a tibble.

Tibbles are important because many popular packages, such as `readr`, `tidyr`, `dplyr`, and `purrr` save data frames as tibbles. Although tibbles have been designed to be "a modern take on data frames", note that they can be used interchangeably with data frames. Any function that requires a data frame can take a tibble and vice versa. To learn more, see <https://r4ds.had.co.nz/tibbles.html>.

A note for programmers

Experienced programmers typically find several aspects of the R language unusual. Here are some features of the language you should be aware of:

- The period (`.`) has no special significance in object names. The dollar sign (`$`) has a somewhat analogous meaning to the period in other object oriented languages, and can be used to identify the parts of a data frame or list. For example, `A$x` refers to variable `x` in data frame `A`.
- R doesn't provide multiline or block comments. You must start each line of a multiline comment with `#`. For debugging purposes, you can also surround code that you want the interpreter to ignore with the statement `if (FALSE) { ... }`. Changing the `FALSE` to `TRUE` allows the code to be executed.

- Assigning a value to a nonexistent element of a vector, matrix, array, or list expands that structure to accommodate the new value. For example, consider the following:

```
> x <- c(8, 6, 4)
> x[7] <- 10
> x
[1] 8 6 4 NA NA NA 10
```

The vector `x` has expanded from three to seven elements through the assignment. `x <- x[1:3]` would shrink it back to three elements.

- R doesn't have scalar values. Scalars are represented as one-element vectors.
- Indices in R start at 1, not at 0. In the vector earlier, `x[1]` is 8.
- Variables can't be declared. They come into existence on first assignment.

To learn more, see John Cook's excellent blog post, "R Language for Programmers" (<http://mng.bz/6NwQ>).

Programmers looking for stylistic guidance may also want to check out "Google's R Style Guide" (<http://mng.bz/i775>).

2.3 Data input

Now that you have data structures, you need to put some data in them! As a data analyst, you're typically faced with data that comes from a variety of sources and in a variety of formats. Your task is to import the data into your tools, analyze the data, and report on the results. R provides a wide range of tools for importing data. The definitive guide for importing data in R is the *R Data Import/Export* manual available at <http://mng.bz/urwn>.

As you can see in figure 2.2, R can import data from the keyboard, from text files, from Microsoft Excel and Access, from popular statistical packages, from a variety of relational database management systems, from specialty databases, and from web sites and online services. Because you never know where your data will come from, we'll cover each of them here. You only need to read about the ones you're going to be using.

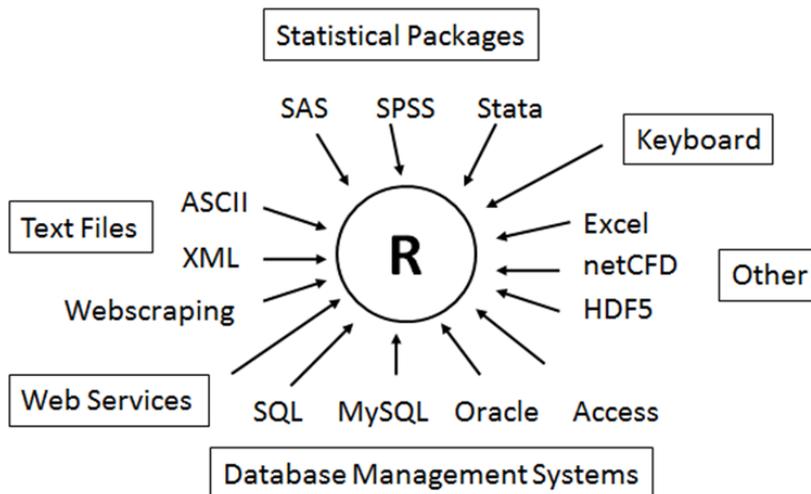


Figure 2.2 Sources of data that can be imported into R

2.3.1 Entering data from the keyboard

Perhaps the simplest way to enter data is from the keyboard. There are two common methods: entering data through R's built-in text editor and embedding data directly into your code. We'll consider the editor first.

The `edit()` function in R invokes a text editor that lets you enter data manually. Here are the steps:

1. Create an empty data frame (or matrix) with the variable names and modes you want to have in the final dataset.
2. Invoke the text editor on this data object, enter your data, and save the results to the data object.

The following example creates a data frame named `mydata` with three variables: `age` (numeric), `gender` (character), and `weight` (numeric). You then invoke the text editor, add your data, and save the results:

```
mydata <- data.frame(age=numeric(0),
  gender=character(0), weight=numeric(0))
mydata <- edit(mydata)
```

Assignments like `age=numeric(0)` create a variable of a specific mode, but without actual data. Note that the result of the editing is assigned back to the object itself. The `edit()` function operates on a copy of the object. If you don't assign it a destination, all of your edits will be lost!

The results of invoking the `edit()` function on a Windows platform are shown in figure 2.3. In this figure, I've added some data. If you click a column title, the editor gives you the option

of changing the variable name and type (numeric or character). You can add variables by clicking the titles of unused columns. When the text editor is closed, the results are saved to the object assigned (`mydata`, in this case). Invoking `mydata <- edit(mydata)` again allows you to edit the data you've entered and to add new data. A shortcut for `mydata <- edit(mydata)` is `fix(mydata)`.

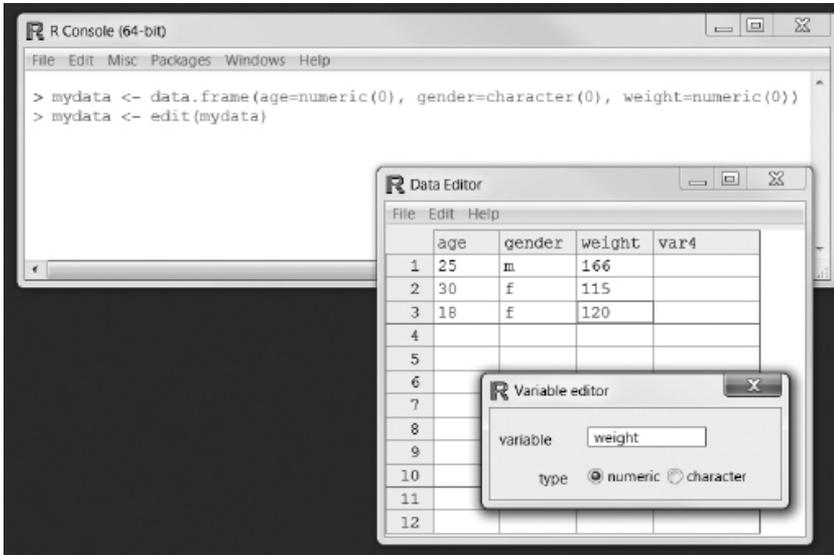


Figure 2.3 Entering data via the built-in editor on a Windows platform

Alternatively, you can embed the data directly in your program. For example, the code

```
mydatatxt <- "
age gender weight
25 m 166
30 f 115
18 f 120
"
mydata <- read.table(header=TRUE, text=mydatatxt)
```

creates the same data frame as that created with the `edit()` function. A character string is created containing the raw data, and the `read.table()` function is used to process the string and return a data frame. The `read.table()` function is described more fully in the next section.

Keyboard data entry can be convenient when you're working with small datasets. For larger datasets, you'll want to use the methods described next: importing data from existing text files, Excel spreadsheets, statistical packages, or database-management systems.

2.3.2 Importing data from a delimited text file

You can import data from delimited text files using `read.table()`, a function that reads a file in table format and saves it as a data frame. Each row of the table appears as one line in the file. The syntax is

```
mydataframe <- read.table(file, options)
```

where *file* is a delimited ASCII file and the *options* are parameters controlling how data is processed. The most common options are listed in table 2.2.

Table 2.2 `read.table()` options

Option	Description
header	A logical value indicating whether the file contains the variable names in the first line.
sep	The delimiter separating data values. The default is <code>sep=""</code> , which denotes one or more spaces, tabs, new lines, or carriage returns. Use <code>sep=","</code> to read comma-delimited files, and <code>sep="\t"</code> to read tab-delimited files.
row.names	An optional parameter specifying one or more variables to represent row identifiers.
col.names	If the first row of the data file doesn't contain variable names (<code>header=FALSE</code>), you can use <code>col.names</code> to specify a character vector containing the variable names. If <code>header=FALSE</code> and the <code>col.names</code> option is omitted, variables will be named V1, V2, and so on.
na.strings	An optional character vector indicating missing-values codes. For example, <code>na.strings=c("-9", "?")</code> converts each -9 and ? value to NA as the data is read.
colClasses	An optional vector of classes to be assign to the columns. For example, <code>colClasses=c("numeric", "numeric", "character", "NULL", "numeric")</code> reads the first two columns as numeric, reads the third column as character, skips the fourth column, and reads the fifth column as numeric. If there are more than five columns in the data, the values in <code>colClasses</code> are recycled. When you're reading large text files, including the <code>colClasses</code> option can speed up processing considerably.
quote	Character(s) used to delimit strings that contain special characters. By default this is either double (") or single (') quotes.
skip	The number of lines in the data file to skip before beginning to read the data. This option is useful for skipping header comments in the file.
stringsAsFactors	A logical value indicating whether character variables should be converted to factors. The default is TRUE unless this is overridden by <code>colClasses</code> . When you're processing large text files, setting <code>stringsAsFactors=FALSE</code> can speed up processing.
text	A character string specifying a text string to process. If <code>text</code> is specified, leave <code>file</code> blank. An example is given in section 2.3.1.

Consider a text file named `studentgrades.csv` containing students' grades in math, science, and social studies. Each line of the file represents a student. The first line contains the variable names, separated with commas. Each subsequent line contains a student's information, also separated with commas. The first few lines of the file are as follows:

```
StudentID,First,Last,Math,Science,Social Studies
011,Bob,Smith,90,80,67
012,Jane,Weary,75,,80
010,Dan,"Thornton, III",65,75,70
040,Mary,"O'Leary",90,95,92
```

The file can be imported into a data frame using the following code:

```
grades <- read.table("studentgrades.csv", header=TRUE,
  row.names="StudentID", sep=",")
```

The results are as follows:

```
> grades

  First      Last Math Science Social.Studies
11 Bob      Smith  90    80         67
12 Jane     Weary  75    NA          80
10 Dan  Thornton, III 65    75          70
40 Mary    O'Leary  90    95          92

> str(grades)

'data.frame':  4 obs. of  5 variables:
 $ First      : chr  "Bob" "Jane" "Dan" "Mary"
 $ Last      : chr  "Smith" "Weary" "Thornton, III" "O'Leary"
 $ Math      : int  90 75 65 90
 $ Science   : int  80 NA 75 95
 $ Social.Studies: int  67 80 70 92
```

There are several interesting things to note about how the data is imported. The variable name `Social Studies` is automatically renamed to follow R conventions. The `StudentID` column is now the row name, no longer has a label, and has lost its leading zero. The missing science grade for Jane is correctly read as missing. I had to put quotation marks around Dan's last name in order to escape the comma between `Thornton` and `III`. Otherwise, R would have seen seven values on that line, rather than six. I also had to put quotation marks around `O'Leary`. Otherwise, R would have read the single quote as a string delimiter (which isn't what I want). Finally, the first and last names are converted to factors.

By default, `read.table()` converts character variables to factors, which may not always be desirable. For example, there would be little reason to convert a character variable containing a respondent's comments into a factor. Additionally, you may want to manipulate or mine the text in a variable, and this is hard to do once it has been converted to a factor. You can suppress this behavior in a number of ways. Including the option `stringsAsFactors=FALSE` turns off this behavior for all character variables. Alternatively, you can use the `colClasses` option to specify a class (for example, logical, numeric, character, or factor) for each column.

Importing the same data with

```
grades <- read.table("studentgrades.csv", header=TRUE,
  row.names="StudentID", sep=";",
  colClasses=c("character", "character", "character",
    "numeric", "numeric", "numeric"))
```

produces the following data frame:

```
> grades

  First      Last Math Science Social.Studies
011 Bob      Smith 90   80      67
012 Jane     Weary 75   NA      80
010 Dan  Thornton, III 65  75      70
040 Mary    O'Leary 90   95      92

> str(grades)

'data.frame': 4 obs. of 5 variables:
 $ First      : chr "Bob" "Jane" "Dan" "Mary"
 $ Last       : chr "Smith" "Weary" "Thornton, III" "O'Leary"
 $ Math       : num 90 75 65 90
 $ Science    : num 80 NA 75 95
 $ Social.Studies: num 67 80 70 92
```

Note that the row names retain their leading zero and `First` and `Last` are no longer factors. Additionally, the grades are stored as real values rather than integers.

The `read.table()` function has many options for fine-tuning data imports. See `help(read.table)` for details.

Importing data via connections

Many of the examples in this chapter import data from files that exist on your computer. R provides several mechanisms for accessing data via connections as well. For example, the functions `file()`, `gzfile()`, `bzfile()`, `xzfile()`, `unz()`, and `url()` can be used in place of the filename. The `file()` function allows you to access files, the clipboard, and C-level standard input. The `gzfile()`, `bzfile()`, `xzfile()`, and `unz()` functions let you read compressed files.

The `url()` function lets you access internet files through a complete URL that includes `http://`, `ftp://`, or `file://`. For HTTP and FTP, proxies can be specified. For convenience, complete URLs (surrounded by double quotation marks) can usually be used directly in place of filenames as well. See `help(file)` for details.

Base R also provides the functions `read.csv()` and `read.delim()` for importing rectangular text files. These are simply wrapper functions that call `read.table()` with specific defaults. For example `read.csv()` calls `read.table()` with `header=TRUE`, and `sep=","` while `read.delim()` calls `read.table()` with `header=TRUE` and `sep="\t"`. Details are provided in the `read.table()` help.

The `readr` package provides a powerful alternative to base R functions for reading rectangular text files. The primary function is `read_delim()` with helper functions `read_csv()` and `read_tsv()` for reading comma delimited and tab delimited files respectively. After installing the package, the previous data could have been read using the code

```
library(readr)
grades <- read_csv("studentgrades.csv")
```

The package also provides for importing fixed width files (where data appears in specific columns), tabular files (where columns are separated by white-space), and web log files.

Functions in the `readr` package provide a number of advantages over those in base R. First and foremost, they are *significantly* faster. This can be a tremendous advantage when reading large data files. Additionally, they are very good at guessing the correct data type of each column (numeric character, date, and date-time). Finally, unlike base R functions, they don't convert character data to factors by default. Functions in the `readr` package return data are returned as tibbles (data frames with some specialized features). To learn more, see <https://readr.tidyverse.org>.

2.3.3 Importing data from Excel

The best way to read an Excel file is to export it to a comma-delimited file from Excel and import it into R using the method described earlier. Alternatively, you can import Excel worksheets directly using the `readxl` package. Be sure to download and install it before you first use it.

The `readxl` package can be used to read both `.xls` and `.xlsx` versions of Excel files. The `read_excel()` function imports a worksheet into a data frame (as a tibble). The simplest format is `read_excel(file, n)` where `file` is the path to an Excel workbook, `n` is the number of the worksheet to be imported, and the first line of the worksheet contains the variable names. For example, on a Windows platform, the code

```
library(readxl)
workbook <- "c:/myworkbook.xlsx"
mydataframe <- read_xlsx(workbook, 1)
```

imports the first worksheet from the workbook `myworkbook.xlsx` stored on the C: drive and saves it as the data frame `mydataframe`.

The `read_excel()` function has options that allow you to specify a specific cell range (e.g. `range = "Mysheet!B2:G14"`, along with the class of each column (`col_types`). See `help(read_excel)` for details.

There are other packages that can help you work with Excel files. They include `xlsx`, `XLConnect` and `openxlsx`. The `xlsx` and `XLConnect` packages depend on Java, while `openxlsx` doesn't. Unlike `readxl`, these packages can do more than import worksheets—they can create and manipulate Excel files as well. Programmers who need to develop an interface between R and Excel should check out one or more of these packages.

2.3.4 Importing data from XML

Increasingly, data is provided in the form of files encoded in XML. R has several packages for handling XML files. For example, the `XML` package written by Duncan Temple Lang allows you to read, write, and manipulate XML files. Coverage of XML is beyond the scope of this text; if you're interested in accessing XML documents from within R, see the excellent package documentation at www.omegahat.org/RXML.

2.3.5 Importing data from the Web

Data can be obtained from the web via *webscraping* or the use of *application programming interfaces (APIs)*. *Webscraping* is used to extract the information embedded in specific web pages, whereas APIs allow you to interact with web services and online data stores.

Typically, webscraping is used to extract data from a web page and save it into an R structure for further analysis. For example, the text on a web page can be downloaded into an R character vector using the `readLines()` function and manipulated with functions such as `grep()` and `gsub()`. The `rvest` package provides functions that can simplify extracting data from web pages, and was inspired by the Python library *Beautiful Soup*. The `RCurl` and `XML` packages can also be used to extract the information desired. For more information, including examples, see "Examples of Web Scraping with R" available from the website *ProgrammingR* (<http://www.programmingr.com>).

APIs specify how software components should interact with each other. A number of R packages use this approach to extract data from web-accessible resources. These include data sources in biology, medicine, Earth sciences, physical science, economics and business, finance, literature, marketing, news, and sports.

For example, if you're interested in social media, you can access Twitter data via `twitterR`, Facebook data via `Rfacebook`, and Flickr data via `Rflickr`. Other packages allow you to access popular web services provide by Google, Amazon, Dropbox, Salesforce, and others. For a comprehensive list of R packages that can help you access web-based resources, see the CRAN Task view on *Web Technologies and Services* (<http://mng.bz/370r>).

2.3.6 Importing data from SPSS

IBM SPSS datasets can be imported into R via the `read_spss()` function in the `haven` package. First, download and install the package:

```
install.packages("haven")
```

Then use the following code to import the data:

```
library(haven)
mydataframe <- read_spss("mydata.sav")
```

The imported dataset is a data frame (as a tibble) and variables containing imported SPSS value labels are assigned the class `labelled`. You can convert these labelled variables to R factors using the following code:

```
labelled_vars <- names(mydataframe)[sapply(mydataframe, is.labelled)]
for (vars in labelled_vars){
  mydataframe[[vars]] = as_factor(mydataframe[[vars]])
}
```

The `haven` package has additional functions for reading SPSS files in compressed (`.zsav`) or transport (`.por`) format.

2.3.7 Importing data from SAS

SAS datasets can be imported using the `read_sas()` in the `haven` package. After installing the package, import that data using

```
library(haven)
mydataframe <- read_sas("mydata.sas7bdat")
```

If the user also has a catalogue of variable formats, they can be imported and applied to the data as well using

```
mydataframe <- read_sas("mydata.sas7bdat",
  catalog_file = "mydata.sas7bcat")
```

In either case, the result is a data frame saved as a tibble.

Alternatively, there is a commercial product named Stat/Transfer (described in section 2.3.10) that does an excellent job of saving SAS datasets (including any existing variable formats) as R data frames.

2.3.8 Importing data from Stata

Importing data from Stata to R is straightforward. Again, using the `haven` package

```
library(haven)
mydataframe <- read_dta("mydata.dta")
```

Here, `mydata.dta` is the Stata dataset, and `mydataframe` is the resulting R data frame, saved as a tibble.

2.3.9 Accessing database management systems (DBMSs)

R can interface with a wide variety of relational database management systems (DBMSs), including Microsoft SQL Server, Microsoft Access, MySQL, Oracle, PostgreSQL, DB2, Sybase, Teradata, and SQLite. Some packages provide access through native database drivers, whereas others offer access via ODBC or JDBC. Using R to access data stored in external DBMSs can be an efficient way to analyze large datasets (see appendix F) and takes advantage of the power of both SQL and R.

THE ODBC INTERFACE

Perhaps the most popular method of accessing a DBMS in R is through the `RODBC` package, which allows R to connect to any DBMS that has an ODBC driver. This includes all the DBMSs listed earlier.

The first step is to install and configure the appropriate ODBC driver for your platform and database (these drivers aren't part of R). If the requisite drivers aren't already installed on your machine, an internet search should provide you with options (*Setting up ODBC Drivers* at <https://db.rstudio.com/best-practices/drivers/> is a good place to start).

Once the drivers are installed and configured for the database(s) of your choice, install the `RODBC` package. You can do so by using the `install.packages("RODBC")` command. The primary functions included with `RODBC` are listed in table 2.3.

Table 2.3 `RODBC` functions

Function	Description
<code>odbcConnect(dsn,uid="",pwd="")</code>	Opens a connection to an ODBC database
<code>sqlFetch(channel,sqltable)</code>	Reads a table from an ODBC database into a data frame
<code>sqlQuery(channel,query)</code>	Submits a query to an ODBC database and returns the results
<code>sqlSave(channel,mydf,tablename = sqltable,append=FALSE)</code>	Writes or updates (<code>append=TRUE</code>) a data frame to a table in the ODBC database
<code>sqlDrop(channel,sqltable)</code>	Removes a table from the ODBC database
<code>close(channel)</code>	Closes the connection

The `RODBC` package allows two-way communication between R and an ODBC-connected SQL database. This means you can not only read data from a connected database into R, but also use R to alter the contents of the database itself. Assume that you want to import two tables (Crime and Punishment) from a DBMS into two R data frames called `crimedat` and `pundat`, respectively. You can accomplish this with code similar to the following:

```
library(RODBC)
myconn <- odbcConnect("mysdn", uid="Rob", pwd="aardvark")
crimedat <- sqlFetch(myconn, Crime)
pundat <- sqlQuery(myconn, "select * from Punishment")
close(myconn)
```

Here, you load the `RODBC` package and open a connection to the ODBC database through a registered data source name (`mysdn`) with a security UID (`rob`) and password (`aardvark`). The connection string is passed to `sqlFetch`, which copies the table `Crime` into the R data frame `crimedat`. You then run the SQL `select` statement against the table `Punishment` and save the results to the data frame `pundat`. Finally, you close the connection.

The `sqlQuery()` function is powerful because any valid SQL statement can be inserted. This flexibility allows you to select specific variables, subset the data, create new variables, and recode and rename existing variables.

DBI-RELATED PACKAGES

The `DBI` package provides a general and consistent client-side interface to DBMS. Building on this framework, the `RJDBC` package provides access to DBMS via a JDBC driver. Be sure to install the necessary JDBC drivers for your platform and database. Other useful DBI-based packages include `RMySQL`, `ROracle`, `RPostgreSQL`, and `RSQLite`. These packages provide native database drivers for their respective databases but may not be available on all platforms. Check the documentation on CRAN (<http://cran.r-project.org>) for details.

2.3.10 Importing data via Stat/Transfer

Before we end our discussion of importing data, it's worth mentioning a commercial product that can make the task significantly easier. Stat/Transfer (www.stattransfer.com) is a standalone application that can transfer data among 34 data formats, including R (see figure 2.4).

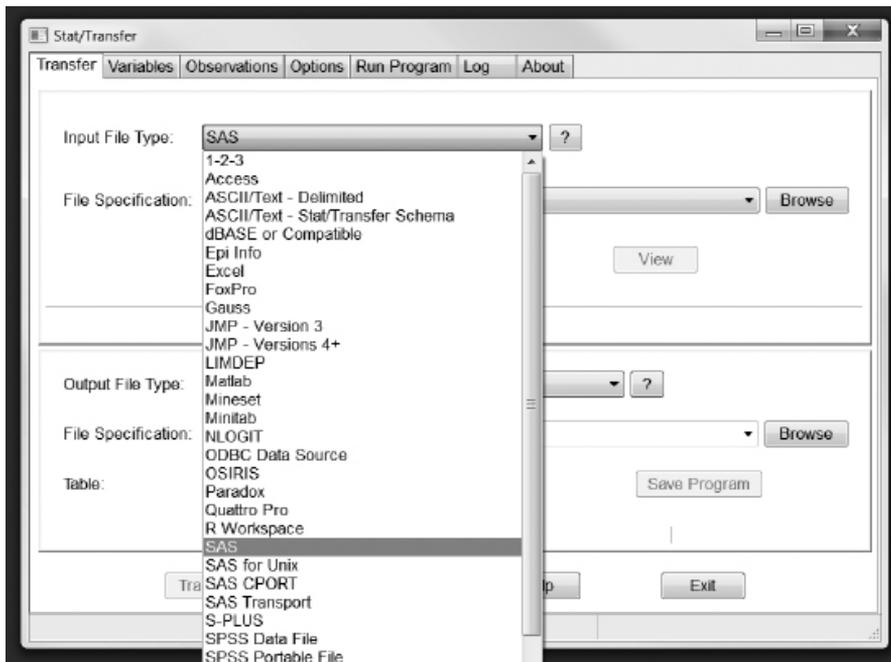


Figure 2.4 Stat/Transfer's main dialog on Windows

Stat/Transfer is available for Windows, Mac, and Unix platforms. It supports the latest versions of the statistical packages we've discussed so far, as well as ODBC-accessed DBMSs such as Oracle, Sybase, Informix, and DB/2.

2.4 Annotating datasets

Data analysts typically annotate datasets to make the results easier to interpret. Annotating generally includes adding descriptive labels to variable names and value labels to the codes used for categorical variables. For example, for the variable `age`, you might want to attach the more descriptive label "Age at hospitalization (in years)." For the variable `gender`, coded 1 or 2, you might want to associate the labels "male" and "female."

2.4.1 Variable labels

Unfortunately, R's ability to handle variable labels is limited. One approach is to use the variable label as the variable's name and then refer to the variable by its position index. Consider the earlier example, where you have a data frame containing patient data. The second column, `age`, contains the ages at which individuals were first hospitalized. The code

```
names(patientdata)[2] <- "Age at hospitalization (in years)"
```

renames `age` to "Age at hospitalization (in years)". Clearly this new name is too long to type repeatedly. Instead, you can refer to this variable as `patientdata[2]`, and the string "Age at hospitalization (in years)" will print wherever `age` would have originally. Obviously, this isn't an ideal approach, and you may be better off trying to come up with better variable names (for example, `admissionAge`).

2.4.2 Value labels

The `factor()` function can be used to create value labels for categorical variables. Continuing the example, suppose you have a variable named `gender`, which is coded 1 for male and 2 for female. You can create value labels with the code

```
patientdata$gender <- factor(patientdata$gender,
                             levels = c(1,2),
                             labels = c("male", "female"))
```

Here `levels` indicates the actual values of the variable, and `labels` refers to a character vector containing the desired labels.

2.5 Useful functions for working with data objects

We'll end this chapter with a brief summary of useful functions for working with data objects (see table 2.4).

Table 2.4 Useful functions for working with data objects

Function	Purpose
<code>length(object)</code>	Gives the number of elements/components.
<code>dim(object)</code>	Gives the dimensions of an object.
<code>str(object)</code>	Gives the structure of an object.
<code>class(object)</code>	Gives the class of an object.
<code>mode(object)</code>	Determines how an object is stored.
<code>names(object)</code>	Gives the names of components in an object.
<code>c(object, object,...)</code>	Combines objects into a vector.
<code>cbind(object, object, ...)</code>	Combines objects as columns.
<code>rbind(object, object, ...)</code>	Combines objects as rows.
<code>object</code>	Prints an object.
<code>head(object)</code>	Lists the first part of an object.
<code>tail(object)</code>	Lists the last part of an object.
<code>ls()</code>	Lists current objects.
<code>rm(object, object, ...)</code>	Deletes one or more objects. The statement <code>rm(list = ls())</code> removes most objects from the working environment.
<code>newobject <- edit(object)</code>	Edits object and saves it as newobject.
<code>fix(object)</code>	Edits an object in place.

We've already discussed most of these functions. `head()` and `tail()` are useful for quickly scanning large datasets. For example, `head(patientdata)` lists the first six rows of the data frame, whereas `tail(patientdata)` lists the last six. We'll cover functions such as `length()`, `cbind()`, and `rbind()` in the next chapter; they're gathered here as a reference.

2.6 Summary

- R provides a wide range of structures for holding data. These include vectors, matrices, arrays, data frames, and lists.
- The ability to specify elements of these structures via single bracket notation `[]` for vectors, matrices and data frames, and double bracket notation `[[]]` for lists, is particularly important for selecting, subsetting, and transforming data.
- R can import data from flat files, web files, statistical packages, spreadsheets, and databases. The imported data is usually stored in data frames or tibbles. Exporting data is covered in appendix C, and methods of working with large datasets (in the gigabyte to terabyte range) are covered in appendix F.

3

Basic data management

This chapter covers

- **Manipulating dates and missing values**
- **Understanding data type conversions**
- **Creating and recoding variables**
- **Sorting, merging, and subsetting datasets**
- **Selecting and dropping variables**

In chapter 2, we covered a variety of methods for importing data into R. Unfortunately, getting your data in the rectangular arrangement of a matrix or data frame is only the first step in preparing it for analysis. To paraphrase Captain Kirk in the *Star Trek* episode “A Taste of Armageddon” (and proving my geekiness once and for all), “Data is a messy business—a very, very messy business.” In my own work, as much as 60% of the time I spend on data analysis is focused on preparing the data for analysis. I’ll go out on a limb and say that the same is probably true in one form or another for most real-world data analysts. Let’s take a look at an example.

3.1 A working example

One of the topics that I study in my current job is how men and women differ in the ways they lead their organizations. Typical questions might be

- Do men and women in management positions differ in the degree to which they defer to superiors?
- Does this vary from country to country, or are these gender differences universal?

One way to address these questions is to have bosses in multiple countries rate their managers on deferential behavior, using questions like the following:

This manager asks my opinion before making personnel decisions.

1	2	3	4	5
strongly disagree	disagree	neither agree nor disagree	agree	strongly agree

The resulting data might resemble that in table 3.1. Each row represents the ratings given to a manager by his or her boss.

Table 3.1 Gender differences in leadership behavior

Manager	Date	Country	Gender	Age	q1	q2	q3	q4	q5
1	10/24/14	US	M	32	5	4	5	5	5
2	10/28/14	US	F	45	3	5	2	5	5
3	10/01/14	UK	F	25	3	5	5	5	2
4	10/12/14	UK	M	39	3	3	4		
5	05/01/14	UK	F	99	2	2	1	2	1

Here, each manager is rated by their boss on five statements (q1 to q5) related to deference to authority. For example, manager 1 is a 32-year-old male working in the US and is rated deferential by his boss, whereas manager 5 is a female of unknown age (99 probably indicates that the information is missing) working in the UK and is rated low on deferential behavior. The Date column captures when the ratings were made.

Although a dataset might have dozens of variables and thousands of observations, we've included only 10 columns and 5 rows to simplify the examples. Additionally, we've limited the number of items pertaining to the managers' deferential behavior to five. In a real-world study, you'd probably use 10–20 such items to improve the reliability and validity of the results. You can create a data frame containing the data in table 3.1 using the following code.

Listing 3.1 Creating the leadership data frame

```
manager <- c(1, 2, 3, 4, 5)
date <- c("10/24/08", "10/28/08", "10/1/08", "10/12/08", "5/1/09")
country <- c("US", "US", "UK", "UK", "UK")
gender <- c("M", "F", "F", "M", "F")
age <- c(32, 45, 25, 39, 99)
q1 <- c(5, 3, 3, 3, 2)
q2 <- c(4, 5, 5, 3, 2)
q3 <- c(5, 2, 5, 4, 1)
q4 <- c(5, 5, 5, NA, 2)
q5 <- c(5, 5, 2, NA, 1)
leadership <- data.frame(manager, date, country, gender, age,
  q1, q2, q3, q4, q5, stringsAsFactors=FALSE)
```

In order to address the questions of interest, you must first deal with several data-management issues. Here's a partial list:

- The five ratings (q1 to q5) need to be combined, yielding a single mean deferential score from each manager.
- In surveys, respondents often skip questions. For example, the boss rating manager 4 skipped questions 4 and 5. You need a method of handling incomplete data. You also need to recode values like 99 for age to *missing*.
- There may be hundreds of variables in a dataset, but you may only be interested in a few. To simplify matters, you'll want to create a new dataset with only the variables of interest.
- Past research suggests that leadership behavior may change as a function of the manager's age. To examine this, you may want to recode the current values of age into a new categorical age grouping (for example, young, middle-aged, elder).
- Leadership behavior may change over time. You might want to focus on deferential behavior during the recent global financial crisis. To do so, you may want to limit the study to data gathered during a specific period of time (say, January 1, 2009 to December 31, 2009).

We'll work through each of these issues in this chapter, as well as other basic data-management tasks such as combining and sorting datasets. Then, in chapter 4, we'll look at some advanced topics.

3.2 Creating new variables

In a typical research project, you'll need to create new variables and transform existing ones. This is accomplished with statements of the form

```
variable <- expression
```

A wide array of operators and functions can be included in the *expression* portion of the statement. Table 3.2 lists R's arithmetic operators. Arithmetic operators are used when developing formulas.

Table 3.2 Arithmetic operators (*continued*)

Operator	Description
+	Addition.
-	Subtraction.
*	Multiplication.
/	Division.
^ or **	Exponentiation.

<code>x%%y</code>	Modulus (x mod y): for example, 5%%2 is 1.
<code>x%/y</code>	Integer division: for example, 5%/2 is 2.

Given the data frame `leadership`, say you want to create a new variable `total_score` that adds the variables `q1` to `q5`, and a new variable called `mean_score` that averages these variables. If you use the code

```
total_score <- q1 + q2 + q3 + q4 + q5
mean_score <- (q1 + q2 + q3 + q4 + q5)/5
```

you'll get an error, because R doesn't know that `q1`, `q2`, `q3`, `q4` and `q5` are from the data frame `leadership`. If you use this code instead

```
total_score <- leadership$q1 + leadership$q2 + leadership$q3 +
  leadership$q4 + leadership$q5
mean_score <- (leadership$q1 + leadership$q2 + leadership$q3 +
  leadership$q4 + leadership$q5)/5
```

the statements will succeed but you'll end up with a data frame (`leadership`) and two separate vectors (`total_score` and `mean_score`). This probably isn't the result you want. Ultimately, you want to incorporate new variables into the original data frame. The following listing provides two separate ways to accomplish this goal. The one you choose is up to you; the results will be the same.

Listing 3.2 Creating new variables

```
leadership$total_score <- leadership$q1 + leadership$q2 + leadership$q3 +
  leadership$q4 + leadership$q5
leadership$mean_score <- (leadership$q1 + leadership$q2 + leadership$q3 +
  leadership$q4 + leadership$q5)/5

leadership <- transform(leadership,
  total_score = q1 + q2 + q3 + q4 + q5,
  mean_score = (q1 + q2 + q3 + q4 + q5)/5)
```

Personally, I prefer the second method, exemplified by the use of the `transform()` function. It simplifies the inclusion of as many new variables as desired and saves the results to the data frame.

3.3 Recoding variables

Recoding involves creating new values of a variable conditional on the existing values of the same and/or other variables. For example, you may want to

- Change a continuous variable into a set of categories
- Replace miscoded values with correct values
- Create a pass/fail variable based on a set of cutoff scores

To recode data, you can use one or more of R's logical operators (see table 3.3). Logical operators are expressions that return `TRUE` or `FALSE`.

Table 3.3 Logical operators

Operator	Description
<code><</code>	Less than
<code><=</code>	Less than or equal to
<code>></code>	Greater than
<code>>=</code>	Greater than or equal to
<code>==</code>	Exactly equal to
<code>!=</code>	Not equal to
<code>!x</code>	Not x
<code>x y</code>	x or y
<code>x & y</code>	x and y
<code>isTRUE(x)</code>	Tests whether x is TRUE

Let's say you want to recode the ages of the managers in the leadership dataset from the continuous variable `age` to the categorical variable `agecat` (Young, Middle Aged, Elder). First, you must recode the value 99 for `age` to indicate that the value is missing using code such as

```
leadership$age[leadership$age == 99] <- NA
```

The statement `variable[condition] <- expression` will only make the assignment when `condition` is `TRUE`.

Once missing values for `age` have been specified, you can then use the following code to create the `agecat` variable:

```
leadership$agecat[leadership$age > 75] <- "Elder"
leadership$agecat[leadership$age >= 55 &
  leadership$age <= 75] <- "Middle Aged"
leadership$agecat[leadership$age < 55] <- "Young"
```

You include the data-frame names in `leadership$agecat` to ensure that the new variable is saved back to the data frame. (I defined middle aged as 55 to 75 so I won't feel so old.) Note that if you hadn't recoded 99 as *missing* for `age` first, manager 5 would've erroneously been given the value "Elder" for `agecat`.

This code can be written more compactly as follows:

```
leadership <- within(leadership,{
  agecat <- NA
  agecat[age > 75] <- "Elder"
```

```
agecat[age >= 55 & age <= 75] <- "Middle Aged"  
agecat[age < 55] <- "Young" }
```

The `within()` function is similar to the `with()` function (section 2.2.4), but it allows you to modify the data frame. First the variable `agecat` is created and set to *missing* for each row of the data frame. Then the remaining statements within the braces are executed in order. Remember that `agecat` is a character variable; you're likely to want to turn it into an ordered factor, as explained in section 2.2.5.

Several packages offer useful recoding functions; in particular, the `car` package's `recode()` function recodes numeric and character vectors and factors very simply. The package `doBy` offers `recodeVar()`, another popular function. Finally, R ships with `cut()`, which allows you to divide the range of a numeric variable into intervals, returning a factor.

3.4 Renaming variables

If you're not happy with your variable names, you can change them interactively or programmatically. Let's say you want to change the variable `manager` to `managerID` and `date` to `testDate`. You can use the following statement to invoke an interactive editor:

```
fix(leadership)
```

Then you click the variable names and rename them in the dialogs that are presented (see figure 3.1).

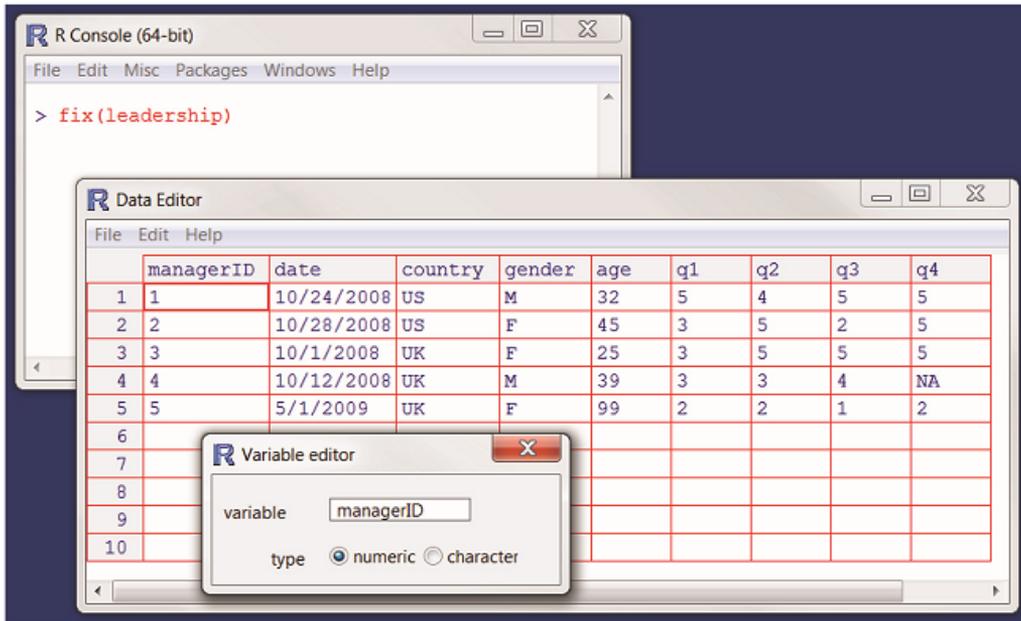


Figure 3.1 Renaming variables interactively using the `fix()` function

Programmatically, you can rename variables via the `names()` function. For example, this statement

```
names(leadership)[2] <- "testDate"
```

renames `date` to `testDate` as demonstrated in the following code:

```
> names(leadership)
[1] "manager" "date" "country" "gender" "age" "q1" "q2"
[8] "q3" "q4" "q5"
> names(leadership)[2] <- "testDate"
> leadership
manager testDate country gender age q1 q2 q3 q4 q5
1 1 10/24/08 US M 32 5 4 5 5 5
2 2 10/28/08 US F 45 3 5 2 5 5
3 3 10/1/08 UK F 25 3 5 5 5 2
4 4 10/12/08 UK M 39 3 3 4 NA NA
5 5 5/1/09 UK F 99 2 2 1 2 1
```

In a similar fashion, the statement

```
names(leadership)[6:10] <- c("item1", "item2", "item3", "item4", "item5")
```

renames `q1` through `q5` to `item1` through `item5`.

3.5 Missing values

In a project of any size, data is likely to be incomplete because of missed questions, faulty equipment, or improperly coded data. In R, missing values are represented by the symbol `NA` (not available). Unlike programs such as SAS, R uses the same missing-value symbol for character and numeric data.

R provides a number of functions for identifying observations that contain missing values. The function `is.na()` allows you to test for the presence of missing values. Assume that you have this vector:

```
y <- c(1, 2, 3, NA)
```

Then the following function returns `c(FALSE, FALSE, FALSE, TRUE)`:

```
is.na(y)
```

Notice how the `is.na()` function works on an object. It returns an object of the same size, with the entries replaced by `TRUE` if the element is a missing value or `FALSE` if the element isn't a missing value. The following listing applies this to the leadership example.

Listing 3.3 Applying the `is.na()` function

```
> is.na(leadership[,6:10])
  q1 q2 q3 q4 q5
[1,] FALSE FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE FALSE
[3,] FALSE FALSE FALSE FALSE FALSE
[4,] FALSE FALSE FALSE TRUE TRUE
[5,] FALSE FALSE FALSE FALSE FALSE
```

Here, `leadership[,6:10]` limits the data frame to columns 6 to 10, and `is.na()` identifies which values are missing.

There are two important things to keep in mind when you're working with missing values in R. First, missing values are considered noncomparable, even to themselves. This means you can't use comparison operators to test for the presence of missing values. For example, the logical test `myvar == NA` is never `TRUE`. Instead, you have to use missing-value functions like `is.na()` to identify the missing values in R data objects.

Second, R doesn't represent infinite or impossible values as missing values. Again, this is different than the way other programs like SAS handle such data. Positive and negative infinity are represented by the symbols `Inf` and `-Inf`, respectively. Thus `5/0` returns `Inf`. Impossible values (for example, `sin(Inf)`) are represented by the symbol `NaN` (not a number). To identify these values, you need to use `is.infinite()` or `is.nan()`.

3.5.1 Recoding values to missing

As demonstrated in section 3.3, you can use assignments to recode values to *missing*. In the leadership example, missing age values are coded as 99. Before analyzing this dataset, you

must let R know that the value 99 means *missing* in this case (otherwise, the mean age for this sample of bosses will be way off!). You can accomplish this by recoding the variable:

```
leadership$age[leadership$age == 99] <- NA
```

Any value of age that's equal to 99 is changed to `NA`. Be sure that any missing data is properly coded as missing before you analyze the data, or the results will be meaningless.

3.5.2 Excluding missing values from analyses

Once you've identified missing values, you need to eliminate them in some way before analyzing your data further. The reason is that arithmetic expressions and functions that contain missing values yield missing values. For example, consider the following code:

```
x <- c(1, 2, NA, 3)
y <- x[1] + x[2] + x[3] + x[4]
z <- sum(x)
```

Both `y` and `z` will be `NA` (missing) because the third element of `x` is missing.

Luckily, most numeric functions have an `na.rm=TRUE` option that removes missing values prior to calculations and applies the function to the remaining values:

```
x <- c(1, 2, NA, 3)
y <- sum(x, na.rm=TRUE)
```

Here, `y` is equal to 6.

When using a function with incomplete data, be sure to check how that function handles missing data by looking at its online help (for example, `help(sum)`). The `sum()` function is only one of many functions we'll consider in chapter 4. Functions allow you to transform data with flexibility and ease.

You can remove *any* observation with missing data by using the `na.omit()` function. `na.omit()` deletes any rows with missing data. Let's apply this to the leadership dataset in the following listing.

Listing 3.4 Using `na.omit()` to delete incomplete observations

```
> leadership
  manager  date country gender age q1 q2 q3 q4 q5 #A
1  1 10/24/08  US    M 32 5 4 5 5 5
2  2 10/28/08  US    F 40 3 5 2 5 5
3  3 10/01/08  UK    F 25 3 5 5 5 2
4  4 10/12/08  UK    M 39 3 3 4 NA NA
5  5 05/01/09  UK    F  NA 2 2 1 2 1

> newdata <- na.omit(leadership)
> newdata
  manager  date country gender age q1 q2 q3 q4 q5 #B
1  1 10/24/08  US    M 32 5 4 5 5 5
2  2 10/28/08  US    F 40 3 5 2 5 5
3  3 10/01/08  UK    F 25 3 5 5 5 2
```

```
#A Data frame with missing data
#B Data frame with complete cases only
```

Any rows containing missing data are deleted from `leadership` before the results are saved to `newdata`.

Deleting all observations with missing data (called *listwise deletion*) is one of several methods of handling incomplete datasets. If there are only a few missing values or they're concentrated in a small number of observations, listwise deletion can provide a good solution to the missing-values problem. But if missing values are spread throughout the data or there's a great deal of missing data in a small number of variables, listwise deletion can exclude a substantial percentage of your data. We'll explore several more sophisticated methods of dealing with missing values in chapter 18. Next, let's look at dates.

3.6 Date values

Dates are typically entered into R as character strings and then translated into date variables that are stored numerically. The function `as.Date()` is used to make this translation. The syntax is `as.Date(x, "input_format")`, where `x` is the character data and `input_format` gives the appropriate format for reading the date (see table 3.4).

Table 3.4 Date formats

Symbol	Meaning	Example
%d	Day as a number (0–31)	01–31
%a	Abbreviated weekday	Mon
%A	Unabbreviated weekday	Monday
%m	Month (00–12)	00–12
%b	Abbreviated month	Jan
%B	Unabbreviated month	January
%y	Two-digit year	07
%Y	Four-digit year	2007

The default format for inputting dates is `yyyy-mm-dd`. The statement

```
mydates <- as.Date(c("2007-06-22", "2004-02-13"))
```

converts the character data to dates using this default format. In contrast,

```
strDates <- c("01/05/1965", "08/16/1975")
dates <- as.Date(strDates, "%m/%d/%Y")
```

reads the data using a `mm/dd/yyyy` format.

In the leadership dataset, date is coded as a character variable in mm/dd/yy format. Therefore:

```
myformat <- "%m/%d/%y"
leadership$date <- as.Date(leadership$date, myformat)
```

uses the specified format to read the character variable and replace it in the data frame as a date variable. Once the variable is in date format, you can analyze and plot the dates using the wide range of analytic techniques covered in later chapters.

Two functions are especially useful for time-stamping data. `Sys.Date()` returns today's date, and `date()` returns the current date and time. As I write this, it's November 27, 2014 at 1:21 pm. So executing those functions produces

```
> Sys.Date()
[1] "2014-11-27"
> date()
[1] "Fri Nov 27 13:21:54 2014"
```

You can use the `format(x, format="output_format")` function to output dates in a specified format and to extract portions of dates:

```
> today <- Sys.Date()
> format(today, format="%B %d %Y")
[1] "November 27 2014"
> format(today, format="%A")
[1] "Thursday"
```

The `format()` function takes an argument (a date in this case) and applies an output format (in this case, assembled from the symbols in table 3.4). The important result here is that there are only two more days until the weekend!

When R stores dates internally, they're represented as the number of days since January 1, 1970, with negative values for earlier dates. That means you can perform arithmetic operations on them. For example,

```
> startdate <- as.Date("2004-02-13")
> enddate <- as.Date("2011-01-22")
> days <- enddate - startdate
> days
Time difference of 2535 days
```

displays the number of days between February 13, 2004 and January 22, 2011.

Finally, you can also use the function `difftime()` to calculate a time interval and express it as seconds, minutes, hours, days, or weeks. Let's assume that I was born on October 12, 1956. How old am I?

```
> today <- Sys.Date()
> dob <- as.Date("1956-10-12")
> difftime(today, dob, units="weeks")
Time difference of 3033 weeks
```

Apparently I am 3,033 weeks old. Who knew? Final test: On which day of the week was I born?

3.6.1 Converting dates to character variables

You can also convert date variables to character variables. Date values can be converted to character values using the `as.character()` function:

```
strDates <- as.character(dates)
```

The conversion allows you to apply a range of character functions to the data values (subsetting, replacement, concatenation, and so on). We'll cover character functions in detail in chapter 4.

3.6.2 Going further

To learn more about converting character data to dates, look at `help(as.Date)` and `help(strftime)`. To learn more about formatting dates and times, see `help(ISOdatetime)`. The `lubridate` package contains a number of functions that simplify working with dates, including functions to identify and parse date-time data, extract date-time components (for example, years, months, days, and so on), and perform arithmetic calculations on date-times. If you need to do complex calculations with dates, the `timeDate` package can also help. It provides a myriad of functions for dealing with dates, can handle multiple time zones at once, and provides sophisticated calendar manipulations that recognize business days, weekends, and holidays.

3.7 Type conversions

In the previous section, we discussed how to convert character data to date values, and vice versa. R provides a set of functions to identify an object's data type and convert it to a different data type.

Type conversions in R work in a similar fashion to those in other statistical programming languages. For example, adding a character string to a numeric vector converts all the elements in the vector to character values. You can use the functions listed in table 3.5 to test for a data type and to convert it to a given type.

Table 3.5 Type-conversion functions

Test	Convert
<code>is.numeric()</code>	<code>as.numeric()</code>
<code>is.character()</code>	<code>as.character()</code>
<code>is.vector()</code>	<code>as.vector()</code>
<code>is.matrix()</code>	<code>as.matrix()</code>

<code>is.data.frame()</code>	<code>as.data.frame()</code>
<code>is.factor()</code>	<code>as.factor()</code>
<code>is.logical()</code>	<code>as.logical()</code>

Functions of the form `is.datatype()` return `TRUE` or `FALSE`, whereas `as.datatype()` converts the argument to that type. The following listing provides an example.

Listing 3.5 Converting from one data type to another

```
> a <- c(1,2,3)
> a
[1] 1 2 3
> is.numeric(a)
[1] TRUE
> is.vector(a)
[1] TRUE
> a <- as.character(a)
> a
[1] "1" "2" "3"
> is.numeric(a)
[1] FALSE
> is.vector(a)
[1] TRUE
> is.character(a)
[1] TRUE
```

When combined with the flow controls (such as `if-then`) that we'll discuss in chapter 4, the `is.datatype()` function can be a powerful tool, allowing you to handle data in different ways depending on its type. Additionally, some R functions require data of a specific type (character or numeric, matrix or data frame), and `as.datatype()` lets you transform your data into the format required prior to analyses.

3.8 Sorting data

Sometimes, viewing a dataset in a sorted order can tell you quite a bit about the data. For example, which managers are most deferential? To sort a data frame in R, you use the `order()` function. By default, the sorting order is ascending. Prepend the sorting variable with a minus sign to indicate descending order. The following examples illustrate sorting with the leadership data frame.

The statement

```
newdata <- leadership[order(leadership$age),]
```

creates a new dataset containing rows sorted from youngest manager to oldest manager. The statement

```
newdata <- leadership[order(leadership$gender, leadership$age),]
```

sorts the rows into female followed by male, and youngest to oldest within each gender.

Finally,

```
newdata <- leadership[order(leadership$gender, -leadership$age),]
```

sorts the rows by gender, and then from oldest to youngest manager within each gender.

3.9 Merging datasets

If your data exists in multiple locations, you'll need to combine it before moving forward. This section shows you how to add columns (variables) and rows (observations) to a data frame.

3.9.1 Adding columns to a data frame

To merge two data frames (datasets) horizontally, you use the `merge()` function. In most cases, two data frames are joined by one or more common key variables (that is, an inner join). For example,

```
total <- merge(dataframeA, dataframeB, by="ID")
```

merges `dataframeA` and `dataframeB` by `ID`. Similarly,

```
total <- merge(dataframeA, dataframeB, by=c("ID","Country"))
```

merges the two data frames by `ID` and `Country`. Horizontal joins like this are typically used to add variables to a data frame.

Horizontal concatenation with `cbind()`

If you're joining two matrices or data frames horizontally and don't need to specify a common key, you can use the `cbind()` function:

```
total <- cbind(A, B)
```

This function horizontally concatenates objects `A` and `B`. For the function to work properly, each object must have the same number of rows and be sorted in the same order.

3.9.2 Adding rows to a data frame

To join two data frames (datasets) vertically, use the `rbind()` function:

```
total <- rbind(dataframeA, dataframeB)
```

The two data frames must have the same variables, but they don't have to be in the same order. If `dataframeA` has variables that `dataframeB` doesn't, then before joining them, do one of the following:

- Delete the extra variables in `dataframeA`.
- Create the additional variables in `dataframeB`, and set them to `NA` (missing).

Vertical concatenation is typically used to add observations to a data frame.

3.10 Subsetting datasets

R has powerful indexing features for accessing the elements of an object. These features can be used to select and exclude variables, observations, or both. The following sections demonstrate several methods for keeping or deleting variables and observations.

3.10.1 Selecting variables

It's a common practice to create a new dataset from a limited number of variables chosen from a larger dataset. In chapter 2, you saw that the elements of a data frame are accessed using the notation `dataframe[row indices, column indices]`. You can use this to select variables. For example,

```
newdata <- leadership[, c(6:10)]
```

selects variables `q1`, `q2`, `q3`, `q4`, and `q5` from the `leadership` data frame and saves them to the data frame `newdata`. Leaving the row indices blank (`,`) selects all the rows by default.

The statements

```
myvars <- c("q1", "q2", "q3", "q4", "q5")
newdata <- leadership[myvars]
```

accomplish the same variable selection. Here, variable names (in quotes) are entered as column indices, thereby selecting the same columns.

Finally, you could use

```
myvars <- paste("q", 1:5, sep="")
newdata <- leadership[myvars]
```

This example uses the `paste()` function to create the same character vector as in the previous example. `paste()` will be covered in chapter 4.

3.10.2 Dropping variables

There are many reasons to exclude variables. For example, if a variable has many missing values, you may want to drop it prior to further analyses. Let's look at some methods of excluding variables.

You can exclude variables `q3` and `q4` with these statements:

```
myvars <- names(leadership) %in% c("q3", "q4")
newdata <- leadership[!myvars]
```

In order to understand why this works, you need to break it down:

1. `names(leadership)` produces a character vector containing the variable names:

```
c("managerID", "testDate", "country", "gender", "age", "q1", "q2", "q3", "q4", "q5")
```

2. `names(leadership) %in% c("q3", "q4")` returns a logical vector with `TRUE` for each element in `names(leadership)` that matches `q3` or `q4` and `FALSE` otherwise:

```
c(FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, TRUE, FALSE)
```

3. The not (`!`) operator reverses the logical values:

```
c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE)
```

4. `leadership[c(TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, FALSE, FALSE, TRUE)]` selects columns with `TRUE` logical values, so `q3` and `q4` are excluded.

Knowing that `q3` and `q4` are the eighth and ninth variables, you can exclude them with the following statement:

```
newdata <- leadership[c(-8,-9)]
```

This works because prepending a column index with a minus sign (`-`) excludes that column.

Finally, the same deletion can be accomplished via

```
leadership$q3 <- leadership$q4 <- NULL
```

Here you set columns `q3` and `q4` to undefined (`NULL`). Note that `NULL` isn't the same as `NA` (missing).

Dropping variables is the converse of keeping variables. The choice depends on which is easier to code. If there are many variables to drop, it may be easier to keep the ones that remain, or vice versa.

3.10.3 Selecting observations

Selecting or excluding observations (rows) is typically a key aspect of successful data preparation and analysis. Several examples are given in the following listing.

Listing 3.6 Selecting observations

```
newdata <- leadership[1:3,] #A
newdata <- leadership[leadership$gender=="M" & #B
  leadership$age > 30,] #B
```

#A Asks for rows 1 through 3 (the first three observations)

#B Selects all men over 30

Each of these examples provides the row indices and leaves the column indices blank (therefore choosing all columns). Let's break down the line of code at `#1` in order to understand it:

1. The logical comparison `leadership$gender=="M"` produces the vector `c(TRUE, FALSE, FALSE, TRUE, FALSE)`.

2. The logical comparison `leadership$age > 30` produces the vector `c(TRUE, TRUE, FALSE, TRUE, TRUE)`.
3. The logical comparison `c(TRUE, FALSE, FALSE, TRUE, FALSE) & c(TRUE, TRUE, FALSE, TRUE, TRUE)` produces the vector `c(TRUE, FALSE, FALSE, TRUE, FALSE)`.
4. `leadership[c(TRUE, FALSE, FALSE, TRUE, FALSE),]` selects the first and fourth observations from the data frame (when the row index is `TRUE`, the row is included; when it's `FALSE`, the row is excluded). This meets the selection criteria (men over 30).

At the beginning of this chapter, I suggested that you might want to limit your analyses to observations collected between January 1, 2009 and December 31, 2009. How can you do this? Here's one solution:

```
leadership$date <- as.Date(leadership$date, "%m/%d/%y") #A

startdate <- as.Date("2009-01-01") #B
enddate <- as.Date("2009-12-31") #C

newdata <- leadership[which(leadership$date >= startdate & #D
  leadership$date <= enddate),]
```

#A Converts the date values read in originally as character values to date values using the format mm/dd/yy
#B Creates starting date
#C Creates ending date
#D Selects cases meeting your desired criteria, as in the previous example

Note that the default for the `as.Date()` function is `yyyy-mm-dd`, so you don't have to supply it here.

3.10.4 The `subset()` function

The examples in the previous two sections are important because they help describe the ways in which logical vectors and comparison operators are interpreted in R. Understanding how these examples work will help you to interpret R code in general. Now that you've done things the hard way, let's look at a shortcut.

The `subset()` function is probably the easiest way to select variables and observations. Here are two examples:

```
newdata <- subset(leadership, age >= 35 | age < 23, #A
  select=c(q1, q2, q3, q4)) #A

newdata <- subset(leadership, gender=="M" & age > 25, #B
  select=gender:q4) #B
```

#A Selects all rows that have a value of `age` greater than or equal to 35 or less than 23. Keeps variables `q1` through `q4`
#B Selects all men over the age of 25, and keeps variables `gender` through `q4` (`gender`, `q4`, and all columns between them)

You saw the colon operator `from:to` in chapter 2. Here, it provides all variables in a data frame between the `from` variable and the `to` variable, inclusive.

3.10.5 Random samples

Sampling from larger datasets is a common practice in data mining and machine learning. For example, you may want to select two random samples, creating a predictive model from one and validating its effectiveness on the other. The `sample()` function enables you to take a random sample (with or without replacement) of size n from a dataset.

You could take a random sample of size 3 from the `leadership` dataset using the following statement:

```
mysample <- leadership[sample(1:nrow(leadership), 3, replace=FALSE),]
```

The first argument to `sample()` is a vector of elements to choose from. Here, the vector is 1 to the number of observations in the data frame. The second argument is the number of elements to be selected, and the third argument indicates sampling without replacement. `sample()` returns the randomly sampled elements, which are then used to select rows from the data frame.

R has extensive facilities for sampling, including drawing and calibrating survey samples (see the `sampling` package) and analyzing complex survey data (see the `survey` package). Other methods that rely on sampling, including bootstrapping and resampling statistics, are described in chapter 12.

3.11 Using dplyr to manipulate data frames

So far, we've manipulated R data frames using base R functions. The `dplyr` package provides a series of shortcuts that allow you to complete the same data management tasks in a streamlined fashion. It is rapidly becoming one of the most popular R packages for data management.

3.11.1 Basic dplyr functions

The `dplyr` package provides a set of functions that can be used to select variables and observations, transform variables, rename variables, and sort rows. The relevant functions are listed in table 3.6.

Table 3.5 `dplyr` functions for manipulating data frames

Function	Use
<code>select()</code>	Select variables/columns
<code>filter()</code>	Select observations/rows
<code>mutate()</code>	Transform or recode variables
<code>rename()</code>	Rename variables/columns
<code>recode()</code>	Recode variable values

```
arrange()
```

```
Order rows by variable values
```

Let's return to the data frame created in table 3.1 and reproduced in table 3.6 for convenience.

Table 3.6 Gender differences in leadership behavior

Manager	Date	Country	Gender	Age	q1	q2	q3	q4	q5
1	10/24/14	US	M	32	5	4	5	5	5
2	10/28/14	US	F	45	3	5	2	5	5
3	10/01/14	UK	F	25	3	5	5	5	2
4	10/12/14	UK	M	39	3	3	4		
5	05/01/14	UK	F	99	2	2	1	2	1

This time we'll use `dplyr` functions to manipulate the dataset. The code is provided in listing 3.7. Since `dplyr` is not part of base R, install it (`install.packages("dplyr")`) before first use.

Listing 3.7 Manipulating data with dplyr

```
manager <- c(1, 2, 3, 4, 5)
date <- c("10/24/08", "10/28/08", "10/1/08", "10/12/08", "5/1/09")
country <- c("US", "US", "UK", "UK", "UK")
gender <- c("M", "F", "F", "M", "F")
age <- c(32, 45, 25, 39, 99)
q1 <- c(5, 3, 3, 3, 2)
q2 <- c(4, 5, 5, 3, 2)
q3 <- c(5, 2, 5, 4, 1)
q4 <- c(5, 5, 5, NA, 2)
q5 <- c(5, 5, 2, NA, 1)
leadership <- data.frame(manager, date, country, gender, age,
  q1, q2, q3, q4, q5, stringsAsFactors=FALSE)

library(dplyr) #A

leadership <- mutate(leadership, #B
  total_score = q1 + q2 + q3 + q4 + q5, #B
  mean_score = total_score / 5) #B

leadership$gender <- recode(leadership$gender, #C
  "M" = "male", "F" = "female") #C

leadership <- rename(leadership, ID = "manager", sex = "gender")#D

leadership <- arrange(leadership, sex, total_score) #E

leadership_ratings <- select(leadership, ID, mean_score) #F

leadership_men_high <- filter(leadership, #G
```

```
sex == "male" & total_score > 10) #G
```

```
#A Load the dplyr package
#B Create two summary variables
#C Recode M and F to male and female
#D Rename the manager and gender variables
#E Sort the data by sex and then total score within sex
#F Create a new data frame containing the rating variables
#G Create a new data frame containing males with total scores above 10
```

First the `dplyr` package is loaded #A. Then `mutate()` function is used to create a total score and mean score #B. The format is

```
dataframe <- mutate(dataframe,
  newvar1 = expression,
  newvar2 = expression, ...).
```

Note that when using `dplyr`, you don't place quote marks around variable names. The new variables are added to the data frame.

Next, the `recode()` function is used to modify the values of the gender variable #C. The format is

```
vector <- recode(vector,
  oldvalue1 = newvalue2,
  oldvalue2 = newvalue2, ...).
```

Vector values that are not given new values are left unchanged. For example

```
x <- c("a", "b", "c")
x <- recode(x, "a" = "apple", "b" = "banana")
x
[1] "apple" "banana" "c"
```

For numeric values, use back ticks to quote the original values.

```
> y <- c(1, 2, 3)
> y <- recode(y, `1` = 10, `2` = 15)
> y
[1] 10 15 3
```

Next, the `rename()` function to change the variable names #D. The format is

```
dataframe <- recode(dataframe,
  newname1 = "oldname1",
  newname2 = "oldname2", ...).
```

The data are then sorted using the `arrange()` function #E. First the rows are sorted in ascending order by sex (females followed by males). Next the rows are in ascending order by `total_score` (low scores to high scores) separately within each sex group. The `desc()` function is used to reverse the order of the sorting. For example

```
leadership <- arrange(leadership, sex, desc(total_score))
```

would sort the data in ascending order by sex and in descending order (high to low `total_scores`) within each sex.

The `select` statement is used to select or exclude variables #F. In this case, the variables `ID` and `mean_score` are selected.

The format for the `select()` function is

```
dataframe <- select(dataframe, variablelist1, variablelist2, ...)
```

Variable lists are typically variable names without quotes. The colon operator (`:`) can be used to select a range of variables. Additionally, functions can be used to select variables contain specific text strings. For example, the statement

```
leadership_subset <- select(leadership,
  ID, country:age, starts_with("q"))
```

would select the variables `ID`, `country`, `sex`, `age`, `q1`, `q2`, `q3`, `q4` and `q5`. See `help(select_helpers)` for a list of functions that can be used to aid in the selection of variables.

A minus sign (`-`) is used to exclude variables. The statement

```
leadership_subset <- select(leadership, -sex, -age)
```

would include all variables except `sex` and `age`.

Finally, the `filter()` function is used to select the observations or rows in a data frame meeting a given set of criteria #G. Here, men with total scores greater than 10 are retained. The format is

```
dataframe <- filter(dataframe, expression)
```

and rows are retained if the expression is TRUE. Any of the logical operators in table 3.3 can be used and parentheses can be used to clarify the precedence of these operators. For example

```
extreme_men <- filter(leadership,
  sex == "male" &
  (mean_score < 2 | mean_score > 4))
```

would create a data frame containing all male managers with mean scores below 2 or above 4.

3.11.2 Using pipe operators to chain statements

The `dplyr` package allows you to write code in a compact format using the pipe operator (`%>%`) provided by the `magrittr` package. Consider the following three statements.

```
high_potentials <- filter(leadership, total_score > 10)
high_potentials <- select(high_potential, ID, country, mean_score)
high_potentials <- arrange(high_potential, country, mean_score)
```

These statements can be rewritten as a single statement using the pipe operator.

```
high_potentials <- filter(leadership, total_score > 10) %>%
```

```
select(ID, country, mean_score) %>%
  arrange(country, mean_score)
```

The `%>%` operator (pronounced THEN) passes the result on the left-hand side to the first parameter of the function on the right-hand side. A statement rewritten this way is often easier to read.

Although we have covered basic `dplyr` functions, the package also contains has functions summarizing, combining, and restructure data. These additional functions will be discussed in chapter 4.

3.12 Using SQL statements to manipulate data frames

Until now, you've been using R statements and functions to manipulate data. But many data analysts come to R well versed in Structured Query Language (SQL). It would be a shame to lose all that accumulated knowledge. Therefore, before we end, let me briefly mention the existence of the `sqldf` package. (If you're unfamiliar with SQL, please feel free to skip this section.)

After downloading and installing the package (`install.packages("sqldf")`), you can use the `sqldf()` function to apply SQL `SELECT` statements to data frames. Two examples are given in the following listing.

Listing 3.7 Using SQL statements to manipulate data frames

```
> library(sqldf) #A
> newdf <- sqldf("select * from mtcars where carb=1 order by mpg", #A
  row.names=TRUE) #A
> newdf #A
  mpg cyl disp hp drat wt qsec vs am gear carb
Valiant 18.1 6 225.0 105 2.76 3.46 20.2 1 0 3 1
Hornet 4 Drive 21.4 6 258.0 110 3.08 3.21 19.4 1 0 3 1
Toyota Corona 21.5 4 120.1 97 3.70 2.46 20.0 1 0 3 1
Datsun 710 22.8 4 108.0 93 3.85 2.32 18.6 1 1 4 1
Fiat X1-9 27.3 4 79.0 66 4.08 1.94 18.9 1 1 4 1
Fiat 128 32.4 4 78.7 66 4.08 2.20 19.5 1 1 4 1
Toyota Corolla 33.9 4 71.1 65 4.22 1.83 19.9 1 1 4 1

> sqldf("select avg(mpg) as avg_mpg, avg(displ) as avg_disp, gear #B
  from mtcars where cyl in (4, 6) group by gear") #B
  avg_mpg avg_disp gear
1 20.3 201 3
2 24.5 123 4
3 25.4 120 5
```

#A Selects all variables (columns) from data frame `mtcars`, keeps only automobiles (rows) with one carburetor (`carb`), sorts in ascending order by `mpg`, and saves the results as the data frame `newdf`. The option `row.names=TRUE` carries the row names from the original data frame over to the new one.

#B Prints the mean `mpg` and `disp` within each level of `gear` for automobiles with four or six cylinders (`cyl`)

Experienced SQL users will find the `sqldf` package a useful adjunct to data management in R. See the project home page (<https://github.com/ggrothendieck/sqldf>) for more details.

3.13 Summary

- Base R functions can be used to create new variables and rename existing variables.
- In R, missing values are represented with the symbol `NA`. Functions are provided for recoding data as missing and removing missing values from data frames prior to analyses.
- Date values are typically input as character strings, translated into date values via functions such as `as.Date()`, and stored internally as numerical values (the number of days since January 1, 1970).
- Base R functions are provided for subsetting the columns (variables), and rows (observations) of a data frame, sorting the rows of a data frame, and merging two or more data frames together. The `dplyr` package provides a set of functions for accomplishing these tasks in a format that is often easier to use and understand.

4

Getting started with graphs

This chapter covers

- **An introduction to the `ggplot2` package**
- **Creating a simple bivariate (2-variable) graph**
- **Using grouping and faceting to create multivariate graphs**
- **Saving graphs in multiple formats**

On many occasions, I've presented clients with carefully crafted statistical results in the form of numbers and text, only to have their eyes glaze over while the chirping of crickets permeated the room. Yet those same clients had enthusiastic "Ah-ha!" moments when I presented the same information to them in the form of graphs. Often I can see patterns in data or detect anomalies in data values by looking at graphs—patterns or anomalies that I completely missed when conducting more formal statistical analyses.

Human beings are remarkably adept at discerning relationships from visual representations. A well-crafted graph can help you make meaningful comparisons among thousands of pieces of information, extracting patterns not easily found through other methods. This is one reason why advances in the field of statistical graphics have had such a major impact on data analysis. Data analysts need to *look* at their data, and this is one area where R shines.

The R language has grown organically over the years, through the contributions of many independent software developers. This has led to the creation of four distinct approaches to graph creation in R – `base`, `lattice`, `ggplot2`, and `grid` graphics. Appendix H provides an overview of each of these systems. In this chapter, and throughout a majority of the remaining chapters, we'll focus on `ggplot2`, the most powerful and popular approach currently available in R.

The `ggplot2` package, written by Hadley Wickham (2009a), provides a system for creating graphs based on the grammar of graphics described by Wilkinson (2005) and expanded by Wickham (2009b). The intention of the `ggplot2` package is to provide a comprehensive, grammar-based system for generating graphs in a unified and coherent manner, allowing users to create new and innovative data visualizations.

This chapter will walk you through the major concepts and functions used to create `ggplot2` graphs by using visualizations to address the following questions:

- What is the relationship between a worker's past experience and their salary?
- How can we summarize this relationship simply?
- Is this relationship different for men and women?
- Does it matter what industry the worker is in?

We'll start with a simple scatterplot displaying the relationship between workers' experience and wages. Then in each section, we'll add new features until we've produced a single publication quality plot that addresses these questions. At each step, we'll hopefully gain greater insight into the questions presented.

To answer these questions, we'll use the `CPS85` data frame contained in the `mosaicData` package. The data frame contains a random sample of 534 individuals selected from the 1985 Current Population Survey, and includes information their wages, demographics, and work experience. Be sure to install both the `mosaicData` and `ggplot2` packages before continuing (`install.packages(c("mosaicData", "ggplot2"))`).

4.1 Creating a graph with `ggplot2`

The `ggplot2` package uses a series of functions to build up a graph in layers. We'll build a complex graph by starting with a simple graph and adding additional elements, one at a time. By default, `ggplot2` graphs appear on a grey background with white reference lines. We'll start by setting the default theme to a white background with light grey reference lines. This looks better when printed in black and white. Let's load the `ggplot2` package and set this default theme.

```
library(ggplot2)
theme_set(theme_bw())
```

Themes are described in section 4.1.7.

4.1.1 `ggplot`

The first function in building a graph is the `ggplot()` function. It specifies the

- data frame containing the data to be plotted
- the mapping of the variables to visual properties of the graph. The mappings are placed in an `aes()` function (which stands for aesthetics or "something you can see").

The code

```
library(ggplot2)
library(mosaicData)
ggplot(data = CPS85, mapping = aes(x = exper, y = wage))
```

produces the graph in figure 4.1.

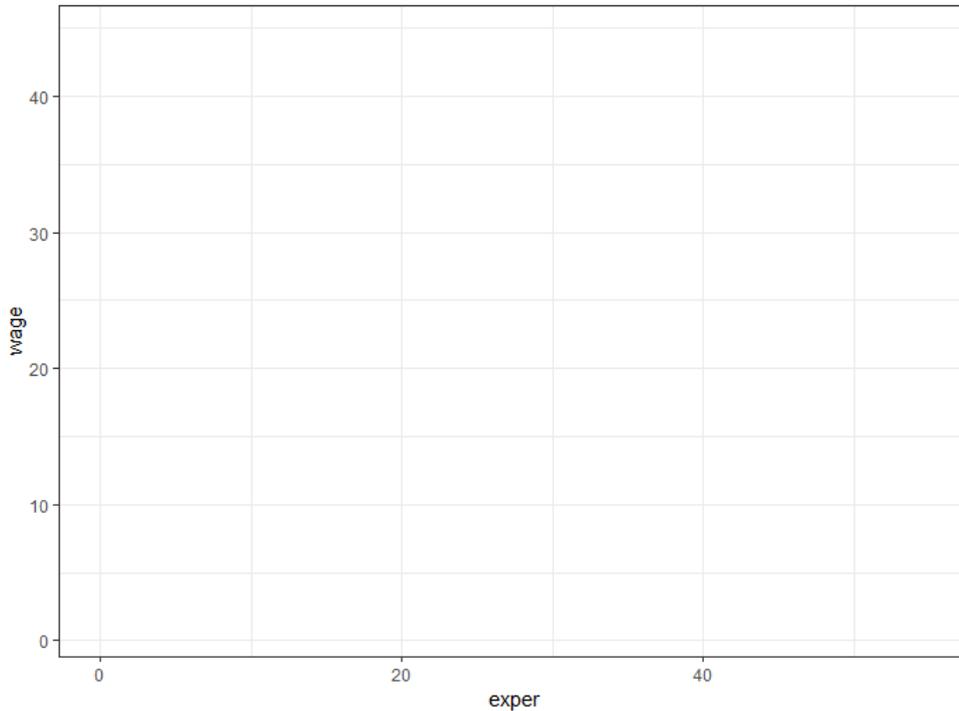


Figure 4.1 Mapping worker experience and wages to the x- and y-axes

Why is the graph empty? We specified that the `exper` variable should be mapped to the x-axis and that the `wage` variable should be mapped to the y-axis, but we haven't yet specified *what* we wanted placed on the graph. In this case, we'll want points to represent each participant.

4.1.2 Geoms

Geoms are the geometric objects (points, lines, bars, and shaded regions) that can be placed on a graph. They are added using functions that start with the phrase `geom_`. Currently, 37 different geoms are available and the list is growing. Table 4.1 describes the more common geoms, along with frequently used options for each.

Table 4.1 Geom functions

Function	Adds	Options
<code>geom_bar()</code>	Bar chart	color, fill, alpha
<code>geom_boxplot()</code>	Box plot	color, fill, alpha, notch, width
<code>geom_density()</code>	Density plot	color, fill, alpha, linetype
<code>geom_histogram()</code>	Histogram	color, fill, alpha, linetype, binwidth
<code>geom_hline()</code>	Horizontal lines	color, alpha, linetype, size
<code>geom_jitter()</code>	Jittered points	color, size, alpha, shape
<code>geom_line()</code>	Line graph	color, alpha, linetype, size
<code>geom_point()</code>	Scatterplot	color, alpha, shape, size
<code>geom_rug()</code>	Rug plot	color, side
<code>geom_smooth()</code>	Fitted line	method, formula, color, fill, linetype, size
<code>geom_text()</code>	Text annotations	Many; see the help for this function
<code>geom_violin()</code>	Violin plot	color, fill, alpha, linetype
<code>geom_vline()</code>	Vertical lines	color, alpha, linetype, size

We'll add points using the `geom_point()` function, creating a scatterplot. In `ggplot2` graphs, functions are chained together using the `+` sign to build a final plot.

```
library(ggplot2)
```

```
library(mosaicData)
ggplot(data = CPS85, mapping = aes(x = exper, y = wage)) + geom_point()
```

The results can be seen in figure 4.2.

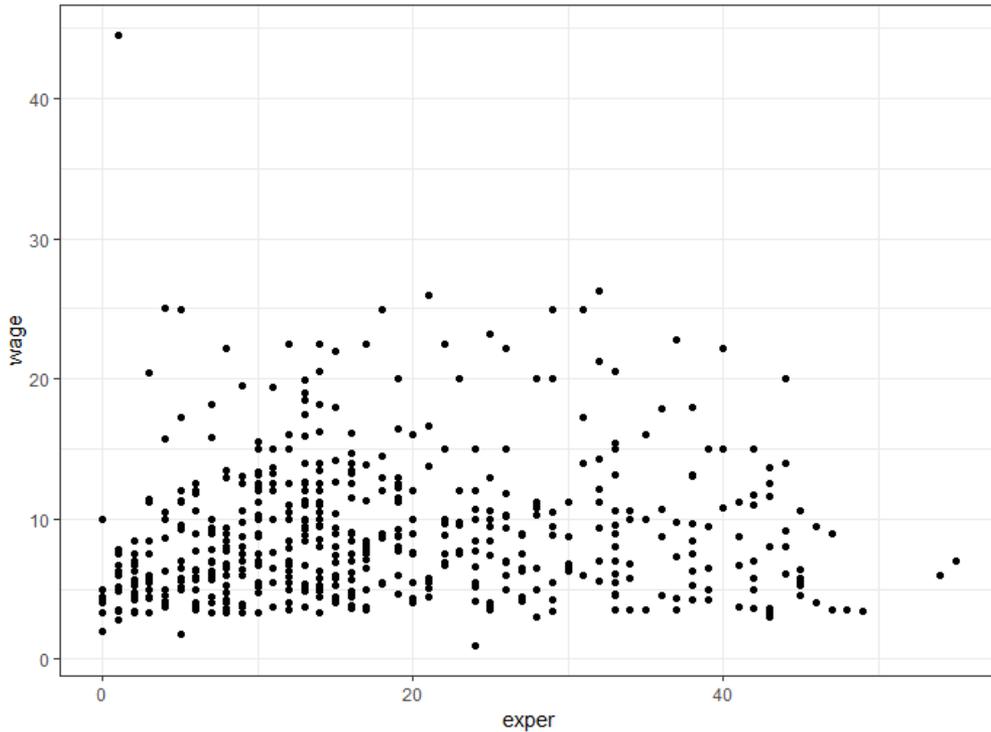


Figure 4.2 Scatterplot of worker experience vs. wages

It appears that as experience goes up, wages go up, but the relationship is weak. The graph also indicates that there is an outlier. One individual has a wage much higher than the rest. We'll delete this case and reproduce the plot.

```
CPS85 <- CPS85[CPS85$wage < 40, ]
ggplot(data = CPS85, mapping = aes(x = exper, y = wage)) +
  geom_point()
```

The new graph is displayed in figure 4.3.

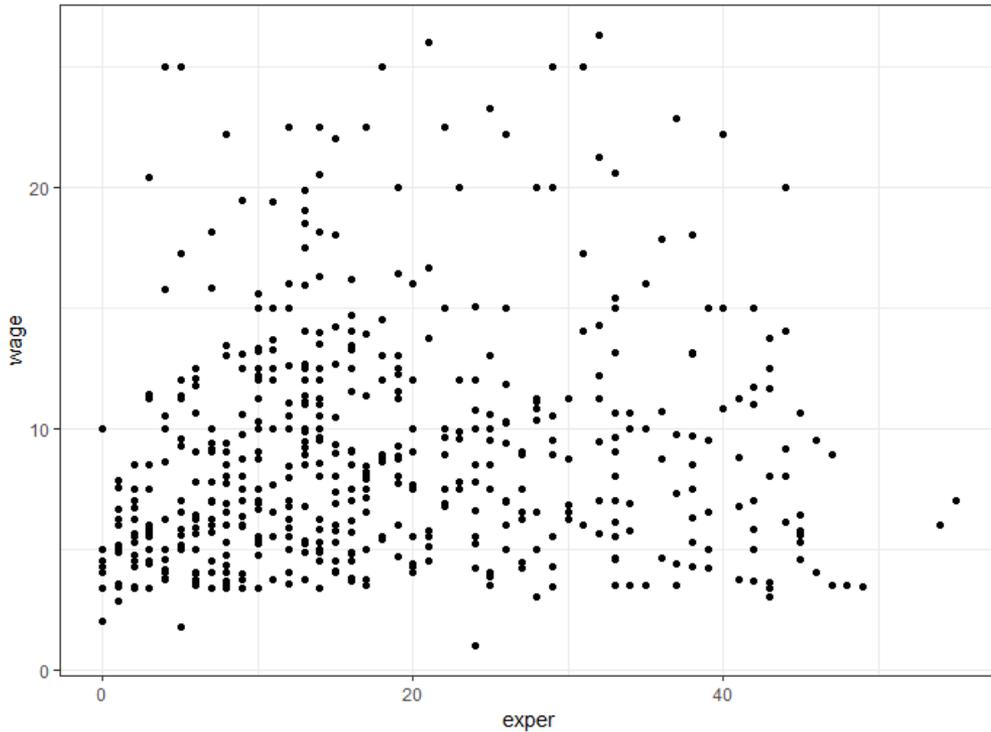


Figure 4.3 Scatterplot of worker experience vs. wages with outlier removed

A number of options can be specified in a `geom_` function (see table 4.1). Options for `geom_point()` include `color`, `size`, `shape`, and `alpha`. These control the point color, size, shape, and transparency, respectively. Colors can be specified by name or hexadecimal code. Shape and `linetype` can be specified by the name or number representing the pattern or symbol respectively. Point size is specified with positive real numbers starting at zero. Large numbers produce larger point sizes. Transparency ranges from 0 (completely transparent) to 1 (completely opaque). Adding a degree of transparency can help visualize overlapping points. Each of these options is described more fully in Chapter 19 (Advanced Graphics with `ggplot2`).

Let's make the points in figure 4.3 larger, semi-transparent, and blue. The code

```
ggplot(data = CPS85, mapping = aes(x = exper, y = wage)) +
  geom_point(color = "cornflowerblue", alpha = .7, size = 3)
```

produces the graph in figure 4.4. We'll also change the gray background to white using `theme` (themes are described in section 4.1.7). I might argue that the chart is more attractive (at least if you have color output), but it doesn't add to our insights. It would be helpful if the graph had a line summarizing the trend between experience and wages.

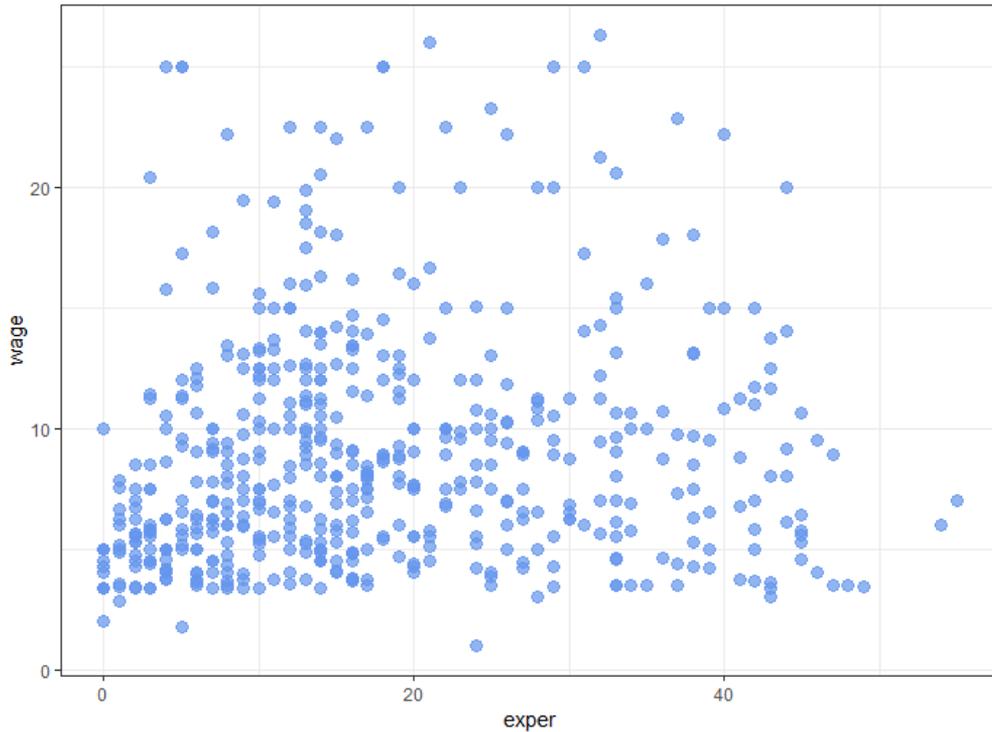


Figure 4.4 Scatterplot of worker experience vs. wages with outlier removed with modified point color, transparency, and point size

We can add this line with the `geom_smooth()` function. Options control the type of line (linear, quadratic, nonparametric), the thickness of the line, the line's color, and the presence or absence of a confidence interval. Each of these is discussed in Chapter 11 (Intermediate Graphics). Here we request a linear regression (`method = lm`) line (where `lm` stands for linear model).

```
ggplot(data = CPS85, mapping = aes(x = exper, y = wage)) +
  geom_point(color = "cornflowerblue", alpha = .7, size = 3) +
  geom_smooth(method = "lm")
```

The results are given in figure 4.5.

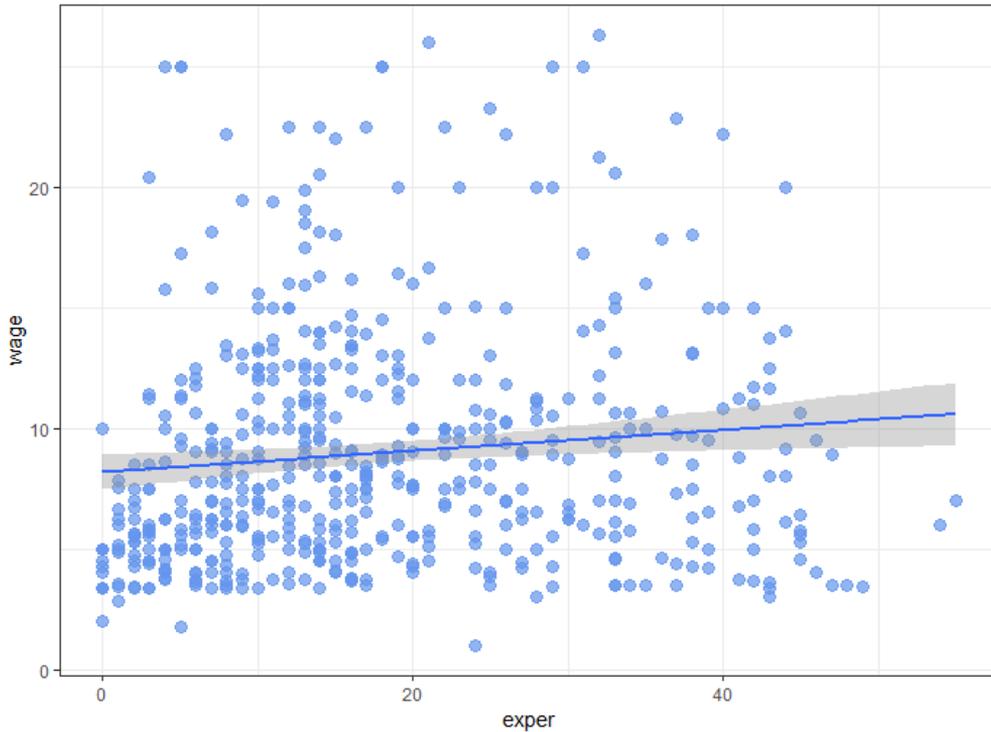


Figure 4.5 Scatterplot of worker experience vs. wages with a line of best fit

We can see from this line that on average, wages appear to increase to a moderate degree with experience. This chapter uses only two geoms. In future chapters, we'll use others to create a wide variety of graph types, including bar charts, histograms, boxplots, density plots, and others.

4.1.3 Grouping

In the previous section, we set graph characteristics such as color and transparency to a *constant* value. However, we can also map variables values to the color, shape, size, transparency, line style, and other visual characteristics of geometric objects. This allows groups of observations to be superimposed in a single graph (a process called grouping).

Let's add sex to the plot and represent it by color, shape, and linetype.

```
ggplot(data = CPS85,
  mapping = aes(x = exper, y = wage,
    color = sex, shape = sex, linetype = sex)) +
  geom_point(alpha = .7, size = 3) +
  geom_smooth(method = "lm", se = FALSE, size = 1.5)
```

By default, the first group (female) is represented by pink filled circles and a solid pink line, while the second group (male) is represented by teal filled triangles and a dashed teal line. The new graph is presented in figure 4.6

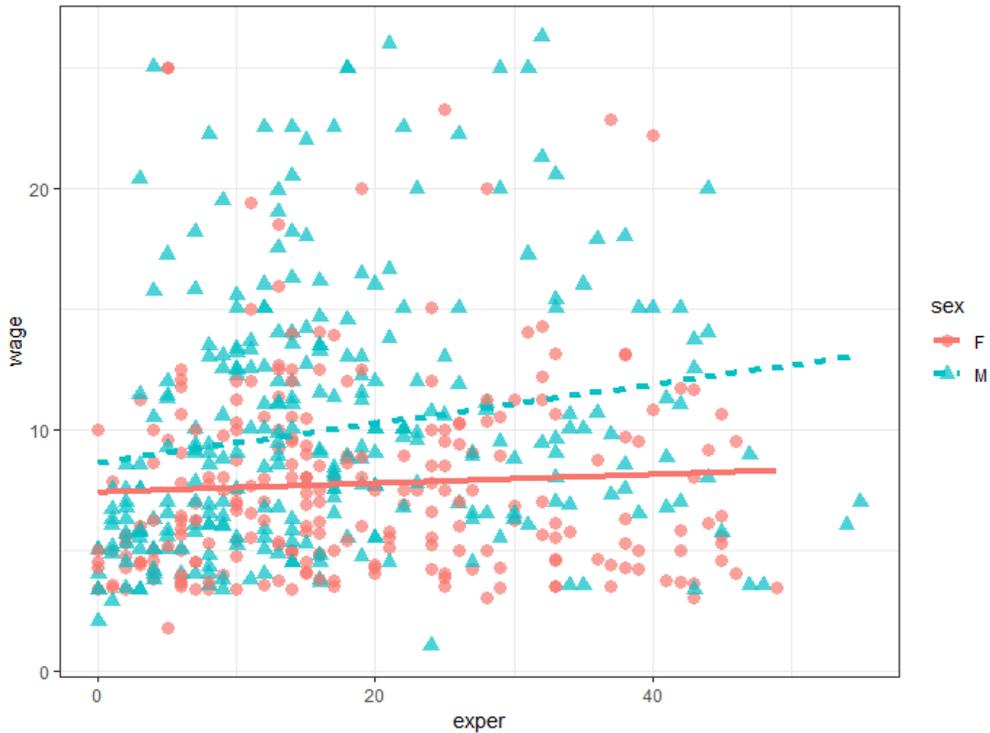


Figure 4.6 Scatterplot of worker experience vs. wages with points colored by sex and separate line of best fit for men and women.

Note that the `color=sex`, `shape=sex`, and `linetype=sex`, options are placed in the `aes()` function because we are mapping a variable to an aesthetic. The `geom_smooth` option (`se = FALSE`) was added to suppress the confidence intervals, making the graph less busy and easier to read. The `size = 1.5` option makes the line a bit thicker.

Simplifying Graphs

In general, our goal is to create graphs that are as simple as possible while conveying the information accurately. In the graphs in this chapter, I would probably map gender to color alone. Adding mappings to shape and line type make the graphs unnecessarily busy. I've added them here in order to create graphs that are more easily readable in both color (e-book) and the greyscale (print) formats of this book.

It now appears that men tend to make more money than women (higher line). Additionally, there may be a stronger relationship between experience and wages for men than for women (steeper line).

4.1.4 Scales

As we've seen, the `aes()` function is used to map variables to the visual characteristics of a plot. Scales specify *how* each of these mappings occurs. For example, `ggplot2` automatically creates plot axes with tick marks, tick mark labels, and axis labels. Often they look fine, but occasionally you'll want to take greater control over their appearance. Colors that represent groups are chosen automatically, but you may want to select a different set of colors based on your tastes or a publication's requirements.

Scale functions (which start with `scale_`) allow you to modify these default scaling. Some common scaling functions are listed in table 4.2.

Table 4.2 Some common scale functions

Function	Description
<code>scale_x_continuous()</code> , <code>scale_y_continuous()</code>	Scales the x and y axes for quantitative variables. Options include <code>breaks</code> for specifying tick marks, <code>labels</code> for specifying tick mark labels, and <code>limits</code> to control the range of the values displayed.
<code>scale_x_discrete()</code> , <code>scale_y_discrete()</code>	Same as above for axes representing categorical variables.
<code>scale_color_manual()</code>	Specifies the colors used to represent the levels of a categorical variable. The <code>values</code> option specifies the colors. A table of colors can be found at http://research.stowers.org/mcm/efg/R/Color/Chart/ColorChart.pdf

In the next plot, we'll change the x- and y-axis scaling, and the colors representing males and females. The x-axis representing `exper` will range from 0 to 60 by 10, and the y-axis representing `wage` will range from 0 to 30 by 5. Females will be coded with an off-red color and males will be coded with an off-blue color. The code

```
ggplot(data = CPS85,
       mapping = aes(x = exper, y = wage,
                    color = sex, shape=sex, linetype=sex)) +
```

```
geom_point(alpha = .7, size = 3) +
geom_smooth(method = "lm", se = FALSE, size = 1.5) +
scale_x_continuous(breaks = seq(0, 60, 10)) +
scale_y_continuous(breaks = seq(0, 30, 5)) +
scale_color_manual(values = c("indianred3", "cornflowerblue"))
```

produces the graph in figure 4.7.

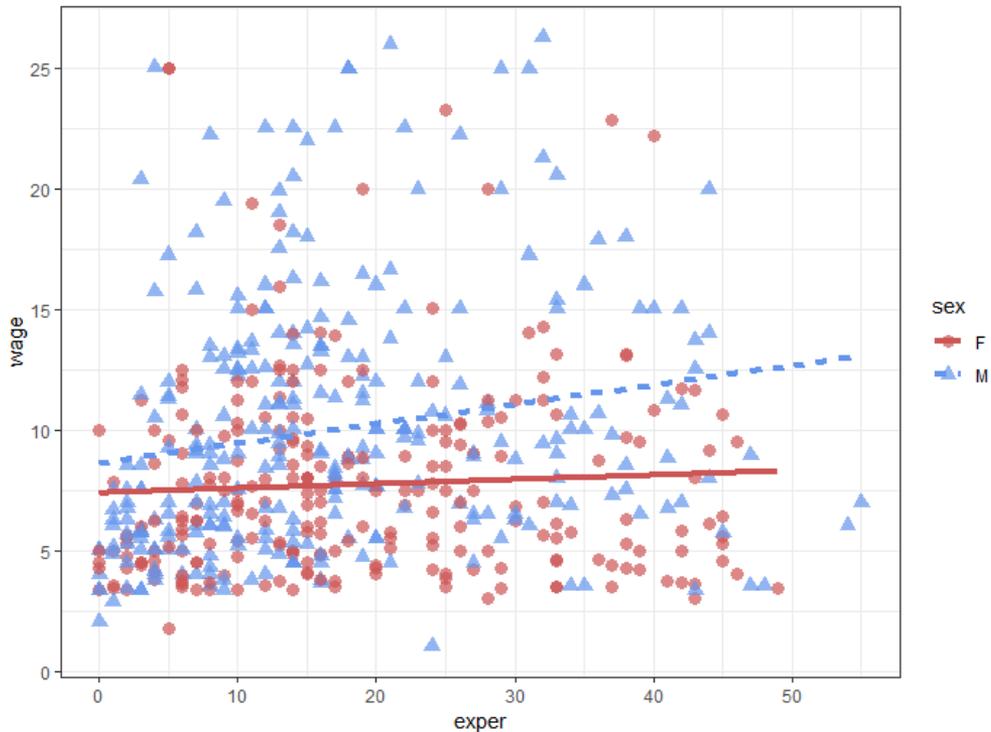


Figure 4.7 Scatterplot of worker experience vs. wages with custom x- and y-axes and custom color mappings for sex.

