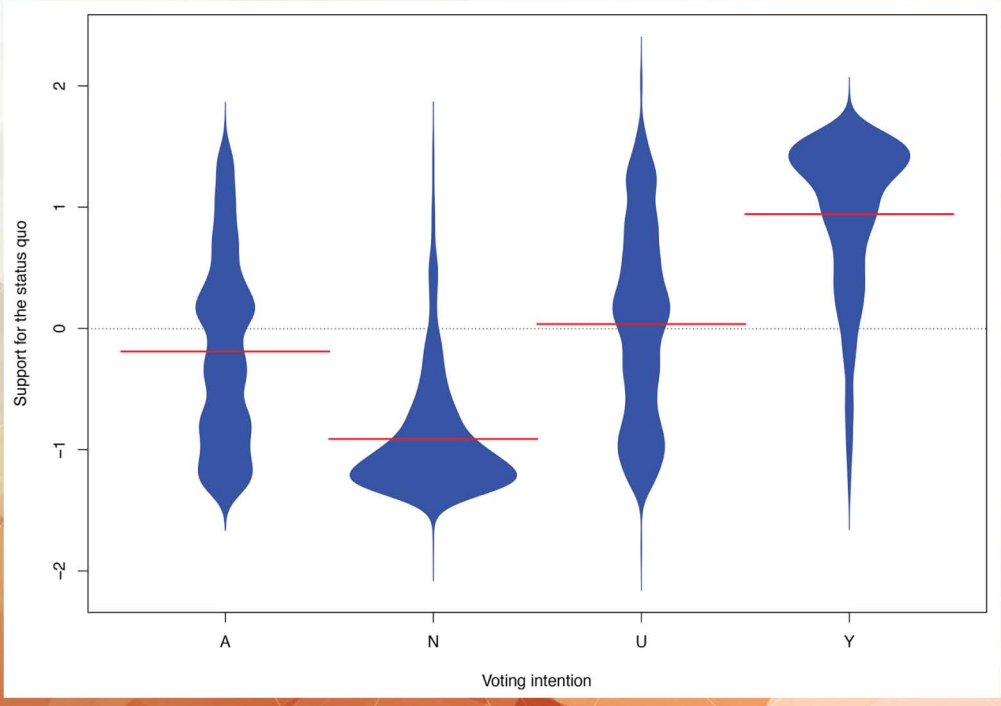
Chapman & Hall/CRC
Data Mining and Knowledge Discovery Series

EXPLORATORY DATA ANALYSIS USING R



Ronald K. Pearson

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

EXPLORATORY DATA ANALYSIS USING R

Chapman & Hall/CRC
Data Mining and Knowledge Series

Series Editor: Vipin Kumar

Computational Business Analytics
Subrata Das

Data Classification
Algorithms and Applications
Charu C. Aggarwal

Healthcare Data Analytics
Chandan K. Reddy and Charu C. Aggarwal

Accelerating Discovery
Mining Unstructured Information for Hypothesis Generation
Scott Spangler

Event Mining
Algorithms and Applications
Tao Li

Text Mining and Visualization
Case Studies Using Open-Source Tools
Markus Hofmann and Andrew Chisholm

Graph-Based Social Media Analysis
Ioannis Pitas

Data Mining
A Tutorial-Based Primer, Second Edition
Richard J. Roiger

Data Mining with R
Learning with Case Studies, Second Edition
Luís Torgo

Social Networks with Rich Edge Semantics
Quan Zheng and David Skillicorn

Large-Scale Machine Learning in the Earth Sciences
Ashok N. Srivastava, Ramakrishna Nemani, and Karsten Steinhaeuser

Data Science and Analytics with Python
Jesus Rogel-Salazar

Feature Engineering for Machine Learning and Data Analytics
Guozhu Dong and Huan Liu

Exploratory Data Analysis Using R
Ronald K. Pearson

For more information about this series please visit:
https://www.crcpress.com/Chapman--HallCRC-Data-Mining-and-Knowledge-Discovery-Series/book-series/CHDAMINODIS

EXPLORATORY DATA ANALYSIS USING R

Ronald K. Pearson

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper
Version Date: 20180312

International Standard Book Number-13: 978-1-1384-8060-5 (Hardback)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface

Much has been written about the abundance of data now available from the Internet and a great variety of other sources. In his aptly named 2007 book *Glut* [81], Alex Wright argued that the total quantity of data then being produced was approximately five *exabytes* per year (5×10^{18} bytes), more than the estimated total number of words spoken by human beings in our entire history. And that assessment was from a decade ago: increasingly, we find ourselves "drowning in a ocean of data," raising questions like "What do we do with it all?" and "How do we begin to make any sense of it?"