The numbers on the x- and y-axes are better, and the colors are more attractive (IMHO). However, wages are in dollars. We can change the labels on the y-axis to represent dollars using the `scales` package. The `scales` package provides label formatting for dollars, euros, percents, and more.

Install the `scales` package (`install.packages("scales")`) and then run the following code.

```
ggplot(data = CPS85,
       mapping = aes(x = exper, y = wage,
                     color = sex, shape=sex, linetype=sex)) +
  geom_point(alpha = .7, size = 3) +
  geom_smooth(method = "lm", se = FALSE, size = 1.5) +
  scale_x_continuous(breaks = seq(0, 60, 10)) +
  scale_y_continuous(breaks = seq(0, 30, 5),
                    label = scales::dollar) +
  scale_color_manual(values = c("indianred3", "cornflowerblue"))
```

The results are provided in Figure 4.8.

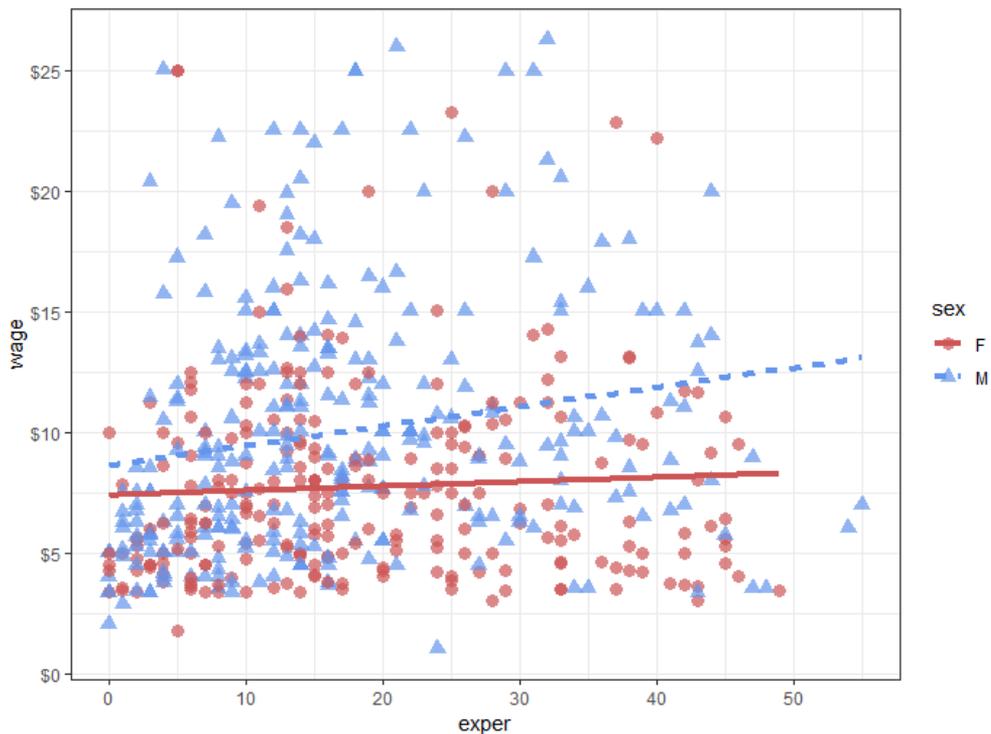


Figure 4.8 Scatterplot of worker experience vs. wages with custom x- and y-axes and custom color mappings for sex. Wages are printed in dollar format.

We are definitely getting there. Here is the next question. Is the relationship between experience, wages and sex the same for each job sector? Let's repeat this graph once for each job sector in order to explore this.

4.1.5 Facets

Sometimes relationships are clearer if groups appear in side-by-side graphs rather than overlapping in a single graph. Facets reproduce a graph for each level of a given variable (or combination of variables). You can create faceted graphs using the `facet_wrap()` and `facet_grid()` functions. The syntax is given in table 14.3, where `var`, `rowvar`, and `colvar` are factors.

Table 4.3 `ggplot2` facet functions

Syntax	Results
<code>facet_wrap(~var, ncol=n)</code>	Separate plots for each level of <code>var</code> arranged into <code>n</code> columns
<code>facet_wrap(~var, nrow=n)</code>	Separate plots for each level of <code>var</code> arranged into <code>n</code> rows
<code>facet_grid(rowvar~colvar)</code>	Separate plots for each combination of <code>rowvar</code> and <code>colvar</code> , where <code>rowvar</code> represents rows and <code>colvar</code> represents columns
<code>facet_grid(rowvar~.)</code>	Separate plots for each level of <code>rowvar</code> , arranged as a single column
<code>facet_grid(.~colvar)</code>	Separate plots for each level of <code>colvar</code> , arranged as a single row

Here, facets will be defined by the eight levels of the sector variable. Since each facet will be smaller than a one panel graph alone, we'll omit `size=3` from `geom_point()` and `size=1.5` from `geom_smooth()`. This will reduce the point and line sizes compared with the previous graphs and looks better in a faceted graph. The code

```
ggplot(data = CPS85,
       mapping = aes(x = exper, y = wage,
                    color = sex, shape = sex, linetype = sex)) +
  geom_point(alpha = .7) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_x_continuous(breaks = seq(0, 60, 10)) +
  scale_y_continuous(breaks = seq(0, 30, 5),
                    label = scales::dollar) +
  scale_color_manual(values = c("indianred3", "cornflowerblue")) +
  facet_wrap(~sector)
```

produces figure 4.9.

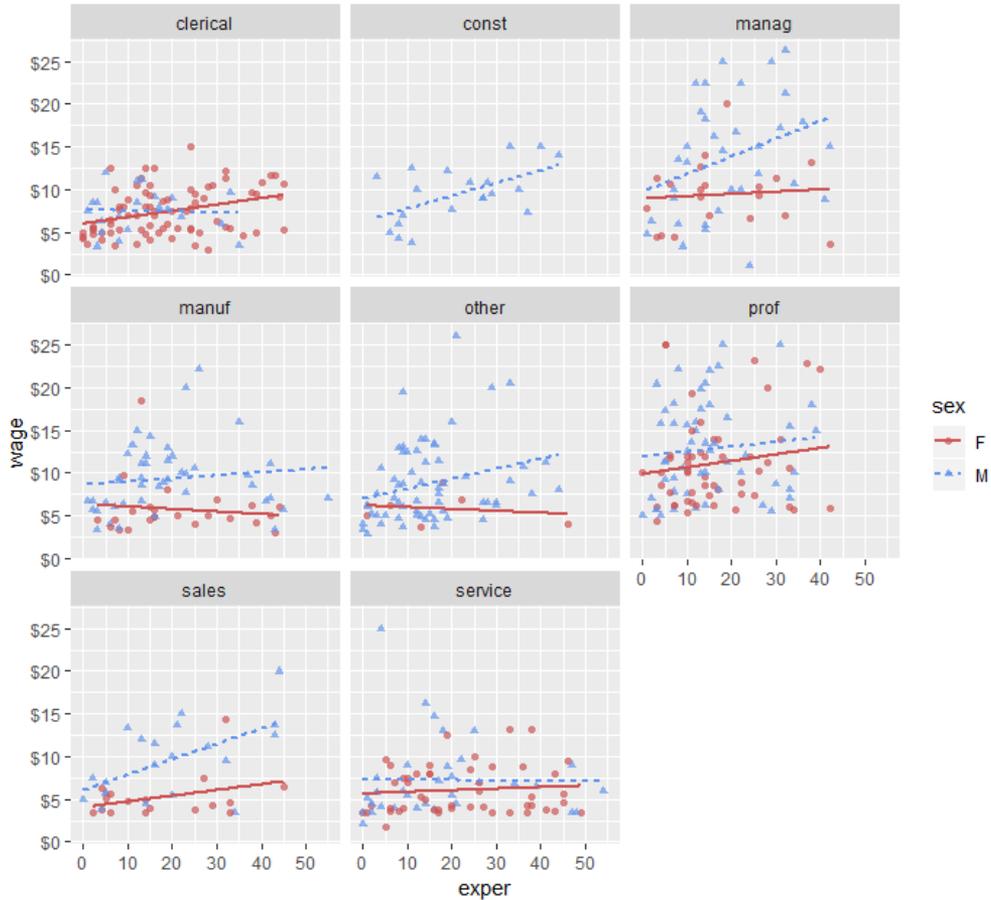


Figure 4.9 Scatterplot of worker experience vs. wages with custom x- and y-axes and custom color mappings for sex. Separate graphs (facets) are provided for each of 8 job sectors.

It appears that the differences between men and women depend on the job sector under consideration. For example, there is a strong positive relationship between experience and wages for male managers, but not for female managers. To a lesser extent, this is also true for sales workers. There appears to be no relationship between experience and wages for both male and female service workers. In either case, males make slightly more. Wages go up with experience for female clerical workers, but may go down for male clerical workers (the relationship may not be significant here). We have gained a great deal of insight into the relationship of wages and experience at this point.

4.1.6 Labels

Graphs should be easy to interpret and informative labels are a key element in achieving this goal. The `labs()` function provides customized labels for the axes and legends. Additionally, a custom title, subtitle, and caption can be added. Let's modify each in the following code.

```
ggplot(data = CPS85,
  mapping = aes(x = exper, y = wage,
    color = sex, shape=sex, linetype=sex)) +
  geom_point(alpha = .7) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_x_continuous(breaks = seq(0, 60, 10)) +
  scale_y_continuous(breaks = seq(0, 30, 5),
    label = scales::dollar) +
  scale_color_manual(values = c("indianred3",
    "cornflowerblue")) +
  facet_wrap(~sector) +
  labs(title = "Relationship between wages and experience",
    subtitle = "Current Population Survey",
    caption = "source: http://mosaic-web.org/",
    x = "Years of Experience",
    y = "Hourly Wage",
    color = "Gender", shape = "Gender", linetype = "Gender")
```

The graph is provided in figure 4.10.

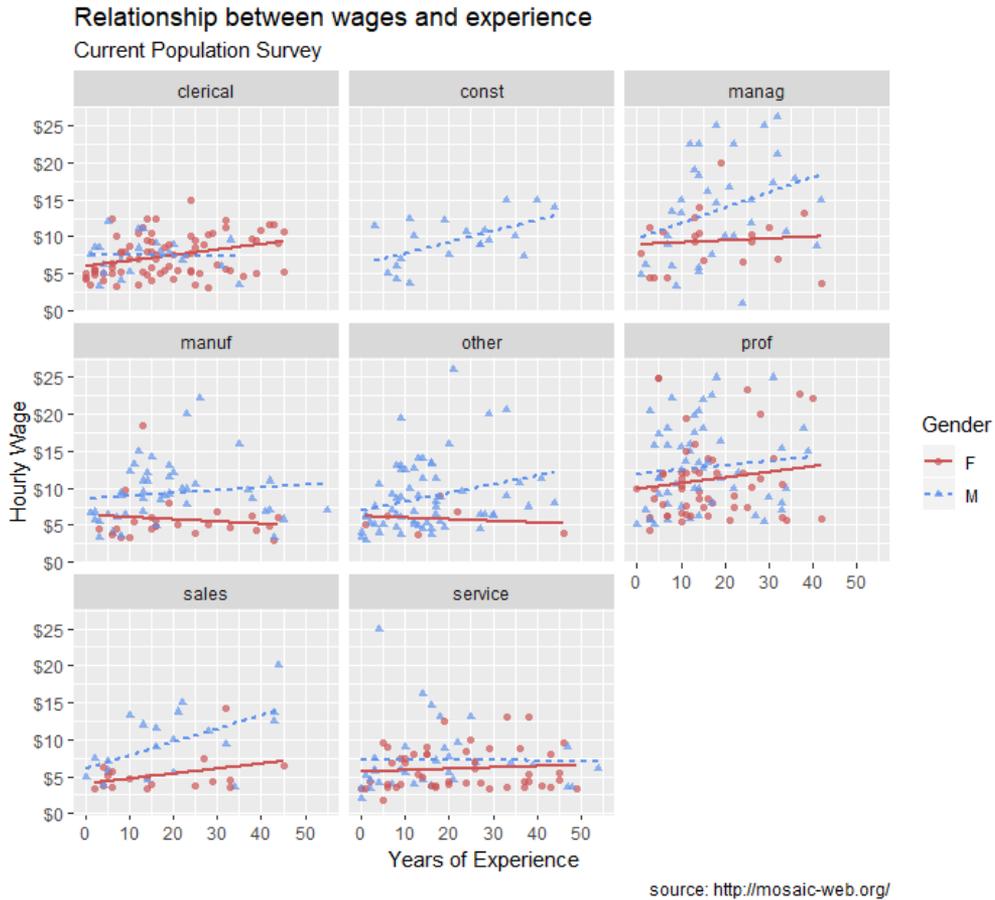


Figure 4.10 Scatterplot of worker experience vs. wages with separate graphs (facets) for each of 8 job sectors and custom titles and labels.

Now a viewer doesn't need to guess what the labels `expr` and `wage` mean, or where the data come from.

4.1.7 Themes

Finally, we can fine tune the appearance of the graph using themes. Theme functions (which start with `theme_`) control background colors, fonts, grid-lines, legend placement, and other non-data related features of the graph. Let's use a cleaner theme. We used themes at the beginning of section 4.1 in order to give each plot a white background. Let's try a different theme – one that is more minimalistic. The code

```
ggplot(data = CPS85,
       mapping = aes(x = exper, y = wage, color = sex)) +
  geom_point(alpha = .6) +
  geom_smooth(method = "lm", se = FALSE) +
  scale_x_continuous(breaks = seq(0, 60, 10)) +
  scale_y_continuous(breaks = seq(0, 30, 5),
                    label = scales::dollar) +
  scale_color_manual(values = c("indianred3", "cornflowerblue")) +
  facet_wrap(~sector) +
  labs(title = "Relationship between wages and experience",
       subtitle = "Current Population Survey",
       caption = "source: http://mosaic-web.org/",
       x = "Years of Experience",
       y = "Hourly Wage",
       color = "Gender") +
  theme_minimal()
```

produces the graph in figure 4.11.

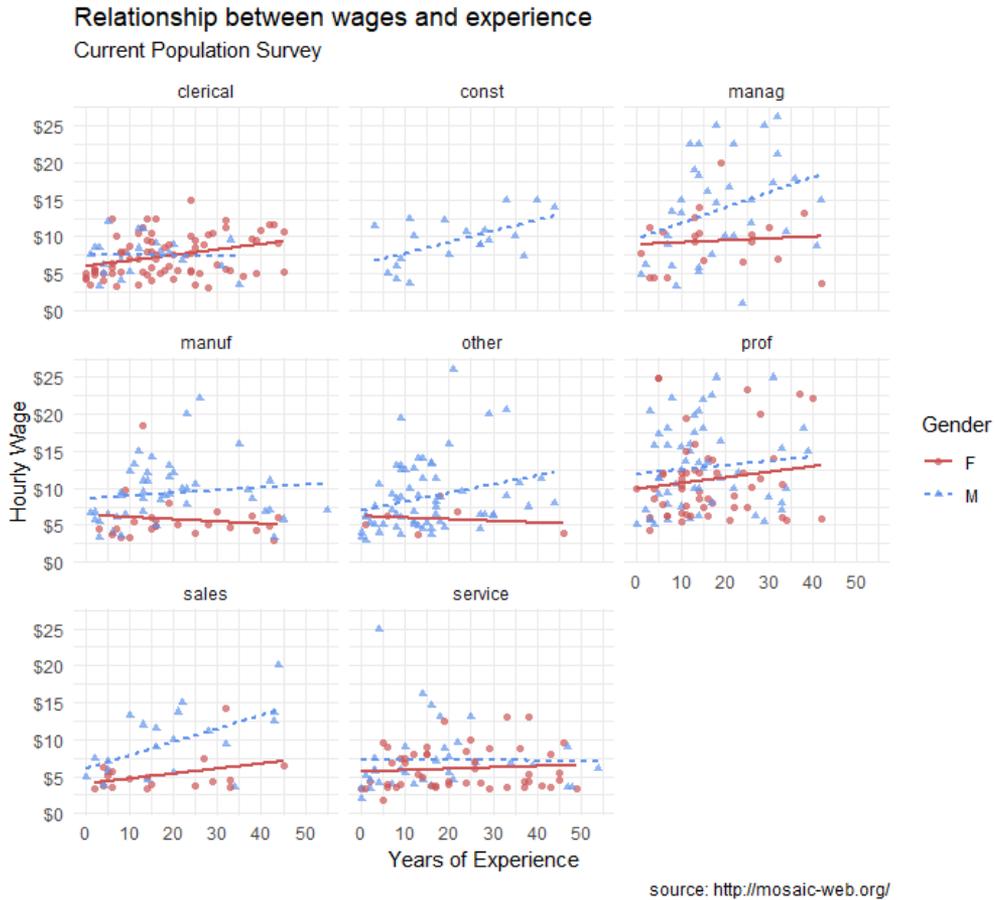


Figure 4.11 Scatterplot of worker experience vs. wages with separate graphs (facets) for each of 8 job sectors and custom titles and labels, and a cleaner theme.

This is our finished graph, ready for publication. Of course, these findings are tentative. They are based on a limited sample size and don't involve statistical testing to assess whether differences may be due to chance variation. Appropriate tests for this type of data will be described in Chapter 8 (Regression).

4.2 ggplot2 details

Before finishing this chapter, there are three important topics to consider: the placement of the `aes()` function, the treatment of `ggplot2` graphs as R objects, and various methods to save your graphs for use in reports and webpages.

4.2.1 Placing the data and mapping options

Plots created with `ggplot2` always start with the `ggplot` function. In the previous examples, the `data=` and `mapping=` options were placed in this function. In this case they apply to each `geom` function that follows.

You can also place these options directly within a `geom`. In that case, they *only* apply to that *specific* `geom`. Consider the following graph.

```
ggplot(CPS85,
  mapping = aes(x = exper, y = wage, color = sex)) +
  geom_point(alpha = .7, size = 3) +
  geom_smooth(method = "lm", se = FALSE, size = 1.5)
```

The resulting plot is displayed in figure 4.12.

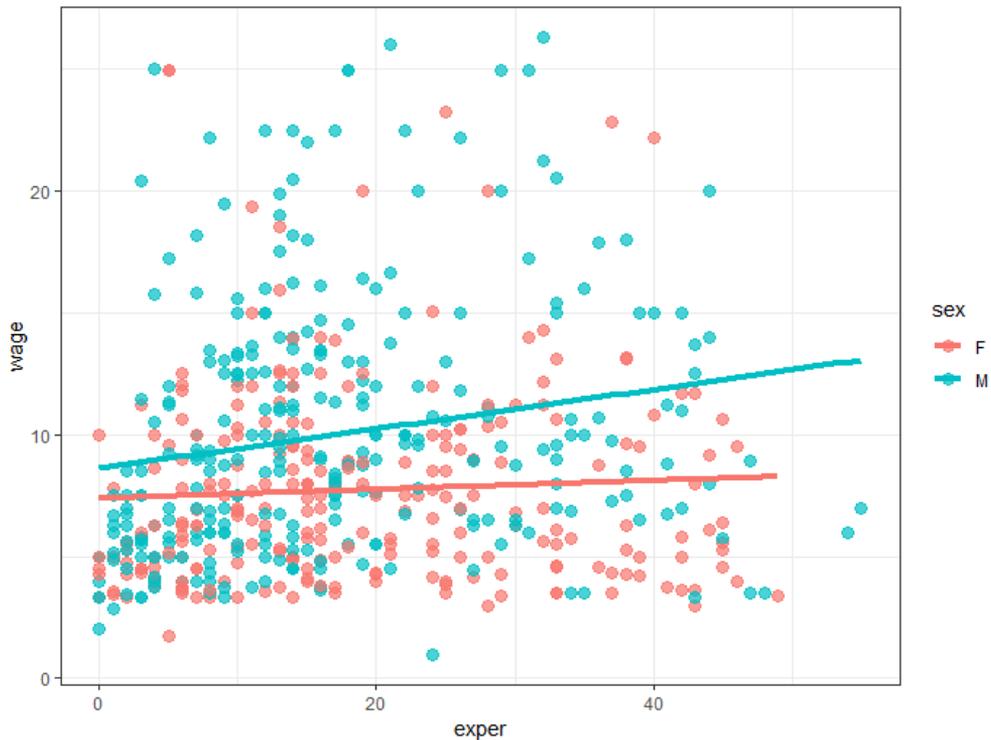


Figure 4.12. Scatterplot of experience and wage by sex, where `aes(color=sex)` is placed in the `ggplot()` function. The mapping is applied to both the `geom_point()` and `geom_smooth()`, producing separate point colors for males and females, along with separate lines of best fit.

Since the mapping of `sex` to color appears in the `ggplot()` function, it applies to both `geom_point` and `geom_smooth`. The color of the point indicates the sex, and a separate colored trend line is produced for men and women. Compare this to

```
ggplot(CPS85, aes(x = exper, y = wage)) +
  geom_point(aes(color = sex), alpha = .7, size = 3) +
  geom_smooth(method = "lm", se = FALSE, size = 1.5)
```

The resulting graph is given in figure 4.13.

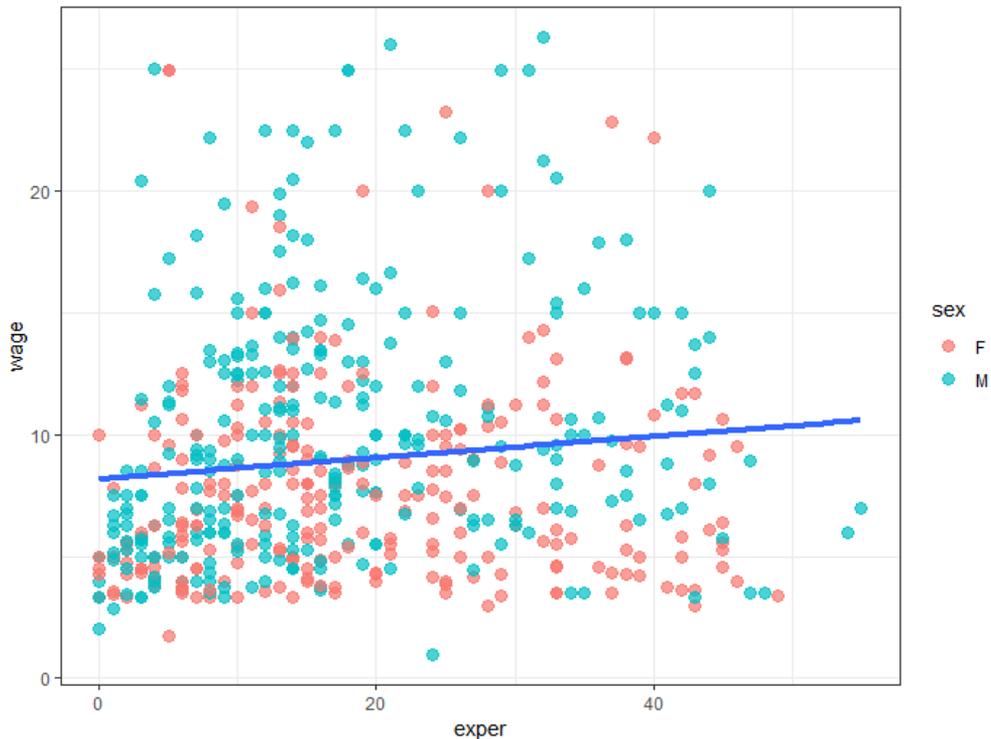


Figure 4.13 Scatterplot of experience and wage by sex, where `aes(color=sex)` is placed in the `geom_point()` function. The mapping is applied to point color producing separate point colors for men and women, but a single line of best fit for or all workers.

Since the sex to color mapping only appears in the `geom_point()` function, it is only used there. A single trend line is created for all observations.

Most of the examples in this book place the data and mapping options in the `ggplot` function. Additionally, the phrases `data=` and `mapping=` are omitted since the first option always refers to data and the second option always refers to mapping.

4.2.2 Graphs as objects

A `ggplot2` graph can be saved as a named R object (a list), manipulated further, and then printed or saved to disk. Consider the code in listing 4.1.

Listing 4.1 Using a `ggplot2` graph as an object

```
data(CPS85 , package = "mosaicData")      #A
CPS85 <- CPS85[CPS85$wage < 40,]          #A

myplot <- ggplot(data = CPS85,            #B
  aes(x = exper, y = wage)) +           #B
  geom_point()                          #B

myplot                                    #C

myplot2 <- myplot + geom_point(size = 3, color = "blue") #D
myplot2                                    #D

myplot + geom_smooth(method = "lm") +    #E
labs(title = "Mildly interesting graph") #E
```

#A Prepare data

#B Create a scatterplot and save it as myplot

#C Display myplot

#D Make the points larger and blue, save it as myplot2 and display the graph

#E Display myplot with a best fit line and a title

First the data are imported and outliers are removed #A. Then a simple scatter plot of experience vs. wages is created and saved as `myplot` #B. Next, the plot is printed #C. The plot is then modified by changing the point size and color, saved as `myplot2` and printed #D. Finally, the original plot is given a line of best fit and title, and printed #E. Note that these changes are not saved.

The ability to save graphs as objects allows you to continue to work with and modify them. This can be a real time saver (and help you avoid carpal tunnel syndrome). It is also handy when saving graphs programmatically, as we'll see in the next section.

4.2.3 Exporting graphs

You can export graphs created by `ggplot2` in a variety of image formats using the RStudio GUI or through your code. To export a graph using the RStudio menus, go to the Plots tab and choose Export (see figure 4.14).

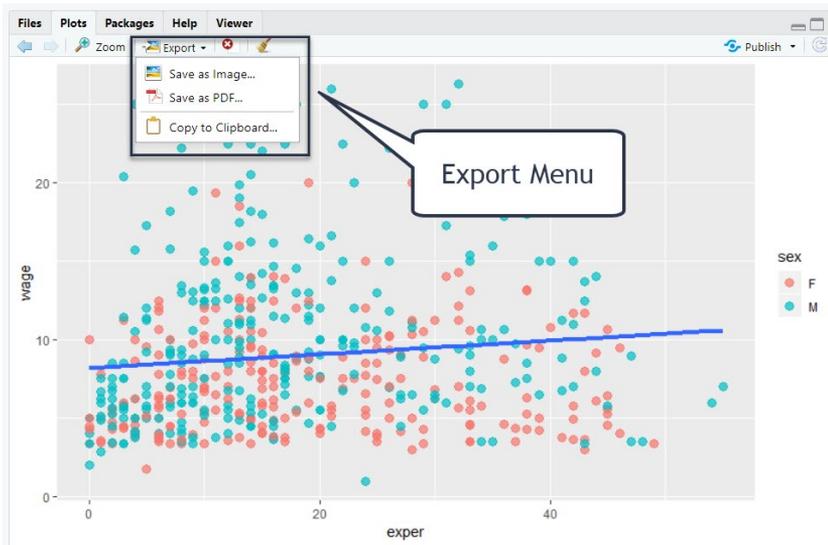


Figure 14.14 Saving a graph using the RStudio interface

To export a graph via code use the `ggsave()` function. You can specify the plot to save, its size and format, and where to save it. For example,

```
ggsave(file="mygraph.png", plot=myplot, width=5, height=4)
```

saves `myplot` as a 5-inch by 4-inch PNG file named `mygraph.png` in the current working directory. You can save the graph in a different format by changing the file extension. A description of the most common formats is provided in table 4.4.

Table 4.4 Image file formats

Extension	Format
pdf	Portable Document Format
jpeg	JPEG
tiff	Tagged Image File Format
png	Portable Network Graphics
svg	Scalable Vector Graphics
wmf	Windows Metafile

The `pdf`, `svg`, and `wmf` formats are lossless - they resize without fuzziness or pixilation. The other formats are lossy - they will pixelate when resized. This is especially noticeable when small images are enlarged. The `png` format is popular for images destined for webpages. The `jpeg` and `tif` formats are usually reserved for photographs.

The `wmf` format is usually recommended for graphs that will appear in Microsoft Word or PowerPoint documents. MS Office does not support `pdf` or `svg` files, and the `wmf` format will rescale well. However, note that `wmf` files will lose any transparency settings that have been set.

If you omit the `plot=` option, the most recently created graph is saved. The code

```
ggplot(data=mtcars, aes(x=mpg)) + geom_histogram()
ggsave(file="mygraph.pdf")
```

is valid and saves the graph to disk as a PDF document. See `help(ggsave)` for additional details.

4.2.4 Common mistakes

After working with `ggplot2` for years, I've found that there are two mistakes that are frequently made. The first is omitting or misplacing a closing parentheses. This happens most often following the `aes()` function. Consider the following code.

```
ggplot(CPS85, aes(x = exper, y = wage, color = sex) +
geom_point()
```

Note the lack of a closing parentheses at the end of the first line. I can't tell you how many times I've done this.

The second error is confusing an assignment for a mapping. The code

```
ggplot(CPS85, aes(x = exper, y = wage, color = "blue")) +
geom_point()
```

produces the graph in figure 14.8. The points are red (not blue) and there is a strange legend. What happened?

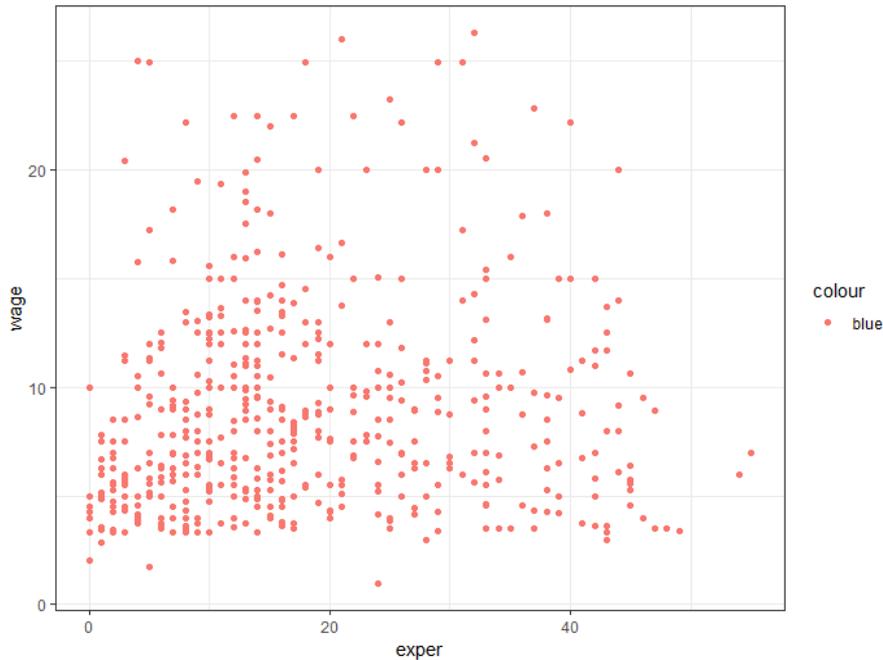


Figure 14.15 Placing an assignment statement in the aes function()

The `aes()` function is used to map *variables* to the visual characteristics of the graph. Assigning *constant* values is done outside the `aes()` function. The correct code would be

```
ggplot(CPS85, aes(x = exper, y = wage) +
  geom_point(color = "blue")
```

4.3 Summary

- The `ggplot2` package provides a powerful platform for creating both simple and complex graphs. Graphs are built up in layers using functions chained together with the plus (+) symbol.
- The `ggplot()` function specifies a data frame containing plot data and an `aes()` function that maps variables to visual aspects of the graph.
- `geom_` functions specify the geometric objects (bars, lines, points, etc.) to be placed on the graph.
- Optional `scale_` functions allow you to customize how a variable's values will be translated into their visual representations on the graph (e.g., the x- and y-axis scales and labels to used, and what colors, shapes, and line-types will be mapped to a variable's values).
- Data from two or more groups can be represented by grouping (superimposing plots

distinguished by visual aspects such as color) or faceting (placing several small plots in a matrix-like array).

- Two common errors of `ggplot` are missing/misplaced parentheses and confusing an assignment for a mapping.
- Graphs can be exported in a wide variety of image formats (such as tiff, pdf, jpg, png, svg, and wmf) using the RStudio GUI or the `ggsave()` function.

5

Advanced data management

This chapter covers

- **Mathematical and statistical functions**
- **Character functions**
- **Looping and conditional execution**
- **User-written functions**
- **Ways to aggregate and reshape data**

In chapter 3, we reviewed the basic techniques used for managing datasets in R. In this chapter, we'll focus on advanced topics. The chapter is divided into three basic parts. In the first part, we'll take a whirlwind tour of R's many functions for mathematical, statistical, and character manipulation. To give this section relevance, we begin with a data-management problem that can be solved using these functions. After covering the functions themselves, we'll look at one possible solution to the data-management problem.

Next, we cover how to write your own functions to accomplish data-management and -analysis tasks. First, we'll explore ways of controlling program flow, including looping and conditional statement execution. Then we'll investigate the structure of user-written functions and how to invoke them once created.

Then, we'll look at ways of aggregating and summarizing data, along with methods of reshaping and restructuring datasets. When aggregating data, you can specify the use of any appropriate built-in or user-written function to accomplish the summarization, so the topics you learn in the first two parts of the chapter will provide a real benefit.

5.1 A data-management challenge

To begin our discussion of numerical and character functions, let's consider a data-management problem. A group of students have taken exams in math, science, and English.

You want to combine these scores in order to determine a single performance indicator for each student. Additionally, you want to assign an A to the top 20% of students, a B to the next 20%, and so on. Finally, you want to sort the students alphabetically. The data are presented in table 5.1.

Table 5.1 Student exam data

Student	Math	Science	English
John Davis	502	95	25
Angela Williams	600	99	22
Bullwinkle Moose	412	80	18
David Jones	358	82	15
Janice Markhammer	495	75	20
Cheryl Cushing	512	85	28
Reuven Ytzhak	410	80	15
Greg Knox	625	95	30
Joel England	573	89	27
Mary Rayburn	522	86	18

Looking at this dataset, several obstacles are immediately evident. First, scores on the three exams aren't comparable. They have widely different means and standard deviations, so averaging them doesn't make sense. You must transform the exam scores into comparable units before combining them. Second, you'll need a method of determining a student's

percentile rank on this score in order to assign a grade. Third, there's a single field for names, complicating the task of sorting students. You'll need to split their names into first name and last name in order to sort them properly.

Each of these tasks can be accomplished through the judicious use of R's numerical and character functions. After working through the functions described in the next section, we'll consider a possible solution to this data-management challenge.

5.2 Numerical and character functions

In this section, we'll review functions in R that can be used as the basic building blocks for manipulating data. They can be divided into numerical (mathematical, statistical, probability) and character functions. After we review each type, I'll show you how to apply functions to the columns (variables) and rows (observations) of matrices and data frames (see section 5.2.6).

5.2.1 Mathematical functions

Table 5.2 lists common mathematical functions along with short examples.

Table 5.2 Mathematical functions

Function	Description
<code>abs(x)</code>	Absolute value <code>abs(-4)</code> returns 4.
<code>sqrt(x)</code>	Square root <code>sqrt(25)</code> returns 5. This is the same as $25^{(0.5)}$.
<code>ceiling(x)</code>	Smallest integer not less than x <code>ceiling(3.475)</code> returns 4.
<code>floor(x)</code>	Largest integer not greater than x <code>floor(3.475)</code> returns 3.
<code>trunc(x)</code>	Integer formed by truncating values in x toward 0

	<code>trunc(5.99)</code> returns 5.
<code>round(x, digits=n)</code>	Rounds x to the specified number of decimal places <code>round(3.475, digits=2)</code> returns 3.48.
<code>signif(x, digits=n)</code>	Rounds x to the specified number of significant digits <code>signif(3.475, digits=2)</code> returns 3.5.
<code>cos(x), sin(x), tan(x)</code>	Cosine, sine, and tangent <code>cos(2)</code> returns -0.416 .
<code>acos(x), asin(x), atan(x)</code>	Arc-cosine, arc-sine, and arc-tangent <code>acos(-0.416)</code> returns 2.
<code>cosh(x), sinh(x), tanh(x)</code>	Hyperbolic cosine, sine, and tangent <code>sinh(2)</code> returns 3.627.
<code>acosh(x), asinh(x), atanh(x)</code>	Hyperbolic arc-cosine, arc-sine, and arc-tangent <code>asinh(3.627)</code> returns 2.
<code>log(x, base=n)</code>	Logarithm of x to the base n
<code>log(x)</code>	For convenience:
<code>log10(x)</code>	<ul style="list-style-type: none"> • <code>log(x)</code> is the natural logarithm. • <code>log10(x)</code> is the common logarithm. • <code>log(10)</code> returns 2.3026. • <code>log10(10)</code> returns 1.
<code>exp(x)</code>	Exponential function <code>exp(2.3026)</code> returns 10.

Data transformation is one of the primary uses for these functions. For example, you often transform positively skewed variables such as income to a log scale before further analyses. Mathematical functions are also used as components in formulas, in plotting functions (for example, `x` versus `sin(x)`), and in formatting numerical values prior to printing.

The examples in table 5.2 apply mathematical functions to *scalars* (individual numbers). When these functions are applied to numeric vectors, matrices, or data frames, they operate on each individual value. For example, `sqrt(c(4, 16, 25))` returns `c(2, 4, 5)`.

5.2.2 Statistical functions

Common statistical functions are presented in table 5.3. Many of these functions have optional parameters that affect the outcome. For example,

```
y <- mean(x)
```

provides the arithmetic mean of the elements in object `x`, and

```
z <- mean(x, trim = 0.05, na.rm=TRUE)
```

provides the trimmed mean, dropping the highest and lowest 5% of scores and any missing values. Use the `help()` function to learn more about each function and its arguments.

Table 5.3 Statistical functions

Function	Description
<code>mean(x)</code>	Mean <code>mean(c(1, 2, 3, 4))</code> returns 2.5.
<code>median(x)</code>	Median <code>median(c(1, 2, 3, 4))</code> returns 2.5.
<code>sd(x)</code>	Standard deviation <code>sd(c(1, 2, 3, 4))</code> returns 1.29.
<code>var(x)</code>	Variance <code>var(c(1, 2, 3, 4))</code> returns 1.67.
<code>mad(x)</code>	Median absolute deviation

	<code>mad(c(1,2,3,4))</code> returns 1.48.
<code>quantile(x, probs)</code>	<p>Quantiles where <code>x</code> is the numeric vector, where quantiles are desired and <code>probs</code> is a numeric vector with probabilities in [0,1]</p> <pre># 30th and 84th percentiles of x y <- quantile(x, c(.3, .84))</pre>
<code>range(x)</code>	<p>Range</p> <pre>x <- c(1,2,3,4) range(x) returns c(1,4). diff(range(x)) returns 3.</pre>
<code>sum(x)</code>	<p>Sum</p> <pre>sum(c(1,2,3,4)) returns 10.</pre>
<code>diff(x, lag=n)</code>	<p>Lagged differences, with <code>lag</code> indicating which lag to use. The default lag is 1.</p> <pre>x<- c(1, 5, 23, 29) diff(x) returns c(4, 18, 6).</pre>
<code>min(x)</code>	<p>Minimum</p> <pre>min(c(1,2,3,4)) returns 1.</pre>
<code>max(x)</code>	<p>Maximum</p> <pre>max(c(1,2,3,4)) returns 4.</pre>
<code>scale(x, center=TRUE, scale=TRUE)</code>	<p>Column center (<code>center=TRUE</code>) or standardize (<code>center=TRUE, scale=TRUE</code>) data object <code>x</code>. An example is given in listing 5.6.</p>

To see these functions in action, look at the next listing. This example demonstrates two ways to calculate the mean and standard deviation of a vector of numbers. The data are the English scores in table 5.1.

Listing 5.1 Calculating the mean and standard deviation

```
> x <- c(25, 22, 18, 15, 20, 28, 15, 30, 27, 18)

> mean(x)           #A
[1] 21.8            #A
> sd(x)            #A
[1] 5.452828       #A

> n <- length(x)   #B
> meanx <- sum(x)/n #B
> css <- sum((x - meanx)^2) #B
> sdx <- sqrt(css / (n-1)) #B
> meanx            #B
[1] 21.8           #B
> sdx              #B
[1] 5.452828      #B
```

#A Short way

#B Long way

It's instructive to view how the corrected sum of squares (`css`) is calculated in the second approach:

1. `x` equals `c(25, 22, 18, 15, 20, 28, 15, 30, 27, 18)`, and `mean x` equals 21.8 (`length(x)` returns the number of elements in `x`).

2. `(x - meanx)` subtracts 4.5 from each element of `x`, resulting in

```
c(3.2, 0.2, -3.8, -6.8, -1.8, 6.2, -6.8, 8.2, 5.2, -3.8)
```

3. `(x - meanx)^2` squares each element of `(x - meanx)`, resulting in

```
c(10.24, 0.04, 14.44, 46.24, 3.24, 38.44, 46.24, 67.24, 27.04, 14.44)
```

4. `sum((x - meanx)^2)` sums each of the elements of `(x - meanx)^2`, resulting in 267.6.

Writing formulas in R has much in common with matrix-manipulation languages such as MATLAB (we'll look more specifically at solving matrix algebra problems in appendix D).

Standardizing data

By default, the `scale()` function standardizes the specified columns of a matrix or data frame to a mean of 0 and a standard deviation of 1:

```
newdata <- scale(mydata)
```

To standardize each column to an arbitrary mean and standard deviation, you can use code similar to the following

```
newdata <- scale(mydata)*SD + M
```

where M is the desired mean and SD is the desired standard deviation. Using the `scale()` function on non-numeric columns produces an error. To standardize a specific column rather than an entire matrix or data frame, you can use code such as this:

```
newdata <- transform(mydata, myvar = scale(myvar)*10+50)
```

This code standardizes the variable `myvar` to a mean of 50 and standard deviation of 10. You'll use the `scale()` function in the solution to the data-management challenge in section 5.3.

5.2.3 Probability functions

You may wonder why probability functions aren't listed with the statistical functions (it was really bothering you, wasn't it?). Although probability functions are statistical by definition, they're unique enough to deserve their own section. Probability functions are often used to generate simulated data with known characteristics and to calculate probability values within user-written statistical functions.

In R, probability functions take the form

```
[dpqr]distribution_abbreviation()
```

where the first letter refers to the aspect of the *distribution* returned:

- d = density
- p = distribution function
- q = quantile function
- r = random generation (random deviates)

The common probability functions are listed in table 5.4.

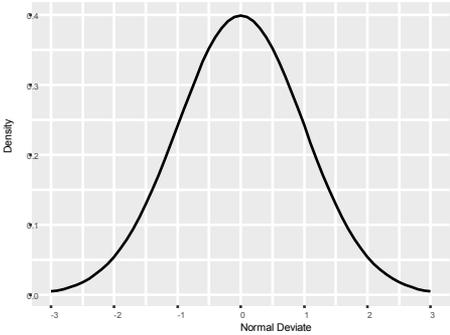
Table 5.4 Probability distributions

Distribution	Abbreviation	Distribution	Abbreviation
Beta	beta	Logistic	logis
Binomial	binom	Multinomial	multinom
Cauchy	cauchy	Negative binomial	nbinom
Chi-squared (noncentral)	chisq	Normal	norm

Exponential	exp	Poisson	pois
F	f	Wilcoxon signed rank	signrank
Gamma	gamma	T	t
Geometric	geom	Uniform	unif
Hypergeometric	hyper	Weibull	weibull
Lognormal	lnorm	Wilcoxon rank sum	wilcox

To see how these work, let's look at functions related to the normal distribution. If you don't specify a mean and a standard deviation, the standard normal distribution is assumed (mean=0, sd=1). Examples of the density (`dnorm`), distribution (`pnorm`), quantile (`qnorm`), and random deviate generation (`rnorm`) functions are given in table 5.5.

Table 5.5 Normal distribution functions

Problem	Solution
<p>Plot the standard normal curve on the interval $[-3, 3]$ (see figure).</p> 	<pre>library(ggplot2) x <- seq(from = -3, to = 3, by = 0.1) y = dnorm(x) data <- data.frame(x = x, y=y) ggplot(data, aes(x, y)) + geom_line() + labs(x = "Normal Deviate", y = "Density") + scale_x_continuous(breaks = seq(-3, 3, 1))</pre>
<p>What is the area under the standard normal curve to the left of $z=1.96$?</p>	<p><code>pnorm(1.96)</code> equals 0.975.</p>
<p>What is the value of the 90th percentile of a normal distribution with a mean of 500 and a standard deviation of 100?</p>	<p><code>qnorm(.9, mean=500, sd=100)</code> equals 628.16.</p>
<p>Generate 50 random normal deviates with a mean of 50 and a standard deviation of 10.</p>	<pre>rnorm(50, mean=50, sd=10)</pre>

SETTING THE SEED FOR RANDOM NUMBER GENERATION

Each time you generate pseudo-random deviates, a different seed, and therefore different results, are produced. To make your results reproducible, you can specify the seed explicitly, using the `set.seed()` function. An example is given in the next listing. Here, the `runif()` function is used to generate pseudo-random numbers from a uniform distribution on the interval 0 to 1.

Listing 5.2 Generating pseudo-random numbers from a uniform distribution

```
> runif(5)
[1] 0.8725344 0.3962501 0.6826534 0.3667821 0.9255909
> runif(5)
[1] 0.4273903 0.2641101 0.3550058 0.3233044 0.6584988
> set.seed(1234)
> runif(5)
[1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154
> set.seed(1234)
> runif(5)
[1] 0.1137034 0.6222994 0.6092747 0.6233794 0.8609154
```

By setting the seed manually, you're able to reproduce your results. This ability can be helpful in creating examples you can access in the future and share with others.

GENERATING MULTIVARIATE NORMAL DATA

In simulation research and Monte Carlo studies, you often want to draw data from a multivariate normal distribution with a given mean vector and covariance matrix. The `draw.d.variate.normal()` function in the `MultiRNG` package makes this easy. The function call is

```
draw.d.variate.normal(n, nvar, mean, sigma)
```

where n is the desired sample size, $nvar$ is the number of variables, $mean$ is the vector of means, and $sigma$ is the variance-covariance (or correlation) matrix. Listing 5.3 samples 500 observations from a three-variable multivariate normal distribution for which the following are true:

Mean vector	230.7	146.7	3.6
Covariance matrix	15360.8	6721.2	-47.1
	6721.2	4700.9	-16.5
	-47.1	-16.5	0.3

Listing 5.3 Generating data from a multivariate normal distribution

```
> library(MultiRNG)
> options(digits=3)
> set.seed(1234) #1

> mean <- c(230.7, 146.7, 3.6)
> sigma <- matrix(c(15360.8, 6721.2, -47.1,
                   6721.2, 4700.9, -16.5,
                   -47.1, -16.5, 0.3),
                 nrow=3, ncol=3)
#2
```

```

-47.1, -16.5, 0.3), nrow=3, ncol=3)
#2
> mydata <- draw.d.variate.normal(500, 3, mean, sigma) #3
> mydata <- as.data.frame(mydata) #3
> names(mydata) <- c("y", "x1", "x2") #3

> dim(mydata) #4
[1] 500 3 #4
> head(mydata, n=10) #4
  y x1 x2
1 81.1 122.6 3.69
2 265.1 110.4 3.49
3 365.1 235.3 2.67
4 -60.0 14.9 4.72
5 283.9 244.8 3.88
6 293.4 163.9 2.66
7 159.5 51.5 4.03
8 163.0 137.7 3.77
9 160.7 131.0 3.59
10 120.4 97.7 4.11

```

#1 Sets the random number seed
#2 Specifies the mean vector and covariance matrix
#3 Generates data
#4 Views the results

In listing 5.3, you set a random number seed so that you can reproduce the results at a later time #1. You specify the desired mean vector and variance-covariance matrix #2 and generate 500 pseudo-random observations #3. For convenience, the results are converted from a matrix to a data frame, and the variables are given names. Finally, you confirm that you have 500 observations and 3 variables, and you print out the first 10 observations #4. Note that because a correlation matrix is also a covariance matrix, you could have specified the correlation structure directly.

The MultiRNG package allows you to generate random data from 10 other multivariate distributions, including multivariate versions of the t, uniform, Bernoulli, hypergeometric, beta, multinomial, Laplace, and Wishart distributions.

The probability functions in R allow you to generate simulated data, sampled from distributions with known characteristics. Statistical methods that rely on simulated data have grown exponentially in recent years, and you'll see several examples of these in later chapters.

5.2.4 Character functions

Whereas mathematical and statistical functions operate on numerical data, character functions extract information from textual data or reformat textual data for printing and reporting. For example, you may want to concatenate a person's first name and last name, ensuring that the first letter of each is capitalized. Or you may want to count the instances of obscenities in open-ended feedback. Some of the most useful character functions are listed in table 5.6.

Table 5.6 Character functions

Function	Description
<code>nchar(x)</code>	Counts the number of characters of <i>x</i> . <code>x <- c("ab", "cde", "fghij")</code> <code>length(x)</code> returns 3 (see table 5.7). <code>nchar(x[3])</code> returns 5.
<code>substr(x, start, stop)</code>	Extracts or replaces substrings in a character vector. <code>x <- "abcdef"</code> <code>substr(x, 2, 4)</code> returns <code>bcd</code> . <code>substr(x, 2, 4) <- "22222"</code> (<i>x</i> is now <code>"a222ef"</code>).
<code>grep(pattern, x, ignore.case=FALSE, fixed=FALSE)</code>	Searches for <i>pattern</i> in <i>x</i> . If <code>fixed=FALSE</code> , then <i>pattern</i> is a regular expression. If <code>fixed=TRUE</code> , then <i>pattern</i> is a text string. Returns the matching indices. <code>grep("A", c("b", "A", "c"), fixed=TRUE)</code> returns 2.
<code>sub(pattern, replacement, x, ignore.case=FALSE, fixed=FALSE)</code>	Finds <i>pattern</i> in <i>x</i> and substitutes the <i>replacement</i> text. If <code>fixed=FALSE</code> , then <i>pattern</i> is a regular expression. If <code>fixed=TRUE</code> , then <i>pattern</i> is a text string. <code>sub("\\s", ".", "Hello There")</code> returns <code>Hello.There</code> . Note that <code>"\s"</code> is a regular expression for finding whitespace; use <code>"\\s"</code> instead, because <code>"\"</code> is R's escape character (see section 1.3.3).
<code>strsplit(x, split, fixed=FALSE)</code>	Splits the elements of character vector <i>x</i> at <i>split</i> . If <code>fixed=FALSE</code> , then <i>pattern</i> is a regular expression . If <code>fixed=TRUE</code> , then <i>pattern</i> is a text string. <code>y <- strsplit("abc", "")</code> returns a one-component, three-element list containing

	<pre>"a" "b" "c"</pre> <p><code>unlist(y)[2]</code> and <code>sapply(y, "[", 2)</code> both return "b".</p>
<code>paste(..., sep="")</code>	<p>Concatenates strings after using the <i>sep</i> string to separate them.</p> <p><code>paste("x", 1:3, sep="")</code> returns <code>c("x1", "x2", "x3")</code>.</p> <p><code>paste("x", 1:3, sep="M")</code> returns <code>c("xM1", "xM2", "xM3")</code>.</p> <p><code>paste("Today is", date())</code> returns</p> <pre>Today is Mon Dec 28 14:17:32 2015</pre> <p>(I changed the date to appear more current.)</p>
<code>toupper(x)</code>	<p>Uppercase.</p> <p><code>toupper("abc")</code> returns "ABC".</p>
<code>tolower(x)</code>	<p>Lowercase.</p> <p><code>tolower("ABC")</code> returns "abc".</p>

Note that the functions `grep()`, `sub()`, and `strsplit()` can search for a text string (`fixed=TRUE`) or a regular expression (`fixed=FALSE`); `FALSE` is the default. Regular expressions provide a clear and concise syntax for matching a pattern of text. For example, the regular expression

```
^[hc]?at
```

matches any string that starts with 0 or one occurrences of `h` or `c`, followed by `at`. The expression therefore matches `hat`, `cat`, and `at`, but not `bat`. To learn more, see the *regular expression* entry in Wikipedia. Helpful tutorials include Ryans Regular Expression Tutorial (<https://ryanstutorials.net/regular-expressions-tutorial/>) and an engaging interactive tutorial from RegexOne (<https://regexone.com>).

5.2.5 Other useful functions

The functions in table 5.7 are also quite useful for data-management and manipulation, but they don't fit cleanly into the other categories.

Table 5.7 Other useful functions

Function	Description
<code>length(x)</code>	Returns the length of object <i>x</i> . <pre>x <- c(2, 5, 6, 9)</pre> <code>length(x)</code> returns 4.
<code>seq(from, to, by)</code>	Generates a sequence. <pre>indices <- seq(1,10,2)</pre> <code>indices</code> is <code>c(1, 3, 5, 7, 9)</code> .
<code>rep(x, n)</code>	Repeats <i>x</i> <i>n</i> times. <pre>y <- rep(1:3, 2)</pre> <code>y</code> is <code>c(1, 2, 3, 1, 2, 3)</code> .
<code>cut(x, n, labels)</code>	Divides the continuous variable <i>x</i> into a factor with <i>n</i> levels, with optional <i>labels</i> . To create an ordered factor, include the option <code>ordered_result = TRUE</code> . <pre>x <- c(6, 2, 4, 3, 5, 1)</pre> <pre>xcat <- cut(x, 3, labels=c("low", "med", "high"))</pre> <code>xcat</code> is <code>c("high", "low", "med", "med", "high", "low")</code>
<code>cat(... , file = "myfile", append = FALSE)</code>	Concatenates the objects in ... and outputs them to the screen or to a file (if one is declared). <pre>name <- c("Jane")</pre> <pre>cat("Hello" , name, "\n")</pre>

The last example in the table demonstrates the use of escape characters in printing. Use `\n` for new lines, `\t` for tabs, `\'` for a single quote, `\b` for backspace, and so forth (type `?Quotes` for more information). For example, the code

```
name <- "Bob"
cat("Hello", name, "\b.\n", "Isn't R", "\t", "GREAT?\n")
```

produces

```
Hello Bob.
  Isn't R   GREAT?
```

Note that the second line is indented one space. When `cat` concatenates objects for output, it separates each by a space. That's why you include the backspace (`\b`) escape character before the period. Otherwise it would produce "Hello Bob ."

How you apply the functions covered so far to numbers, strings, and vectors is intuitive and straightforward, but how do you apply them to matrices and data frames? That's the subject of the next section.

5.2.6 Applying functions to matrices and data frames

One of the interesting features of R functions is that they can be applied to a variety of data objects (scalars, vectors, matrices, arrays, and data frames). The following listing provides an example.

Listing 5.4 Applying functions to data objects

```
> a <- 5
> sqrt(a)
[1] 2.236068
> b <- c(1.243, 5.654, 2.99)
> round(b)
[1] 1 6 3
> c <- matrix(runif(12), nrow=3)
> c
      [,1] [,2] [,3] [,4]
[1,] 0.4205 0.355 0.699 0.323
[2,] 0.0270 0.601 0.181 0.926
[3,] 0.6682 0.319 0.599 0.215
> log(c)
      [,1] [,2] [,3] [,4]
[1,] -0.866 -1.036 -0.358 -1.130
[2,] -3.614 -0.508 -1.711 -0.077
[3,] -0.403 -1.144 -0.513 -1.538
> mean(c)
[1] 0.444
```

Notice that the mean of matrix `c` in listing 5.4 results in a scalar (0.444). The `mean()` function takes the average of all 12 elements in the matrix. But what if you want the three row means or the four column means?

R provides a function, `apply()`, that allows you to `apply` an arbitrary function to any dimension of a matrix, array, or data frame. The format for the `apply()` function is

```
apply(x, MARGIN, FUN, ...)
```

where `x` is the data object, `MARGIN` is the dimension index, `FUN` is a function you specify, and `...` are any parameters you want to pass to `FUN`. In a matrix or data frame, `MARGIN=1` indicates rows and `MARGIN=2` indicates columns. Look at the following examples.

Listing 5.5 Applying a function to the rows (columns) of a matrix

```
> mydata <- matrix(rnorm(30), nrow=6)      #1
> mydata
  [,1] [,2] [,3] [,4] [,5]
[1,] 0.71298 1.368 -0.8320 -1.234 -0.790
[2,] -0.15096 -1.149 -1.0001 -0.725 0.506
[3,] -1.77770 0.519 -0.6675 0.721 -1.350
[4,] -0.00132 -0.308 0.9117 -1.391 1.558
[5,] -0.00543 0.378 -0.0906 -1.485 -0.350
[6,] -0.52178 -0.539 -1.7347 2.050 1.569
> apply(mydata, 1, mean)                #2
[1] -0.155 -0.504 -0.511 0.154 -0.310 0.165
> apply(mydata, 2, mean)                #3
[1] -0.2907 0.0449 -0.5688 -0.3442 0.1906
> apply(mydata, 2, mean, trim=0.2)      #4
[1] -0.1699 0.0127 -0.6475 -0.6575 0.2312
```

#1 Generates data

#2 Calculates the row means

#3 Calculates the column means

#4 Calculates the trimmed column means

You start by generating a 6×5 matrix containing random normal variates #1. Then you calculate the six row means #2 and five column means #3. Finally, you calculate the trimmed column means (in this case, means based on the middle 60% of the data, with the bottom 20% and top 20% of the values discarded) #4.

Because `FUN` can be any R function, including a function that you write yourself (see section 5.4), `apply()` is a powerful mechanism. Whereas `apply()` applies a function over the margins of an array, `lapply()` and `sapply()` apply a function over a list. You'll see an example of `sapply()` (which is a user-friendly version of `lapply()`) in the next section.

You now have all the tools you need to solve the data challenge presented in section 5.1, so let's give it a try.

5.3 A solution for the data-management challenge

Your challenge from section 5.1 is to combine subject test scores into a single performance indicator for each student, grade each student from A to F based on their relative standing

(top 20%, next 20%, and so on), and sort the roster by last name followed by first name. A solution is given in the following listing.

Listing 5.6 A solution to the learning example

```
> options(digits=2)          #1

> Student <- c("John Davis", "Angela Williams", "Bullwinkle Moose",
              "David Jones", "Janice Markhammer", "Cheryl Cushing",
              "Reuven Ytzrhak", "Greg Knox", "Joel England",
              "Mary Rayburn")
> Math <- c(502, 600, 412, 358, 495, 512, 410, 625, 573, 522)
> Science <- c(95, 99, 80, 82, 75, 85, 80, 95, 89, 86)
> English <- c(25, 22, 18, 15, 20, 28, 15, 30, 27, 18)
> roster <- data.frame(Student, Math, Science, English,
                       stringsAsFactors=FALSE)

> z <- scale(roster[,2:4])    #A #2
> score <- apply(z, 1, mean)  #A #3
> roster <- cbind(roster, score) #A #3

> y <- quantile(score, c(.8,.6,.4,.2)) #B #4
> roster$grade <- NA          #B #5
> roster$grade[score >= y[1]] <- "A" #B #5
> roster$grade[score < y[1] & score >= y[2]] <- "B" #B #5
> roster$grade[score < y[2] & score >= y[3]] <- "C" #B #5
> roster$grade[score < y[3] & score >= y[4]] <- "D" #B #5
> roster$grade[score < y[4]] <- "F" #B #5

> name <- strsplit((roster$Student), " ") #C #6
> Lastname <- sapply(name, "[", 2) #C #7
> Firstname <- sapply(name, "[", 1) #C #7
> roster <- cbind(Firstname, Lastname, roster[, -1]) #C #7

> roster <- roster[order(Lastname, Firstname),] #D #8

> roster
  Firstname Lastname Math Science English score grade
6 Cheryl Cushing 512 85 28 0.35 C
1 John Davis 502 95 25 0.56 B
9 Joel England 573 89 27 0.70 B
4 David Jones 358 82 15 -1.16 F
8 Greg Knox 625 95 30 1.34 A
5 Janice Markhammer 495 75 20 -0.63 D
3 Bullwinkle Moose 412 80 18 -0.86 D
10 Mary Rayburn 522 86 18 -0.18 C
2 Angela Williams 600 99 22 0.92 A
7 Reuven Ytzrhak 410 80 15 -1.05 F
```

#1 Step 1

#2 Step 2

#A Obtains the performance scores

#3 Step 3

#B Grades the students

#4 Step 4

- #5 Step 5
- #C Extracts the last and first names
- #6 Step 6
- #7 Step 7
- #D Sorts by last and first names
- #8 Step 8

The code is dense, so let's walk through the solution step by step.

#1 The original student roster is given. `options(digits=2)` limits the number of digits printed after the decimal place and makes the printouts easier to read:

```
> options(digits=2)
> roster
      Student Math Science English
1   John Davis 502  95  25
2  Angela Williams 600  99  22
3 Bullwinkle Moose 412  80  18
4   David Jones 358  82  15
5 Janice Markhammer 495  75  20
6 Cheryl Cushing 512  85  28
7 Reuven Ytzhak 410  80  15
8   Greg Knox 625  95  30
9   Joel England 573  89  27
10  Mary Rayburn 522  86  18
```

#2 Because the math, science, and English tests are reported on different scales (with widely differing means and standard deviations), you need to make them comparable before combining them. One way to do this is to standardize the variables so that each test is reported in standard-deviation units, rather than in their original scales. You can do this with the `scale()` function:

```
> z <- scale(roster[,2:4])
> z
      Math Science English
[1,] 0.013  1.078  0.587
[2,] 1.143  1.591  0.037
[3,] -1.026 -0.847 -0.697
[4,] -1.649 -0.590 -1.247
[5,] -0.068 -1.489 -0.330
[6,] 0.128 -0.205  1.137
[7,] -1.049 -0.847 -1.247
[8,] 1.432  1.078  1.504
[9,] 0.832  0.308  0.954
[10,] 0.243 -0.077 -0.697
```

#3 You can then get a performance score for each student by calculating the row means using the `mean()` function and adding them to the roster using the `cbind()` function:

```
> score <- apply(z, 1, mean)
> roster <- cbind(roster, score)
> roster
      Student Math Science English score
1   John Davis 502  95  25  1.180
2  Angela Williams 600  99  22  1.240
3 Bullwinkle Moose 412  80  18  0.773
4   David Jones 358  82  15  0.640
5 Janice Markhammer 495  75  20  0.940
6 Cheryl Cushing 512  85  28  1.187
7 Reuven Ytzhak 410  80  15  0.773
8   Greg Knox 625  95  30  1.287
9   Joel England 573  89  27  1.053
10  Mary Rayburn 522  86  18  0.773
```

```

1  John Davis 502  95  25 0.56
2  Angela Williams 600  99  22 0.92
3  Bullwinkle Moose 412  80  18 -0.86
4  David Jones 358  82  15 -1.16
5  Janice Markhammer 495  75  20 -0.63
6  Cheryl Cushing 512  85  28 0.35
7  Reuven Ytzhak 410  80  15 -1.05
8  Greg Knox 625  95  30 1.34
9  Joel England 573  89  27 0.70
10 Mary Rayburn 522  86  18 -0.18

```

#4 The `quantile()` function gives you the percentile rank of each student's performance score. You see that the cutoff for an A is 0.74, for a B is 0.44, and so on:

```

> y <- quantile(roster$score, c(.8,.6,.4,.2))
> y
80% 60% 40% 20%
0.74 0.44 -0.36 -0.89

```

#5 Using logical operators, you can recode students' percentile ranks into a new categorical grade variable. This code creates the variable `grade` in the `roster` data frame:

```

> roster$grade <- NA
> roster$grade[score >= y[1]] <- "A"
> roster$grade[score < y[1] & score >= y[2]] <- "B"
> roster$grade[score < y[2] & score >= y[3]] <- "C"
> roster$grade[score < y[3] & score >= y[4]] <- "D"
> roster$grade[score < y[4]] <- "F"
> roster
  Student Math Science English score grade
1  John Davis 502  95  25 0.56 B
2  Angela Williams 600  99  22 0.92 A
3  Bullwinkle Moose 412  80  18 -0.86 D
4  David Jones 358  82  15 -1.16 F
5  Janice Markhammer 495  75  20 -0.63 D
6  Cheryl Cushing 512  85  28 0.35 C
7  Reuven Ytzhak 410  80  15 -1.05 F
8  Greg Knox 625  95  30 1.34 A
9  Joel England 573  89  27 0.70 B
10 Mary Rayburn 522  86  18 -0.18 C

```

#6 You use the `strsplit()` function to break the student names into first name and last name at the space character. Applying `strsplit()` to a vector of strings returns a list:

```

> name <- strsplit((roster$Student), " ")
> name

[[1]]
[1] "John" "Davis"

[[2]]
[1] "Angela" "Williams"

```

```

[[3]]
[1] "Bullwinkle" "Moose"

[[4]]
[1] "David" "Jones"

[[5]]
[1] "Janice" "Markhammer"

[[6]]
[1] "Cheryl" "Cushing"

[[7]]
[1] "Reuven" "Ytzrhak"

[[8]]
[1] "Greg" "Knox"

[[9]]
[1] "Joel" "England"

[[10]]
[1] "Mary" "Rayburn"

```

#7 You use the `sapply()` function to take the first element of each component and put it in a `Firstname` vector, and the second element of each component and put it in a `Lastname` vector. `"["` is a function that extracts part of an object—here the first or second component of the list `name`. You use `cbind()` to add these elements to the roster. Because you no longer need the `student` variable, you drop it (with the `-1` in the roster index):

```

> Firstname <- sapply(name, "[", 1)
> Lastname <- sapply(name, "[", 2)
> roster <- cbind(Firstname, Lastname, roster[,-1])
> roster
  Firstname Lastname Math Science English score grade
1   John   Davis  502   95   25 0.56   B
2  Angela Williams  600   99   22 0.92   A
3 Bullwinkle  Moose  412   80   18 -0.86   D
4   David   Jones  358   82   15 -1.16   F
5  Janice Markhammer 495   75   20 -0.63   D
6  Cheryl  Cushing  512   85   28 0.35   C
7  Reuven  Ytzrhak  410   80   15 -1.05   F
8   Greg   Knox   625   95   30 1.34   A
9   Joel  England  573   89   27 0.70   B
10  Mary   Rayburn  522   86   18 -0.18   C

```

#8 Finally, you sort the dataset by first and last name using the `order()` function:

```

> roster[order(Lastname,Firstname),]
  Firstname Lastname Math Science English score grade
6  Cheryl  Cushing  512   85   28 0.35   C
1   John   Davis  502   95   25 0.56   B
9   Joel  England  573   89   27 0.70   B
4   David   Jones  358   82   15 -1.16   F

```

```

8 Greg Knox 625 95 30 1.34 A
5 Janice Markhammer 495 75 20 -0.63 D
3 Bullwinkle Moose 412 80 18 -0.86 D
10 Mary Rayburn 522 86 18 -0.18 C
2 Angela Williams 600 99 22 0.92 A
7 Reuven Ytzrhak 410 80 15 -1.05 F

```

Voilà! Piece of cake!

There are many other ways to accomplish these tasks, but this code helps capture the flavor of these functions. Now it's time to look at control structures and user-written functions.

5.4 Control flow

In the normal course of events, the statements in an R program are executed sequentially from the top of the program to the bottom. But there are times that you'll want to execute some statements repetitively while executing other statements only if certain conditions are met. This is where control-flow constructs come in.

R has the standard control structures you'd expect to see in a modern programming language. First we'll go through the constructs used for conditional execution, followed by the constructs used for looping.

For the syntax examples throughout this section, keep the following in mind:

- *statement* is a single R statement or a compound statement (a group of R statements enclosed in curly braces `{ }` and separated by semicolons).
- *cond* is an expression that resolves to `TRUE` or `FALSE`.
- *expr* is a statement that evaluates to a number or character string.
- *seq* is a sequence of numbers or character strings.

After we discuss control-flow constructs, you'll learn how to write your own functions.

5.4.1 Repetition and looping

Looping constructs repetitively execute a statement or series of statements until a condition isn't true. These include the `for` and `while` structures.

FOR

The `for` loop executes a statement repetitively until a variable's value is no longer contained in the sequence `seq`. The syntax is

```
for (var in seq) statement
```

In this example

```
for (i in 1:10) print("Hello")
```

the word *Hello* is printed 10 times.

WHILE

A `while` loop executes a statement repetitively until the condition is no longer true. The syntax is

```
while (cond) statement
```

In a second example, the code

```
i <- 10
while (i > 0) {print("Hello"); i <- i - 1}
```

once again prints the word *Hello* 10 times. Make sure the statements inside the brackets modify the `while` condition so that sooner or later it's no longer true—otherwise the loop will never end! In the previous example, the statement

```
i <- i - 1
```

subtracts 1 from object `i` on each loop, so that after the tenth loop it's no longer larger than 0. If you instead added 1 on each loop, R would never stop saying hello. This is why `while` loops can be more dangerous than other looping constructs.

Looping in R can be inefficient and time consuming when you're processing the rows or columns of large datasets. Whenever possible, it's better to use R's built-in numerical and character functions in conjunction with the `apply` family of functions.

5.4.2 Conditional execution

In conditional execution, a statement or statements are executed only if a specified condition is met. These constructs include `if-else`, `ifelse`, and `switch`.

IF-ELSE

The `if-else` control structure executes a statement if a given condition is true. Optionally, a different statement is executed if the condition is false. The syntax is

```
if (cond) statement
if (cond) statement1 else statement2
```

Here are some examples:

```
if (is.character(grade)) grade <- as.factor(grade)
if (!is.factor(grade)) grade <- as.factor(grade) else print("Grade already
is a factor")
```

In the first instance, if `grade` is a character vector, it's converted into a factor. In the second instance, one of two statements is executed. If `grade` isn't a factor (note the `!` symbol), it's turned into one. If it's a factor, then the message is printed.

IFELSE

The `ifelse` construct is a compact and vectorized version of the `if-else` construct. The syntax is

```
ifelse(cond, statement1, statement2)
```

The first statement is executed if `cond` is `TRUE`. If `cond` is `FALSE`, the second statement is executed. Here are some examples:

```
ifelse(score > 0.5, print("Passed"), print("Failed"))
outcome <- ifelse(score > 0.5, "Passed", "Failed")
```

Use `ifelse` when you want to take a binary action or when you want to input and output vectors from the construct.

SWITCH

`switch` chooses statements based on the value of an expression. The syntax is

```
switch(expr, ...)
```

where `...` represents statements tied to the possible outcome values of `expr`. It's easiest to understand how `switch` works by looking at the example in the following listing.

Listing 5.7 A `switch` example

```
> feelings <- c("sad", "afraid")
> for (i in feelings)
  print(
    switch(i,
      happy = "I am glad you are happy",
      afraid = "There is nothing to fear",
      sad = "Cheer up",
      angry = "Calm down now"
    )
  )
```

```
[1] "Cheer up"
[1] "There is nothing to fear"
```

This is a silly example, but it shows the main features. You'll learn how to use `switch` in user-written functions in the next section.

5.5 User-written functions

One of R's greatest strengths is the user's ability to add functions. In fact, many of the functions in R are functions of existing functions. The structure of a function looks like this:

```
myfunction <- function(arg1, arg2, ...){
  statements
  return(object)
}
```

Objects in the function are local to the function. The object returned can be any data type, from scalar to list. Let's look at an example.

Say you'd like to have a function that calculates the central tendency and spread of data objects. The function should give you a choice between parametric (mean and standard deviation) and nonparametric (median and median absolute deviation) statistics. The results should be returned as a named list. Additionally, the user should have the choice of automatically printing the results or not. Unless otherwise specified, the function's default behavior should be to calculate parametric statistics and not print the results. One solution is given in the following listing.

Listing 5.8 `mystats()` : a user-written function for summary statistics

```
mystats <- function(x, parametric=TRUE, print=FALSE) {
  if (parametric) {
    center <- mean(x); spread <- sd(x)
  } else {
    center <- median(x); spread <- mad(x)
  }
  if (print & parametric) {
    cat("Mean=", center, "\n", "SD=", spread, "\n")
  } else if (print & !parametric) {
    cat("Median=", center, "\n", "MAD=", spread, "\n")
  }
  result <- list(center=center, spread=spread)
  return(result)
}
```

To see this function in action, first generate some data (a random sample of size 500 from a normal distribution):

```
set.seed(1234)
x <- rnorm(500)
```

After executing the statement

```
y <- mystats(x)
```

`y$center` contains the mean (0.00184) and `y$spread` contains the standard deviation (1.03). No output is produced. If you execute the statement

```
y <- mystats(x, parametric=FALSE, print=TRUE)
```

`y$center` contains the median (-0.0207) and `y$spread` contains the median absolute deviation (1.001). In addition, the following output is produced:

```
Median= -0.0207
MAD= 1
```

Next, let's look at a user-written function that uses the `switch` construct. This function gives the user a choice regarding the format of today's date. Values that are assigned to parameters

in the function declaration are taken as defaults. In the `mydate()` function, `long` is the default format for dates if `type` isn't specified:

```
mydate <- function(type="long") {
  switch(type,
    long = format(Sys.time(), "%A %B %d %Y"),
    short = format(Sys.time(), "%m-%d-%y"),
    cat(type, "is not a recognized type\n")
  )
}
```

Here's the function in action:

```
> mydate("long")
[1] "Monday July 14 2014"
> mydate("short")
[1] "07-14-14"
> mydate()
[1] "Monday July 14 2014"
> mydate("medium")
medium is not a recognized type
```

Note that the `cat()` function is executed only if the entered type doesn't match "long" or "short". It's usually a good idea to have an expression that catches user-supplied arguments that have been entered incorrectly.

Several functions are available that can help add error trapping and correction to your functions. You can use the function `warning()` to generate a warning message, `message()` to generate a diagnostic message, and `stop()` to stop execution of the current expression and carry out an error action. Error trapping and debugging are discussed more fully in section 20.5.

After creating your own functions, you may want to make them available in every session. Appendix B describes how to customize the R environment so that user-written functions are loaded automatically at startup. We'll look at additional examples of user-written functions in chapters 6 and 8.

You can accomplish a great deal using the basic techniques provided in this section. Control flow and other programming topics are covered in greater detail in chapter 20. Creating a package is covered in chapter 21. If you'd like to explore the subtleties of function writing, or you want to write professional-level code that you can distribute to others, I recommend reading these two chapters and then reviewing three excellent books that you'll find in the References section at the end of this book: Venables & Ripley (2000), Chambers (2008), and Wickham(2019). Together, they provide a significant level of detail and breadth of examples.

Now that we've covered user-written functions, we'll end this chapter with a discussion of data aggregation and reshaping.

5.6 Reshaping data

When you *reshape* data, you alter the structure (rows and columns) determining how the data is organized. The three most common reshaping tasks are: (1) transposing a dataset; (2) converting a wide dataset to a long dataset; and (3) converting a long dataset to a wide dataset. Each is described in the following sections.

5.6.1 Transpose

Transposing (reversing rows and columns) is perhaps the simplest method of reshaping a dataset. Use the `t()` function to transpose a matrix or a data frame. In the latter case, the data frame is converted to a matrix first and row names become variable (column) names.

We'll illustrate the transpose using the `mtcars` data frame that's included with the base installation of R. This dataset, extracted from *Motor Trend* magazine (1974), describes the design and performance characteristics (number of cylinders, displacement, horsepower, mpg, and so on) for 34 automobiles. To learn more about the dataset, see `help(mtcars)`.

An example of the transpose operation is given in listing 5.9. A subset of the dataset is used in order to conserve space on the page.

Listing 5.9 Transposing a dataset

```
> cars <- mtcars[1:5,1:4]
> cars
      mpg cyl disp  hp
Mazda RX4    21.0  6 160 110
Mazda RX4 Wag 21.0  6 160 110
Datsun 710    22.8  4 108  93
Hornet 4 Drive 21.4  6 258 110
Hornet Sportabout 18.7  8 360 175
> t(cars)
      Mazda RX4 Mazda RX4 Wag Datsun 710 Hornet 4 Drive Hornet Sportabout
mpg    21      21    22.8      21.4      18.7
cyl     6       6     4.0       6.0       8.0
disp   160     160   108.0     258.0     360.0
hp     110     110   93.0     110.0     175.0
```

The `t()` function always returns a matrix. Since a matrix can only have one type (numeric, character, or logical), the transpose operation works best when all the variables in the original dataset are numeric or logical. If there are any character variables in the dataset, *the entire* dataset will be converted to character values in the resulting transpose.

5.6.2 Converting between wide to long dataset formats

A rectangular dataset is typically in either wide or long format. In *wide format*, each row represents a unique *observation*. An example is given in table 5.8. The table contains the life expectancy estimates for 4 countries in 1990, 2000, and 2010. It is part of a much larger

dataset obtained from Our World in Data (<https://ourworldindata.org/life-expectancy>). Note that each row represent the data gathered on a country.

Table 5.8 Life expectancy by year and country – wide format

ID	Country	LExp1990	LExp2000	LExp2010
AU	Australia	76.9	79.6	82.0
CN	China	69.3	72.0	75.2
PRK	North Korea	69.9	65.3	69.6

In *long format*, each row represents a unique *measurement*. An example with the same data in long format is given if table 5.9.

Table 5.9 Life expectancy by year and country – long format

ID	Country	Variable	LifeExp
AU	Australia	LExp1990	76.9
CN	China	LExp1990	69.3
PRK	North Korea	LExp1990	69.9
AU	Australia	LExp2000	79.6
CN	China	LExp2000	72.0
PRK	North Korea	LExp2000	65.3
AU	Australia	LExp2010	82.0
CN	China	LExp2010	75.2
PRK	North Korea	LExp2010	69.6

Different types of data analysis can require different data formats. For example, if you want to identify countries that have similar life expectancy trends over time, you could use cluster analysis (chapter 8). Cluster analysis requires data that are in wide format. On the other hand, you may want to predict life expectancy from country and year using multiple regression (chapter 8). In this case, the data would have to be in long format.

While most R functions expect wide format data frames, some require the data to be in a long format. Fortunately, the `tidyr` package provides functions that can easily convert data frames from one format to the other. Use `install.packages("tidyr")` to install the package before continuing.

The `gather()` function in the `tidyr` package converts a wide format data frame to a long format data frame. The syntax is

```
longdata <- gather(widedata, key, value, variable list)
```

where

- *widedata* is the data frame to be converted
- *key* specifies the name to be used for the variable column ("Variable" in this example)
- *value* specifies the name to be used for the value column ("LifeExp" in this example)
- *variable list* specifies the variables to be stacked (LExp1990, LExp2000, LExp2010 in this example)

An example is given in listing 5.10

Listing 5.10 Converting a wide format data frame to a long format

```
> library(tidyr)

> data_wide <- data.frame(ID = c("AU", "CN", "PRK"),
  Country = c("Australia", "China", "North Korea"),
  LExp1990 = c(76.9, 69.3, 69.9),
  LExp2000 = c(79.6, 72.0, 65.3),
  LExp2010 = c(82.0, 75.2, 69.6))

> data_wide
  ID Country LExp1990 LExp2000 LExp2010
1 AU Australia  76.9   79.6   82.0
2 CN  China    69.3   72.0   75.2
3 PRK North Korea 69.9   65.3   69.6
>
>
> data_long <- gather(data_wide, key="Variable", value="Life_Exp",
  c(LExp1990, LExp2000, LExp2010))
> data_long
  ID Country Variable Life_Exp
1 AU Australia LExp1990  76.9
2 CN  China    LExp1990  69.3
3 PRK North Korea LExp1990 69.9
4 AU Australia LExp2000  79.6
5 CN  China    LExp2000  72.0
6 PRK North Korea LExp2000 65.3
7 AU Australia LExp2010  82.0
8 CN  China    LExp2010  75.2
9 PRK North Korea LExp2010 69.6
```

The `spread()` function in the `tidyr` package converts a long format data frame to a wide format data frame. The format is

```
widedata <- spread(longdata, key, value)
```

where

- *longdata* is the data frame to be converted
- *key* is the column containing the variable names
- *value* is the column containing the variable values

Continuing the example, the code in listing 5.11 is used to convert the long format data frame back to a wide format.

Listing 5.11 Converting a long format data frame to a wide format

```
> data_wide <- spread(data_long, key=Variable, value=Life_Exp)
> data_wide
  ID Country LExp1990 LExp2000 LExp2010
1 AU Australia  76.9   79.6   82.0
2 CN  China    69.3   72.0   75.2
3 PRK North Korea 69.9   65.3   69.6
```

To learn more about the long and wide data formats, see Simon Ejdemyr's excellent tutorial (<https://sejdemyr.github.io/r-tutorials/basics/wide-and-long/>).

5.7 Aggregating data

When you *aggregate* data, you replace groups of observations with summary statistics based on those observations. Data aggregation can be a precursor to statistical analyses or a method of summarizing data for presentation in tables or graphs.

It's relatively easy to collapse data in R using one or more *by* variables and a defined function. In base R, the `aggregate()` is typically used. The format is

```
aggregate(x, by, FUN)
```

where *x* is the data object to be collapsed, *by* is a list of variables that will be crossed to form the new observations, and *FUN* is a function used to calculate the summary statistics that will make up the new observation values. The *by* variables must be enclosed in a list (even if there's only one).

As an example, let's aggregate the `mtcars` data by number of cylinders and gears, returning means for each of the numeric variables.

Listing 5.12 Aggregating data with the `aggregate()` function

```
> options(digits=3)
> aggdata <- aggregate(mtcars,
  by=list(mtcars$cyl,mtcars$gear),
  FUN=mean, na.rm=TRUE)
> aggdata
  Group.1 Group.2 mpg cyl disp hp drat wt qsec vs am gear carb
1 4 3 21.5 4 120 97 3.70 2.46 20.0 1.0 0.00 3 1.00
2 6 3 19.8 6 242 108 2.92 3.34 19.8 1.0 0.00 3 1.00
3 8 3 15.1 8 358 194 3.12 4.10 17.1 0.0 0.00 3 3.08
4 4 4 26.9 4 103 76 4.11 2.38 19.6 1.0 0.75 4 1.50
```

```

5 6 4 19.8 6 164 116 3.91 3.09 17.7 0.5 0.50 4 4.00
6 4 5 28.2 4 108 102 4.10 1.83 16.8 0.5 1.00 5 2.00
7 6 5 19.7 6 145 175 3.62 2.77 15.5 0.0 1.00 5 6.00
8 8 5 15.4 8 326 300 3.88 3.37 14.6 0.0 1.00 5 6.00

```

In these results, `Group.1` represents the number of cylinders (4, 6, or 8), and `Group.2` represents the number of gears (3, 4, or 5). For example, cars with 4 cylinders and 3 gears have a mean of 21.5 miles per gallon (mpg). Here we used the `mean` function, but any function in R or any user defined function that computes summary statistics can be used.

There are two limitations to this code. First, `Group.1` and `Group.2` are terribly uninformative variable names. Second, the original `cyl` and `gear` variables are included in the aggregated data frame. These columns are now redundant.

You can declare custom names for the grouping variables from within the list. For instance, `by=list(Cylinders=mtcars$cyl, Gears=mtcars$gear)` will replace `Group.1` and `Group.2` with `Cylinders` and `Gears`. The redundant columns can be dropped from the input data frame using bracket notation (`mtcars[-c(2, 10)]`). An improved version is given in listing 5.13.

Listing 5.13 Improved code for aggregating data with `aggregate()`

```

> aggdata <- aggregate(mtcars[-c(2, 10)],
  by=list(Cylinders=mtcars$cyl, Gears=mtcars$gear),
  FUN=mean, na.rm=TRUE)
> aggdata
  Cylinders Gears mpg disp hp drat wt qsec vs am carb
1 4 3 21.5 120 97 3.70 2.46 20.0 1.0 0.00 1.00
2 6 3 19.8 242 108 2.92 3.34 19.8 1.0 0.00 1.00
3 8 3 15.1 358 194 3.12 4.10 17.1 0.0 0.00 3.08
4 4 4 26.9 103 76 4.11 2.38 19.6 1.0 0.75 1.50
5 6 4 19.8 164 116 3.91 3.09 17.7 0.5 0.50 4.00
6 4 5 28.2 108 102 4.10 1.83 16.8 0.5 1.00 2.00
7 6 5 19.7 145 175 3.62 2.77 15.5 0.0 1.00 6.00
8 8 5 15.4 326 300 3.88 3.37 14.6 0.0 1.00 6.00

```

The `dplyr` package provides a more natural method of aggregating data. Consider the code in listing 5.14.

Listing 5.14 Aggregating data with the `dplyr` package

```

> mtcars %>%
  group_by(cyl, gear) %>%
  summarise_all(list(mean), na.rm=TRUE)

# A tibble: 8 x 11
# Groups: cyl [3]
  cyl gear mpg disp hp drat wt qsec vs am carb
<dbl> <dbl>
1 4 3 21.5 120. 97 3.7 2.46 20.0 1 0 1
2 4 4 26.9 103. 76 4.11 2.38 19.6 1 0.75 1.5
3 4 5 28.2 108. 102 4.1 1.83 16.8 0.5 1 2
4 6 3 19.8 242. 108. 2.92 3.34 19.8 1 0 1

```

```

5 6 4 19.8 164. 116. 3.91 3.09 17.7 0.5 0.5 4
6 6 5 19.7 145 175 3.62 2.77 15.5 0 1 6
7 8 3 15.0 358. 194. 3.12 4.10 17.1 0 0 3.08
8 8 5 15.4 326 300. 3.88 3.37 14.6 0 1 6

```

The grouping variables retain their names, and are not duplicated in the data. We'll expand on `dplyr`'s powerful summarization capabilities when discussing summary statistics in chapter 7.

Now that you've gathered the tools you need to get your data into shape (no pun intended), you're ready to bid part 1 goodbye and enter the exciting world of data analysis! In upcoming chapters, we'll begin to explore the many statistical and graphical methods available for turning data into information.

5.8 Summary

- Base R contains hundreds of mathematical, statistical, and probability functions that are useful for manipulating data. They can be applied to a wide range of data objects including vectors, matrices, and data frames.
- Functions for conditional execution and looping allow you to execute some statements repetitively and execute other statements only when certain conditions are met.
- You can easily write your own functions, vastly increasing the power of your programs.
- Data often have to be aggregated and/or restructured before further analyses are possible.

6

Basic graphs

This chapter covers

- **Bar, box, and dot plots**
- **Pie charts and tree maps**
- **Histograms and kernel density plots**

Whenever we analyze data, the first thing we should do is *look* at it. For each variable, what are the most common values? How much variability is present? Are there any unusual observations? R provides a wealth of functions for visualizing data. In this chapter, we'll look at graphs that help you understand a single categorical or continuous variable. This topic includes

- Visualizing the distribution of a variable
- Comparing the distribution of a variable across two or more groups

In both cases, the variable can be continuous (for example, car mileage as miles per gallon) or categorical (for example, treatment outcome as none, some, or marked). In later chapters, we'll explore graphs that display more complex relationships among variables.

The following sections explore the use of bar charts, pie charts, tree maps, histograms, kernel density plots, box plots, violin plots, and dot plots. Some of these may be familiar to you, whereas others (such as tree charts or violin plots) may be new to you. The goal, as always, is to understand your data better and to communicate this understanding to others. Let's start with bar charts.

6.1 Bar charts

A bar plot displays the distribution (frequency) of a categorical variable through vertical or horizontal bars. Using the `ggplot2` package, we can create a bar chart using the code

```
ggplot(data, aes(x=catvar) + geom_bar()
```

where *data* is a data frame and *catvar* is a categorical variable.

In the following examples, you'll plot the outcome of a study investigating a new treatment for rheumatoid arthritis. The data are contained in the `Arthritis` data frame distributed with the `vcd` package. This package isn't included in the default R installation, so install it before first use (`install.packages("vcd")`). Note that the `vcd` package isn't needed to create bar charts. You're installing it in order to gain access to the `Arthritis` dataset.

6.1.1 Simple bar charts

In the `Arthritis` study, the variable `Improved` records the patient outcomes for individuals receiving a placebo or drug:

```
> data(Arthritis, package="vcd")
> table(Arthritis$Improved)

None  Some Marked
 42   14   28
```

Here, you see that 28 patients showed marked improvement, 14 showed some improvement, and 42 showed no improvement. We'll discuss the use of the `table()` function to obtain cell counts more fully in chapter 7.

You can graph these counts using a vertical or horizontal bar chart. The code is provided in the following listing, and the resulting graphs are displayed in figure 6.1.

Listing 6.1 Simple bar charts

```
library(ggplot2)
ggplot(Arthritis, aes(x=Improved)) + geom_bar() + #A
  labs(title="Simple Bar chart", #A
        x="Improvement", #A
        y="Frequency") #A

ggplot(Arthritis, aes(x=Improved)) + geom_bar() + #B
  labs(title="Horizontal Bar chart", #B
        x="Improvement", #B
        y="Frequency") + #B
  coord_flip() #B
```

#A Simple bar chart

#B Horizontal bar chart

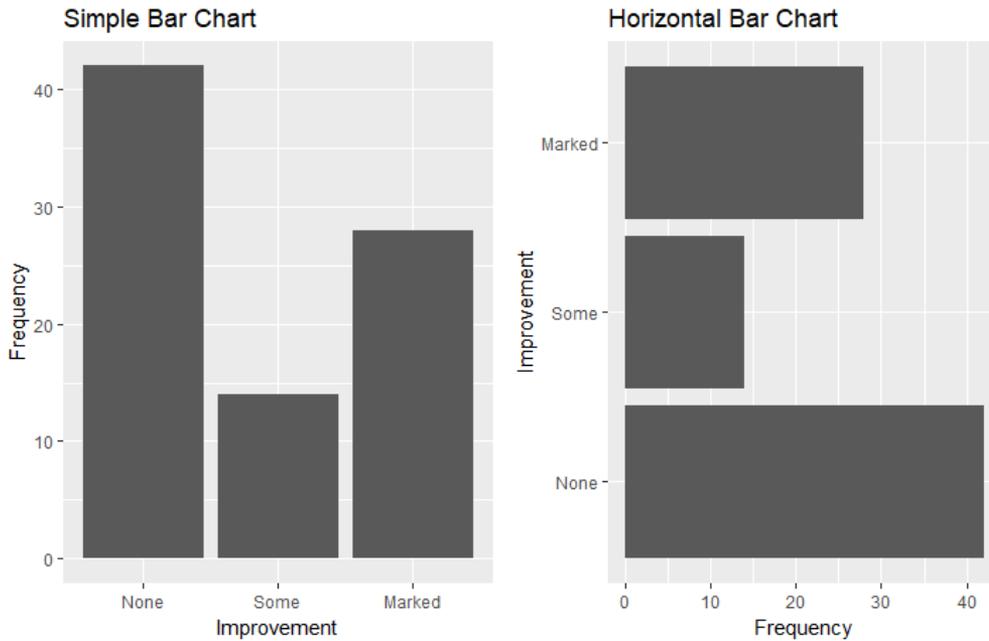


Figure 6.1 Simple vertical and horizontal bar charts

What happens if you have long labels? In section 6.1.4, you'll see how to tweak labels so that they don't overlap.

6.1.2 Stacked, grouped and filled bar charts

The central question in the Arthritis study is "How does the level of improvement vary between the placebo and treated conditions?". The `table()` function can be used to generate a cross-tabulation of the variables.

```
> table(Arthritis$Improved, Arthritis$Treatment)
```

	Treatment	
Improved	Placebo	Treated
None	29	13
Some	7	7
Marked	7	21

While the tabulation is helpful, the results are easier to grasp with a bar chart. The relationship between two categorical variables can be plotted using *stacked*, *grouped*, or *filled* bar charts. The code is provided in listing 6.2 and the graph is displayed in figures 6.2.

Listing 6.2 Stacked, grouped, and filled bar charts

```

library(ggplot2)
ggplot(Arthritis, aes(x=Treatment, fill=Improved)) + #A
  geom_bar(position = "stack") + #A
  labs(title="Stacked Bar chart", #A
        x="Treatment", #A
        y="Frequency") #A

ggplot(Arthritis, aes(x=Treatment, fill=Improved)) + #B
  geom_bar(position = "dodge") + #B
  labs(title="Grouped Bar chart", #B
        x="Treatment", #B
        y="Frequency") #B

ggplot(Arthritis, aes(x=Treatment, fill=Improved)) + #C
  geom_bar(position = "fill") + #C
  labs(title="Stacked Bar chart", #C
        x="Treatment", #C
        y="Frequency") #C

```

#A Stacked bar chart

#B Grouped bar chart

#C Filled bar chart

In the stacked bar chart, each segment represents the frequency or proportion of cases within in a given Treatment (Placebo, Treated) and Improvement (None, Some, Marked) level combination. The segments are stacked separately for each Treatment level. The grouped bar chart places the segments representing Improvement side by side within each Treatment level. The filled bar chart is a stacked bar chart rescaled so that the height of each bar is 1 and the segment heights represent proportions.

Filled bar charts are particularly useful for comparing the proportions of one categorical variable over the levels of another categorical variable. For example, the filled bar chart in figure 6.2 clearly displays the larger percentage of treated patients with marked improvement compared with patients receiving a placebo.

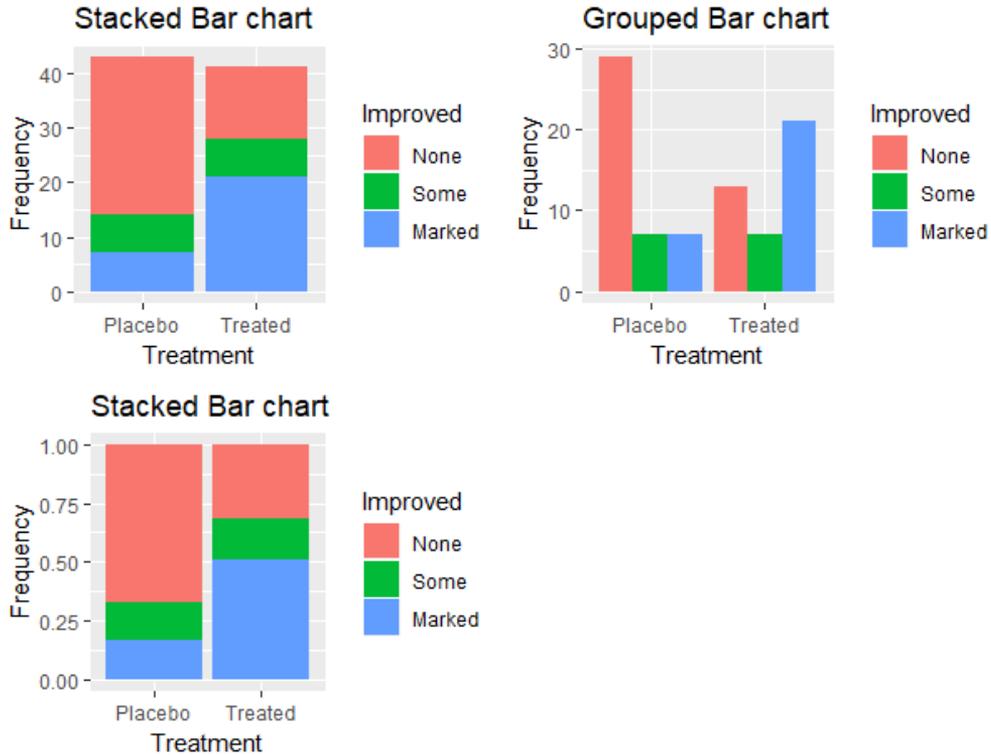


Figure 6.2 Stacked, grouped, and filled bar charts

6.1.3 Mean bar charts

Bar plots needn't be based on counts or frequencies. You can create bar charts that represent means, medians, percents, standard deviations, and so forth by summarizing the data with an appropriate statistic and passing the results to `ggplot2`.

In the following graph, we'll plot the mean illiteracy rate for regions of the United States in 1970. The built-in R dataset `state.x77` has the illiteracy rates by state, and the dataset `state.region` has the region names for each state. The following listing provides the code needed to create the graph in figure 6.3.

Listing 6.3 Bar chart for sorted mean values

```
> states <- data.frame(state.region, state.x77)
> library(dplyr) #1
> plotdata <- states %>%
  group_by(state.region) %>%
  summarize(mean = mean(illiteracy))
plotdata
```

```
# A tibble: 4 x 2
  state.region mean
  <fct>      <dbl>
1 Northeast  1
2 South     1.74
3 North Central 0.7
4 West      1.02

> ggplot(plotdata, aes(x=reorder(state.region, mean), y=mean)) + #2
  geom_bar(stat="identity") +
  labs(x="Region",
       y="",
       title = "Mean Illiteracy Rate")
```

#1 Generate means by region
#2 Plot means in a sorted bar chart

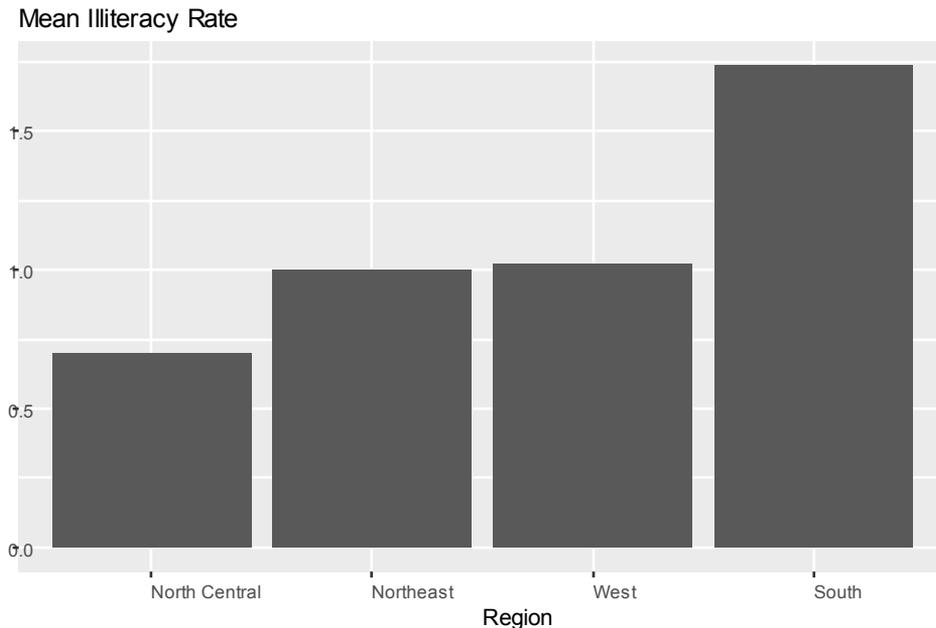


Figure 6.3 Bar chart of mean illiteracy rates for US regions sorted by rate

First, the mean illiteracy rate is calculated for each region #1. Next, the means are plotted in sorted in ascending order as bars #2. Normally, the `geom_bar()` function calculates and plots cell counts but adding the `stat="identity"` option forces the function to plot the numbers provided (means in this case). The `reorder()` function is used to order the bars by increasing mean illiteracy.

When plotting summary statistics such as means, it's good practice to indicate the variability of the estimates involved. One measure of variability is the standard error of the statistic – an estimate of the expected variation of the statistic across hypothetical repeated samples. The following plot adds error bars using the standard error of the mean.

Listing 6.4 Bar chart of mean values with error bars

```
> plotdata <- states %>% #1
  group_by(state.region) %>%
  summarize(n=n(),
            mean = mean(illiteracy),
            se = sd(illiteracy)/sqrt(n))

> plotdata

# A tibble: 4 x 4
  state.region   n mean   se
  <fct>         <int> <dbl> <dbl>
1 Northeast     9  1 0.0928
2 South        16  1.74 0.138
3 North Central 12  0.7 0.0408
4 West         13  1.02 0.169

> ggplot(plotdata, aes(x=reorder(state.region, mean), y=mean)) + #2
  geom_bar(stat="identity", fill="skyblue") +
  geom_errorbar(aes(ymin=mean-se, ymax=mean+se), width=0.2) + #3
  labs(x="Region",
       y="",
       title = "Mean Illiteracy Rate",
       subtitle = "with standard error bars")
```

#1 Generate means and standard errors by region

#2 Plot means in a sorted bar chart

#3 Add error bars

The means and standard errors are calculated for each region #1. The bars are then plotted in order of increasing illiteracy. The color is changed from a default dark grey to a lighter shade (sky blue) so that error bars to be added in the next step will stand out #2. Finally, the error bars are plotted #3. The `width` option in the `geom_errorbar()` function controls the horizontal width of the error bars and is purely aesthetic – it has no statistical meaning. In addition to displaying the mean illiteracy rates, we can see that the mean for the North Central region is the most reliable (least variability) and the West region is least reliable (largest variability).

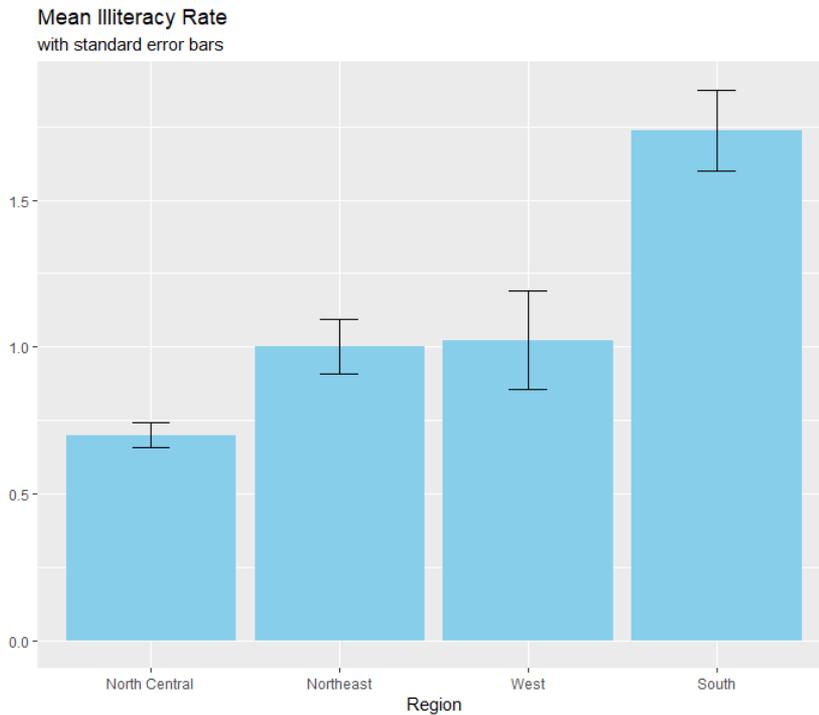


Figure 6.4 Bar chart of mean illiteracy rates for US regions sorted by rate. The standard error of the mean has been added to each bar.

6.1.4 Tweaking bar charts

There are several ways to tweak the appearance of a bar chart. The most common are customizing the bar colors and labels. We'll look at each in turn.

BAR CHART COLORS

Custom colors can be selected for the bar areas and borders. In the `geom_bar()` function the option `fill="color"` assigns a color for the area, while `color="color"` assigns a color for the border.

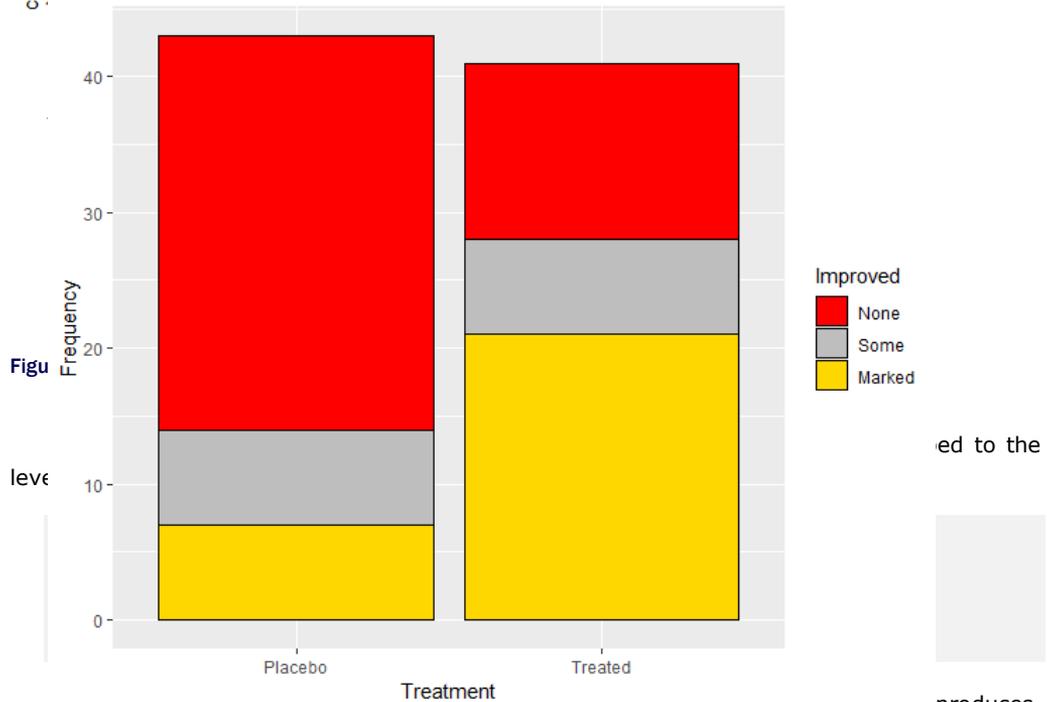
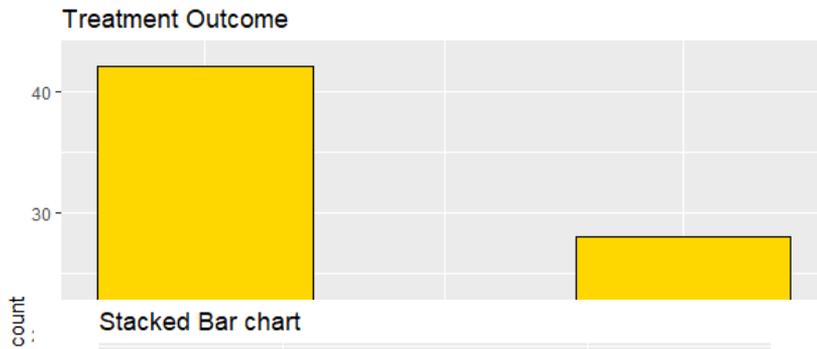
Fill vs. Color

In general, `ggplot2` uses *fill* to specify the color of geometric objects that have area (such as bars, pie slices, boxes), and the term *color* to when referring to the color of geometric objects without area (such as lines, points, and borders).

For example, the code

```
data(Arthritis, package="vcd")
ggplot(Arthritis, aes(x=Improved)) +
  geom_bar(fill="gold", color="black") +
  labs(title="Treatment Outcome")
```

produces the graph in figure 6.5.



the graph in figure 6.6.

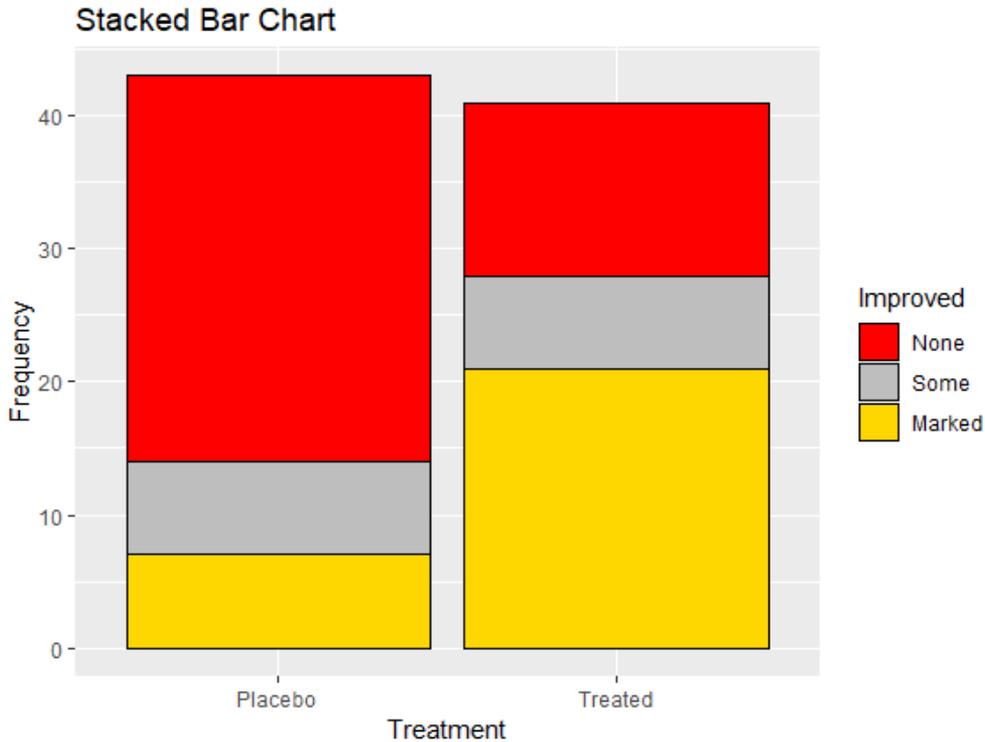


Figure 6.6 Stacked bar chart with custom fill colors mapped to Improvement

Here, bar fill colors are mapped to the levels of the variable `Improved`. The `scale_fill_manual()` function specifies red for `None`, grey for `Some`, and gold for `Marked` improvement. Color names can be obtained from <http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf>. Other methods of selecting colors are discussed in chapter 19 (Advanced Graphics with `ggplot2`).

BAR CHART LABELS

When there are many bars or long labels, bar chart labels tend to overlap and become unreadable. Consider the following example. The dataset `mpg` in the `ggplot2` package describes fuel economy data from for 38 popular car models in 1999 and 2008. Each model has several configurations (transmission type, number of cylinders, etc.). Let's say that we want a count of how many instances of each model are in the dataset. The code

```
ggplot(mpg, aes(x=model)) +
  geom_bar() +
  labs(title="Car models in the mpg dataset",
```

```
y="Frequency", x="")
```

produces the graph in figure 6.7.

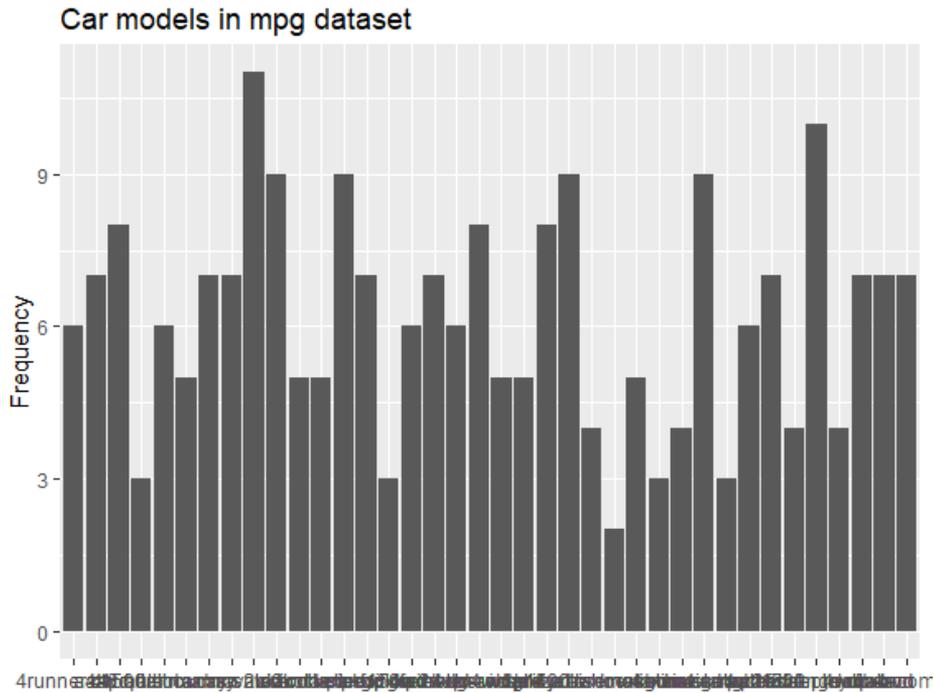


Figure 6.7 Bar chart with overlapping labels

Even with my glasses (or a glass of wine), I can't read this. Two simple tweaks will make the labels readable. First, we can plot the data as a horizontal bar chart.

```
ggplot(mpg, aes(x=model)) +
  geom_bar() +
  labs(title="Car models in the mpg dataset",
        y="Frequency", x="") +
  coord_flip()
```

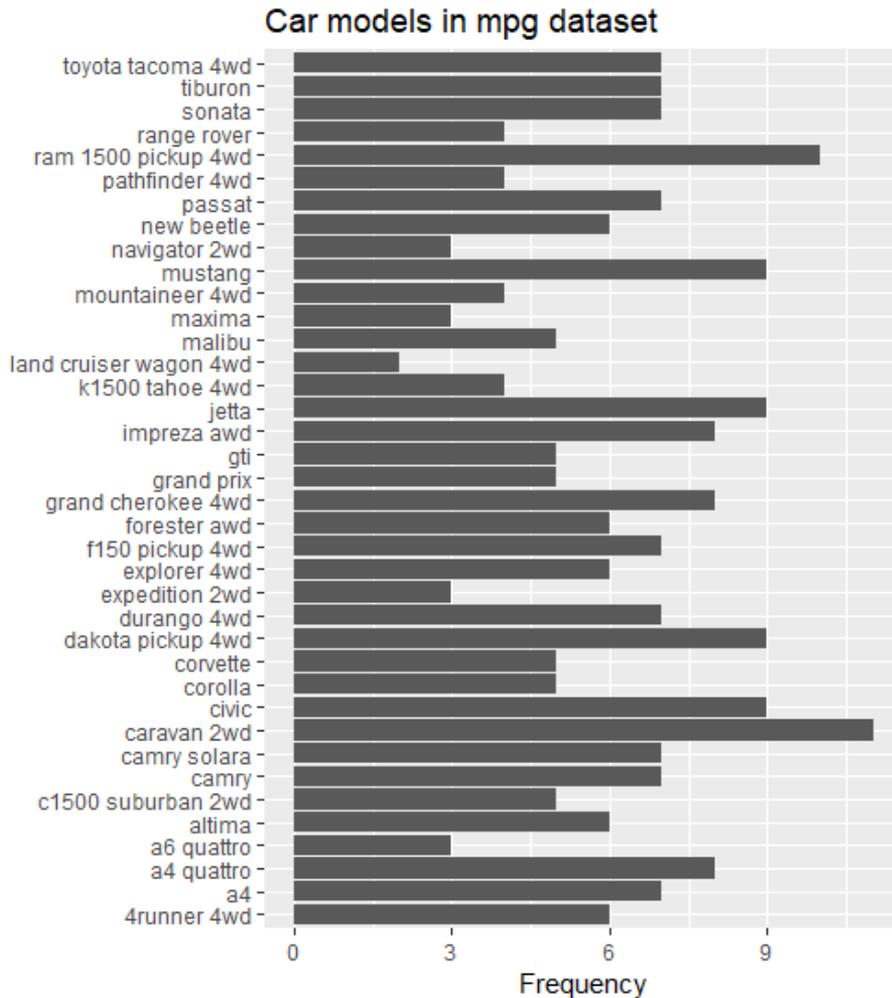


Figure 6.8 A horizontal bar chart avoids label overlap.

Second, we can angle the label text and use a smaller font.

```
ggplot(mpg, aes(x=model)) +
  geom_bar() +
  labs(title="Model names in the mpg dataset",
       y="Frequency", x="") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1, size=8))
```

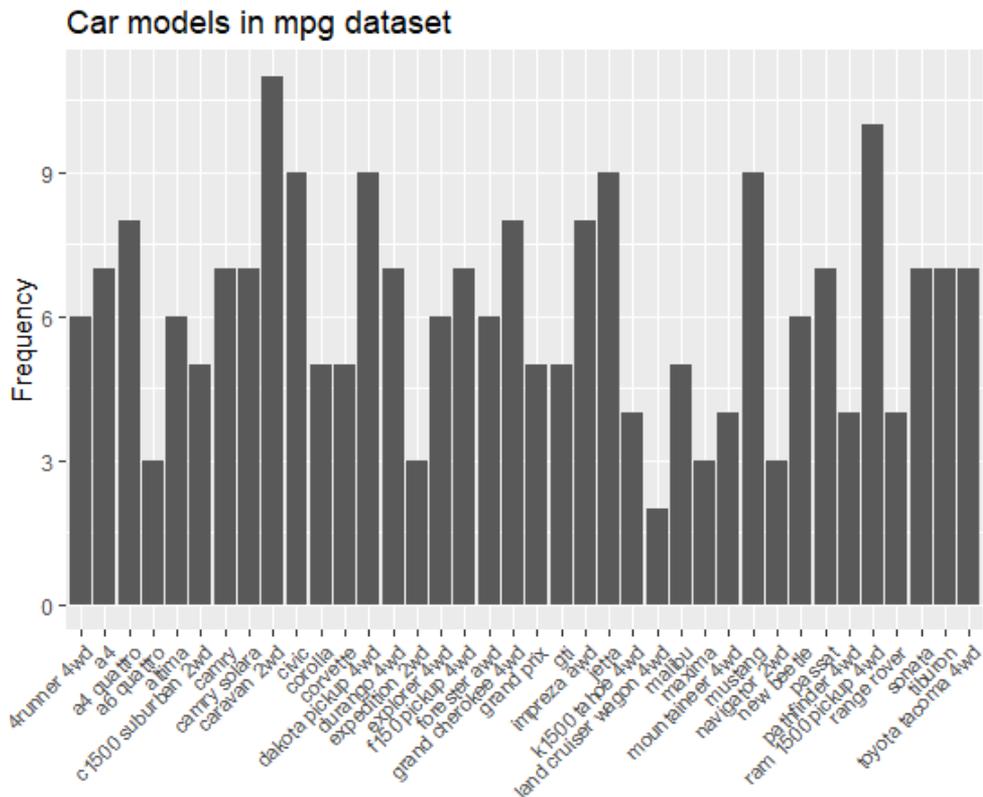


Figure 6.9 Bar chart with angled labels and a smaller label font.

The `theme()` function is discussed more fully in chapter 19 (Advanced Graphics with `ggplot2`). In addition to bar charts, pie charts are a popular vehicle for displaying the distribution of a categorical variable. We'll consider them next.

6.2 Pie charts

Pie charts are ubiquitous in the business world, but they're denigrated by most statisticians, including the authors of the R documentation. They recommend bar or dot plots over pie charts because people are able to judge length more accurately than volume. Perhaps for this reason, the pie chart options in R are severely limited when compared with other statistical platforms.

However, there are times when pie charts can be useful. In particular, they can capture part-whole relationships well. For example, a pie chart can be used to display the percentage of tenured faculty at a university who are female.

You can create a pie chart in base R using the `pie()` function, but as I've said, the functionality is limited and the plots are unattractive. To address this, I've created a packaged called `ggpie` that allows you to create a wide variety of pie charts using `ggplot2` (no flame emails please!). You can install it with the following code.

```
if(!require(devtools) install.packages("devtools"))
devtools::install_github("rkabacoff/ggpie")
```

The basic syntax is

```
ggpie(data, x, by, offset, percent, legend, title)
```

where

- `data` is a data frame
- `x` is the categorical variable to be plotted
- `by` is an optional second categorical variable. If present, a pie will be produced for each level of this variable.
- `offset` is the distance of the pie slice labels from the origin. A value of 0.5 will place the labels in the center of the slices, and a value greater than 1.0 will place them outside the slice.
- `percent` is logical. If `FALSE`, percentage printing is suppressed.
- `legend` is logical. If `FALSE`, the legend is omitted and each pie slice is labeled.
- `title` is a an option title.

Additional options (described on the `ggpie` website) allow you to customize the pie chart's appearance.

Let's create a pie chart displaying the distribution of car classes in the `mpg` data frame.

```
library(ggplot2)
library(ggpie)
ggpie(mpg, class)
```

The results are given in figure 6.10. From the graph, we see that 26% percent of cars are SUVs, while only 2% are two-seaters.

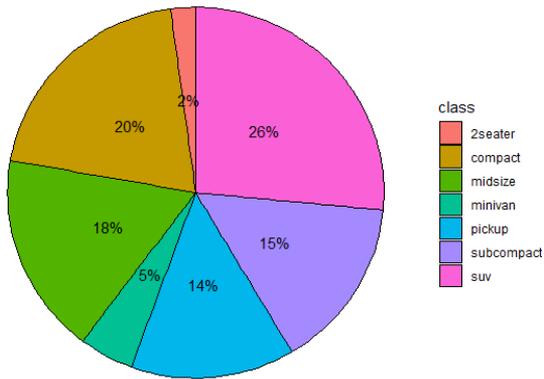


Figure 6.10. Pie chart displaying the percentage of each car class in the mpg data frame.

In the next version, the legend is removed and each pie slice is labeled. In addition, the labels are placed outside the pie area, and title is added.

```
ggpie(mpg, class, legend=FALSE, offset=1.3,
      title="Automobiles by Car Class")
```

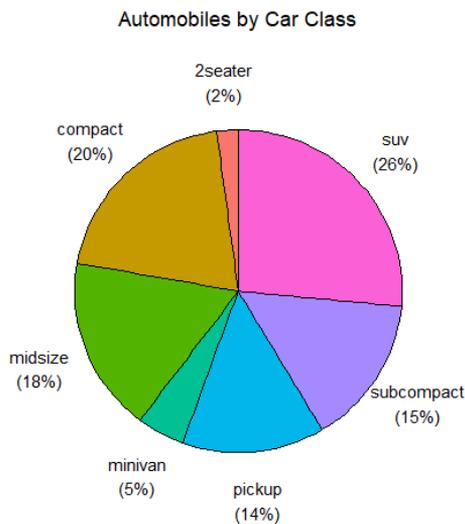


Figure 6.11. Pie chart with labels displayed outside the pie.

In the final example, the distribution of car class is displayed by year.

```
ggpie(mpg, class, year,
      legend=FALSE, offset=1.3, title="Car Class by Year")
```

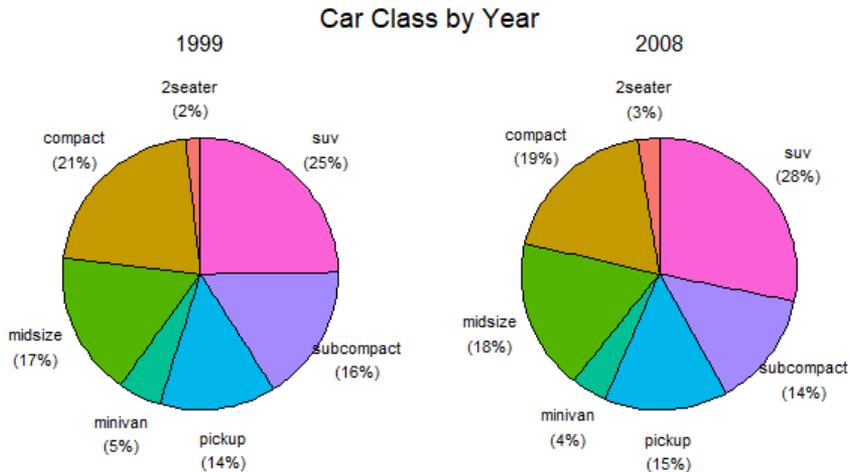


Figure 6.12. Pie charts displaying the distribution of car classes by year

Between 1999 and 2008, the distribution of car classes appears to have remained rather constant. The `ggpie` package can create more complex and customized pie charts. See the documentation (<http://rkabacoff.github.io/ggpie>) for details.

6.3 Tree maps

An alternative to a pie chart is a tree map. A tree map displays the distribution of a categorical variable using rectangles that are proportional to variable levels. Unlike pie charts, tree maps can handle categorical variables with *many* levels. We'll create tree maps using the `treemapify` package. Be sure to install it before proceeding (`install.packages("treemapify")`).

We'll start by creating a tree map displaying the distribution of car manufacturers in the `mpg` data frame. The code is given in listing 6.6. The resulting graph is given in figure 6-12.

Listing 6.6 Simple Tree Map

```
library(ggplot2)
library(dplyr)
library(treemapify)
```

```

plotdata <- mpg %>% count(manufacturer) #1

ggplot(plotdata,          #2
       aes(fill = manufacturer,
           area = n,
           label = manufacturer)) +
geom_treemap() +
geom_tree_text() +
theme(legend.position = FALSE)

```

#1 Summarize the data

#2 Create the tree map

First we calculate the frequency counts for each level the `manufacturer` variable #1. This information is passed to `ggplot2` to create the graph #2. In the `aes()` function, `fill` refers to the categorical variable, `area` is the count for level, and `label` is the option variable used to label the cells. The `geom_treemap()` function creates the tree map and the `geom_tree_text()` function adds the labels to each cell. The `theme()` function is used to suppress the legend, which is redundant here, since each cell is labeled.

Simple Tree Map



Figure 6.13. Tree map displaying the distribution of car manufacturers in the mpg data set. Rectangle size is proportional to the number of cars from each manufacturer.

In the next example, a second variable is added – drivetrain. The number of cars by manufacturer is plotted for front-wheel, rear-wheel, and four-wheel drives. The code is provided in listing 6.7 and the plot is displayed in figure 6.14.

Listing 6.7 Tree Map with Subgrouping

```

plotdata <- mpg %>%                                #1
count(manufacturer, drv)
plotdata$drv <- factor(plotdata$drv,              #2
  levels=c("4", "f", "r"),
  labels=c("4-wheel", "front-wheel", "rear"))

ggplot(plotdata,                                  #3
  aes(fill = manufacturer,
    area = n,
    label = manufacturer,
    subgroup=drv)) +
geom_treemap() +
geom_treemap_subgroup_border() +
geom_treemap_subgroup_text(
  place = "middle",
  colour = "black",
  alpha = 0.5,
  grow = FALSE) +
geom_treemap_text(colour = "white",
  place = "centre",
  grow=FALSE) +
theme(legend.position = "none")

```

#1 Compute cell counts

#2 Provide better labels for drivetrains

#3 Create tree map

First, the frequencies for each manufacturer-drivetrain combination is calculated #1. Next, better labels are provided for the drivetrain variable #2. The new data frame is passed to `ggplot2` to produce the tree map #3. The `subgroup` option in the `aes()` function creates separate subplots for each drivetrain type. The `geom_treemap_border()` and `geom_treemap_subgroup_text()` add borders and labels for the subgroups respectively. Options in each function control their appearance. The subgroup text is centered and given some transparency (`alpha=0.5`). The text font remains a constant size, rather than growing to fill the area (`grow=FALSE`). The tree map cell text is print in a white font, centered in each cell, and does not grow to fill the boxes.

From the graph in figure 6.14, it is clear for example, that Hyundai has front-wheel cars, but not rear-wheel or four-wheel cars. The manufacturers with rear-wheel cars are primarily Ford and Chevrolet. Many of the four-wheel cars are made by Dodge.

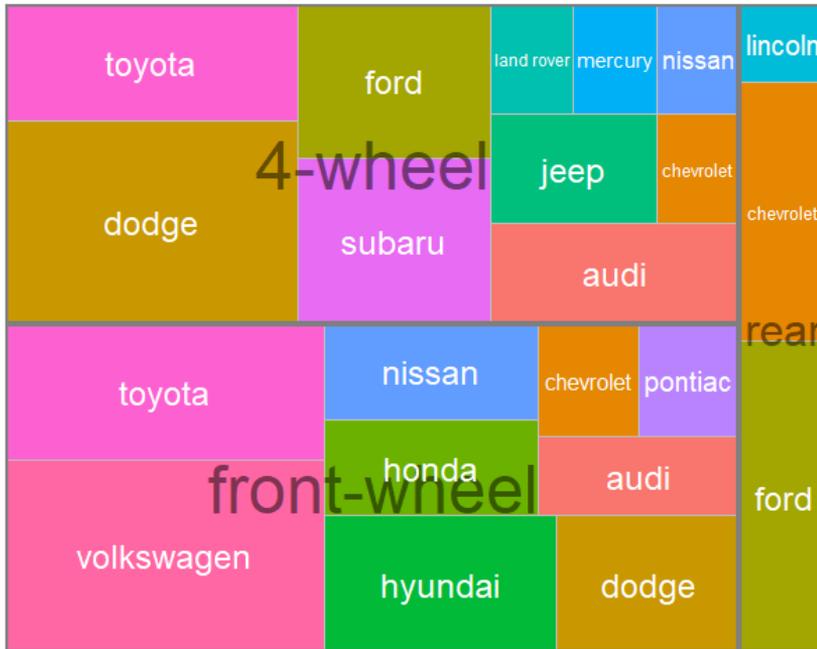


Figure 6.14 Tree map with car manufactures by drive-train type.

Now that we've covered pie charts and tree maps, let's move on to histograms. Unlike bar charts, pie charts, and tree maps, histograms describe the distribution of a continuous variable.

6.4 Histograms

Histograms display the distribution of a continuous variable by dividing the range of scores into a specified number of bins on the x-axis and displaying the frequency of scores in each bin on the y-axis. You can create histograms using

```
ggplot(data, aes(x = contvar)) + geom_histogram()
```

where *data* is a data frame and *contvar* is a continuous variable. Using the `mpg` data set in the `ggplot` package, we'll examine the distribution of city miles per gallon (`cty`) for 117 automobile configurations in 2008. Four variations of a histogram are created in listing 6.8 and the results graphs are presented in figure 6.15.

Listing 6.6 Histograms

```
library(ggplot2)
library(scales)
```

```

data(mpg)
cars2008 <- mpg[mpg$year == 2008, ]

ggplot(cars2008, aes(x=hwy)) +           #1
  geom_histogram() +                   #1
  labs(title="Default histogram")      #1

ggplot(cars2008, aes(x=hwy)) +           #2
  geom_histogram(bins=20, color="white", fill="steelblue") + #2
  labs(title="Colored histogram with 20 bins", #2
        x="City Miles Per Gallon", #2
        y="Frequency")

ggplot(cars2008, aes(x=hwy, y=..density..)) + #3
  geom_histogram(bins=20, color="white", fill="steelblue") + #3
  scale_y_continuous(labels=scales::percent) + #3
  labs(title="Histogram with percentages", #3
        y="Percent", #3
        x="City Miles Per Gallon") #3

ggplot(cars2008, aes(x=hwy, y=..density..)) + #4
  geom_histogram(bins=20, color="white", fill="steelblue") + #4
  scale_y_continuous(labels=scales::percent) + #4
  geom_density(color="red", size=1) + #4
  labs(title="Histogram with density curve", #4
        y="Percent", #4
        x="City Miles Per Gallon") #4

```

#1 Simple histogram

#2 Colored histogram with 20 bins

#3 Histogram with percentages

#4 Histogram with density curve

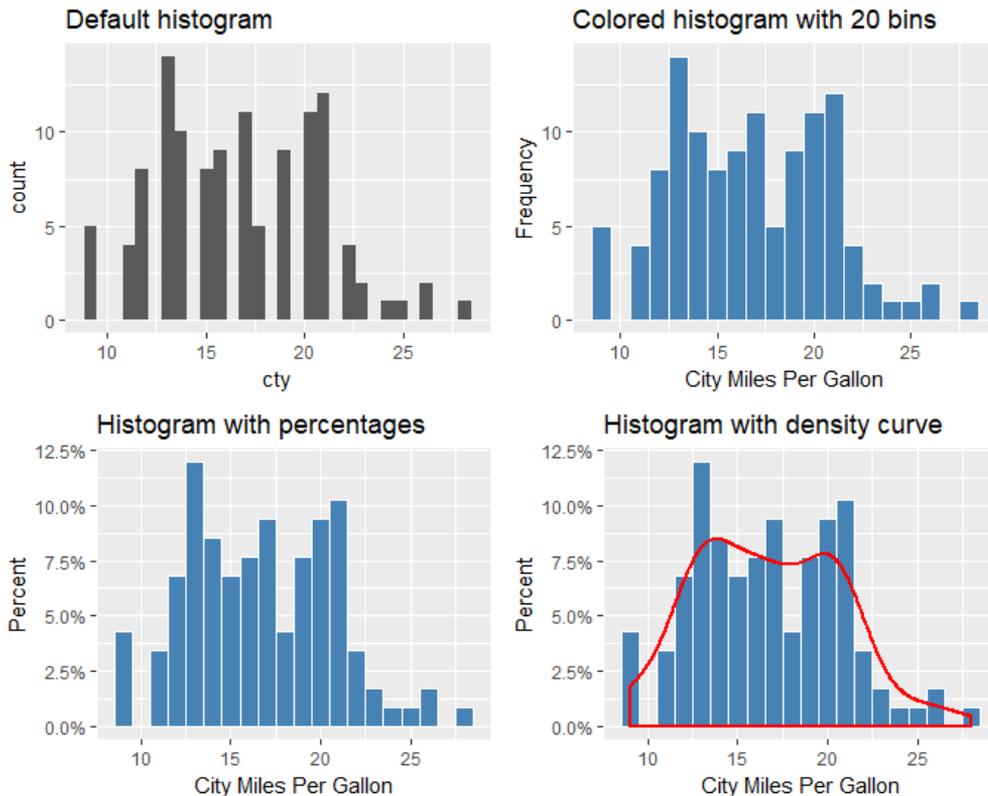


Figure 6.15 Histogram examples

The first histogram #1 demonstrates the default plot when no options are specified. In this case, 30 bins are created. For the second histogram #2, 20 bins, a steel blue fill, and a white border color are specified. In addition, more informative labels have been added. The number of bins can strongly influence the appearance of the histogram. It is a good idea to experiment with the `bins` value until you find one that captures the distribution well. With 20 bins, it appears that there are two peaks to the distribution – one around 13 mpg and one around 20.5 mpg.

The third histogram #3 plots the data as percents rather than frequencies. This is accomplished by assigning the built-in variable `..density..` to the y axis. The `scales` package is used to format the y-axis as percents. Be sure to install the package (`install.packages("scales")`) before running this part of the code.

The fourth histogram #4 is similar to the previous plot, but adds a density curve. The density curve is a kernel density estimate and is described in the next section. It provides a smoother description of the distribution of scores. The `geom_density()` function is used to plot

the kernel curve in a red color and a width that's slightly larger the default thickness for lines. The density curve also suggests a bimodal distribution (two peaks).

6.5 Kernel density plots

In the previous section, you saw a kernel density plot superimposed on a histogram. Technically, kernel density estimation is a nonparametric method for estimating the probability density function of a random variable. Basically, we're trying to draw a smoothed histogram, where the area under the curve equals one. Although the mathematics are beyond the scope of this text, density plots can be an effective way to view the distribution of a continuous variable. The format for a density plot is

```
ggplot(data, aes(x = contvar)) + geom_density()
```

where *data* is a data frame and *contvar* is a continuous variable. Again, let's plot the distribution of city miles per gallon (*cty*) for cars in 2008. Three kernel density examples are given in the next listing, and the results are provided in figure 6.16.

Listing 6.7 Kernel density plots

```
library(ggplot2)
data(mpg)
cars2008 <- mpg[mpg$year == 2008, ]

ggplot(cars2008, aes(x=cty)) +          #1
  geom_density() +                    #1
  labs(title="Default kernel density plot") #1

ggplot(cars2008, aes(x=cty)) +          #2
  geom_density(fill="red") +          #2
  labs(title="Filled kernel density plot") #2

> bw.nrd0(cars2008$cty)                #3
1.408                                  #3

ggplot(cars2008, aes(x=cty)) +          #4
  geom_density(fill="red", bw=.5) +    #4
  labs(title="Kernel density plot with bw=0.5") #4
```

#1 Default density plot

#2 Filled density plot

#3 Print default bandwidth

#4 Density plot with smaller bandwidth

The default kernel density plot is given first #1. In the second example, the area under the curve is fill with red. The smoothness of the curve is control with a bandwidth parameter, which is calculated from the data being plotted. The code `bw.nrd0(cars2008$cty)` displays this value (1.408) #3. Using a larger bandwidth will give a smoother curve with less details. A smaller value will give a more squiggly curve (I don't squiggly an official term, but I couldn't

think of a better one). The third example uses a smaller bandwidth (`bw=`), allowing us to see more detail⁴. As with the `bins` parameter for histograms, it is a good idea to try several bandwidth values to see which value helps you visualize the data most effectively.

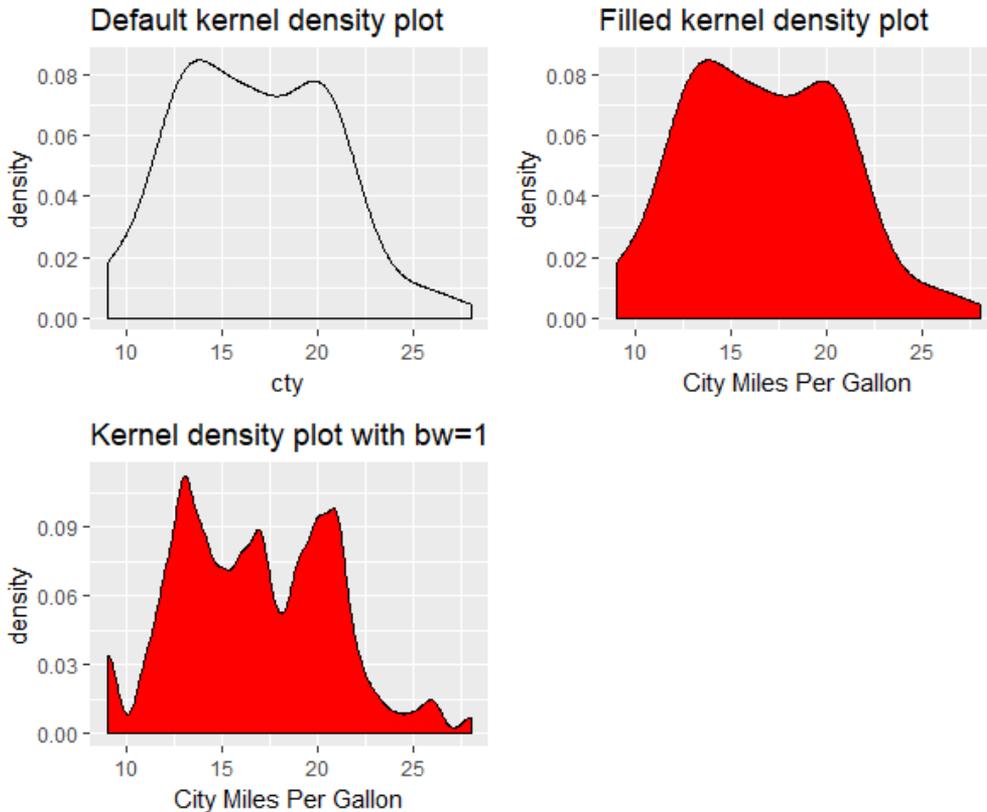


Figure 6.16 Kernel density plots

Kernel density plots can be used to compare groups. This is a highly underutilized approach, probably due to a general lack of easily accessible software. Fortunately, the `ggplot2` package fills this gap nicely.

For this example, we'll compare the 2008 city gas mileage estimates for 4-, 6-, and 8-cylinder cars. There are only a handful of cars with 5 cylinders so we will drop them from the analyses. The code is presented in listing 6.7. The resulting graphs are given in figures 6.17 and 6.18.

Listing 6.8 Comparative kernel density plots

```

data(mpg, package="ggplot2")           #1
cars2008 <- mpg[mpg$year == 2008 & mpg$cyl != 5,] #1
cars2008$cylinders <- factor(cars2008$cyl)      #1

ggplot(cars2008, aes(x=cty, color=Cylinders, linetype=Cylinders)) + #2
  geom_density(size=1) + #2
  labs(title="Fuel Efficiency by Number of Cylinders", #2
        x = "City Miles per Gallon") #2

ggplot(cars2008, aes(x=cty, fill=Cylinders)) +
  geom_density(alpha=.4) + #3
  labs(title="Fuel Efficiency by Number of Cylinders", #3
        x = "City Miles per Gallon") #3

```

#1 Prepare the data

#2 Plots the density curves

#3 Plot filled density curves

First, a fresh copy of the data is loaded and 2008 data for cars with 4, 6, or 8 cylinders are retained #1. The number of cylinders (`cyl`) is saved as a categorical factor (`Cylinders`). The transformation is required because `ggplot2` expects the grouping variable to be categorical (and `cyl` is stored as a continuous variable).

A kernel density curve is plotted for each level of the `Cylinders` variable #2. Both the color (red, green, blue) and line type (solid, dotted, dashed) are mapped to the number of cylinders. Finally, the same plot is produced with filled curves #3. Transparency is added (`alpha=0.4`), since the filled curves overlap and we want to be able to see each one.

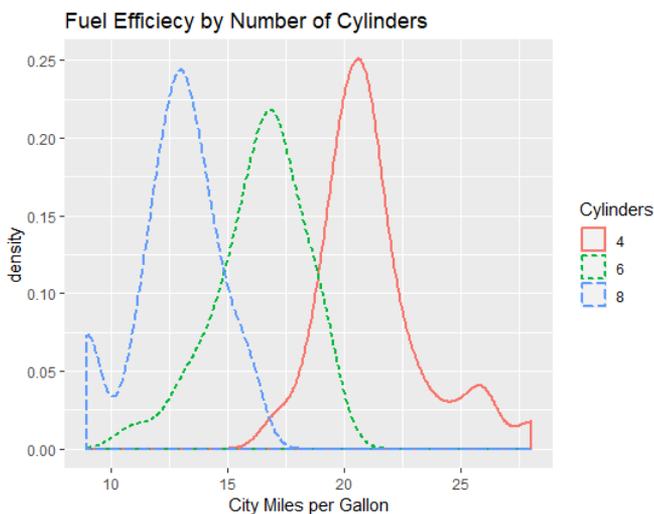


Figure 6.17 Kernel density curves of city mpg by number of cylinders

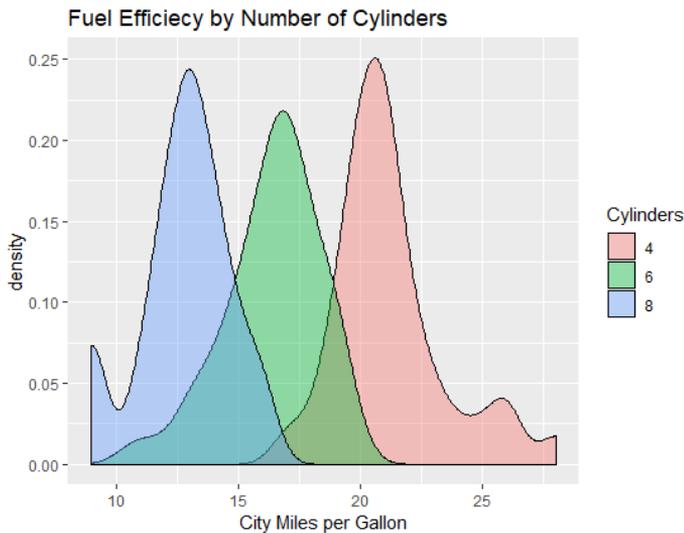


Figure 6.18 Filled kernel density curves of city mpg by number of cylinders.

Overlapping kernel density plots can be a powerful way to compare groups of observations on an outcome variable. Here you can see both the shapes of the distributions and the amount of overlap between groups. (The moral of the story is that my next car will have four cylinders—or a battery.)

Box plots are also a wonderful (and more commonly used) graphical approach to visualizing distributions and differences among groups. We'll discuss them next.

6.6 Box plots

A *box-and-whiskers plot* describes the distribution of a continuous variable by plotting its five-number summary: the minimum, lower quartile (25th percentile), median (50th percentile), upper quartile (75th percentile), and maximum. It can also display observations that may be outliers (values outside the range of $\pm 1.5 \cdot \text{IQR}$, where IQR is the interquartile range defined as the upper quartile minus the lower quartile). For example, the following code produces the plot shown in figure 6.19:

```
ggplot(mtcars, aes(x="", y=mpg)) +
  geom_boxplot() +
  labs(y = "Miles Per Gallon", x="", title="Box Plot")
```

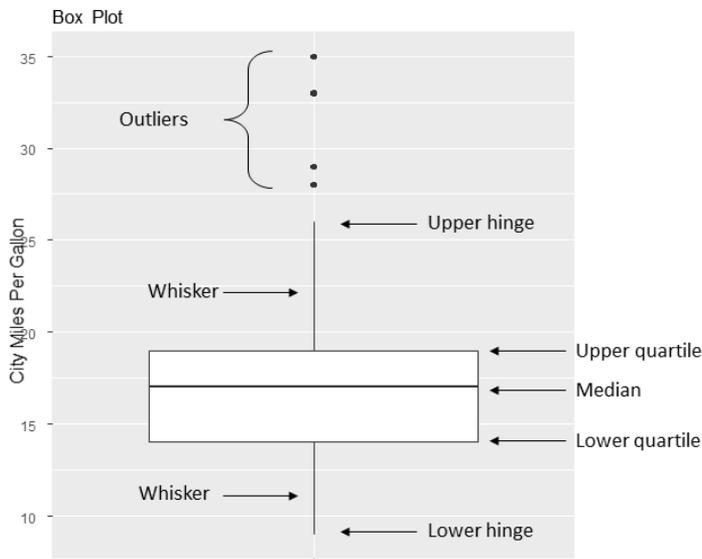


Figure 6.19 Box plot with annotations added by hand

I've added annotations by hand to illustrate the components. By default, each whisker extends to the most extreme data point, which is no more than 1.5 times the interquartile range for the box. Values outside this range are depicted as dots.

For example, in this sample of cars, the median mpg is 17, 50% of the scores fall between 14 and 19, the smallest value is 9, and the largest value is 35. How did I read this so precisely from the graph? Issuing `boxplot.stats(mtcars$mpg)` prints the statistics used to build the graph (in other words, I cheated). There is four outliers (greater than the upper hinge of 26). These values would be expected to occur less than 1% of the time in a normal distribution.

6.6.1 Using parallel box plots to compare groups

Box plots are a useful method of comparing the distribution of a quantitative variable across the levels of a categorical variable. Once again, let's compare city gas mileage for 3-, 6-, and 8-cylinder cars, but this time use both 1999 and 2008 data. Since there are only a few 5-cylinder cars, we will delete them. We'll also convert `year` and `cyl` from continuous numeric variables into categorical (grouping) factors.

```
library(ggplot2)
cars <- mpg[mpg$cyl != 5, ]
cars$cylinders <- factor(cars$cyl)
cars$Year <- factor(cars$year)
```

```
The code
ggplot(cars, aes(x=Cylinders, y=cty)) +
  geom_boxplot() +
  labs(x="Number of Cylinders",
       y="Miles Per Gallon",
       title="Car Mileage Data")
```

produces the graph in figure 6.20. You can see that there's a good separation of groups based on gas mileage, with fuel efficiency dropping as the number of cylinders increases. There are also four outliers (cars with unusually high mileage) in the four-cylinder group.

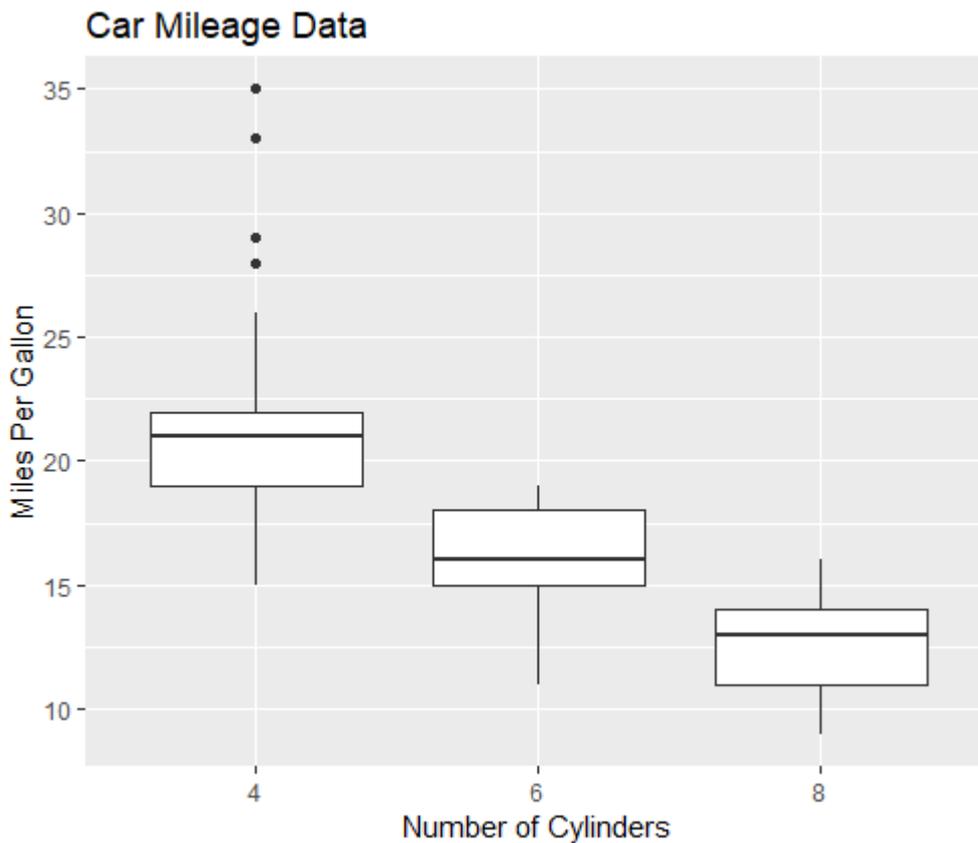


Figure 6.20 Box plots of car mileage vs. number of cylinders

Box plots are very versatile. By adding `notch=TRUE`, you get *notched* box plots. If two boxes' notches don't overlap, there's strong evidence that their medians differ (Chambers et al., 1983, p. 62). The following code creates notched box plots for the mileage example:

```
ggplot(cars, aes(x=Cylinder, y=cty)) +
  geom_boxplot(notch=TRUE,
              fill="steelblue",
              varwidth=TRUE) +
  labs(x="Number of Cylinders",
       y="Miles Per Gallon",
       title="Car Mileage Data")
```

The `fill` option fills the box plots with a red color. In a standard box plot, the box width has no meaning. Adding `varwidth=TRUE`, draws box widths proportional to the square roots of the number of observations in each group.

You can see in figure 6.21 that the median car mileage for four-, six-, and eight-cylinder cars differs. Mileage clearly decreases with number of cylinders. Additionally, there are fewer 8-cylinder cars, than 4- or 6-cylinder cars (although the difference is subtle).

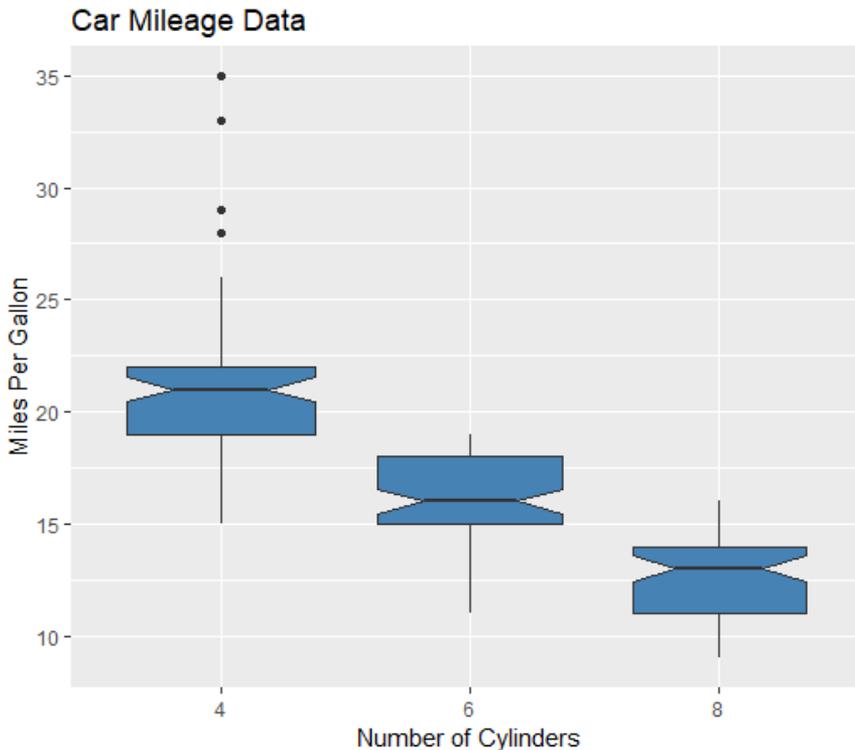


Figure 6.21 Notched box plots for car mileage vs. number of cylinders

Finally, you can produce box plots for more than one grouping factor. The following code provides box plots for city miles per gallon versus the number of cylinders by year (see figure 6.21). The `scale_fill_manual()` function has been added in order to customize the fill colors.

```
ggplot(cars, aes(x=Cylinders, y=cty, fill=Year)) +
  geom_boxplot() +
  labs(x="Number of Cylinders",
       y="Miles Per Gallon",
       title="City Mileage by # Cylinders and Year") +
  scale_fill_manual(values=c("gold", "green"))
```

From figure 6.22, it's again clear that median mileage decreases with number of cylinders. Additionally, for each group, mileage has increased between 1999 and 2008.

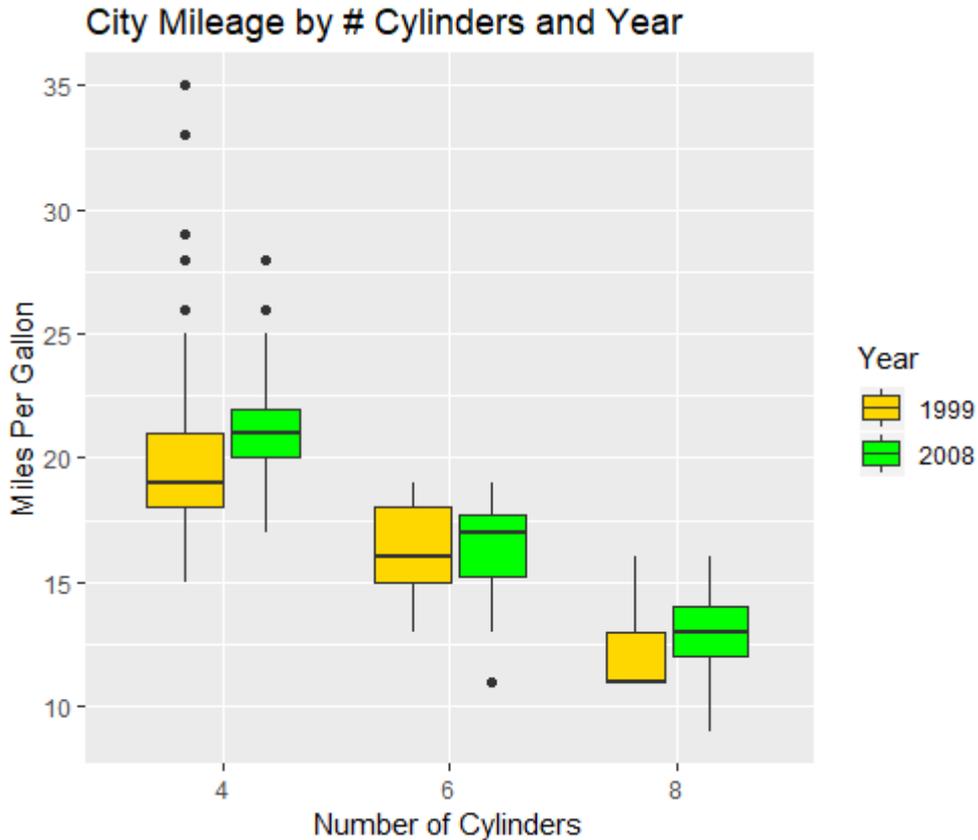


Figure 6.22 Box plots for car mileage vs. year and number of cylinders

6.6.2 Violin plots

Before we end our discussion of box plots, it's worth examining a variation called a *violin plot*. A violin plot is a combination of a box plot and a kernel density plot. You can create one using the `geom_violin()` function. In listing 6.9, we'll add violin plots to the box plots in figure 6.23.

Listing 6.9 Violin plots

```
library(ggplot2)
cars <- mpg[mpg$cyl != 5, ]
cars$Cylinders <- factor(cars$cyl)

ggplot(cars, aes(x=Cylinders, y=cty)) +
  geom_boxplot(width=0.2,
              fill="green") +
  geom_violin(fill="gold",
             alpha=0.3) +
  labs(x="Number of Cylinders",
       y="City Miles Per Gallon",
       title="Violin Plots of Miles Per Gallon")
```

The width of the box plots are set to 0.2 so that they will fit inside the violin plots. The violin plots are set with a transparency level of 0.3 so that the box plots are still visible.

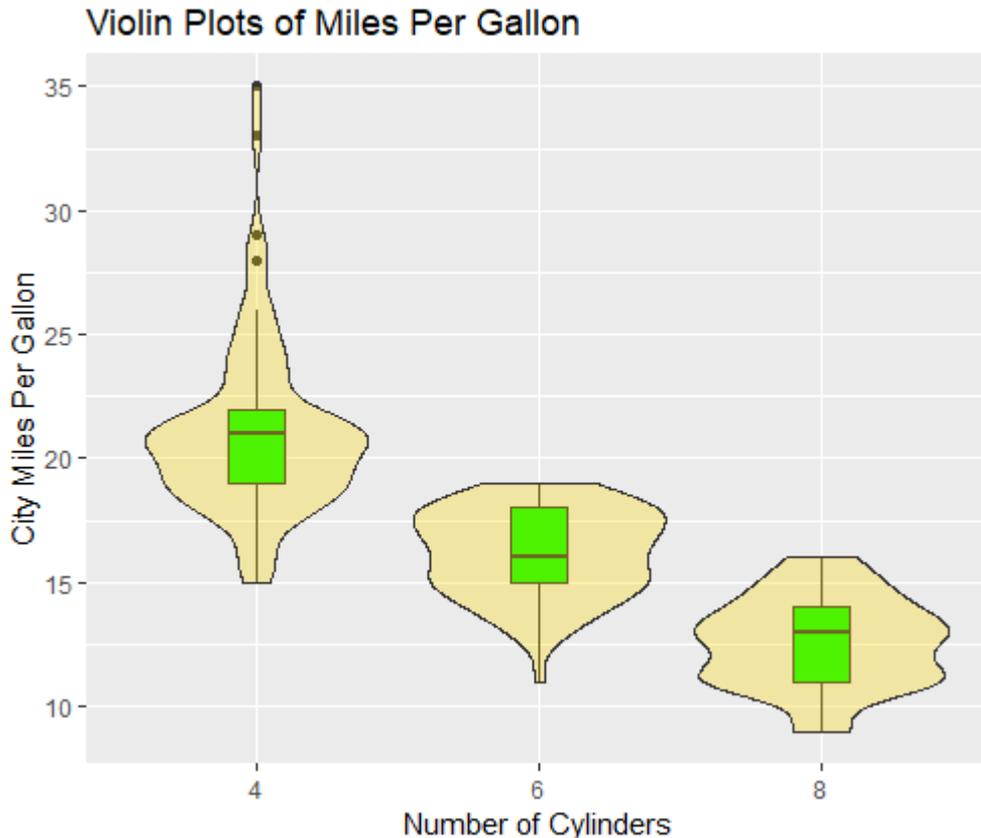


Figure 6.23 Violin plots of mpg vs. number of cylinders

Violin plots are basically kernel density plots superimposed in a mirror-image fashion over box plots. The middle lines are the medians, the black boxes range from the lower to the upper quartile, and the thin black lines represent the whiskers. Dots are outliers. The outer shape provides the kernel density plot. Here we can see that the distribution of gas mileage for 8-cylinder cars may be bimodal – a fact that is obscured by using box plots alone. Violin plots haven't really caught on yet. Again, this may be due to a lack of easily accessible software; time will tell.

We'll end this chapter with a look at dot plots. Unlike the graphs you've seen previously, dot plots plot every value for a variable.

6.7 Dot plots

Dot plots provide a method of plotting a large number of labeled values on a simple horizontal scale. You create them with the `dotchart()` function, using the format

```
ggplot(data, aes(x=contvar, y=catvar)) + geom_point()
```

where `data` is a data frame, `contvar` is a continuous variable, and `catvar` is a categorical variable. Here's an example using the highway gas mileage for the 2008 automobiles in the `mpg` dataset. Highway gas mileage is averaged by car model.

```
library(ggplot2)
library(dplyr)
plotdata <- mpg %>%
  filter(year == "2008") %>%
  group_by(model) %>%
  summarize(meanHwy=mean(hwy))

> plotdata

# A tibble: 38 x 2
  model      meanHwy
  <chr>      <dbl>
1 4runner 4wd    18.5
2 a4        29.3
3 a4 quattro 26.2
4 a6 quattro 24
5 altima    29
6 c1500 suburban 2wd 18
7 camry     30
8 camry solara 29.7
9 caravan 2wd 22.2
10 civic    33.8
# ... with 28 more rows

ggplot(plotdata, aes(x=meanHwy, y=model)) +
  geom_point() +
  labs(x="Miles Per Gallon",
       y="",
       title="Gas Mileage for Car Models")
```

The resulting plot is given in figure 6.24.

This graph allows you to see the mpg for each car model on the same horizontal axis. Dot plots typically become most useful when they're sorted. The following code sorts the cars from lowest to highest mileage.

```
ggplot(plotdata, aes(x=meanHwy, y=reorder(model, meanHwy))) +
  geom_point() +
```

```
labs(x="Miles Per Gallon",
     y="",
     title="Gas Mileage for Car Models")
```

The resulting graph is given in figure 6.25. To plot in descending order, use `reorder(model, -meanHwy)`.

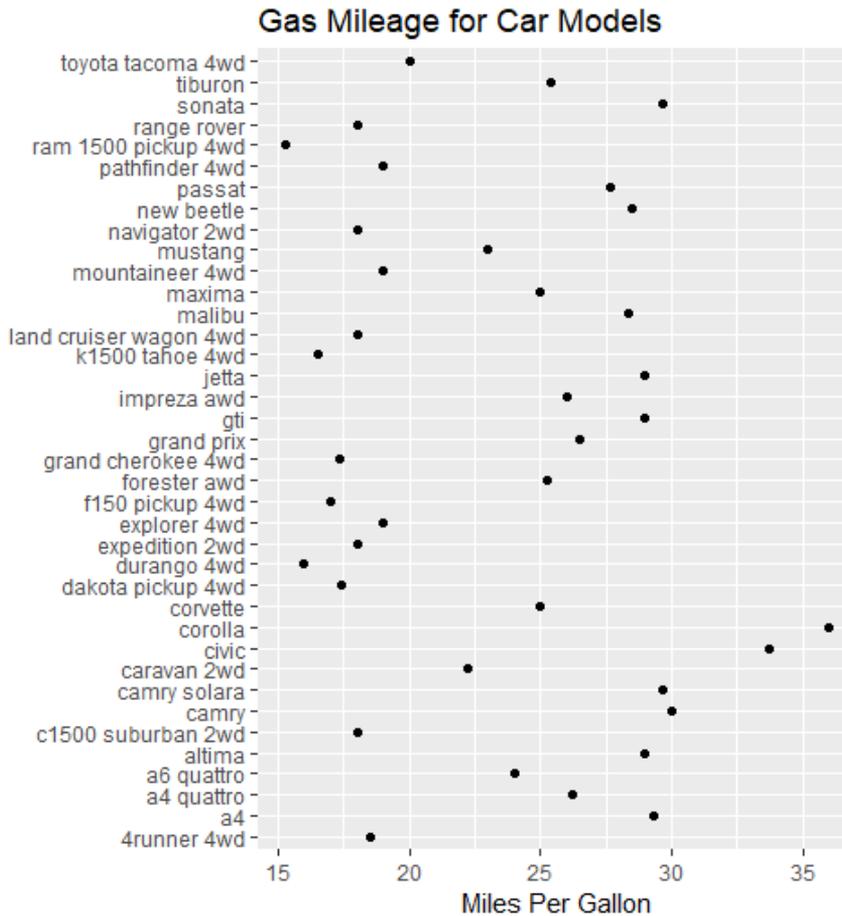


Figure 6.24 Dot plot of mpg for each car model

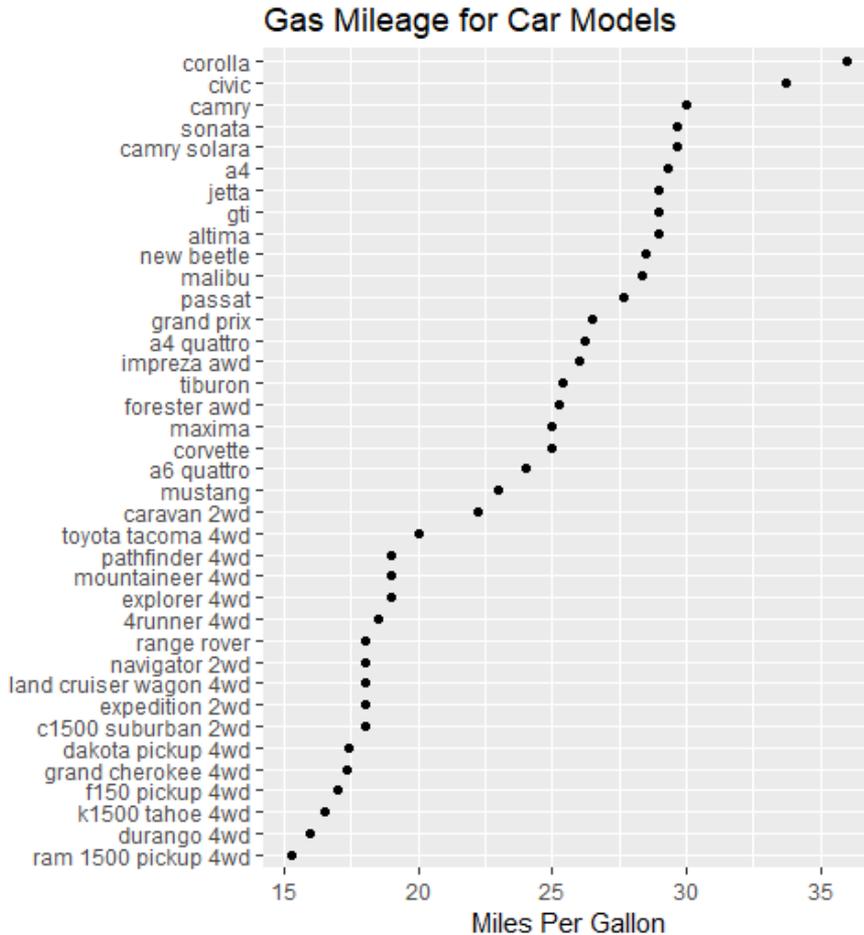


Figure 6.25 Dot plot of mpg for car models sorted by mileage

You can gain significant insight from the dot plot in this example because each point is labeled, the value of each point is inherently meaningful, and the points are arranged in a manner that promotes comparisons. But as the number of data points increases, the utility of the dot plot decreases.

6.8 Summary

- Bar charts (and to a lesser extent pie charts and tree maps) can be used to gain insight into the distribution of a categorical variable.
- Stacked, grouped, and filled bar charts can help you understand how groups differ on

a categorical outcome.

- Histograms, box plots, violin plots, and dot plots can help you visualize the distribution of continuous variables.
- Overlapping kernel density plots and parallel box plots can help you visualize group differences on a continuous outcome variable.

7

Basic statistics

This chapter covers

- Descriptive statistics
- Frequency and contingency tables
- Correlations and covariances
- t-tests
- Nonparametric statistics

In previous chapters, you learned how to import data into R and use a variety of functions to organize and transform the data into a useful format. We then reviewed basic methods for visualizing data.

Once your data is properly organized and you've begun to explore the data visually, the next step is typically to describe the distribution of each variable numerically, followed by an exploration of the relationships among selected variables two at a time. The goal is to answer questions like these:

- What kind of mileage are cars getting these days? Specifically, what's the distribution of miles per gallon (mean, standard deviation, median, range, and so on) in a survey of automobile makes and models?
- After a new drug trial, what's the outcome (no improvement, some improvement, marked improvement) for drug versus placebo groups? Does the gender of the participants have an impact on the outcome?
- What's the correlation between income and life expectancy? Is it significantly different from zero?
- Are you more likely to receive imprisonment for a crime in different regions of the United States? Are the differences between regions statistically significant?

In this chapter, we'll review R functions for generating basic descriptive and inferential statistics. First, we'll look at measures of location and scale for quantitative variables. Then you'll learn how to generate frequency and contingency tables (and associated chi-square tests) for categorical variables. Next, we'll examine the various forms of correlation coefficients available for continuous and ordinal variables. Finally, we'll turn to the study of group differences through parametric (t-tests) and nonparametric (Mann–Whitney U test, Kruskal–Wallis test) methods. Although our focus is on numerical results, we'll refer to graphical methods for visualizing these results throughout.

The statistical methods covered in this chapter are typically taught in a first-year undergraduate statistics course. If these methodologies are unfamiliar to you, two excellent references are McCall (2000) and Kirk (2007). Alternatively, many informative online resources are available (such as Wikipedia) for each of the topics covered.

7.1 Descriptive statistics

In this section, we'll look at measures of central tendency, variability, and distribution shape for continuous variables. For illustrative purposes, we'll use several of the variables from the Motor Trend Car Road Tests (`mtcars`) dataset you first saw in chapter 1. Our focus will be on miles per gallon (`mpg`), horsepower (`hp`), and weight (`wt`):

```
> myvars <- c("mpg", "hp", "wt")
> head(mtcars[myvars])
      mpg  hp  wt
Mazda RX4      21.0 110 2.62
Mazda RX4 Wag  21.0 110 2.88
Datsun 710     22.8  93 2.32
Hornet 4 Drive 21.4 110 3.21
Hornet Sportabout 18.7 175 3.44
Valiant       18.1 105 3.46
```

First, we'll look at descriptive statistics for all 32 cars. Then we'll examine descriptive statistics by transmission type (`am`) engine cylinder configuration (`vs`). The former is coded 0=automatic, 1>manual, and the later is coded 0=V-shape and 1=straight.

7.1.1 A menagerie of methods

When it comes to calculating descriptive statistics, R has an embarrassment of riches. Let's start with functions that are included in the base installation. Then we'll look at extensions that are available through the use of user-contributed packages.

In the base installation, you can use the `summary()` function to obtain descriptive statistics. An example is presented in the following listing.

Listing 7.1 Descriptive statistics via `summary()`

```
> myvars <- c("mpg", "hp", "wt")
> summary(mtcars[myvars])
      mpg      hp      wt
```

```

Min. :10.4 Min. :52.0 Min. :1.51
1st Qu.:15.4 1st Qu.:96.5 1st Qu.:2.58
Median :19.2 Median :123.0 Median :3.33
Mean :20.1 Mean :146.7 Mean :3.22
3rd Qu.:22.8 3rd Qu.:180.0 3rd Qu.:3.61
Max. :33.9 Max. :335.0 Max. :5.42

```

The `summary()` function provides the minimum, maximum, quartiles, and mean for numerical variables and frequencies for factors and logical vectors. You can use the `apply()` and `sapply()` function from chapter 5 to provide any descriptive statistics you choose. The `apply()` function is used with matrices and the `sapply()` function is used with data frames. The format for the for the `sapply()` function is

```
sapply(x, FUN, options)
```

where `x` is the data frame and `FUN` is an arbitrary function. If `options` are present, they're passed to `FUN`. Typical functions that you can plug in here are `mean()`, `sd()`, `var()`, `min()`, `max()`, `median()`, `length()`, `range()`, and `quantile()`. The function `fivenum()` returns Tukey's five-number summary (minimum, lower-hinge, median, upper-hinge, and maximum).

Surprisingly, the base installation doesn't provide functions for skew and kurtosis, but you can add your own. The example in the next listing provides several descriptive statistics, including skew and kurtosis.

Listing 7.2 Descriptive statistics via `sapply()`

```

> mystats <- function(x, na.omit=FALSE){
  if (na.omit)
    x <- x[!is.na(x)]
  m <- mean(x)
  n <- length(x)
  s <- sd(x)
  skew <- sum((x-m)^3/s^3)/n
  kurt <- sum((x-m)^4/s^4)/n - 3
  return(c(n=n, mean=m, stdev=s,
          skew=skew, kurtosis=kurt))
}

> myvars <- c("mpg", "hp", "wt")
> sapply(mtcars[myvars], mystats)
      mpg      hp      wt
n      32.000   32.000  32.0000
mean   20.091  146.688   3.2172
stdev   6.027   68.563   0.9785
skew    0.611   0.726   0.4231
kurtosis -0.373  -0.136  -0.0227

```

For cars in this sample, the mean mpg is 20.1, with a standard deviation of 6.0. The distribution is skewed to the right (+0.61) and is somewhat flatter than a normal distribution (-0.37). This is most evident if you graph the data. Note that if you wanted to omit missing values, you could use `sapply(mtcars[myvars], mystats, na.omit=TRUE)`.

7.1.2 Even more methods

Several user-contributed packages offer functions for descriptive statistics, including `Hmisc`, `pastecs`, `psych`, `skimr`, and `summytools`. Due to space limitations, we'll only demonstrate the first three, but you can generate useful summaries with any of the five. Because these packages aren't included in the base distribution, you'll need to install them on first use (see section 1.4).

The `describe()` function in the `Hmisc` package returns the number of variables and observations, the number of missing and unique values, the mean, quantiles, and the five highest and lowest values. An example is provided in the following listing.

Listing 7.3 Descriptive statistics via `describe()` in the `Hmisc` package

```
> library(Hmisc)
> myvars <- c("mpg", "hp", "wt")
> describe(mtcars[myvars])

3 Variables   32 Observations
-----
mpg
n missing unique Mean .05 .10 .25 .50 .75 .90 .95
32  0  25  20.09 12.00 14.34 15.43 19.20 22.80 30.09 31.30

lowest : 10.4 13.3 14.3 14.7 15.0, highest: 26.0 27.3 30.4 32.4 33.9
-----
hp
n missing unique Mean .05 .10 .2 .50 .75 .90 .95
32  0  22  146.7 63.65 66.00 96.50 123.00 180.00 243.50 253.55

lowest : 52 62 65 66 91, highest: 215 230 245 264 335
-----
wt
n missing unique Mean .05 .10 .25 .50 .75 .90 .95
32  0  29  3.217 1.736 1.956 2.581 3.325 3.610 4.048 5.293

lowest : 1.513 1.615 1.835 1.935 2.140, highest: 3.845 4.070 5.250 5.345 5.424
-----
```

The `pastecs` package includes a function named `stat.desc()` that provides a wide range of descriptive statistics. The format is

```
stat.desc(x, basic=TRUE, desc=TRUE, norm=FALSE, p=0.95)
```

where `x` is a data frame or time series. If `basic=TRUE` (the default), the number of values, null values, missing values, minimum, maximum, range, and sum are provided. If `desc=TRUE` (also the default), the median, mean, standard error of the mean, 95% confidence interval for the mean, variance, standard deviation, and coefficient of variation are also provided. Finally, if `norm=TRUE` (not the default), normal distribution statistics are returned, including skewness and kurtosis (and their statistical significance) and the Shapiro–Wilk test of normality. A `p`-value option is used to calculate the confidence interval for the mean (.95 by default). The next listing gives an example.

Listing 7.4 Descriptive statistics via `stat.desc()` in the `pastecs` package

```

> library(pastecs)
> myvars <- c("mpg", "hp", "wt")
> stat.desc(mtcars[myvars])
      mpg  hp  wt
nbr.val 32.00 32.000 32.000
nbr.null 0.00 0.000 0.000
nbr.na   0.00 0.000 0.000
min      10.40 52.000 1.513
max      33.90 335.000 5.424
range    23.50 283.000 3.911
sum      642.90 4694.000 102.952
median   19.20 123.000 3.325
mean     20.09 146.688 3.217
SE.mean  1.07 12.120 0.173
CI.mean.0.95 2.17 24.720 0.353
var      36.32 4700.867 0.957
std.dev  6.03 68.563 0.978
coef.var 0.30 0.467 0.304

```

As if this isn't enough, the `psych` package also has a function called `describe()` that provides the number of nonmissing observations, mean, standard deviation, median, trimmed mean, median absolute deviation, minimum, maximum, range, skew, kurtosis, and standard error of the mean. You can see an example in the following listing.

Listing 7.5 Descriptive statistics via `describe()` in the `psych` package

```

> library(psych)
Attaching package: 'psych'
  The following object(s) are masked from package:Hmisc :
    describe
> myvars <- c("mpg", "hp", "wt")
> describe(mtcars[myvars])
  var n mean sd median trimmed mad min max
mpg 1 32 20.09 6.03 19.20 19.70 5.41 10.40 33.90
hp  2 32 146.69 68.56 123.00 141.19 77.10 52.00 335.00
wt  3 32 3.22 0.98 3.33 3.15 0.77 1.51 5.42
  range skew kurtosis se
mpg 23.50 0.61 -0.37 1.07
hp  283.00 0.73 -0.14 12.12
wt  3.91 0.42 -0.02 0.17

```

I told you that it was an embarrassment of riches!

NOTE In the previous examples, the packages `psych` and `Hmisc` both provide a function named `describe()`. How does R know which one to use? Simply put, the package last loaded takes precedence, as shown in listing 7.5. Here, `psych` is loaded after `Hmisc`, and a message is printed indicating that the `describe()` function in `Hmisc` is masked by the function in `psych`. When you type in the `describe()` function and R searches for it, R comes to the `psych` package first and executes it. If you want the `Hmisc` version instead, you can type `Hmisc::describe(mt)`. The function is still there. You have to give R more information to find it.

Now that you know how to generate descriptive statistics for the data as a whole, let's review how to obtain statistics for subgroups of the data.

7.1.3 Descriptive statistics by group

When comparing groups of individuals or observations, the focus is usually on the descriptive statistics of each group, rather than the total sample. Group statistics can be generated using base R's `by()` function. The format is

```
by(data, INDICES, FUN)
```

where `data` is a data frame or matrix, `INDICES` is a factor or list of factors that defines the groups, and `FUN` is an arbitrary function that operates on all the columns of a data frame. The next listing provides an example.

Listing 7.6 Descriptive statistics by group using `by()`

```
> dstats <- function(x) sapply(x, mystats)
> myvars <- c("mpg", "hp", "wt")
> by(mtcars[myvars], mtcars$am, dstats)
```

```
mtcars$am: 0
      mpg   hp  wt
n    19.000 19.0000 19.000
mean  17.147 160.2632 3.769
stdev  3.834  53.9082 0.777
skew   0.014  -0.0142 0.976
kurtosis -0.803 -1.2097 0.142
-----
```

```
mtcars$am: 1
      mpg   hp  wt
n    13.0000 13.000 13.000
mean  24.3923 126.846 2.411
stdev  6.1665  84.062 0.617
skew   0.0526  1.360 0.210
kurtosis -1.4554  0.563 -1.174
```

In this case, `dstats()` applies the `mystats()` function from listing 7.2 to each column of the data frame. Placing it in the `by()` function gives you summary statistics for each level of `am`.

In the next example (listing 7.7), summary statistics are generated for two `by` variables (`am` and `vs`) and the results for each group are printed with custom labels. Additionally, missing values are omitted before calculating statistics.

Listing 7.7 Descriptive statistics for groups defined by multiple variables

```
> dstats <- function(x) sapply(x, mystats, na.omit=TRUE)
> myvars <- c("mpg", "hp", "wt")
> by(mtcars[myvars],
     list(Transmission=mtcars$am,
          Engine=mtcars$vs),
     FUN=dstats)
```

```
Transmission: 0
```

```

Engine: 0
  mpg   hp   wt
n 12.000000 12.000000 12.000000
mean 15.050000 194.166667 4.1040833
stdev 2.7743959 33.3598379 0.7683069
skew -0.2843325 0.2785849 0.8542070
kurtosis -0.9635443 -1.4385375 -1.1433587
-----
Transmission: 1
Engine: 0
  mpg   hp   wt
n 5.0000000 6.0000000 6.0000000
mean 19.5000000 180.8333333 2.85750000
stdev 4.4294469 98.8158219 0.48672117
skew 0.3135121 0.4842372 0.01270294
kurtosis -1.7595065 -1.7270981 -1.40961807
-----
Transmission: 0
Engine: 1
  mpg   hp   wt
n 7.0000000 7.0000000 7.0000000
mean 20.7428571 102.1428571 3.1942857
stdev 2.4710707 20.9318622 0.3477598
skew 0.1014749 -0.7248459 -1.1532766
kurtosis -1.7480372 -0.7805708 -0.1170979
-----
Transmission: 1
Engine: 1
  mpg   hp   wt
n 7.0000000 7.0000000 7.0000000
mean 28.3714286 80.5714286 2.0282857
stdev 4.7577005 24.1444068 0.4400840
skew -0.3474537 0.2609545 0.4009511
kurtosis -1.7290639 -1.9077611 -1.3677833

```

Although the previous examples used the `mystats()` function, you could have used the `describe()` function from the `Hmisc` and `psych` packages, or the `stat.desc()` function from the `pastecs` package. In fact, the `by()` function provides a general mechanism for repeated *any* analysis by subgroups.

7.1.4 Summarizing data interactively with `dplyr`

So far, we've focused on methods that generate a comprehensive set of descriptive statistics for a given data frame. However, in interactive, exploratory data analyses, our goal is to answer targeted questions. In this case, we'll want to obtain a limited number of statistics on specific groups of observations.

The `dplyr` package, introduced in section 3.11, provides us with tools to quickly and flexibly accomplish this. The `summarize()`, and `summarize_all()` functions can be used to calculate any statistic, and the `group_by()` function can be used to specify the groups on which to calculate those statistics.

As a demonstration, let's ask and answer a set of questions using the `Salaries` data frame in the `carData` package. The dataset contains 2008-2009 9-month salaries in US dollars (`salary`) for 397 faculty members at a university in the United States. The data were collected as part of ongoing efforts to monitor salary differences between male and female faculty.

Before continuing, be sure that the `carData` and `dplyr` packages are installed (`install.packages(c("carData", "dplyr"))`). Then load the packages.

```
library(dplyr)
library(carData)
```

We're now ready to interrogate the data.

(1) What is the median salary and salary range for the 397 professors?
> `Salaries %>%`

```
summarize(med = median(salary),
           min = min(salary),
           max = max(salary))

  med min max
1 107300 57800 231545
```

The `Salaries` dataset is passed to the `summarize()` function, which calculates the median, minimum and maximum value for salary and returns the result as a one row tibble (data frame). The median 9-month salary is \$107,300 and at least one person was making more than \$230,000. I clearly need to ask for a raise.

(2) What is the faculty count, median salary, and salary range by sex and rank?

```
> Salaries %>%
  group_by(rank, sex) %>%
  summarize(n = length(salary),
            med = median(salary),
            min = min(salary),
            max = max(salary))

  rank sex    n med min max
<fct> <fct> <int> <dbl> <int> <int>
1 AsstProf Female 11 77000 63100 97032
2 AsstProf Male 56 80182 63900 95079
3 AssocProf Female 10 90556.62884 109650
4 AssocProf Male 54 95626.70000 126431
5 Prof Female 18 120258.90450 161101
6 Prof Male 248 123996 57800 231545
```

When categorical variables are specified in a `by_group()` statement, the `summarize()` function generates a row of statistics for each combination of their levels. Women have a

lower median salary than men within each faculty rank. In addition, there are a very large number of male full professors at this university.

(3) What is the mean years of service and years since Ph.D. for faculty by sex and rank?

```
> Salaries %>%
  group_by(rank, sex) %>%
  select(yrs.service, yrs.since.phd) %>%
  summarize_all(mean)

  rank  sex  yrs.service yrs.since.phd
  <fct> <fct> <dbl>         <dbl>
1 AsstProf Female    2.55         5.64
2 AsstProf Male     2.34          5
3 AssocProf Female  11.5         15.5
4 AssocProf Male   12.0         15.4
5 Prof Female    17.1         23.7
6 Prof Male     23.2         28.6
```

The `summarize_all()` function calculates a summary statistic each non-grouping variable (`yrs.service` and `yrs.since.phd` here). If you want more than one statistic for each variable, provide them in a list. For example, `summarize_all(list(mean=mean, std=sd))` would calculate the mean and standard deviation for each variable. Men and women have comparable experience histories at the Assistant and Associate Professor levels. However, female Full Professors have fewer years of experience than their male counterparts.

One advantage of the `dplyr` approach is that results are returned as tibbles (data frames). This allows you to analyze these summary results further, plot them, and reformat them for printing. It also provides an easy mechanism for aggregating data.

In general, data analysts have their own preferences for which descriptive statistics to display and how they like to see them formatted. This is probably why there are many variations available. Choose the one that works best for you, or create your own!

7.1.5 Visualizing results

Numerical summaries of a distribution's characteristics are important, but they're no substitute for a visual representation. For quantitative variables, you have histograms (section 6.4), density plots (section 6.5), box plots (section 6.6), and dot plots (section 6.7). They can provide insights that are easily missed by reliance on a small set of descriptive statistics.

The functions considered so far provide summaries of quantitative variables. The functions in the next section allow you to examine the distributions of categorical variables.

7.2 Frequency and contingency tables

In this section, we'll look at frequency and contingency tables from categorical variables, along with tests of independence, measures of association, and methods for graphically displaying

results. We'll be using functions in the basic installation, along with functions from the `vcd` and `gmodels` packages. In the following examples, assume that A, B, and C represent categorical variables.

The data for this section come from the `Arthritis` dataset included with the `vcd` package. The data are from Kock & Edward (1988) and represent a double-blind clinical trial of new treatments for rheumatoid arthritis. Here are the first few -observations:

```
> library(vcd)
> head(Arthritis)
  ID Treatment Sex Age Improved
1  57 Treated Male  27      Some
2  46 Treated Male  29      None
3  77 Treated Male  30      None
4  17 Treated Male  32     Marked
5  36 Treated Male  46     Marked
6  23 Treated Male  58     Marked
```

Treatment (Placebo, Treated), Sex (Male, Female), and Improved (None, Some, Marked) are all categorical factors. In the next section, you'll create frequency and contingency tables (cross-classifications) from the data.

7.2.1 Generating frequency tables

R provides several methods for creating frequency and contingency tables. The most important functions are listed in table 7.1.

Table 7.1 Functions for creating and manipulating contingency tables

Function	Description
<code>table(var1, var2, ..., varN)</code>	Creates an N-way contingency table from N categorical variables (factors)
<code>xtabs(formula, data)</code>	Creates an N-way contingency table based on a formula and a matrix or data frame
<code>prop.table(table, margins)</code>	Expresses table entries as fractions of the marginal table defined by the <code>margins</code>
<code>margin.table(table, margins)</code>	Computes the sum of table entries for a marginal table defined

	by the margins
<code>addmargins(table, margins)</code>	Puts summary margins (sums by default) on a table
<code>ftable(table)</code>	Creates a compact, “flat” contingency table

In the following sections, we’ll use each of these functions to explore categorical variables. We’ll begin with simple frequencies, followed by two-way contingency tables, and end with multiway contingency tables. The first step is to create a table using either the `table()` or `xtabs()` function and then manipulate it using the other functions.

ONE-WAY TABLES

You can generate simple frequency counts using the `table()` function. Here’s an example:

```
> mytable <- with(Arthritis, table(Improved))
> mytable
Improved
None Some Marked
42 14 28
```

You can turn these frequencies into proportions with `prop.table()`

```
> prop.table(mytable)
Improved
None Some Marked
0.500 0.167 0.333
```

or into percentages using `prop.table()*100`:

```
> prop.table(mytable)*100
Improved
None Some Marked
50.0 16.7 33.3
```

Here you can see that 50% of study participants had some or marked improvement (16.7 + 33.3).

TWO-WAY TABLES

For two-way tables, the format for the `table()` function is

```
mytable <- table(A, B)
```

where `A` is the row variable and `B` is the column variable. Alternatively, the `xtabs()` function allows you to create a contingency table using formula-style input. The format is

```
mytable <- xtabs(~ A + B, data=mydata)
```

where *mydata* is a matrix or data frame. In general, the variables to be cross-classified appear on the right of the formula (that is, to the right of the `~`) separated by `+` signs. If a variable is included on the left side of the formula, it's assumed to be a vector of frequencies (useful if the data have already been tabulated).

For the `Arthritis` data, you have

```
> mytable <- xtabs(~ Treatment+Improved, data=Arthritis)
> mytable
      Improved
Treatment None Some Marked
Placebo  29   7   7
Treated  13   7  21
```

You can generate marginal frequencies and proportions using the `margin.table()` and `prop.table()` functions, respectively. For row sums and row proportions, you have

```
> margin.table(mytable, 1)
Treatment
Placebo Treated
  43    41
> prop.table(mytable, 1)
      Improved
Treatment None Some Marked
Placebo 0.674 0.163 0.163
Treated 0.317 0.171 0.512
```

The index (1) refers to the first variable in the `xtabs()` statement –the row variable. The proportions in each row add up to one. Looking at the table, you can see that 51% of treated individuals had marked improvement, compared to 16% of those receiving a placebo.

For column sums and column proportions, you have

```
> margin.table(mytable, 2)
Improved
None Some Marked
  42  14  28
> prop.table(mytable, 2)
      Improved
Treatment None Some Marked
Placebo 0.690 0.500 0.250
Treated 0.310 0.500 0.750
```

Here, the index (2) refers to the second variable in the `xtabs()` statement – i.e., the columns. The proportions in each column add up to one.

Cell proportions are obtained with this statement:

```
> prop.table(mytable)
      Improved
Treatment None Some Marked
Placebo 0.3452 0.0833 0.0833
Treated 0.1548 0.0833 0.2500
```

The sum of all the cell proportions add up to one.

You can use the `addmargins()` function to add marginal sums to these tables. For example, the following code adds a Sum row and column:

```
> addmargins(mytable)
      Improved
Treatment None Some Marked Sum
Placebo   29   7   7   43
Treated   13   7  21   41
Sum       42  14  28   84
> addmargins(prop.table(mytable))
      Improved
Treatment None Some Marked Sum
Placebo 0.3452 0.0833 0.0833 0.5119
Treated 0.1548 0.0833 0.2500 0.4881
Sum     0.5000 0.1667 0.3333 1.0000
```

When using `addmargins()`, the default is to create sum margins for all variables in a table. In contrast, the following code adds a Sum column alone:

```
> addmargins(prop.table(mytable, 1), 2)
      Improved
Treatment None Some Marked Sum
Placebo 0.674 0.163 0.163 1.000
Treated 0.317 0.171 0.512 1.000
```

Similarly, this code adds a Sum row:

```
> addmargins(prop.table(mytable, 2), 1)
      Improved
Treatment None Some Marked
Placebo 0.690 0.500 0.250
Treated 0.310 0.500 0.750
Sum     1.000 1.000 1.000
```

In the table, you see that 25% of those patients with marked improvement received a placebo.