Fortunately, the open-source software movement has provided us with—at least partial—solutions like the *R* programming language. While *R* is not the only relevant software environment for analyzing data—*Python* is another option with a growing base of support—*R* probably represents the most flexible data analysis software platform that has ever been available. *R* is largely based on *S*, a software system developed by John Chambers, who was awarded the 1998 Software System Award by the Association for Computing Machinery (ACM) for its development; the award noted that *S* "has forever altered the way people analyze, visualize, and manipulate data."

The other side of this software coin is educational: given the availability and sophistication of *R*, the situation is analogous to someone giving you an F-15 fighter aircraft, fully fueled with its engines running. If you know how to fly it, this can be a great way to get from one place to another very quickly. But it is not enough to just have the plane: you also need to know how to take off in it, how to land it, and how to navigate from where you are to where you want to go. Also, you need to have an idea of where you do want to go. With *R*, the situation is analogous: the software can do a lot, but you need to know both how to use it and what you want to do with it.

The purpose of this book is to address the most important of these questions. Specifically, this book has three objectives:

1. To provide a basic introduction to *exploratory data analysis (EDA)*;

2. To introduce the range of "interesting"—good, bad, and ugly—features we can expect to find in data, and why it is important to find them;

3. To introduce the mechanics of using *R* to explore and explain data.

This book grew out of materials I developed for the course "Data Mining Using R" that I taught for the University of Connecticut Graduate School of Business. The students in this course typically had little or no prior exposure to data analysis, modeling, statistics, or programming. This was not universally true, but it was typical, so it was necessary to make minimal background assumptions, particularly with respect to programming. Further, it was also important to keep the treatment relatively non-mathematical: data analysis is an inherently mathematical subject, so it is not possible to avoid mathematics altogether, but for this audience it was necessary to assume no more than the minimum essential mathematical background.

The intended audience for this book is students—both advanced undergraduates and entry-level graduate students—along with working professionals who want a detailed but introductory treatment of the three topics listed in the book's title: data, exploratory analysis, and *R*. Exercises are included at the ends of most chapters, and an instructor's solution manual giving complete solutions to all of the exercises is available from the publisher.

Author

Ronald K. Pearson is a Senior Data Scientist with GeoVera Holdings, a property insurance company in Fairfield, California, involved primarily in the exploratory analysis of data, particularly text data. Previously, he held the position of Data Scientist with DataRobot in Boston, a software company whose products support large-scale predictive modeling for a wide range of business applications and are based on Python and R, where he was one of the authors of the `datarobot` *R* package. He is also the developer of the `GoodmanKruskal` *R* package and has held a variety of other industrial, business, and academic positions. These positions include both the DuPont Company and the Swiss Federal Institute of Technology (ETH Zürich), where he was an active researcher in the area of nonlinear dynamic modeling for industrial process control, the Tampere University of Technology where he was a visiting professor involved in teaching and research in nonlinear digital filters, and the Travelers Companies, where he was involved in predictive modeling for insurance applications. He holds a PhD in Electrical Engineering and Computer Science from the Massachusetts Institute of Technology and has published conference and journal papers on topics ranging from nonlinear dynamic model structure selection to the problems of disguised missing data in predictive modeling. Dr. Pearson has authored or co-authored five previous books, including *Exploring Data in Engineering, the Sciences, and Medicine* (Oxford University Press, 2011) and *Nonlinear Digital Filtering with Python*, co-authored with Moncef Gabbouj (CRC Press, 2016). He is also the developer of the *DataCamp* course on base *R* graphics.

Taylor & Francis
Taylor & Francis Group
http://taylorandfrancis.com

Chapter 1

Data, Exploratory Analysis, and R

1.1 Why do we analyze data?

The basic subject of this book is data analysis, so it is useful to begin by addressing the question of why we might want to do this. There are at least three motivations for analyzing data:

1. to understand what has happened or what is happening;

2. to predict what is likely to happen, either in the future or in other circumstances we haven't seen yet;

3. to guide us in making decisions.