NOTE The `table()` function ignores missing values (NAs) by default. To include NA as a valid category in the frequency counts, include the table option `useNA="ifany"`.

A third method for creating two-way tables is the `CrossTable()` function in the `gmodels` package. The `CrossTable()` function produces two-way tables modeled after `PROC FREQ` in SAS or `CROSSTABS` in SPSS. The following listing shows an example.

Listing 7.8 Two-way table using `CrossTable`

```
> library(gmodels)
> CrossTable(Arthritis$Treatment, Arthritis$Improved)
```

Cell Contents

```

|-----|
|      N |
| Chi-square contribution |
|      N / Row Total |
|      N / Col Total |
|      N / Table Total |
|-----|

```

Total Observations in Table: 84

Arthritis\$Treatment	Arthritis\$Improved			Row Total
	None	Some	Marked	
Placebo	29	7	7	43
	2.616	0.004	3.752	
	0.674	0.163	0.163	0.512
	0.690	0.500	0.250	
	0.345	0.083	0.083	
Treated	13	7	21	41
	2.744	0.004	3.935	
	0.317	0.171	0.512	0.488
	0.310	0.500	0.750	
	0.155	0.083	0.250	
Column Total	42	14	28	84
	0.500	0.167	0.333	

The `CrossTable()` function has options to report percentages (row, column, and cell); specify decimal places; produce chi-square, Fisher, and McNemar tests of independence; report expected and residual values (Pearson, standardized, and adjusted standardized); include missing values as valid; annotate with row and column titles; and format as SAS or SPSS style output. See `help(CrossTable)` for details.

If you have more than two categorical variables, you're dealing with multidimensional tables. We'll consider these next.

MULTIDIMENSIONAL TABLES

Both `table()` and `xtabs()` can be used to generate multidimensional tables based on three or more categorical variables. The `margin.table()`, `prop.table()`, and `addmargins()` functions extend naturally to more than two dimensions. Additionally, the `ftable()` function can be used to print multidimensional tables in a compact and attractive manner. An example is given in the next listing.

Listing 7.9 Three-way contingency table

```

> mytable <- xtabs(~ Treatment+Sex+Improved, data=Arthritis) #1
> mytable
, , Improved = None

Sex

```

```

Treatment Female Male
Placebo    19  10
Treated    6   7

,, Improved = Some

      Sex
Treatment Female Male
Placebo    7   0
Treated    5   2

,, Improved = Marked

      Sex
Treatment Female Male
Placebo    6   1
Treated   16   5

> ftable(mytable)
      Sex Female Male
Treatment Improved
Placebo None      19  10
      Some      7   0
      Marked    6   1
Treated None      6   7
      Some      5   2
      Marked   16   5

> margin.table(mytable, 1)           #2

Treatment
Placebo Treated
  43    41
> margin.table(mytable, 2)
Sex
Female  Male
  59    25
> margin.table(mytable, 3)
Improved
None  Some  Marked
  42   14   28
> margin.table(mytable, c(1, 3))     #3
Improved
Treatment None Some Marked
Placebo  29   7   7
Treated  13   7  21
> ftable(prop.table(mytable, c(1, 2))) #4
      Improved None  Some  Marked
Treatment Sex
Placebo Female  0.594 0.219 0.188
      Male   0.909 0.000 0.091
Treated Female  0.222 0.185 0.593
      Male   0.500 0.143 0.357

> ftable(addmargins(prop.table(mytable, c(1, 2)), 3))
      Improved None  Some  Marked  Sum
Treatment Sex

```

Placebo	Female	0.594	0.219	0.188	1.000
	Male	0.909	0.000	0.091	1.000
Treated	Female	0.222	0.185	0.593	1.000
	Male	0.500	0.143	0.357	1.000

#1 Cell frequencies

#2 Marginal frequencies

#3 Treatment × Improved marginal frequencies

#4 Improved proportions for Treatment × Sex

The code at #1 produces cell frequencies for the three-way classification. The code also demonstrates how the `fTable()` function can be used to print a more compact and attractive version of the table.

The code at #2 produces the marginal frequencies for Treatment, Sex, and Improved. Because you created the table with the formula `~Treatment+Sex + Improved`, Treatment is referred to by index 1, Sex is referred to by index 2, and Improved is referred to by index 3.

The code at #3 produces the marginal frequencies for the Treatment × Improved classification, summed over Sex. The proportion of patients with `None`, `Some`, and `Marked` improvement for each Treatment × Sex combination is provided in #4. Here you see that 36% of treated males had marked improvement, compared to 59% of treated females. In general, the proportions will add to 1 over the indices not included in the `prop.table()` call (the third index, or Improved in this case). You can see this in the last example, where you add a sum margin over the third index.

If you want percentages instead of proportions, you can multiply the resulting table by 100. For example, this statement

```
fTable(addmargins(prop.table(mytable, c(1, 2)), 3)) * 100
```

produces this table:

	Sex	Female	Male	Sum
Treatment Improved				
Placebo	None	65.5	34.5	100.0
	Some	100.0	0.0	100.0
	Marked	85.7	14.3	100.0
Treated	None	46.2	53.8	100.0
	Some	71.4	28.6	100.0
	Marked	76.2	23.8	100.0

Contingency tables tell you the frequency or proportions of cases for each combination of the variables that make up the table, but you're probably also interested in whether the variables in the table are related or independent. Tests of independence are covered in the next section.

7.2.2 Tests of independence

R provides several methods of testing the independence of categorical variables. The three tests described in this section are the chi-square test of independence, the Fisher exact test, and the Cochran-Mantel-Haenszel test.

CHI-SQUARE TEST OF INDEPENDENCE

You can apply the function `chisq.test()` to a two-way table in order to produce a chi-square test of independence of the row and column variables. See the next listing for an example.

Listing 7.10 Chi-square test of independence

```
> library(vcd)
> mytable <- xtabs(~Treatment+Improved, data=Arthritis)
> chisq.test(mytable)
  Pearson's Chi-squared test
data: mytable
X-squared = 13.1, df = 2, p-value = 0.001463      #1

> mytable <- xtabs(~Improved+Sex, data=Arthritis)
> chisq.test(mytable)
  Pearson's Chi-squared test
data: mytable
X-squared = 4.84, df = 2, p-value = 0.0889      #2

Warning message:
In chisq.test(mytable) : Chi-squared approximation may be incorrect
```

#1 Treatment and Improved aren't independent.

#2 Gender and Improved are independent.

From the results #1, there appears to be a relationship between treatment received and level of improvement ($p < .01$). But there doesn't appear to be a relationship #2 between patient sex and improvement ($p > .05$). The p-values are the probability of obtaining the sampled results, assuming independence of the row and column variables in the population. Because the probability is small for #1, you reject the hypothesis that treatment type and outcome are independent. Because the probability for #2 isn't small, it's not unreasonable to assume that outcome and gender are independent. The warning message in listing 7.10 is produced because one of the six cells in the table (male-some improvement) has an expected value less than five, which may invalidate the chi-square approximation.

FISHER'S EXACT TEST

You can produce a Fisher's exact test via the `fisher.test()` function. Fisher's exact test evaluates the null hypothesis of independence of rows and columns in a contingency table with fixed marginals. The format is `fisher.test(mytable)`, where `mytable` is a two-way table. Here's an example:

```
> mytable <- xtabs(~Treatment+Improved, data=Arthritis)
> fisher.test(mytable)
  Fisher's Exact Test for Count Data
data: mytable
p-value = 0.001393
alternative hypothesis: two.sided
```

In contrast to many statistical packages, the `fisher.test()` function can be applied to any two-way table with two or more rows and columns, not just a 2×2 table.

COCHRAN–MANTEL–HAENSZEL TEST

The `mantelhaen.test()` function provides a Cochran–Mantel–Haenszel chi-square test of the null hypothesis that two nominal variables are conditionally independent in each stratum of a third variable. The following code tests the hypothesis that the Treatment and Improved variables are independent within each level for Sex. The test assumes that there’s no three-way (Treatment \times Improved \times Sex) interaction:

```
> mytable <- xtabs(~Treatment+Improved+Sex, data=Arthritis)
> mantelhaen.test(mytable)
Cochran-Mantel-Haenszel test
data: mytable
Cochran-Mantel-Haenszel M^2 = 14.6, df = 2, p-value = 0.0006647
```

The results suggest that the treatment received and the improvement reported aren’t independent within each level of Sex (that is, treated individuals improved more than those receiving placebos when controlling for sex).

7.2.3 Measures of association

The significance tests in the previous section evaluate whether sufficient evidence exists to reject a null hypothesis of independence between variables. If you can reject the null hypothesis, your interest turns naturally to measures of association in order to gauge the strength of the relationships present. The `assocstats()` function in the `vcd` package can be used to calculate the phi coefficient, contingency coefficient, and Cramer’s V for a two-way table. An example is given in the following listing.

Listing 7.11 Measures of association for a two-way table

```
> library(vcd)
> mytable <- xtabs(~Treatment+Improved, data=Arthritis)
> assocstats(mytable)
      X^2 df P(> X^2)
Likelihood Ratio 13.530  2 0.0011536
Pearson          13.055  2 0.0014626

Phi-Coefficient : 0.394
Contingency Coeff.: 0.367
Cramer's V      : 0.394
```

In general, larger magnitudes indicate stronger associations. The `vcd` package also provides a `kappa()` function that can calculate Cohen’s kappa and weighted kappa for a confusion matrix (for example, the degree of agreement between two judges classifying a set of objects into categories).

7.2.4 Visualizing results

R has mechanisms for visually exploring the relationships among categorical variables that go well beyond those found in most other statistical platforms. You typically use bar charts to visualize frequencies in one dimension (see section 6.1). The `vcd` package has excellent functions for visualizing relationships among categorical variables in multidimensional datasets using mosaic and association plots (see section 11.4). Finally, correspondence-analysis functions in the `ca` package allow you to visually explore relationships between rows and columns in contingency tables using various geometric representations (Nenadic and Greenacre, 2007).

This ends the discussion of contingency tables, until we take up more advanced topics in chapters 11 and 15. Next, let's look at various types of correlation coefficients.

7.3 Correlations

Correlation coefficients are used to describe relationships among quantitative variables. The sign (plus or minus) indicates the direction of the relationship (positive or inverse), and the magnitude indicates the strength of the relationship (ranging from 0 for no relationship to 1 for a perfectly predictable relationship).

In this section, we'll look at a variety of correlation coefficients, as well as tests of significance. We'll use the `state.x77` dataset available in the base R installation. It provides data on the population, income, illiteracy rate, life expectancy, murder rate, and high school graduation rate for the 50 US states in 1977. There are also temperature and land-area measures, but we'll drop them to save space. Use `help(state.x77)` to learn more about the file. In addition to the base installation, we'll be using the `psych` and `ggm` packages.

7.3.1 Types of correlations

R can produce a variety of correlation coefficients, including Pearson, Spearman, Kendall, partial, polychoric, and polyserial. Let's look at each in turn.

PEARSON, SPEARMAN, AND KENDALL CORRELATIONS

The Pearson product-moment correlation assesses the degree of linear relationship between two quantitative variables. Spearman's rank-order correlation coefficient assesses the degree of relationship between two rank-ordered variables. Kendall's tau is also a nonparametric measure of rank correlation.

The `cor()` function produces all three correlation coefficients, whereas the `cov()` function provides covariances. There are many options, but a simplified format for producing correlations is

```
cor(x, use= , method= )
```

The options are described in table 7.2.

Table 7.2 `cor/cov` options

Option	Description
X	Matrix or data frame.
Use	Specifies the handling of missing data. The options are <code>all.obs</code> (assumes no missing data—missing data will produce an error), <code>everything</code> (any correlation involving a case with missing values will be set to <code>missing</code>), <code>complete.obs</code> (listwise deletion), and <code>pairwise.complete.obs</code> (pairwise deletion).
Method	Specifies the type of correlation. The options are <code>pearson</code> , <code>spearman</code> , and <code>kendall</code> .

The default options are `use="everything"` and `method="pearson"`. You can see an example in the following listing.

Listing 7.12 Covariances and correlations

```
> states<- state.x77[,1:6]
> cov(states)
      Population Income Illiteracy Life Exp Murder HS Grad
Population 19931684 571230  292.868 -407.842 5663.52 -3551.51
Income      571230 377573 -163.702 280.663 -521.89 3076.77
Illiteracy   293 -164  0.372 -0.482  1.58 -3.24
Life Exp    -408 281  -0.482  1.802 -3.87  6.31
Murder       5664 -522  1.582 -3.869 13.63 -14.55
HS Grad     -3552 3077  -3.235  6.313 -14.55  65.24

> cor(states)
      Population Income Illiteracy Life Exp Murder HS Grad
Population  1.0000 0.208  0.108 -0.068 0.344 -0.0985
Income      0.2082 1.000  -0.437 0.340 -0.230 0.6199
Illiteracy   0.1076 -0.437  1.000 -0.588 0.703 -0.6572
Life Exp    -0.0681 0.340  -0.588  1.000 -0.781 0.5822
Murder       0.3436 -0.230  0.703 -0.781 1.000 -0.4880
HS Grad     -0.0985 0.620  -0.657 0.582 -0.488 1.0000
> cor(states, method="spearman")
      Population Income Illiteracy Life Exp Murder HS Grad
Population  1.000 0.125  0.313 -0.104 0.346 -0.383
Income      0.125 1.000  -0.315 0.324 -0.217 0.510
Illiteracy   0.313 -0.315  1.000 -0.555 0.672 -0.655
Life Exp    -0.104 0.324  -0.555  1.000 -0.780 0.524
Murder       0.346 -0.217  0.672 -0.780 1.000 -0.437
HS Grad     -0.383 0.510  -0.655 0.524 -0.437 1.000
```

The first call produces the variances and covariances. The second provides Pearson product-moment correlation coefficients, and the third produces Spearman rank-order correlation coefficients. You can see, for example, that a strong positive correlation exists between income and high school graduation rate and that a strong negative correlation exists between illiteracy rates and life expectancy.

Notice that you get square matrices by default (all variables crossed with all other variables). You can also produce nonsquare matrices, as shown in the following example:

```
> x <- states[,c("Population", "Income", "Illiteracy", "HS Grad")]
> y <- states[,c("Life Exp", "Murder")]
> cor(x,y)
      Life Exp Murder
Population -0.068 0.344
Income    0.340 -0.230
Illiteracy -0.588 0.703
HS Grad   0.582 -0.488
```

This version of the function is particularly useful when you're interested in the relationships between one set of variables and another. Notice that the results don't tell you if the correlations differ significantly from 0 (that is, whether there's sufficient evidence based on the sample data to conclude that the population correlations differ from 0). For that, you need tests of significance (described in section 7.3.2).

PARTIAL CORRELATIONS

A *partial* correlation is a correlation between two quantitative variables, controlling for one or more other quantitative variables. You can use the `pcor()` function in the `ggm` package to provide partial correlation coefficients. The `ggm` package isn't installed by default, so be sure to install it on first use. The format is

```
pcor(u, S)
```

where u is a vector of numbers, with the first two numbers being the indices of the variables to be correlated, and the remaining numbers being the indices of the conditioning variables (that is, the variables being partialled out). S is the covariance matrix among the variables. An example will help clarify this:

```
> library(ggm)
> colnames(states)
[1] "Population" "Income" "Illiteracy" "Life Exp" "Murder" "HS Grad"
> pcor(c(1,5,2,3,6), cov(states))
[1] 0.346
```

In this case, 0.346 is the correlation between population (variable 1) and murder rate (variable 5), controlling for the influence of income, illiteracy rate, and high school graduation rate (variables 2, 3, and 6 respectively). The use of partial correlations is common in the social sciences.

OTHER TYPES OF CORRELATIONS

The `hetcor()` function in the `polycor` package can compute a heterogeneous correlation matrix containing Pearson product-moment correlations between numeric variables, polyserial correlations between numeric and ordinal variables, polychoric correlations between ordinal variables, and tetrachoric correlations between two dichotomous variables. Polyserial, polychoric, and tetrachoric correlations assume that the ordinal or dichotomous variables are derived from underlying normal distributions. See the documentation that accompanies this package for more information.

7.3.2 Testing correlations for significance

Once you've generated correlation coefficients, how do you test them for statistical significance? The typical null hypothesis is no relationship (that is, the correlation in the population is 0). You can use the `cor.test()` function to test an individual Pearson, Spearman, and Kendall correlation coefficient. A simplified format is

```
cor.test(x, y, alternative = , method = )
```

where `x` and `y` are the variables to be correlated, `alternative` specifies a two-tailed or one-tailed test ("two.side", "less", or "greater"), and `method` specifies the type of correlation ("pearson", "kendall", or "spearman") to compute. Use `alternative = "less"` when the research hypothesis is that the population correlation is less than 0. Use `alternative = "greater"` when the research hypothesis is that the population correlation is greater than 0. By default, `alternative = "two.side"` (population correlation isn't equal to 0) is assumed. See the following listing for an example.

Listing 7.13 Testing a correlation coefficient for significance

```
> cor.test(states[,3], states[,5])
```

```
    Pearson's product-moment correlation
```

```
data: states[, 3] and states[, 5]
```

```
t = 6.85, df = 48, p-value = 1.258e-08
```

```
alternative hypothesis: true correlation is not equal to 0
```

```
95 percent confidence interval:
```

```
0.528 0.821
```

```
sample estimates:
```

```
cor
0.703
```

This code tests the null hypothesis that the Pearson correlation between life expectancy and murder rate is 0. Assuming that the population correlation is 0, you'd expect to see a sample correlation as large as 0.703 less than 1 time out of 10 million (that is, $p = 1.258e-08$). Given how unlikely this is, you reject the null hypothesis in favor of the research hypothesis, that the population correlation between life expectancy and murder rate is *not* 0.

Unfortunately, you can test only one correlation at a time using `cor.test()`. Luckily, the `corr.test()` function provided in the `psych` package allows you to go further. The

`corr.test()` function produces correlations and significance levels for matrices of Pearson, Spearman, and Kendall correlations. An example is given in the following listing.

Listing 7.14 Correlation matrix and tests of significance via `corr.test()`

```
> library(psych)
> corr.test(states, use="complete")

Call:corr.test(x = states, use = "complete")
Correlation matrix
  Population Income Illiteracy Life Exp Murder HS Grad
Population  1.00 0.21   0.11 -0.07 0.34 -0.10
Income      0.21 1.00  -0.44 0.34 -0.23 0.62
Illiteracy  0.11 -0.44  1.00 -0.59 0.70 -0.66
Life Exp    -0.07 0.34 -0.59 1.00 -0.78 0.58
Murder      0.34 -0.23 0.70 -0.78 1.00 -0.49
HS Grad     -0.10 0.62 -0.66 0.58 -0.49 1.00

Sample Size
[1] 50

Probability value
  Population Income Illiteracy Life Exp Murder HS Grad
Population  0.00 0.15   0.46 0.64 0.01 0.5
Income      0.15 0.00   0.00 0.02 0.11 0.0
Illiteracy  0.46 0.00   0.00 0.00 0.00 0.0
Life Exp    0.64 0.02   0.00 0.00 0.00 0.0
Murder      0.01 0.11   0.00 0.00 0.00 0.0
HS Grad     0.50 0.00   0.00 0.00 0.00 0.0
```

The `use=` options can be "pairwise" or "complete" (for pairwise or listwise deletion of missing values, respectively). The `method=` option is "pearson" (the default), "spearman", or "kendall". Here you see that the correlation between illiteracy and life expectancy (-0.59) is significantly different from zero ($p=0.00$) and suggests that as the illiteracy rate goes up, life expectancy tends to go down. However, the correlation between population size and high school graduation rate (-0.10) is not significantly different from 0 ($p = 0.5$).

OTHER TESTS OF SIGNIFICANCE

In section 7.4.1, we looked at partial correlations. The `pcor.test()` function in the `psych` package can be used to test the conditional independence of two variables controlling for one or more additional variables, assuming multivariate normality. The format is

```
pcor.test(r, q, n)
```

where *r* is the partial correlation produced by the `pcor()` function, *q* is the number of variables being controlled, and *n* is the sample size.

Before leaving this topic, it should be mentioned that the `r.test()` function in the `psych` package also provides a number of useful significance tests. The function can be used to test the following:

- The significance of a correlation coefficient
- The difference between two independent correlations
- The difference between two dependent correlations sharing a single variable
- The difference between two dependent correlations based on completely different variables

See `help(r.test)` for details.

7.3.3 Visualizing correlations

The bivariate relationships underlying correlations can be visualized through scatter plots and scatter plot matrices, whereas correlograms provide a unique and powerful method for comparing a large number of correlation coefficients in a meaningful way. Each is covered in chapter 11.

7.4 T-tests

The most common activity in research is the comparison of two groups. Do patients receiving a new drug show greater improvement than patients using an existing medication? Does one manufacturing process produce fewer defects than another? Which of two teaching methods is most cost-effective? If your outcome variable is categorical, you can use the methods described in section 7.3. Here, we'll focus on group comparisons, where the outcome variable is continuous and assumed to be distributed -normally.

For this illustration, we'll use the `UScrime` dataset distributed with the `MASS` package. It contains information about the effect of punishment regimes on crime rates in 47 US states in 1960. The outcome variables of interest will be `Prob` (the probability of imprisonment), `U1` (the unemployment rate for urban males ages 14–24), and `U2` (the unemployment rate for urban males ages 35–39). The categorical variable `So` (an indicator variable for Southern states) will serve as the grouping variable. The data have been rescaled by the original authors. (Note: I considered naming this section "Crime and Punishment in the Old South," but cooler heads prevailed.)

7.4.1 Independent t-test

Are you more likely to be imprisoned if you commit a crime in the South? The comparison of interest is Southern versus non-Southern states, and the dependent variable is the probability of incarceration. A two-group independent t-test can be used to test the hypothesis that the two population means are equal. Here, you assume that the two groups are independent and that the data is sampled from normal populations. The format is either

```
t.test(y ~ x, data)
```

where `y` is numeric and `x` is a dichotomous variable, or

```
t.test(y1, y2)
```

where y_1 and y_2 are numeric vectors (the outcome variable for each group). The optional `data` argument refers to a matrix or data frame containing the variables. In contrast to most statistical packages, the default test assumes unequal variance and applies the Welch degrees-of-freedom modification. You can add a `var.equal=TRUE` option to specify equal variances and a pooled variance estimate. By default, a two-tailed alternative is assumed (that is, the means differ but the direction isn't specified). You can add the option `alternative="less"` or `alternative="greater"` to specify a directional test.

The following code compares Southern (group 1) and non-Southern (group 0) states on the probability of imprisonment using a two-tailed test without the assumption of equal variances:

```
> library(MASS)
> t.test(Prob ~ So, data=UScrime)

Welch Two Sample t-test

data: Prob by So
t = -3.8954, df = 24.925, p-value = 0.0006506
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.03852569 -0.01187439
sample estimates:
mean in group 0 mean in group 1
 0.03851265  0.06371269
```

You can reject the hypothesis that Southern states and non-Southern states have equal probabilities of imprisonment ($p < .001$).

NOTE Because the outcome variable is a proportion, you might try to transform it to normality before carrying out the t-test. In the current case, all reasonable transformations of the outcome variable ($Y/1-Y$, $\log(Y/1-Y)$, $\arcsin(Y)$, and $\arcsin(\sqrt{Y})$) would lead to the same conclusions. Transformations are covered in detail in chapter 8.

7.4.2 Dependent t-test

As a second example, you might ask if the unemployment rate for younger males (14–24) is greater than for older males (35–39). In this case, the two groups aren't independent. You wouldn't expect the unemployment rate for younger and older males in Alabama to be unrelated. When observations in the two groups are related, you have a dependent-groups design. Pre-post or repeated-measures designs also produce dependent groups.

A dependent t-test assumes that the difference between groups is normally distributed. In this case, the format is

```
t.test(y1, y2, paired=TRUE)
```

where y_1 and y_2 are the numeric vectors for the two dependent groups. The results are as follows:

```

> library(MASS)
> sapply(UScrime[c("U1","U2")], function(x)(c(mean=mean(x),sd=sd(x))))
  U1  U2
mean 95.5 33.98
sd   18.0  8.45

> with(UScrime, t.test(U1, U2, paired=TRUE))

Paired t-test

data:  U1 and U2
t = 32.4066, df = 46, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 57.67003 65.30870
sample estimates:
mean of the differences
 61.48936

```

The mean difference (61.5) is large enough to warrant rejection of the hypothesis that the mean unemployment rate for older and younger males is the same. Younger males have a higher rate. In fact, the probability of obtaining a sample difference this large if the population means are equal is less than 0.00000000000000022 (that is, $2.2e-16$).

7.4.3 When there are more than two groups

What do you do if you want to compare more than two groups? If you can assume that the data are independently sampled from normal populations, you can use analysis of variance (ANOVA). ANOVA is a comprehensive methodology that covers many experimental and quasi-experimental designs. As such, it has earned its own chapter. Feel free to abandon this section and jump to chapter 9 at any time.

7.5 Nonparametric tests of group differences

If you're unable to meet the parametric assumptions of a t-test or ANOVA, you can turn to nonparametric approaches. For example, if the outcome variables are severely skewed or ordinal in nature, you may wish to use the techniques in this section.

7.5.1 Comparing two groups

If the two groups are independent, you can use the Wilcoxon rank sum test (more popularly known as the Mann-Whitney U test) to assess whether the observations are sampled from the same probability distribution (that is, whether the probability of obtaining higher scores is greater in one population than the other). The format is either

```
wilcox.test(y ~ x, data)
```

where y is numeric and x is a dichotomous variable, or

```
wilcox.test(y1, y2)
```

where y_1 and y_2 are the outcome variables for each group. The optional `data` argument refers to a matrix or data frame containing the variables. The default is a two-tailed test. You can add the option `exact` to produce an exact test, and `-alternative="less"` or `alternative="greater"` to specify a directional test.

If you apply the Mann–Whitney U test to the question of incarceration rates from the previous section, you’ll get these results:

```
> with(UScrime, by(Prob, So, median))

So: 0
[1] 0.0382
-----
So: 1
[1] 0.0556

> wilcox.test(Prob ~ So, data=UScrime)

Wilcoxon rank sum test

data: Prob by So
W = 81, p-value = 8.488e-05
alternative hypothesis: true location shift is not equal to 0
```

Again, you can reject the hypothesis that incarceration rates are the same in Southern and non-Southern states ($p < .001$).

The Wilcoxon signed rank test provides a nonparametric alternative to the dependent sample t-test. It’s appropriate in situations where the groups are paired and the assumption of normality is unwarranted. The format is identical to the Mann–Whitney U test, but you add the `paired=TRUE` option. Let’s apply it to the unemployment question from the previous section:

```
> sapply(UScrime[c("U1", "U2")], median)
U1 U2
92 34

> with(UScrime, wilcox.test(U1, U2, paired=TRUE))

Wilcoxon signed rank test with continuity correction

data: U1 and U2
V = 1128, p-value = 2.464e-09
alternative hypothesis: true location shift is not equal to 0
```

Again, you reach the same conclusion reached with the paired t-test.

In this case, the parametric t-tests and their nonparametric equivalents reach the same conclusions. When the assumptions for the t-tests are reasonable, the parametric tests are more powerful (more likely to find a difference if it exists). The nonparametric tests are more appropriate when the assumptions are grossly unreasonable (for example, rank-ordered data).

7.5.2 Comparing more than two groups

When there are more than two groups to be compared, you must turn to other methods. Consider the `state.x77` dataset from section 7.4. It contains population, income, illiteracy rate, life expectancy, murder rate, and high school graduation rate data for US states. What if you want to compare the illiteracy rates in four regions of the country (Northeast, South, North Central, and West)? This is called a *one-way design*, and there are both parametric and nonparametric approaches available to address the question.

If you can't meet the assumptions of ANOVA designs, you can use nonparametric methods to evaluate group differences. If the groups are independent, a Kruskal–Wallis test provides a useful approach. If the groups are dependent (for example, repeated measures or randomized block design), the Friedman test is more appropriate.

The format for the Kruskal–Wallis test is

```
kruskal.test(y ~ A, data)
```

where `y` is a numeric outcome variable and `A` is a grouping variable with two or more levels (if there are two levels, it's equivalent to the Mann–Whitney U test). For the Friedman test, the format is

```
friedman.test(y ~ A | B, data)
```

where `y` is the numeric outcome variable, `A` is a grouping variable, and `B` is a blocking variable that identifies matched observations. In both cases, `data` is an option argument specifying a matrix or data frame containing the variables.

Let's apply the Kruskal–Wallis test to the illiteracy question. First, you'll have to add the region designations to the dataset. These are contained in the dataset `state.region` distributed with the base installation of R:

```
states <- data.frame(state.region, state.x77)
```

Now you can apply the test:

```
> kruskal.test(illiteracy ~ state.region, data=states)
Kruskal-Wallis rank sum test
data: states$illiteracy by states$state.region
Kruskal-Wallis chi-squared = 22.7, df = 3, p-value = 4.726e-05
```

The significance test suggests that the illiteracy rate isn't the same in each of the four regions of the country ($p < .001$).

Although you can reject the null hypothesis of no difference, the test doesn't tell you *which* regions differ significantly from each other. To answer this question, you could compare groups two at a time using the Wilcoxon test. A more elegant approach is to apply a multiple-comparisons procedure that computes all pairwise comparisons, while controlling the type I error rate (the probability of finding a difference that isn't there). I have created a function

called `wmc()` that can be used for this purpose. It compares groups two at a time using the Wilcoxon test and adjusts the probability values using the `p.adjust()` function.

To be honest, I'm stretching the definition of *basic* in the chapter title quite a bit, but because the function fits well here, I hope you'll bear with me. You can download a text file containing `wmc()` from www.statmethods.net/RiA/wmc.txt. The following listing uses this function to compare the illiteracy rates in the four US regions.

Listing 7.15 Nonparametric multiple comparisons

```
> source("http://www.statmethods.net/RiA/wmc.txt") #1
> states <- data.frame(state.region, state.x77)
> wmc(illiteracy ~ state.region, data=states, method="holm")
```

```
Descriptive Statistics #2

      West North Central Northeast South
n    13.00    12.00    9.0 16.00
median 0.60    0.70    1.1 1.75
mad    0.15    0.15    0.3 0.59

Multiple Comparisons (Wilcoxon Rank Sum Tests) #3
Probability Adjustment = holm

      Group.1 Group.2 W p
1      West North Central 88 8.7e-01
2      West Northeast 46 8.7e-01
3      West South 39 1.8e-02 *
4 North Central Northeast 20 5.4e-02 .
5 North Central South 2 8.1e-05 ***
6 Northeast South 18 1.2e-02 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

#1 Accesses the function

#2 Basic statistics

#3 Pairwise comparisons

The `source()` function downloads and executes the R script defining the `wmc()` function #1. The function's format is `wmc(y ~ A, data, method)`, where `y` is a numeric outcome variable, `A` is a grouping variable, `data` is the data frame containing these variables, and `method` is the approach used to limit Type I errors. Listing 7.15 uses an adjustment method developed by Holm (1979). It provides strong control of the family-wise error rate (the probability of making one or more Type I errors in a set of comparisons). See `help(p.adjust)` for a description of the other methods available.

The `wmc()` function first provides the sample sizes, medians, and median absolute deviations for each group #2. The West has the lowest illiteracy rate, and the South has the highest. The function then generates six statistical comparisons (West versus North Central, West versus Northeast, West versus South, North Central versus Northeast, North Central versus South, and Northeast versus South) #3. You can see from the two-sided p-values (`p`)

that the South differs significantly from the other three regions and that the other three regions don't differ from each other at a $p < .05$ level.

Nonparametric multiple comparisons are a useful set of techniques that aren't easily accessible in R. In chapter 21, you'll have an opportunity to expand the `wmc()` function into a fully developed package that includes error checking and informative graphics.

7.6 Visualizing group differences

In sections 7.4 and 7.5, we looked at statistical methods for comparing groups. Examining group differences visually is also a crucial part of a comprehensive data-analysis strategy. It allows you to assess the magnitude of the differences, identify any distributional characteristics that influence the results (such as skew, bimodality, or outliers), and evaluate the appropriateness of the test assumptions. R provides a wide range of graphical methods for comparing groups, including box plots (simple, notched, and violin), covered in section 6.6; overlapping kernel density plots, covered in section 6.5; and graphical methods for visualizing outcomes in an ANOVA framework, discussed in chapter 9. Advanced methods for visualizing group differences, including grouping and faceting, are discussed in chapter 19.

7.7 Summary

- Descriptive statistics are used to describe the distribution a quantitative variable numerically. Many packages in R provide descriptive statistics for data frames. The choice among packages is primarily a matter of personal preference.
- Frequency tables and cross tabulations are used to summarize the distributions of categorical variables.
- The t-tests and the Mann-Whitney U test can be used to compare two groups on a quantitative outcome.
- A chi-square test can be used evaluate the association between two categorical variables. The correlation coefficient is used to evaluate the association between two quantitative variables.
- Numeric summaries and statistical tests should usually be accompanied by data visualizations. Otherwise, import features of the data may be missed.

8

Regression

This chapter covers

- Fitting and interpreting linear models
- Evaluating model assumptions
- Selecting among competing models

In many ways, regression analysis lives at the heart of statistics. It's a broad term for a set of methodologies used to predict a response variable (also called a *dependent*, *criterion*, or *outcome* variable) from one or more predictor variables (also called *independent* or *explanatory* variables). In general, regression analysis can be used to *identify* the explanatory variables that are related to a response variable, to *describe* the form of the relationships involved, and to provide an equation for *predicting* the response variable from the explanatory variables.

For example, an exercise physiologist might use regression analysis to develop an equation for predicting the expected number of calories a person will burn while exercising on a treadmill. The response variable is the number of calories burned (calculated from the amount of oxygen consumed), and the predictor variables might include duration of exercise (minutes), percentage of time spent at their target heart rate, average speed (mph), age (years), gender, and body mass index (BMI).

From a theoretical point of view, the analysis will help answer such questions as these:

- What's the relationship between exercise duration and calories burned? Is it linear or curvilinear? For example, does exercise have less impact on the number of calories burned after a certain point?
- How does effort (the percentage of time at the target heart rate, the average walking speed) factor in?

- Are these relationships the same for young and old, male and female, heavy and slim?

From a practical point of view, the analysis will help answer such questions as the -following:

- How many calories can a 30-year-old man with a BMI of 28.7 expect to burn if he walks for 45 minutes at an average speed of 4 miles per hour and stays within his target heart rate 80% of the time?
- What's the minimum number of variables you need to collect in order to accurately predict the number of calories a person will burn when walking?
- How accurate will your prediction tend to be?

Because regression analysis plays such a central role in modern statistics, we'll cover it in some depth in this chapter. First, we'll look at how to fit and interpret regression models. Next, we'll review a set of techniques for identifying potential problems with these models and how to deal with them. Third, we'll explore the issue of variable selection. Of all the potential predictor variables available, how do you decide which ones to include in your final model? Fourth, we'll address the question of generalizability. How well will your model work when you apply it in the real world? Finally, we'll consider relative importance. Of all the predictors in your model, which one is the most important, the second most important, and the least important?

As you can see, we're covering a lot of ground. Effective regression analysis is an interactive, holistic process with many steps, and it involves more than a little skill. Rather than break it up into multiple chapters, I've opted to present this topic in a single chapter in order to capture this flavor. As a result, this will be the longest and most involved chapter in the book. Stick with it to the end, and you'll have all the tools you need to tackle a wide variety of research questions. Promise!

8.1 The many faces of regression

The term *regression* can be confusing because there are so many specialized varieties (see table 8.1). In addition, R has powerful and comprehensive features for fitting regression models, and the abundance of options can be confusing as well. For example, in 2005, Vito Ricci created a list of more than 205 functions in R that are used to generate regression analyses (<http://mng.bz/NJhu>).

Table 8.1 Varieties of regression analysis

Type of regression	Typical use
Simple linear	Predicting a quantitative response variable from a quantitative explanatory variable.

Polynomial	Predicting a quantitative response variable from a quantitative explanatory variable, where the relationship is modeled as an n th order polynomial.
Multiple linear	Predicting a quantitative response variable from two or more explanatory variables.
Multilevel	Predicting a response variable from data that have a hierarchical structure (for example, students within classrooms within schools). Also called <i>hierarchical</i> , <i>nested</i> , or <i>mixed</i> models.
Multivariate	Predicting more than one response variable from one or more explanatory variables.
Logistic	Predicting a categorical response variable from one or more explanatory variables.
Poisson	Predicting a response variable representing counts from one or more explanatory variables.
Cox proportional hazards	Predicting time to an event (death, failure, relapse) from one or more explanatory variables.
Time-series	Modeling time-series data with correlated errors.
Nonlinear	Predicting a quantitative response variable from one or more explanatory variables, where the form of the model is nonlinear.
Nonparametric	Predicting a quantitative response variable from one or more explanatory variables, where the form of the model is derived from the data and not specified a priori.
Robust	Predicting a quantitative response variable from one or more explanatory variables using an approach that's resistant to the effect of influential observations.

In this chapter, we'll focus on regression methods that fall under the rubric of *ordinary least squares (OLS) regression*, including simple linear regression, polynomial regression, and multiple linear regression. OLS regression is the most common variety of statistical analysis today. Other types of regression models (including logistic regression and Poisson regression) will be covered in chapter 13.

8.1.1 Scenarios for using OLS regression

In OLS regression, a quantitative dependent variable is predicted from a weighted sum of predictor variables, where the weights are parameters estimated from the data. Let's take a look at a concrete example (no pun intended), loosely adapted from Fwa (2006).

An engineer wants to identify the most important factors related to bridge deterioration (such as age, traffic volume, bridge design, construction materials and methods, construction quality, and weather conditions) and determine the mathematical form of these relationships. She collects data on each of these variables from a representative sample of bridges and models the data using OLS regression.

The approach is highly interactive. She fits a series of models, checks their compliance with underlying statistical assumptions, explores any unexpected or aberrant findings, and finally chooses the "best" model from among many possible models. If successful, the results will help her to:

- Focus on important variables, by determining which of the many collected variables are useful in predicting bridge deterioration, along with their relative importance.
- Look for bridges that are likely to be in trouble, by providing an equation that can be used to predict bridge deterioration for new cases (where the values of the predictor variables are known, but the degree of bridge deterioration isn't).
- Take advantage of serendipity, by identifying unusual bridges. If she finds that some bridges deteriorate much faster or slower than predicted by the model, a study of these outliers may yield important findings that could help her to understand the mechanisms involved in bridge deterioration.

Bridges may hold no interest for you. I'm a clinical psychologist and statistician, and I know next to nothing about civil engineering. But the general principles apply to an amazingly wide selection of problems in the physical, biological, and social sciences. Each of the following questions could also be addressed using an OLS approach:

- What's the relationship between surface stream salinity and paved road surface area (Montgomery, 2007)?
- What aspects of a user's experience contribute to the overuse of massively multiplayer online role-playing games (MMORPGs) (Hsu, Wen, & Wu, 2009)?
- Which qualities of an educational environment are most strongly related to higher student achievement scores?
- What's the form of the relationship between blood pressure, salt intake, and age? Is it the same for men and women?
- What's the impact of stadiums and professional sports on metropolitan area development (Baade & Dye, 1990)?
- What factors account for interstate differences in the price of beer (Culbertson & Bradford, 1991)? (That one got your attention!)

Our primary limitation is our ability to formulate an interesting question, devise a useful response variable to measure, and gather appropriate data.

8.1.2 What you need to know

For the remainder of this chapter, I'll describe how to use R functions to fit OLS regression models, evaluate the fit, test assumptions, and select among competing models. I assume you've had exposure to least squares regression as typically taught in a second-semester undergraduate statistics course. But I've made efforts to keep the mathematical notation to a minimum and focus on practical rather than theoretical issues. A number of excellent texts are available that cover the statistical material outlined in this chapter. My favorites are John Fox's *Applied Regression Analysis and Generalized Linear Models* (for theory) and *An R and S-Plus Companion to Applied Regression* (for application). They both served as major sources for this chapter. A good nontechnical overview is provided by Licht (1995).

8.2 OLS regression

For most of this chapter, we'll be predicting the response variable from a set of predictor variables (also called *regressing* the response variable on the predictor variables—hence the

name) using OLS. OLS regression fits models of the form where n is the number of observations and k is the number of predictor variables. (Although I've tried to keep equations out of these discussions, this is one of the few places where it simplifies things.) In this equation:

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_{1i} + \dots + \hat{\beta}_k X_{1k} \quad i = 1 \dots n$$

\hat{Y}_i is the predicted value of the dependent variable for observation i (specifically, it's the estimated mean of the Y distribution, conditional on the set of predictor values).

X_{ij} is the j th predictor value for the i th observation.

$\hat{\beta}_0$ is the intercept (the predicted value of Y when all the predictor variables equal zero).

$\hat{\beta}_j$ is the regression coefficient for the j th predictor (slope representing the change in Y for a unit change in X_j).

Our goal is to select model parameters (intercept and slopes) that minimize the difference between actual response values and those predicted by the model. Specifically, model parameters are selected to minimize the sum of squared residuals:

$$\sum (Y_i - \hat{Y}_i)^2 = \sum (Y_i - \hat{\beta}_0 + \hat{\beta}_1 X_{1i} + \dots + \hat{\beta}_k X_{1k})^2 = \sum \varepsilon_i^2$$

To properly interpret the coefficients of the OLS model, you must satisfy a number of statistical assumptions:

- *Normality*—For fixed values of the independent variables, the dependent variable is normally distributed.
- *Independence*—The Y_i values are independent of each other.
- *Linearity*—The dependent variable is linearly related to the independent variables.
- *Homoscedasticity*—The variance of the dependent variable doesn't vary with the levels of the independent variables. (I could call this constant *variance*, but saying *homoscedasticity* makes me feel smarter.)

If you violate these assumptions, your statistical significance tests and confidence intervals may not be accurate. Note that OLS regression also assumes that the independent variables are fixed and measured without error, but this assumption is typically relaxed in practice.

8.2.1 Fitting regression models with `lm()`

In R, the basic function for fitting a linear model is `lm()`. The format is

```
myfit <- lm(formula, data)
```

where *formula* describes the model to be fit and *data* is the data frame containing the data to be used in fitting the model. The resulting object (`myfit`, in this case) is a list that contains extensive information about the fitted model. The formula is typically written as

```
Y ~ X1 + X2 + ... + Xk
```

where the `~` separates the response variable on the left from the predictor variables on the right, and the predictor variables are separated by `+` signs. Other symbols can be used to modify the formula in various ways (see table 8.2).

Table 8.2 Symbols commonly used in R formulas

Symbol	Usage
~	Separates response variables on the left from the explanatory variables on the right. For example, a prediction of y from x , z , and w would be coded $y \sim x + z + w$.
+	Separates predictor variables.
:	Denotes an interaction between predictor variables. A prediction of y from x , z , and the interaction between x and z would be coded $y \sim x + z + x:z$.

*	A shortcut for denoting all possible interactions. The code $y \sim x * z * w$ expands to $y \sim x + z + w + x:z + x:w + z:w + x:z:w$.
^	Denotes interactions up to a specified degree. The code $y \sim (x + z + w)^2$ expands to $y \sim x + z + w + x:z + x:w + z:w$.
.	A placeholder for all other variables in the data frame except the dependent variable. For example, if a data frame contained the variables $x, y, z,$ and $w,$ then the code $y \sim .$ would expand to $y \sim x + z + w$.
-	A minus sign removes a variable from the equation. For example, $y \sim (x + z + w)^2 - x:w$ expands to $y \sim x + z + w + x:z + z:w$.
-1	Suppresses the intercept. For example, the formula $y \sim x - 1$ fits a regression of y on $x,$ and forces the line through the origin at $x=0$.
I()	Elements within the parentheses are interpreted arithmetically. For example, $y \sim x + (z + w)^2$ would expand to $y \sim x + z + w + z:w$. In contrast, the code $y \sim x + I((z + w)^2)$ would expand to $y \sim x + h,$ where h is a new variable created by squaring the sum of z and w .
<i>function</i>	Mathematical functions can be used in formulas. For example, $\log(y) \sim x + z + w$ would predict $\log(y)$ from $x, z,$ and w .

In addition to `lm()`, table 8.3 lists several functions that are useful when generating a simple or multiple regression analysis. Each of these functions is applied to the object returned by `lm()` in order to generate additional information based on that fitted model.

Table 8.3 Other functions that are useful when fitting linear models

Function	Action
<code>summary()</code>	Displays detailed results for the fitted model
<code>coefficients()</code>	Lists the model parameters (intercept and slopes) for the fitted model
<code>confint()</code>	Provides confidence intervals for the model parameters (95% by default)
<code>fitted()</code>	Lists the predicted values in a fitted model

<code>residuals()</code>	Lists the residual values in a fitted model
<code>anova()</code>	Generates an ANOVA table for a fitted model, or an ANOVA table comparing two or more fitted models
<code>vcov()</code>	Lists the covariance matrix for model parameters
<code>AIC()</code>	Prints Akaike's Information Criterion
<code>plot()</code>	Generates diagnostic plots for evaluating the fit of a model
<code>predict()</code>	Uses a fitted model to predict response values for a new dataset

When the regression model contains one dependent variable and one independent variable, the approach is called *simple linear regression*. When there's one predictor variable but powers of the variable are included (for example, X , X^2 , X^3), it's called *polynomial regression*. When there's more than one predictor variable, it's called *multiple linear regression*. We'll start with an example of simple linear regression, then progress to examples of polynomial and multiple linear regression, and end with an example of multiple regression that includes an interaction among the predictors.

8.2.2 Simple linear regression

Let's look at the functions in table 8.3 through a simple regression example. The dataset `women` in the base installation provides the height and weight for a set of 15 women

ages 30 to 39. Suppose you want to predict weight from height. Having an equation for predicting weight from height can help you to identify overweight or underweight individuals. The analysis is provided in the following listing, and the resulting graph is shown in figure 8.1

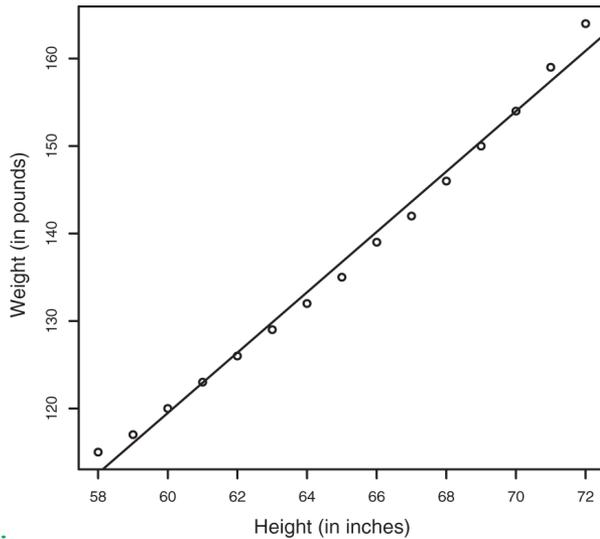


Figure 8.1 Scatter plot with regression line for weight predicted from height

Listing 8.1 Simple linear regression

```
> fit <- lm(weight ~ height, data=women)
> summary(fit)

Call:
lm(formula=weight ~ height, data=women)

Residuals:
    Min     1Q   Median     3Q     Max
-1.733 -1.133 -0.383  0.742  3.117

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -87.5167    5.9369  -14.7 1.7e-09 ***
height       3.4500    0.0911   37.9 1.1e-14 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.53 on 13 degrees of freedom
Multiple R-squared:  0.991,    Adjusted R-squared:  0.99
F-statistic: 1.43e+03 on 1 and 13 DF, p-value: 1.09e-14

> women$weight

[1] 115 117 120 123 126 129 132 135 139 142 146 150 154 159 164

> fitted(fit)

    1    2    3    4    5    6    7    8    9
112.58 116.03 119.48 122.93 126.38 129.83 133.28 136.73 140.18
```

```

10 11 12 13 14 15
143.63 147.08 150.53 153.98 157.43 160.88

> residuals(fit)

 1  2  3  4  5  6  7  8  9 10 11
2.42 0.97 0.52 0.07 -0.38 -0.83 -1.28 -1.73 -1.18 -1.63 -1.08
12 13 14 15
-0.53 0.02 1.57 3.12

> plot(women$height,women$weight,
      xlab="Height (in inches)",
      ylab="Weight (in pounds)")
> abline(fit)

```

From the output, you see that the prediction equation is

$$\text{weight} = 87.52 + 3.45 \text{ height}$$

Because a height of 0 is impossible, you wouldn't try to give a physical interpretation to the intercept. It merely becomes an adjustment constant. From the $\Pr(>|t|)$ column, you see that the regression coefficient (3.45) is significantly different from zero ($p < 0.001$) and indicates that there's an expected increase of 3.45 pounds of weight for every 1 inch increase in height. The multiple R-squared (0.991) indicates that the model accounts for 99.1% of the variance in weights. The multiple R-squared is also the squared correlation between the actual and predicted value (that is, $R^2 = r_{\hat{y}y}^2$). The residual standard error (1.53 pounds) can be thought of as the average error in predicting weight from height using this model. The F statistic tests whether the predictor variables, taken together, predict the response variable above chance levels. Because there's only one predictor variable in simple regression, in this example the F test is equivalent to the t-test for the regression coefficient for height.

For demonstration purposes, we've printed out the actual, predicted, and residual values. Evidently, the largest residuals occur for low and high heights, which can also be seen in the plot (figure 8.1).

The plot suggests that you might be able to improve on the prediction by using a line with one bend. For example, a model of the form $\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_i + \hat{\beta}_2 X_i^2$ may provide a better fit to the data. Polynomial regression allows you to predict a response variable from an explanatory variable, where the form of the relationship is an n th-degree polynomial.

8.2.3 Polynomial regression

The plot in figure 8.1 suggests that you might be able to improve your prediction using a regression with a quadratic term (that is, X^2). You can fit a quadratic equation using the statement

```
fit2 <- lm(weight ~ height + I(height^2), data=women)
```

The new term $I(\text{height}^2)$ requires explanation. `height^2` adds a height-squared term to the prediction equation. The `I()` function treats the contents within the parentheses as an R expression. You need this because the `^` operator has a special meaning in formulas that you don't want to invoke here (see table 8.2).

The following listing shows the results of fitting the quadratic equation.

Listing 8.2 Polynomial regression

```
> fit2 <- lm(weight ~ height + I(height^2), data=women)
> summary(fit2)

Call:
lm(formula=weight ~ height + I(height^2), data=women)

Residuals:
    Min     1Q   Median     3Q     Max
-0.5094 -0.2961 -0.0094  0.2862  0.5971

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 261.87818   25.19677   10.39 2.4e-07 ***
height      -7.34832    0.77769   -9.45 6.6e-07 ***
I(height^2)  0.08306    0.00598   13.89 9.3e-09 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.384 on 12 degrees of freedom
Multiple R-squared:  0.999,    Adjusted R-squared:  0.999
F-statistic: 1.14e+04 on 2 and 12 DF, p-value: <2e-16

> plot(women$height,women$weight,
       xlab="Height (in inches)",
       ylab="Weight (in lbs)")
> lines(women$height,fitted(fit2))
```

From this new analysis, the prediction equation is

$$\text{weight} = 261.88 - 7.35 \text{ height} + 0.083 \text{ height}^2$$

and both regression coefficients are significant at the $p < 0.0001$ level. The amount of variance accounted for has increased to 99.9%. The significance of the squared term ($t = 13.89$, $p < .001$) suggests that inclusion of the quadratic term improves the model fit. If you look at the plot of `fit2` (figure 8.2) you can see that the curve does indeed provide a better fit.

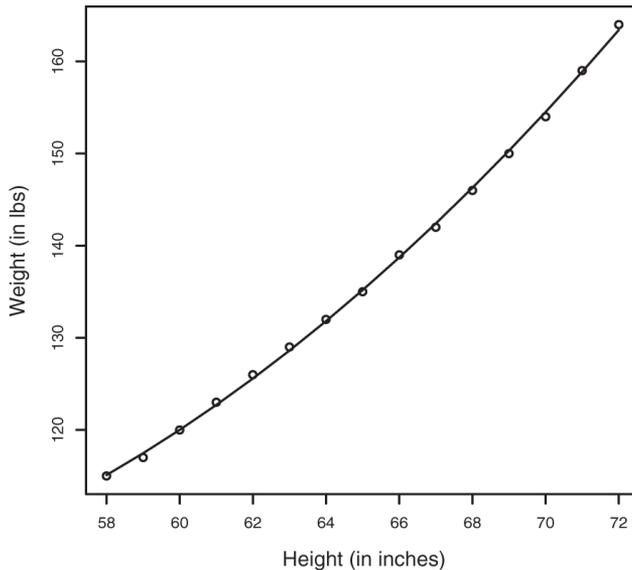


Figure 8.2 Quadratic regression for weight predicted by height

Linear vs. nonlinear models

Note that this polynomial equation still fits under the rubric of linear regression. It's linear because the equation involves a weighted sum of predictor variables (height and height-squared in this case). Even a model such as

$$\hat{Y}_i = \hat{\beta}_0 \times \log(X_1) + \hat{\beta}_2 \times \sin X_2$$

would be considered a linear model (linear in terms of the parameters) and fit with the formula

$$Y \sim \log(X1) + \sin(X2)$$

In contrast, here's an example of a truly nonlinear model:

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 e^{X_i/p_2}$$

Nonlinear models of this form can be fit with the `nls()` function.

In general, an n th-degree polynomial produces a curve with $n-1$ bends. To fit a cubic polynomial, you'd use

```
fit3 <- lm(weight ~ height + I(height^2) + I(height^3), data=women)
```

Although higher polynomials are possible, I've rarely found that terms higher than cubic are necessary.

8.2.4 Multiple linear regression

When there's more than one predictor variable, simple linear regression becomes multiple linear regression, and the analysis grows more involved. Technically, polynomial regression is

a special case of multiple regression. Quadratic regression has two predictors (X and X^2), and cubic regression has three predictors (X , X^2 , and X^3). Let's look at a more general example.

We'll use the `state.x77` dataset in the base package for this example. Suppose you want to explore the relationship between a state's murder rate and other characteristics of the state, including population, illiteracy rate, average income, and frost levels (mean number of days below freezing).

Because the `lm()` function requires a data frame (and the `state.x77` dataset is contained in a matrix), you can simplify your life with the following code:

```
states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])
```

This code creates a data frame called `states`, containing the variables you're interested in. You'll use this new data frame for the remainder of the chapter.

A good first step in multiple regression is to examine the relationships among the variables two at a time. The bivariate correlations are provided by the `cor()` function, and scatter plots are generated from the `scatterplotMatrix()` function in the `car` package (see the following listing and figure 8.3).

Listing 8.3 Examining bivariate relationships

```
> states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])

> cor(states)
      Murder Population Illiteracy Income Frost
Murder  1.00   0.34   0.70 -0.23 -0.54
Population 0.34   1.00   0.11  0.21 -0.33
Illiteracy 0.70   0.11   1.00 -0.44 -0.67
Income   -0.23   0.21  -0.44  1.00  0.23
Frost   -0.54  -0.33  -0.67  0.23  1.00

> library(car)
> scatterplotMatrix(states, smooth=FALSE, main="Scatter Plot Matrix")
```

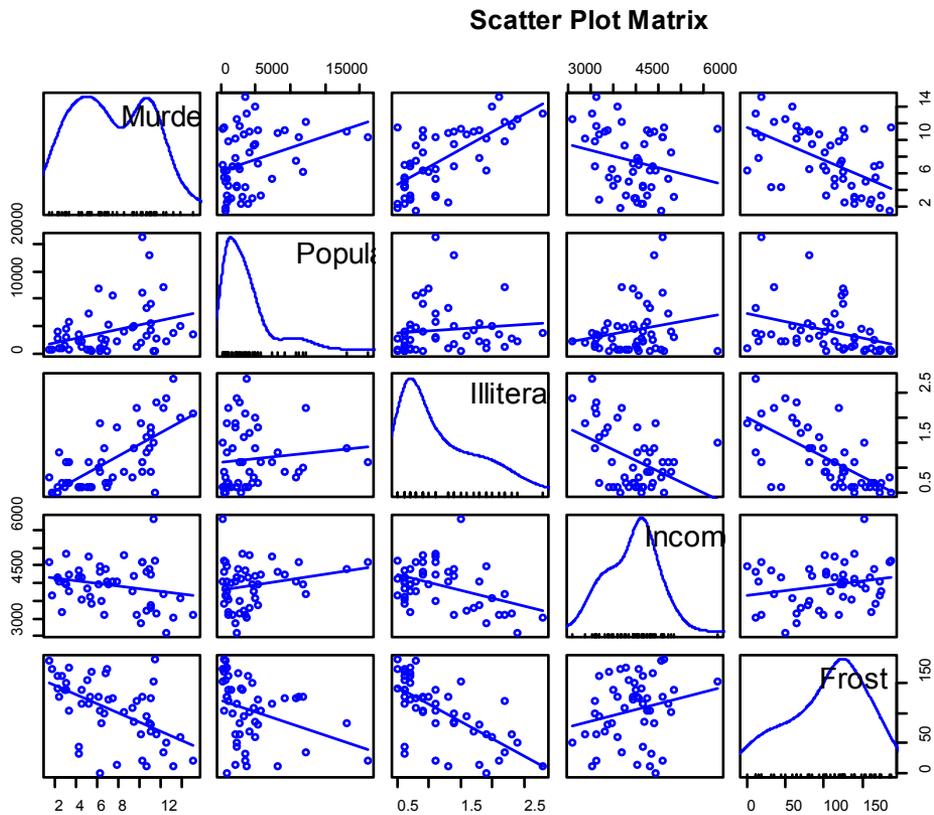


Figure 8.3 Scatter plot matrix of dependent and independent variables for the states data, including linear and smoothed fits, and marginal distributions (kernel-density plots and rug plots)

By default, the `scatterplotMatrix()` function provides scatter plots of the variables with each other in the off-diagonals and superimposes smoothed (loess) and linear fit lines on these plots. The principal diagonal contains density and rug plots for each variable. The smoothed lines are suppressed with the argument `smooth=FALSE`.

You can see that murder rate may be bimodal and that each of the predictor variables is skewed to some extent. Murder rates rise with population and illiteracy, and they fall with higher income levels and frost. At the same time, colder states have lower illiteracy rates,, lower population, and higher incomes.

Now let's fit the multiple regression model with the `lm()` function (see the following listing).

Listing 8.4 Multiple linear regression

```

> states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])

> fit <- lm(Murder ~ Population + Illiteracy + Income + Frost,
  data=states)
> summary(fit)

Call:
lm(formula=Murder ~ Population + Illiteracy + Income + Frost,
  data=states)

Residuals:
  Min   1Q Median   3Q   Max
-4.7960 -1.6495 -0.0811  1.4815  7.6210

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 1.23e+00  3.87e+00  0.32  0.751
Population  2.24e-04  9.05e-05  2.47  0.017 *
Illiteracy  4.14e+00  8.74e-01  4.74  2.2e-05 ***
Income     6.44e-05  6.84e-04  0.09  0.925
Frost      5.81e-04  1.01e-02  0.06  0.954
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 'v' 1

Residual standard error: 2.5 on 45 degrees of freedom
Multiple R-squared: 0.567, Adjusted R-squared: 0.528
F-statistic: 14.7 on 4 and 45 DF, p-value: 9.13e-08

```

When there's more than one predictor variable, the regression coefficients indicate the increase in the dependent variable for a unit change in a predictor variable, holding all other predictor variables constant. For example, the regression coefficient for Illiteracy is 4.14, suggesting that an increase of 1% in illiteracy is associated with a 4.14% increase in the murder rate, controlling for population, income, and temperature. The coefficient is significantly different from zero at the $p < .0001$ level. On the other hand, the coefficient for Frost isn't significantly different from zero ($p = 0.954$) suggesting that Frost and Murder aren't linearly related when controlling for the other predictor variables. Taken together, the predictor variables account for 57% of the variance in murder rates across states.

Up to this point, we've assumed that the predictor variables don't interact. In the next section, we'll consider a case in which they do.

8.2.5 Multiple linear regression with interactions

Some of the most interesting research findings are those involving interactions among predictor variables. Consider the automobile data in the `mtcars` data frame. Let's say that you're interested in the impact of automobile weight and horsepower on mileage. You could fit a regression model that includes both predictors, along with their interaction, as shown in the next listing.

Listing 8.5 Multiple linear regression with a significant interaction term

```

> fit <- lm(mpg ~ hp + wt + hp:wt, data=mtcars)
> summary(fit)

Call:
lm(formula=mpg ~ hp + wt + hp:wt, data=mtcars)

Residuals:
    Min     1Q   Median     3Q     Max
-3.063 -1.649 -0.736  1.421  4.551

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 49.80842   3.60516   13.82 5.0e-14 ***
hp          -0.12010   0.02470   -4.86 4.0e-05 ***
wt          -8.21662   1.26971   -6.47 5.2e-07 ***
hp:wt        0.02785   0.00742    3.75 0.00081 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.1 on 28 degrees of freedom
Multiple R-squared: 0.885,    Adjusted R-squared: 0.872
F-statistic: 71.7 on 3 and 28 DF, p-value: 2.98e-13

```

You can see from the $\text{Pr}(>|t|)$ column that the interaction between horsepower and car weight is significant. What does this mean? A significant interaction between two predictor variables tells you that the relationship between one predictor and the response variable depends on the level of the other predictor. Here it means the relationship between miles per gallon and horsepower varies by car weight.

The model for predicting mpg is $\text{mpg} = 49.81 - 0.12 \times \text{hp} - 8.22 \times \text{wt} + 0.03 \times \text{hp} \times \text{wt}$. To interpret the interaction, you can plug in various values of wt and simplify the equation. For example, you can try the mean of wt (3.2) and one standard deviation below and above the mean (2.2 and 4.2, respectively). For $\text{wt}=2.2$, the equation simplifies to $\text{mpg} = 49.81 - 0.12 \times \text{hp} - 8.22 \times (2.2) + 0.03 \times \text{hp} \times (2.2) = 31.41 - 0.06 \times \text{hp}$. For $\text{wt}=3.2$, this becomes $\text{mpg} = 23.37 - 0.03 \times \text{hp}$. Finally, for $\text{wt}=4.2$ the equation becomes $\text{mpg} = 15.33 - 0.003 \times \text{hp}$. You see that as weight increases (2.2, 3.2, 4.2), the expected change in mpg from a unit increase in hp decreases (0.06, 0.03, 0.003).

You can visualize interactions using the `effect()` function in the `effects` package. The format is

```
plot(effect(term, mod,, xlevels), multiline=TRUE)
```

where `term` is the quoted model term to plot, `mod` is the fitted model returned by `lm()`, and `xlevels` is a list specifying the variables to be set to constant values and the values to employ. The `multiline=TRUE` option superimposes the lines being plotted and the `lines` option specifies the line type for each line (where 1= solid, 2=dashed, and 3=dotted, etc.). For the previous model, this becomes

```
library(effects)
plot(effect("hp:wt", fit, list(wt=c(2.2,3.2,4.2))),
     lines=c(1,2,3), multiline=TRUE)
```

The resulting graph is displayed in figure 8.5.

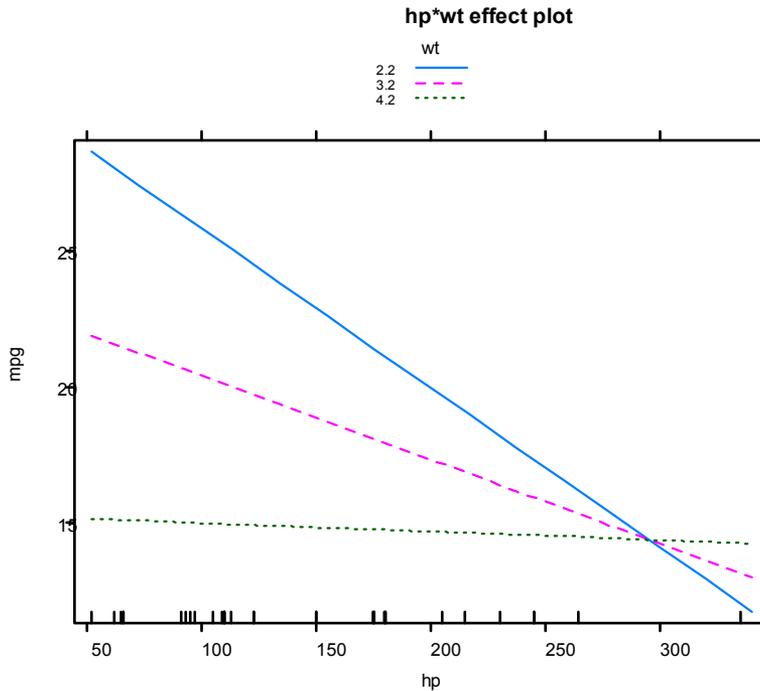


Figure 8.4 Interaction plot for $hp*wt$. This plot displays the relationship between mpg and hp at three values of wt.

You can see from this graph that as the weight of the car increases, the relationship between horsepower and miles per gallon weakens. For $wt=4.2$, the line is almost horizontal, indicating that as hp increases, mpg doesn't change.

Unfortunately, fitting the model is only the first step in the analysis. Once you fit a regression model, you need to evaluate whether you've met the statistical assumptions underlying your approach before you can have confidence in the inferences you draw. This is the topic of the next section.

8.3 Regression diagnostics

In the previous section, you used the `lm()` function to fit an OLS regression model and the `summary()` function to obtain the model parameters and summary statistics. Unfortunately, nothing in this printout tells you whether the model you've fit is appropriate. Your confidence in inferences about regression parameters depends on the degree to which you've met the statistical assumptions of the OLS model. Although the `summary()` function in listing 8.4 describes the model, it provides no information concerning the degree to which you've satisfied the statistical assumptions *underlying* the model.

Why is this important? Irregularities in the data or misspecifications of the relationships between the predictors and the response variable can lead you to settle on a model that's wildly inaccurate. On the one hand, you may conclude that a predictor and a response variable are unrelated when, in fact, they are. On the other hand, you may conclude that a predictor and a response variable are related when, in fact, they aren't! You may also end up with a model that makes poor predictions when applied in real-world settings, with significant and unnecessary error.

Let's look at the output from the `confint()` function applied to the `states` multiple regression problem in section 8.2.4:

```
> states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])
> fit <- lm(Murder ~ Population + Illiteracy + Income + Frost, data=states)
> confint(fit)
      2.5 % 97.5 %
(Intercept) -6.55e+00 9.021318
Population   4.14e-05 0.000406
Illiteracy   2.38e+00 5.903874
Income      -1.31e-03 0.001441
Frost       -1.97e-02 0.020830
```

The results suggest that you can be 95% confident that the interval [2.38, 5.90] contains the true change in murder rate for a 1% change in illiteracy rate. Additionally, because the confidence interval for Frost contains 0, you can conclude that a change in temperature is unrelated to murder rate, holding the other variables constant. But your faith in these results is only as strong as the evidence you have that your data satisfies the statistical assumptions underlying the model.

A set of techniques called *regression diagnostics* provides the necessary tools for evaluating the appropriateness of the regression model and can help you to uncover and correct problems. We'll start with a standard approach that uses functions that come with R's base installation. Then we'll look at newer, improved methods available through the `car` package.

8.3.1 A typical approach

R's base installation provides numerous methods for evaluating the statistical assumptions in a regression analysis. The most common approach is to apply the `plot()` function to the object returned by the `lm()`. Doing so produces four graphs that are useful for evaluating the model fit. Applying this approach to the simple linear regression example

```
fit <- lm(weight ~ height, data=women)
par(mfrow=c(2,2))
plot(fit)
par(mfrow=c(1,1))
```

produces the graphs shown in figure 8.5. The `par(mfrow=c(2,2))` statement is used to combine the four plots produced by the `plot()` function into one large 2×2 graph. The second `par()` function returns you to single graphs.

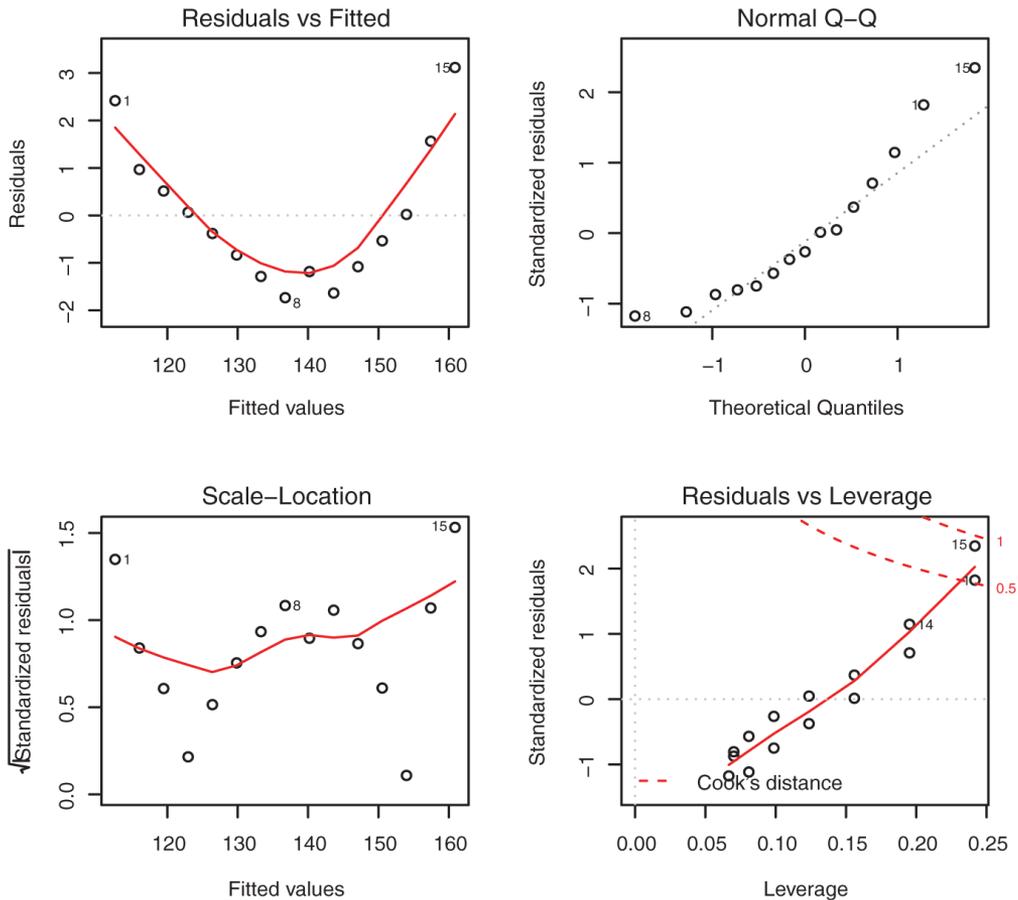


Figure 8.5 Diagnostic plots for the regression of weight on height

To understand these graphs, consider the assumptions of OLS regression:

- *Normality*—If the dependent variable is normally distributed for a fixed set of predictor values, then the residual values should be normally distributed with a mean of 0. The Normal Q-Q plot (upper right) is a probability plot of the standardized residuals against the values that would be expected under normality. If you've met the normality assumption, the points on this graph should fall on the straight 45-degree line. Because they don't, you've clearly violated the normality assumption.
- *Independence*—You can't tell if the dependent variable values are independent from these plots. You have to use your understanding of how the data was collected. There's no a priori reason to believe that one woman's weight influences another woman's

weight. If you found out that the data were sampled from families, you might have to adjust your assumption of independence.

- *Linearity*—If the dependent variable is linearly related to the independent variables, there should be no systematic relationship between the residuals and the predicted (that is, fitted) values. In other words, the model should capture all the systematic variance present in the data, leaving nothing but random noise. In the Residuals vs. Fitted graph (upper left), you see clear evidence of a curved relationship, which suggests that you may want to add a quadratic term to the regression.
- *Homoscedasticity*—If you’ve met the constant variance assumption, the points in the Scale-Location graph (bottom left) should be a random band around a horizontal line. You seem to meet this assumption.

Finally, the Residuals vs. Leverage graph (bottom right) provides information about individual observations that you may wish to attend to. The graph identifies outliers, high-leverage points, and influential observations. Specifically:

- An *outlier* is an observation that isn’t predicted well by the fitted regression model (that is, has a large positive or negative residual).
- An observation with a high *leverage* value has an unusual combination of predictor values. That is, it’s an outlier in the predictor space. The dependent variable value isn’t used to calculate an observation’s leverage.
- An *influential observation* is an observation that has a disproportionate impact on the determination of the model parameters. Influential observations are identified using a statistic called *Cook’s distance*, or *Cook’s D*.

To be honest, I find the Residuals vs. Leverage plot difficult to read and not useful. You’ll see better representations of this information in later sections.

Although these standard diagnostic plots are helpful, better tools are now available in R and I recommend their use over the `plot(fit)` approach.

8.3.2 An enhanced approach

The `car` package provides a number of functions that significantly enhance your ability to fit and evaluate regression models (see table 8.4).

Table 8.4 Useful functions for regression diagnostics (car package)

Function	Purpose
<code>qqPlot()</code>	Quantile comparisons plot

<code>durbinWatsonTest()</code>	Durbin–Watson test for autocorrelated errors
<code>crPlots()</code>	Component plus residual plots
<code>ncvTest()</code>	Score test for nonconstant error variance
<code>spreadLevelPlot()</code>	Spread-level plots
<code>outlierTest()</code>	Bonferroni outlier test
<code>avPlots()</code>	Added variable plots
<code>influencePlot()</code>	Regression influence plots
<code>scatterplot()</code>	Enhanced scatter plots
<code>scatterplotMatrix()</code>	Enhanced scatter plot matrixes
<code>vif()</code>	Variance inflation factors

Let's look at each in turn, by applying them to our multiple regression example.

NORMALITY

The `qqPlot()` function provides a more accurate method of assessing the normality assumption than that provided by the `plot()` function in the base package. It plots the *studentized residuals* (also called *studentized deleted residuals* or *jackknifed residuals*) against a *t* distribution with $n - p - 1$ degrees of freedom, where n is the sample size and p is the number of regression parameters (including the intercept). The code follows:

```
library(car)
states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])
fit <- lm(Murder ~ Population + Illiteracy + Income + Frost, data=states)
qqPlot(fit, labels=row.names(states), id=list(method="identify"),
  simulate=TRUE, main="Q-Q Plot")
```

The `qqPlot()` function generates the probability plot displayed in figure 8.6. The option `id=list(method="identify")` makes the plot interactive—after the graph is drawn, mouse clicks on points in the graph will label them with values specified in the `labels` option of the function. Pressing the Esc key, or pressing the Finish button in the upper right corner of the graph, turns off this interactive mode. Here, I identified Nevada. When `simulate=TRUE`, a 95% confidence envelope is produced using a parametric bootstrap. (Bootstrap methods are considered in chapter 12.)

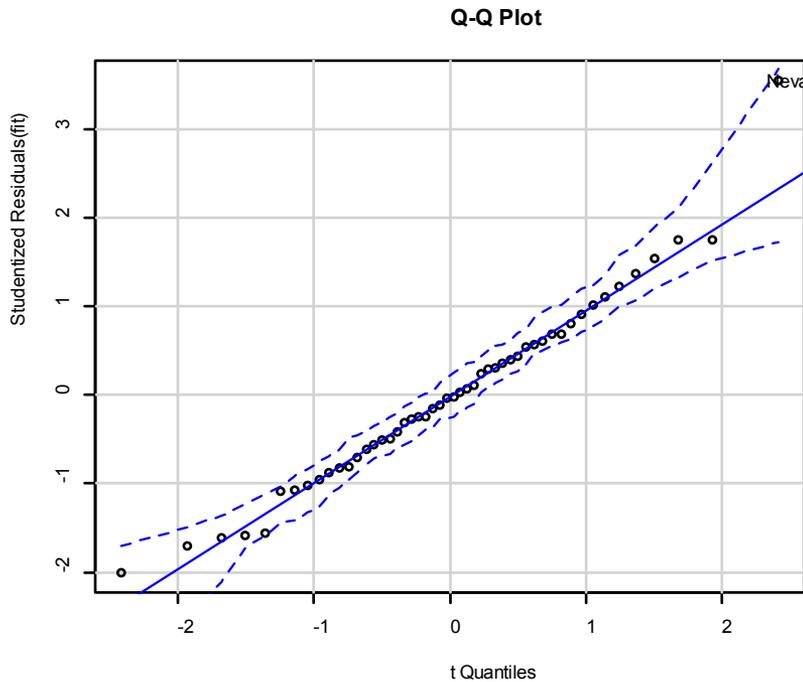


Figure 8.6 Q-Q plot for studentized residuals

With the exception of Nevada, all the points fall close to the line and are within the confidence envelope, suggesting that you've met the normality assumption fairly well. But you should definitely look at Nevada. It has a large positive residual (actual – predicted), indicating that the model underestimates the murder rate in this state. Specifically:

```
> states["Nevada",]
      Murder Population Illiteracy Income Frost
Nevada  11.5      590      0.5  5149  188

> fitted(fit)["Nevada"]
Nevada
3.878958

> residuals(fit)["Nevada"]
Nevada
7.621042

> rstudent(fit)["Nevada"]
```

```
Nevada
3.542929
```

Here you see that the murder rate is 11.5%, but the model predicts a 3.9% murder rate.

The question that you need to ask is, “Why does Nevada have a higher murder rate than predicted from population, income, illiteracy, and temperature?” Anyone (who hasn’t see *Casino*) want to guess?

INDEPENDENCE OF ERRORS

As indicated earlier, the best way to assess whether the dependent variable values (and thus the residuals) are independent is from your knowledge of how the data were collected. For example, time series data often display autocorrelation—observations collected closer in time are more correlated with each other than with observations distant in time. The `car` package provides a function for the Durbin–Watson test to detect such serially correlated errors. You can apply the Durbin–Watson test to the multiple-regression problem with the following code:

```
> durbinWatsonTest(fit)
lag Autocorrelation D-W Statistic p-value
1 -0.201 2.32 0.282
Alternative hypothesis: rho != 0
```

The nonsignificant p-value ($p=0.282$) suggests a lack of autocorrelation and, conversely, an independence of errors. The lag value (1 in this case) indicates that each observation is being compared with the one next to it in the dataset. Although appropriate for time-dependent data, the test is less applicable for data that isn’t clustered in this fashion. Note that the `durbinWatsonTest()` function uses bootstrapping (see chapter 12) to derive p-values. Unless you add the option `simulate=FALSE`, you’ll get a slightly different value each time you run the test.

LINEARITY

You can look for evidence of nonlinearity in the relationship between the dependent variable and the independent variables by using *component plus residual plots* (also known as *partial residual plots*). The plot is produced by the `crPlots()` function in the `car` package. You’re looking for any systematic departure from the linear model that you’ve specified.

$$\varepsilon_i + \hat{\beta}_k \times X_{ik} \text{ vs. } X_{ik}$$

To create a component plus residual plot for variable k , you plot the points

where the residuals are based on the full model (containing all the predictors), and $i = 1 \dots n$. The straight line in each graph is given by $\hat{\beta}_k \times X_{ik}$ vs. X_{ik} . A loess line (a smoothed

nonparametric fit line) is also provided for each plot. Loess lines are described in chapter 11. The code to produce these plots is as follows:

```
> library(car)
> crPlots(fit)
```

The resulting plots are provided in figure 8.7. Nonlinearity in any of these plots suggests that you may not have adequately modeled the functional form of that predictor in the regression. If so, you may need to add curvilinear components such as polynomial terms, transform one or more variables (for example, use `log(X)` instead of `X`), or abandon linear regression in favor of some other regression variant. Transformations are discussed later in this chapter.

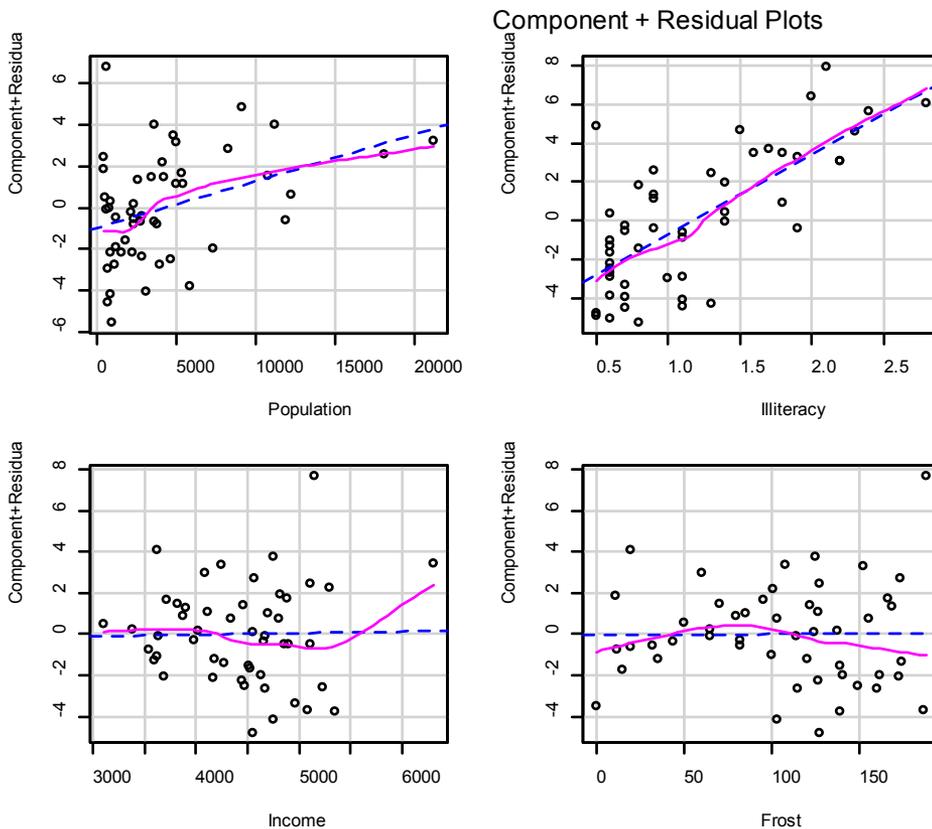


Figure 8.7 Component plus residual plots for the regression of murder rate on state characteristics

The component plus residual plots confirm that you've met the linearity assumption. The form of the linear model seems to be appropriate for this dataset.

HOMOSCEDASTICITY

The `car` package also provides two useful functions for identifying non-constant error variance. The `ncvTest()` function produces a score test of the hypothesis of constant error variance against the alternative that the error variance changes with the level of the fitted values. A significant result suggests heteroscedasticity (nonconstant error variance).

The `spreadLevelPlot()` function creates a scatter plot of the absolute standardized residuals versus the fitted values and superimposes a line of best fit. Both functions are demonstrated in the next listing.

Listing 8.6 Assessing homoscedasticity

```
> library(car)
> ncvTest(fit)

Non-constant Variance Score Test
Variance formula: ~ fitted.values
Chisquare=1.7 Df=1 p=0.19

> spreadLevelPlot(fit)

Suggested power transformation: 1.2
```

The score test is nonsignificant ($p = 0.19$), suggesting that you've met the constant variance assumption. You can also see this in the spread-level plot (figure 8.12). The points form a random horizontal band around a horizontal line of best fit. If you'd violated the assumption, you'd expect to see a nonhorizontal line. The suggested power transformation in listing 8.7 is the suggested power p (Y^p) that would stabilize the nonconstant error variance. For example, if the plot showed a nonhorizontal trend and the suggested power transformation was 0.5, then using \sqrt{Y} rather than Y in the regression equation might lead to a model that satisfied homoscedasticity. If the suggested power was 0, you'd use a log transformation. In the current example, there's no evidence of heteroscedasticity, and the suggested power is close to 1 (no transformation required).

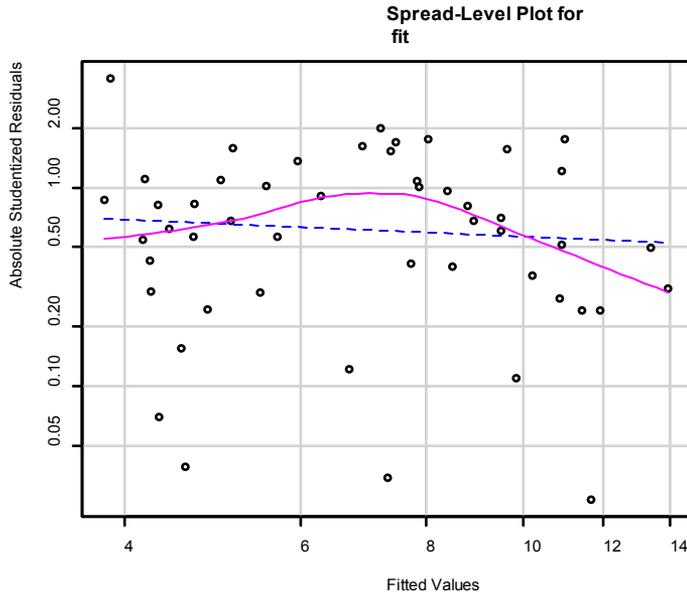


Figure 8.8 Spread-level plot for assessing constant error variance

8.3.3 Multicollinearity

Before leaving this section on regression diagnostics, let's focus on a problem that's not directly related to statistical assumptions but is important in allowing you to interpret multiple regression results. Imagine you're conducting a study of grip strength. Your independent variables include date of birth (DOB) and age. You regress grip strength on DOB and age and find a significant overall F test at $p < .001$. But when you look at the individual regression coefficients for DOB and age, you find that they're both nonsignificant (that is, there's no evidence that either is related to grip strength). What happened?

The problem is that DOB and age are perfectly correlated within rounding error. A regression coefficient measures the impact of one predictor variable on the response variable, holding all other predictor variables constant. This amounts to looking at the relationship of grip strength and age, holding age constant. The problem is called *multicollinearity*. It leads to large confidence intervals for model parameters and makes the interpretation of individual coefficients difficult.

Multicollinearity can be detected using a statistic called the *variance inflation factor* (VIF). For any predictor variable, the square root of the VIF indicates the degree to which the confidence interval for that variable's regression parameter is expanded relative to a model with uncorrelated predictors (hence the name). VIF values are provided by the `vif()` function in

the `car` package. As a general rule, a `vif > 10` indicates a multicollinearity problem. The code is provided in the following listing. The results indicate that multicollinearity isn't a problem with these predictor variables.

Listing 8.7 Evaluating multicollinearity

```
> library(car)
> vif(fit)

Population Illiteracy Income Frost
      1.2      2.2      1.3      2.1

> vif(fit) > 10 # problem?

Population Illiteracy Income Frost
      FALSE      FALSE      FALSE      FALSE
```

8.4 Unusual observations

A comprehensive regression analysis will also include a screening for unusual observations—namely outliers, high-leverage observations, and influential observations. These are data points that warrant further investigation, either because they're different than other observations in some way, or because they exert a disproportionate amount of influence on the results. Let's look at each in turn.

8.4.1 Outliers

Outliers are observations that aren't predicted well by the model. They have unusually large positive or negative residuals $(Y_i - \hat{Y}_i)$. Positive residuals indicate that the model is underestimating the response value, whereas negative residuals indicate an overestimation.

You've already seen one way to identify outliers. Points in the Q-Q plot of figure 8.6 that lie outside the confidence band are considered outliers. A rough rule of thumb is that standardized residuals that are larger than 2 or less than -2 are worth attention.

The `car` package also provides a statistical test for outliers. The `outlierTest()` function reports the Bonferroni adjusted p-value for the largest absolute studentized residual:

```
> library(car)
> outlierTest(fit)
      rstudent unadjusted p-value Bonferonni p
Nevada  3.5      0.00095      0.048
```

Here, you see that Nevada is identified as an outlier ($p = 0.048$). Note that this function tests the single largest (positive or negative) residual for significance as an outlier. If it isn't significant, there are no outliers in the dataset. If it's significant, you must delete it and rerun the test to see if others are present.

8.4.2 High-leverage points

Observations that have high leverage are outliers with regard to the other predictors. In other words, they have an unusual combination of predictor values. The response value isn't involved in determining leverage.

Observations with high leverage are identified through the *hat statistic*. For a given dataset, the average hat value is p/n , where p is the number of parameters estimated in the model (including the intercept) and n is the sample size. Roughly speaking, an observation with a hat value greater than 2 or 3 times the average hat value should be examined. The code that follows plots the hat values:

```
hat.plot <- function(fit) {  
  p <- length(coefficients(fit))  
  n <- length(fitted(fit))  
  plot(hatvalues(fit), main="Index Plot of Hat Values")  
  abline(h=c(2,3)*p/n, col="red", lty=2)  
  identify(1:n, hatvalues(fit), names(hatvalues(fit)))  
}  
hat.plot(fit)
```

The resulting graph is shown in figure 8.9.

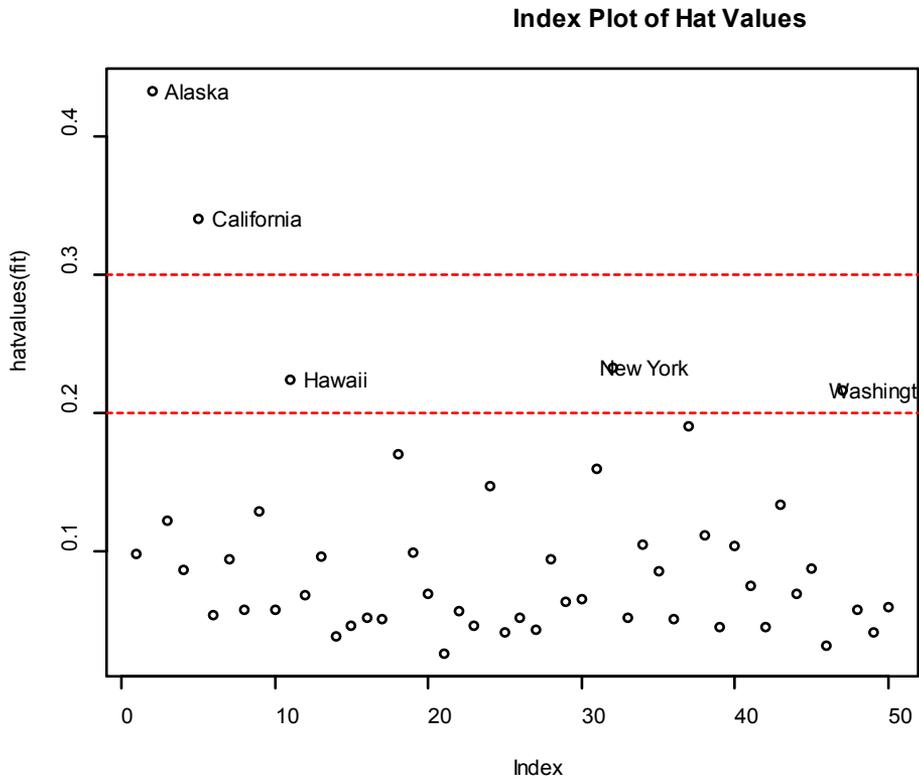


Figure 8.9 Index plot of hat values for assessing observations with high leverage

Horizontal lines are drawn at 2 and 3 times the average hat value. The locator function places the graph in interactive mode. Clicking points of interest labels them until the user presses Esc, or the Finish button in the upper right corner of the graph.

Here you see that Alaska and California are particularly unusual when it comes to their predictor values. Alaska has a much higher income than other states, while having a lower population and temperature. California has a much higher population than other states, while having a higher income and higher temperature. These states are atypical compared with the other 48 observations.

High-leverage observations may or may not be influential observations. That will depend on whether they're also outliers.

8.4.3 Influential observations

Influential observations have a disproportionate impact on the values of the model parameters. Imagine finding that your model changes dramatically with the removal of a single observation. It's this concern that leads you to examine your data for influential points.

There are two methods for identifying influential observations: Cook's distance (or D statistic) and *added variable* plots. Roughly speaking, Cook's D values greater than $-4/(n - k - 1)$, where n is the sample size and k is the number of predictor variables, indicate influential observations. You can create a Cook's D plot (figure 8.10) with the following code:

```
cutoff <- 4/(nrow(states)-length(fit$coefficients)-2)
plot(fit, which=4, cook.levels=cutoff)
abline(h=cutoff, lty=2, col="red")
```

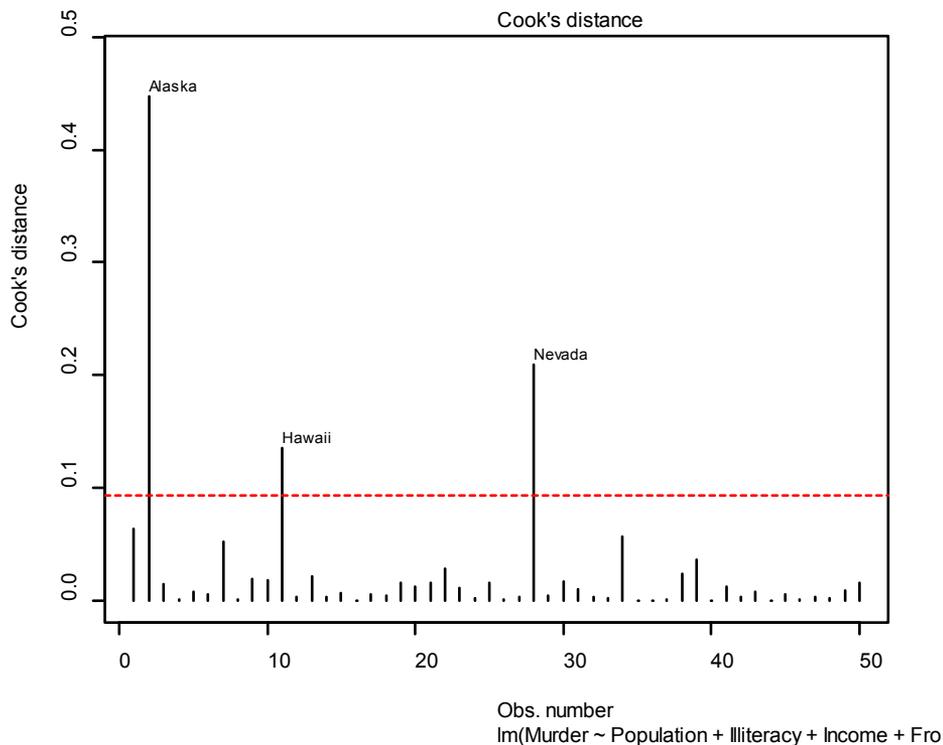


Figure 8.10 Cook's D plot for identifying influential observations

The graph identifies Alaska, Hawaii, and Nevada as influential observations. Deleting these states will have a notable impact on the values of the intercept and slopes in the regression model. Note that although it's useful to cast a wide net when searching for influential observations, I tend to find a cutoff of 1 more generally useful than $4/(n - k - 1)$. Given a criterion of $D=1$, none of the observations in the dataset would appear to be influential.

Cook's D plots can help identify influential observations, but they don't provide information about how these observations affect the model. Added-variable plots can help in this regard. For one response variable and k predictor variables, you'd create k added-variable plots as follows.

For each predictor X_k , plot the residuals from regressing the response variable on the other $k - 1$ predictors versus the residuals from regressing X_k on the other $k - 1$ predictors. Added-variable plots can be created using the `avPlots()` function in the `car` package:

```
library(car)
avPlots(fit, ask=FALSE, id.method="identify")
```

The resulting graphs are provided in figure 8.11. The graphs are produced one at a time, and users can click points to identify them. Press Esc or press the Finish button on the upper right corner of the graph to move to the next plot. Here, I've identified Alaska in the bottom-left plot.

Added-Variable Plots

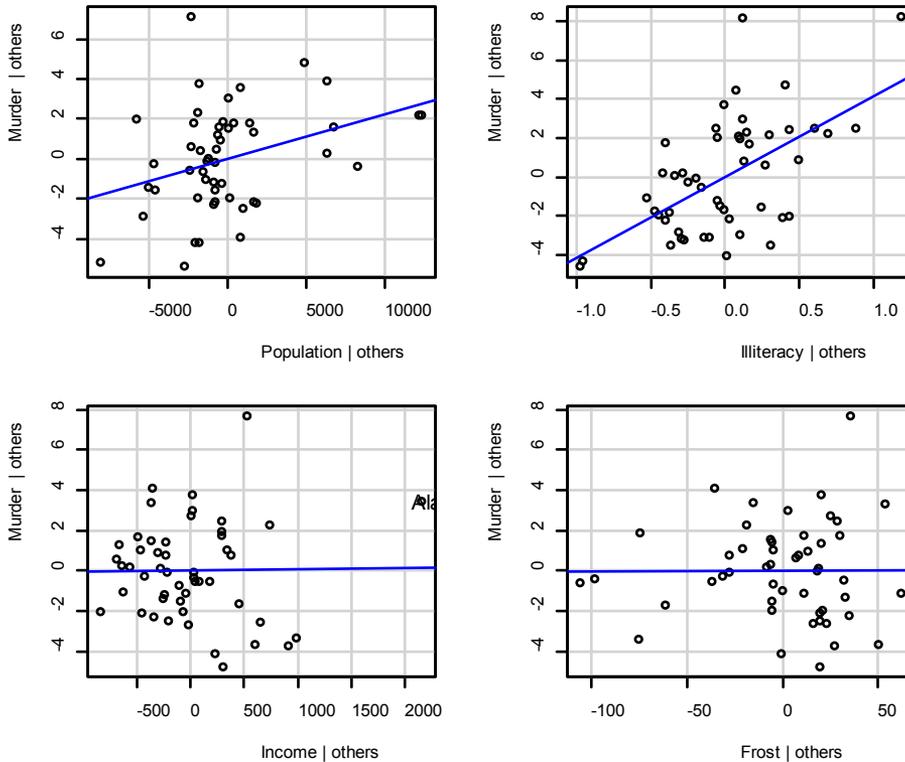


Figure 8.11 Added-variable plots for assessing the impact of influential observations

The straight line in each plot is the actual regression coefficient for that predictor variable. You can see the impact of influential observations by imagining how the line would change if the point representing that observation was deleted. For example, look at the graph of Murder | Others versus Income | Others in the lower-left corner. You can see that eliminating the point labeled Alaska would move the line in a negative direction. In fact, deleting Alaska changes the regression coefficient for Income from positive (.00006) to negative (-.00085).

You can combine the information from outlier, leverage, and influence plots into one highly informative plot using the `influencePlot()` function from the `car` package:

```
library(car)
influencePlot(fit, id="noteworthy", main="Influence Plot",
             sub="Circle size is proportional to Cook's distance")
```

The resulting plot (figure 8.12) identifies observations that are particularly noteworthy. In particular, it shows that Nevada and Rhode Island are outliers; California, and Hawaii have high leverage; and Nevada and Alaska are influential observations.

Replacing `id="noteworthy"` with `id=list(method="identify")` allows you to identify points interactively with mouse clicks (ending with ESC or pressing the Finish button).

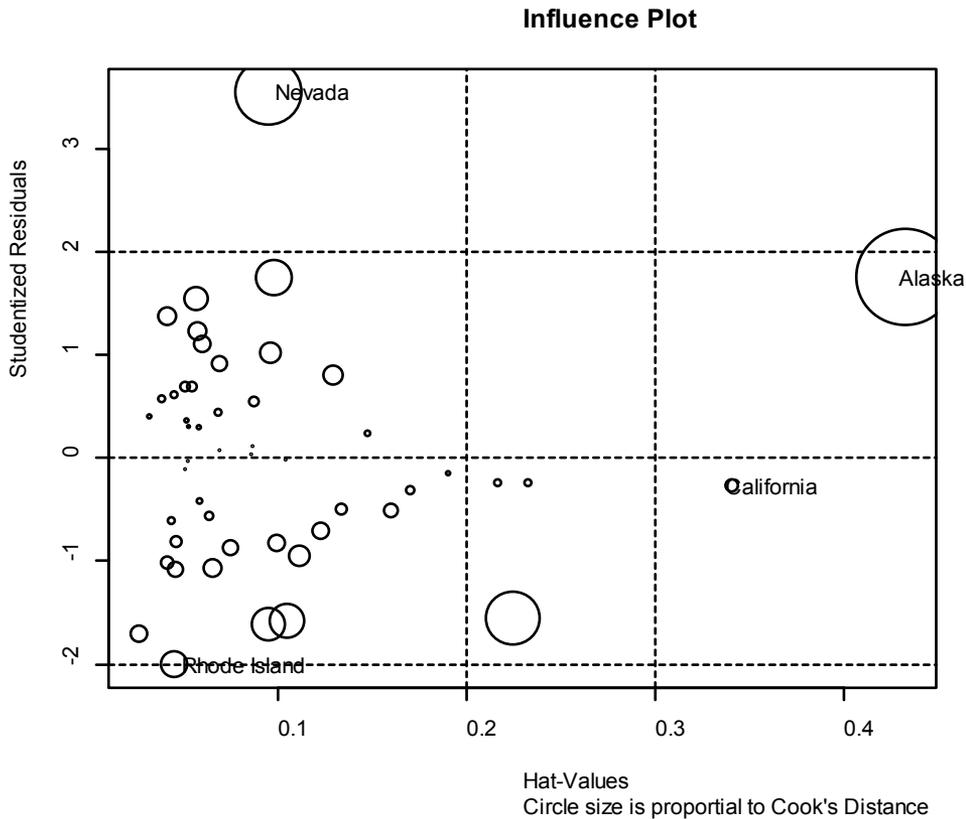


Figure 8.12 Influence plot. States above +2 or below -2 on the vertical axis are considered outliers. States above 0.2 or 0.3 on the horizontal axis have high leverage (unusual combinations of predictor values). Circle size is proportional to influence. Observations depicted by large circles may have disproportionate influence on the parameter estimates of the model.

8.5 Corrective measures

Having spent the last 16 pages learning about regression diagnostics, you may ask, “What do you do if you identify problems?” There are four approaches to dealing with violations of regression assumptions:

- Deleting observations
- Transforming variables
- Adding or deleting variables
- Using another regression approach

Let’s look at each in turn.

8.5.1 Deleting observations

Deleting outliers can often improve a dataset’s fit to the normality assumption. Influential observations are often deleted as well, because they have an inordinate impact on the results. The largest outlier or influential observation is deleted, and the model is refit. If there are still outliers or influential observations, the process is repeated until an acceptable fit is obtained.

Again, I urge caution when considering the deletion of observations. Sometimes you can determine that the observation is an outlier because of data errors in recording, or because a protocol wasn’t followed, or because a test subject misunderstood instructions. In these cases, deleting the offending observation seems perfectly reasonable.

In other cases, the unusual observation may be the most interesting thing about the data you’ve collected. Uncovering why an observation differs from the rest can contribute great insight to the topic at hand and to other topics you might not have thought of. Some of our greatest advances have come from the serendipity of noticing that something doesn’t fit our preconceptions (pardon the hyperbole).

8.5.2 Transforming variables

When models don’t meet the normality, linearity, or homoscedasticity assumptions, transforming one or more variables can often improve or correct the situation. Transformations typically involve replacing a variable Y with Y^λ . Common values of λ and their interpretations are given in table 8.5. If Y is a proportion, a logit transformation $[\log_e (Y/(1-Y))]$ is often used. When Y is highly skewed, a log transformation is often helpful.

Table 8.5 Common transformations

λ	-2	-1	-0.5	0	0.5	1	2
Transformation	$1/Y^2$	$1/Y$	$1/\sqrt{Y}$	$\log(Y)$	\sqrt{Y}	None	Y^2

When the model violates the normality assumption, you typically attempt a transformation of the response variable. You can use the `powerTransform()` function in the `car` package to generate a maximum-likelihood estimation of the power λ most likely to normalize the variable X^λ . In the next listing, this is applied to the `states` data.

Listing 8.10 Box-Cox transformation to normality

```
> library(car)
> summary(powerTransform(states$Murder))
bcPower Transformation to Normality

      Est.Power Std.Err. Wald Lower Bound Wald Upper Bound
states$Murder  0.6  0.26    0.088      1.1

Likelihood ratio tests about transformation parameters
      LRT df  pval
LR test, lambda=(0) 5.7  1 0.017
LR test, lambda=(1) 2.1  1 0.145
```

The results suggest that you can normalize the variable `Murder` by replacing it with `Murder0.6`. Because 0.6 is close to 0.5, you could try a square-root transformation to improve the model's fit to normality. But in this case, the hypothesis that $\lambda = 1$ can't be rejected ($p = 0.145$), so there's no strong evidence that a transformation is needed in this case. This is consistent with the results of the Q-Q plot in figure 8.9.

Interpreting a log transformation

Log transformations are often used to make highly skewed distributions less skewed. For example, the variable `income` is often right skewed, with more individuals at the lower end of the scale, and a few individuals with very high incomes. How do we interpret regression coefficients with the response variable has been log transformed?

We normally interpret the regression coefficient for X as the expected change in Y for a unit change in X . Consider the model $Y = 3 + 0.6X$. We would predict a 0.6 increase in Y for a one-unit increase in X . Similarly, at 10 unit change in X would be associated with a 0.6(10) or 6 point change in Y .

However, if the model is $\log_e(Y) = 3 + 0.6X$, then a one unit change in X *multiplies* the expected value of Y by $e^{0.6} = 1.06$. Thus, a one-unit increase in X would predict a 6% increase in Y . A 10 unit increase in X would multiply the expected value of Y by $e^{0.6(10)} = 1.82$. Thus, a 10-unit increase in X would predict an 82% increase in Y .

To learn more about interpreting log transformations in linear regression, see Kenneth's Benoit's excellent guide (<https://kenbenoit.net/assets/courses/ME104/logmodels2.pdf>).

When the assumption of linearity is violated, a transformation of the predictor variables can often help. The `boxTidwell()` function in the `car` package can be used to generate maximum-likelihood estimates of predictor powers that can improve linearity. An example of applying the Box–Tidwell transformations to a model that predicts state murder rates from their population and illiteracy rates follows:

```
> library(car)
> boxTidwell(Murder~Population+Illiteracy,data=states)
```

	Score	Statistic	p-value	MLE of lambda
Population	-0.32	0.75	0.87	
Illiteracy	0.62	0.54	1.36	

The results suggest trying the transformations `Population.87` and `Population1.36` to achieve greater linearity. But the score tests for `Population` ($p = .75$) and `Illiteracy` ($p = .54$) suggest that neither variable needs to be transformed. Again, these results are consistent with the component plus residual plots in figure 8.7.

Finally, transformations of the response variable can help in situations of heteroscedasticity (nonconstant error variance). You saw in listing 8.8 that the `spreadLevelPlot()` function in the `car` package offers a power transformation for improving homoscedasticity. Again, in the case of the `states` example, the constant error-variance assumption is met, and no transformation is necessary.

A caution concerning transformations

There's an old joke in statistics: if you can't prove A, prove B and pretend it was A. (For statisticians, that's pretty funny.) The relevance here is that if you transform your variables, your interpretations must be based on the transformed variables, not the original variables. If the transformation makes sense, such as the log of income or the inverse of distance, the interpretation is easier. But how do you interpret the relationship between the frequency of suicidal ideation and the cube root of depression? If a transformation doesn't make sense, you should avoid it.

8.5.3 Adding or deleting variables

Changing the variables in a model will impact the fit of the model. Sometimes, adding an important variable will correct many of the problems that we've discussed. Deleting a troublesome variable can do the same thing.

Deleting variables is a particularly important approach for dealing with multicollinearity. If your only goal is to make predictions, then multicollinearity isn't a problem. But if you want to make interpretations about individual predictor variables, then you must deal with it. The most common approach is to delete one of the variables involved in the multicollinearity (that is,

one of the variables with a $vif > 10$). An alternative is to use lasso or ridge regression, variants of multiple regression designed to deal with multicollinearity situations.

8.5.4 Trying a different approach

As you've just seen, one approach to dealing with multicollinearity is to fit a different type of model (ridge or lasso regression in this case). If there are outliers and/or influential observations, you can fit a robust regression model rather than an OLS regression. If you've violated the normality assumption, you can fit a nonparametric regression model. If there's significant nonlinearity, you can try a nonlinear regression model. If you've violated the assumptions of independence of errors, you can fit a model that specifically takes the error structure into account, such as time-series models or multilevel regression models. Finally, you can turn to generalized linear models to fit a wide range of models in situations where the assumptions of OLS regression don't hold.

We'll discuss some of these alternative approaches in chapter 13. The decision regarding when to try to improve the fit of an OLS regression model and when to try a different approach is a complex one. It's typically based on knowledge of the subject matter and an assessment of which approach will provide the best result.

Speaking of best results, let's turn now to the problem of deciding which predictor variables to include in a regression model.

8.6 Selecting the “best” regression model

When developing a regression equation, you're implicitly faced with a selection of many possible models. Should you include all the variables under study, or drop ones that don't make a significant contribution to prediction? Should you add polynomial and/or interaction terms to improve the fit? The selection of a final regression model always involves a compromise between predictive accuracy (a model that fits the data as well as possible) and parsimony (a simple and replicable model). All things being equal, if you have two models with approximately equal predictive accuracy, you favor the simpler one. This section describes methods for choosing among competing models. The word “best” is in quotation marks because there's no single criterion you can use to make the decision. The final decision requires judgment on the part of the investigator. (Think of it as job security.)

8.6.1 Comparing models

You can compare the fit of two nested models using the `anova()` function in the base installation. A *nested model* is one whose terms are completely included in the other model. In the `states` multiple-regression model, you found that the regression coefficients for `Income` and `Frost` were nonsignificant. You can test whether a model without these two variables predicts as well as one that includes them (see the following listing).

Listing 8.11 Comparing nested models using the `anova()` function

```
> states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])
> fit1 <- lm(Murder ~ Population + Illiteracy + Income + Frost,
  data=states)
> fit2 <- lm(Murder ~ Population + Illiteracy, data=states)
> anova(fit2, fit1)
```

Analysis of Variance Table

```
Model 1: Murder ~ Population + Illiteracy
Model 2: Murder ~ Population + Illiteracy + Income + Frost
 Res.Df  RSS Df Sum of Sq  F Pr(>F)
1    47 289.246
2    45 289.167 2    0.079 0.0061 0.994
```

Here, model 1 is nested within model 2. The `anova()` function provides a simultaneous test that Income and Frost add to linear prediction above and beyond Population and Illiteracy. Because the test is nonsignificant ($p = .994$), you conclude that they don't add to the linear prediction and you're justified in dropping them from your model.

The Akaike Information Criterion (AIC) provides another method for comparing models. The index takes into account a model's statistical fit and the number of parameters needed to achieve this fit. Models with *smaller* AIC values—indicating adequate fit with fewer parameters—are preferred. The criterion is provided by the `AIC()` function (see the following listing).

Listing 8.12 Comparing models with the AIC

```
> fit1 <- lm(Murder ~ Population + Illiteracy + Income + Frost,
  data=states)
> fit2 <- lm(Murder ~ Population + Illiteracy, data=states)
> AIC(fit1,fit2)

  df  AIC
fit1 6 241.6429
fit2 4 237.6565
```

The AIC values suggest that the model without Income and Frost is the better model. Note that although the ANOVA approach requires nested models, the AIC approach doesn't.

Comparing two models is relatively straightforward, but what do you do when there are 4, or 10, or 100 possible models to consider? That's the topic of the next section.

8.6.2 Variable selection

Two popular approaches to selecting a final set of predictor variables from a larger pool of candidate variables are stepwise methods and all-subsets regression.

STEPWISE REGRESSION

In stepwise selection, variables are added to or deleted from a model one at a time, until some stopping criterion is reached. For example, in *forward stepwise* regression, you add predictor variables to the model one at a time, stopping when the addition of variables would no longer improve the model. In *backward stepwise* regression, you start with a model that includes all predictor variables, and then you delete them one at a time until removing variables would degrade the quality of the model. In *stepwise stepwise* regression (usually called *stepwise* to avoid sounding silly), you combine the forward and backward stepwise approaches. Variables are entered one at a time, but at each step, the variables in the model are reevaluated, and those that don't contribute to the model are deleted. A predictor variable may be added to, and deleted from, a model several times before a final solution is reached.

The implementation of stepwise regression methods varies by the criteria used to enter or remove variables. The `step()` function in base R performs stepwise model selection (forward, backward, or stepwise) using an AIC criterion. The next listing applies backward stepwise regression to the multiple regression problem.

Listing 8.13 Backward stepwise selection

```
> states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])

> fit <- lm(Murder ~ Population + Illiteracy + Income + Frost,
  data=states)
> step(fit, direction="backward")

      Start: AIC=97.75
Murder ~ Population + Illiteracy + Income + Frost

      Df Sum of Sq  RSS   AIC
- Frost   1  0.021 289.19 95.753
- Income  1  0.057 289.22 95.759
<none>                289.17 97.749
- Population 1 39.238 328.41 102.111
- Illiteracy 1 144.264 433.43 115.986

Step: AIC=95.75
Murder ~ Population + Illiteracy + Income

      Df Sum of Sq  RSS   AIC
- Income  1  0.057 289.25 93.763
<none>                289.19 95.753
- Population 1 43.658 332.85 100.783
- Illiteracy 1 236.196 525.38 123.605

Step: AIC=93.76
Murder ~ Population + Illiteracy

      Df Sum of Sq  RSS   AIC
<none>                289.25 93.763
```

```
- Population 1 48.517 337.76 99.516
- Illiteracy 1 299.646 588.89 127.311
```

Call:

```
lm(formula = Murder ~ Population + Illiteracy, data = states)
```

Coefficients:

```
(Intercept) Population Illiteracy
1.6515497 0.0002242 4.0807366
```

You start with all four predictors in the model. For each step, the AIC column provides the model AIC resulting from the deletion of the variable listed in that row. The AIC value for `<none>` is the model AIC if no variables are removed. In the first step, Frost is removed, decreasing the AIC from 97.75 to 95.75. In the second step, Income is removed, decreasing the AIC to 93.76. Deleting any more variables would increase the AIC, so the process stops.

Stepwise regression is controversial. Although it may find a good model, there's no guarantee that it will find the "best" model. This is because not every possible model is evaluated. An approach that attempts to overcome this limitation is *all subsets regression*.

ALL SUBSETS REGRESSION

In all subsets regression, every possible model is inspected. The analyst can choose to have all possible results displayed or ask for the *nbest* models of each subset size (one predictor, two predictors, and so on). For example, if `nbest=2`, the two best one-predictor models are displayed, followed by the two best two-predictor models, followed by the two best three-predictor models, up to a model with all predictors.

All subsets regression is performed using the `regsubsets()` function from the `leaps` package. You can choose the R-squared, Adjusted R-squared, or Mallows Cp statistic as your criterion for reporting "best" models.

As you've seen, R-squared is the amount of variance accounted for in the response variable by the predictors variables. Adjusted R-squared is similar but takes into account the number of parameters in the model. R-squared always increases with the addition of predictors. When the number of predictors is large compared to the sample size, this can lead to significant overfitting. The Adjusted R-squared is an attempt to provide a more honest estimate of the population R-squared—one that's less likely to take advantage of chance variation in the data.

In listing 8.14, we'll apply all subsets regression to the `states` data. The `leaps` package presents the results in a plot, but I've found that many people are confused by this graph. The code below presents the same results in the form of a table, which I believe will be easier to understand.

Listing 8.14 All subsets regression

```
library(leaps)
states <- as.data.frame(state.x77[,c("Murder", "Population",
```

```

      "Illiteracy", "Income", "Frost"))

leaps <- regsubsets(Murder ~ Population + Illiteracy + Income +
  Frost, data=states, nbest=4)

subsTable <- function(obj){
  x <- summary(leaps)
  m <- cbind(round(x[[scale]],3), x$which[,-1])
  colnames(m)[1] <- scale
  m[order(m[,1]), ]
}

subsTable(leaps, scale="adjr2)

```

adjr2	Population	Illiteracy	Income	Frost
1 0.033	0	0	1	0
1 0.100	1	0	0	0
1 0.276	0	0	0	1
2 0.292	1	0	0	1
3 0.309	1	0	1	1
3 0.476	0	1	1	1
2 0.480	0	1	1	0
2 0.481	0	1	0	1
1 0.484	0	1	0	0
4 0.528	1	1	1	1
3 0.539	1	1	1	0
3 0.539	1	1	0	1
2 0.548	1	1	0	0

Each line of the table represents a model. The first column indicates the number of predictors in the model. The second column is the scale (adjusted r-squared in this case) used to describe each model's fit and rows are sorted by this scale. (Note: other scale values can be used in place of `adjr2`. See? `regsubsets` for a list of options). The 1/0s in the row indicate which variables are included or excluded from the model.

For example, a model based on the single predictor `Income` has an adjusted R-square of 0.033. A model with the predictors `Population`, `Illiteracy`, and `Income` has an adjusted R-square of 0.539. In contrast, a model using the predictors `Population` and `Illiteracy` alone has an adjusted R-square of 0.548. Here you see that a model with fewer predictors actually has a larger adjusted R-square (something that can't happen with an unadjusted R-square). The table suggests that the two-predictor model (`Population` and `Illiteracy`) is the best.

In most instances, all subsets regression is preferable to stepwise regression, because more models are considered. But when the number of predictors is large, the procedure can require significant computing time. In general, automated variable-selection methods should be seen as an aid rather than a directing force in model selection. A well-fitting model that doesn't make sense doesn't help you. Ultimately, it's your knowledge of the subject matter that should guide you.

8.7 Taking the analysis further

We'll end our discussion of regression by considering methods for assessing model generalizability and predictor relative importance.

8.7.1 Cross-validation

In the previous section, we examined methods for selecting the variables to include in a regression equation. When description is your primary goal, the selection and interpretation of a regression model signals the end of your labor. But when your goal is prediction, you can justifiably ask, "How well will this equation perform in the real world?"

By definition, regression techniques obtain model parameters that are optimal for a given set of data. In OLS regression, the model parameters are selected to minimize the sum of squared errors of prediction (residuals) and, conversely, maximize the amount of variance accounted for in the response variable (R-squared). Because the equation has been optimized for the given set of data, it's unlikely perform as well with a new set of data.

We began this chapter with an example involving a research physiologist who wanted to predict the number of calories an individual will burn from the duration and intensity of their exercise, age, gender, and BMI. If you fit an OLS regression equation to this data, you'll obtain model parameters that uniquely maximize the R-squared for this *particular* set of observations. But our researcher wants to use this equation to predict the calories burned by individuals in general, not only those in the original study. You know that the equation won't perform as well with a new sample of observations, but how much will you lose? *Cross-validation* is a useful method for evaluating the generalizability of a regression equation.

In cross-validation, a portion of the data is selected as the training sample, and a portion is selected as the hold-out sample. A regression equation is developed on the training sample and then applied to the hold-out sample. Because the hold-out sample wasn't involved in the selection of the model parameters, the performance on this sample is a more accurate estimate of the operating characteristics of the model with new data.

In *k-fold cross-validation*, the sample is divided into k subsamples. Each of the k subsamples serves as a hold-out group, and the combined observations from the remaining $k - 1$ subsamples serve as the training group. The performance for the k prediction equations applied to the k hold-out samples is recorded and then averaged. (When k equals n , the total number of observations, this approach is called *jackknifing*.)

You can perform k-fold cross-validation using the `crossval()` function in the `bootstrap` package. The following listing provides a function (called `shrinkage()`) for cross-validating a model's R-square statistic using k-fold cross-validation.

Listing 8.15 Function for k-fold cross-validated R-square

```
shrinkage <- function(fit, k=10, seed=1){
```

```

require(bootstrap)

theta.fit <- function(x,y){lsfit(x,y)}
theta.predict <- function(fit,x){cbind(1,x)%*%fit$coef}

x <- fit$model[,2:ncol(fit$model)]
y <- fit$model[,1]

set.seed(seed)
results <- crossval(x, y, theta.fit, theta.predict, ngroup=k)
r2 <- cor(y, fit$fitted.values)^2
r2cv <- cor(y, results$cv.fit)^2
cat("Original R-square =", r2, "\n")
cat(k, "Fold Cross-Validated R-square =", r2cv, "\n")
}

```

Using this listing, you define your functions, create a matrix of predictor and predicted values, get the raw R-squared and residual standard error, and get the cross-validated R-squared and residual standard error. (Chapter 12 covers bootstrapping in detail.)

The `shrinkage()` function is then used to perform a 10-fold cross-validation with the `states` data, using a model with all four predictor variables:

```

> states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])
> fit <- lm(Murder ~ Population + Income + Illiteracy + Frost, data=states)
> shrinkage(fit)

Original R-square = 0.567
10 Fold Cross-Validated R-square = 0.356

```

You can see that the R-square based on the sample (0.567) is overly optimistic. A better estimate of the amount of variance in murder rates that this model will account for with new data is the cross-validated R-square (0.356). (Note that observations are assigned to the k groups randomly, so a random number seed is provided to make the results reproducible.)

You could use cross-validation in variable selection by choosing a model that demonstrates better generalizability. For example, a model with two predictors (`Population` and `Illiteracy`) shows less R-square shrinkage than the full model:

```

> fit2 <- lm(Murder ~ Population + Illiteracy,data=states)
> shrinkage(fit2)

Original R-square = 0.567
10 Fold Cross-Validated R-square = 0.515

```

This may make the two-predictor model a more attractive alternative.

All other things being equal, a regression equation that's based on a larger training sample and one that's more representative of the population of interest will cross-validate better. You'll get less R-squared shrinkage and make more accurate predictions.

8.7.2 Relative importance

Up to this point in the chapter, we've been asking, "Which variables are useful for predicting the outcome?" But often your real interest is in the question, "Which variables are *most important* in predicting the outcome?" You implicitly want to rank-order the predictors in terms of relative importance. There may be practical grounds for asking the second question. For example, if you could rank-order leadership practices by their relative importance for organizational success, you could help managers focus on the behaviors they most need to develop.

If predictor variables were uncorrelated, this would be a simple task. You would rank-order the predictor variables by their correlation with the response variable. In most cases, though, the predictors are correlated with each other, and this complicates the task significantly.

There have been many attempts to develop a means for assessing the relative importance of predictors. The simplest has been to compare standardized regression coefficients. Standardized regression coefficients describe the expected change in the response variable (expressed in standard deviation units) for a standard deviation change in a predictor variable, holding the other predictor variables constant. You can obtain the standardized regression coefficients in R by standardizing each of the variables in your dataset to a mean of 0 and standard deviation of 1 using the `scale()` function, before submitting the dataset to a regression analysis. (Note that because the `scale()` function returns a matrix and the `lm()` function requires a data frame, you convert between the two in an intermediate step.) The code and results for the multiple regression problem are shown here:

```
> states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])
> zstates <- as.data.frame(scale(states))
> zfit <- lm(Murder~Population + Income + Illiteracy + Frost, data=zstates)
> coef(zfit)

(Intercept) Population    Income Illiteracy    Frost
-9.406e-17  2.705e-01  1.072e-02  6.840e-01  8.185e-03
```

Here you see that a one-standard-deviation increase in illiteracy rate yields a 0.68 standard deviation increase in murder rate, when controlling for population, income, and temperature. Using standardized regression coefficients as your guide, Illiteracy is the most important predictor and Frost is the least.

There have been many other attempts at quantifying relative importance. Relative importance can be thought of as the contribution each predictor makes to R-square, both alone and in combination with other predictors. Several possible approaches to relative importance are captured in the `relaimpo` package written by Ulrike Grömping (<http://mng.bz/KDYF>).

A new method called *relative weights* shows significant promise. The method closely approximates the average increase in R-square obtained by adding a predictor variable across

all possible submodels (Johnson, 2004; Johnson and Lebreton, 2004; Le-Breton and Tonidandel, 2008). A function for generating relative weights is provided in the next listing.

Listing 8.16 `relweights()` for calculating relative importance of predictors

```
relweights <- function(fit,...){
  R <- cor(fit$model)
  nvar <- ncol(R)
  rxx <- R[2:nvar, 2:nvar]
  rxy <- R[2:nvar, 1]
  svd <- eigen(rxx)
  evec <- svd$vectors
  ev <- svd$values
  delta <- diag(sqrt(ev))
  lambda <- evec %*% delta %*% t(evec)
  lambdasq <- lambda ^ 2
  beta <- solve(lambda) %*% rxy
  rsquare <- colSums(beta ^ 2)
  rawwgt <- lambdasq %*% beta ^ 2
  import <- (rawwgt / rsquare) * 100
  import <- as.data.frame(import)
  row.names(import) <- names(fit$model[2:nvar])
  names(import) <- "Weights"
  import <- import[order(import),1, drop=FALSE]
  dotchart(import$Weights, labels=row.names(import),
    xlab="% of R-Square", pch=19,
    main="Relative Importance of Predictor Variables",
    sub=paste("Total R-Square=", round(rsquare, digits=3)),
    ...)
  return(import)
}
```

NOTE The code in listing 8.16 is adapted from an SPSS program generously provided by Dr. Johnson. See Johnson (2000, *Multivariate Behavioral Research*, 35, 1–19) for an explanation of how the relative weights are derived.

In listing 8.17, the `relweights()` function is applied to the `states` data with murder rate predicted by the population, illiteracy, income, and temperature.

You can see from figure 8.19 that the total amount of variance accounted for by the model (R-square=0.567) has been divided among the predictor variables. Illiteracy accounts for 59% of the R-square, Frost accounts for 20.79%, and so forth. Based on the method of relative weights, Illiteracy has the greatest relative importance, followed by Frost, Population, and Income, in that order.

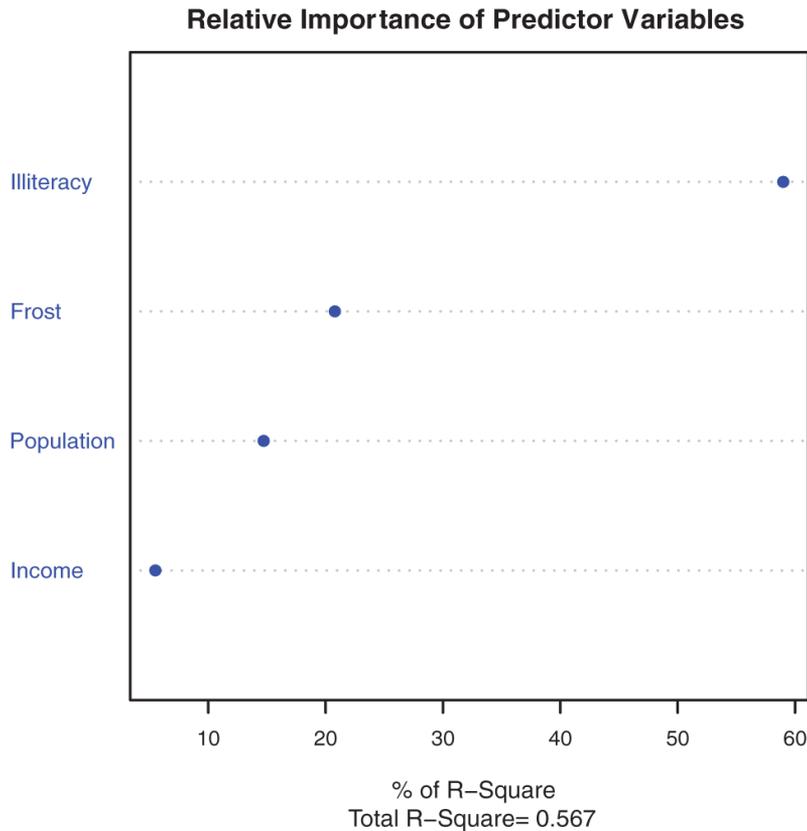


Figure 8.19 Dot chart of relative weights for the states multiple regression problem. Larger weights indicate relatively more important predictors. For example, Illiteracy accounts for 59% of the total explained variance (0.567), whereas Income only accounts for 5.49%. Thus, Illiteracy has greater relative importance than Income in this model.

Listing 8.17 Applying the `relweights()` function

```
> states <- as.data.frame(state.x77[,c("Murder", "Population",
  "Illiteracy", "Income", "Frost")])
> fit <- lm(Murder ~ Population + Illiteracy + Income + Frost, data=states)
> relweights(fit, col="blue")
```

```
Weights
Income    5.49
Population 14.72
Frost    20.79
Illiteracy 59.00
```

Relative-importance measures (and, in particular, the method of relative weights) have wide applicability. They come much closer to our intuitive conception of relative importance than

standardized regression coefficients do, and I expect to see their use increase dramatically in coming years.

8.8 Summary

- *Regression analysis* is a highly interactive and iterative approach that involves fitting models, assessing their fit to statistical assumptions, modifying both the data and the models, and refitting to arrive at a final result.
- *Regression diagnostics* are used to assess the data's fit to statistical assumptions and select methods for modifying the model or the data to meet these assumptions more closely.
- *Numerous methods are available for* selecting the variables to include in a final regression model, including the use of significance tests, fit statistics, and automated solutions such as stepwise and all subsets regression.
- Cross-validation can be used to evaluate a predictive model's likely performance on new samples of data.
- The method of relative weights can be used to address the thorny problem of variable importance: identifying which variables are the most important for predicting an outcome.

9

Analysis of variance

This chapter covers

- Using R to model basic experimental designs
- Fitting and interpreting ANOVA type models
- Evaluating model assumptions

In chapter 7, we looked at regression models for predicting a quantitative response variable from quantitative predictor variables. But there's no reason that we couldn't have included nominal or ordinal factors as predictors as well. When factors are included as explanatory variables, our focus usually shifts from prediction to understanding group differences, and the methodology is referred to as *analysis of variance (ANOVA)*. ANOVA methodology is used to analyze a wide variety of experimental and quasi-experimental designs. This chapter provides an overview of R functions for analyzing common research designs.

First, we'll look at design terminology, followed by a general discussion of R's approach to fitting ANOVA models. Then we'll explore several examples that illustrate the analysis of common designs. Along the way, you'll treat anxiety disorders, lower blood cholesterol levels, help pregnant mice have fat babies, assure that pigs grow long in the tooth, facilitate breathing in plants, and learn which grocery shelves to avoid.

In addition to the base installation, you'll be using the `car`, `rrcov`, `multcomp`, `effects`, `MASS`, `dplyr`, `ggplot2`, and `mvoutlier` packages in the examples. Be sure to install them before trying out the sample code.

9.1 A crash course on terminology

Experimental design in general, and analysis of variance in particular, has its own language. Before discussing the analysis of these designs, we'll quickly review some important terms.

We'll use a series of increasingly complex study designs to introduce the most significant concepts.

Say you're interested in studying the treatment of anxiety. Two popular therapies for anxiety are cognitive behavior therapy (CBT) and eye movement desensitization and reprocessing (EMDR). You recruit 10 anxious individuals and randomly assign half of them to receive five weeks of CBT and half to receive five weeks of EMDR. At the conclusion of therapy, each patient is asked to complete the State-Trait Anxiety Inventory (STAI), a self-report measure of anxiety. The design is outlined in table 9.1.

Table 9.1 One-way between-groups ANOVA

Treatment	
CBT	EMDR
s1	s6
s2	s7
s3	s8
s4	s9
s5	s10

In this design, Treatment is a *between-groups* factor with two levels (CBT, EMDR). It's called a between-groups factor because patients are assigned to one and only one group. No patient receives both CBT and EMDR. The s characters represent the subjects (patients). STAI is the *dependent variable*, and Treatment is the *independent variable*. Because there is an equal number of observations in each treatment condition, you have a *balanced design*. When the sample sizes are unequal across the cells of a design, you have an *unbalanced design*.

The statistical design in table 9.1 is called a *one-way ANOVA* because there's a single classification variable. Specifically, it's a one-way between-groups ANOVA. Effects in ANOVA designs are primarily evaluated through F tests. If the F test for Treatment is significant, you can conclude that the mean STAI scores for two therapies differed after five weeks of treatment.

If you were interested in the effect of CBT on anxiety over time, you could place all 10 patients in the CBT group and assess them at the conclusion of therapy and again six months later. This design is displayed in table 9.2.

Table 9.2 One-way within-groups ANOVA

Patient	Time	
	5 weeks	6 months
s1		
s2		
s3		
s4		
s5		
s6		
s7		
s8		
s9		
s10		

Time is a *within-groups* factor with two levels (five weeks, six months). It's called a within-groups factor because each patient is measured under both levels. The statistical design is a *one-way within-groups ANOVA*. Because each subject is measured more than once, the design is also called a *repeated measures ANOVA*. If the F test for Time is significant, you can conclude that patients' mean STAI scores changed between five weeks and six months.

If you were interested in both treatment differences *and* change over time, you could combine the first two study designs and randomly assign five patients to CBT and five patients to EMDR, and assess their STAI results at the end of therapy (five weeks) and at six months (see table 9.3).

Table 9.3 Two-way factorial ANOVA with one between-groups and one within-groups factor

		Patient	Time	
			5 weeks	6 months
Therapy	CBT	s1		
		s2		
		s3		
		s4		
		s5		
	EMDR	s6		
		s7		
		s8		
		s9		
		s10		

By including both Therapy and Time as factors, you're able to examine the impact of Therapy (averaged across time), Time (averaged across therapy type), and the interaction of Therapy and Time. The first two are called the *main effects*, whereas the interaction is (not surprisingly) called an *interaction effect*.

When you cross two or more factors, as is done here, you have a *factorial ANOVA* design. Crossing two factors produces a two-way ANOVA, crossing three factors produces a three-way ANOVA, and so forth. When a factorial design includes both between-groups and within-groups factors, it's also called a *mixed-model ANOVA*. The current design is a two-way mixed-model factorial ANOVA (phew!).

In this case, you'll have three F tests: one for Therapy, one for Time, and one for the Therapy × Time interaction. A significant result for Therapy indicates that CBT and EMDR differ in their

impact on anxiety. A significant result for Time indicates that anxiety changed from week five to the six-month follow-up. A significant Therapy \times Time interaction indicates that the two treatments for anxiety had a differential impact over time (that is, the change in anxiety from five weeks to six months was different for the two treatments).

Now let's extend the design a bit. It's known that depression can have an impact on therapy, and that depression and anxiety often co-occur. Even though subjects were randomly assigned to treatment conditions, it's possible that the two therapy groups differed in patient depression levels at the initiation of the study. Any post-therapy differences might then be due to the preexisting depression differences and not to your experimental manipulation. Because depression could also explain the group differences on the dependent variable, it's a *confounding* factor. And because you're not interested in depression, it's called a *nuisance* variable.

If you recorded depression levels using a self-report depression measure such as the Beck Depression Inventory (BDI) when patients were recruited, you could statistically adjust for any treatment group differences in depression before assessing the impact of therapy type. In this case, BDI would be called a *covariate*, and the design would be called an *analysis of covariance* (ANCOVA).

Finally, you've recorded a single dependent variable in this study (the STAI). You could increase the validity of this study by including additional measures of anxiety (such as family ratings, therapist ratings, and a measure assessing the impact of anxiety on their daily functioning). When there's more than one dependent variable, the design is called a *multivariate analysis of variance* (MANOVA). If there are covariates present, it's called a *multivariate analysis of covariance* (MANCOVA).

Now that you have the basic terminology under your belt, you're ready to amaze your friends, dazzle new acquaintances, and learn how to fit ANOVA/ANCOVA/MANOVA models with R.

9.2 Fitting ANOVA models

Although ANOVA and regression methodologies developed separately, functionally they're both special cases of the general linear model. You could analyze ANOVA models using the same `lm()` function used for regression in chapter 7. But you'll primarily use the `aoV()` function in this chapter. The results of `lm()` and `aoV()` are equivalent, but the `aoV()` function presents these results in a format that's more familiar to ANOVA methodologists. For completeness, I'll provide an example using `lm()` at the end of this chapter.

9.2.1 The `aoV()` function

The syntax of the `aoV()` function is `aoV(formula, data=dataframe)`. Table 9.4 describes special symbols that can be used in the formulas. In this table, y is the dependent variable and the letters A , B , and C represent factors.

Table 9.4 Special symbols used in R formulas

Symbol	Usage
~	Separates response variables on the left from the explanatory variables on the right. For example, a prediction of y from A, B, and C would be coded $y \sim A + B + C$
:	Denotes an interaction between variables. A prediction of y from A, B, and the interaction between A and B would be coded $y \sim A + B + A:B$
*	Denotes the complete crossing variables. The code $y \sim A*B*C$ expands to $y \sim A + B + C + A:B + A:C + B:C + A:B:C$
^	Denotes crossing to a specified degree. The code $y \sim (A+B+C)^2$ expands to $y \sim A + B + C + A:B + A:C + A:B$
.	Denotes all remaining variables. The code $y \sim .$ expands to $y \sim A + B + C$

Table 9.5 provides formulas for several common research designs. In this table, lowercase letters are quantitative variables, uppercase letters are grouping factors, and `Subject` is a unique identifier variable for subjects.

Table 9.5 Formulas for common research designs

Design	Formula
One-way ANOVA	$y \sim A$
One-way ANCOVA with 1 covariate	$y \sim x + A$
Two-way factorial ANOVA	$y \sim A * B$
Two-way factorial ANCOVA with 2 covariates	$y \sim x1 + x2 + A * B$

Randomized block	$y \sim B + A$ (where B is a blocking factor)
One-way within-groups ANOVA	$y \sim A + \text{Error}(\text{Subject}/A)$
Repeated measures ANOVA with 1 within-groups factor (W) and 1 between-groups factor (B)	$y \sim B * W + \text{Error}(\text{Subject}/W)$

We'll explore in-depth examples of several of these designs later in this chapter.

9.2.2 The order of formula terms

The order in which the effects appear in a formula matters when (a) there's more than one factor and the design is unbalanced, or (b) covariates are present. When either of these two conditions is present, the variables on the right side of the equation will be correlated with each other. In this case, there's no unambiguous way to divide up their impact on the dependent variable. For example, in a two-way ANOVA with unequal numbers of observations in the treatment combinations, the model $y \sim A*B$ will *not* produce the same results as the model $y \sim B*A$.

By default, R employs the Type I (sequential) approach to calculating ANOVA effects (see the sidebar "Order counts!"). The first model can be written as $y \sim A + B + A:B$. The resulting R ANOVA table will assess

- The impact of A on y
- The impact of B on y, controlling for A
- The interaction of A and B, controlling for the A and B main effects

Order counts!

When independent variables are correlated with each other or with covariates, there's no unambiguous method for assessing the independent contributions of these variables to the dependent variable. Consider an unbalanced two-way factorial design with factors A and B and dependent variable y. There are three effects in this design: the A and B main effects and the A × B interaction. Assuming that you're modeling the data using the formula $Y \sim A + B + A:B$ there are three typical approaches for partitioning the variance in y among the effects on the right side of this equation.

Type I (sequential)

Effects are adjusted for those that appear earlier in the formula. A is unadjusted. B is adjusted for the A. The A:B interaction is adjusted for A and B.

Type II (hierarchical)

Effects are adjusted for other effects at the same or lower level. A is adjusted for B. B is adjusted for A. The A:B interaction is adjusted for both A and B.

Type III (marginal)

Each effect is adjusted for every other effect in the model. A is adjusted for B and A:B. B is adjusted for A and A:B. The A:B interaction is adjusted for A and B.

R employs the Type I approach by default. Other programs such as SAS and SPSS employ the Type III approach by default.

The greater the imbalance in sample sizes, the greater the impact that the order of the terms will have on the results. In general, more fundamental effects should be listed earlier in the formula. In particular, covariates should be listed first, followed by main effects, followed by two-way interactions, followed by three-way interactions, and so on. For main effects, more fundamental variables should be listed first. Thus, gender would be listed before treatment. Here's the bottom line: when the research design isn't orthogonal (that is, when the factors and/or covariates are correlated), be careful when specifying the order of effects.

Before moving on to specific examples, note that the `Anova()` function in the `car` package (not to be confused with the standard `anova()` function) provides the option of using the Type II or Type III approach, rather than the Type I approach used by the `aov()` function. You may want to use the `Anova()` function if you're concerned about matching your results to those provided by other packages such as SAS and SPSS. See `help(Anova, package="car")` for details.

9.3 One-way ANOVA

In a one-way ANOVA, you're interested in comparing the dependent variable means of two or more groups defined by a categorical grouping factor. This example comes from the `cholesterol` dataset in the `multcomp` package, taken from Westfall, Tobia, Rom, & Hochberg (1999). Fifty patients received one of five cholesterol-reducing drug regimens (`trt`). Three of the treatment conditions involved the same drug administered as 20 mg once per day (1time), 10mg twice per day (2times), or 5 mg four times per day (4times). The two remaining conditions (drugD and drugE) represented competing drugs. Which drug regimen produced the greatest cholesterol reduction (response)? The analysis is provided in the following listing.

Listing 9.1 One-way ANOVA

```
> library(dplyr)
> data(cholesterol, package="multcomp")
> plotdata <- cholesterol %>% #1
  group_by(trt) %>%
  summarize(n = n(),
            mean = mean(response),
            sd = sd(response),
            ci = qt(0.975, df = n - 1) * sd / sqrt(n))
> plotdata

  trt   n mean  sd  ci
<fct> <int> <dbl> <dbl> <dbl>
1 1time   10  5.78  2.88  2.06
2 2times  10  9.22  3.48  2.49
3 4times  10 12.4  2.92  2.09
4 drugD   10 15.4  3.45  2.47
5 drugE   10 20.9  3.35  2.39

> fit <- aov(response ~ trt, data=cholesterol) #2
```

```

> summary(fit)

      Df Sum Sq Mean Sq F value    Pr(>F)
trt     4  1351    338    32.4 9.8e-13 ***
Residuals 45  469    10
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> library(ggplot2)                #3
> ggplot(plotdata,
  aes(x = trt, y = mean, group = 1)) +
  geom_point(size = 3, color="red") +
  geom_line(linetype="dashed", color="darkgrey") +
  geom_errorbar(aes(ymin = mean - ci,
    ymax = mean + ci),
    width = .1) +
  theme_bw() +
  labs(x="Treatment",
    y="Response",
    title="Mean Plot with 95% Confidence Interval")

```

#1 Group sample sizes, means, standard deviations, and 95% confidence intervals

#2 Tests for group differences (ANOVA)

#3 Plots group means and confidence intervals

Looking at the output, you can see that 10 patients received each of the drug regimens #1. From the means, it appears that drugE produced the greatest cholesterol reduction, whereas 1time produced the least #2. Standard deviations were relatively constant across the five groups, ranging from 2.88 to 3.48. We assume that each treatment group in our study is a sample from a larger potential population of patients that could receive the treatment. For each treatment, the sample mean \pm ci gives us an interval that we are 95% confident includes the true population mean. The ANOVA F test for treatment (trt) is significant ($p < .0001$), providing evidence that the five treatments aren't all equally effective #2.

The `ggplot2` functions are used to create a graph of group means and their confidence intervals #3. A plot of the treatment means, with 95% confidence limits, is provided in figure 9.1 and allows you to clearly see these treatment differences.

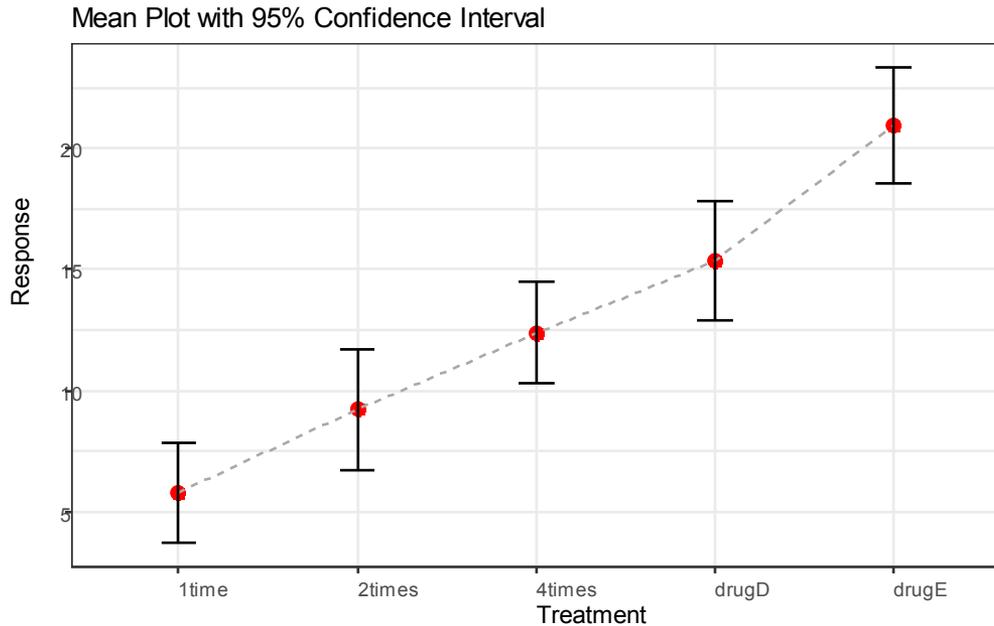


Figure 9.1 Treatment group means with 95% confidence intervals for five cholesterol-reducing drug regimens

By including the confidence intervals in Figure 9.1, we show the degree of certainty (or uncertainty) in our estimates of the population means.

9.3.1 Multiple comparisons

The ANOVA F test for treatment tells you that the five drug regimens aren't equally effective, but it doesn't tell you *which* treatments differ from one another. You can use a multiple comparison procedure to answer this question. For example, the `TukeyHSD()` function provides a test of all pairwise differences between group means, as shown next.

Listing 9.2 Tukey HSD pairwise group comparisons

```
> pairwise <- TukeyHSD(fit)          #1
> pairwise

Fit: aov(formula = response ~ trt)

$trt
      diff   lwr   upr p adj
2times-1time  3.44 -0.658  7.54 0.138
4times-1time  6.59  2.492 10.69 0.000
drugD-1time   9.58  5.478 13.68 0.000
drugE-1time  15.17 11.064 19.27 0.000
4times-2times  3.15 -0.951  7.25 0.205
```

```

drugD-2times 6.14 2.035 10.24 0.001
drugE-2times 11.72 7.621 15.82 0.000
drugD-4times 2.99 -1.115 7.09 0.251
drugE-4times 8.57 4.471 12.67 0.000
drugE-drugD 5.59 1.485 9.69 0.003

> plotdata <- as.data.frame(pairwise[[1]])          #2
> plotdata$conditions <- row.names(plotdata)

> library(ggplot2)                                #3
> ggplot(data=plotdata, aes(x=conditions, y=diff)) +
  geom_point(size=3, color="red") +
  geom_errorbar(aes(ymin=lwr, ymax=upr, width=.2)) +
  geom_hline(yintercept=0, color="red", linetype="dashed") +
  labs(y="Difference in mean levels", x="",
       title="95% family-wise confidence level") +
  theme_bw() +
  coord_flip()

```

```

#1 Calculate pairwise comparisons
#2 Create a dataset of the results
#3 Plot the results

```

For example, the mean cholesterol reductions for 1time and 2times aren't significantly different from each other ($p = 0.138$), whereas the difference between 1time and 4times is significantly different ($p < .001$).

The pairwise comparisons are plotted in figure 9.2. In this graph, confidence intervals that include 0 indicate treatments that aren't significantly different ($p > 0.5$). Here, we can see that the largest mean difference is between `drugE` and `1time` and that the difference is significant (the confidence interval does not include 0).

Before moving on, I should point out that we could have created the graphs in figures 9.2 using base graphics. In this case the code would simply be `plot(pairwise)`. The advantage of the `ggplot2` approach is that it creates a more attractive plot and allows you to fully customize the graph to meet your needs.

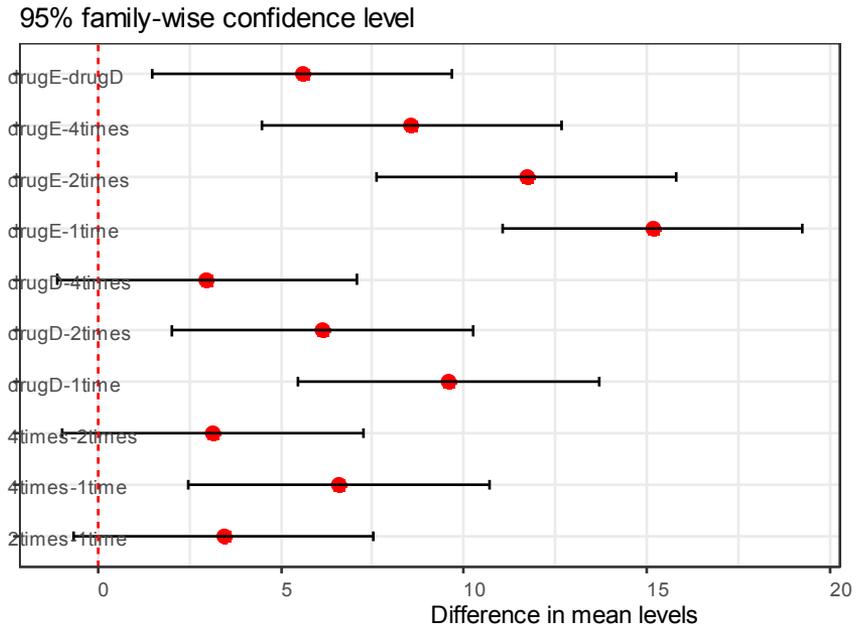


Figure 9.2 Plot of Tukey HSD pairwise mean comparisons

The `glht()` function in the `multcomp` package provides a much more comprehensive set of methods for multiple mean comparisons that you can use for both linear models (such as those described in this chapter) and generalized linear models (covered in chapter 13). The following code reproduces the Tukey HSD test, along with a different graphical representation of the results (figure 9.3):

```
> tuk <- glht(fit, linfct=mcp(trt="Tukey"))
> summary(tuk)
```

Simultaneous Tests for General Linear Hypotheses

Multiple Comparisons of Means: Tukey Contrasts

Fit: aov(formula = response ~ trt, data = cholesterol)

Linear Hypotheses:

	Estimate	Std. Error	t value	Pr(> t)
2times - 1time == 0	3.443	1.443	2.385	0.13812
4times - 1time == 0	6.593	1.443	4.568	< 0.001 ***
drugD - 1time == 0	9.579	1.443	6.637	< 0.001 ***
drugE - 1time == 0	15.166	1.443	10.507	< 0.001 ***
4times - 2times == 0	3.150	1.443	2.182	0.20504
drugD - 2times == 0	6.136	1.443	4.251	< 0.001 ***

```

drugE - 2times == 0 11.723  1.443  8.122 < 0.001 ***
drugD - 4times == 0  2.986  1.443  2.069 0.25120
drugE - 4times == 0  8.573  1.443  5.939 < 0.001 ***
drugE - drugD == 0  5.586  1.443  3.870 0.00308 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
(Adjusted p values reported -- single-step method)

> labels1 <- cld(tuk, level=.05)$mclletters$Letters
> labels2 <- paste(names(labels1), "\n", labels1)
> ggplot(data=fit$model, aes(x=trt, y=response)) +
  scale_x_discrete(breaks=names(labels1), labels=labels2) +
  geom_boxplot(fill="lightgrey") +
  theme_bw() +
  labs(x="Treatment",
       title="Distribution of Response Scores by Treatment",
       subtitle="Groups without overlapping letters differ significantly (p < .05)")

```

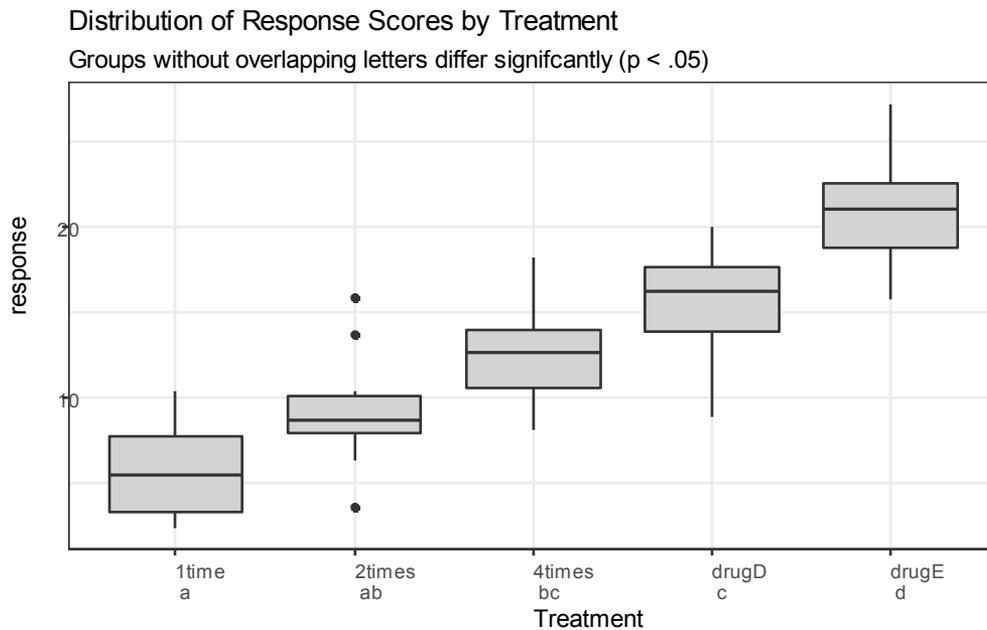


Figure 9.3 Tukey HSD tests provided by the multcomp package

The `level` option in the `cld()` function provides the significance level to use (0.05, or 95% confidence in this case).

Groups (represented by box plots) that have the same letter don't have significantly different means. You can see that 1time and 2times aren't significantly different (they both have the

letter a) and that 2times and 4times aren't significantly different (they both have the letter b); but that 1time and 4times are different (they don't share a letter). Personally, I find figure 9.3 easier to read than figure 9.2. It also has the advantage of providing information on the distribution of scores within each group.

From these results, you can see that taking the cholesterol-lowering drug in 5 mg doses four times a day was better than taking a 20 mg dose once per day. The competitor drugD wasn't superior to this four-times-per-day regimen. But competitor drugE was superior to both drugD and all three dosage strategies for the focus drug.

Multiple comparisons methodology is a complex and rapidly changing area of study. To learn more, see Bretz, Hothorn, and Westfall (2010).

9.3.2 Assessing test assumptions

As you saw in the previous chapter, confidence in results depends on the degree to which your data satisfies the assumptions underlying the statistical tests. In a one-way ANOVA, the dependent variable is assumed to be normally distributed and have equal variance in each group. You can use a Q-Q plot to assess the normality assumption:

```
> library(car)
> fit <- aov(response ~ trt, data=cholesterol)
> qqPlot(fit, simulate=TRUE, main="Q-Q Plot")
```

The graph is provided in figure 9.4. By default, the two observations with the highest standardized residuals are identified by data frame row number. The data falls within the 95% confidence envelope, suggesting that the normality assumption has been met fairly well.

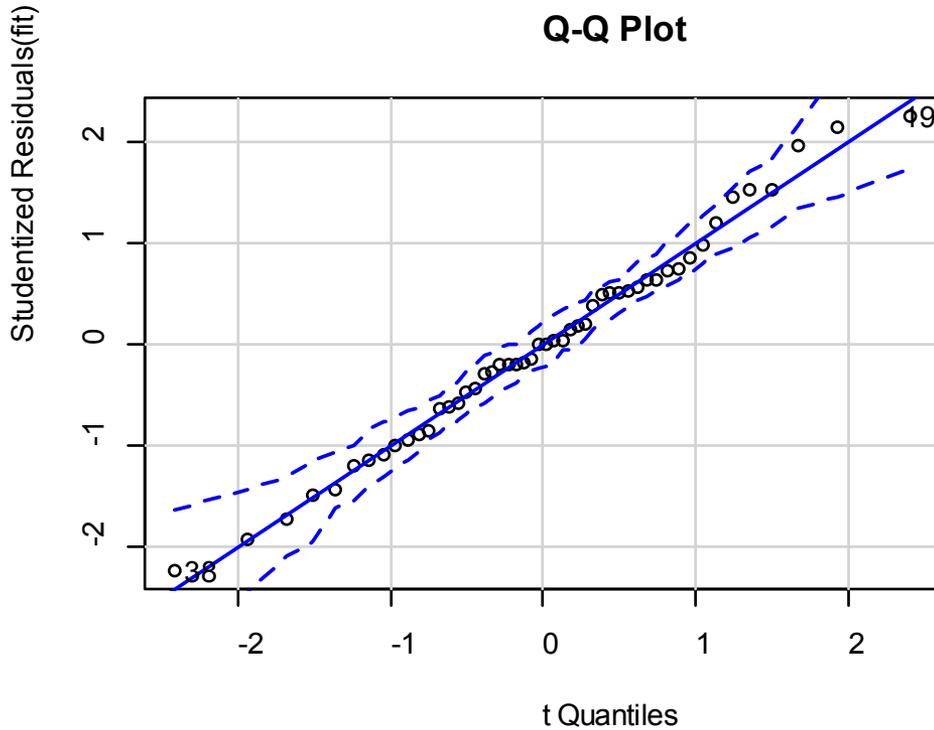


Figure 9.4 Test of normality

R provides several tests for the equality (homogeneity) of variances. For example, you can perform Bartlett's test with this code:

```
> bartlett.test(response ~ trt, data=cholesterol)

Bartlett test of homogeneity of variances

data: response by trt
Bartlett's K-squared = 0.5797, df = 4, p-value = 0.9653
```

Bartlett's test indicates that the variances in the five groups don't differ significantly ($p = 0.97$). Other possible tests include the Fligner-Killeen test (provided by the `fligner.test()` function) and the Brown-Forsythe test (provided by the `hov()` function in the `HH` package). Although not shown, the other two tests reach the same conclusion.

Finally, analysis of variance methodologies can be sensitive to the presence of outliers. You can test for outliers using the `outlierTest()` function in the `car` package:

```
> library(car)
> outlierTest(fit)

No Studentized residuals with Bonferonni p < 0.05
Largest |rstudent|:
  rstudent unadjusted p-value Bonferonni p
19 2.251149    0.029422      NA
```

From the output, you can see that there's no indication of outliers in the cholesterol data (NA occurs when $p > 1$). Taking the Q-Q plot, Bartlett's test, and outlier test together, the data appear to fit the ANOVA model quite well. This, in turn, adds to your confidence in the results.

9.4 One-way ANCOVA

A one-way analysis of covariance (ANCOVA) extends the one-way ANOVA to include one or more quantitative covariates. This example comes from the `litter` dataset in the `multcomp` package (see Westfall et al., 1999). Pregnant mice were divided into four treatment groups; each group received a different dose of a drug (0, 5, 50, or 500). The mean post-birth weight for each litter was the dependent variable, and gestation time was included as a covariate. The analysis is given in the following listing.

Listing 9.3 One-way ANCOVA

```
> library(multcomp)
> library(dplyr)
> litter %>%
  group_by(dose) %>%
  summarise(n=n(), mean=mean(gesttime), sd=sd(gesttime))

  dose  n mean  sd
<fct> <int> <dbl> <dbl>
1 0    20 22.1 0.438
2 5    19 22.2 0.451
3 50   18 21.9 0.404
4 500  17 22.2 0.431

> fit <- aov(weight ~ gesttime + dose, data=litter)
> summary(fit)
      Df Sum Sq Mean Sq F value Pr(>F)
gesttime  1 134.3  134.30   8.049 0.00597 **
dose      3  137.1   45.71   2.739 0.04988 *
Residuals 69 1151.3  16.69
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

From the `summarise()` function, you can see that there is an unequal number of litters at each dosage level, with 20 litters at zero dosage (no drug) and 17 litters at dosage 500. Based on the group means, the *no-drug* group had the highest mean litter weight (32.3). The ANCOVA F tests indicate that (a) gestation time was related to birth weight, and (b) drug dosage was related to birth weight after controlling for gestation time. The mean birth weight isn't the same for each of the drug dosages, after controlling for gestation time.

Because you're using a covariate, you may want to obtain adjusted group means—that is, the group means obtained after partialing out the effects of the covariate. You can use the `effect()` function in the `effects` library to calculate adjusted means:

```
> library(effects)
> effect("dose", fit)

dose effect
dose
  0  5  50 500
32.4 28.9 30.6 29.3
```

These are the mean litter weights for each treatment dose, after statistically adjusting for initial differences in gestation time. In this case, the adjusted means differ quite a bit from the unadjusted means produced by the `summarise()` function. The `effects` package provides a powerful method of obtaining adjusted means for complex research designs and presenting them visually. See the package documentation on CRAN for more details.

As with the one-way ANOVA example in the last section, the F test for dose indicates that the treatments don't have the same mean birth weight, but it doesn't tell you which means differ from one another. Again, you can use the multiple comparison procedures provided by the `multcomp` package to compute all pairwise mean comparisons. Additionally, the `multcomp` package can be used to test specific user-defined hypotheses about the means.

Suppose you're interested in whether the no-drug condition differs from the three-drug condition. The code in the following listing can be used to test this hypothesis.

Listing 9.4 Multiple comparisons employing user-supplied contrasts

```
> library(multcomp)
> contrast <- rbind("no drug vs. drug" = c(3, -1, -1, -1))
> summary(glht(fit, linfct=mcp(dose=contrast)))

Multiple Comparisons of Means: User-defined Contrasts

Fit: aov(formula = weight ~ gesttime + dose)

Linear Hypotheses:
      Estimate Std. Error t value Pr(>|t|)
no drug vs. drug == 0  8.284   3.209  2.581 0.0120 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The contrast `c(3, -1, -1, -1)` specifies a comparison of the first group with the average of the other three. The hypothesis is tested with a t statistic (2.581 in this case), which is significant at the $p < .05$ level. Therefore, you can conclude that the no-drug group has a higher birth weight than drug conditions. Other contrasts can be added to the `rbind()` function (see `help(glht)` for details).

9.4.1 Assessing test assumptions

ANCOVA designs make the same normality and homogeneity of variance assumptions described for ANOVA designs, and you can test these assumptions using the same procedures described in section 9.3.2. In addition, standard ANCOVA designs assume homogeneity of regression slopes. In this case, it's assumed that the regression slope for predicting birth weight from gestation time is the same in each of the four treatment groups. A test for the homogeneity of regression slopes can be obtained by including a gestation \times dose interaction term in your ANCOVA model. A significant interaction would imply that the relationship between gestation and birth weight depends on the level of the dose variable. The code and results are provided in the following listing.

Listing 9.5 Testing for homogeneity of regression slopes

```
> library(multcomp)
> fit2 <- aov(weight ~ gesttime*dose, data=litter)
> summary(fit2)
      Df Sum Sq Mean Sq F value Pr(>F)
gesttime  1  134   134  8.29 0.0054 **
dose      3  137    46  2.82 0.0456 *
gesttime:dose 3  82   27  1.68 0.1789
Residuals 66 1069   16
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The interaction is nonsignificant, supporting the assumption of equality of slopes. If the assumption is untenable, you could try transforming the covariate or dependent variable, using a model that accounts for separate slopes, or employing a nonparametric ANCOVA method that doesn't require homogeneity of regression slopes. See the `sm.ancova()` function in the `sm` package for an example of the latter.

9.4.2 Visualizing the results

We can use `ggplot2` to visualize of the relationship between the dependent variable, the covariate, and the factor. For example,

```
pred <- predict(fit)
library(ggplot2)
ggplot(data = cbind(litter, pred),
       aes(gesttime, weight)) + geom_point() +
  facet_wrap(~ dose, nrow=1) + geom_line(aes(y=pred)) +
  labs(title="ANCOVA for weight by gesttime and dose") +
  theme_bw() +
  theme(axis.text.x = element_text(angle=45, hjust=1),
        legend.position="none")
```

produces the plot shown in figure 9.5.

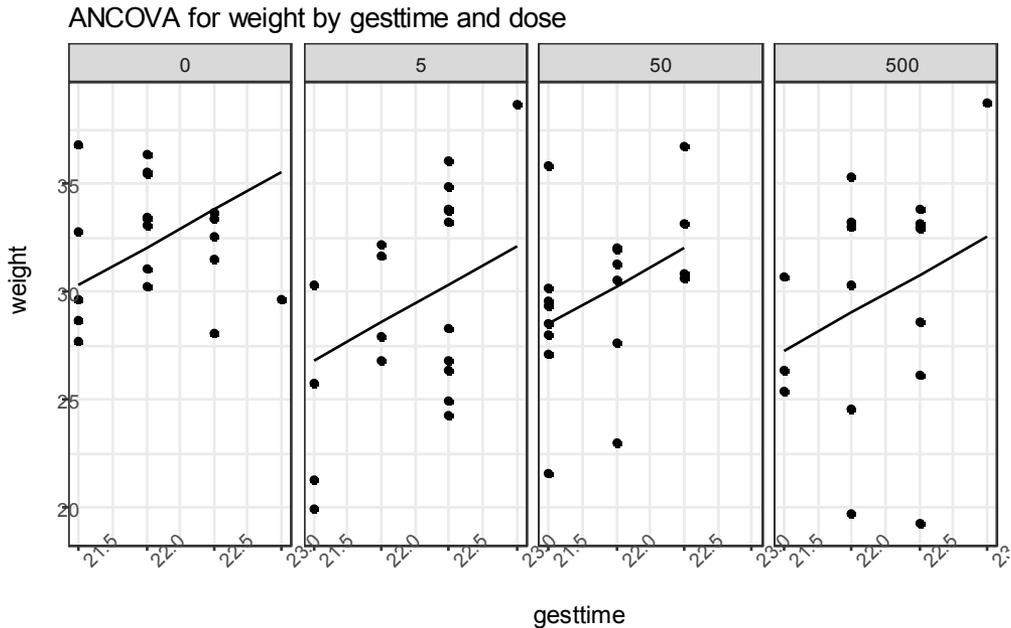


Figure 9.5 Plot of the relationship between gestation time and birth weight for each of four drug treatment groups

Here you can see that the regression lines for predicting birth weight from gestation time are parallel in each group but have different intercepts. As gestation time increases, birth weight increases. Additionally, you can see that the zero-dose group has the largest intercept and the five-dose group has the lowest intercept. The lines are parallel because they've been specified to be. If you used the code

```
ggplot(data = litter, aes(gesttime, weight)) +
  geom_point() + geom_smooth(method="lm", se=FALSE) +
  facet_wrap(~ dose, nrow=1)
```

instead, you'd generate a plot that allows both the slopes and intercepts to vary by group. This approach is useful for visualizing the case where the homogeneity of regression slopes doesn't hold.

9.5 Two-way factorial ANOVA

In a two-way factorial ANOVA, subjects are assigned to groups that are formed from the cross-classification of two factors. This example uses the `ToothGrowth` dataset in the base installation to demonstrate a two-way between-groups ANOVA. Sixty guinea pigs are randomly assigned to receive one of three levels of ascorbic acid (0.5, 1, or 2 mg) and one of two

delivery methods (orange juice or Vitamin C), under the restriction that each treatment combination has 10 guinea pigs. The dependent variable is tooth length. The following listing shows the code for the analysis.

Listing 9.6 Two-way ANOVA

```
> library(dplyr)
> data(ToothGrowth)
> ToothGrowth$dose <- factor(ToothGrowth$dose)      #1
> stats <- ToothGrowth %>%                          #2
  group_by(supp, dose) %>%
  summarise(n=n(), mean=mean(len), sd=sd(len),
            ci = qt(0.975, df = n - 1) * sd / sqrt(n))
> stats

# A tibble: 6 x 6
# Groups:   supp [2]
  supp dose   n mean  sd  ci
<fct> <fct> <int> <dbl> <dbl> <dbl>
1 OJ  0.5   10 13.2  4.46  3.19
2 OJ  1     10 22.7  3.91  2.80
3 OJ  2     10 26.1  2.66  1.90
4 VC  0.5   10  7.98  2.75  1.96
5 VC  1     10 16.8  2.52  1.80
6 VC  2     10 26.1  4.80  3.43

> fit <- aov(len ~ supp*dose, data=ToothGrowth)      #3
> summary(fit)

      Df Sum Sq Mean Sq F value Pr(>F)
supp   1  205.4   205.4  15.572 0.000231 ***
dose   2 2426.4  1213.2   92.000 < 2e-16 ***
supp:dose  2  108.3    54.2   4.107 0.021860 *
Residuals 54  712.1    13.2

---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

#1 Prepare data

#2 Calculate summary statistics

#3 Fit 2-way ANOVA model

First, the `dose` variable is converted to a factor so that the `aov()` function will treat it as a grouping variable, rather than a numeric covariate #1. Next, summary statistics (n, mean, standard deviation and confidence interval for the mean) are calculated for each combination of treatments #2. The sample sizes indicate that you have a balanced design (equal sample sizes in each cell of the design). The 2-way ANOVA model is fitted to the data #3, and the `summary()` function indicates that both main effects (`supp` and `dose`) and the interaction between these factors are significant.

You can visualize the results in several ways, including the `interaction.plot()` function in base R, the `plotmeans()` function in the `gplots` package, and the `interaction2wt()` function in the `HH` package. In the code below, we'll use `ggplot2` to plot the means and 95%

confidence intervals for the means for this two-way ANOVA. One advantage of using ggplot2 is that we can customize the graph to suite are research and esthetic needs. The resulting graph is presented in figure 9.6.

```
library(ggplot2)
pd <- position_dodge(0.2)
ggplot(data=stats,
  aes(x = dose, y = mean,
    group=supp,
    color=supp,
    linetype=supp)) +
  geom_point(size = 2,
    position=pd) +
  geom_line(position=pd) +
  geom_errorbar(aes(ymin = mean - ci, ymax = mean + ci),
    width = .1,
    position=pd) +
  theme_bw() +
  scale_color_manual(values=c("blue", "red")) +
  labs(x="Dose",
    y="Mean Length",
    title="Mean Plot with 95% Confidence Interval")
```

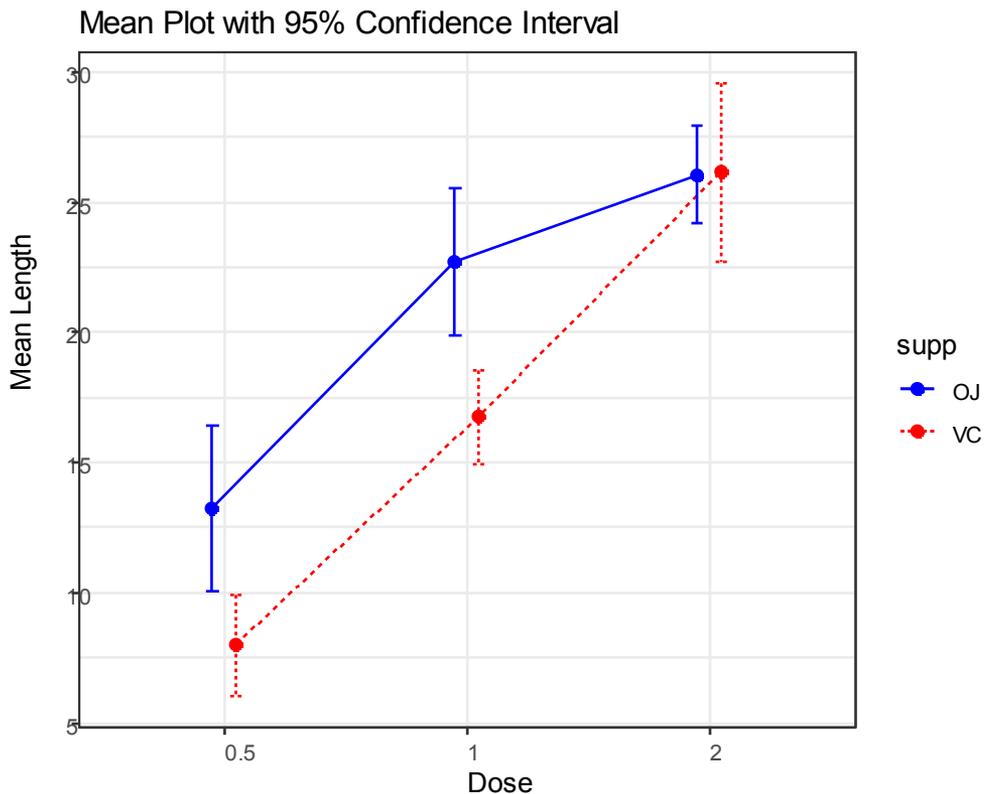


Figure 9.6 Interaction between dose and delivery mechanism on tooth growth. The plot of means was created using the `ggplot2` code.

The graph indicates that tooth growth increases with the dose of ascorbic acid for both orange juice and Vitamin C. For the 0.5 and 1 mg doses, orange juice produced more tooth growth than Vitamin C. For 2 mg of ascorbic acid, both delivery methods produced identical growth.

Although I don't cover the tests of model assumptions and mean comparison procedures, they're a natural extension of the methods you've seen so far. Additionally, the design is balanced, so you don't have to worry about the order of effects.

9.6 Repeated measures ANOVA

In repeated measures ANOVA, subjects are measured more than once. This section focuses on a repeated measures ANOVA with one within-groups and one between-groups factor (a common design). We'll take our example from the field of physiological ecology. Physiological ecologists study how the physiological and biochemical processes of living systems respond to variations in environmental factors (a crucial area of study given the realities of global warming). The `co2` dataset included in the base installation contains the results of a study of cold tolerance in Northern and Southern plants of the grass species *Echinochloa crus-galli* (Potvin, Lechowicz, & Tardif, 1990). The photosynthetic rates of chilled plants were compared with the photosynthetic rates of nonchilled plants at several ambient CO₂ concentrations. Half the plants were from Quebec, and half were from Mississippi.

In this example, we'll focus on chilled plants. The dependent variable is carbon dioxide uptake (uptake) in ml/L, and the independent variables are Type (Quebec versus Mississippi) and ambient CO₂ concentration (conc) with seven levels (ranging from 95 to 1000 $\mu\text{mol}/\text{m}^2 \text{ sec}$). Type is a between-groups factor, and conc is a within-groups factor. Type is already stored as a factor, but you'll need to convert conc to a factor before continuing. The analysis is presented in the next listing.

Listing 9.7 Repeated measures ANOVA with one between- and within-groups factor

```
> data(CO2)
> CO2$conc <- factor(CO2$conc)
> w1b1 <- subset(CO2, Treatment=='chilled')
> fit <- aov(uptake ~ conc*Type + Error(Plant/(conc)), w1b1)
> summary(fit)
```

Error: Plant

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Type	1	2667	2667	60.4	0.0015 **
Residuals	4	177	44		

Signif. codes: 0 '****' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Error: Plant:conc

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
--	----	--------	---------	---------	--------

```

conc      6 1472 245.4 52.5 1.3e-12 ***
conc:Type 6  429  71.5 15.3 3.7e-07 ***
Residuals 24  112   4.7
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> library(dplyr)
> stats <- CO2 %>%
  group_by(conc, Type) %>%
  summarise(mean_conc = mean(uptake))

> library(ggplot2)
> ggplot(data=stats, aes(x=conc, y=mean_conc,
  group=Type, color=Type, linetype=Type)) +
  geom_point(size=2) +
  geom_line(size=1) +
  theme_bw() + theme(legend.position="top") +
  labs(x="Concentration", y="Mean Uptake",
  title="Interaction Plot for Plant Type and Concentration")

```

The ANOVA table indicates that the Type and concentration main effects and the Type \times concentration interaction are all significant at the 0.01 level. A plot of the interaction is provided in figure 9.7. In this case, I've left out confidence intervals to keep the graph from becoming too busy.

Interaction Plot for Plant Type and Concentration

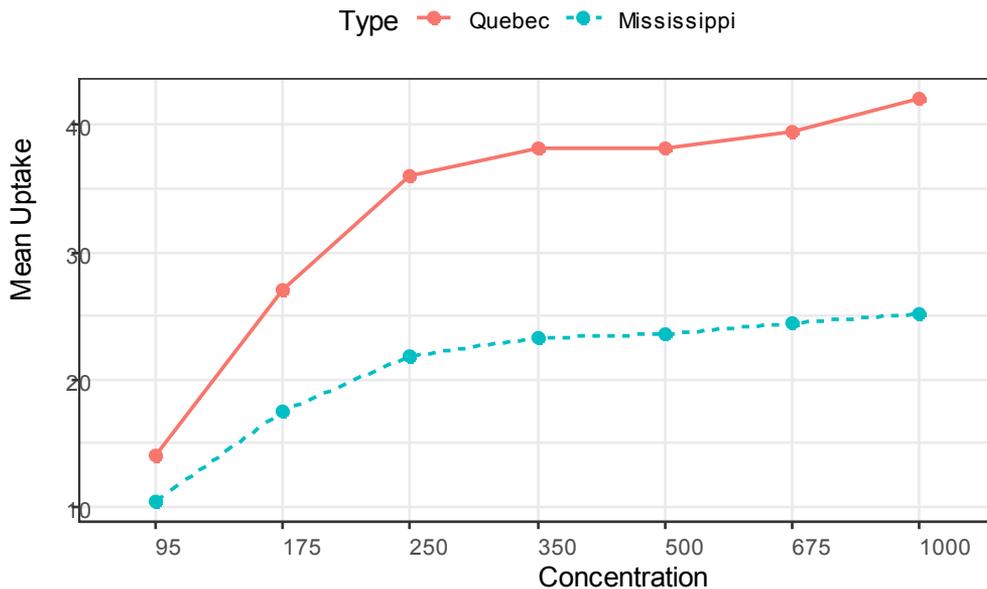


Figure 9.7 Interaction of ambient CO2 concentration and plant type on CO2 uptake.

In order to demonstrate a different presentation of the interaction, the `geom_boxplot()` function is used to plot the same data. The results are provided in figure 9.8.

```
library(ggplot2)
ggplot(data=CO2, aes(x=conc, y=uptake, fill=Type)) +
  geom_boxplot() +
  theme_bw() + theme(legend.position="top") +
  scale_fill_manual(values=c("gold", "green"))+
  labs(x="Concentration", y="Uptake",
       title="Chilled Quebec and Mississippi Plants")
```

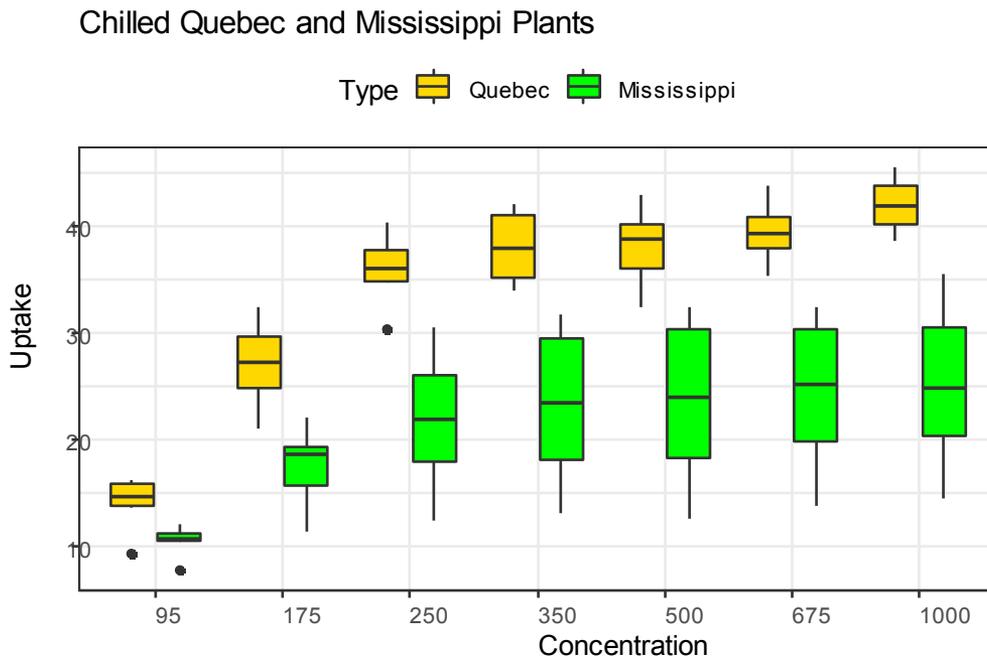


Figure 9.8 Interaction of ambient CO₂ concentration and plant type on CO₂ uptake..

From either graph, you can see that there's a greater carbon dioxide uptake in plants from Quebec compared to Mississippi. The difference is more pronounced at higher ambient CO₂ concentrations.

NOTE Datasets are typically in *wide format*, where columns are variables and rows are observations, and there's a single row for each subject. The `litter` data frame from section 9.4 is a good example. When dealing with repeated measures designs, you typically need the data in *long format* before fitting models. In long format, each measurement of the dependent variable is placed in its own row. The `CO2` dataset follows this form. Luckily, the `tidyr` package described in chapter 5 (section 5.6.2) can easily reorganize your data into the required format.

The many approaches to mixed-model designs

The CO2 example in this section was analyzed using a traditional repeated measures ANOVA. The approach assumes that the covariance matrix for any within-groups factor follows a specified form known as sphericity. Specifically, it assumes that the variances of the differences between any two levels of the within-groups factor are equal. In real-world data, it's unlikely that this assumption will be met. This has led to a number of alternative approaches, including the following:

- a. Using the `lmer()` function in the `lme4` package to fit linear mixed models (Bates, 2005)
- b. Using the `Anova()` function in the `car` package to adjust traditional test statistics to account for lack of sphericity (for example, the Geisser–Greenhouse correction)
- c. Using the `gls()` function in the `nlme` package to fit generalized least squares models with specified variance-covariance structures (UCLA, 2009)
- d. Using multivariate analysis of variance to model repeated measured data (Hand, 1987)

Coverage of these approaches is beyond the scope of this text. If you're interested in learning more, check out Pinheiro and Bates (2000) and Zuur et al. (2009).

Up to this point, all the methods in this chapter have assumed that there's a single dependent variable. In the next section, we'll briefly consider designs that include more than one outcome variable.

9.7 Multivariate analysis of variance (MANOVA)

If there's more than one dependent (outcome) variable, you can test them simultaneously using a multivariate analysis of variance (MANOVA). The following example is based on the `UScereal` dataset in the `MASS` package. The dataset comes from Venables & Ripley (1999). In this example, you're interested in whether the calories, fat, and sugar content of US cereals vary by store shelf, where 1 is the bottom shelf, 2 is the middle shelf, and 3 is the top shelf. Calories, fat, and sugars are the dependent variables, and shelf is the independent variable, with three levels (1, 2, and 3). The analysis is presented in the following listing.

Listing 9.8 One-way MANOVA

```
> data(UScereal, package="MASS")
> shelf <- factor(UScereal$shelf)
> shelf <- factor(shelf)
> y <- cbind(UScereal$calories, UScereal$fat, UScereal$sugars)
> colnames(y) <- c("calories", "fat", "sugars")
> aggregate(y, by=list(shelf=shelf), FUN=mean)

  shelf calories  fat sugars
1    1    119 0.662   6.3
2    2    130 1.341  12.5
3    3    180 1.945  10.9

> cov(y)

  calories fat sugars
calories 3895.2 60.67 180.38
```

```

fat      60.7 2.71 4.00
sugars   180.4 4.00 34.05

> fit <- manova(y ~ shelf)
> summary(fit)

      Df Pillai approx F num Df den Df Pr(>F)
shelf  2 0.402  5.12   6 122 1e-04 ***
Residuals 62
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

> summary.aov(fit)                                #1

Response calories :
      Df Sum Sq Mean Sq F value Pr(>F)
shelf  2 50435 25218  7.86 0.00091 ***
Residuals 62 198860 3207
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Response fat :
      Df Sum Sq Mean Sq F value Pr(>F)
shelf  2  18.4  9.22  3.68 0.031 *
Residuals 62 155.2  2.50
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Response sugars :
      Df Sum Sq Mean Sq F value Pr(>F)
shelf  2  381  191  6.58 0.0026 **
Residuals 62 1798  29
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

#1 Prints univariate results

First, the shelf variable is converted to a factor so that it can represent a grouping variable in the analyses. Next, the `cbind()` function is used to form a matrix of the three dependent variables (calories, fat, and sugars). The `aggregate()` function provides the shelf means, and the `cov()` function provides the variance and the covariances across cereals.

The `manova()` function provides the multivariate test of group differences. The significant F value indicates that the three groups differ on the set of nutritional measures. Note that the shelf variable was converted to a factor so that it can represent a grouping variable.

Because the multivariate test is significant, you can use the `summary.aov()` function to obtain the univariate one-way ANOVAs #1. Here, you see that the three groups differ on each nutritional measure considered separately. Finally, you can use a mean comparison procedure (such as `TukeyHSD`) to determine which shelves differ from each other for each of the three dependent variables (omitted here to save space).

9.7.1 Assessing test assumptions

The two assumptions underlying a one-way MANOVA are multivariate normality and homogeneity of variance-covariance matrices. The first assumption states that the vector of dependent variables jointly follows a multivariate normal distribution. You can use a Q-Q plot to assess this assumption (see the sidebar “A theory interlude” for a statistical explanation of how this works).

A theory interlude

If you have $p \times 1$ multivariate normal random vector x with mean μ and covariance matrix Σ , then the squared Mahalanobis distance between x and μ is chi-square distributed with p degrees of freedom. The Q-Q plot graphs the quantiles of the chi-square distribution for the sample against the Mahalanobis D-squared values. To the degree that the points fall along a line with slope 1 and intercept 0, there's evidence that the data is multivariate normal.

The code is provided in listing 9.9, and the resulting graph is displayed in figure 9.9.

Listing 9.9 Assessing multivariate normality

```
> center <- colMeans(y)
> n <- nrow(y)
> p <- ncol(y)
> cov <- cov(y)
> d <- mahalanobis(y,center,cov)
> coord <- qqplot(qchisq(ppoints(n),df=p),
  d, main="Q-Q Plot Assessing Multivariate Normality",
  ylab="Mahalanobis D2")
> abline(a=0,b=1)
> identify(coord$x, coord$y, labels=row.names(UScereal))
```

If the data follow a multivariate normal distribution, then points will fall on the line. The `identify()` function allows you to interactively identify points in the graph. Click on each point of interest, then hit ESC or the Finish button. Here, the dataset appears to violate multivariate normality, primarily due to the observations for Wheaties Honey Gold and Wheaties. You may want to delete these two cases and rerun the analyses.

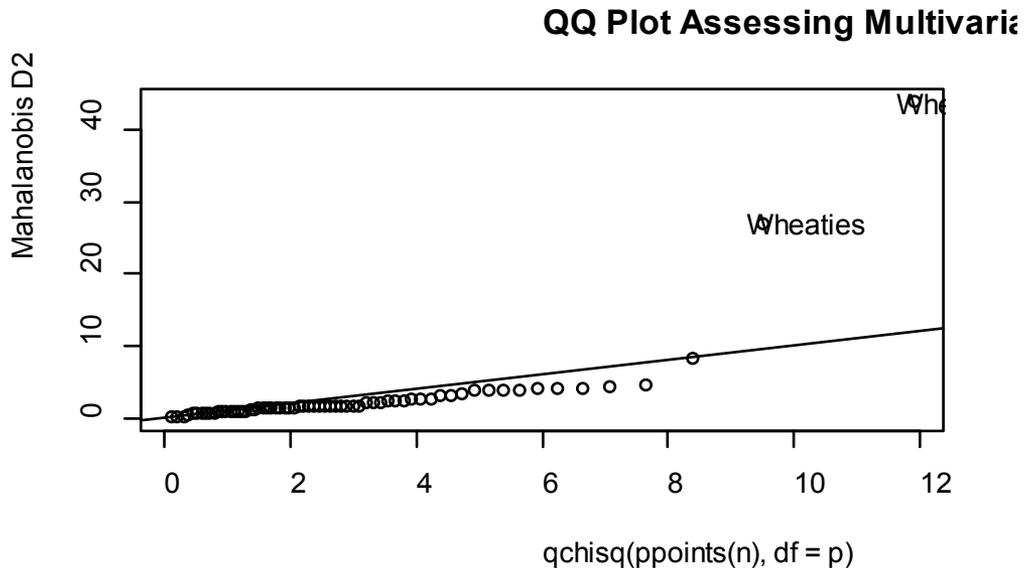


Figure 9.9 A Q-Q plot for assessing multivariate normality

The homogeneity of variance-covariance matrices assumption requires that the covariance matrix for each group is equal. The assumption is usually evaluated with a Box's M test. R doesn't include a function for Box's M, but an internet search will provide the appropriate code. Unfortunately, the test is sensitive to violations of normality, leading to rejection in most typical cases. This means that we don't yet have a good working method for evaluating this important assumption (but see Anderson [2006] and Silva et al. [2008] for interesting alternative approaches not yet available in R).

Finally, you can test for multivariate outliers using the `aq.plot()` function in the `mvoutlier` package. The code in this case looks like this:

```
library(mvoutlier)
outliers <- aq.plot(y)
outliers
```

Try it, and see what you get!

9.7.2 Robust MANOVA

If the assumptions of multivariate normality or homogeneity of variance-covariance matrices are untenable, or if you're concerned about multivariate outliers, you may want to consider using a robust or nonparametric version of the MANOVA test instead. A robust version of the

one-way MANOVA is provided by the `Wilks.test()` function in the `rrcov` package. The `adonis()` function in the `vegan` package can provide the equivalent of a nonparametric MANOVA. The following listing applies `Wilks.test()` to the example.

Listing 9.10 Robust one-way MANOVA

```
> library(rrcov)
> Wilks.test(y,shelf,method="mcd")

Robust One-way MANOVA (Bartlett Chi2)

data: x
Wilks' Lambda = 0.511, Chi2-Value = 23.96, DF = 4.98, p-value =
0.0002167
sample estimates:
 calories fat sugars
1  120 0.701  5.66
2  128 1.185 12.54
3  161 1.652 10.35
```

From the results, you can see that using a robust test that's insensitive to both outliers and violations of MANOVA assumptions still indicates that the cereals on the top, middle, and bottom store shelves differ in their nutritional profiles.

9.8 ANOVA as regression

In section 9.2, we noted that ANOVA and regression are both special cases of the same general linear model. As such, the designs in this chapter could have been analyzed using the `lm()` function. But in order to understand the output, you need to understand how R deals with categorical variables when fitting models.

Consider the one-way ANOVA problem in section 9.3, which compares the impact of five cholesterol-reducing drug regimens (`trt`):

```
> library(multcomp)
> levels(cholesterol$trt)

[1] "1time" "2times" "4times" "drugD" "drugE"
```

First, let's fit the model using the `aov()` function:

```
> fit.aov <- aov(response ~ trt, data=cholesterol)
> summary(fit.aov)

      Df Sum Sq Mean Sq F value Pr(>F)
trt    4 1351.37  337.84  32.433 9.819e-13 ***
Residuals 45  468.75  10.42
```

Now, let's fit the same model using `lm()`. In this case, you get the results shown in the next listing.

Listing 9.11 A regression approach to the ANOVA problem in section 9.3

```
> fit.lm <- lm(response ~ trt, data=cholesterol)
> summary(fit.lm)
```

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	5.782	1.021	5.665	9.78e-07 ***
trt2times	3.443	1.443	2.385	0.0213 *
trt4times	6.593	1.443	4.568	3.82e-05 ***
trtdrugD	9.579	1.443	6.637	3.53e-08 ***
trtdrugE	15.166	1.443	10.507	1.08e-13 ***

Residual standard error: 3.227 on 45 degrees of freedom
 Multiple R-squared: 0.7425, Adjusted R-squared: 0.7196
 F-statistic: 32.43 on 4 and 45 DF, p-value: 9.819e-13

What are you looking at? Because linear models require numeric predictors, when the `lm()` function encounters a factor, it replaces that factor with a set of numeric variables representing contrasts among the levels. If the factor has k levels, $k - 1$ contrast variables are created. R provides five built-in methods for creating these contrast variables (see table 9.6). You can also create your own (we won't cover that here). By default, treatment contrasts are used for unordered factors, and orthogonal polynomials are used for ordered factors.