The primary focus of this book is on *exploratory data analysis*, discussed further in the next section and throughout the rest of this book, and this approach is most useful in addressing problems of the first type: understanding our data. That said, the predictions required in the second type of problem listed above are typically based on mathematical models like those discussed in Chapters 5 and 10, which are optimized to give reliable predictions for data we have available, in the hope and expectation that they will also give reliable predictions for cases we haven't yet considered. In building these models, it is important to use *representative, reliable data*, and the exploratory analysis techniques described in this book can be extremely useful in making certain this is the case. Similarly, in the third class of problems listed above—making decisions—it is important that we base them on an accurate understanding of the situation and/or accurate predictions of what is likely to happen next. Again, the techniques of exploratory data analysis described here can be extremely useful in verifying and/or improving the accuracy of our data and our predictions.

1.2 The view from 90,000 feet

This book is intended as an introduction to the three title subjects—data, its exploratory analysis, and the *R* programming language—and the following sections give high-level overviews of each, emphasizing key details and interrelationships.

1.2.1 Data

Loosely speaking, the term "data" refers to a collection of details, recorded to characterize a source like one of the following:

- an entity, e.g.: family history from a patient in a medical study; manufacturing lot information for a material sample in a physical testing application; or competing company characteristics in a marketing analysis;
- an event, e.g.: demographic characteristics of those who voted for different political candidates in a particular election;
- a process, e.g.: operating data from an industrial manufacturing process.

This book will generally use the term "data" to refer to a rectangular array of observed values, where each row refers to a different observation of entity, event, or process characteristics (e.g., distinct patients in a medical study), and each column represents a different characteristic (e.g., diastolic blood pressure) recorded—or at least potentially recorded—for each row. In *R's* terminology, this description defines a *data frame*, one of *R's* key data types.