Table 9.6 Built-in contrasts

Contrast	Description
contr.helmert	Contrasts the second level with the first, the third level with the average of the first two, the fourth level with the average of the first three, and so on.
contr.poly	Contrasts are used for trend analysis (linear, quadratic, cubic, and so on) based on orthogonal polynomials. Use for ordered factors with equally spaced levels.
contr.sum	Contrasts are constrained to sum to zero. Also called <i>deviation contrasts</i> , they compare the mean of each level to the overall mean across levels.
contr.treatment	Contrasts each level with the baseline level (first level by default). Also called <i>dummy coding</i> .
contr.SAS	Similar to <code>contr.treatment</code> , but the baseline level is the last level. This produces coefficients similar to contrasts used in most SAS procedures.

With treatment contrasts, the first level of the factor becomes the reference group, and each subsequent level is compared with it. You can see the coding scheme via the `contrasts()` function:

```
> contrasts(cholesterol$trt)
      2times 4times drugD drugE
1time    0    0    0    0
2times    1    0    0    0
4times    0    1    0    0
drugD     0    0    1    0
drugE     0    0    0    1
```

If a patient is in the drugD condition, then the variable drugD equals 1, and the variables 2times, 4times, and drugE each equal zero. You don't need a variable for the first group, because a zero on each of the four indicator variables uniquely determines that the patient is in the 1times condition.

In listing 9.11, the variable `trt2times` represents a contrast between the levels 1time and 2time. Similarly, `trt4times` is a contrast between 1time and 4times, and so on. You can see from the probability values in the output that each drug condition is significantly different from the first (1time).

You can change the default contrasts used in `lm()` by specifying a `contrasts` option. For example, you can specify Helmert contrasts by using

```
fit.lm <- lm(response ~ trt, data=cholesterol, contrasts="contr.helmert")
```

You can change the default contrasts used during an R session via the `options()` function. For example,

```
options(contrasts = c("contr.SAS", "contr.helmert"))
```

would set the default contrast for unordered factors to `contr.SAS` and for ordered factors to `contr.helmert`. Although we've limited our discussion to the use of contrasts in linear models, note that they're applicable to other modeling functions in R. This includes the generalized linear models covered in chapter 13.

9.9 Summary

- Analysis of variance (ANOVA) is a set of statistical methods frequently used when analyzing data from experimental and quasi-experimental research.
- ANOVA methodologies are particularly helpful when investigating the relationship between a quantitative outcome variable and one or more categorical explanatory variables.
- If a quantitative outcome variable is related to a categorical explanatory variable with *more than* two levels, post hoc tests are conducted to identify which levels/groups differ on that outcome.
- When there are two or more categorical explanatory variables, a factorial ANOVA can

be used to study their unique and joint effects on the outcome variable.

- When the effects of one or more quantitative nuisance variables are statistically controlled (removed), the design is called an analysis of covariance (ANCOVA).
- When there is more than one outcome variable, the design is called a multivariate analysis of variance or covariance.
- ANOVA and multiple regression are two equivalent expressions of the general linear model. The different terminologies, R functions, and output formats for these two approaches reflect their separate origins in different fields of research. When studies focus on group differences, ANOVA results are often easier to understand and communicate to others.

10

Power analysis

This chapter covers

- Determining sample size requirements
- Calculating effect sizes
- Assessing statistical power

As a statistical consultant, I'm often asked, "How many subjects do I need for my study?" Sometimes the question is phrased this way: "I have x number of people available for this study. Is the study worth doing?" Questions like these can be answered through *power analysis*, an important set of techniques in experimental design.

Power analysis allows you to determine the sample size required to detect an effect of a given size with a given degree of confidence. Conversely, it allows you to determine the probability of detecting an effect of a given size with a given level of confidence, under sample size constraints. If the probability is unacceptably low, you'd be wise to alter or abandon the experiment.

In this chapter, you'll learn how to conduct power analyses for a variety of statistical tests, including tests of proportions, t-tests, chi-square tests, balanced one-way ANOVA, tests of correlations, and linear models. Because power analysis applies to hypothesis testing situations, we'll start with a brief review of null hypothesis significance testing (NHST). Then we'll review conducting power analyses within R, focusing primarily on the `pwr` package. Finally, we'll consider other approaches to power analysis available with R.

10.1 A quick review of hypothesis testing

To help you understand the steps in a power analysis, we'll briefly review statistical hypothesis testing in general. If you have a statistical background, feel free to skip to section 10.2.

In statistical hypothesis testing, you specify a hypothesis about a population parameter (your *null hypothesis*, or H_0). You then draw a sample from this population and calculate a statistic that's used to make inferences about the population parameter. Assuming that the null hypothesis is true, you calculate the probability of obtaining the observed sample statistic or one more extreme. If the probability is sufficiently small, you reject the null hypothesis in favor of its opposite (referred to as the *alternative* or *research hypothesis*, H_1).

An example will clarify the process. Say you're interested in evaluating the impact of cell phone use on driver reaction time. Your null hypothesis is $H_0: \mu_1 - \mu_2 = 0$, where μ_1 is the mean response time for drivers using a cell phone and μ_2 is the mean response time for drivers that are cell phone free (here, $\mu_1 - \mu_2$ is the population parameter of interest). If you reject this null hypothesis, you're left with the alternate or research hypothesis, namely $H_1: \mu_1 - \mu_2 \neq 0$. This is equivalent to $\mu_1 \neq \mu_2$, that the mean reaction times for the two conditions are not equal.

A sample of individuals is selected and randomly assigned to one of two conditions. In the first condition, participants react to a series of driving challenges in a simulator while talking on a cell phone. In the second condition, participants complete the same series of challenges but without a cell phone. Overall reaction time is assessed for each individual.

Based on the sample data, you can calculate the statistic $(\bar{x}_1 - \bar{x}_2) / (s / \sqrt{n})$, where \bar{x}_1 and \bar{x}_2 are the sample reaction time means in the two conditions, s is the pooled sample standard deviation, and n is the number of participants in each condition. If the null hypothesis is true and you can assume that reaction times are normally distributed, this sample statistic will follow a t distribution with $2n - 2$ degrees of freedom. Using this fact, you can calculate the probability of obtaining a sample statistic this large or larger. If the probability (p) is smaller than some predetermined cutoff (say $p < .05$), you reject the null hypothesis in favor of the alternate hypothesis. This predetermined cutoff (0.05) is called the *significance level* of the test.

Note that you use *sample* data to make an inference about the *population* it's drawn from. Your null hypothesis is that the mean reaction time of *all* drivers talking on cell phones isn't different from the mean reaction time of *all* drivers who aren't talking on cell phones, not just those drivers in your sample. The four possible outcomes from your decision are as follows:

- If the null hypothesis is false and the statistical test leads you to reject it, you've made a correct decision. You've correctly determined that reaction time is affected by cell phone use.
- If the null hypothesis is true and you don't reject it, again you've made a correct decision. Reaction time isn't affected by cell phone use.
- If the null hypothesis is true but you reject it, you've committed a Type I error. You've concluded that cell phone use affects reaction time when it doesn't.
- If the null hypothesis is false and you fail to reject it, you've committed a Type II error. Cell phone use affects reaction time, but you've failed to discern this.

Each of these outcomes is illustrated in the following table:

		Decision	
		Reject H_0	Fail to Reject H_0
Actual	H_0 true	Type I error	correct
	H_0 false	correct	Type II error

Controversy surrounding null hypothesis significance testing

Null hypothesis significance testing isn't without controversy and detractors have raised numerous concerns about the approach, particularly as practiced in the field of psychology. They point to a widespread misunderstanding of p values, reliance on statistical significance over practical significance, the fact that the null hypothesis is never exactly true and will always be rejected for sufficient sample sizes, and a number of logical inconsistencies in NHST practices.

An in-depth discussion of this topic is beyond the scope of this book. Interested readers are referred to Harlow, Mulaik, and Steiger (1997).

In planning research, the researcher typically pays special attention to four quantities (see figure 10.1):

- *Sample size* refers to the number of observations in each condition/group of the experimental design.
- The *significance level* (also referred to as *alpha*) is defined as the probability of making a Type I error. The significance level can also be thought of as the probability of finding an effect that is *not* there.
- *Power* is defined as one minus the probability of making a Type II error. Power can be thought of as the probability of finding an effect that *is* there.
- *Effect size* is the magnitude of the effect under the alternate or research hypothesis. The formula for effect size depends on the statistical methodology employed in the hypothesis testing.

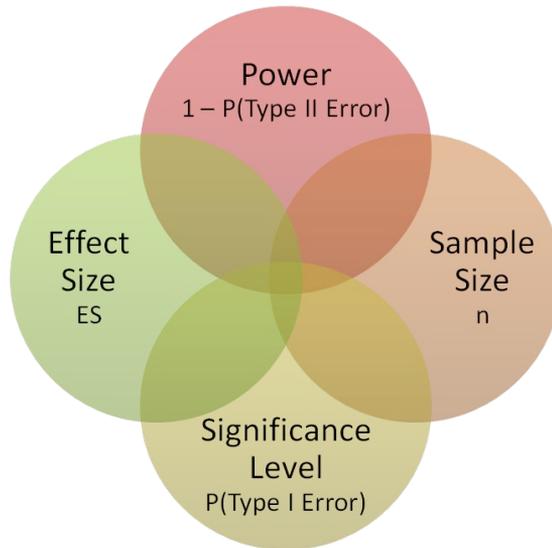


Figure 10.1. Four primary quantities considered in a study design power analysis. Given any three, you can calculate the fourth.

Although the sample size and significance level are under the direct control of the researcher, power and effect size are affected more indirectly. For example, as you relax the significance level (in other words, make it easier to reject the null hypothesis), power increases. Similarly, increasing the sample size increases power.

Your research goal is typically to maximize the power of your statistical tests while maintaining an acceptable significance level and employing as small a sample size as possible. That is, you want to maximize the chances of finding a real effect and minimize the chances of finding an effect that isn't really there, while keeping study costs within reason.

The four quantities (sample size, significance level, power, and effect size) have an intimate relationship. *Given any three, you can determine the fourth.* You'll use this fact to carry out various power analyses throughout the remainder of the chapter. In the next section, we'll look at ways of implementing power analyses using the R package `pwr`. Later, we'll briefly look at some highly specialized power functions that are used in biology and genetics.

10.2 Implementing power analysis with the `pwr` package

The `pwr` package, developed by Stéphane Champely, implements power analysis as outlined by Cohen (1988). Some of the more important functions are listed in table 10.1. For each function, the user can specify three of the four quantities (sample size, significance level, power, effect size), and the fourth will be calculated.

Table 10.1 `pwr` package functions

Function	Power calculations for ...
<code>pwr.2p.test</code>	Two proportions (equal n)
<code>pwr.2p2n.test</code>	Two proportions (unequal n)
<code>pwr.anova.test</code>	Balanced one way ANOVA
<code>pwr.chisq.test</code>	Chi-square test
<code>pwr.f2.test</code>	General linear model
<code>pwr.p.test</code>	Proportion (one sample)
<code>pwr.r.test</code>	Correlation
<code>pwr.t.test</code>	t-tests (one sample, two samples, paired)
<code>pwr.t2n.test</code>	t-test (two samples with unequal n)

Of the four quantities, effect size is often the most difficult to specify. Calculating effect size typically requires some experience with the measures involved and knowledge of past research. But what can you do if you have no clue what effect size to expect in a given study? You'll look at this difficult question in section 10.2.7. In the remainder of this section, you'll look at the application of `pwr` functions to common statistical tests. Before invoking these functions, be sure to install and load the `pwr` package.

10.2.1 t-tests

When the statistical test to be used is a t-test, the `pwr.t.test()` function provides a number of useful power analysis options. The format is

```
pwr.t.test(n=, d=, sig.level=, power=, alternative=)
```

where

- `n` is the sample size.
- `d` is the effect size defined as the standardized mean difference.

$$d = \frac{|\mu_1 - \mu_2|}{\sigma}$$

μ_1 = mean of group 1
 μ_2 = mean of group 2
 σ^2 = common error variance

- `sig.level` is the significance level (0.05 is the default).
- `power` is the power level.
- `type` is two-sample t-test ("`two.sample`"), a one-sample t-test ("`one.sample`"), or a dependent sample t-test ("`paired`"). A two-sample test is the default.
- `alternative` indicates whether the statistical test is two-sided ("`two.sided`") or one-sided ("`less`" or "`greater`"). A two-sided test is the default.

Let's work through an example. Continuing the experiment from section 10.1 involving cell phone use and driving reaction time, assume that you'll be using a two-tailed independent sample t-test to compare the mean reaction time for participants in the cell phone condition with the mean reaction time for participants driving unencumbered.

Let's assume that you know from past experience that reaction time has a standard deviation of 1.25 seconds. Also suppose that a 1-second difference in reaction time is considered an important difference. You'd therefore like to conduct a study in which you're able to detect an effect size of $d = 1/1.25 = 0.8$ or larger. Additionally, you want to be 90% sure to detect such a difference if it exists, and 95% sure that you won't declare a difference to be significant when it's actually due to random variability. How many participants will you need in your study?

Entering this information in the `pwr.t.test()` function, you have the following:

```
> library(pwr)
> pwr.t.test(d=.8, sig.level=.05, power=.9, type="two.sample",
  alternative="two.sided")
```

Two-sample t test power calculation

```
  n = 34
  d = 0.8
sig.level = 0.05
power = 0.9
alternative = two.sided
```

NOTE: n is number in *each* group

The results suggest that you need 34 participants in each group (for a total of 68 participants) in order to detect an effect size of 0.8 with 90% certainty and no more than a 5% chance of erroneously concluding that a difference exists when, in fact, it doesn't.

Let's alter the question. Assume that in comparing the two conditions you want to be able to detect a 0.5 standard deviation difference in population means. You want to limit the chances of falsely declaring the population means to be different to 1 out of 100. Additionally, you can only afford to include 40 participants in the study. What's the probability that you'll be able to detect a difference between the population means that's this large, given the constraints outlined?

Assuming that an equal number of participants will be placed in each condition, you have

```
> pwr.t.test(n=20, d=.5, sig.level=.01, type="two.sample",
  alternative="two.sided")
```

Two-sample t test power calculation

```
  n = 20
  d = 0.5
sig.level = 0.01
power = 0.14
```

```
alternative = two.sided
```

```
NOTE: n is number in *each* group
```

With 20 participants in each group, an a priori significance level of 0.01, and a dependent variable standard deviation of 1.25 seconds, you have less than a 14% chance of declaring a difference of 0.625 seconds or less significant ($d = 0.5 = 0.625/1.25$). Conversely, there's an 86% chance that you'll miss the effect that you're looking for. You may want to seriously rethink putting the time and effort into the study as it stands.

The previous examples assumed that there are equal sample sizes in the two groups. If the sample sizes for the two groups are unequal, the function

```
pwr.t2n.test(n1=, n2=, d=, sig.level=, power=, alternative=)
```

can be used. Here, `n1` and `n2` are the sample sizes, and the other parameters are the same as for `pwr.t.test`. Try varying the values input to the `pwr.t2n.test` function and see the effect on the output.

10.2.2 ANOVA

The `pwr.anova.test()` function provides power analysis options for a balanced one-way analysis of variance. The format is

```
pwr.anova.test(k=, n=, f=, sig.level=, power=)
```

where `k` is the number of groups and `n` is the common sample size in each group.

For a one-way ANOVA, effect size is measured by f , where

$$f = \sqrt{\frac{\sum_{i=1}^k p_i \times (\mu_i - \mu)^2}{\sigma^2}}$$

$p_i = n_i / N$
 $n_i =$ number of observations in group i
 $N =$ total number of observations
 $\mu_i =$ mean of group i
 $\mu =$ grand mean
 $\sigma^2 =$ error variance within groups

Let's try an example. For a one-way ANOVA comparing five groups, calculate the sample size needed in each group to obtain a power of 0.80, when the effect size is 0.25 and a significance level of 0.05 is employed. The code looks like this:

```
> pwr.anova.test(k=5, f=.25, sig.level=.05, power=.8)
```

```
Balanced one-way analysis of variance power calculation
```

```
  k = 5
```

```
n = 39
f = 0.25
sig.level = 0.05
power = 0.8
```

NOTE: n is number in each group

The total sample size is therefore 5×39 , or 195. Note that this example requires you to estimate what the means of the five groups will be, along with the common variance. When you have no idea what to expect, the approaches described in section 10.2.7 may help.

10.2.3 Correlations

The `pwr.r.test()` function provides a power analysis for tests of correlation coefficients. The format is as follows

```
pwr.r.test(n=, r=, sig.level=, power=, alternative=)
```

where `n` is the number of observations, `r` is the effect size (as measured by a linear correlation coefficient), `sig.level` is the significance level, `power` is the power level, and `alternative` specifies a two-sided ("`two.sided`") or a one-sided ("`less`" or "`greater`") significance test.

For example, let's assume that you're studying the relationship between depression and loneliness. Your null and research hypotheses are

$H_0: \rho \leq 0.25$ versus $H_1: \rho > 0.25$

where ρ is the population correlation between these two psychological variables. You've set your significance level to 0.05, and you want to be 90% confident that you'll reject H_0 if it's false. How many observations will you need? This code provides the answer:

```
> pwr.r.test(r=.25, sig.level=.05, power=.90, alternative="greater")
```

```
approximate correlation power calculation (arctangh transformation)
```

```
n = 134
r = 0.25
sig.level = 0.05
power = 0.9
alternative = greater
```

Thus, you need to assess depression and loneliness in 134 participants in order to be 90% confident that you'll reject the null hypothesis if it's false.

10.2.4 Linear models

For linear models (such as multiple regression), the `pwr.f2.test()` function can be used to carry out a power analysis. The format is

```
pwr.f2.test(u=, v=, f2=, sig.level=, power=)
```

where u and v are the numerator and denominator degrees of freedom and f^2 is the effect size.

$$f^2 = \frac{R^2}{1 - R^2} \quad \text{where } R^2 = \text{population multiple correlation}$$

$$f^2 = \frac{R_{AB}^2 - R_A^2}{1 - R_{AB}^2} \quad \text{where } R_A^2 = \text{variance accounted for in the population by variable set A}$$

$R_{AB}^2 = \text{variance accounted for in the population by variable set A and B together}$

The first formula for f^2 is appropriate when you're evaluating the impact of a set of predictors on an outcome. The second formula is appropriate when you're evaluating the impact of one set of predictors above and beyond a second set of predictors (or covariates).

Let's say you're interested in whether a boss's leadership style impacts workers' satisfaction above and beyond the salary and perks associated with the job. Leadership style is assessed by four variables, and salary and perks are associated with three variables. Past experience suggests that salary and perks account for roughly 30% of the variance in worker satisfaction. From a practical standpoint, it would be interesting if leadership style accounted for at least 5% above this figure. Assuming a significance level of 0.05, how many subjects would be needed to identify such a contribution with 90% confidence?

Here, `sig.level=0.05`, `power=0.90`, `u=3` (total number of predictors minus the number of predictors in set B), and the effect size is $f^2 = (.35 - .30)/(1 - .35) = 0.0769$. Entering this into the function yields the following:

```
> pwr.f2.test(u=3, f2=0.0769, sig.level=0.05, power=0.90)
```

Multiple regression power calculation

```
u = 3
v = 184.2426
f2 = 0.0769
sig.level = 0.05
power = 0.9
```

In multiple regression, the denominator degrees of freedom equals $N - k - 1$, where N is the number of observations and k is the number of predictors. In this case, $N - 7 - 1 = 185$, which means the required sample size is $N = 185 + 7 + 1 = 193$.

10.2.5 Tests of proportions

The `pwr.2p.test()` function can be used to perform a power analysis when comparing two proportions. The format is

```
pwr.2p.test(h=, n=, sig.level=, power=)
```

where `h` is the effect size and `n` is the common sample size in each group. The effect size `h` is defined as

$$h = 2 \arcsin\left(\sqrt{p_1}\right) - 2 \arcsin\left(\sqrt{p_2}\right)$$

and can be calculated with the function `ES.h(p1, p2)`.

For unequal `ns`, the desired function is

```
pwr.2p2n.test(h=, n1=, n2=, sig.level=, power=)
```

The `alternative=` option can be used to specify a two-tailed ("`two.sided`") or one-tailed ("`less`" or "`greater`") test. A two-tailed test is the default.

Let's say that you suspect that a popular medication relieves symptoms in 60% of users. A new (and more expensive) medication will be marketed if it improves symptoms in 65% of users. How many participants will you need to include in a study comparing these two medications if you want to detect a difference this large?

Assume that you want to be 90% confident in a conclusion that the new drug is better and 95% confident that you won't reach this conclusion erroneously. You'll use a one-tailed test because you're only interested in assessing whether the new drug is better than the standard. The code looks like this:

```
> pwr.2p.test(h=ES.h(.65, .6), sig.level=.05, power=.9,
  alternative="greater")
```

```
Difference of proportion power calculation for binomial
distribution (arcsine transformation)
```

```
h = 0.1033347
n = 1604.007
sig.level = 0.05
power = 0.9
alternative = greater
```

NOTE: same sample sizes

Based on these results, you'll need to conduct a study with 1,605 individuals receiving the new drug and 1,605 receiving the existing drug in order to meet the criteria.

10.2.6 Chi-square tests

Chi-square tests are often used to assess the relationship between two categorical variables. The null hypothesis is typically that the variables are independent versus a research hypothesis that they aren't. The `pwr.chisq.test()` function can be used to evaluate the power, effect size, or requisite sample size when employing a chi-square test. The format is

```
pwr.chisq.test(w = , N = , df = , sig.level = , power = )
```

where w is the effect size, N is the total sample size, and df is the degrees of freedom. Here, effect size w is defined as

$$w = \sqrt{\sum_{i=1}^m \frac{(p_{0i} - p_{1i})^2}{p_{0i}}} \quad \text{where } \begin{array}{l} p_{0i} = \text{cell probability in the } i\text{th cell under } H_0 \\ p_{1i} = \text{cell probability in the } i\text{th cell under } H_1 \end{array}$$

The summation goes from 1 to m , where m is the number of cells in the contingency table. The function `ES.w2(P)` can be used to calculate the effect size corresponding the alternative hypothesis in a two-way contingency table. Here, P is a hypothesized two-way probability table.

As a simple example, let's assume that you're looking the relationship between ethnicity and promotion. You anticipate that 70% of your sample will be Caucasian, 10% will be African American, and 20% will be Hispanic. Further, you believe that 60% of Caucasians tend to be promoted, compared with 30% for African Americans and 50% for Hispanics. Your research hypothesis is that the probability of promotion follows the values in table 10.2.

Table 10.2 Proportion of individuals expected to be promoted based on the research hypothesis

Ethnicity	Promoted	Not Promoted
Caucasian	0.42	0.28
African American	0.03	0.07
Hispanic	0.10	0.10

For example, you expect that 42% of the population will be promoted Caucasians ($.42 = .70 \times .60$) and 7% of the population will be nonpromoted African Americans ($.07 = .10 \times .70$). Let's assume a significance level of 0.05 and that the desired power level is 0.90. The degrees of freedom in a two-way contingency table are $(r - 1) \times (c - 1)$, where r is the number of rows and c is the number of columns. You can calculate the hypothesized effect size with the following code:

```
> prob <- matrix(c(.42, .28, .03, .07, .10, .10), byrow=TRUE, nrow=3)
> ES.w2(prob)
[1] 0.1853198
```

Using this information, you can calculate the necessary sample size like this:

```
> pwr.chisq.test(w=.1853, df=2, sig.level=.05, power=.9)
```

Chi squared power calculation

```
w = 0.1853
N = 368.5317
df = 2
sig.level = 0.05
power = 0.9
```

NOTE: N is the number of observations

The results suggest that a study with 369 participants will be adequate to detect a relationship between ethnicity and promotion given the effect size, power, and significance level specified.

10.2.7 Choosing an appropriate effect size in novel situations

In power analysis, the expected effect size is the most difficult parameter to determine. It typically requires that you have experience with the subject matter and the measures employed. For example, the data from past studies can be used to calculate effect sizes, which can then be used to plan future studies.

But what can you do when the research situation is completely novel and you have no past experience to call upon? In the area of behavioral sciences, Cohen (1988) attempted to provide benchmarks for “small,” “medium,” and “large” effect sizes for various statistical tests. These guidelines are provided in table 10.3.

Table 10.3 Cohen’s effect size benchmarks

Statistical method	Effect size measures	Suggested guidelines for effect size		
		<i>Small</i>	<i>Medium</i>	<i>Large</i>
t-test	d	0.20	0.50	0.80
ANOVA	f	0.10	0.25	0.40
Linear models	f ²	0.02	0.15	0.35
Test of proportions	h	0.20	0.50	0.80
Chi-square	w	0.10	0.30	0.50

When you have no idea what effect size may be present, this table may provide some guidance. For example, what’s the probability of rejecting a false null hypothesis (that is, finding a real effect) if you’re using a one-way ANOVA with 5 groups, 25 subjects per group, and a significance level of 0.05?

Using the `pwr.anova.test()` function and the suggestions in the `f` row of table 10.3, the power would be 0.118 for detecting a small effect, 0.574 for detecting a moderate effect, and 0.957 for detecting a large effect. Given the sample size limitations, you're only likely to find an effect if it's large.

It's important to keep in mind that Cohen's benchmarks are just general suggestions derived from a range of social research studies and may not apply to your particular field of research. An alternative is to vary the study parameters and note the impact on such things as sample size and power. For example, again assume that you want to compare five groups using a one-way ANOVA and a 0.05 significance level. The following listing computes the sample sizes needed to detect a range of effect sizes and plots the results in figure 10.2.

Listing 10.1 Sample sizes for detecting significant effects in a one-way ANOVA

```
library(pwr)
es <- seq(.1, .5, .01)
nes <- length(es)

samsize <- NULL
for (i in 1:nes){
  result <- pwr.anova.test(k=5, f=es[i], sig.level=.05, power=.9)
  samsize[i] <- ceiling(result$n)
}

plotdata <- data.frame(es, samsize)
library(ggplot2)
ggplot(plotdata, aes(x=samsize, y=es)) +
  geom_line(color="red", size=1) +
  theme_bw() +
  labs(title="One Way ANOVA (5 groups)",
       subtitle="Power = 0.90, Alpha = 0.05",
       x="Sample Size (per group)",
       y="Effect Size")
```

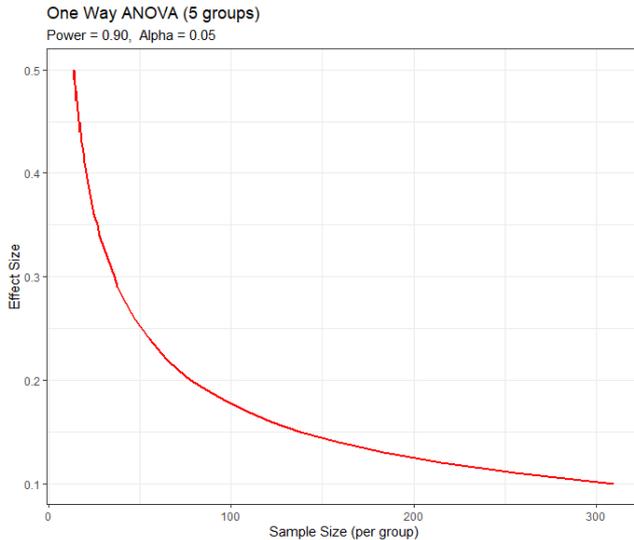


Figure 10.2 Sample size needed to detect various effect sizes in a one-way ANOVA with five groups (assuming a power of 0.90 and significance level of 0.05)

Graphs such as these can help you estimate the impact of various conditions on your experimental design. For example, there appears to be little bang for the buck in increasing the sample size above 200 observations per group. We'll look at another plotting example in the next section.

10.3 Creating power analysis plots

Before leaving the `pwr` package, let's look at a more involved graphing example. Suppose you'd like to see the sample size necessary to declare a correlation coefficient statistically significant for a range of effect sizes and power levels. You can use the `pwr.r.test()` function and `for` loops to accomplish this task, as shown in the following listing.

Listing 10.2 Sample size curves for detecting correlations of various sizes

```
library(pwr)
r <- seq(.1,.5,.01)          #1
p <- seq(.4,.9,.1)

df <- expand.grid(r, p)
colnames(df) <- c("r", "p")

for (i in 1:nrow(df)){      #2
  result <- pwr.r.test(r = df$r[i],
                      sig.level = .05, power = df$p[i],
                      alternative = "two.sided")
  df$n[i] <- ceiling(result$n)
```

```

}
library(ggplot2) #3
ggplot(data=df,
  aes(x=r, y=n, color=factor(p))) +
  geom_line(size=1) +
  theme_bw() +
  labs(title="Sample Size Estimation for Correlation Studies",
    subtitle="Sig=0.05 (Two-tailed)",
    x="Correlation Coefficient (r)",
    y="Samsple Size (n)",
    color="Power")

```

- 1 Sets the range of correlations and power values
- 2 Obtains sample sizes
- 3 Plot power curves

Listing 10.2 uses the `seq()` function to generate a range of effect sizes r (correlation coefficients under H_1) and power levels p #1. The `expand.grid()` function is used to create a data frame with every combination of these two variables. A `for` loop then cycles through rows of the data frame, calculating the sample size (n) for that row's correlation and power level, and saving the result #2. The `ggplot2` package is then used to plot a sample size vs. correlation curve for each power level #3. The resulting graph is displayed in figure 10.3.

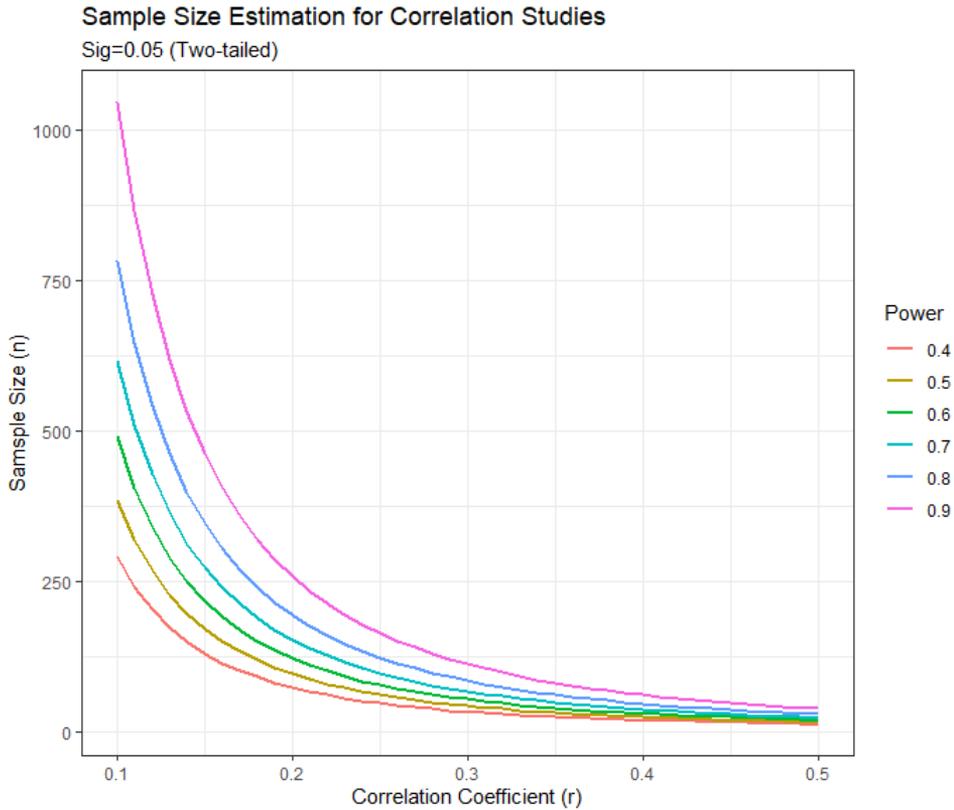


Figure 10.3 Sample size curves for detecting a significant correlation at various power levels

As you can see from the graph, you'd need a sample size of approximately 75 to detect a correlation of 0.20 with 40% confidence. You'd need approximately 185 additional observations ($n = 260$) to detect the same correlation with 90% confidence. With simple modifications, the same approach can be used to create sample size and power curve graphs for a wide range of statistical tests.

We'll close this chapter by briefly looking at other R functions that are useful for power analysis.

10.4 Other packages

There are many other packages in R that can be useful in the planning stages of studies. Several are listed in table 10.4. Some contain general tools, whereas some are highly specialized. The last four in the table are particularly focused on power analysis in genetic studies. Genome-wide association studies (GWAS) are studies used to identify genetic

associations with observable traits. For example, these studies would focus on why some people get a specific type of heart disease.

Table 10.4 Specialized power analysis packages

Package	Purpose
asypow	Power calculations via asymptotic likelihood ratio methods
longpower	Sample-size calculations for longitudinal data
PwrGSD	Power analysis for group sequential designs
pamm	Power analysis for random effects in mixed models
powerSurvEpi	Power and sample-size calculations for survival analysis in epidemiological studies
powerMediation	Power and sample-size calculations for mediation effects in linear, logistic, Poisson, and cox regression
semPower	Power analyses for structural equation models (SEM)
powerpkg	Power analyses for the affected sib pair and the TDT (transmission disequilibrium test) design
powerGWASinteraction	Power calculations for interactions for GWAS
gap	Functions for power and sample-size calculations in case-cohort designs
ssize.fdr	Sample-size calculations for microarray experiments

Finally, the MBESS and `WebPower` packages contains a wide range of functions that can be used for various forms of power analysis and sample size determination. The functions are particularly relevant for researchers in the behavioral, educational, and social sciences.

10.5 Summary

- In this chapter, we focused on the planning stages of such research. Power analysis

helps you to determine the sample sizes needed to discern an effect of a given size with a given degree of confidence. It can also tell you the probability of detecting such an effect for a given sample size. You can directly see the tradeoff between limiting the likelihood of wrongly declaring an effect significant (a Type I error) with the likelihood of rightly identifying a real effect (power).

- The bulk of this chapter has focused on the use of functions provided by the `pwr` package. These functions can be used to carry out power and sample-size determinations for common statistical methods (including t-tests, chi-square tests, and tests of proportions, ANOVA, and regression). Pointers to more specialized methods were provided in the final section.
- Power analysis is typically an interactive process. The investigator varies the parameters of sample size, effect size, desired significance level, and desired power to observe their impact on each other. The results are used to plan studies that are more likely to yield meaningful results. Information from past research (particularly regarding effect sizes) can be used to design more effective and efficient future research.
- An important side benefit of power analysis is the shift that it encourages, away from a singular focus on binary hypothesis testing (that is, does an effect exist or not), toward an appreciation of the *size* of the effect under consideration. Journal editors are increasingly requiring authors to include effect sizes as well as p values when reporting research results. This helps you to determine both the practical implications of the research and provides you with information that can be used to plan future studies.

11

Intermediate graphs

This chapter covers

- Visualizing bivariate and multivariate relationships
- Working with scatter and line plots
- Understanding corrgrams
- Using mosaic and association plots

In chapter 6 (basic graphs), we considered a wide range of graph types for displaying the distribution of single categorical or continuous variables. Chapter 8 (regression) reviewed graphical methods that are useful when predicting a continuous outcome variable from a set of predictor variables. In chapter 9 (analysis of variance), we considered techniques that are particularly useful for visualizing how groups differ on a continuous outcome variable. In many ways, the current chapter is a continuation and extension of the topics covered so far.

In this chapter, we'll focus on graphical methods for displaying relationships between two variables (bivariate relationships) and between many variables (multivariate relationships). For example:

- What's the relationship between automobile mileage and car weight? Does it vary by the number of cylinders the car has?
- How can you picture the relationships among an automobile's mileage, weight, displacement, and rear axle ratio in a single graph?
- When plotting the relationship between two variables drawn from a large dataset (say, 10,000 observations), how can you deal with the massive overlap of data points you're likely to see? In other words, what do you do when your graph is one big smudge?
- How can you visualize the multivariate relationships among three variables at once (given a 2D computer screen or sheet of paper, and a budget slightly less than the latest *Star Wars* movie)?

- How can you display the growth of several trees over time?
- How can you visualize the correlations among a dozen variables in a single graph? How does it help you to understand the structure of your data?
- How can you visualize the relationship of class, gender, and age with passenger survival on the *Titanic*? What can you learn from such a graph?

These are the types of questions that can be answered with the methods described in this chapter. The datasets that we'll use are examples of what's possible. It's the general techniques that are most important. If the topic of automobile characteristics or tree growth isn't interesting to you, plug in your own data!

We'll start with scatter plots and scatter-plot matrices. Then, we'll explore line charts of various types. These approaches are well known and widely used in research. Next, we'll review the use of corrgrams for visualizing correlations and mosaic plots for visualizing multivariate relationships among categorical variables. These approaches are also useful but much less well known among researchers and data analysts. You'll see examples of how you can use each of these approaches to gain a better understanding of your data and communicate these findings to others.

11.1 Scatter plots

As you've seen in previous chapters, scatter plots describe the relationship between two continuous variables. In this section, we'll start with a depiction of a single bivariate relationship (x versus y). We'll then explore ways to enhance this plot by superimposing additional information. Next, you'll learn how to combine several scatter plots into a scatter-plot matrix so that you can view many bivariate relationships at once. We'll also review the special case where many data points overlap, limiting your ability to picture the data, and we'll discuss several ways around this difficulty. Finally, we'll extend the two-dimensional graph to three dimensions, with the addition of a third continuous variable. This will include 3D scatter plots and bubble plots. Each can help you understand the multivariate relationship among three variables at once.

We'll start by visualizing the relationship between automobile weight and fuel efficiency.

Listing 11.1 A scatter plot with best-fit lines

```
data(mtcars) #1
ggplot(mtcars, aes(x=wt, y=mpg)) + geom_point() #2
geom_smooth(method="lm", se=FALSE, color="red") + #3
geom_smooth(method="loess", se=FALSE, #4
  color="blue", linetype="dashed") +
  labs(title = "Basic Scatter Plot of MPG vs. Weight", #5
    x = "Car Weight (lbs/1000)",
    y = "Miles Per Gallon")
```

#1 Load data

#2 Create scatter plot

#3 Add linear fit

#4 Add loess fit
 #5 Add annotations

The resulting graph is provided in figure 11.1.

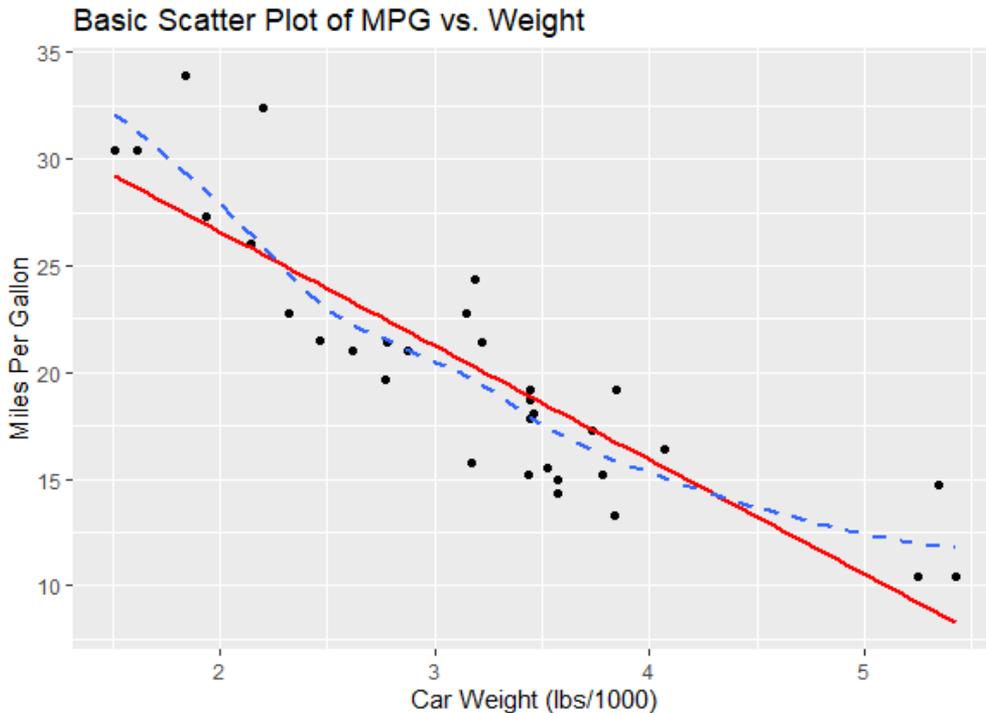


Figure 11.1 Scatter plot of car mileage vs. weight, with superimposed linear and loess fit lines

The code in listing 11.1 loads a fresh copy of the built-in data frame `mtcars` #1, and creates a basic scatter plot using filled circles for the plotting symbol #2. As expected, as car weight increases, miles per gallon decreases, although the relationship isn't perfectly linear. The first `geom_smooth()` function adds a linear fit line (solid red) #3. The `se=FALSE` option suppresses the 95% confidence interval for the line. The second `geom_smooth()` function adds a *loess* fit (dashed blue line) #4. The loess line is a nonparametric fit line based on locally weighted polynomial regression and provides a smoothed trend line for the data. See Cleveland (1981) for technical details on the algorithm. Josh Starmer provides a highly intuitive explanation of loess fit lines on YouTube (www.youtube.com/watch?v=Vf7oJ6z2LCc).

What if we want to look at the relationship between car weight and fuel efficiency separately for 4, 6, and 8-cylinder cars? This is easy to do with `ggplot2`, and a few simple modifications of the previous code. The graph is provided in figure 11.2.

Listing 11.2 A scatter plot with separate best-fit lines

```
ggplot(mtcars,
  aes(x=wt, y=mpg,
    color=factor(cyl),
    shape=factor(cyl))) +
  geom_point(size=2) +
  geom_smooth(method="lm", se=FALSE) +
  geom_smooth(method="loess", se=FALSE, linetype="dashed") +
  labs(title = "Scatter Plot of MPG vs. Weight",
    subtitle = "By Number of Cylinders",
    x = "Car Weight (lbs/1000)",
    y = "Miles Per Gallon",
    color = "Number of \nCylinders",
    shape = "Number of \nCylinders") +
  theme_bw()
```

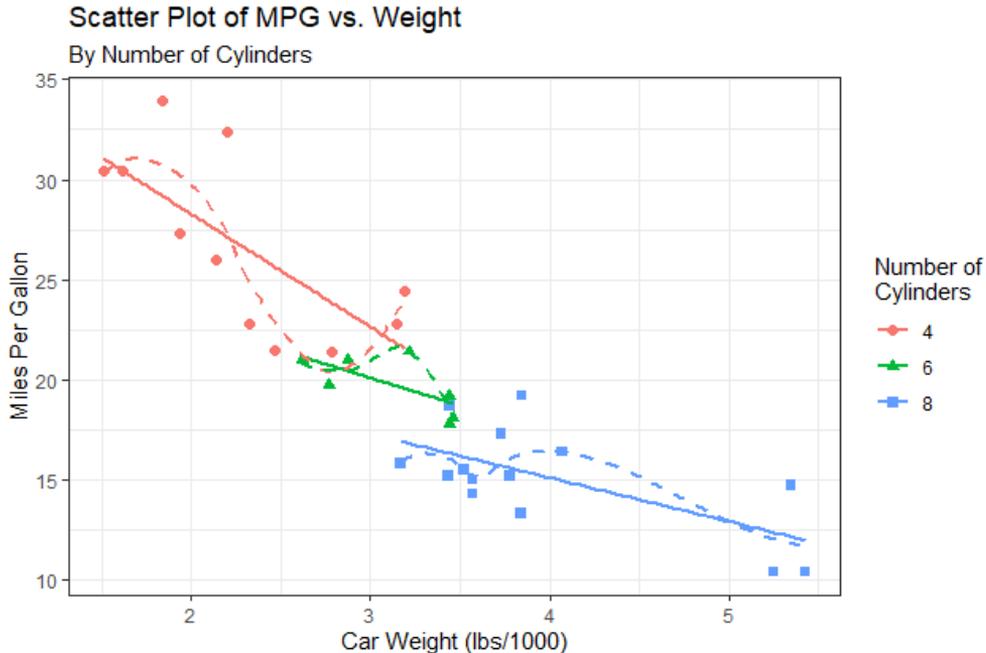


Figure 11.2 Scatter plot with subgroups and separately estimated fit lines.

By mapping the number of cylinders to color and shape in the `aes()` function, the three groups (4, 6, or 8 cylinders) are differentiated by color and plotting symbol, and separate linear and loess lines. Since the `cyl` variable is numeric, `factor(cyl)` is used to convert the variable into discrete categories.

You can control the smoothness of the loess lines using the `span` parameter. The default is `geom_smooth(method="loess", span=0.75)`. Larger values lead to smoother fits. In this

example, the loess lines overfit the data (follow the points too closely). A value of `span=4` (not shown) provides a much smoother fit.

Scatter plots help you visualize relationships between quantitative variables two at a time. But what if you wanted to look at the bivariate relationships between automobile mileage, weight, displacement (cubic inch), and rear axle ratio? When there are several quantitative variables, you can represent their relationships using a scatter-plot matrix, covered next.

11.1.1 Scatter-plot matrices

There are many useful functions for creating scatter-plot matrices in R. Base R provides the `pairs()` function for creating simple scatter-plot matrices. Section 8.2.4 (multiple linear regression) demonstrates the creation of scatter-plot matrices using the `scatterplotMatrix` function from the `car` package.

In this section, we'll use the `ggpairs()` function in the `GGally` package to create a `ggplot2` version of a scatter-plot matrix. As you'll see, this approach provides options for creating highly customized graphs. Be sure to install the `GGally` package (`install.packages("GGally")`) before proceeding.

First, let's create a default scatter-plot matrix for the `mpg`, `disp`, `drat`, and `wt` variables in the `mtcars` data frame.

```
library(GGally)
ggpairs(mtcars[c("mpg","disp","drat","wt")])
```

The graph is provided in figure 11.3.

By default, the principal diagonal of the matrix contains the kernel density curve for each variable (see section 6.5 for details). Miles per gallon is right skewed (there are a few high values) and the rear axle ratio appears to be bimodal. The six scatter plots are placed below the principal diagonal. The scatter plot between miles per gallon and engine displacement is found in at the intersection of these two variables (2th row, 1st column) and indicates a negative relationship. The Pearson correlation coefficients between each pair of variables is placed above the principal diagonal. The correlation between miles per gallon and engine displacement is -0.848 (1st row, 2nd column) and supports our conclusion that as engine displacement increases, gas mileage decreases.

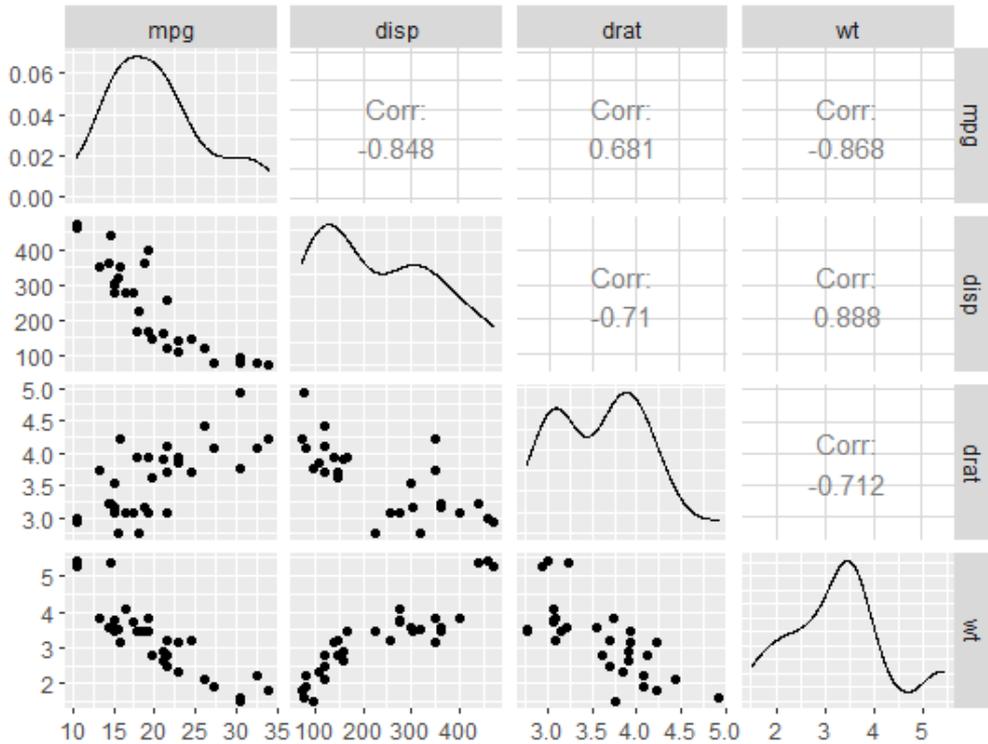


Figure 11.3 Scatter-plot matrix created by the `ggpairs()` function

Next, we'll create a highly customized scatter-plot matrix, adding fit lines, histograms, and a personalized theme. The `ggpairs()` function allows you specify separate functions for creating plots on, below, and above the principal diagonal. The code is provided in listing 11.3.

Listing 11.3 A scatter-plot matrix with fit lines, histograms, and correlation coefficients

```
library(GGally)

diagplots <- function(data, mapping) {                #1
  ggplot(data = data, mapping = mapping) +
  geom_histogram(fill="lightblue", color="black")
}

lowerplots <- function(data, mapping) {              #2
  ggplot(data = data, mapping = mapping) +
  geom_point(color="darkgrey") +
  geom_smooth(method = "lm", color = "steelblue", se=FALSE) +
  geom_smooth(method="loess", color="red", se=FALSE, linetype="dashed")
}
```

```

upperplots <- function(data, mapping) {           #3
  ggally_cor(data=data, mapping=mapping,
             displayGrid=FALSE, size=3.5, color="black")
}

mytheme <- theme(strip.background = element_blank(),      #4
                 panel.grid       = element_blank(),
                 panel.background = element_blank(),
                 panel.border     = element_rect(color="grey20", fill=NA))

ggpairs(mtcars,                                       #5
        columns=c("mpg", "disp", "drat", "wt"),
        columnLabels=c("MPG", "Displacement",
                       "R Axel Ratio", "Weight"),
        title = "Scatterplot Matrix with Linear and Loess Fits",
        lower = list(continuous = lowerplots),
        diag = list(continuous = diagplots),
        upper = list(continuous = upperplots)) +
mytheme

```

#1 Function for plots on principal diagonal

#2 Function for plots below diagonal

#3 Function for plots above diagonal

#4 Customized theme

#5 Generate scatter-plot matrix

First, a function is defined for creating a histogram using light blue bars with black borders #1. Next, a function is created for generating a scatter plot with dark grey points, a steel blue line of best fit, and a dashed red loess smoothed line. Confidence intervals are suppressed (`se=FALSE`) #2. A third function is specified for displaying correlation coefficients #3. This function uses the `ggally_cor()` function to obtain and print the coefficient, while the `size` and `color` option affect the appearance and the `displayGrid` option suppresses grid lines. A customize theme has also been added #4. This optional step eliminates facet strips and grid lines and surrounds each cell with a grey box.

Finally, the `ggpairs()` function uses these functions to create the customized graph seen in figure 11.4. The `columns` option specifies the variables, and the `columnLabels` option provides descriptive names. The `lower`, `diag`, and `upper` options specify the functions that will be used to create the cell plots in each portion of the matrix. This approach provides you with a great deal of flexibility in designing the finished graph.

Scatterplot Matrix with Linear and Loess Fits

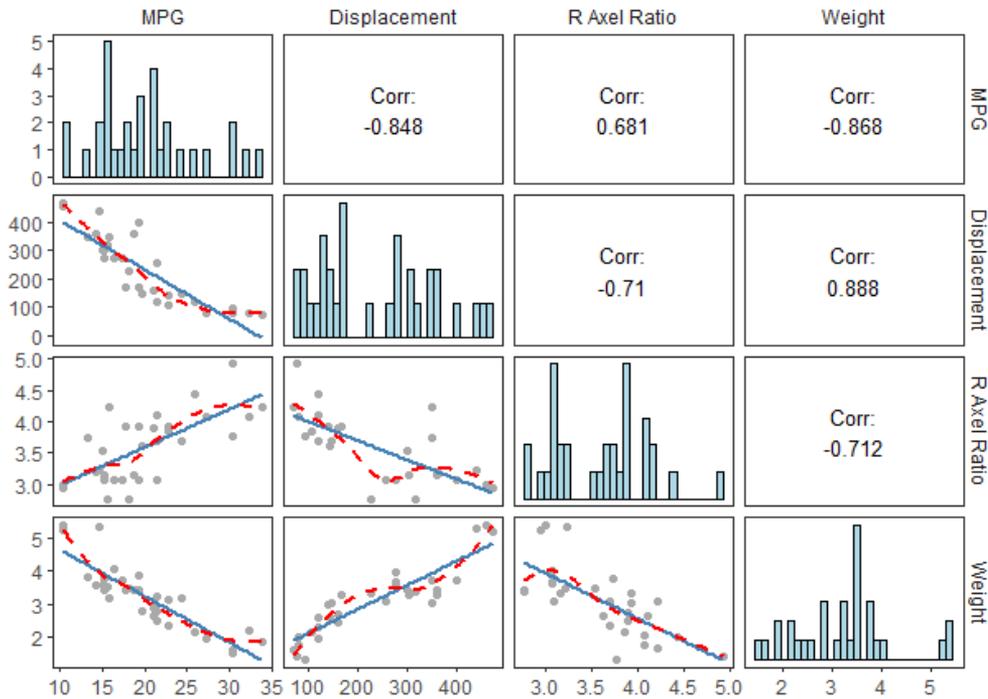


Figure 11.4 A scatter-plot matrix created with the `ggpairs()` function and user-supplied functions for the scatter-plots, histograms, and correlations.

R provides many other ways to create scatter-plot matrices. You may want to explore the `splom()` function in the `lattice` package, the `pairs2()` function in the `TeachingDemos` package, the `xysplom()` function in the `HH` package, the `kdepairs()` function in the `ResourceSelection` package, and `pairs.mod()` in the `SMPracticals` package. Each adds its own unique twist. Analysts must love scatter-plot matrices!

11.1.2 High-density scatter plots

When there's a significant overlap among data points, scatter plots become less useful for observing relationships. Consider the following contrived example with 10,000 observations falling into two overlapping clusters of data:

```
set.seed(1234)
n <- 10000
c1 <- matrix(rnorm(n, mean=0, sd=.5), ncol=2)
c2 <- matrix(rnorm(n, mean=3, sd=2), ncol=2)
mydata <- rbind(c1, c2)
```

```
mydata <- as.data.frame(mydata)
names(mydata) <- c("x", "y")
```

If you generate a standard scatter plot between these variables using the following code

```
ggplot(mydata, aes(x=x, y=y)) + geom_point() +
  ggtitle("Scatter Plot with 10,000 Observations")
```

you'll obtain a graph like the one in figure 11.5.

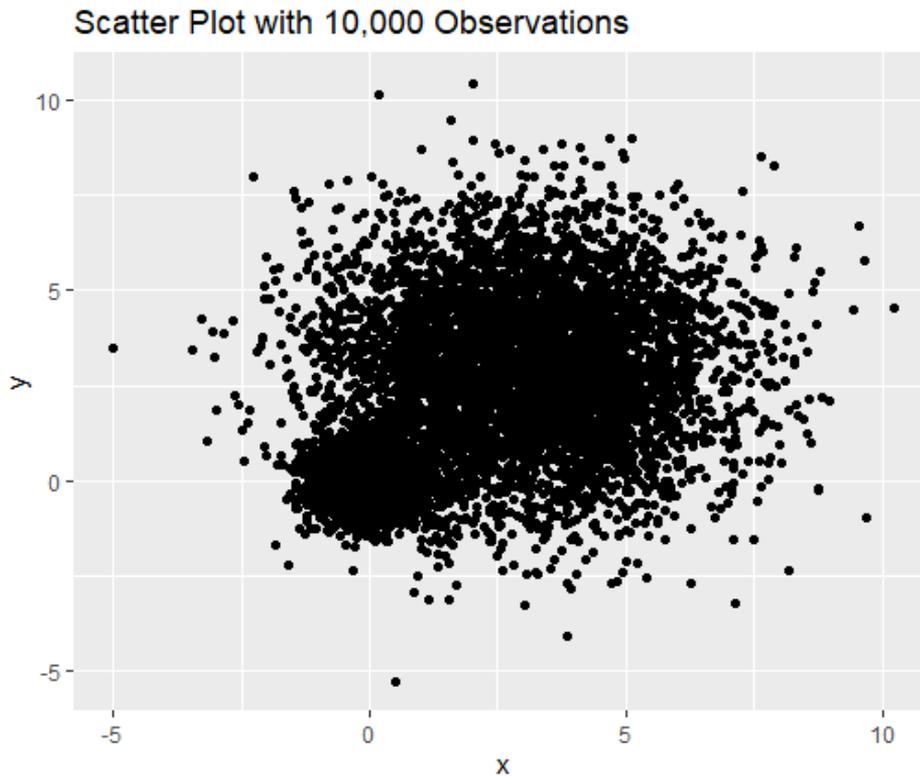


Figure 11.5 Scatter plot with 10,000 observations and significant overlap of data points. Note that the overlap of data points makes it difficult to discern where the concentration of data is greatest.

The overlap of data points in figure 11.5 makes it difficult to discern the relationship between x and y . R provides several graphical approaches that can be used when this occurs. They include the use of binning, color, and transparency to indicate the number of overprinted data points at any point on the graph.

The `smoothScatter()` function uses a kernel-density estimate to produce smoothed color density representations of the scatter plot. The following code

```
with(mydata,
      smoothScatter(x, y,
                    main="Scatter Plot Colored by Smoothed Densities"))
```

produces the graph in figure 11.6.

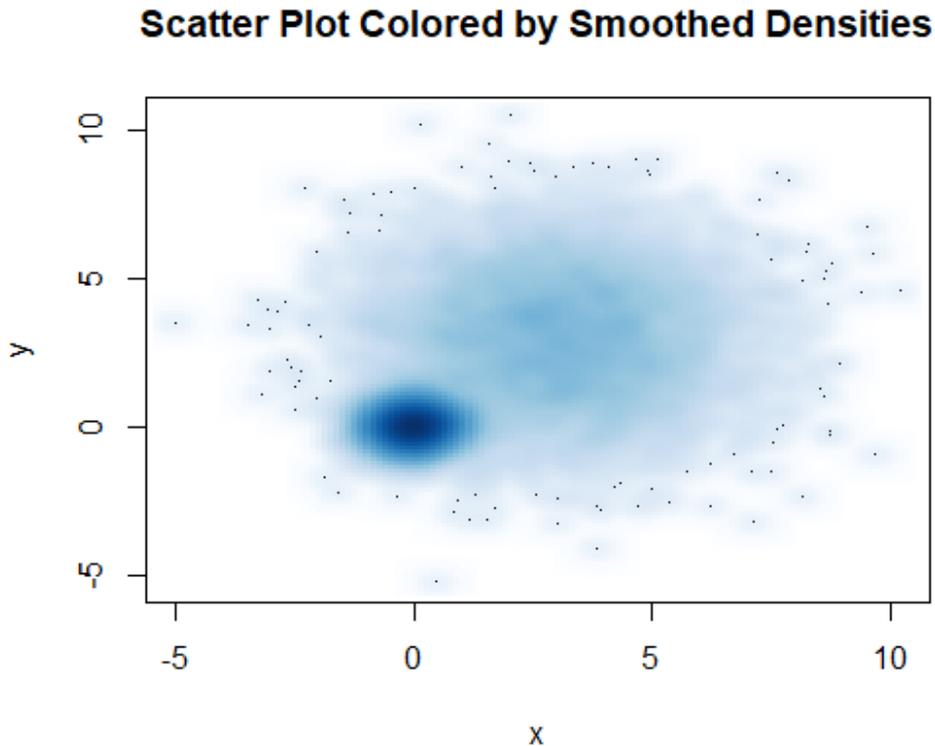


Figure 11.6 Scatter plot using `smoothScatter()` to plot smoothed density estimates. Densities are easy to read from the graph.

Using an alternative approach, the `geom_hex()` function in the `ggplot2` package provides bivariate binning into hexagonal cells (it looks better than it sounds). Basically, the plot area is divided into a grid of hexagonal cells and the number of points in each cell is displayed using color or shading. Applying this function to the dataset

```
ggplot(mydata, aes(x=x, y=y)) +
  geom_hex(bins=50) +
  scale_fill_continuous(trans = 'reverse') +
  ggtitle("Scatter Plot with 10,000 Observations")
```

gives you the scatter plot in figure 11.7.

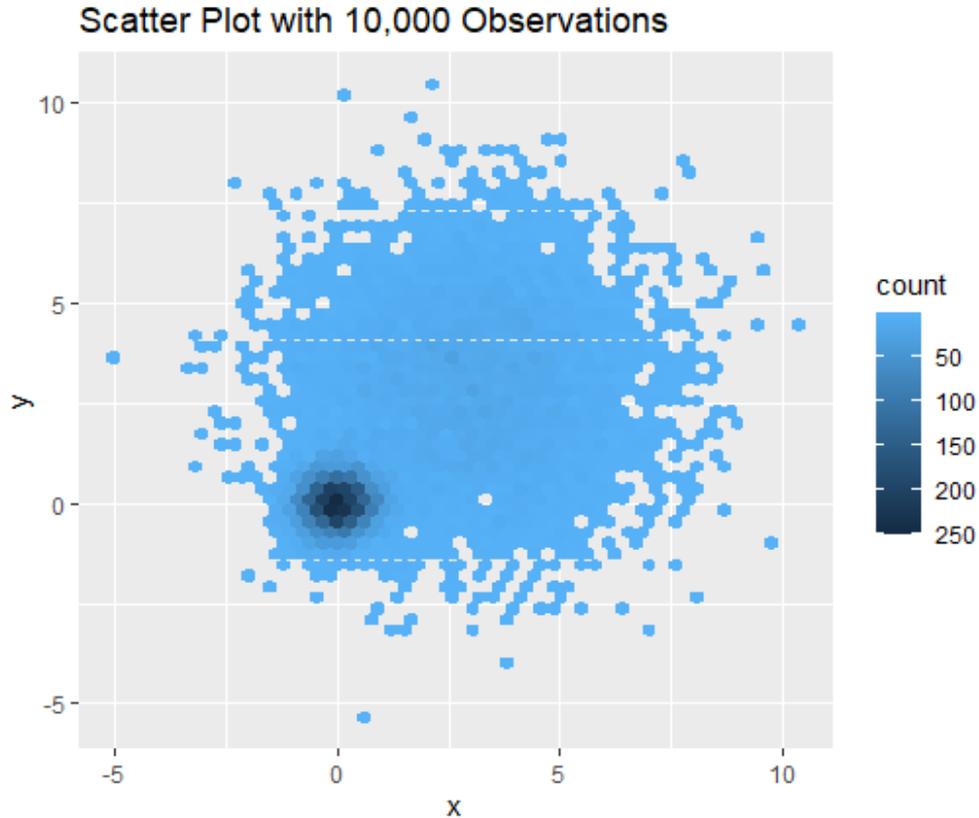


Figure 11.17 Scatter plot using hexagonal binning to display the number of observations at each point. Data concentrations are easy to see, and counts can be read from the legend.

By default, `geom_hex()` uses lighter colors to indicate greater density. In your code, the function `scale_fill_continuous(trans = 'reverse')` ensures that darker colors are used to indicate areas of greater density. I think that this is more intuitive, and matches the approach of other R functions used to visualize large datasets.

It's useful to note that the `hexbin()` function in the `hexbin` package, along with the `iplot()` function in the `IDPmisc` package, can be used to create readable scatter plot matrices for large datasets as well. See `?hexbin` and `?iplot` for examples.

11.1.3 3D scatter plots

Scatter plots and scatter-plot matrices display bivariate relationships. What if you want to visualize the interaction of three quantitative variables at once? In this case, you can use a 3D scatter plot.

For example, say that you're interested in the relationship between automobile mileage, weight, and displacement. You can use the `scatterplot3d()` function in the `scatterplot3d` package to picture their relationship. The format is

```
scatterplot3d(x, y, z)
```

where x is plotted on the horizontal axis, y is plotted on the vertical axis, and z is plotted in perspective. Continuing the example,

```
library(scatterplot3d)
with(mtcars,
     scatterplot3d(wt, disp, mpg,
                  main="Basic 3D Scatter Plot"))
```

produces the 3D scatter plot in figure 11.8.

Basic 3D Scatter Plot

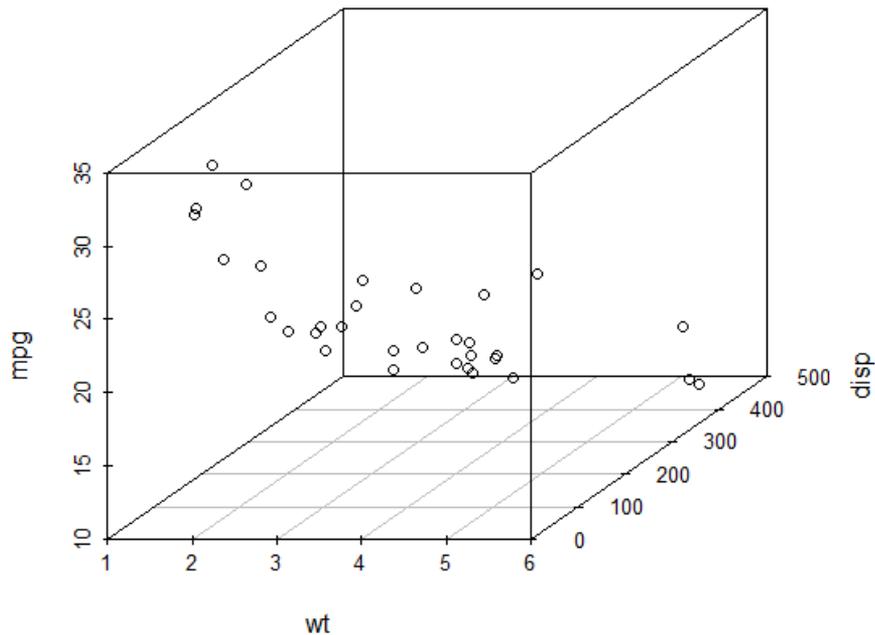


Figure 11.8 3D scatter plot of miles per gallon, auto weight, and displacement

The `scatterplot3d()` function offers many options, including the ability to specify symbols, axes, colors, lines, grids, highlighting, and angles. For example, the code

```
library(scatterplot3d)
with(mtcars,
  scatterplot3d(wt, disp, mpg,
    pch=16,
    highlight.3d=TRUE,
    type="h",
    main="3D Scatter Plot with Vertical Lines"))
```

produces a 3D scatter plot with highlighting to enhance the impression of depth, and vertical lines connecting points to the horizontal plane (see figure 11.9).

3D Scatter Plot with Vertical Lines

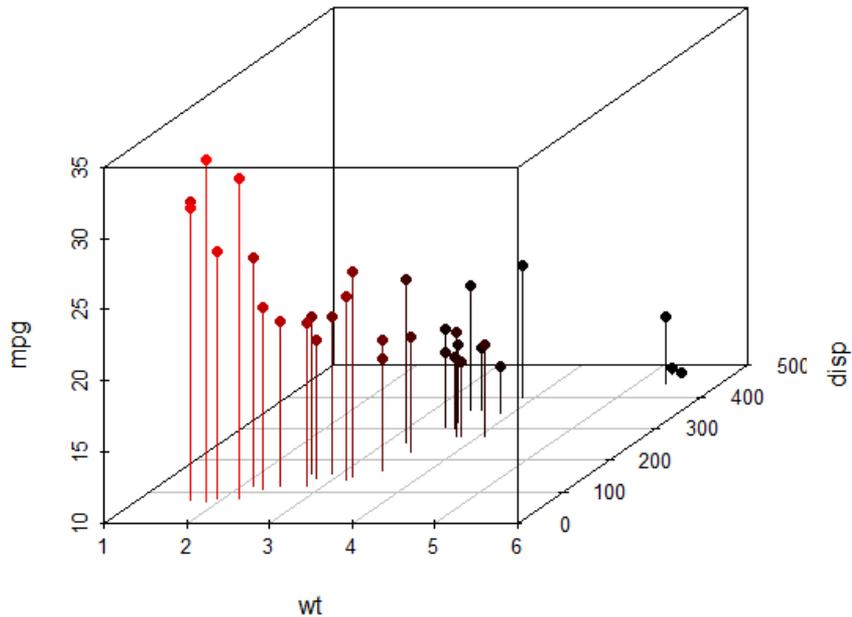


Figure 11.9 3D scatter plot with vertical lines and shading

As a final example, let's take the previous graph and add a regression plane. The necessary code is

```
library(scatterplot3d)
s3d <- with(mtcars,
  scatterplot3d(wt, disp, mpg,
    pch=16,
    highlight.3d=TRUE,
    type="h",
    main="3D Scatter Plot with Vertical Lines and Regression Plane"))
fit <- lm(mpg ~ wt+disp, data=mtcars)
s3d$plane3d(fit)
```

The resulting graph is provided in figure 11.10.

3D Scatter Plot with Vertical Lines and Regression Plane

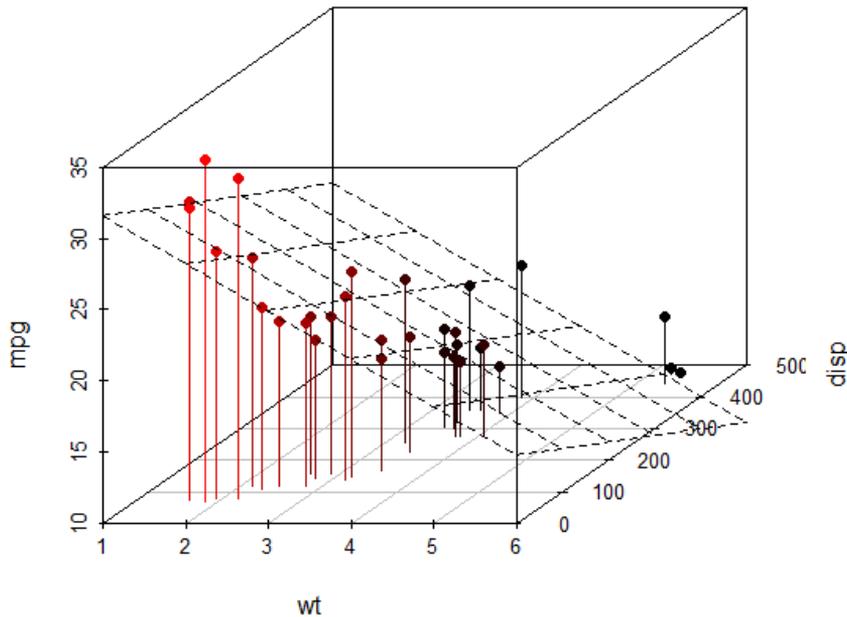


Figure 11.10 3D scatter plot with vertical lines, shading, and overlaid regression plane

The graph allows you to visualize the prediction of miles per gallon from automobile weight and displacement using a multiple regression equation. The plane represents the predicted values, and the points are the actual values. The vertical distances from the plane to the points are the residuals. Points that lie above the plane are under-predicted, whereas points that lie below the line are over-predicted. Multiple regression is covered in chapter 8.

11.1.4 Spinning 3D scatter plots

Three-dimensional scatter plots are much easier to interpret if you can interact with them. R provides several mechanisms for rotating graphs so that you can see the plotted points from more than one angle.

For example, you can create an interactive 3D scatter plot using the `plot3d()` function in the `rgl` package. It creates a spinning 3D scatter plot that can be rotated with the mouse. The format is

```
plot3d(x, y, z)
```

where x , y , and z are numeric vectors representing points. You can also add options like `col` and `size` to control the color and size of the points, respectively. Continuing the example, try this code:

```
library(rgl)
with(mtcars,
     plot3d(wt, disp, mpg, col="red", size=5))
```

You should get a graph like the one depicted in figure 11.11. Use the mouse to rotate the axes. I think you'll find that being able to rotate the scatter plot in three dimensions makes the graph much easier to understand.

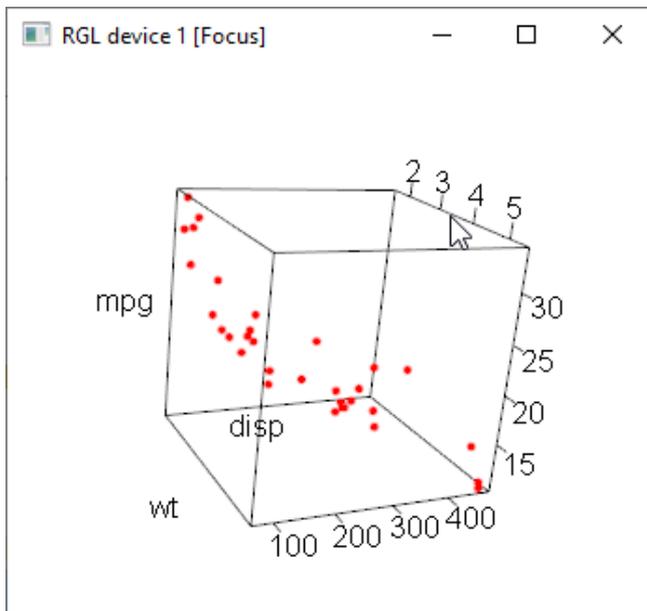


Figure 11.11 Rotating 3D scatter plot produced by the `plot3d()` function in the `rgl` package

You can perform a similar function with `scatter3d()` in the `car` package:

```
library(car)
with(mtcars,
     scatter3d(wt, disp, mpg))
```

The results are displayed in figure 11.12.

The `scatter3d()` function can include a variety of regression surfaces, such as linear, quadratic, smooth, and additive. The linear surface depicted is the default. Additionally, there are options for interactively identifying points. See `help(scatter3d)` for more details.

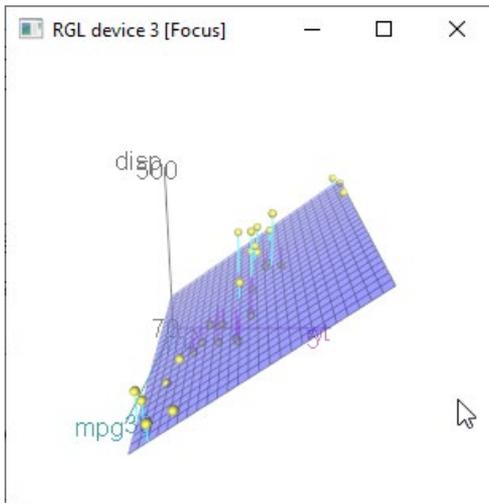


Figure 11.12 Spinning 3D scatter plot produced by the `scatter3d()` function in the `car` package

11.1.5 Bubble plots

In the previous section, you displayed the relationship between three quantitative variables using a 3D scatter plot. Another approach is to create a 2D scatter plot and use the size of the plotted point to represent the value of the third variable. This approach is referred to as a *bubble plot*.

A simple example of a bubble plot is given in following listing.

```
ggplot(mtcars,
  aes(x = wt, y = mpg, size = disp)) +
  geom_point() +
  labs(title="Bubble Plot with point size proportional to displacement",
    x="Weight of Car (lbs/1000)",
    y="Miles Per Gallon")
```

The result is a scatter plot displaying the relationship between car weight and fuel efficiency, where point size is proportional to each car's engine displacement. The graph is displayed in figure 11.13.

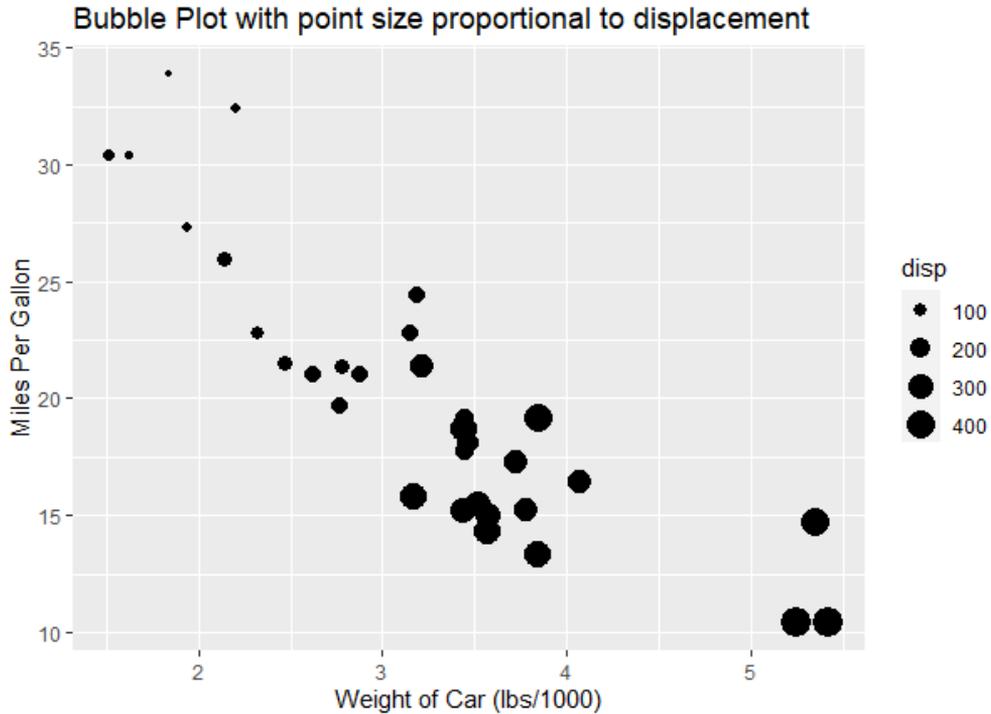


Figure 11.13 Bubble plot of car weight vs. mpg, where point size is proportional to engine displacement

While useful, we can improve on the default appearance by choosing a different point shape and color and adding transparency to deal with point overlaps. We'll also increase the possible range of bubble sizes to make discrimination easier. Finally, we'll use color to add the number of cylinders as a fourth variable. The code is given in listing 11.4 and the resulting graph is provided in figure 11.14.

Listing 11.4 An enhanced bubble plot

```
ggplot(mtcars,
  aes(x = wt, y = mpg, size = disp, fill=factor(cyl))) +
  geom_point(alpha = .5,
    color = "black",
    shape = 21) +
  scale_size_continuous(range = c(1, 10)) +
  labs(title = "Auto mileage by weight and horsepower",
    subtitle = "Motor Trend US Magazine (1973-74 models)",
    x = "Weight (1000 lbs)",
    y = "Miles/(US) gallon",
    size = "Engine\ndisplacement",
    fill = "Cylinders") +
  theme_minimal()
```

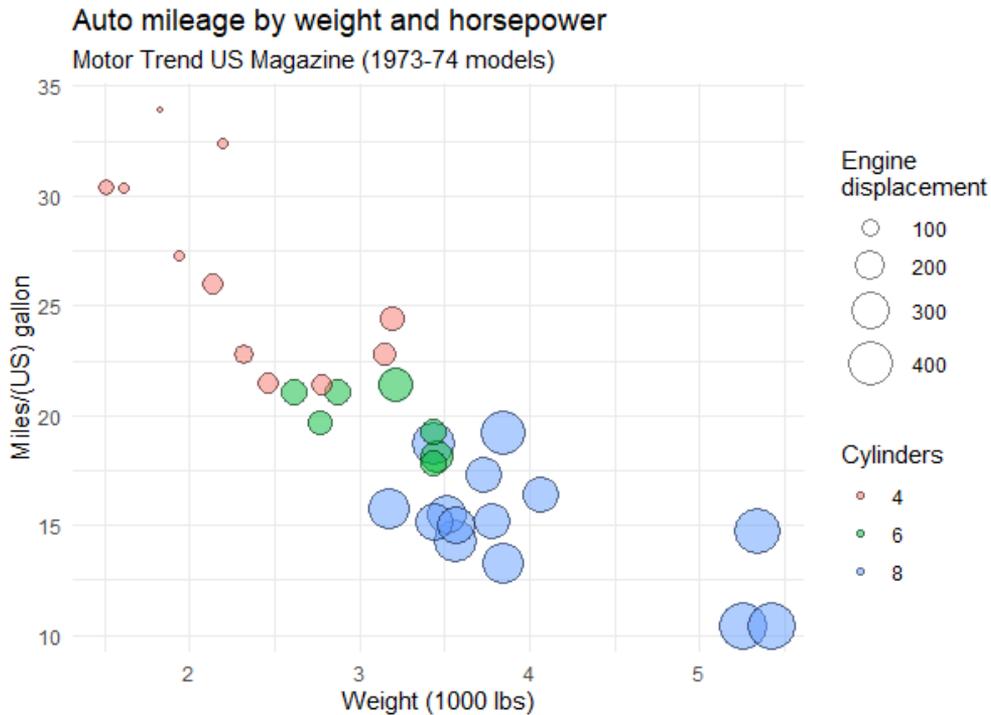


Figure 11.14. Enhanced bubble plot. Automobiles with more engine cylinders tend to have increased weight and engine displacement, and poorer fuel efficiency.

In general, statisticians involved in the R project tend to avoid bubble plots for the same reason they avoid pie charts. Humans typically have a harder time making judgments about volume than distance. But bubble charts are popular in the business world, so I'm including them here for completeness.

I've certainly had a lot to say about scatter plots. This attention to detail is due, in part, to the central place that scatter plots hold in data analysis. Although simple, they can help you visualize your data in an immediate and straightforward manner, uncovering relationships that might otherwise be missed.

11.2 Line charts

If you connect the points in a scatter plot moving from left to right, you have a line plot. The dataset `Orange` that come with the base installation contains age and circumference data for five orange trees. Consider the growth of the first orange tree, depicted in figure 11.15. The

plot on the left is a scatter plot, and the plot on the right is a line chart. As you can see, line charts are particularly good vehicles for conveying change. The graphs in figure 11.15 were created with the code in the following listing.

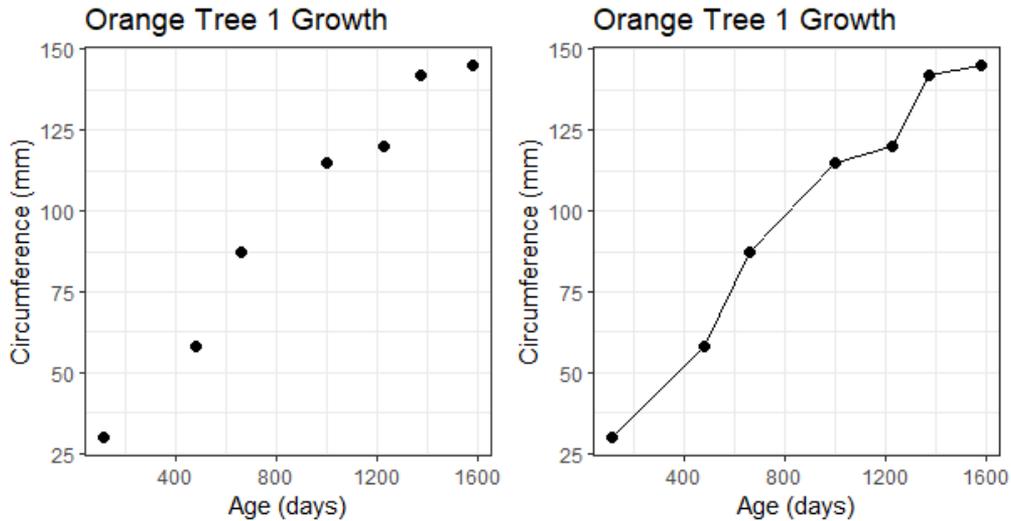


Figure 11.15 Comparison of a scatter plot and a line plot. A line charts helps the reader see growth and trend in the data.

Listing 11.5 Scatter plots vs. line plots

```
library(ggplot2)
tree1 <- subset(Orange, Tree == 1)
ggplot(data=tree1,
  aes(x=age, y=circumference)) +
  geom_point(size=2) +
  labs(title="Orange Tree 1 Growth",
    x = "Age (days)",
    y = "Circumference (mm)") +
  theme_bw()

ggplot(data=tree1,
  aes(x=age, y=circumference)) +
  geom_point(size=2) +
  geom_line() +
  labs(title="Orange Tree 1 Growth",
    x = "Age (days)",
    y = "Circumference (mm)") +
  theme_bw()
```

The only difference between the code for the two plots is the addition of the `geom_line()` function. Common options for this function are given in table 11.1. Each can be assigned a value or mapped to a categorical variable.

Table 11.1 `geom_line()` options

Option	Effect
<code>size</code>	Thickness of the line
<code>color</code>	Line color
<code>linetype</code>	Line pattern (e.g., dashed)

Possible line types are given in figure 11.16.

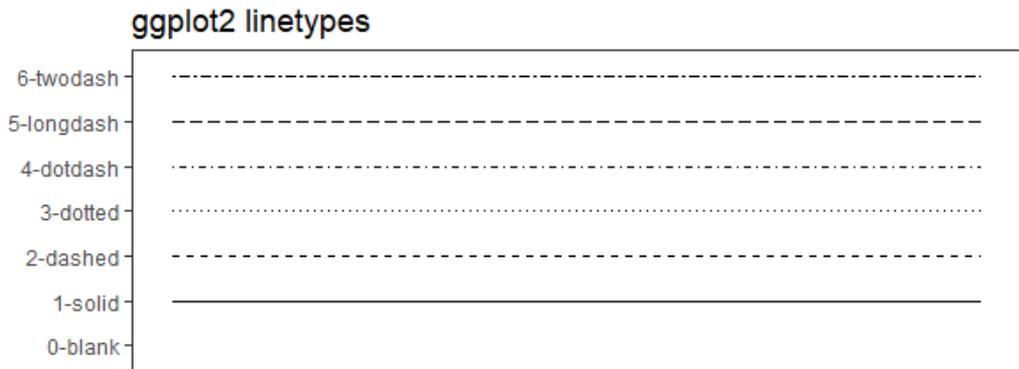


Figure 11.16 `ggplot2` linetypes. You can specify either the name or the number.

To demonstrate the creation of a more complex line chart, let's plot the growth of all five orange trees over time. Each tree will have its own distinctive line and color. The code is shown in the next listing and the results in figure 11.17.

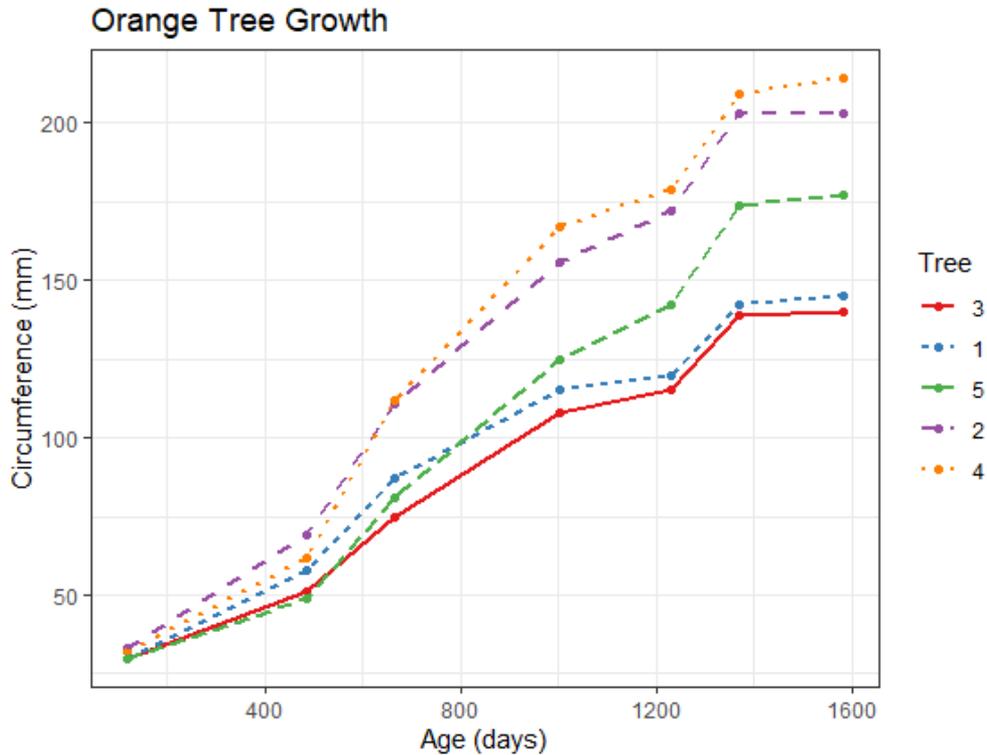


Figure 11.17 Line chart displaying the growth of five orange trees.

Listing 11.6 Line chart displaying the growth of five orange trees over time

```
library(ggplot2)
ggplot(data=Orange,
  aes(x=age, y=circumference, linetype=Tree, color=Tree)) +
  geom_point() +
  geom_line(size=1) +
  scale_color_brewer(palette="Set1") +
  labs(title="Orange Tree Growth",
    x="Age (days)",
    y="Circumference (mm)") +
  theme_bw()
```

In listing 11.6, the `aes()` function maps the tree number to both line type and color. The `scale_color_brewer()` function is used to select a color palette. Since I am chromatically challenged (i.e., I'm awful at choosing good colors), I rely heavily on predefined color palettes like those provided by the `RColorBrewer` package. Color palettes are described in greater detail in chapter 19 (advanced graphics).

You can see in the figure that tree 4 and tree 2 demonstrated the greatest growth across the range of days measured, and that tree 4 overtakes tree 2 at around 664 days. By default, the legend lists the lines in the opposite order that they appear on the chart (top to bottom in the legend is bottom to top in the graph). To make the orders match top to bottom, add

```
+ guides(color = guide_legend(reverse = TRUE),
         linetype = guide_legend(reverse = TRUE))
```

to the code in listing 11.6. In the next section, you'll explore ways of examining a number of correlation coefficients at once.

11.3 Corrgrams

Correlation matrices are a fundamental aspect of multivariate statistics. Which variables under consideration are strongly related to each other, and which aren't? Are there clusters of variables that relate in specific ways? As the number of variables grows, such questions can be harder to answer. *Corrgrams* are a relatively recent tool for visualizing the data in correlation matrices.

It's easier to explain a corrgram once you've seen one. Consider the correlations among the variables in the `mtcars` data frame. Here you have 11 variables, each measuring some aspect of 32 automobiles. You can get the correlations using the following code:

```
> round(cor(mtcars), 2)
      mpg  cyl  disp  hp  drat  wt  qsec  vs  am  gear  carb
mpg  1.00 -0.85 -0.85 -0.78  0.68 -0.87  0.42  0.66  0.60  0.48 -0.55
cyl -0.85  1.00  0.90  0.83 -0.70  0.78 -0.59 -0.81 -0.52 -0.49  0.53
disp -0.85  0.90  1.00  0.79 -0.71  0.89 -0.43 -0.71 -0.59 -0.56  0.39
hp   -0.78  0.83  0.79  1.00 -0.45  0.66 -0.71 -0.72 -0.24 -0.13  0.75
drat  0.68 -0.70 -0.71 -0.45  1.00 -0.71  0.09  0.44  0.71  0.70 -0.09
wt   -0.87  0.78  0.89  0.66 -0.71  1.00 -0.17 -0.55 -0.69 -0.58  0.43
qsec  0.42 -0.59 -0.43 -0.71  0.09 -0.17  1.00  0.74 -0.23 -0.21 -0.66
vs   0.66 -0.81 -0.71 -0.72  0.44 -0.55  0.74  1.00  0.17  0.21 -0.57
am   0.60 -0.52 -0.59 -0.24  0.71 -0.69 -0.23  0.17  1.00  0.79  0.06
gear  0.48 -0.49 -0.56 -0.13  0.70 -0.58 -0.21  0.21  0.79  1.00  0.27
carb -0.55  0.53  0.39  0.75 -0.09  0.43 -0.66 -0.57  0.06  0.27  1.00
```

Which variables are most related? Which variables are relatively independent? Are there any patterns? It isn't that easy to tell from the correlation matrix without significant time and effort (and probably a set of colored pens to make notations).

You can display that same correlation matrix using the `corrgram()` function in the `corrgram` package (see figure 11.18). The code is

```
library(corrgram)
corrgram(mtcars, order=TRUE, lower.panel=panel.shade,
         upper.panel=panel.pie, text.panel=panel.txt,
         main="Corrgram of mtcars intercorrelations")
```

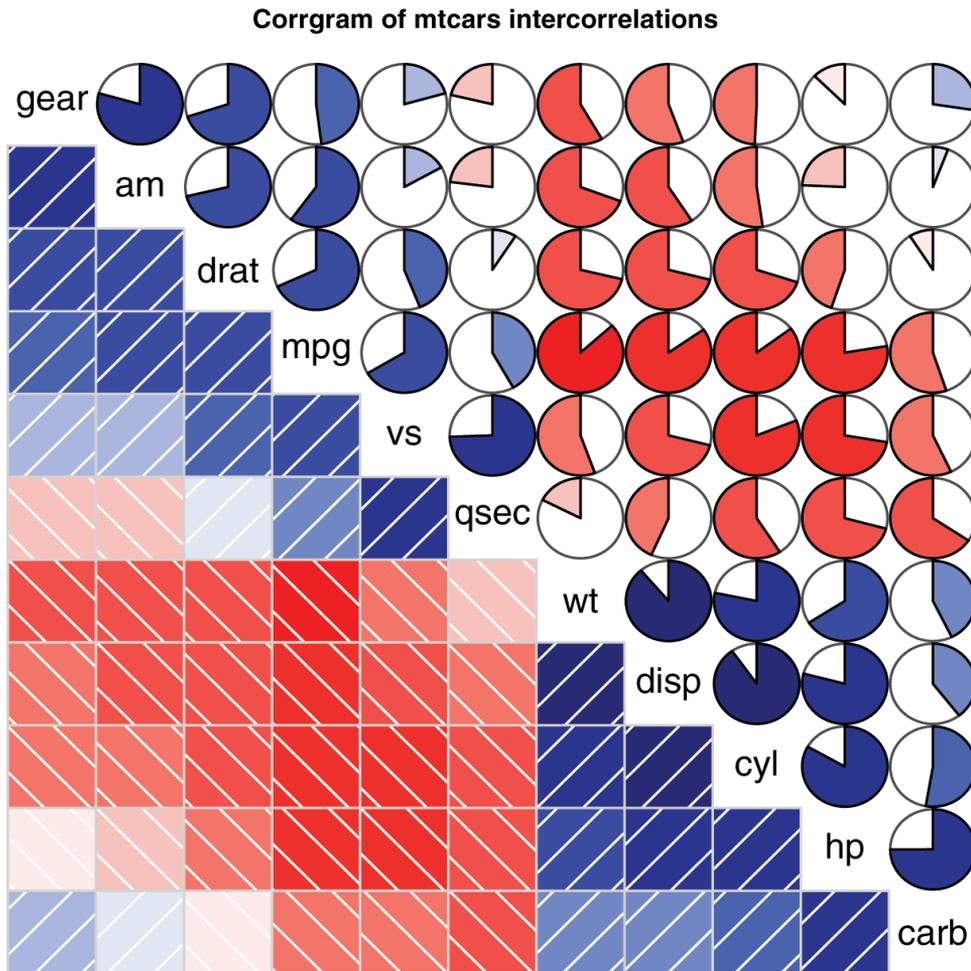


Figure 11.18 Corrgram of the correlations among the variables in the `mtcars` data frame. Rows and columns have been reordered using principal components analysis.

To interpret this graph, start with the lower triangle of cells (the cells below the principal diagonal). By default, a blue color and hashing that goes from lower left to upper right represent a positive correlation between the two variables that meet at that cell. Conversely, a red color and hashing that goes from the upper left to lower right represent a negative correlation. The darker and more saturated the color, the greater the magnitude of the correlation. Weak correlations, near zero, appear washed out. In the current graph, the rows and columns have been reordered (using principal components analysis discussed in chapter 14) to cluster variables together that have similar correlation patterns.

You can see from the shaded cells that `gear`, `am`, `drat`, and `mpg` are positively correlated with one another. You can also see that `wt`, `disp`, `cyl`, `hp`, and `carb` are positively correlated with one another. But the first group of variables is negatively correlated with the second group of variables. You can also see that the correlation between `carb` and `am` is weak, as is the correlation between `vs` and `gear`, `vs` and `am`, and `drat` and `qsec`.

The upper triangle of cells displays the same information using pies. Here, color plays the same role, but the strength of the correlation is displayed by the size of the filled pie slice. Positive correlations fill the pie starting at 12 o'clock and moving in a clockwise direction. Negative correlations fill the pie by moving in a counterclockwise direction.

The format of the `corrgram()` function is

```
corrgram(x, order=, panel=, text.panel=, diag.panel=)
```

where `x` is a data frame with one observation per row. When `order=TRUE`, the variables are reordered using a principal component analysis of the correlation matrix. Reordering can help make patterns of bivariate relationships more obvious.

The option `panel` specifies the type of off-diagonal panels to use. Alternatively, you can use the options `lower.panel` and `upper.panel` to choose different options below and above the main diagonal. The `text.panel` and `diag.panel` options refer to the main diagonal. Allowable values for `panel` are described in table 11.2.

Table 11.2 Panel options for the `corrgram()` function

Placement	Panel Option	Description
Off diagonal	<code>panel.pie</code>	The filled portion of the pie indicates the magnitude of the correlation.
	<code>panel.shade</code>	The depth of the shading indicates the magnitude of the correlation.
	<code>panel.ellipse</code>	Plots a confidence ellipse and smoothed line.
	<code>panel.pts</code>	Plots a scatter plot.
	<code>panel.conf</code>	Prints correlations and their confidence intervals.
	<code>panel.cor</code>	Prints correlations without their confidence intervals.

Main diagonal	<code>panel.txt</code>	Prints the variable name.
	<code>panel.minmax</code>	Prints the minimum and maximum value and variable name.
	<code>panel.density</code>	Prints the kernel density plot and variable name.

Let's try a second example. The code

```
library(corrgram)
corrgram(mtcars, order=TRUE, lower.panel=panel.ellipse,
         upper.panel=panel.pts, text.panel=panel.txt,
         diag.panel=panel.minmax,
         main="Corrgram of mtcars data using scatter plots
         and ellipses")
```

produces the graph in figure 11.19. Here you're using smoothed fit lines and confidence ellipses in the lower triangle and scatter plots in the upper triangle.

Corrgram of mtcars data using scatter plots and ellipses

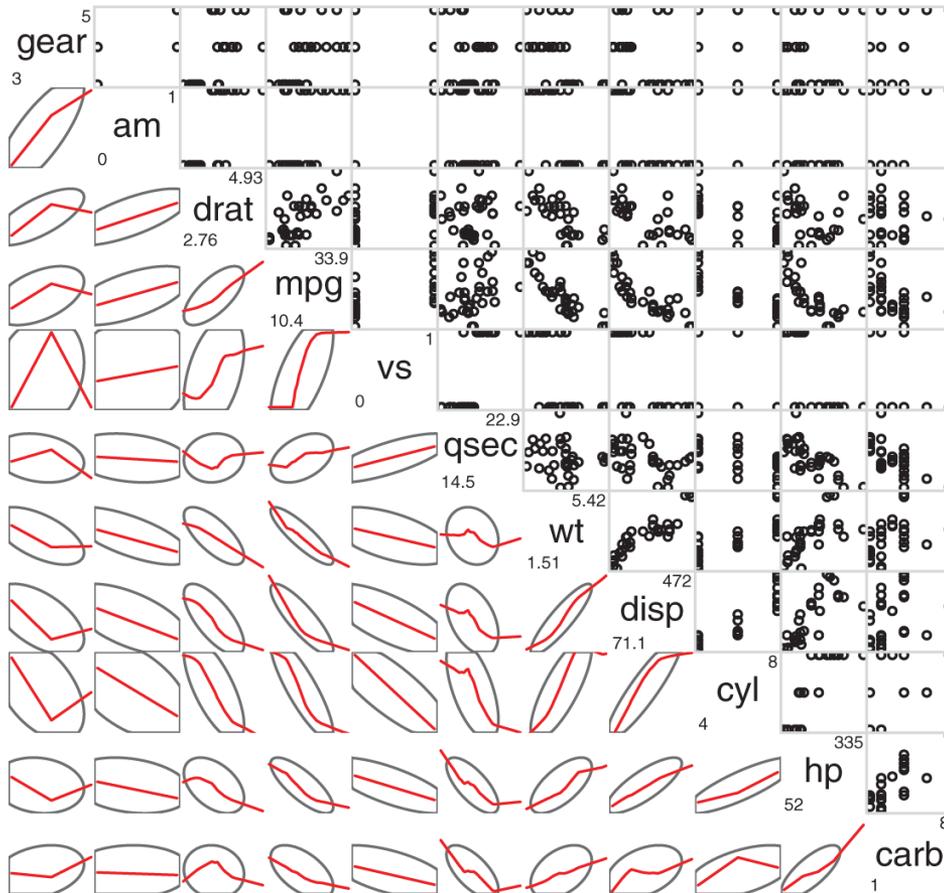


Figure 11.19 Corrgram of the correlations among the variables in the `mtcars` data frame. The lower triangle contains smoothed best-fit lines and confidence ellipses, and the upper triangle contains scatter plots. The diagonal panel contains minimum and maximum values. Rows and columns have been reordered using principal components analysis.

Why do the scatter plots look odd?

Several of the variables that are plotted in figure 11.19 have limited allowable values. For example, the number of gears is 3, 4, or 5. The number of cylinders is 4, 6, or 8. Both `am` (transmission type) and `vs` (V/S) are dichotomous. This explains the odd-looking scatter plots in the upper diagonal.

Always be careful that the statistical methods you choose are appropriate to the form of the data. Specifying these variables as ordered or unordered factors can serve as a useful check. When R knows that a variable is categorical or ordinal, it attempts to apply statistical methods that are appropriate to that level of measurement.

We'll finish with one more example. The code

```
corrgram(mtcars, order=TRUE, lower.panel=panel.shade,
         upper.panel=panel.cor,
         main="Corrgram of mtcars data using shading and coefficients")
```

produces the graph in figure 11.20. Here you're using shading in the lower triangle, order variables to emphasize correlation patterns, and printing the correlation values in the upper triangle.

Corrgram of mtcars data using shading and coefficients

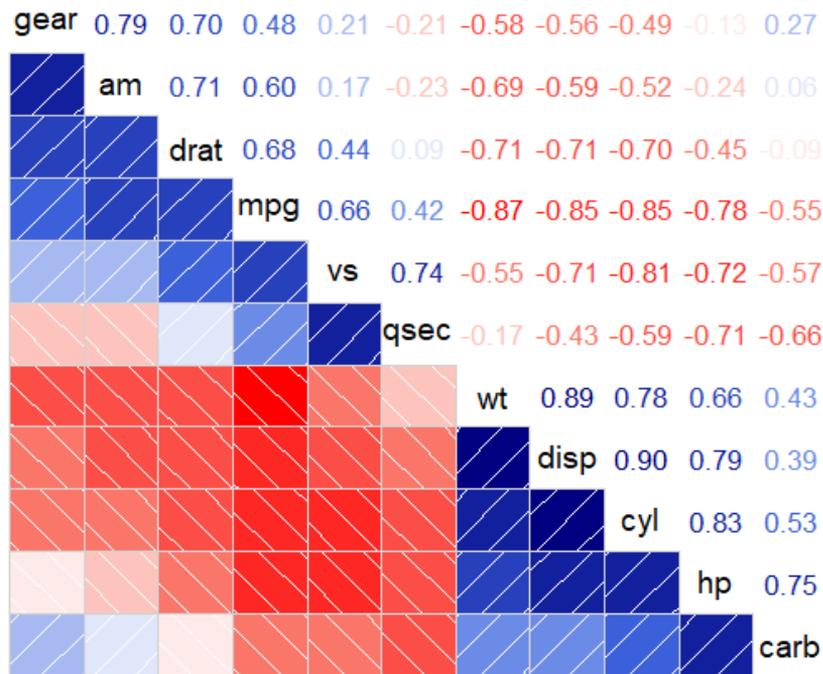


Figure 11.20 Corrgram of the correlations among the variables in the `mtcars` data frame. The lower triangle is shaded to represent the magnitude and direction of the correlations. Rows and columns have been reordered using principal components analysis. Correlation coefficients are printed in the upper triangle.

Before moving on, I should point out that you can control the colors used by the `corrgram()` function. To do so, specify four colors in the `colorRampPalette()` function, and include the results using the `col.regions` option. Here's an example:

```
library(corrgram)
cols <- colorRampPalette(c("darkgoldenrod4", "burlywood1",
                          "darkkhaki", "darkgreen"))
corrgram(mtcars, order=TRUE, col.regions=cols,
         lower.panel=panel.shade,
         upper.panel=panel.conf, text.panel=panel.txt,
         main="A Corrgram (or Horse) of a Different Color")
```

Try it and see what you get.

Corrgrams can be a useful way to examine large numbers of bivariate relationships among quantitative variables. Because they're relatively new, the greatest challenge is to educate the recipient on how to interpret them. To learn more, see Michael Friendly's article "Corrgrams: Exploratory Displays for Correlation Matrices," available at www.math.yorku.ca/SCS/Papers/corrgram.pdf.

11.4 Mosaic plots

Up to this point, we've been exploring methods of visualizing relationships among quantitative/continuous variables. But what if your variables are categorical? When you're looking at a single categorical variable, you can use a bar or pie chart. If there are two categorical variables, you can use a stacked bar chart (section 6.1.2). But what do you do if there are more than two categorical variables?

One approach is to use *mosaic plots*. In a mosaic plot, the frequencies in a multidimensional contingency table are represented by nested rectangular regions that are proportional to their cell frequency. Color and/or shading can be used to represent residuals from a fitted model. For details, see Meyer, Zeileis, and Hornick (2006), or Michael Friendly's excellent tutorial (<http://mng.bz/3p0d>).

Mosaic plots can be created with the `mosaic()` function from the `vcd` library (there's a `mosaicplot()` function in the basic installation of R, but I recommend you use the `vcd` package for its more extensive features). As an example, consider the *Titanic* dataset available in the base installation. It describes the number of passengers who survived or died, cross-classified by their class (1st, 2nd, 3rd, Crew), sex (Male, Female), and age (Child, Adult). This is a well-studied dataset. You can see the cross-classification using the following code:

```
> ftable(Titanic)
      Survived No Yes
Class Sex Age
1st  Male Child    0  5
     Adult   118  57
     Female Child  0  1
     Adult    414  40
2nd  Male Child    0 11
     Adult   154  14
     Female Child  0 13
```

```

      Adult    13 80
3rd Male Child    35 13
      Adult   387 75
      Female Child 17 14
      Adult    89 76
Crew Male Child    0 0
      Adult   670 192
      Female Child  0 0
      Adult     3 20

```

The `mosaic()` function can be invoked as

```
mosaic(table)
```

where *table* is a contingency table in array form, or

```
mosaic(formula, data=)
```

where *formula* is a standard R formula, and *data* specifies either a data frame or a table. Adding the option `shade=TRUE` colors the figure based on Pearson residuals from a fitted model (independence by default), and the option `legend=TRUE` displays a legend for these residuals.

For example, both

```
library(vcd)
mosaic(Titanic, shade=TRUE, legend=TRUE)
```

and

```
library(vcd)
mosaic(~Class+Sex+Age+Survived, data=Titanic, shade=TRUE, legend=TRUE)
```

will produce the graph shown in figure 11.21. The formula version gives you greater control over the selection and placement of variables in the graph.

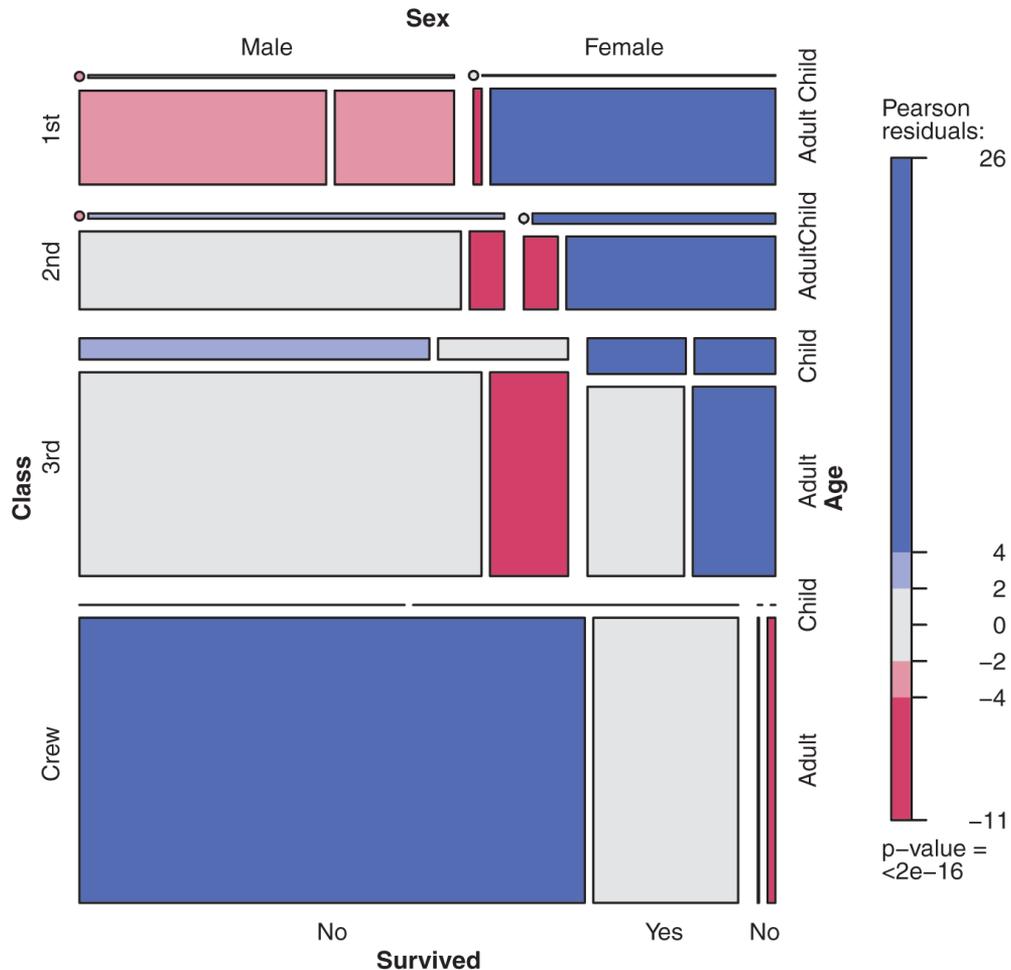


Figure 11.21 Mosaic plot describing *Titanic* survivors by class, sex, and age

A great deal of information is packed into this one picture. For example, as a person moves from crew to first class, the survival rate increases precipitously. Most children were in third and second class. Most females in first class survived, whereas only about half the females in third class survived. There were few females in the crew, causing the Survived labels (No, Yes at the bottom of the chart) to overlap for this group. Keep looking, and you'll see many more interesting facts. Remember to look at the relative widths and heights of the rectangles. What else can you learn about that night?

Extended mosaic plots add color and shading to represent the residuals from a fitted model. In this example, the blue shading indicates cross-classifications that occur more often than expected, assuming that survival is unrelated to class, gender, and age. Red shading indicates cross-classifications that occur less often than expected under the independence model. Be sure to run the example so that you can see the results in color. The graph indicates that more first-class women survived, and more male crew members died than would be expected under an independence model. Fewer third-class men survived than would be expected if survival was independent of class, gender, and age. If you'd like to explore mosaic plots in greater detail, try running `example(mosaic)`.

11.5 Summary

- Scatter plots and scatter plot matrices allow you to visualize relationships between quantitative variables two at a time. The plots can be enhanced with linear and loess fit lines showing trends.
- When creating a scatter plot based on a large volume of data, methods that plot densities rather than points are particularly useful.
- The relationships among three quantitative variables can be explored using 3D scatter plots or 2D bubble charts.
- Change over time can be described effectively with line charts.
- Large correlation matrices are difficult to understand in table form, but easily explored via corrgrams – visual plots of correlation matrices.
- The relationships between two or more categorical variables can be visualized with mosaic charts.

12

Resampling statistics and bootstrapping

This chapter covers

- Understanding the logic of permutation tests
- Applying permutation tests to linear models
- Using bootstrapping to obtain confidence intervals

In chapters 7, 8, and 9, we reviewed statistical methods that test hypotheses and estimate confidence intervals for population parameters by assuming that the observed data is sampled from a normal distribution or some other well-known theoretical distribution. But there will be many cases in which this assumption is unwarranted. Statistical approaches based on randomization and resampling can be used in cases where the data is sampled from unknown or mixed distributions, where sample sizes are small, where outliers are a problem, or where devising an appropriate test based on a theoretical distribution is too complex and mathematically intractable.

In this chapter, we'll explore two broad statistical approaches that use randomization: permutation tests and bootstrapping. Historically, these methods were only available to experienced programmers and expert statisticians. Contributed packages in R now make them readily available to a wider audience of data analysts.

We'll also revisit problems that were initially analyzed using traditional methods (for example, t-tests, chi-square tests, ANOVA, and regression) and see how they can be approached using these robust, computer-intensive methods. To get the most out of section 12.2, be sure to read chapter 7 first. Chapters 8 and 9 serve as prerequisites for section 12.3. Other sections can be read on their own.

12.1 Permutation tests

Permutation tests, also called *randomization* or *re-randomization* tests, have been around for decades, but it took the advent of high-speed computers to make them practically available. To understand the logic of a permutation test, consider the following hypothetical problem. Ten subjects have been randomly assigned to one of two treatment conditions (A or B), and an outcome variable (score) has been recorded. The results of the experiment are presented in table 12.1.

Table 12.1 Hypothetical two-group problem

Treatment A	Treatment B
40	57
57	64
45	55
55	62
58	65

The data are also displayed in figure 12.1. Is there enough evidence to conclude that the treatments differ in their impact?

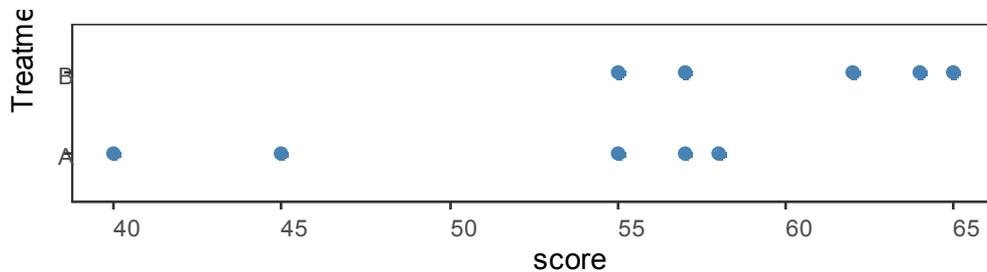


Figure 12.1 Strip chart of the hypothetical treatment data in table 12.1

In a parametric approach, you might assume that the data are sampled from normal populations with equal variances and apply a two-tailed independent-groups t-test. The null hypothesis is that the population mean for Treatment A is equal to the population mean for Treatment B. You'd calculate a t-statistic from the data and compare it to the theoretical distribution. If the observed t-statistic is sufficiently extreme, say outside the middle 95% of values in the theoretical distribution, you'd reject the null hypothesis and declare that the population means for the two groups are unequal at the 0.05 level of significance.

A permutation test takes a different approach. If the two treatments are truly equivalent, the label (Treatment A or Treatment B) assigned to an observed score is arbitrary. To test for differences between the two treatments, you could follow these steps:

1. Calculate the observed t-statistic, as in the parametric approach; call this t_0 .
2. Place all 10 scores in a single group.
3. Randomly assign five scores to Treatment A and five scores to Treatment B.
4. Calculate and record the new observed t-statistic.
5. Repeat steps 3–4 for every possible way of assigning five scores to Treatment A and five scores to Treatment B. There are 252 such possible arrangements.
6. Arrange the 252 t-statistics in ascending order. This is the empirical distribution, based on (or conditioned on) the sample data.
7. If t_0 falls outside the middle 95% of the empirical distribution, reject the null hypothesis that the population means for the two treatment groups are equal at the 0.05 level of significance.

Notice that the same t-statistic is calculated in both the permutation and parametric approaches. But instead of comparing the statistic to a theoretical distribution in order to determine if it was extreme enough to reject the null hypothesis, it's compared to an empirical distribution created from permutations of the observed data. This logic can be extended to most classical statistical tests and linear models.

In the previous example, the empirical distribution was based on all possible permutations of the data. In such cases, the permutation test is called an *exact* test. As the sample sizes increase, the time required to form all possible permutations can become prohibitive. In such cases, you can use Monte Carlo simulation to sample from all possible permutations. Doing so provides an approximate test.

If you're uncomfortable assuming that the data is normally distributed, concerned about the impact of outliers, or feel that the dataset is too small for standard parametric approaches, a permutation test provides an excellent alternative. R has some of the most comprehensive and sophisticated packages for performing permutation tests currently available. The remainder of this section focuses on two contributed packages: the `coin` package and the `lmPerm` package. The `coin` package provides a comprehensive framework for permutation tests applied to independence problems, whereas the `lmPerm` package provides permutation tests for ANOVA and regression designs. We'll consider each package in turn. Be sure to install them (`install.packages(c("coin", "lmPerm"))`) before continuing.

Setting the random number seed

Before moving on, it's important to remember that permutation tests use pseudo-random numbers to sample from all possible permutations (when performing an approximate test). Therefore, the results will change each time the test is performed. Setting the random-number seed in R allows you to fix the random numbers generated. This is particularly useful when you want to share your examples with others, because results will always be the same if the calls are made with the same seed. Setting the random number seed to **1234** (that is, `set.seed(1234)`) will allow you to replicate the results presented in this chapter.

12.2 Permutation tests with the coin package

The `coin` package provides a general framework for applying permutation tests to independence problems. With this package, you can answer such questions as

- Are responses independent of group assignment?
- Are two numeric variables independent?
- Are two categorical variables independent?

Using convenience functions provided in the package (see table 12.2), you can perform permutation test equivalents for most of the traditional statistical tests covered in chapter 7.

Table 12.2 `coin` functions providing permutation test alternatives to traditional tests

Test	<code>coin</code> function
Two- and K-sample permutation test	<code>oneway_test(y ~ A)</code>
Wilcoxon–Mann–Whitney rank-sum test	<code>wilcox_test(y ~ A)</code>
Kruskal–Wallis test	<code>kruskal_test(y ~ A)</code>
Pearson's chi-square test	<code>chisq_test(A ~ B)</code>
Cochran–Mantel–Haenszel test	<code>cmh_test(A ~ B C)</code>
Linear-by-linear association test	<code>lbl_test(D ~ E)</code>

Spearman's test	<code>spearman_test(y ~ x)</code>
Friedman test	<code>friedman_test(y ~ A C)</code>
Wilcoxon signed-rank test	<code>wilcoxsign_test(y1 ~ y2)</code>

In the `coin` function column, `y` and `x` are numeric variables, `A` and `B` are categorical factors, `C` is a categorical blocking variable, `D` and `E` are ordered factors, and `y1` and `y2` are matched numeric variables.

Each of the functions listed in table 12.2 takes the form

```
function_name(formula, data, distribution=)
```

where

- `formula` describes the relationship among variables to be tested. Examples are given in the table.
- `data` identifies a data frame.
- `distribution` specifies how the empirical distribution under the null hypothesis should be derived. Possible values are `exact`, `asymptotic`, and `approximate`.

If `distribution="exact"`, the distribution under the null hypothesis is computed exactly (that is, from all possible permutations). The distribution can also be approximated by its asymptotic distribution (`distribution="asymptotic"`) or via Monte Carlo resampling (`distribution="approximate(nresample=n)"`), where `n` indicates the number of random replications used to approximate the exact distribution. The default is 10,000 replications. At present, `distribution="exact"` is only available for two-sample problems.

NOTE In the `coin` package, categorical variables and ordinal variables must be coded as factors and ordered factors, respectively. Additionally, the data must be stored in a data frame.

In the remainder of this section, you'll apply several of the permutation tests described in table 12.2 to problems from previous chapters. This will allow you to compare the results to more traditional parametric and nonparametric approaches. We'll end this discussion of the `coin` package by considering advanced extensions.

12.2.1 Independent two-sample and k-sample tests

To begin, let's compare an independent samples t-test with a one-way exact test applied to the hypothetical data in table 12.2. The results are given in the following listing.

Listing 12.1 t-test vs. one-way permutation test for the hypothetical data

```

> library(coin)
> score <- c(40, 57, 45, 55, 58, 57, 64, 55, 62, 65)
> treatment <- factor(c(rep("A",5), rep("B",5)))
> mydata <- data.frame(treatment, score)
> t.test(score~treatment, data=mydata, var.equal=TRUE)

      Two Sample t-test

data:  score by treatment
t = -2.345, df = 8, p-value = 0.04705
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -19.0405455 -0.1594545
sample estimates:
mean in group A mean in group B
      51.0      60.6

> oneway_test(score~treatment, data=mydata, distribution="exact")

      Exact Two-Sample Fisher-Pitman Permutation Test

data:  score by treatment (A, B)
Z = -1.9147, p-value = 0.07143
alternative hypothesis: true mu is not equal to 0

```

The traditional t-test indicates a significant group difference ($p < .05$), whereas the exact test doesn't ($p > 0.072$). With only 10 observations, I'd be more inclined to trust the results of the permutation test and attempt to collect more data before reaching a final conclusion.

Next, consider the Wilcoxon–Mann–Whitney U test. In chapter 7, we examined the difference in the probability of imprisonment in Southern versus non-Southern US states using the `wilcox.test()` function. Using an exact Wilcoxon rank-sum test, you'd get

```

> library(MASS)
> UScrime$So <- factor(UScrime$So)
> wilcox_test(Prob ~ So, data=UScrime, distribution="exact")

      Exact Wilcoxon Mann-Whitney Rank Sum Test

data:  Prob by So (0, 1)
Z = -3.7, p-value = 8.488e-05
alternative hypothesis: true mu is not equal to 0

```

suggesting that incarceration is more likely in Southern states. Note that in the previous code, the numeric variable `So` was transformed into a factor. This is because the `coin` package requires that all categorical variables be coded as factors. Additionally, you may have noted that these results agree exactly with the results of the `wilcox.test()` function in chapter 7. This is because `wilcox.test()` also computes an exact distribution by default.

Finally, consider a k-sample test. In chapter 9, you used a one-way ANOVA to evaluate the impact of five drug regimens on cholesterol reduction in a sample of 50 patients. An approximate k-sample permutation test can be performed instead, using this code:

```
> library(multcomp)
> set.seed(1234)
> oneway_test(response~trt, data=cholesterol,
distribution=approximate(nresample=9999))
```

Approximative K-Sample Fisher-Pitman Permutation Test

```
data: response by trt (1time, 2times, 4times, drugD, drugE)
chi-squared = 36.381, p-value < 1e-04
```

Here, the reference distribution is based on 9,999 permutations of the data. The random-number seed is set so that your results will be the same as mine. There's clearly a difference in response among patients in the various groups.

12.2.2 Independence in contingency tables

You can use permutation tests to assess the independence of two categorical variables using either the `chisq_test()` or `cmh_test()` function. The latter function is used when data is stratified on a third categorical variable. If both variables are ordinal, you can use the `lbl_test()` function to test for a linear trend.

In chapter 7, you applied a chi-square test to assess the relationship between arthritis treatment and improvement. Treatment had two levels (Placebo and Treated), and Improved had three levels (None, Some, and Marked). The Improved variable was encoded as an ordered factor.

If you want to perform a permutation version of the chi-square test, you can use the following code:

```
> library(coin)
> library(vcd)
> Arthritis <- transform(Arthritis,
Improved=as.factor(as.numeric(Improved)))
> set.seed(1234)
> chisq_test(Treatment~Improved, data=Arthritis,
distribution=approximate(nresample=9999))
```

Approximative Pearson Chi-Squared Test

```
data: Treatment by Improved (1, 2, 3)
chi-squared = 13.055, p-value = 0.0018
```

This gives you an approximate chi-square test based on 9,999 replications. You might ask why you transformed the variable Improved from an ordered factor to a categorical factor. (Good question!) If you'd left it an ordered factor, `coin()` would have generated a linear \times linear trend test instead of a chi-square test. Although a trend test would be a good choice in this situation, keeping it a chi-square test allows you to compare the results with those reported in chapter 7.

12.2.3 Independence between numeric variables

The `spearman_test()` function provides a permutation test of the independence of two numeric variables. In chapter 7, we examined the correlation between illiteracy rates and murder rates for US states. You can test the association via permutation, using the following code:

```
> states <- as.data.frame(state.x77)
> set.seed(1234)
> spearman_test(illiteracy~Murder, data=states,
  distribution=approximate(B=9999))

Approximative Spearman Correlation Test

data: illiteracy by Murder
Z = 4.7065, p-value < 1e-04
alternative hypothesis: true rho is not equal to 0
```

Based on an approximate permutation test with 9,999 replications, the hypothesis of independence can be rejected. Note that `state.x77` is a matrix. It had to be converted into a data frame for use in the `coin` package.

12.2.4 Dependent two-sample and k-sample tests

Dependent sample tests are used when observations in different groups have been matched or when repeated measures are used. For permutation tests with two paired groups, the `wilcoxsign_test()` function can be used. For more than two groups, use the `friedman_test()` function.

In chapter 7, we compared the unemployment rate for urban males age 14–24 (U1) with urban males age 35–39 (U2). Because the two variables are reported for each of the 50 US states, you have a two-dependent groups design (state is the matching variable). You can use an exact Wilcoxon signed-rank test to see if unemployment rates for the two age groups are equal:

```
> library(coin)
> library(MASS)
> wilcoxsign_test(U1~U2, data=UScrime, distribution="exact")

Exact Wilcoxon-Signed-Rank Test

data: y by x (neg, pos)
stratified by block
Z = 5.9691, p-value = 1.421e-14
alternative hypothesis: true mu is not equal to 0
```

Based on the results, you'd conclude that the unemployment rates differ.

12.2.5 Going further

The `coin` package provides a general framework for testing that one group of variables is independent of a second group of variables (with optional stratification on a blocking variable) against arbitrary alternatives, via approximate permutation tests. In particular, the `independence_test()` function lets you approach most traditional tests from a permutation perspective and create new and novel statistical tests for situations not covered by traditional methods. This flexibility comes at a price: a high level of statistical knowledge is required to use the function appropriately. See the vignettes that accompany the package (accessed via `vignette("coin")`) for further details.

In the next section, you'll learn about the `lmPerm` package. This package provides a permutation approach to linear models, including regression and analysis of variance.

12.3 Permutation tests with the `lmPerm` package

The `lmPerm` package provides support for a permutation approach to linear models. In particular, the `lmp()` and `aovp()` functions are the `lm()` and `aov()` functions modified to perform permutation tests rather than normal theory tests.

The parameters in the `lmp()` and `aovp()` functions are similar to those in the `lm()` and `aov()` functions, with the addition of a `perm=` parameter. The `perm=` option can take the value `Exact`, `Prob`, or `SPR`. `Exact` produces an exact test, based on all possible permutations. `Prob` samples from all possible permutations. Sampling continues until the estimated standard deviation falls below 0.1 of the estimated p-value. The stopping rule is controlled by an optional `Ca` parameter. Finally, `SPR` uses a sequential probability ratio test to decide when to stop sampling. Note that if the number of observations is greater than 10, `perm="Exact"` will automatically default to `perm="Prob"`; exact tests are only available for small problems.

To see how this works, you'll apply a permutation approach to simple regression, polynomial regression, multiple regression, one-way analysis of variance, one-way analysis of covariance, and a two-way factorial design.

12.3.1 Simple and polynomial regression

In chapter 8, you used linear regression to study the relationship between weight and height for a group of 15 women. Using `lmp()` instead of `lm()` generates the permutation test results shown in the following listing.

Listing 12.2 Permutation tests for simple linear regression

```
> library(lmPerm)
> set.seed(1234)
> fit <- lmp(weight~height, data=women, perm="Prob")
[1] "Settings: unique SS : numeric variables centered"
> summary(fit)

Call:
lmp(formula = weight ~ height, data = women, perm = "Prob")
```

```

Residuals:
  Min   1Q Median   3Q   Max
-1.733 -1.133 -0.383  0.742  3.117

Coefficients:
  Estimate Iter Pr(Prob)
height    3.45 5000 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.5 on 13 degrees of freedom
Multiple R-Squared:  0.991,    Adjusted R-squared:  0.99
F-statistic: 1.43e+03 on 1 and 13 DF, p-value: 1.09e-14

```

To fit a quadratic equation, you could use the code in this next listing.

Listing 12.3 Permutation tests for polynomial regression

```

> library(lmPerm)
> set.seed(1234)
> fit <- lmp(weight~height + I(height^2), data=women, perm="Prob")
[1] "Settings: unique SS : numeric variables centered"
> summary(fit)

Call:
lmp(formula = weight ~ height + I(height^2), data = women, perm = "Prob")

Residuals:
  Min   1Q Median   3Q   Max
-0.5094 -0.2961 -0.0094  0.2862  0.5971

Coefficients:
  Estimate Iter Pr(Prob)
height    -7.3483 5000 <2e-16 ***
I(height^2)  0.0831 5000 <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.38 on 12 degrees of freedom
Multiple R-Squared:  0.999,    Adjusted R-squared:  0.999
F-statistic: 1.14e+04 on 2 and 12 DF, p-value: <2e-16

```

As you can see, it's a simple matter to test these regressions using permutation tests and requires little change in the underlying code. The output is also similar to that produced by the `lm()` function. Note that an `Iter` column is added, indicating how many iterations were required to reach the stopping rule.

12.3.2 Multiple regression

In chapter 8, multiple regression was used to predict the murder rate based on population, illiteracy, income, and frost for 50 US states. Applying the `lmp()` function to this problem results in the following output.

Listing 12.4 Permutation tests for multiple regression

```

> library(lmPerm)
> set.seed(1234)
> states <- as.data.frame(state.x77)
> fit <- lmp(Murder~Population + Illiteracy+Income+Frost,
  data=states, perm="Prob")
[1] "Settings: unique SS : numeric variables centered"
> summary(fit)

Call:
lmp(formula = Murder ~ Population + Illiteracy + Income + Frost,
  data = states, perm = "Prob")

Residuals:
  Min   1Q Median   3Q   Max
-4.79597 -1.64946 -0.08112  1.48150  7.62104

Coefficients:
      Estimate lter Pr(Prob)
Population 2.237e-04  51  1.0000
Illiteracy 4.143e+00 5000 0.0004 ***
Income    6.442e-05  51  1.0000
Frost     5.813e-04  51  0.8627
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.535 on 45 degrees of freedom
Multiple R-Squared: 0.567, Adjusted R-squared: 0.5285
F-statistic: 14.73 on 4 and 45 DF, p-value: 9.133e-08

```

Looking back to chapter 8, both Population and Illiteracy are significant ($p < 0.05$) when normal theory is used. Based on the permutation tests, the Population variable is no longer significant. When the two approaches don't agree, you should look at your data more carefully. It may be that the assumption of normality is untenable or that outliers are present.

12.3.3 One-way ANOVA and ANCOVA

Each of the analysis of variance designs discussed in chapter 9 can be performed via permutation tests. First, let's look at the one-way ANOVA problem considered in section 9.1 on the impact of treatment regimens on cholesterol reduction. The code and results are given in the next listing.

Listing 12.5 Permutation test for one-way ANOVA

```

> library(lmPerm)
> library(multcomp)
> set.seed(1234)
> fit <- aovp(response~trt, data=cholesterol, perm="Prob")
[1] "Settings: unique SS "
> anova(fit)
Component 1 :
      Df R Sum Sq R Mean Sq lter Pr(Prob)
trt    4 1351.37  337.84 5000 < 2.2e-16 ***

```

```
Residuals 45 468.75 10.42
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The results suggest that the treatment effects are not all equal.

This second example in this section applies a permutation test to a one-way analysis of covariance. The problem is from chapter 9, where you investigated the impact of four drug doses on the litter weights of rats, controlling for gestation times. The next listing shows the permutation test and results.

Listing 12.6 Permutation test for one-way ANCOVA

```
> library(lmPerm)
> set.seed(1234)
> fit <- aovp(weight ~ gesttime + dose, data=litter, perm="Prob")
[1] "Settings: unique SS : numeric variables centered"
> anova(fit)
Component 1 :
      Df R Sum Sq R Mean Sq Iter Pr(Prob)
gesttime  1 161.49 161.493 5000 0.0006 ***
dose      3 137.12  45.708 5000 0.0392 *
Residuals 69 1151.27 16.685
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Based on the p-values, the four drug doses don't equally impact litter weights, controlling for gestation time.

12.3.4 Two-way ANOVA

You'll end this section by applying permutation tests to a factorial design. In chapter 9, you examined the impact of vitamin C on the tooth growth in guinea pigs. The two manipulated factors were dose (three levels) and delivery method (two levels). Ten guinea pigs were placed in each treatment combination, resulting in a balanced 3×2 factorial design. The permutation tests are provided in the next listing.

Listing 12.7 Permutation test for two-way ANOVA

```
> library(lmPerm)
> set.seed(1234)
> fit <- aovp(len~supp*dose, data=ToothGrowth, perm="Prob")
[1] "Settings: unique SS : numeric variables centered"
> anova(fit)
Component 1 :
      Df R Sum Sq R Mean Sq Iter Pr(Prob)
supp   1 205.35  205.35 5000 < 2e-16 ***
dose   1 2224.30 2224.30 5000 < 2e-16 ***
supp:dose 1  88.92  88.92 2032 0.04724 *
Residuals 56 933.63 16.67
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

At the .05 level of significance, all three effects are statistically different from zero. At the .01 level, only the main effects are significant.

It's important to note that when `aovp()` is applied to ANOVA designs, it defaults to unique sums of squares (also called *SAS Type III sums of squares*). Each effect is adjusted for every other effect. The default for parametric ANOVA designs in R is sequential sums of squares (*SAS Type I sums of squares*). Each effect is adjusted for those that appear *earlier* in the model. For balanced designs, the two approaches will agree, but for unbalanced designs with unequal numbers of observations per cell, they won't. The greater the imbalance, the greater the disagreement. If desired, specifying `seqs=TRUE` in the `aovp()` function will produce sequential sums of squares. For more on Type I and Type III sums of squares, see section 9.2.

12.4 Additional comments on permutation tests

Permutation tests provide a powerful alternative to tests that rely on a knowledge of the underlying sampling distribution. In each of the permutation tests described, you were able to test statistical hypotheses without recourse to the normal, t, F, or chi-square distributions.

You may have noticed how closely the results of the tests based on normal theory agreed with the results of the permutation approach in previous sections. The data in these problems were well behaved, and the agreement between methods is a testament to how well normal-theory methods work in such cases.

Permutation tests really shine in cases where the data are clearly non-normal (for example, highly skewed), outliers are present, samples sizes are small, or no parametric tests exist. But if the original sample is a poor representation of the population of interest, no test, including permutation tests, will improve the inferences generated.

Permutation tests are primarily useful for generating p-values that can be used to test null hypotheses. They can help answer the question, "Does an effect exist?" It's more difficult to use permutation methods to obtain confidence intervals and estimates of measurement precision. Fortunately, this is an area in which bootstrapping excels.

12.5 Bootstrapping

Bootstrapping generates an empirical distribution of a test statistic or set of test statistics by repeated random sampling with replacement from the original sample. It allows you to generate confidence intervals and test statistical hypotheses without having to assume a specific underlying theoretical distribution.

It's easiest to demonstrate the logic of bootstrapping with an example. Say that you want to calculate the 95% confidence interval for a sample mean. Your sample has 10 observations, a sample mean of 40, and a sample standard deviation of 5. If you're willing to assume that the sampling distribution of the mean is normally distributed, the $(1 - \alpha/2)\%$ confidence interval can be calculated using

$$\bar{X} - t \frac{s}{\sqrt{n}} < \mu < \bar{X} + t \frac{s}{\sqrt{n}}$$

where t is the upper $1-\alpha/2$ critical value for a t distribution with $n - 1$ degrees of freedom. For a 95% confidence interval, you have $40 - 2.262(5/3.163) < \mu < 40 + 2.262(5/3.163)$ or $36.424 < \mu < 43.577$. You'd expect 95% of confidence intervals created in this way to surround the true population mean.

But what if you aren't willing to assume that the sampling distribution of the mean is normally distributed? You can use a bootstrapping approach instead:

1. Randomly select 10 observations from the sample, with replacement after each selection. Some observations may be selected more than once, and some may not be selected at all.
2. Calculate and record the sample mean.
3. Repeat the first two steps 1,000 times.
4. Order the 1,000 sample means from smallest to largest.
5. Find the sample means representing the 2.5th and 97.5th percentiles. In this case, it's the 25th number from the bottom and top. These are your 95% confidence limits.

In the present case, where the sample mean is likely to be normally distributed, you gain little from the bootstrap approach. Yet there are many cases where the bootstrap approach is advantageous. What if you wanted confidence intervals for the sample median, or the difference between two sample medians? There are no simple normal-theory formulas here, and bootstrapping is the approach of choice. If the underlying distributions are unknown, if outliers are a problem, if sample sizes are small, or if parametric approaches don't exist, bootstrapping can often provide a useful method of generating confidence intervals and testing hypotheses.

12.6 Bootstrapping with the boot package

The `boot` package provides extensive facilities for bootstrapping and related resampling methods. You can bootstrap a single statistic (for example, a median) or a vector of statistics (for example, a set of regression coefficients). Be sure to download and install the `boot` package before first use:

```
install.packages("boot")
```

The bootstrapping process will seem complicated, but once you review the examples it should make sense.

In general, bootstrapping involves three main steps:

1. Write a function that returns the statistic or statistics of interest. If there is a single statistic (for example, a median), the function should return a number. If there is a set

of statistics (for example, a set of regression coefficients), the function should return a vector.

2. Process this function through the `boot()` function in order to generate R bootstrap replications of the statistic(s).
3. Use the `boot.ci()` function to obtain confidence intervals for the statistic(s) generated in step 2.

Now to the specifics.

The main bootstrapping function is `boot()`. It has the format

```
bootobject <- boot(data=, statistic=, R=, ...)
```

The parameters are described in table 12.3.

Table 12.3 Parameters of the `boot()` function

Parameter	Description
<code>data</code>	A vector, matrix, or data frame.
<code>statistic</code>	A function that produces the k statistics to be bootstrapped (k=1 if bootstrapping a single statistic). The function should include an indices parameter that the <code>boot()</code> function can use to select cases for each replication (see the examples in the text).
<code>R</code>	Number of bootstrap replicates.
<code>...</code>	Additional parameters to be passed to the function that produces the statistic of interest.

The `boot()` function calls the `statistic` function `R` times. Each time, it generates a set of random indices, with replacement, from the integers `1:nrow(data)`. These indices are used in the `statistic` function to select a sample. The statistics are calculated on the sample, and the results are accumulated in `bootobject`. The `bootobject` structure is described in table 12.4.

Table 12.4 Elements of the object returned by the `boot()` function

Element	Description
<code>t0</code>	The observed values of k statistics applied to the original data
<code>t</code>	An $R \times k$ matrix, where each row is a bootstrap replicate of the k statistics

You can access these elements as `bootobject$t0` and `bootobject$t`.

Once you generate the bootstrap samples, you can use `print()` and `plot()` to examine the results. If the results look reasonable, you can use the `boot.ci()` function to obtain confidence intervals for the statistic(s). The format is

```
boot.ci(bootobject, conf=, type=)
```

The parameters are given in table 12.5.

Table 12.5 Parameters of the `boot.ci()` function

Parameter	Description
<code>bootobject</code>	The object returned by the <code>boot()</code> function.
<code>conf</code>	The desired confidence interval (default: <code>conf=0.95</code>).
<code>type</code>	The type of confidence interval returned. Possible values are <code>norm</code> , <code>basic</code> , <code>stud</code> , <code>perc</code> , <code>bca</code> , and <code>all</code> (default: <code>type="all"</code>)

The `type` parameter specifies the method for obtaining the confidence limits. The `perc` method (percentile) was demonstrated in the sample mean example. `bca` provides an interval that makes simple adjustments for bias. I find `bca` preferable in most circumstances. See Mooney and Duval (1993) for an introduction to these methods.

In the remaining sections, we'll look at bootstrapping a single statistic and a vector of statistics.

12.6.1 Bootstrapping a single statistic

The `mtcars` dataset contains information on 32 automobiles reported in the 1974 *Motor Trend* magazine. Suppose you're using multiple regression to predict miles per gallon from a car's weight (lb/1,000) and engine displacement (cu. in.). In addition to the standard regression statistics, you'd like to obtain a 95% confidence interval for the R-squared value (the percent of variance in the response variable explained by the predictors). The confidence interval can be obtained using nonparametric bootstrapping.

The first task is to write a function for obtaining the R-squared value:

```
rsq <- function(formula, data, indices) {
  d <- data[indices,]
  fit <- lm(formula, data=d)
  return(summary(fit)$r.square)
}
```

The function returns the R-squared value from a regression. The `d <- data[indices,]` statement is required for `boot()` to be able to select samples.

You can then draw a large number of bootstrap replications (say, 1,000) with the following code:

```
library(boot)
set.seed(1234)
results <- boot(data=mtcars, statistic=rsq,
               R=1000, formula=mpg~wt+disp)
```

The `boot` object can be printed using

```
> print(results)

ORDINARY NONPARAMETRIC BOOTSTRAP

Call:
boot(data = mtcars, statistic = rsq, R = 1000, formula = mpg ~
      wt + disp)

Bootstrap Statistics :
   original    bias std. error
t1* 0.7809306 0.01333670 0.05068926
```

and plotted using `plot(results)`. The resulting graph is shown in figure 12.2.

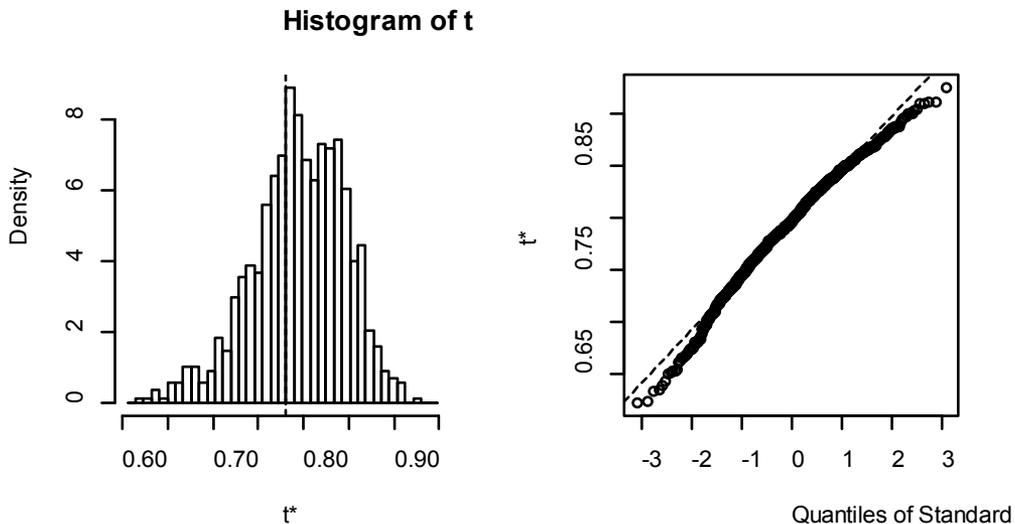


Figure 12.2 Distribution of bootstrapped R-squared values

In figure 12.2, you can see that the distribution of bootstrapped R-squared values isn't normally distributed. A 95% confidence interval for the R-squared values can be obtained using

```
> boot.ci(results, type=c("perc", "bca"))
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 1000 bootstrap replicates

CALL :
boot.ci(boot.out = results, type = c("perc", "bca"))

Intervals :
Level Percentile  BCa
95% ( 0.6838, 0.8833 ) ( 0.6344, 0.8549 )
Calculations and Intervals on Original Scale
Some BCa intervals may be unstable
```

You can see from this example that different approaches to generating the confidence intervals can lead to different intervals. In this case, the bias-adjusted interval is moderately different from the percentile method. In either case, the null hypothesis $H_0: R\text{-square} = 0$ would be rejected, because zero is outside the confidence limits.

In this section, you estimated the confidence limits of a single statistic. In the next section, you'll estimate confidence intervals for several statistics.

12.6.2 Bootstrapping several statistics

In the previous example, bootstrapping was used to estimate the confidence interval for a single statistic (R-squared). Continuing the example, let's obtain the 95% confidence intervals for a vector of statistics. Specifically, let's get confidence intervals for the three model regression coefficients (intercept, car weight, and engine displacement).

First, create a function that returns the vector of regression coefficients:

```
bs <- function(formula, data, indices) {
  d <- data[indices,]
  fit <- lm(formula, data=d)
  return(coef(fit))
}
```

Then use this function to bootstrap 1,000 replications:

```
library(boot)
set.seed(1234)
results <- boot(data=mtcars, statistic=bs,
               R=1000, formula=mpg~wt+disp)
> print(results)
ORDINARY NONPARAMETRIC BOOTSTRAP
Call:
boot(data = mtcars, statistic = bs, R = 1000, formula = mpg ~
      wt + disp)

Bootstrap Statistics :
   original  bias  std. error
t1*  34.9606  0.137873   2.48576
t2*  -3.3508 -0.053904   1.17043
t3*  -0.0177 -0.000121   0.00879
```

When bootstrapping multiple statistics, add an index parameter to the `plot()` and `boot.ci()` functions to indicate which column of `bootobject$t` to analyze. In this example, index 1 refers to the intercept, index 2 is car weight, and index 3 is the engine displacement. To plot the results for car weight, use

```
plot(results, index=2)
```

The graph is given in figure 12.3.

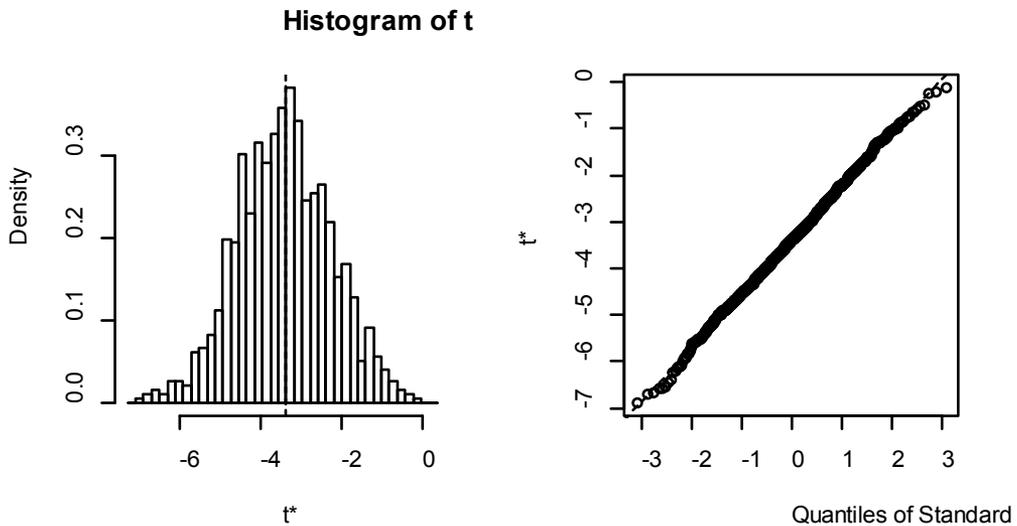


Figure 12.3 Distribution of bootstrapping regression coefficients for car weight

To get the 95% confidence intervals for car weight and engine displacement, use

```
> boot.ci(results, type="bca", index=2)
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 1000 bootstrap replicates
```

```
CALL :
boot.ci(boot.out = results, type = "bca", index = 2)
```

```
Intervals :
Level BCa
95% (-5.66, -1.19 )
Calculations and Intervals on Original Scale
```

```
> boot.ci(results, type="bca", index=3)
```

```
BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
Based on 1000 bootstrap replicates
```

```
CALL :
boot.ci(boot.out = results, type = "bca", index = 3)
```

```
Intervals :
Level BCa
95% (-0.0331, 0.0010 )
Calculations and Intervals on Original Scale
```

NOTE The previous example resamples the entire sample of data each time. If you can assume that the predictor variables have fixed levels (typical in planned experiments), you'd do better to only resample residual terms. See Mooney and Duval (1993, pp. 16–17) for a simple explanation and algorithm.

Before we leave bootstrapping, it's worth addressing two questions that come up often:

- How large does the original sample need to be?
- How many replications are needed?

There's no simple answer to the first question. Some say that an original sample size of 20–30 is sufficient for good results, as long as the sample is representative of the population. Random sampling from the population of interest is the most trusted method for assuring the original sample's representativeness. With regard to the second question, I find that 1,000 replications are more than adequate in most cases. Computer power is cheap, and you can always increase the number of replications if desired.

There are many helpful sources of information about permutation tests and bootstrapping. An excellent starting place is an online article by Yu (2003). Good (2006) provides a comprehensive overview of resampling in general and includes R code. A good, accessible introduction to bootstrapping is provided by Mooney and Duval (1993). The definitive source on bootstrapping is Efron and Tibshirani (1998). Finally, there are a number of great online resources, including Simon (1997), Canty (2002), Shah (2005), and Fox (2002).

12.7 Summary

- Resampling statistics and bootstrapping are computer-intensive methods that allow you to test hypotheses and form confidence intervals without reference to a known theoretical distribution.
- They are particularly valuable when your data comes from unknown population distributions, when there are serious outliers, when your sample sizes are small, and when there are no existing parametric methods to answer the hypotheses of interest.
- They are particularly exciting because they provide an avenue for answering questions when your standard data assumptions are clearly untenable or when you have no other idea how to approach the problem.
- However, they aren't a panaceas. They can't turn bad data into good data. If your original samples aren't representative of the population of interest or are too small to accurately reflect it, then these techniques won't help.

13

Generalized linear models

This chapter covers

- **Formulating a generalized linear model**
- **Predicting categorical outcomes**
- **Modeling count data**

In chapters 8 (regression) and 9 (ANOVA), we explored linear models that can be used to predict a normally distributed response variable from a set of continuous and/or categorical predictor variables. But there are many situations in which it's unreasonable to assume that the dependent variable is normally distributed (or even continuous). For example:

- The outcome variable may be categorical. Binary variables (for example, yes/no, passed/failed, lived/died) and polytomous variables (for example, poor/good/excellent, republican/democrat/independent) clearly aren't normally distributed.
- The outcome variable may be a count (for example, number of traffic accidents in a week, number of drinks per day). Such variables take on a limited number of values and are never negative. Additionally, their mean and variance are often related (which isn't true for normally distributed variables).

Generalized linear models extend the linear-model framework to include dependent variables that are decidedly non-normal.

In this chapter, we'll start with a brief overview of generalized linear models and the `glm()` function used to estimate them. Then we'll focus on two popular models in this framework: *logistic regression* (where the dependent variable is categorical) and *Poisson regression* (where the dependent variable is a count variable).

To motivate the discussion, you'll apply generalized linear models to two research questions that aren't easily addressed with standard linear models:

- What personal, demographic, and relationship variables predict marital infidelity? In this case, the outcome variable is binary (affair/no affair).
- What impact does a drug treatment for seizures have on the number of seizures experienced over an eight-week period? In this case, the outcome variable is a count (number of seizures).

You'll apply logistic regression to address the first question and Poisson regression to address the second. Along the way, we'll consider extensions of each technique.

13.1 Generalized linear models and the `glm()` function

A wide range of popular data-analytic methods are subsumed within the framework of the generalized linear model. In this section, we'll briefly explore some of the theory behind this approach. You can safely skip this section if you like and come back to it later.

Let's say that you want to model the relationship between a response variable Y and a set of p predictor variables $X_1 \dots X_p$. In the standard linear model, you assume that Y is normally distributed and that the form of the relationship is

$$\mu_y = \beta_0 + \sum_{j=1}^p \beta_j X_j$$

This equation states that the conditional mean of the response variable is a linear combination of the predictor variables. The β_j are the parameters specifying the expected change in Y for a unit change in X_j , and β_0 is the expected value of Y when all the predictor variables are 0. You're saying that you can predict the mean of the Y distribution for observations with a given set of X values by applying the proper weights to the X variables and adding them up.

Note that you've made no distributional assumptions about the predictor variables, X_j . Unlike Y , there's no requirement that they be normally distributed. In fact, they're often categorical (for example, ANOVA designs). Additionally, nonlinear functions of the predictors are allowed. You often include such predictors as X^2 or $X_1 \times X_2$. What is important is that the equation is linear in the parameters ($\beta_0, \beta_1, \dots, \beta_p$).

In generalized linear models, you fit models of the form

$$g(\mu_y) = \beta_0 + \sum_{j=1}^p \beta_j X_j$$

where $g(\mu_y)$ is a function of the conditional mean (called the *link function*). Additionally, you relax the assumption that Y is normally distributed. Instead, you assume that Y follows a distribution that's a member of the exponential family. You specify the link function and the probability distribution, and the parameters are derived through an iterative maximum-likelihood-estimation procedure.

13.1.1 The `glm()` function

Generalized linear models are typically fit in R through the `glm()` function (although other specialized functions are available). The form of the function is similar to `lm()` but includes additional parameters. The basic format of the function is

```
glm(formula, family=family(link=function), data=)
```

where the probability distribution (*family*) and corresponding default link function (*function*) are given in table 13.1.

Table 13.1 `glm()` parameters

Family	Default link function
binomial	(link = "logit")
gaussian	(link = "identity")
gamma	(link = "inverse")
inverse.gaussian	(link = "1/mu^2")
poisson	(link = "log")
quasi	(link = "identity", variance = "constant")
quasibinomial	(link = "logit")
quasipoisson	(link = "log")

The `glm()` function allows you to fit a number of popular models, including logistic regression, Poisson regression, and survival analysis (not considered here). You can demonstrate this for the first two models as follows. Assume that you have a single response variable (Y), three predictor variables (X_1 , X_2 , X_3), and a data frame (`mydata`) containing the data.

Logistic regression is applied to situations in which the response variable is dichotomous (0 or 1). The model assumes that Y follows a binomial distribution and that you can fit a linear model of the form

$$\log_e \left(\frac{\pi}{1 - \pi} \right) = \beta_0 + \sum_{j=1}^p \beta_j X_j$$

where $\pi = \mu_Y$ is the conditional mean of Y (that is, the probability that $Y = 1$ given a set of X values), $(\pi/1 - \pi)$ is the odds that $Y = 1$, and $\log(\pi/1 - \pi)$ is the log odds, or *logit*. In this case, $\log(\pi/1 - \pi)$ is the link function, the probability distribution is binomial, and the logistic regression model can be fit using

```
glm(Y~X1+X2+X3, family=binomial(link="logit"), data=mydata)
```

Logistic regression is described more fully in section 13.2.

Poisson regression is applied to situations in which the response variable is the number of events to occur in a given period of time. The Poisson regression model assumes that Y follows a Poisson distribution and that you can fit a linear model of the form

$$\log_e (\lambda) = \beta_0 + \sum_{j=1}^p \beta_j X_j$$

where λ is the mean (and variance) of Y . In this case, the link function is $\log(\lambda)$, the probability distribution is Poisson, and the Poisson regression model can be fit using

```
glm(Y~X1+X2+X3, family=poisson(link="log"), data=mydata)
```

Poisson regression is described in section 13.3.

It's worth noting that the standard linear model is also a special case of the generalized linear model. If you let the link function $g(\mu_Y) = \mu_Y$ or the identity function and specify that the probability distribution is normal (Gaussian), then

```
glm(Y~X1+X2+X3, family=gaussian(link="identity"), data=mydata)
```

would produce the same results as

```
lm(Y~X1+X2+X3, data=mydata)
```

To summarize, generalized linear models extend the standard linear model by fitting a *function* of the conditional mean response (rather than the conditional mean response) and assuming that the response variable follows a member of the *exponential* family of distributions (rather than being limited to the normal distribution). The parameter estimates are derived via maximum likelihood rather than least squares.

13.1.2 Supporting functions

Many of the functions that you used in conjunction with `lm()` when analyzing standard linear models have corresponding versions for `glm()`. Some commonly used functions are given in table 13.2.

Table 13.2 Functions that support `glm()`

Function	Description
<code>summary()</code>	Displays detailed results for the fitted model
<code>coefficients()</code> , <code>coef()</code>	Lists the model parameters (intercept and slopes) for the fitted model
<code>confint()</code>	Provides confidence intervals for the model parameters (95% by default)
<code>residuals()</code>	Lists the residual values in a fitted model
<code>anova()</code>	Generates an ANOVA table comparing two fitted models
<code>plot()</code>	Generates diagnostic plots for evaluating the fit of a model
<code>predict()</code>	Uses a fitted model to predict response values for a new dataset
<code>deviance()</code>	Deviance for the fitted model
<code>df.residual()</code>	Residual degrees of freedom for the fitted model

We'll explore examples of these functions in later sections. In the next section, we'll briefly consider the assessment of model adequacy.

13.1.3 Model fit and regression diagnostics

The assessment of model adequacy is as important for generalized linear models as it is for standard (OLS) linear models. Unfortunately, there's less agreement in the statistical

community regarding appropriate assessment procedures. In general, you can use the techniques described in chapter 8, with the following caveats.

When assessing model adequacy, you'll typically want to plot predicted values expressed in the metric of the original response variable against residuals of the deviance type. For example, a common diagnostic plot would be

```
plot(predict(model, type="response"),
      residuals(model, type="deviance"))
```

where `model` is the object returned by the `glm()` function.

The hat values, studentized residuals, and Cook's D statistics that R provides will be approximate values. Additionally, there's no general consensus on cutoff values for identifying problematic observations. Values have to be judged relative to each other. One approach is to create index plots for each statistic and look for unusually large values. For example, you could use the following code to create three diagnostic plots:

```
plot(hatvalues(model))
plot(rstudent(model))
plot(cooks.distance(model))
```

Alternatively, you could use the code

```
library(car)
influencePlot(model)
```

to create one omnibus plot. In the latter graph, the horizontal axis is the leverage, the vertical axis is the studentized residual, and the plotted symbol is proportional to the Cook's distance.

Diagnostic plots tend to be most helpful when the response variable takes on many values. When the response variable can only take on a limited number of values (for example, logistic regression), the utility of these plots is decreased.

For more on regression diagnostics for generalized linear models, see Fox (2008) and Faraway (2006). In the remaining portion of this chapter, we'll consider two of the most popular forms of the generalized linear model in detail: logistic regression and Poisson regression.

13.2 Logistic regression

Logistic regression is useful when you're predicting a binary outcome from a set of continuous and/or categorical predictor variables. To demonstrate this, let's explore the data on infidelity contained in the data frame `Affairs`, provided with the AER package. Be sure to download and install the package (using `install.packages("AER")`) before first use.

The infidelity data, known as Fair's Affairs, is based on a cross-sectional survey conducted by *Psychology Today* in 1969 and is described in Greene (2003) and Fair (1978). It contains 9 variables collected on 601 participants and includes how often the respondent engaged in extramarital sexual intercourse during the past year, as well as their gender, age, years married, whether they had children, their religiousness (on a 5-point scale from 1=anti to

5=very), education, occupation (Hollingshead 7-point classification with reverse numbering), and a numeric self-rating of their marriage (from 1=very unhappy to 5=very happy).

Let's look at some descriptive statistics:

```
> data(Affairs, package="AER")
> summary(Affairs)
  affairs  gender  age  yearsmarried  children
Min. : 0.000  female:315  Min. :17.50  Min. : 0.125  no :171
1st Qu.: 0.000  male :286  1st Qu.:27.00  1st Qu.: 4.000  yes:430
Median : 0.000          Median :32.00  Median : 7.000
Mean   : 1.456          Mean   :32.49  Mean   : 8.178
3rd Qu.: 0.000          3rd Qu.:37.00  3rd Qu.:15.000
Max.   :12.000          Max.   :57.00  Max.   :15.000
religiousness  education  occupation  rating
Min. :1.000  Min. : 9.00  Min. :1.000  Min. :1.000
1st Qu.:2.000  1st Qu.:14.00  1st Qu.:3.000  1st Qu.:3.000
Median :3.000  Median :16.00  Median :5.000  Median :4.000
Mean   :3.116  Mean   :16.17  Mean   :4.195  Mean   :3.932
3rd Qu.:4.000  3rd Qu.:18.00  3rd Qu.:6.000  3rd Qu.:5.000
Max.   :5.000  Max.  :20.00  Max.  :7.000  Max.  :5.000

> table(Affairs$affairs)
 0  1  2  3  7 12
451 34 17 19 42 38
```

From these statistics, you can see that that 52% of respondents were female, that 72% had children, and that the median age for the sample was 32 years. With regard to the response variable, 75% of respondents reported not engaging in an infidelity in the past year (451/601). The largest number of encounters reported was 12 (6%).

Although the *number* of indiscretions was recorded, your interest here is in the binary outcome (had an affair/didn't have an affair). You can transform affairs into a dichotomous factor called `ynaffair` with the following code.

```
> Affairs$ynaffair <- ifelse(Affairs$affairs > 0, 1, 0)
> Affairs$ynaffair <- factor(Affairs$ynaffair,
  levels=c(0,1),
  labels=c("No","Yes"))
> table(Affairs$ynaffair)
No Yes
451 150
```

This dichotomous factor can now be used as the outcome variable in a logistic regression model:

```
> fit.full <- glm(ynaffair ~ gender + age + yearsmarried + children +
  religiousness + education + occupation +rating,
  data=Affairs, family=binomial())
> summary(fit.full)

Call:
glm(formula = ynaffair ~ gender + age + yearsmarried + children +
  religiousness + education + occupation + rating, family = binomial(),
  data = Affairs)
```

```

Deviance Residuals:
  Min   1Q   Median   3Q   Max
-1.571 -0.750 -0.569 -0.254  2.519

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.3773    0.8878   1.55 0.12081
gendermale   0.2803    0.2391   1.17 0.24108
age          -0.0443    0.0182  -2.43 0.01530 *
yearsmarried 0.0948    0.0322   2.94 0.00326 **
childrenyes  0.3977    0.2915   1.36 0.17251
religiousness -0.3247    0.0898  -3.62 0.00030 ***
education    0.0211    0.0505   0.42 0.67685
occupation   0.0309    0.0718   0.43 0.66663
rating       -0.4685    0.0909  -5.15 2.6e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 675.38 on 600 degrees of freedom
Residual deviance: 609.51 on 592 degrees of freedom
AIC: 627.5

Number of Fisher Scoring iterations: 4

```

From the p-values for the regression coefficients (last column), you can see that gender, presence of children, education, and occupation may not make a significant contribution to the equation (you can't reject the hypothesis that the parameters are 0). Let's fit a second equation without them and test whether this reduced model fits the data as well:

```

> fit.reduced <- glm(yNAffair ~ age + yearsmarried + religiousness +
  rating, data=Affairs, family=binomial())
> summary(fit.reduced)
Call:
glm(formula = yNAffair ~ age + yearsmarried + religiousness + rating,
    family = binomial(), data = Affairs)

Deviance Residuals:
  Min   1Q   Median   3Q   Max
-1.628 -0.755 -0.570 -0.262  2.400

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  1.9308    0.6103   3.16 0.00156 **
age          -0.0353    0.0174  -2.03 0.04213 *
yearsmarried 0.1006    0.0292   3.44 0.00057 ***
religiousness -0.3290    0.0895  -3.68 0.00023 ***
rating       -0.4614    0.0888  -5.19 2.1e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

```

```
Null deviance: 675.38 on 600 degrees of freedom
Residual deviance: 615.36 on 596 degrees of freedom
AIC: 625.4
```

```
Number of Fisher Scoring iterations: 4
```

Each regression coefficient in the reduced model is statistically significant ($p < .05$). Because the two models are nested (`fit.reduced` is a subset of `fit.full`), you can use the `anova()` function to compare them. For generalized linear models, you'll want a chi-square version of this test:

```
> anova(fit.reduced, fit.full, test="Chisq")
Analysis of Deviance Table

Model 1: ynaffair ~ age + yearsmarried + religiousness + rating
Model 2: ynaffair ~ gender + age + yearsmarried + children +
  religiousness + education + occupation + rating
Resid. Df Resid. Dev Df Deviance P(>|Chi|)
1    596     615
2    592     610 4    5.85    0.21
```

The nonsignificant chi-square value ($p = 0.21$) suggests that the reduced model with four predictors fits as well as the full model with nine predictors, reinforcing your belief that gender, children, education, and occupation don't add significantly to the prediction above and beyond the other variables in the equation. Therefore, you can base your interpretations on the simpler model.

13.2.1 Interpreting the model parameters

Let's look at the regression coefficients:

```
> coef(fit.reduced)
(Intercept)    age yearsmarried religiousness    rating
    1.931    -0.035    0.101    -0.329    -0.461
```

In a logistic regression, the response being modeled is the $\log(\text{odds})$ that $Y = 1$. The regression coefficients give the change in $\log(\text{odds})$ in the response for a unit change in the predictor variable, holding all other predictor variables constant.

Because $\log(\text{odds})$ are difficult to interpret, you can exponentiate them to put the results on an odds scale:

```
> exp(coef(fit.reduced))
(Intercept)    age yearsmarried religiousness    rating
    6.895    0.965    1.106    0.720    0.630
```

Now you can see that the odds of an extramarital encounter are increased by a factor of 1.106 for a one-year increase in years married (holding age, religiousness, and marital rating constant). Conversely, the odds of an extramarital affair are multiplied by a factor of 0.965 for every year increase in age. The odds of an extramarital affair increase with years married and

decrease with age, religiousness, and marital rating. Because the predictor variables can't equal 0, the intercept isn't meaningful in this case.

If desired, you can use the `confint()` function to obtain confidence intervals for the coefficients. For example, `exp(confint(fit.reduced))` would print 95% confidence intervals for each of the coefficients on an odds scale.

Finally, a one-unit change in a predictor variable may not be inherently interesting. For binary logistic regression, the change in the odds of the higher value on the response variable for an n unit change in a predictor variable is $\exp(\beta_j)^n$. If a one-year increase in years married multiplies the odds of an affair by 1.106, a 10-year increase would increase the odds by a factor of 1.106^{10} , or 2.7, holding the other predictor variables constant.

13.2.2 Assessing the impact of predictors on the probability of an outcome

For many of us, it's easier to think in terms of probabilities than odds. You can use the `predict()` function to observe the impact of varying the levels of a predictor variable on the probability of the outcome. The first step is to create an artificial dataset containing the values of the predictor variables you're interested in. Then you can use this artificial dataset with the `predict()` function to predict the probabilities of the outcome event occurring for these values.

Let's apply this strategy to assess the impact of marital ratings on the probability of having an extramarital affair. First, create an artificial dataset where age, years married, and religiousness are set to their means, and marital rating varies from 1 to 5:

```
> testdata <- data.frame(rating=c(1, 2, 3, 4, 5), age=mean(Affairs$age),
                        yearsmarried=mean(Affairs$yearsmarried),
                        religiousness=mean(Affairs$religiousness))
> testdata
  rating age yearsmarried religiousness
1     1 32.5      8.18      3.12
2     2 32.5      8.18      3.12
3     3 32.5      8.18      3.12
4     4 32.5      8.18      3.12
5     5 32.5      8.18      3.12
```

Next, use the test dataset and prediction equation to obtain probabilities:

```
> testdata$prob <- predict(fit.reduced, newdata=testdata, type="response")
testdata
  rating age yearsmarried religiousness prob
1     1 32.5      8.18      3.12 0.530
2     2 32.5      8.18      3.12 0.416
3     3 32.5      8.18      3.12 0.310
4     4 32.5      8.18      3.12 0.220
5     5 32.5      8.18      3.12 0.151
```

From these results, you see that the probability of an extramarital affair decreases from 0.53 when the marriage is rated 1=very unhappy to 0.15 when the marriage is rated 5=very happy (holding age, years married, and religiousness constant). Now look at the impact of age:

```

> testdata <- data.frame(rating=mean(Affairs$rating),
                        age=seq(17, 57, 10),
                        yearsmarried=mean(Affairs$yearsmarried),
                        religiousness=mean(Affairs$religiousness))
> testdata
  rating age yearsmarried religiousness
1  3.93  17      8.18      3.12
2  3.93  27      8.18      3.12
3  3.93  37      8.18      3.12
4  3.93  47      8.18      3.12
5  3.93  57      8.18      3.12
> testdata$prob <- predict(fit.reduced, newdata=testdata, type="response")
> testdata
  rating age yearsmarried religiousness  prob
1  3.93  17      8.18      3.12  0.335
2  3.93  27      8.18      3.12  0.262
3  3.93  37      8.18      3.12  0.199
4  3.93  47      8.18      3.12  0.149
5  3.93  57      8.18      3.12  0.109

```

Here, you see that as age increases from 17 to 57, the probability of an extramarital encounter decreases from 0.34 to 0.11, holding the other variables constant. Using this approach, you can explore the impact of each predictor variable on the outcome.

13.2.3 Overdispersion

The expected variance for data drawn from a binomial distribution is

$$\sigma^2 = n\pi(1 - \pi)$$

where n is the number of observations and π is the probability of belonging to the $Y = 1$ group. *Overdispersion* occurs when the observed variance of the response variable is larger than what would be expected from a binomial distribution. Overdispersion can lead to distorted test standard errors and inaccurate tests of significance.

When overdispersion is present, you can still fit a logistic regression using the `glm()` function, but in this case, you should use the quasibinomial distribution rather than the binomial distribution.

One way to detect overdispersion is to compare the residual deviance with the residual degrees of freedom in your binomial model. If the ratio

$$\phi = \frac{\text{Residual deviance}}{\text{Residual df}}$$

is considerably larger than 1, you have evidence of overdispersion. Applying this to the `Affairs` example, you have

```
> deviance(fit.reduced)/df.residual(fit.reduced)
```

```
[1] 1.032
```

which is close to 1, suggesting no overdispersion.

You can also test for overdispersion. To do this, you fit the model twice, but in the first instance you use `family="binomial"` and in the second instance you use `family="quasibinomial"`. If the `glm()` object returned in the first case is called `fit` and the object returned in the second case is called `fit.od`, then

```
pchisq(summary(fit.od)$dispersion * fit$df.residual,
        fit$df.residual, lower = F)
```

provides the p-value for testing the null hypothesis $H_0: \Phi = 1$ versus the alternative hypothesis $H_1: \Phi \neq 1$. If `p` is small (say, less than 0.05), you'd reject the null hypothesis.

Applying this to the `Affairs` dataset, you have

```
> fit <- glm(ynaffair ~ age + yearsmarried + religiousness +
            rating, family = binomial(), data = Affairs)
> fit.od <- glm(ynaffair ~ age + yearsmarried + religiousness +
              rating, family = quasibinomial(), data = Affairs)
> pchisq(summary(fit.od)$dispersion * fit$df.residual,
          fit$df.residual, lower = F)
```

```
[1] 0.34
```

The resulting p-value (0.34) is clearly not significant ($p > 0.05$), strengthening your belief that overdispersion isn't a problem. We'll return to the issue of overdispersion when we discuss Poisson regression.

13.2.4 Extensions

Several logistic regression extensions and variations are available in R:

- *Robust logistic regression*—The `glmRob()` function in the `robustbase` package can be used to fit a robust generalized linear model, including robust logistic regression. Robust logistic regression can be helpful when fitting logistic regression models to data containing outliers and influential observations.
- *Multinomial logistic regression*—If the response variable has more than two unordered categories (for example, married/widowed/divorced), you can fit a polytomous logistic regression using the `mlogit()` function in the `mlogit` package. Alternatively, you can use the `multinom()` function in the `nnet` package.
- *Ordinal logistic regression*—If the response variable is a set of ordered categories (for example, credit risk as poor/good/excellent), you can fit an ordinal logistic regression using the `polyr()` function in the `MASS` package.

The ability to model a response variable with multiple categories (both ordered and unordered) is an important extension, but it comes at the expense of greater interpretive complexity. Assessing model fit and regression diagnostics in these cases will also be more complex.

In the `Affairs` example, the number of extramarital contacts was dichotomized into a yes/no response variable because our interest centered on whether respondents had an affair in the past year. If our interest had been centered on magnitude—the number of encounters in the past year—we would have analyzed the count data directly. One popular approach to analyzing count data is Poisson regression, the next topic we'll address.

13.3 Poisson regression

Poisson regression is useful when you're predicting an outcome variable representing counts from a set of continuous and/or categorical predictor variables. A comprehensive yet accessible introduction to Poisson regression is provided by Cox, West, and Aiken (2009).

To illustrate the fitting of a Poisson regression model, along with some issues that can come up in the analysis, we'll use the Breslow seizure data (Breslow, 1993) provided in the `robustbase` package. Specifically, we'll consider the impact of an antiepileptic drug treatment on the number of seizures occurring over an eight-week period following the initiation of therapy. Be sure to install the `robustbase` package before continuing.

Data were collected on the age and number of seizures reported by patients suffering from simple or complex partial seizures during an eight-week period before, and eight-week period after, randomization into a drug or placebo condition. `Ysum` (the number of seizures in the eight-week period post-randomization) is the response variable. Treatment condition (`Trt`), age in years (`Age`), and number of seizures reported in the baseline eight-week period (`Base`) are the predictor variables. The baseline number of seizures and age are included because of their potential effect on the response variable. We're interested in whether or not evidence exists that the drug treatment decreases the number of seizures after accounting for these covariates.

First, let's look at summary statistics for the dataset:

```
> data(epilepsy, package="robustbase")
> names(epilepsy)
[1] "ID" "Y1" "Y2" "Y3" "Y4" "Base" "Age" "Trt" "Ysum"
[10] "Age10" "Base4"

> summary(breslow.dat[6:9])
  Base   Age   Trt   Ysum
Min. : 6.0 Min. :18.0 placebo :28 Min. : 0.0
1st Qu.: 12.0 1st Qu.:23.0 progabide:31 1st Qu.: 11.5
Median : 22.0 Median :28.0           Median : 16.0
Mean   : 31.2 Mean   :28.3           Mean   : 33.1
3rd Qu.: 41.0 3rd Qu.:32.0           3rd Qu.: 36.0
Max.   :151.0 Max.   :42.0           Max.   :302.0
```

Note that although there are 11 variables in the dataset, we're limiting our attention to the 4 described earlier. Both the baseline and post-randomization number of seizures are highly skewed. Let's look at the response variable in more detail. The following code produces the graphs in figure 13.1:

```
library(ggplot2)
```

```
ggplot(epilepsy, aes(x=Ysum)) +
  geom_histogram(color="black", fill="white") +
  labs(title="Distribution of seizures",
       x="Seizure Count",
       y="Frequency") +
  theme_bw()
ggplot(epilepsy, aes(x=Trt, y=Ysum)) +
  geom_boxplot() +
  labs(title="Group comparisons", x="", y="") +
  theme_bw()
```

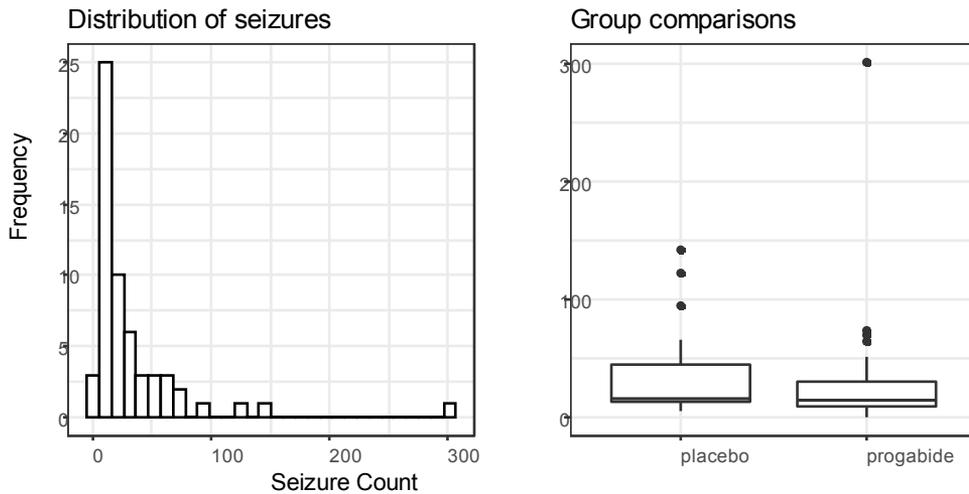


Figure 13.1 Distribution of post-treatment seizure counts (source: Breslow seizure data)

You can clearly see the skewed nature of the dependent variable and the possible presence of outliers. At first glance, the number of seizures in the drug condition appears to be smaller and has a smaller variance. (You'd expect a smaller variance to accompany a smaller mean with Poisson distributed data.) Unlike standard OLS regression, this heterogeneity of variance isn't a problem in Poisson regression.

The next step is to fit the Poisson regression:

```
> fit <- glm(Ysum ~ Base + Age + Trt, data=epilepsy, family=poisson())
> summary(fit)
```

```
Call:
glm(formula = Ysum ~ Base + Age + Trt, family = poisson(), data = epilepsy)
```

```
Deviance Residuals:
   Min    1Q  Median    3Q   Max
-6.057 -2.043 -0.940  0.793 11.006
```

```
Coefficients:
```

```

      Estimate Std. Error z value Pr(>|z|)
(Intercept) 1.948826  0.135619 14.37 < 2e-16 ***
Base        0.022652  0.000509 44.48 < 2e-16 ***
Age         0.022740  0.004024  5.65 1.6e-08 ***
Trtprogabide -0.152701  0.047805 -3.19 0.0014 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 2122.73 on 58 degrees of freedom
Residual deviance: 559.44 on 55 degrees of freedom
AIC: 850.7

Number of Fisher Scoring iterations: 5

```

The output provides the deviances, regression parameters, standard errors, and tests that these parameters are 0. Note that each of the predictor variables is significant at the $p < 0.05$ level.

13.3.1 Interpreting the model parameters

The model coefficients are obtained using the `coef()` function or by examining the `Coefficients` table in the `summary()` function output:

```

> coef(fit)
(Intercept)   Base   Age Trtprogabide
  1.9488      0.0227 0.0227   -0.1527

```

In a Poisson regression, the dependent variable being modeled is the log of the conditional mean $\log_e(\lambda)$. The regression parameter 0.0227 for Age indicates that a one-year increase in age is associated with a 0.02 increase in the log mean number of seizures, holding baseline seizures and treatment condition constant. The intercept is the log mean number of seizures when each of the predictors equals 0. Because you can't have a zero age and none of the participants had a zero number of baseline seizures, the intercept isn't meaningful in this case.

It's usually much easier to interpret the regression coefficients in the original scale of the dependent variable (number of seizures, rather than log number of seizures). To accomplish this, exponentiate the coefficients:

```

> exp(coef(fit))
(Intercept)   Base   Age Trtprogabide
  7.020      1.023 1.023   0.858

```

Now you see that a one-year increase in age *multiplies* the expected number of seizures by 1.023, holding the other variables constant. This means that increased age is associated with higher numbers of seizures. More important, a one-unit change in Trt (that is, moving from placebo to progabide) multiplies the expected number of seizures by 0.86. You'd expect a 14% (i.e., $1 - 0.86$) decrease in the number of seizures for the drug group compared with the placebo group, holding baseline number of seizures and age constant.

It's important to remember that, like the exponentiated parameters in logistic regression, the exponentiated parameters in the Poisson model have a multiplicative rather than an additive effect on the response variable. Also, as with logistic regression, you must evaluate your model for overdispersion.

13.3.2 Overdispersion

In a Poisson distribution, the variance and mean are equal. Overdispersion occurs in Poisson regression when the observed variance of the response variable is larger than would be predicted by the Poisson distribution. Because overdispersion is often encountered when dealing with count data and can have a negative impact on the interpretation of the results, we'll spend some time discussing it.

There are several reasons why overdispersion may occur (Coxe et al., 2009):

- The omission of an important predictor variable can lead to overdispersion.
- Overdispersion can also be caused by a phenomenon known as *state dependence*. Within observations, each event in a count is assumed to be independent. For the seizure data, this would imply that for any patient, the probability of a seizure is independent of each other seizure. But this assumption is often untenable. For a given individual, the probability of having a first seizure is unlikely to be the same as the probability of having a 40th seizure, given that they've already had 39.
- In longitudinal studies, overdispersion can be caused by the clustering inherent in repeated measures data. We won't discuss longitudinal Poisson models here.

If overdispersion is present and you don't account for it in your model, you'll get standard errors and confidence intervals that are too small, and significance tests that are too liberal (that is, you'll find effects that aren't really there).

As with logistic regression, overdispersion is suggested if the ratio of the residual deviance to the residual degrees of freedom is much larger than 1. For the seizure data, the ratio is

```
> deviance(fit)/df.residual(fit)
[1] 10.17
```

which is clearly much larger than 1.

The `qcc` package provides a test for overdispersion in the Poisson case. (Be sure to download and install this package before first use.) You can test for overdispersion in the seizure data using the following code:

```
> library(qcc)
> qcc.overdispersion.test(breslow.dat$sumY, type="poisson")
Overdispersion test Obs.Var/Theor.Var Statistic p-value
poisson data      62.9  3646  0
```

Not surprisingly, the significance test has a p-value less than 0.05, strongly suggesting the presence of overdispersion.

You can still fit a model to your data using the `glm()` function, by replacing `family="poisson"` with `family="quasipoisson"`. Doing so is analogous to the approach to logistic regression when overdispersion is present:

```
> fit.od <- glm(sumY ~ Base + Age + Trt, data=breslow.dat,
  family=quasipoisson())
> summary(fit.od)

Call:
glm(formula = sumY ~ Base + Age + Trt, family = quasipoisson(),
  data = breslow.dat)

Deviance Residuals:
  Min    1Q  Median    3Q   Max
-6.057 -2.043 -0.940  0.793 11.006

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.94883   0.46509   4.19 0.00010 ***
Base          0.02265   0.00175  12.97 < 2e-16 ***
Age           0.02274   0.01380   1.65 0.10509
Trtprogabide -0.15270   0.16394  -0.93 0.35570
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for quasipoisson family taken to be 11.8)

Null deviance: 2122.73 on 58 degrees of freedom
Residual deviance: 559.44 on 55 degrees of freedom
AIC: NA

Number of Fisher Scoring iterations: 5
```

Notice that the parameter estimates in the quasi-Poisson approach are identical to those produced by the Poisson approach. The standard errors are much larger, though. In this case, the larger standard errors have led to p-values for `Trt` (and `Age`) that are greater than 0.05. When you take overdispersion into account, there's insufficient evidence to declare that the drug regimen reduces seizure counts more than receiving a placebo, after controlling for baseline seizure rate and age.

Please remember that this example is used for demonstration purposes only. The results shouldn't be taken to imply anything about the efficacy of progabide in the real world. I'm not a doctor—at least not a medical doctor—and I don't even play one on TV.

We'll finish this exploration of Poisson regression with a discussion of some important variants and extensions.

13.3.3 Extensions

R provides several useful extensions to the basic Poisson regression model, including models that allow varying time periods, models that correct for too many zeros, and robust models

that are useful when data includes outliers and influential observations. I'll describe each separately.

POISSON REGRESSION WITH VARYING TIME PERIODS

Our discussion of Poisson regression has been limited to response variables that measure a count over a fixed length of time (for example, number of seizures in an eight-week period, number of traffic accidents in the past year, or number of pro-social behaviors in a day). The length of time is constant across observations. But you can fit Poisson regression models that allow the time period to vary for each observation. In this case, the outcome variable is a rate.

To analyze rates, you must include a variable (for example, time) that records the length of time over which the count occurs for each observation. You then change the model from

$$\log_e(\lambda) = \beta_0 + \sum_{j=1}^p \beta_j X_j$$

to

$$\log_e\left(\frac{\lambda}{\text{time}}\right) = \beta_0 + \sum_{j=1}^p \beta_j X_j$$

or equivalently

$$\log_e(\lambda) = \log_e(\text{time}) + \beta_0 + \sum_{j=1}^p \beta_j X_j$$

To fit this new model, you use the `offset` option in the `glm()` function. For example, assume that the length of time that patients participated post-randomization in the Breslow study varied from 14 days to 60 days. You could use the rate of seizures as the dependent variable (assuming you had recorded time for each patient in days) and fit the model

```
fit <- glm(Ysum ~ Base + Age + Trt, data=epilepsy,
          offset=log(time), family=poisson)
```

where `Ysum` is the number of seizures that occurred post-randomization for a patient during the time the patient was studied. In this case, you're assuming that rate doesn't vary over time (for example, 2 seizures in 4 days is equivalent to 10 seizures in 20 days).

ZERO-INFLATED POISSON REGRESSION

There are times when the number of zero counts in a dataset is larger than would be predicted by the Poisson model. This can occur when there's a subgroup of the population that would never engage in the behavior being counted. For example, in the `Affairs` dataset described in the section on logistic regression, the original outcome variable (`affairs`) counted the number

of extramarital sexual intercourse experiences participants had in the past year. It's likely that there's a subgroup of faithful marital partners who would never have an affair, no matter how long the period of time studied. These are called *structural zeros* (primarily by the swingers in the group).

In such cases, you can analyze the data using an approach called *zero-inflated Poisson regression*. The approach fits two models simultaneously—one that predicts who would or would not have an affair, and the second that predicts how many affairs a participant would have if you excluded the permanently faithful. Think of this as a model that combines a logistic regression (for predicting structural zeros) and a Poisson regression model (that predicts counts for observations that aren't structural zeros). Zero-inflated Poisson regression can be fit using the `zeroinfl()` function in the `pscl` package.

ROBUST POISSON REGRESSION

Finally, the `glmRob()` function in the `robustbase` package can be used to fit a robust generalized linear model, including robust Poisson regression. As mentioned previously, this can be helpful in the presence of outliers and influential observations.

Going further

Generalized linear models are a complex and mathematically sophisticated subject, but many fine resources are available for learning about them. A good, short introduction to the topic is Dunteman and Ho (2006). The classic (and advanced) text on generalized linear models is provided by McCullagh and Nelder (1989). Comprehensive and accessible presentations are provided by Dobson and Barnett (2008) and Fox (2008). Faraway (2006) and Fox (2002) provide excellent introductions within the context of R.

13.4 Summary

- Generalized linear models allows you to analyze response variables that are decidedly non-normal, including categorical outcomes and discrete counts.
- Logistic regression can be used when analyzing studies with a dichotomous (yes/no) outcome.
- Poisson regression can be used to analyze studies when outcomes are measured as counts or rates.
- Regression diagnostics can be more difficult for generalized linear models than for the linear models described in chapter 8. In particular, you should evaluate logistic and poisson regression models for overdispersion. If overdispersion is found, consider using an alternate error distribution such as quasi-binomial or quasi-poisson when fitting the model.

14

Principal components and factor analysis

This chapter covers

- **Principal components analysis**
- **Exploratory factor analysis**
- **Understanding other latent variable models**

One of the most challenging aspects of multivariate data is the sheer complexity of the information. If you have a dataset with 100 variables, how do you make sense of all the interrelationships present? Even with 20 variables, there are 190 pairwise correlations to consider when you're trying to understand how the individual variables relate to one another. Two related but distinct methodologies for exploring and simplifying complex multivariate data are principal components and exploratory factor analysis.

Principal components analysis (PCA) is a data-reduction technique that transforms a larger number of correlated variables into a much smaller set of uncorrelated variables called *principal components*. For example, you might use PCA to transform 30 correlated (and possibly redundant) environmental variables into 5 uncorrelated composite variables that retain as much information from the original set of variables as possible.

In contrast, *exploratory factor analysis (EFA)* is a collection of methods designed to uncover the latent structure in a given set of variables. It looks for a smaller set of underlying or *latent* variables that can explain the relationships among the observed or *manifest* variables. For example, the dataset `Harman74.cor` contains the correlations among 24 psychological tests given to 145 seventh- and eighth-grade children. If you apply EFA to this data, the results suggest that the 276 test intercorrelations can be explained by the children's abilities on 4 underlying factors (verbal ability, processing speed, deduction, and memory).

The 24 psychological tests are the observed or manifest variables, and the four underlying factors or latent variables are derived from the correlations among these observed variables.

The differences between the PCA and EFA models can be seen in figure 14.1. Principal components (PC1 and PC2) are linear combinations of the observed variables (X1 to X5). The weights used to form the linear composites are chosen to maximize the variance each principal component accounts for, while keeping the components uncorrelated.

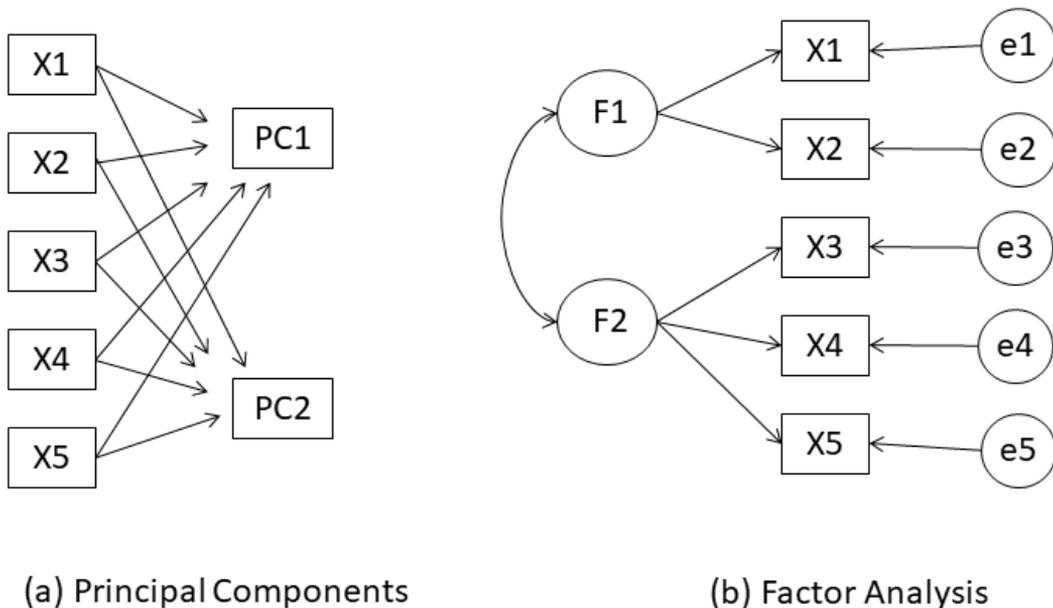


Figure 14.1 Comparing the principal components and factor analysis models. The diagrams show the observed variables (X1 to X5), the principal components (PC1, PC2), factors (F1, F2), and errors (e1 to e5).

In contrast, factors (F1 and F2) are assumed to underlie or “cause” the observed variables, rather than being linear combinations of them. The errors (e1 to e5) represent the variance in the observed variables unexplained by the factors. The circles indicate that the factors and errors aren’t directly observable but are inferred from the correlations among the variables. In this example, the curved arrow between the factors indicates that they’re correlated. Correlated factors are common, but not required, in the EFA model.

The methods described in this chapter require large samples to derive stable solutions. What constitutes an adequate sample size is somewhat complicated. Until recently, analysts used rules of thumb like “factor analysis requires 5–10 times as many subjects as variables.” Recent studies suggest that the required sample size depends on the number of factors, the number of variables associated with each factor, and how well the set of factors explains the

variance in the variables (Bandalos and Boehm-Kaufman, 2009). I'll go out on a limb and say that if you have several hundred observations, you're probably safe. In this chapter, we'll look at artificially small problems in order to keep the output (and page count) manageable.

We'll start by reviewing the functions in R that can be used to perform PCA or EFA and give a brief overview of the steps involved. Then we'll work carefully through two PCA examples, followed by an extended EFA example. A brief overview of other packages in R that can be used for fitting latent variable models is provided at the end of the chapter. This discussion includes packages for confirmatory factor analysis, structural equation modeling, correspondence analysis, and latent class analysis.

14.1 Principal components and factor analysis in R

In the base installation of R, the functions for PCA and EFA are `princomp()` and `factanal()`, respectively. In this chapter, we'll focus on functions provided in the `psych` package. They offer many more useful options than their base counterparts. Additionally, the results are reported in a metric that will be more familiar to social scientists and more likely to match the output provided by corresponding programs in other statistical packages such as SAS and IBM SPSS.

The `psych` package functions that are most relevant here are listed in table 14.1. Be sure to install the package before trying the examples in this chapter.

Table 14.1 Useful factor analytic functions in the `psych` package

Function	Description
<code>principal()</code>	Principal components analysis with optional rotation
<code>fa()</code>	Factor analysis by principal axis, minimum residual, weighted least squares, or maximum likelihood
<code>fa.parallel()</code>	Scree plots with parallel analyses
<code>factor.plot()</code>	Plot the results of a factor or principal components analysis
<code>fa.diagram()</code>	Graph factor or principal components loading matrices
<code>scree()</code>	Scree plot for factor and principal components analysis

EFA (and to a lesser degree PCA) are often confusing to new users. The reason is that they describe a wide range of approaches, and each approach requires several steps (and decisions) to achieve a final result. The most common steps are as follows:

1. *Prepare the data.* Both PCA and EFA derive their solutions from the correlations among the observed variables. You can input either the raw data matrix or the correlation matrix to the `principal()` and `fa()` functions. If raw data is input, the correlation matrix is automatically calculated. Be sure to screen the data for missing values before proceeding. By default, the `psych` package uses pairwise deletion when calculating correlations.
2. *Select a factor model.* Decide whether PCA (data reduction) or EFA (uncovering latent structure) is a better fit for your research goals. If you select an EFA approach, you'll also need to choose a specific factoring method (for example, maximum likelihood).
3. *Decide how many components/factors to extract.*
4. *Extract the components/factors.*
5. *Rotate the components/factors.*
6. *Interpret the results.*
7. *Compute component or factor scores.*

In the remainder of this chapter, we'll carefully consider each of the steps, starting with PCA. At the end of the chapter, you'll find a detailed flow chart of the possible steps in PCA/EFA (figure 14.7). The chart will make more sense once you've read through the intervening material.

14.2 Principal components

The goal of PCA is to replace a large number of correlated variables with a smaller number of uncorrelated variables while capturing as much information in the original variables as possible. These derived variables, called *principal components*, are linear combinations of the observed variables. Specifically, the first principal component

$$PC = a_1X_1 + a_2X_2 + \dots + a_kX_k$$

is the weighted combination of the k observed variables that accounts for the most variance in the original set of variables. The second principal component is the linear combination that accounts for the most variance in the original variables, under the constraint that it's *orthogonal* (uncorrelated) to the first principal component. Each subsequent component maximizes the amount of variance accounted for, while at the same time remaining uncorrelated with all previous components. Theoretically, you can extract as many principal components as there are variables. But from a practical viewpoint, you hope that you can approximate the full set of variables with a much smaller set of components. Let's look at a simple example.

The dataset `USJudgeRatings` contains lawyers' ratings of state judges in the US Superior Court. The data frame contains 43 observations on 12 numeric variables. The variables are listed in table 14.2.

Table 14.2 Variables in the USJudgeRatings dataset

Variable	Description	Variable	Description
CONT	Number of contacts of lawyer with judge	PREP	Preparation for trial
INTG	Judicial integrity	FAMI	Familiarity with law
DMNR	Demeanor	ORAL	Sound oral rulings
DILG	Diligence	WRIT	Sound written rulings
CFMG	Case flow managing	PHYS	Physical ability
DECI	Prompt decisions	RTEN	Worthy of retention

From a practical point of view, can you summarize the 11 evaluative ratings (INTG to RTEN) with a smaller number of composite variables? If so, how many will you need, and how will they be defined? Because the goal is to simplify the data, you'll approach this problem using PCA. The data are in raw score format, and there are no missing values. Therefore, your next step is deciding how many principal components you'll need.

14.2.1 Selecting the number of components to extract

Several criteria are available for deciding how many components to retain in a PCA. They include

- Basing the number of components on prior experience and theory
- Selecting the number of components needed to account for some threshold cumulative amount of variance in the variables (for example, 80%)
- Selecting the number of components to retain by examining the eigenvalues of the $k \times k$ correlation matrix among the variables

The most common approach is based on the eigenvalues. Each component is associated with an eigenvalue of the correlation matrix. The first PC is associated with the largest eigenvalue, the second PC with the second-largest eigenvalue, and so on. The *Kaiser-Harris criterion* suggests retaining components with eigenvalues greater than 1. Components with eigenvalues less than 1 explain less variance than contained in a single variable. In the *Cattell Scree test*, the eigenvalues are plotted against their component numbers. Such plots typically demonstrate a bend or elbow, and the components above this sharp break are retained. Finally, you can run simulations, extracting eigenvalues from random data matrices of the same size as the original matrix. If an eigenvalue based on real data is larger than the average corresponding eigenvalues from a set of random data matrices, that component is

retained. The approach is called *parallel analysis* (see Hayton, Allen, and Scarpello, 2004, for more details).

You can assess all three eigenvalue criteria at the same time via the `fa.parallel()` function. For the 11 ratings (dropping the CONT variable), the necessary code is as follows:

```
library(psych)
fa.parallel(USJudgeRatings[,-1], fa="pc", n.iter=100,
            show.legend=FALSE, main="Scree plot with parallel analysis")
abline(h=1)
```

This code produces the graph shown in figure 14.2. The plot displays the scree test based on the observed eigenvalues (as straight-line segments and x's), the mean eigenvalues derived from 100 random data matrices (as dashed lines), and the eigenvalues greater than 1 criteria (as a horizontal line at $y=1$). The `abline()` function is used to add a horizontal line at $y=1$.

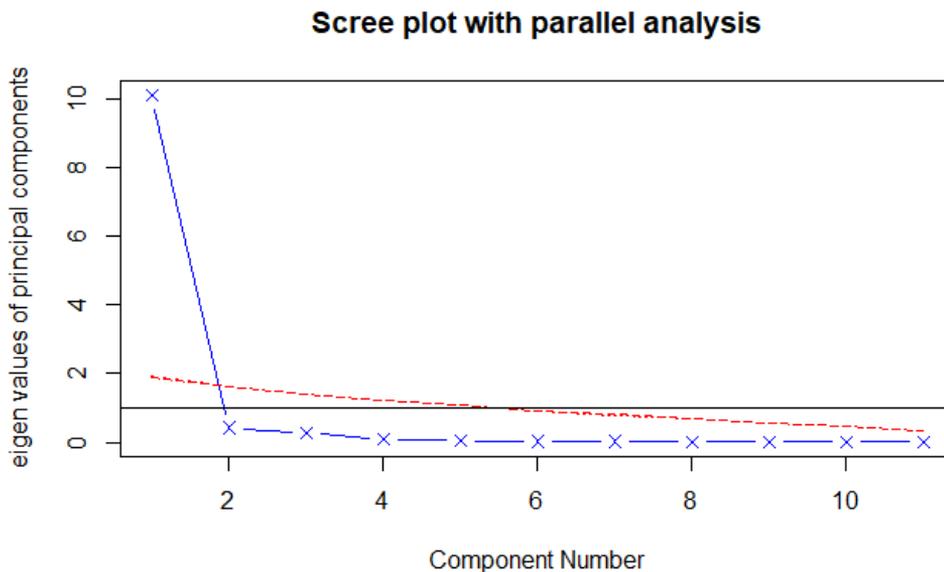


Figure 14.2 Assessing the number of principal components to retain for the `USJudgeRatings` example. A scree plot (the line with x's), eigenvalues greater than 1 criteria (horizontal line), and parallel analysis with 100 simulations (dashed line) suggest retaining a single component.

All three criteria suggest that a single component is appropriate for summarizing this dataset. Your next step is to extract the principal component using the `principal()` function.

14.2.2 Extracting principal components

As indicated earlier, the `principal()` function performs a principal components analysis starting with either a raw data matrix or a correlation matrix. The format is

```
principal(r, nfactors=, rotate=, scores=)
```

where

- `r` is a correlation matrix or a raw data matrix.
- `nfactors` specifies the number of principal components to extract (1 by default).
- `rotate` indicates the rotation to be applied (varimax by default; see section 14.2.3).
- `scores` specifies whether to calculate principal-component scores (false by default).

To extract the first principal component, you can use the code in the following listing.

Listing 14.1 Principal components analysis of `USJudgeRatings`

```
> library(psych)
> pc <- principal(USJudgeRatings[,-1], nfactors=1)
> pc

Principal Components Analysis
Call: principal(r = USJudgeRatings[, -1], nfactors=1)
Standardized loadings based upon correlation matrix
  PC1  h2  u2
INTG 0.92 0.84 0.157
DMNR 0.91 0.83 0.166
DILG 0.97 0.94 0.061
CFMG 0.96 0.93 0.072
DECI 0.96 0.92 0.076
PREP 0.98 0.97 0.030
FAMI 0.98 0.95 0.047
ORAL 1.00 0.99 0.009
WRIT 0.99 0.98 0.020
PHYS 0.89 0.80 0.201
RTEN 0.99 0.97 0.028

      PC1
SS loadings  10.13
Proportion Var 0.92
[... additional output omitted ...]
```

Here, you're inputting the raw data without the `CONT` variable and specifying that one unrotated component should be extracted. (Rotation is explained in section 14.3.3.) Because PCA is performed on a correlation matrix, the raw data is automatically converted to a correlation matrix before the components are extracted.

The column labeled `PC1` contains the component *loadings*, which are the correlations of the observed variables with the principal component(s). If you extracted more than one principal component, there would be columns for `PC2`, `PC3`, and so on. Component loadings are used to interpret the meaning of components. You can see that each variable correlates highly with the first component (`PC1`). It therefore appears to be a general evaluative dimension.

The column labeled h^2 contains the component *communalities*—the amount of variance in each variable explained by the components. The u^2 column contains the component *uniquenesses*—the amount of variance not accounted for by the components (or $1 - h^2$). For example, 80% of the variance in physical ability (PHYS) ratings is accounted for by the first PC, and 20% isn't. PHYS is the variable least well represented by a one-component solution.

The row labeled SS Loadings contains the eigenvalues associated with the components. The eigenvalues are the standardized variance associated with a particular component (in this case, the value for the first component is 10). Finally, the row labeled Proportion Var represents the amount of variance accounted for by each component. Here you see that the first principal component accounts for 92% of the variance in the 11 variables.

Let's consider a second example, one that results in a solution with more than one principal component. The dataset `Harman23.cor` contains data on 8 body measurements for 305 girls. In this case, the dataset consists of the correlations among the variables rather than the original data (see table 14.3).

Table 14.3 Correlations among body measurements for 305 girls (`Harman23.cor`)

	Height	Arm span	Forearm	Lower leg	Weight	Bitro diameter	Chest girth	Chest width
Height	1.00	0.85	0.80	0.86	0.47	0.40	0.30	0.38
Arm span	0.85	1.00	0.88	0.83	0.38	0.33	0.28	0.41
Forearm	0.80	0.88	1.00	0.80	0.38	0.32	0.24	0.34
Lower leg	0.86	0.83	0.8	1.00	0.44	0.33	0.33	0.36
Weight	0.47	0.38	0.38	0.44	1.00	0.76	0.73	0.63
Bitro diameter	0.40	0.33	0.32	0.33	0.76	1.00	0.58	0.58
Chest girth	0.30	0.28	0.24	0.33	0.73	0.58	1.00	0.54
Chest width	0.38	0.41	0.34	0.36	0.63	0.58	0.54	1.00

Source: H. H. Harman, *Modern Factor Analysis, Third Edition Revised*, University of Chicago Press, 1976, Table 2.3.

Again, you wish to replace the original physical measurements with a smaller number of derived variables. You can determine the number of components to extract using the following code. In this case, you need to identify the correlation matrix (the `cov` component of the `Harman23.cor` object) and specify the sample size (`n.obs`):

```
library(psych)
fa.parallel(Harman23.cor$cov, n.obs=302, fa="pc", n.iter=100,
            show.legend=FALSE, main="Scree plot with parallel analysis")
abline(h=1)
```

The resulting graph is displayed in figure 14.3.

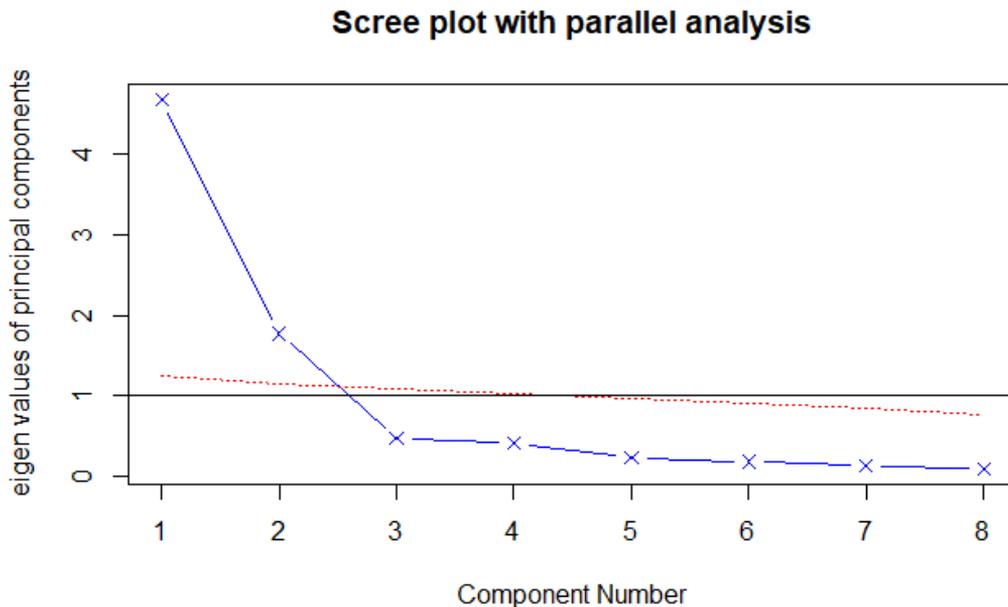


Figure 14.3 Assessing the number of principal components to retain for the body measurements example. The scree plot (line with x's), eigenvalues greater than 1 criteria (horizontal line), and parallel analysis with 100 simulations (dashed line) suggest retaining two components.

You can see from the plot that a two-component solution is suggested. As in the first example, the Kaiser–Harris criteria, scree test, and parallel analysis agree. This won't always be the case, and you may need to extract different numbers of components and select the solution that appears most useful. The next listing extracts the first two principal components from the correlation matrix.

Listing 14.2 Principal components analysis of body measurements

```

> library(psych)
> pc <- principal(Harman23.cor$cov, nfactors=2, rotate="none")
> pc

Principal Components Analysis
Call: principal(r = Harman23.cor$cov, nfactors = 2, rotate = "none")
Standardized loadings based upon correlation matrix

    PC1  PC2  h2  u2
height  0.86 -0.37 0.88 0.123
arm.span 0.84 -0.44 0.90 0.097
forearm  0.81 -0.46 0.87 0.128
lower.leg 0.84 -0.40 0.86 0.139
weight   0.76 0.52 0.85 0.150
bitro.diameter 0.67 0.53 0.74 0.261
chest.girth 0.62 0.58 0.72 0.283
chest.width 0.67 0.42 0.62 0.375

    PC1  PC2
SS loadings 4.67 1.77
Proportion Var 0.58 0.22
Cumulative Var 0.58 0.81

[... additional output omitted ...]

```

If you examine the PC1 and PC2 columns in listing 14.2, you see that the first component accounts for 58% of the variance in the physical measurements, whereas the second component accounts for 22%. Together, the two components account for 81% of the variance. The two components together account for 88% of the variance in the height variable.

Components and factors are interpreted by examining their loadings. The first component correlates positively with each physical measure and appears to be a general size factor. The second component contrasts the first four variables (height, arm span, forearm, and lower leg), with the second four variables (weight, bitro diameter, chest girth, and chest width). It therefore appears to be a length-versus-volume factor. Conceptually, this isn't an easy construct to work with. Whenever two or more components have been extracted, you can rotate the solution to make it more interpretable. This is the topic we'll turn to next.

14.2.3 Rotating principal components

Rotations are a set of mathematical techniques for transforming the component loading matrix into one that's more interpretable. They do this by "purifying" the components as much as possible. Rotation methods differ with regard to whether the resulting components remain uncorrelated (*orthogonal rotation*) or are allowed to correlate (*oblique rotation*). They also differ in their definition of purifying. The most popular orthogonal rotation is the *varimax* rotation, which attempts to purify the columns of the loading matrix, so that each component is defined by a limited set of variables (that is, each column has a few large loadings and many very small loadings). Applying a varimax rotation to the body measurement data, you

get the results provided in the next listing. You'll see an example of an oblique rotation in section 14.4.

Listing 14.3 Principal components analysis with varimax rotation

```
> rc <- principal(Harman23.cor$cov, nfactors=2, rotate="varimax")
> rc
```

Principal Components Analysis

Call: principal(r = Harman23.cor\$cov, nfactores = 2, rotate = "varimax")

Standardized loadings based upon correlation matrix

	RC1	RC2	h2	u2
height	0.90	0.25	0.88	0.123
arm.span	0.93	0.19	0.90	0.097
forearm	0.92	0.16	0.87	0.128
lower.leg	0.90	0.22	0.86	0.139
weight	0.26	0.88	0.85	0.150
bitro.diameter	0.19	0.84	0.74	0.261
chest.girth	0.11	0.84	0.72	0.283
chest.width	0.26	0.75	0.62	0.375

	RC1	RC2
SS loadings	3.52	2.92
Proportion Var	0.44	0.37
Cumulative Var	0.44	0.81

[... additional output omitted ...]

The column names change from PC to RC to denote rotated components. Looking at the loadings in column RC1, you see that the first component is primarily defined by the first four variables (length variables). The loadings in the column RC2 indicate that the second component is primarily defined by variables 5 through 8 (volume variables). Note that the two components are still uncorrelated and that together, they still explain the variables equally well. You can see that the rotated solution explains the variables equally well because the variable communalities haven't changed. Additionally, the cumulative variance accounted for by the two-component rotated solution (81%) hasn't changed. But the proportion of variance accounted for by each individual component has changed (from 58% to 44% for component 1 and from 22% to 37% for component 2). This spreading out of the variance across components is common, and technically you should now call them components rather than principal components (because the variance-maximizing properties of individual components haven't been retained).

The ultimate goal is to replace a larger set of correlated variables with a smaller set of derived variables. To do this, you need to obtain scores for each observation on the components.

14.2.4 Obtaining principal components scores

In the `USJudgeRatings` example, you extracted a single principal component from the raw data describing lawyers' ratings on 11 variables. The `principal()` function makes it easy to obtain scores for each participant on this derived variable (see the next listing).

Listing 14.4 Obtaining component scores from raw data

```
> library(psych)
> pc <- principal(USJudgeRatings[, -1], nfactors=1, score=TRUE)
> head(pc$score)
  PC1
AARONSON,L.H. -0.1857981
ALEXANDER,J.M. 0.7469865
ARMENTANO,A.J. 0.0704772
BERDON,R.I. 1.1358765
BRACKEN,J.J. -2.1586211
BURNS,E.B. 0.7669406
```

The principal component scores are saved in the `scores` element of the object returned by the `principal()` function when the option `scores=TRUE`. If you wanted, you could now get the correlation between the number of contacts occurring between a lawyer and a judge and their evaluation of the judge using

```
> cor(USJudgeRatings$CONT, pc$score)
  PC1
[1,] -0.008815895
```

Apparently, there's no relationship between the lawyer's familiarity and their opinions!

When the principal components analysis is based on a correlation matrix and the raw data aren't available, getting principal component scores for each observation is clearly not possible. But you can get the coefficients used to calculate the principal components.

In the body measurement data, you have correlations among body measurements, but you don't have the individual measurements for these 305 girls. You can get the scoring coefficients using the code in the following listing.

Listing 14.5 Obtaining principal component scoring coefficients

```
> library(psych)
> rc <- principal(Harman23.cor$cov, nfactors=2, rotate="varimax")
> round(unclass(rc$weights), 2)
  RC1 RC2
height    0.28 -0.05
arm.span  0.30 -0.08
forearm   0.30 -0.09
lower.leg 0.28 -0.06
weight   -0.06 0.33
bitro.diameter -0.08 0.32
chest.girth -0.10 0.34
chest.width -0.04 0.27
```

The component scores are obtained using the formulas

```
PC1 = 0.28*height + 0.30*arm.span + 0.30*forearm + 0.29*lower.leg -
      0.06*weight - 0.08*bitro.diameter - 0.10*chest.girth -
      0.04*chest.width
```

and

```
PC2 = -0.05*height - 0.08*arm.span - 0.09*forearm - 0.06*lower.leg +
      0.33*weight + 0.32*bitro.diameter + 0.34*chest.girth +
      0.27*chest.width
```

These equations assume that the physical measurements have been standardized (mean = 0, sd = 1). Note that the weights for PC1 tend to be around 0.3 or 0. The same is true for PC2. As a practical matter, you could simplify your approach further by taking the first composite variable as the mean of the standardized scores for the first four variables. Similarly, you could define the second composite variable as the mean of the standardized scores for the second four variables. This is typically what I'd do in practice.

Little Jiffy conquers the world

There's quite a bit of confusion among data analysts regarding PCA and EFA. One reason for this is historical and can be traced back to a program called Little Jiffy (no kidding). Little Jiffy was one of the most popular early programs for factor analysis, and it defaulted to a principal components analysis, extracting components with eigenvalues greater than 1 and rotating them to a varimax solution. The program was so widely used that many social scientists came to think of this default behavior as synonymous with EFA. Many later statistical packages also incorporated these defaults in their EFA programs.

As I hope you'll see in the next section, there are important and fundamental differences between PCA and EFA. To learn more about the PCA/EFA confusion, see Hayton, Allen, and Scarpello, 2004.

If your goal is to look for latent underlying variables that explain your observed variables, you can turn to factor analysis. This is the topic of the next section.

14.3 Exploratory factor analysis

The goal of EFA is to explain the correlations among a set of observed variables by uncovering a smaller set of more fundamental unobserved variables underlying the data. These hypothetical, unobserved variables are called *factors*. (Each factor is assumed to explain the variance shared among two or more observed variables, so technically, they're called *common factors*.)

The model can be represented as

$$X_i = a_1F_1 + a_2F_2 + \dots + a_pF_p + U_i$$

where X_i is the i th observed variable ($i = 1 \dots k$), F_j are the common factors ($j = 1 \dots p$), and $p < k$. U_i is the portion of variable X_i unique to that variable (not explained by the common factors). The a_i can be thought of as the degree to which each factor contributes to the composition of an observed variable. If we go back to the `Harman74.cor` example at the

beginning of this chapter, we'd say that an individual's scores on each of the 24 observed psychological tests is due to a weighted combination of their ability on 4 underlying psychological constructs.

Although the PCA and EFA models differ, many of the steps appear similar. To illustrate the process, you'll apply EFA to the correlations among six psychological tests. One hundred twelve individuals were given six tests, including a nonverbal measure of general intelligence (general), a picture-completion test (picture), a block design test (blocks), a maze test (maze), a reading comprehension test (reading), and a vocabulary test (vocab). Can you explain the participants' scores on these tests with a smaller number of underlying or latent psychological constructs?

The covariance matrix among the variables is provided in the dataset `ability.cov`. You can transform this into a correlation matrix using the `cov2cor()` function:

```
> options(digits=2)
> covariances <- ability.cov$cov
> correlations <- cov2cor(covariances)
> correlations
      general picture blocks maze reading vocab
general  1.00  0.47  0.55 0.34  0.58  0.51
picture  0.47  1.00  0.57 0.19  0.26  0.24
blocks   0.55  0.57  1.00 0.45  0.35  0.36
maze     0.34  0.19  0.45 1.00  0.18  0.22
reading  0.58  0.26  0.35 0.18  1.00  0.79
vocab    0.51  0.24  0.36 0.22  0.79  1.00
```

Because you're looking for hypothetical constructs that explain the data, you'll use an EFA approach. As in PCA, the next task is to decide how many factors to extract.

14.3.1 Deciding how many common factors to extract

To decide on the number of factors to extract, turn to the `fa.parallel()` function:

```
> library(psych)
> covariances <- ability.cov$cov
> correlations <- cov2cor(covariances)
> fa.parallel(correlations, n.obs=112, fa="both", n.iter=100,
             main="Scree plots with parallel analysis")
> abline(h=c(0, 1))
```

The resulting plot is shown in figure 14.4. Notice you've requested that the function display results for both a principal-components and common-factor approach, so that you can compare them (`fa = "both"`).

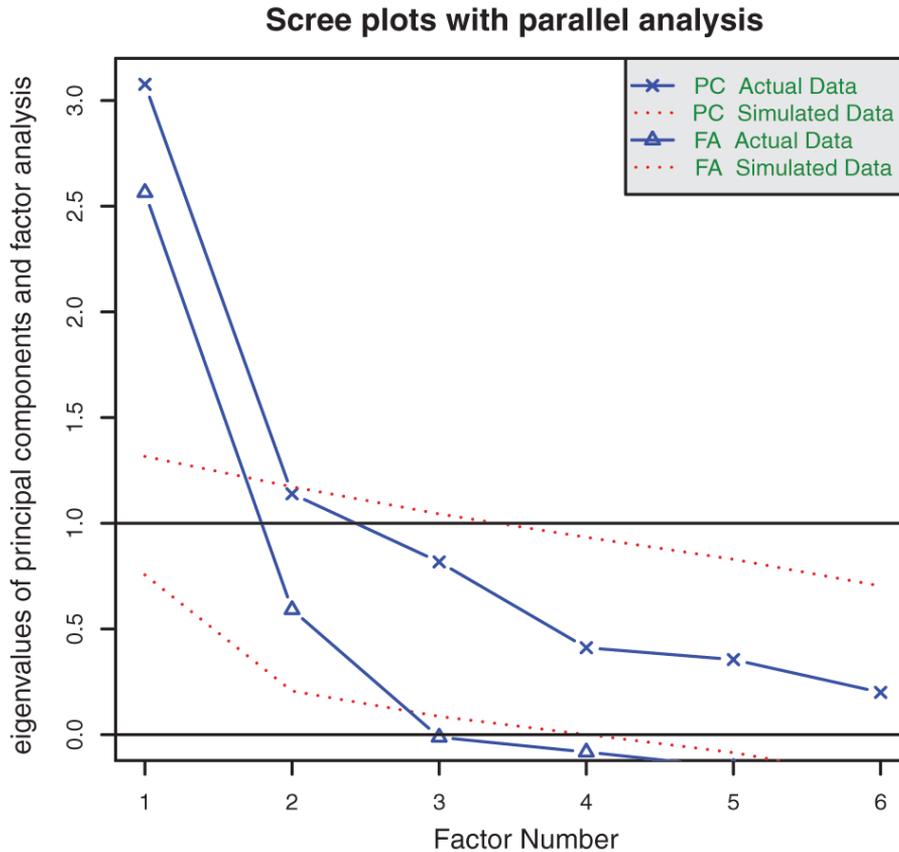


Figure 14.4 Assessing the number of factors to retain for the psychological tests example. Results for both PCA and EFA are present. The PCA results suggest one or two components. The EFA results suggest two factors.

There are several things to notice in this graph. If you'd taken a PCA approach, you might have chosen one component (scree test, parallel analysis) or two components (eigenvalues greater than 1). When in doubt, it's usually a better idea to overfactor than to underfactor. Overfactoring tends to lead to less distortion of the "true" solution.

Looking at the EFA results, a two-factor solution is clearly indicated. The first two eigenvalues (triangles) are above the bend in the scree test and also above the mean eigenvalues based on 100 simulated data matrices. For EFA, the Kaiser–Harris criterion is number of eigenvalues above 0, rather than 1. (Most people don't realize this, so it's a good way to win bets at parties.) In the present case the Kaiser–Harris criteria also suggest two factors.

14.3.2 Extracting common factors

Now that you've decided to extract two factors, you can use the `fa()` function to obtain your solution. The format of the `fa()` function is

```
fa(r, nfactors=, n.obs=, rotate=, scores=, fm=)
```

where

- `r` is a correlation matrix or a raw data matrix.
- `nfactors` specifies the number of factors to extract (1 by default).
- `n.obs` is the number of observations (if a correlation matrix is input).
- `rotate` indicates the rotation to be applied (oblimin by default).
- `scores` specifies whether or not to calculate factor scores (false by default).
- `fm` specifies the factoring method (minres by default).

Unlike PCA, there are many methods of extracting common factors. They include maximum likelihood (`ml`), iterated principal axis (`pa`), weighted least square (`wls`), generalized weighted least squares (`gls`), and minimum residual (`minres`). Statisticians tend to prefer the maximum likelihood approach because of its well-defined statistical model. Sometimes, this approach fails to converge, in which case the iterated principal axis option often works well. To learn more about the different approaches, see Mulaik (2009) and Gorsuch (1983).

For this example, you'll extract the unrotated factors using the iterated principal axis (`fm = "pa"`) approach. The results are given in the next listing.

Listing 14.6 Principal axis factoring without rotation

```
> fa <- fa(correlations, nfactors=2, rotate="none", fm="pa")
> fa
Factor Analysis using method = pa
Call: fa(r = correlations, nfactors = 2, rotate = "none", fm = "pa")
Standardized loadings based upon correlation matrix
  PA1 PA2 h2 u2
general 0.75 0.07 0.57 0.43
picture 0.52 0.32 0.38 0.62
blocks 0.75 0.52 0.83 0.17
maze 0.39 0.22 0.20 0.80
reading 0.81 -0.51 0.91 0.09
vocab 0.73 -0.39 0.69 0.31
  PA1 PA2
SS loadings 2.75 0.83
Proportion Var 0.46 0.14
Cumulative Var 0.46 0.60
[... additional output deleted ...]
```

You can see that the two factors account for 60% of the variance in the six psychological tests. When you examine the loadings, though, they aren't easy to interpret. Rotating them should help.

14.3.3 Rotating factors

You can rotate the two-factor solution from section 14.3.4 using either an orthogonal rotation or an oblique rotation. Let's try both so you can see how they differ. First try an orthogonal rotation (in the next listing).

Listing 14.7 Factor extraction with orthogonal rotation

```
> fa.varimax <- fa(correlations, nfactores=2, rotate="varimax", fm="pa")
> fa.varimax
Factor Analysis using method = pa
Call: fa(r = correlations, nfactores = 2, rotate = "varimax", fm = "pa")
Standardized loadings based upon correlation matrix
  PA1 PA2 h2 u2
general 0.49 0.57 0.57 0.43
picture 0.16 0.59 0.38 0.62
blocks  0.18 0.89 0.83 0.17
maze    0.13 0.43 0.20 0.80
reading 0.93 0.20 0.91 0.09
vocab   0.80 0.23 0.69 0.31

  PA1 PA2
SS loadings  1.83 1.75
Proportion Var 0.30 0.29
Cumulative Var 0.30 0.60

[... additional output omitted ...]
```

Looking at the factor loadings, the factors are certainly easier to interpret. Reading and vocabulary load on the first factor; and picture completion, block design, and mazes load on the second factor. The general nonverbal intelligence measure loads on both factors. This suggests that the correlations among the 6 psychological tests (the manifest variables) may be explained by two underlying latent variables (a verbal intelligence factor and a nonverbal intelligence factor).

By using an orthogonal rotation, you artificially force the two factors to be uncorrelated. What would you find if you allowed the two factors to correlate? You can try an oblique rotation such as *promax* (see the next listing).

Listing 14.8 Factor extraction with oblique rotation

```
> fa.promax <- fa(correlations, nfactores=2, rotate="promax", fm="pa")
> fa.promax
Factor Analysis using method = pa
Call: fa(r = correlations, nfactores = 2, rotate = "promax", fm = "pa")
Standardized loadings based upon correlation matrix
  PA1 PA2 h2 u2
general 0.36 0.49 0.57 0.43
picture -0.04 0.64 0.38 0.62
blocks  -0.12 0.98 0.83 0.17
maze    -0.01 0.45 0.20 0.80
reading 1.01 -0.11 0.91 0.09
vocab   0.84 -0.02 0.69 0.31
```

```

      PA1 PA2
SS loadings  1.82 1.76
Proportion Var 0.30 0.29
Cumulative Var 0.30 0.60

With factor correlations of
      PA1 PA2
PA1 1.00 0.57
PA2 0.57 1.00
[... additional output omitted ...]

```

Several differences exist between the orthogonal and oblique solutions. In an orthogonal solution, attention focuses on the *factor structure matrix* (the correlations of the variables with the factors). In an oblique solution, there are three matrices to consider: the factor structure matrix, the factor pattern matrix, and the factor intercorrelation matrix.

The *factor pattern matrix* is a matrix of standardized regression coefficients. They give the weights for predicting the variables from the factors. The *factor intercorrelation matrix* gives the correlations among the factors.

In listing 14.8, the values in the PA1 and PA2 columns constitute the factor pattern matrix. They're standardized regression coefficients rather than correlations. Examination of the columns of this matrix is still used to name the factors (although there's some controversy here). Again, you'd find a verbal and nonverbal factor.

The factor intercorrelation matrix indicates that the correlation between the two factors is 0.57. This is a hefty correlation. If the factor intercorrelations had been low, you might have gone back to an orthogonal solution to keep things simple.

The *factor structure matrix* (or factor loading matrix) isn't provided. But you can easily calculate it using the formula $F = P * \Phi$, where F is the factor loading matrix, P is the factor pattern matrix, and Φ is the factor intercorrelation matrix. A simple function for carrying out the multiplication is as follows:

```

fsm <- function(oblique) {
  if (class(oblique)[2]!="fa" & is.null(oblique$Phi)) {
    warning("Object doesn't look like oblique EFA")
  } else {
    P <- unclass(oblique$loading)
    F <- P %*% oblique$Phi
    colnames(F) <- c("PA1", "PA2")
    return(F)
  }
}

```

Applying this to the example, you get

```

> fsm(fa.promax)
      PA1 PA2
general 0.64 0.69
picture 0.33 0.61
blocks  0.44 0.91
maze    0.25 0.45

```

```
reading 0.95 0.47
vocab 0.83 0.46
```

Now you can review the correlations between the variables and the factors. Comparing them to the factor loading matrix in the orthogonal solution, you see that these columns aren't as pure. This is because you've allowed the underlying factors to be correlated. Although the oblique approach is more complicated, it's often a more realistic model of the data.

You can graph an orthogonal or oblique solution using the `factor.plot()` or `fa.diagram()` function. The code

```
factor.plot(fa.promax, labels=rownames(fa.promax$loadings))
```

produces the graph in figure 14.5.

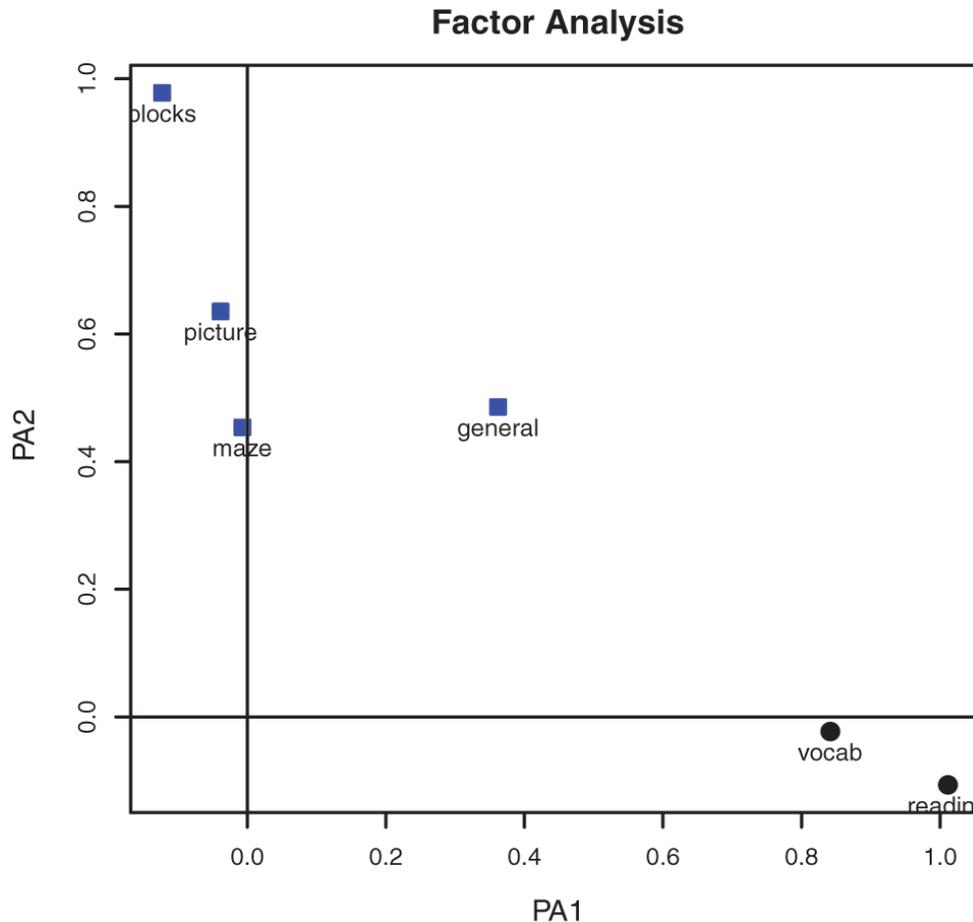


Figure 14.5 Two-factor plot for the psychological tests in `ability.cov`, `vocab` and `reading` load on the first

factor (PA1), and `blocks`, `picture`, and `maze` load on the second factor (PA2). The general intelligence test loads on both.

The code

```
fa.diagram(fa.promax, simple=FALSE)
```

produces the diagram in figure 14.6. If you let `simple = TRUE`, only the largest loading per item is displayed. It shows the largest loadings for each factor, as well as the correlations between the factors. This type of diagram is helpful when there are several factors.

Factor Analysis

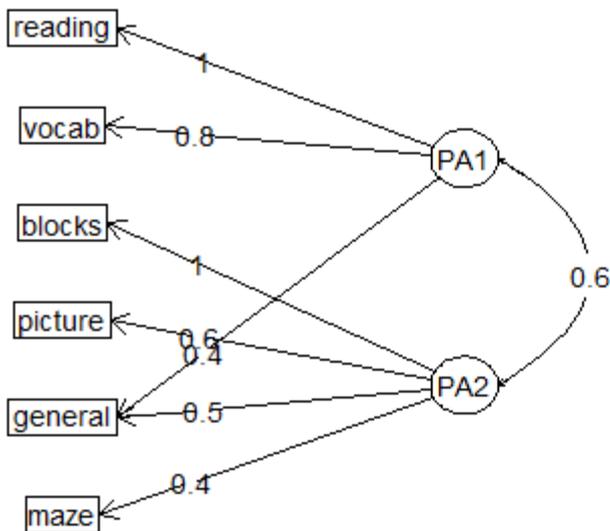


Figure 14.6 Diagram of the oblique two-factor solution for the psychological test data in `ability.csv`

When you're dealing with data in real life, it's unlikely that you'd apply factor analysis to a dataset with so few variables. We've done it here to keep things manageable. If you'd like to test your skills, try factor-analyzing the 24 psychological tests contained in `Harman74.csv`. The code

```
library(psych)
fa.24tests <- fa(Harman74.csv$cov, nfactors=4, rotate="promax")
```

should get you started!

14.3.4 Factor scores

Compared with PCA, the goal of EFA is much less likely to be the calculation of factor scores. But these scores are easily obtained from the `fa()` function by including the `score = TRUE` option (when raw data are available). Additionally, the scoring coefficients (standardized regression weights) are available in the `weights` element of the object returned.

For the `ability.cov` dataset, you can obtain the beta weights for calculating the factor score estimates for the two-factor oblique solution using

```
> fa.promax$weights
  [,1] [,2]
general 0.080 0.210
picture 0.021 0.090
blocks  0.044 0.695
maze    0.027 0.035
reading 0.739 0.044
vocab   0.176 0.039
```

Unlike component scores, which are calculated exactly, factor scores can only be estimated. Several methods exist. The `fa()` function uses the regression approach. To learn more about factor scores, see DiStefano, Zhu, and Míndrila, (2009).

Before moving on, let's briefly review other R packages that are useful for exploratory factor analysis.

14.3.5 Other EFA-related packages

R contains a number of other contributed packages that are useful for conducting factor analyses. The `FactoMineR` package provides methods for PCA and EFA, as well as other latent variable models. It provides many options that we haven't considered here, including the use of both numeric and categorical variables. The `FAiR` package estimates factor analysis models using a genetic algorithm that permits the ability to impose inequality restrictions on model parameters. The `GPArotation` package offers many additional factor rotation methods. Finally, the `nFactors` package offers sophisticated techniques for determining the number of factors underlying data.

14.4 Other latent variable models

EFA is only one of a wide range of latent variable models used in statistics. We'll end this chapter with a brief description of other models that can be fit within R. These include models that test a priori theories, that can handle mixed data types (numeric and categorical), or that are based solely on categorical multiway tables.

In EFA, you allow the data to determine the number of factors to be extracted and their meaning. But you could start with a theory about how many factors underlie a set of variables, how the variables load on those factors, and how the factors correlate with one another. You

could then test this theory against a set of collected data. The approach is called *confirmatory factor analysis (CFA)*.

CFA is a subset of a methodology called *structural equation modeling (SEM)*. SEM allows you to posit not only the number and composition of underlying factors but also how these factors impact one another. You can think of SEM as a combination of confirmatory factor analyses (for the variables) and regression analyses (for the factors). The resulting output includes statistical tests and fit indices. There are several excellent packages for CFA and SEM in R. They include `sem`, `OpenMx`, and `lavaan`.

The `ltm` package can be used to fit latent models to the items contained in tests and questionnaires. The methodology is often used to create large-scale standardized tests. Examples include the Scholastic Aptitude Test (SAT) and the Graduate Record Exam (GRE).

Latent class models (where the underlying factors are assumed to be categorical rather than continuous) can be fit with the `FlexMix`, `lcmm`, `randomLCA`, and `poLCA` packages. The `lcda` package performs latent class discriminant analysis, and the `lsa` package performs latent semantic analysis, a methodology used in natural language processing.

The `ca` package provides functions for simple and multiple correspondence analysis. These methods allow you to explore the structure of categorical variables in two-way and multiway tables, respectively.

Finally, R contains numerous methods for *multidimensional scaling (MDS)*. MDS is designed to detect underlying dimensions that explain the similarities and distances between a set of measured objects (for example, countries). The `cmdscale()` function in the base installation performs a classical MDS, whereas the `isoMDS()` function in the `MASS` package performs a nonmetric MDS. The `vegan` package also contains functions for classical and nonmetric MDS.

14.5 Summary

- Principal components analysis (PCA) a useful data-reduction method that can replace many correlated variables with a smaller number of uncorrelated composite variables.
- Exploratory factor analysis (EFA) contains a broad range of methods for identifying latent or unobserved constructs (factors) that may underlie a set of observed or manifest variables.
- While the goal of PCA is typically to summarize data and reduce its dimensionality, EFA can be used as a hypothesis-generating tool, useful when you're trying to understand the relationships among variables. It's often used in the social sciences for theory development.
- PCA and EFA are both multistep processes that require the data analyst to make choices at each step. These steps are outlined in Figure 14.7.

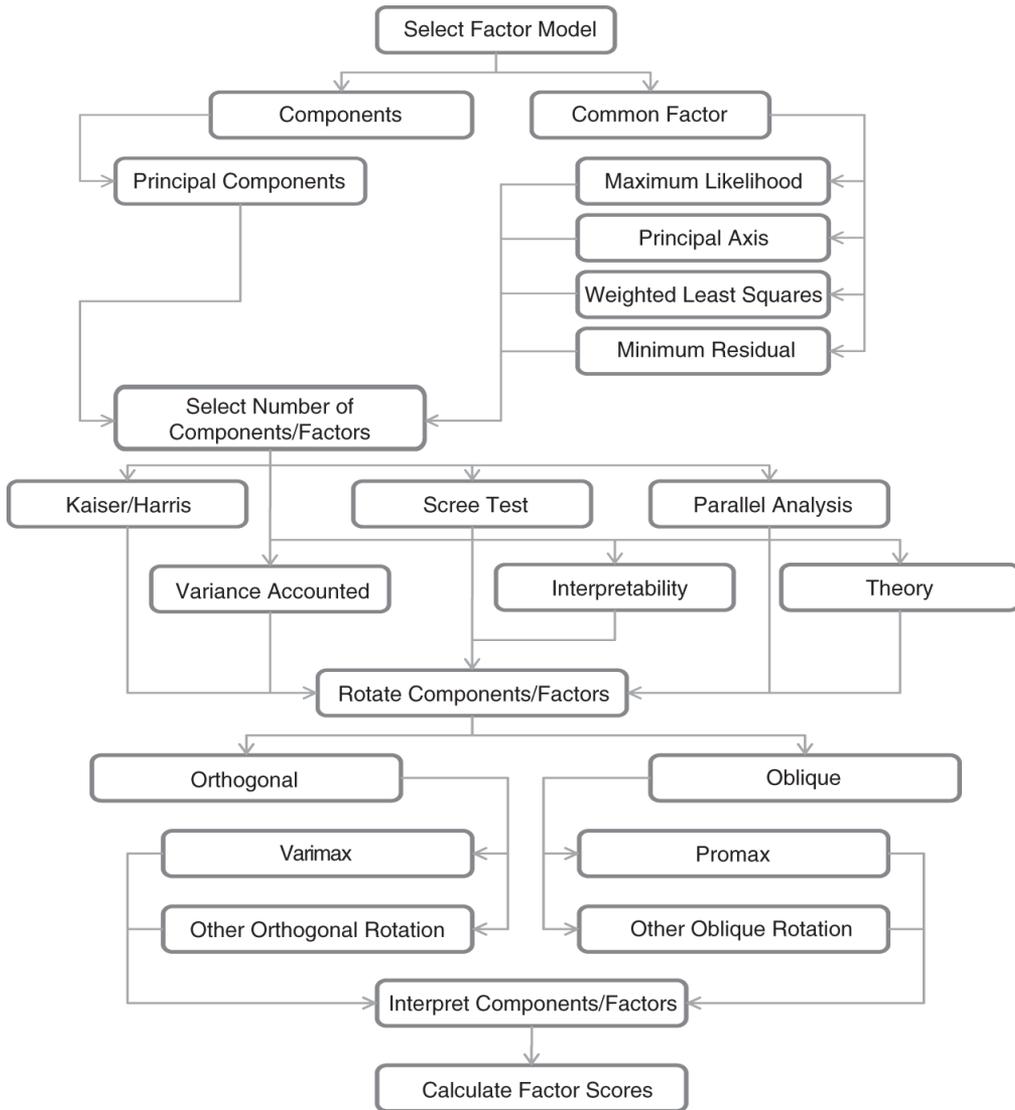


Figure 14.7 A principal components/exploratory factor analysis decision chart

15

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 $Y_1, Y_2, Y_3, \dots, Y_t, \dots, Y_T$, 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 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.

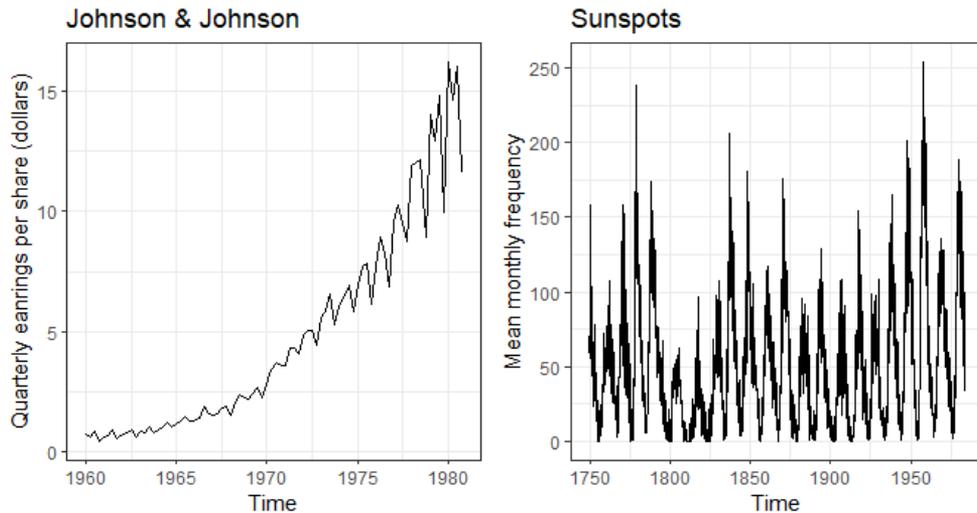


Figure 15.1 Time series plots for (a) Johnson & Johnson quarterly earnings per share (in dollars) from 1960 to 1980, and (b) the monthly mean relative sunspot numbers recorded from 1749 to 1983

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 analytical capabilities available anywhere. This chapter explores some of the most common descriptive and forecasting approaches and the R functions used to perform them. Table 15.1 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 15.1 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.

In order to reproduce the analyses in this chapter, be sure to install the `xts`, `forecast`, `tseries`, and `directlabels` packages before continuing (`install.packages(c("xts", "forecast", "tseries", "directlabels"))`).

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 and date specifications for the observations. Once the data are in a time-series object, you can use numerous functions to manipulate, model, and plot the data.

R packages offers a variety of structures for holding time series (see Time series objects in R). In this chapter, we'll use the `xts` class offered by the `xts` package. It supports both regularly and irregularly spaced time series and has a wide range of function for manipulating time series data.

Time series objects in R

It is easy to get lost among the many objects R provides for holding time series data. Base R comes with `ts` for holding a single time series with regularly spaced time intervals, and `mts` for multiple time series with regularly spaced intervals. The `zoo` package offers a `zoo` class which can hold time series with irregularly spaced intervals, and the `xts` package offers a superset of the `zoo` class, with more supporting functions. Other popular formats include `tsibble`, `timeSeries`, `irts`, and `tis`. Luckily, the `tsbox` package provides functions to convert a data frame into any of these formats and can also convert one time series format into another.

To create a `xts` time series, you'll use

```
library(xts)
myseries <- xts(data, index)
```

where `data` is a numeric vector at values, and `index` is a date vector indicating when the values were observed. An example is given in the following listing. The data consist of monthly sales figures for two years, starting in January 2018.

Listing 15.1 Creating a time-series object

```
library(xts)
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)
date <- seq(from = as.Date("2018/1/1"),
            to = as.Date("2019/12/1"),
            by = "month")
sales.xts <- xts(sales, date)
```

Time series objects in `xts` format can be subset using bracket `[]` notation. For example, `sales.xts["2018"]` will return all data from 2018. Specifying `sales.xts["2018-3/2019-5"]` will return all data from March 2018 to May 2019.

There are also `apply` functions designed to execute a function on each distinct period of a time series object. They are particularly useful for aggregating a time series into larger time periods. The format is

```
newseries <- apply.period(x, FUN, ...)
```

where `period` can be `daily`, `weekly`, `monthly`, `quarterly`, or `yearly`, `x` is an `xts` time series object, `FUN` is the function to be applied, and `...` are arguments passed to `FUN`.

For example, `quarterlies <- apply.quarterly(sales.xts, sum)` will return a time series with 8 quarterly sales totals. The `sum` function could be replaced with `mean`, `median`, `min`, `max`, or any other function returning a single value.

The `autoplot()` function in the `forecast` package can be used for plotting time series data as `ggplot2` graphs. Listing 15.2 provides two examples.

Listing 15.2 Plotting time-series

```
library(ggplot2)
library(forecast)
autoplot(sales.xts) #1

autoplot(sales.xts) +
  geom_line(color="blue") + #2
  scale_x_date(date_breaks="1 months", #3
              date_labels="%b %y") +
  labs(x="", y="Sales", title="Customized Time Series Plot") +
  theme_bw() + #4
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust=1),
        panel.grid.minor.x=element_blank())
```

#1 Default graph

#2 Set line color

#3 Specify x-axis labels

#4 Adjust theme

In the first example, the `autoplot()` function is used to create `ggplot2` graph #1. The graph is provided in figure 15.2.

In the second example, the plot is modified to make it more appealing. The line color is changed to blue #2. The `scale_x_date()` function is used to provide better labels for the x-axis #2. The `date_breaks` option specifies the distance between tick marks and take on values like "1 day", "2 weeks", "5 years" or whatever is appropriate. The `date_labels` option specifies the format for the labels. Here "%b %y" specifies month (3 letters) and year (2 digits) with a space in between. See section 3.6 for a table of these codes. Finally, a black and white theme is chosen, the x-axis labels are rotated 90-degrees, and the vertical minor grid lines are suppressed #4. The customized graph is displayed in figure 15.3.

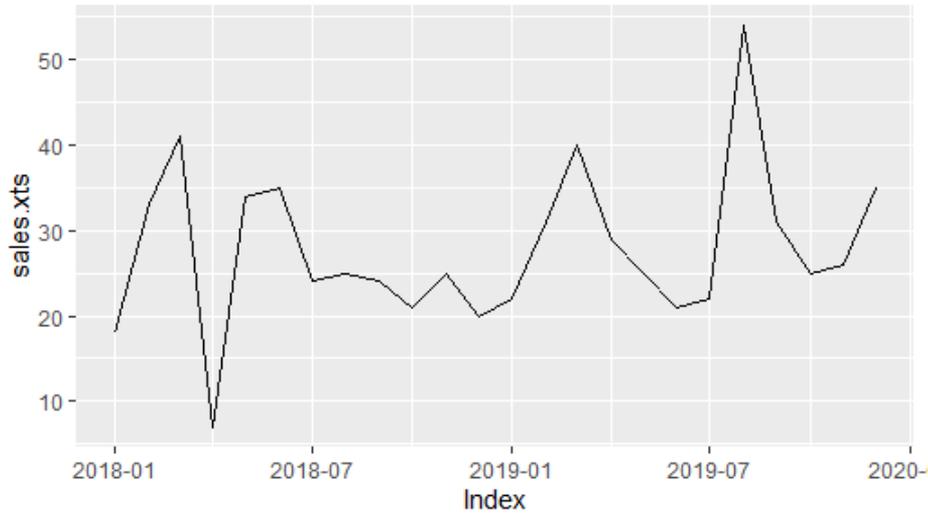


Figure 15.2 Time-series plot for the sales data in listing 15.1. This is the default format provided by the `autoplot()` function.

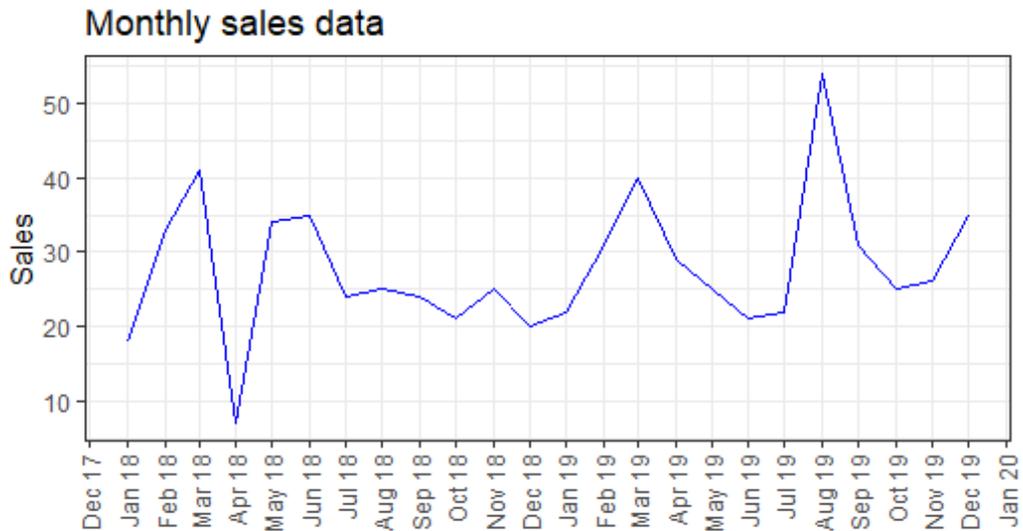


Figure 15.3. Times-series plot for the sales data in listing 15.1. The graph is customized with color, better labeling and cleaner theme elements.

The time series examples that come with base R (table 15.1) are actually in `ts` format, but luckily the functions introduced in this chapter can handle time series in either `ts` or `xts` format.

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

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

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 q 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 15.3 Simple moving averages

```
library(forecast)
library(ggplot2)

theme_set(theme_bw())
ylim <- c(min(Nile), max(Nile))
```

```

autoplot(Nile) +
  ggtitle("Raw time series") +
  scale_y_continuous(limits=yylim)

autoplot(ma(Nile, 3)) +
  ggtitle("Simple Moving Averages (k=3)") +
  scale_y_continuous(limits=yylim)

autoplot(ma(Nile, 7)) +
  ggtitle("Simple Moving Averages (k=7)") +
  scale_y_continuous(limits=yylim)

autoplot(ma(Nile, 15)) +
  ggtitle("Simple Moving Averages (k=15)") +
  scale_y_continuous(limits=yylim)

```

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 over-smoothing. 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 15.4, 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.

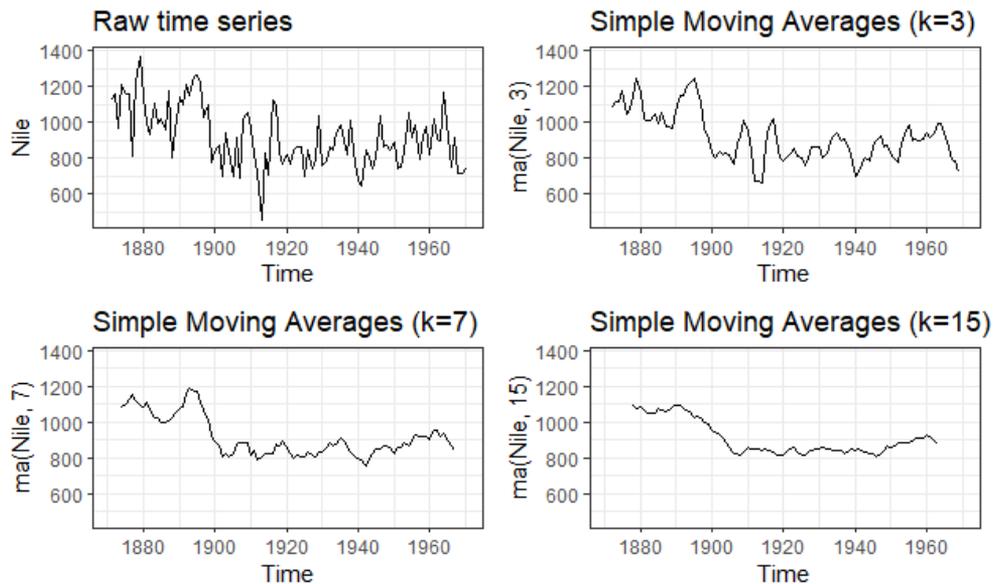


Figure 15.4 The Nile time series measuring annual river flow at Ashwan from 1871–1970 (upper left). The other plots are smoothed versions using simple moving averages at three smoothing levels ($k=3$, 7, and 15).

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.

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 15.5.

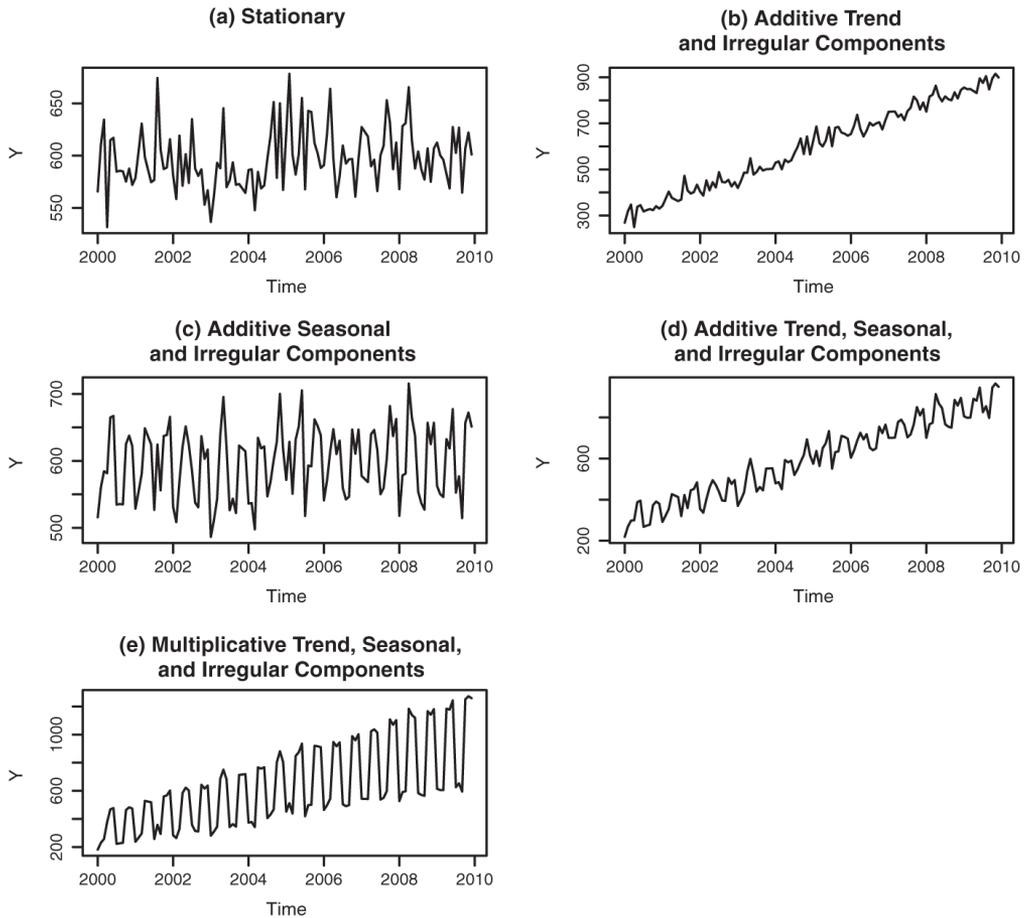


Figure 15.5 Time-series examples consisting of different combinations of trend, seasonal, and irregular components

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. 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:

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

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 15.6. From the graph, it appears that variability of the series increases with the level, suggesting a multiplicative model.

The plot in the lower portion of figure 15.6 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.

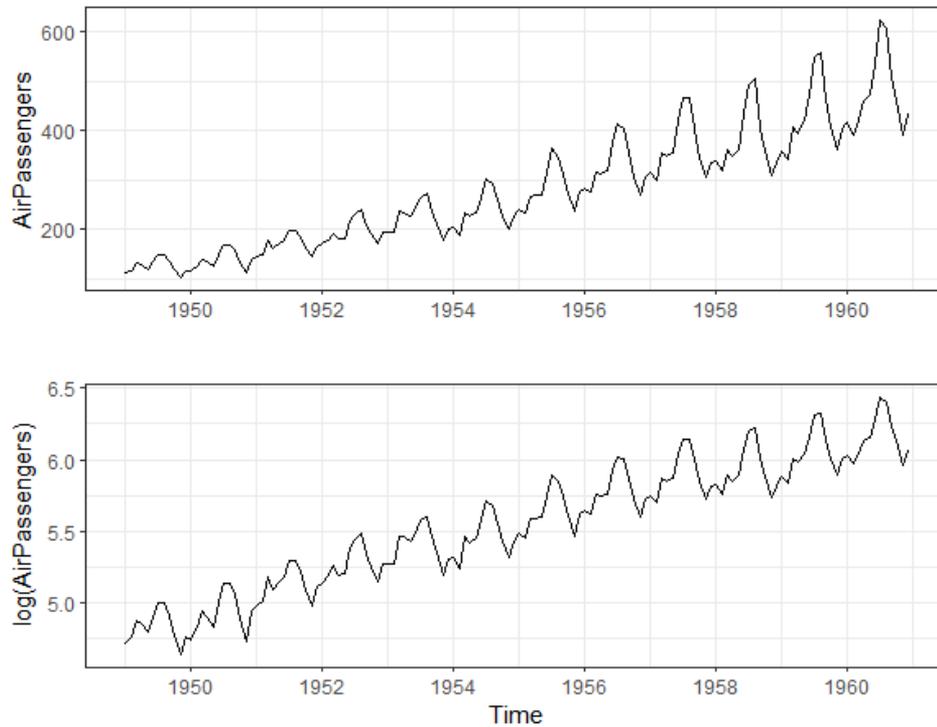


Figure 15.6 Plot of the `AirPassengers` time series (top). The time series contains the monthly totals (in thousands) of international airline passengers between 1949 and 1960. The log-transformed time series (bottom) stabilizes the variance and fits an additive seasonal decomposition model better.

Listing 15.4 Seasonal decomposition using `stl()`

```
> library(forecast)
> library(ggplot2)
> autoplot(AirPassengers)           #1
> lAirPassengers <- log(AirPassengers)
> autoplot(lAirPassengers, ylab="log(AirPassengers)")

> fit <- stl(lAirPassengers, s.window="period")  #2
> autoplot(fit)

> fit$time.series                   #3
      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
... 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 1.2415 126.6 0.9417
```

```
... output omitted ...
```

#1 Plots the time series

#2 Decomposes the time series

#3 Components for each observation

First, the time series is plotted and transformed #1. A seasonal decomposition is performed and saved in an object called fit #2. 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 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.

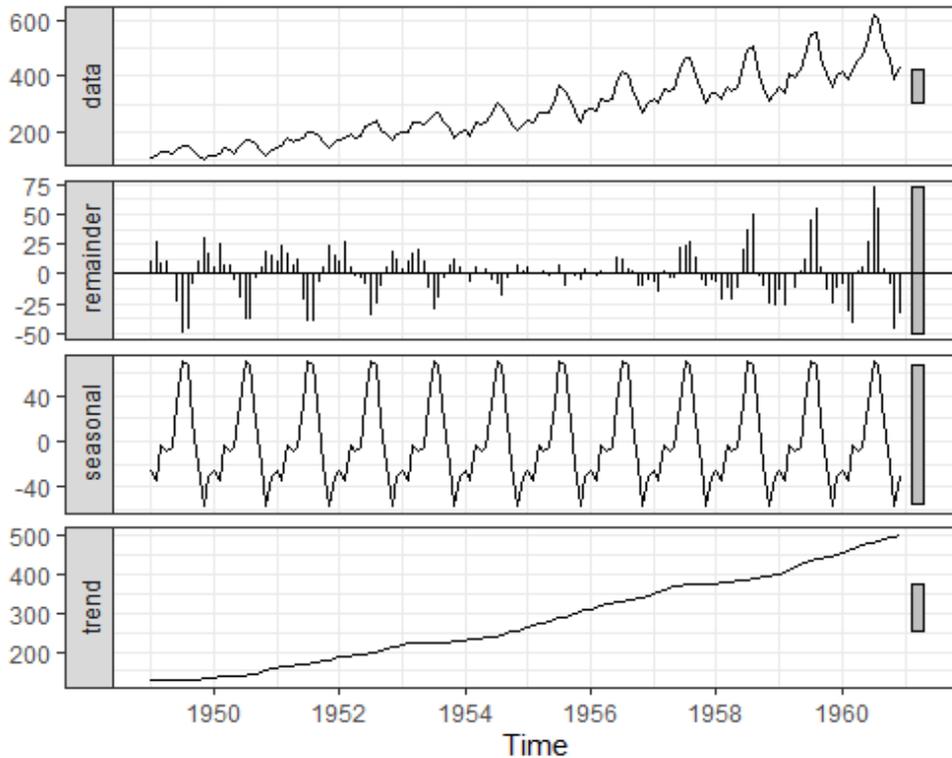


Figure 15.6 A seasonal decomposition of the logged `AirPassengers` time series using the `stl()` function. The time series (data) is decomposed into seasonal, trend, and irregular components.

The object returned by the `stl()` function contains a component called `time.series` that contains the trend, season, and irregular portion of each observation #3. 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 decreased by 11% in February (with a multiplier of .89) and increased by 24% in July (a multiplier of 1.24).

The `forecast` package provides additional tools for visualizing the seasonal decomposition. Listing 15.5 demonstrates the creation a month plot and seasonal plot.

Listing 15.5 Month and season plots

```
library(forecast)
library(ggplot2)
library(directlabels)
```

```

ggmonthplot(AirPassengers) + #1
labs(title="Month plot: AirPassengers",
      x="",
      y="Passengers (thousands)")

p <- ggseasonplot(AirPassengers) + geom_point() + #2
labs(title="Seasonal plot: AirPassengers",
      x="",
      y="Passengers (thousands)")
direct.label(p)

```

#1 Monthplot
#2 Seasonplot

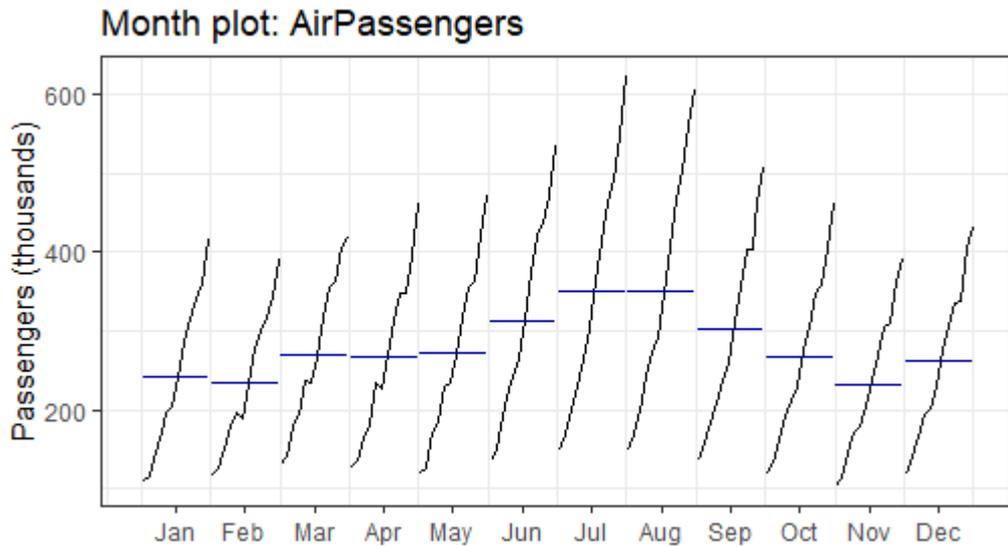


Figure 15.7 A month plot of `AirPassenger` time-series. The month plot displays the subseries for each month (all January values from 1949 to 1960 connected, all February values connected, and so on), along with the average of each subseries. There is a uniform increasing trend for each month, and the greatest number of passengers tend to fly in July and August.

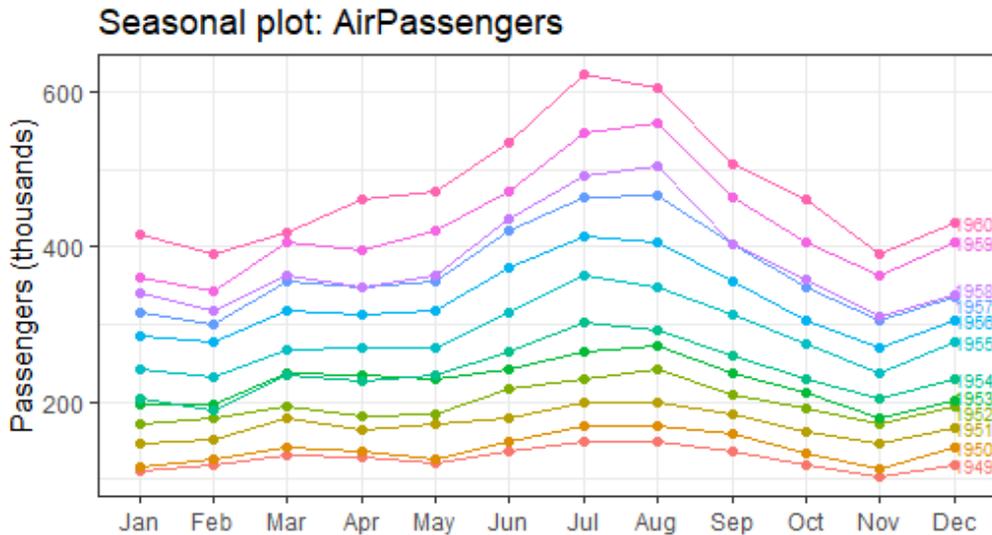


Figure 15.8 A season plot (bottom) for the `AirPassengers` time series. Each shows an increasing trend and similar seasonal pattern year to year.

The month plot (Figure 15.7) 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 (Figure 15.8) displays the subseries by year. Again you see a similar pattern, with increases in passengers each year, and the same seasonal pattern. By default, the `ggplot2` package would create a legend for the year variable. The `directlabels` package is used to place the year labels directly on the graph, next to each line in the time series.

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 `ets()` function that comes with the `forecast` package. 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 15.2.

Table 15.2 Functions for fitting simple, double, and triple exponential forecasting models

Type	Parameters fit	Functions
simple	level	<code>ets(ts, model="ANN")</code> <code>ses(ts)</code>
double	level, slope	<code>ets(ts, model="AAN")</code> <code>holt(ts)</code>
triple	level, slope, seasonal	<code>ets(ts, model="AAA")</code> <code>hw(ts)</code>

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.

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

$$Y_t = \text{level} + \text{irregular}_t$$

The prediction at time Y_{t+1} (called the *1-step ahead forecast*) is written as

$$Y_{t+1} = c_0 Y_t + c_1 Y_{t-1} + c_2 Y_{t-2} + \dots$$

where $c_i = \alpha(1 - \alpha)^i$, $i = 0, 1, 2, \dots$ 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 15.9.

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 15.6 Simple exponential smoothing

```
> library(forecast)
> fit <- ets(nhtemp, model="ANN")          #1
> fit

ETS(A,N,N)

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

Smoothing parameters:
  alpha = 0.1819

Initial states:
  l = 50.2762

sigma: 1.1455

  AIC  AICc  BIC
265.9298 266.3584 272.2129

> forecast(fit, 1)                        #2

  Point Forecast Lo 80 Hi 80 Lo 95 Hi 95
1972      51.87 50.402 53.338 49.625 54.115

> autoplot(forecast(fit, 1)) +
  labs(x = "Year",
       y = expression(paste("Temperature (", degree*"F", ")")),
       title = "New Haven Annual Mean Temperature")

> accuracy(fit)                          #3

  ME RMSE MAE MPE MAPE MASE
Training set 0.146 1.126 0.895 0.242 1.749 0.751
```

```
#1 Fits the model
#2 1-step ahead forecast
#3 Prints accuracy measures
```

The `ets(mode="ANN")` statement fits the simple exponential model to the `nhtemp` time series #1. 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 (49.6°F to 54.1°F) #2. The time series, the forecasted value, and the 80% and 95% confidence intervals are plotted in figure 15.8 #3.

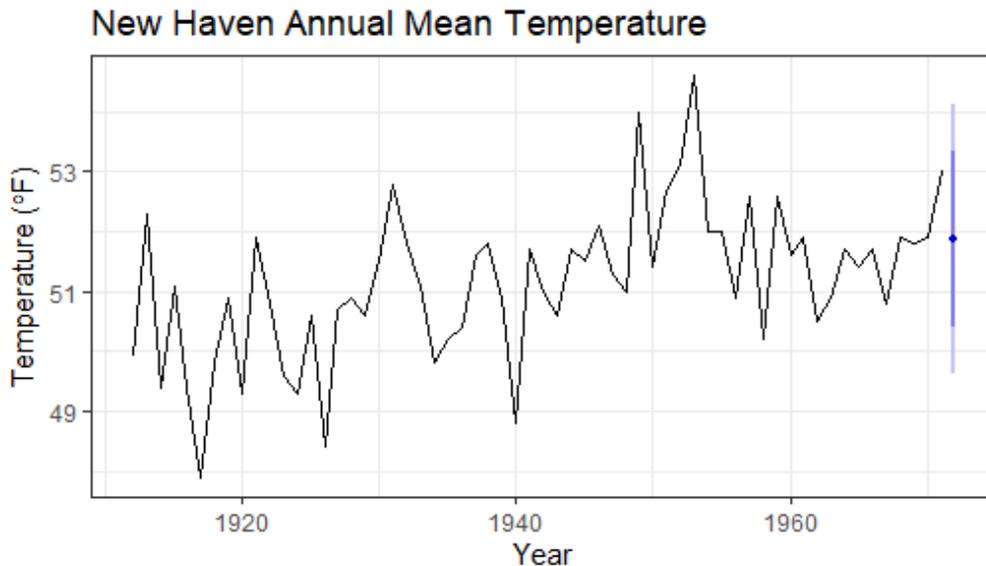


Figure 15.9. Average yearly temperatures in New Haven, Connecticut; and a 1-step ahead prediction from a simple exponential forecast using the `ets()` function

The `forecast` package also provides an `accuracy()` function that displays the most popular predictive accuracy measures for time-series forecasts #3. A description of each is given in table 15.3. The e_t represent the error or irregular component of each observation $(Y_t - \hat{Y}_t)$.

Table 15.3 Predictive accuracy measures

Measure	Abbreviation	Definition
Mean error	ME	$\text{mean}(e_t)$
Root mean squared error	RMSE	$\text{sqrt}(\text{mean}(e_t^2))$
Mean absolute error	MAE	$\text{mean}(e_t)$
Mean percentage error	MPE	$\text{mean}(100 * e_t / Y_t)$
Mean absolute percentage error	MAPE	$\text{mean}(100 * e_t / Y_t)$
Mean absolute scaled error	MASE	$\text{mean}(q_t)$ where $q_t = e_t / (1/(T-1) * \text{sum}(y_t - y_{t-1}))$, 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

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

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

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

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 15.7 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 = 0.6975
beta = 0.0031
gamma = 1e-04

Initial states:
l = 4.7925
b = 0.0111
s = -0.1045 -0.2206 -0.0787 0.0562 0.2049 0.2149
    0.1146 -0.0081 -0.0059 0.0225 -0.1113 -0.0841

sigma: 0.0383

AIC AICc BIC
-207.17 -202.31 -156.68

> accuracy(fit)

      ME RMSE MAE MPE MAPE MASE
Training set -0.0018307 0.03607 0.027709 -0.034356 0.50791 0.22892
> pred <- forecast(fit, 5)
#2
> pred
      Point Forecast Lo 80 Hi 80 Lo 95 Hi 95
Jan 1961    6.1093 6.0603 6.1584 6.0344 6.1843
Feb 1961    6.0925 6.0327 6.1524 6.0010 6.1841
Mar 1961    6.2366 6.1675 6.3057 6.1310 6.3423
```

```

Apr 1961    6.2185 6.1412 6.2958 6.1003 6.3367
May 1961    6.2267 6.1420 6.3115 6.0971 6.3564

> autoplot(pred) +
  labs(title = "Forecast for Air Travel",
        y = "Log(AirPassengers)",
        x = "Time")

> pred$mean <- exp(pred$mean)           #3
> pred$lower <- exp(pred$lower)        #3
> pred$upper <- exp(pred$upper)        #3
> p <- cbind(pred$mean, pred$lower, pred$upper)
> dimnames(p)[[2]] <- c("mean", "Lo 80", "Lo 95", "Hi 80", "Hi 95")
> p

      mean Lo 80 Lo 95 Hi 80 Hi 95
Jan 1961 450.04 428.51 417.53 472.65 485.08
Feb 1961 442.54 416.83 403.83 469.85 484.97
Mar 1961 511.13 477.01 459.88 547.69 568.10
Apr 1961 501.97 464.63 446.00 542.30 564.95
May 1961 506.10 464.97 444.57 550.87 576.15

```

#1 Smoothing parameters**#2 Future forecasts****#3 Makes forecasts in the original scale**

The smoothing parameters for the level (.70), trend (.0004), and seasonal components (.003) are given in #1. The low value for the trend (.0001) 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 #2 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 #3. 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.

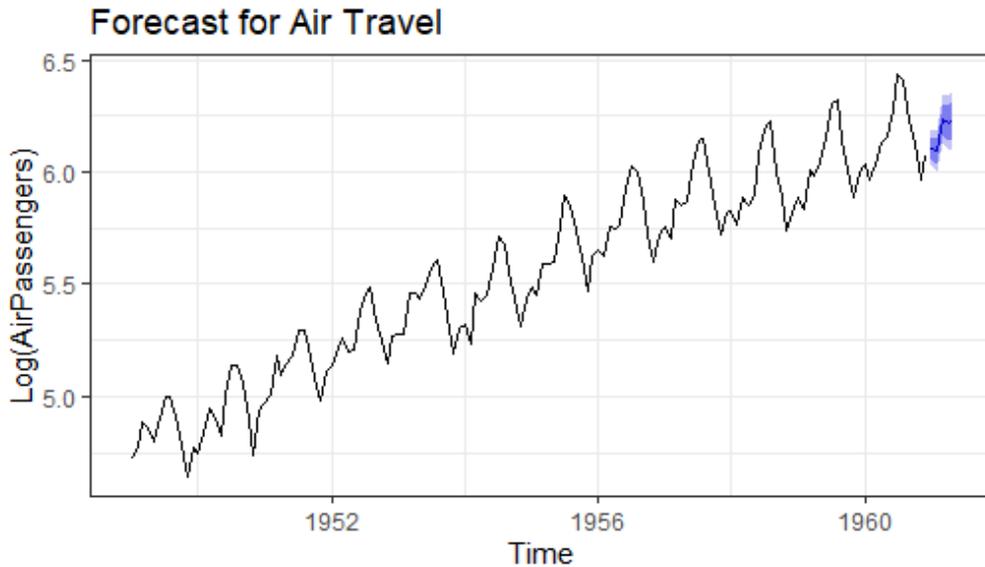


Figure 15.10 Five-year forecast of $\log(\text{number of international airline passengers in thousands})$ based on a Holt-Winters exponential smoothing model. Data are from the `AirPassengers` time series.

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 as `ets(AirPassengers, model="MAM")`. 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 15.8 Automatic exponential forecasting with `ets()`

```

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

ETS(M,M,M)

Call:
ets(y = JohnsonJohnson)

Smoothing parameters:
alpha = 0.2776
beta  = 0.0636
gamma = 0.5867

Initial states:
l = 0.6276
b = 0.0165
s = -0.2293 0.1913 -0.0074 0.0454

sigma: 0.0921

AIC AICc BIC
163.64 166.07 185.52

> autoplot(forecast(fit)) +
  labs(x = "Time",
       y = "Quarterly Earnings (Dollars)",
       title="Johnson and Johnson Forecasts")

```

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 15.11.

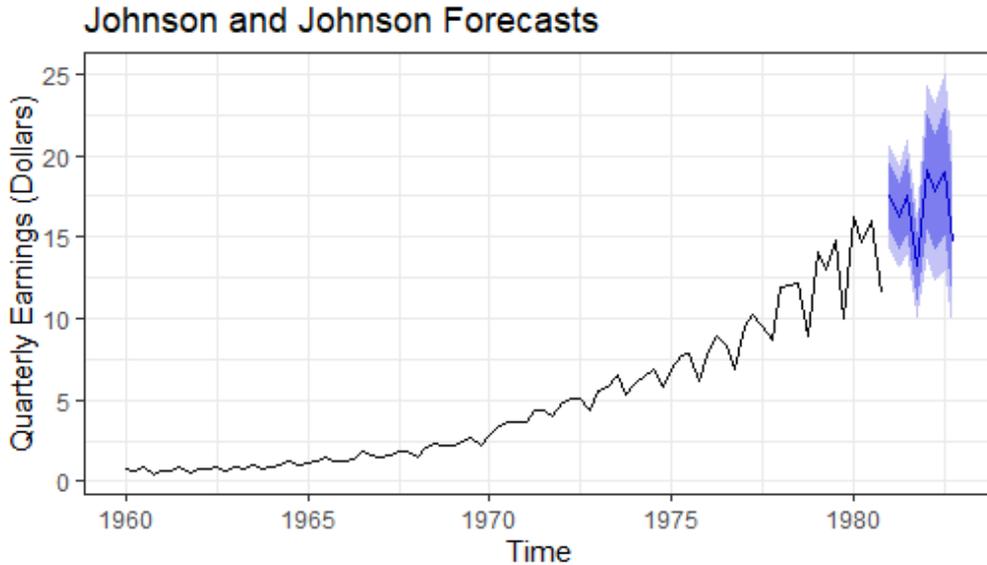


Figure 15.11 Multiplicative exponential smoothing forecast with trend and seasonal components. The forecasts are a dashed line, and the 80% and 95% confidence intervals are provided in light and dark blue, respectively.

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 15.4. 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 15.4 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, \dots, 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 `forecast` package. 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 15.13.

A *partial autocorrelation* is the correlation between Y_t and Y_{t-k} with the effects of all Y values between the two ($Y_{t-1}, Y_{t-2}, \dots, Y_{t-k+1}$) removed. Partial autocorrelations can also be plotted for multiple values of k . The PACF plot can be generated with the `pacf()` function in the `forecast` package. 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 15.13.

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

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

where Y_t is a given value of the series, μ is the mean of the series, the β s 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

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

where the ε s are the errors of prediction and the θ s 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

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

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.

Listing 15.9 Transforming the time series and assessing stationarity

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

[1] 1

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

Augmented Dickey-Fuller Test

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

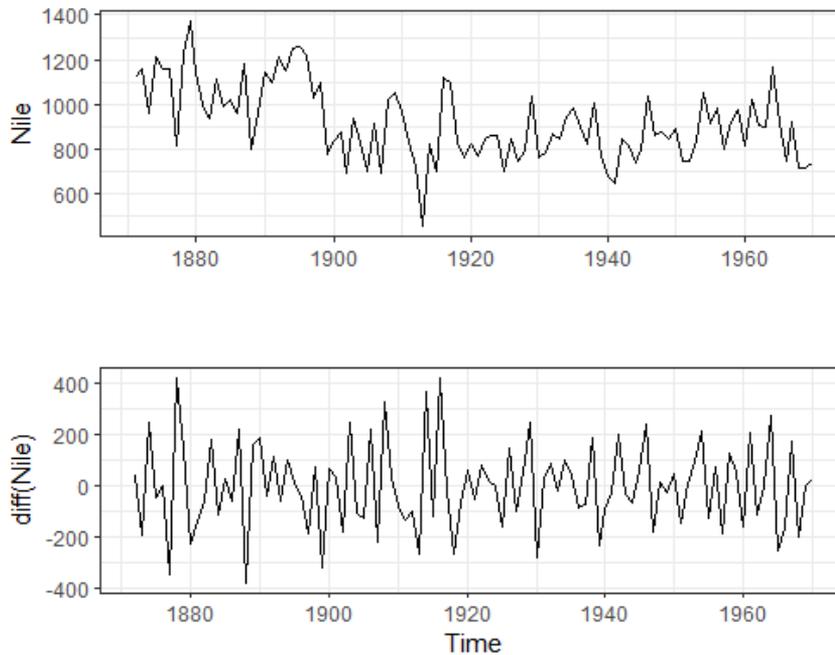


Figure 15.12 Time series displaying the annual flow of the river Nile at Ashwan from 1871 to 1970 (top) along with the times series differenced once (bottom). The differencing removes the decreasing trend evident in the original plot.

The series is differenced once ($\text{lag}=1$ is the default) and saved as `dNile`. The differenced time series is plotted in the bottom half of figure 15.12 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:

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

The resulting plots are given in figure 15.13.

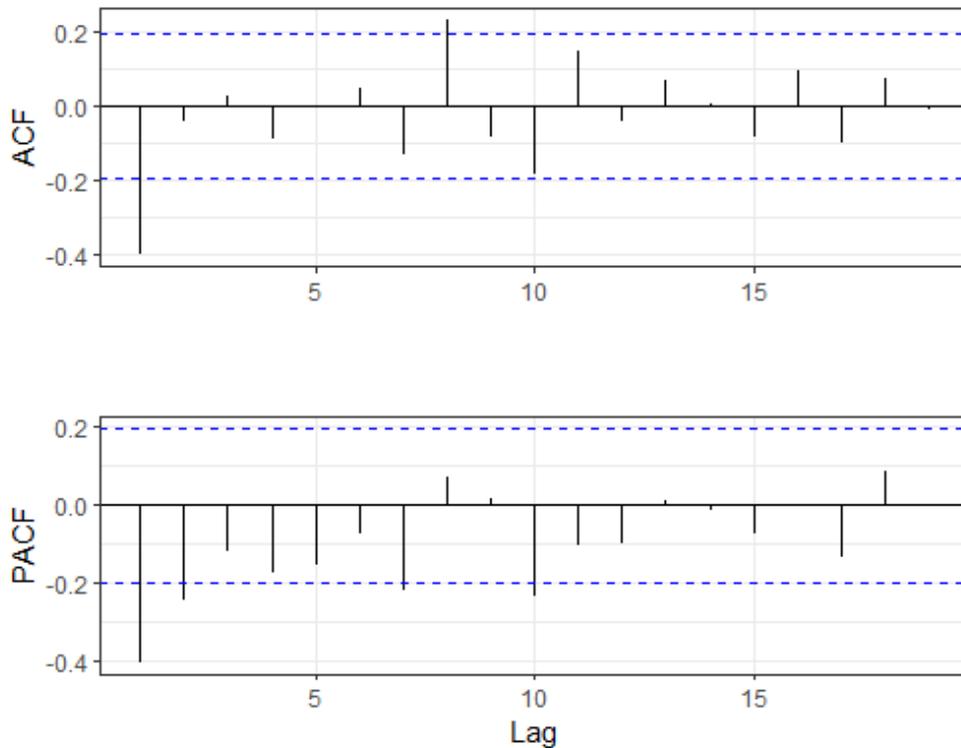


Figure 15.13 Autocorrelation and partial autocorrelation plots for the differenced Nile time series

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 15.5.

Table 15.5 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 15.5 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 15.13, 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

```
> library(ggplot2)
> df <- data.frame(resid = as.numeric(fit$residuals)) #1
> ggplot(df, aes(sample = resid)) + #2
  stat_qq() + stat_qq_line() +
  labs(title="Normal Q-Q Plot")

> Box.test(fit$residuals, type="Ljung-Box") #3
Box-Ljung test

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

#1 Extract residuals

#2 Create Q-Q plot

#3 Test autocorrelations are zero for all lags

First, the residuals are extracted from the `fit` object and saved in a data frame. Then `qq_*` functions are used to produce the Q-Q plot (figure 15.14). Normally distributed data should fall along the line. In this case, the results look good.

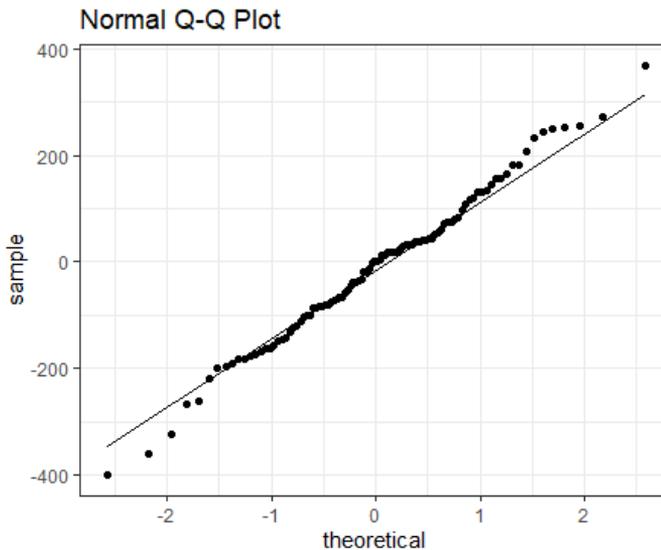


Figure 15.14 Normal Q-Q plot for determining the normality of the time-series residuals. Normally distributed values are expected to fall along the line.

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.

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

> autoplot(forecast(fit, 3)) + labs(x="Year", y="Annual Flow")
```

The `autoplot()` function is used to plot the forecast in figure 15.15. Point estimates are given by the black line, and 80% and 95% confidence bands are represented by dark and light blue bands, respectively.

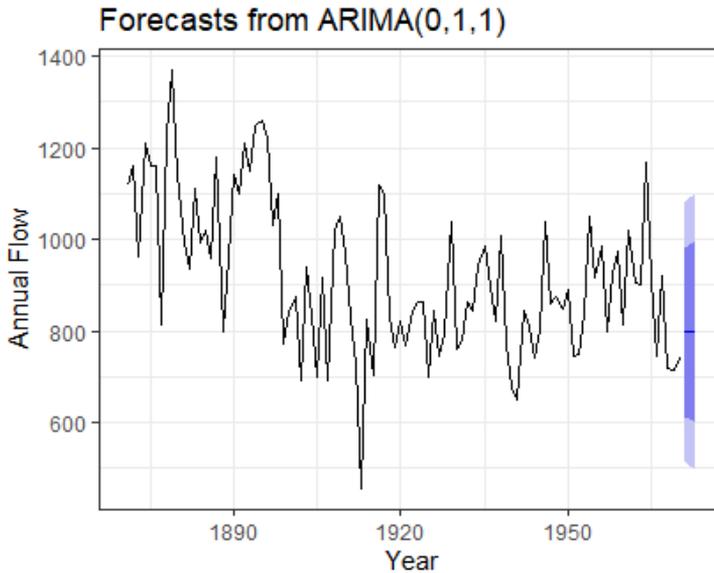


Figure 15.15 Three-year forecast for the Nile time series from a fitted ARIMA(0,1,1) model. The black line represents point estimates, and the light and dark blue bands represent the 80% and 95% confidence bands limits, respectively.

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.

A caveat on forecasting

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.

15.5 Going further

There are many good books on time-series analysis and forecasting. *Forecasting: Principles and Practice* (<http://otexts.com/fpp2>, 2018) 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

- Time series are important because they help us make future predictions based on past experience.
- R provides a wide array of data structures for holding time-series data. Base R offers classes for holding one (`ts`) or more than one (`mts`) series of observations recorded at regular intervals. The `xts` and `zoo` packages extends this to include observations recorded at irregular intervals.
- Time-series data stored as `xts` objects can be easily subsetted using bracket `[]` notation, and aggregated using `apply.period` functions.

- The `forecast` package provides several functions for visually exploring time-series data. The `autoplot()` function can be used to plot time-series data as `ggplot2` graphs. The `ma()` function can be used to smooth irregularities in a time-series in order to highlight trends. The `stl()` function can be used to decompose a time series into trend, seasonal, and irregular (residual) components.
- The `forecast` package can also be used to forecast future values of a time series. We covered two popular forecasting approaches - exponential models and autoregressive integrated moving average (ARIMA) models.