The `mtcars` data frame is one of many built-in data examples in *R*. This data frame has 32 rows, each one corresponding to a different car. Each of these cars is characterized by 11 variables, which constitute the columns of the data frame. These variables include the car's mileage (in miles per gallon, mpg), the number of gears in its transmission, the transmission type (manual or automatic), the number of cylinders, the horsepower, and various other characteristics. The original source of this data was a comparison of 32 cars from model years 1973 and 1974 published in *Motor Trend Magazine*. The first six records of this data frame may be examined using the `head` command in *R:*

```
head(mtcars)

##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

An important feature of data frames in *R* is that both rows and columns have names associated with them. In favorable cases, these names are informative, as they are here: the row names identify the particular cars being characterized, and the column names identify the characteristics recorded for each car.

A more complete description of this dataset is available through *R's* built-in help facility. Typing "help(mtcars)" at the *R* command prompt will bring up a help page that gives the original source of the data, cites a paper from the statistical literature that analyzes this dataset [39], and briefly describes the variables included. This information constitutes *metadata* for the `mtcars` data frame: metadata is "data about data," and it can vary widely in terms of its completeness, consistency, and general accuracy. Since metadata often provides much of our preliminary insight into the contents of a dataset, it is extremely important, and any limitations of this metadata—incompleteness, inconsistency, and/or inaccuracy—can cause serious problems in our subsequent analysis. For these reasons, discussions of metadata will recur frequently throughout this book. The key point here is that, potentially valuable as metadata is, we cannot afford to accept it uncritically: we should always cross-check the metadata with the actual data values, with our intuition and prior understanding of the subject matter, and with other sources of information that may be available.

As a specific illustration of this last point, a popular benchmark dataset for evaluating binary classification algorithms (i.e., computational procedures that attempt to predict a binary outcome from other variables) is the Pima Indians diabetes dataset, available from the UCI Machine Learning Repository, an important Internet data source discussed further in Chapter 4. In this particular case, the dataset characterizes female adult members of the Pima Indians tribe, giving a number of different medical status and history characteristics (e.g., diastolic blood pressure, age, and number of times pregnant), along with a binary diagnosis indicator with the value 1 if the patient had been diagnosed with diabetes and 0 if they had not. Several versions of this dataset are available: the one considered here was the UCI website on May 10, 2014, and it has 768 rows and 9 columns. In contrast, the data frame `Pima.tr` included in *R's* `MASS` package is a subset of this original, with 200 rows and 8 columns. The metadata available for this dataset from the UCI Machine Learning Repository now indicates that this dataset exhibits missing values, but there is also a note that prior to February 28, 2011 the metadata indicated that there were no missing values. In fact, the missing values in this dataset are not coded explicitly as missing with a special code (e.g., *R's* "NA" code), but are instead coded as zero. As a result, a number of studies characterizing binary classifiers have been published using this dataset as a benchmark where the authors were not aware that data values were missing, in some cases, quite a large fraction of the total observations. As a specific example, the serum insulin measurement included in the dataset is 48.7% missing.

Finally, it is important to recognize the essential role our *assumptions* about data can play in its subsequent analysis. As a simple and amusing example, consider the following "data analysis" question: how many planets are there orbiting the Sun? Until about 2006, the generally accepted answer was nine, with Pluto the outermost member of this set. Pluto was subsequently re-classified as a "dwarf planet," in part because a larger, more distant body was found in the Kuiper Belt and enough astronomers did not want to classify this object as the "tenth planet" that Pluto was demoted to dwarf planet status. In his book,

Is Pluto a Planet? [72], astronomer David Weintraub argues that Pluto should remain a planet, based on the following defining criteria for planethood:

1. the object must be too small to generate, or to have ever generated, energy through nuclear fusion;
2. the object must be big enough to be spherical;
3. the object must have a primary orbit around a star.

The first of these conditions excludes dwarf stars from being classed as planets, and the third excludes moons from being declared planets (since they orbit planets, not stars). Weintraub notes, however, that under this definition, there are at least 24 planets orbiting the Sun: the eight now generally regarded as planets, Pluto, and 15 of the largest objects from the asteroid belt between Mars and Jupiter and from the Kuiper Belt beyond Pluto. This example illustrates that definitions are both extremely important and not to be taken for granted: everyone knows what a planet is, don't they? In the broader context of data analysis, the key point is that unrecognized disagreements in the definition of a variable are possible between those who measure and record it, and those who subsequently use it in analysis; these discrepancies can lie at the heart of unexpected findings that turn out to be erroneous. For example, if we wish to combine two medical datasets, characterizing different groups of patients with "the same" disease, it is important that the same diagnostic criteria be used to declare patients "diseased" or "not diseased." For a more detailed discussion of the role of definitions in data analysis, refer to Sec. 2.4 of *Exploring Data in Engineering, the Sciences, and Medicine* [58]. (Although the book is generally quite mathematical, this is not true of the discussions of data characteristics presented in Chapter 2, which may be useful to readers of this book.)

1.2.2 Exploratory analysis

Roughly speaking, exploratory data analysis (EDA) may be defined as the art of looking at one or more datasets in an effort to understand the underlying structure of the data contained there. A useful description of how we might go about this is offered by Diaconis [21]:

> We look at numbers or graphs and try to find patterns. We pursue leads suggested by background information, imagination, patterns perceived, and experience with other data analyses.

Note that this quote suggests—although it does not strictly imply—that the data we are exploring consists of numbers. Indeed, even if our dataset contains nonnumerical data, our analysis of it is likely to be based largely on numerical characteristics computed from these nonnumerical values. As a specific example, categorical variables appearing in a dataset like "city," "political party affiliation," or "manufacturer" are typically tabulated, converted from discrete named values into counts or relative frequencies. These derived representations

can be particularly useful in exploring data when the number of levels—i.e., the number of distinct values the original variable can exhibit—is relatively small. In such cases, many useful exploratory tools have been developed that allow us to examine the character of these nonnumeric variables and their relationship with other variables, whether categorical or numeric. Simple graphical examples include boxplots for looking at the distribution of numerical values across the different levels of a categorical variable, or mosaic plots for looking at the relationship between categorical variables; both of these plots and other, closely related ones are discussed further in Chapters 2 and 3.

Categorical variables with many levels pose more challenging problems, and these come in at least two varieties. One is represented by variables like U.S. postal zipcode, which identifies geographic locations at a much finer-grained level than state does and exhibits about 40,000 distinct levels. A detailed discussion of dealing with this type of categorical variable is beyond the scope of this book, although one possible approach is described briefly at the end of Chapter 10. The second type of many-level categorical variable arises in settings where the inherent structure of the variable can be exploited to develop specialized analysis techniques. Text data is a case in point: the number of distinct words in a document or a collection of documents can be enormous, but special techniques for analyzing text data have been developed. Chapter 8 introduces some of the methods available in *R* for analyzing text data.

The mention of "graphs" in the Diaconis quote is particularly important since humans are much better at seeing patterns in graphs than in large collections of numbers. This is one of the reasons *R* supports so many different graphical display methods (e.g., scatterplots, barplots, boxplots, quantile-quantile plots, histograms, mosaic plots, and many, many more), and one of the reasons this book places so much emphasis on them. That said, two points are important here. First, graphical techniques that are useful to the data analyst in finding important structure in a dataset are not necessarily useful in explaining those findings to others. For example, large arrays of two-variable scatterplots may be a useful screening tool for finding related variables or anomalous data subsets, but these are extremely poor ways of presenting results to others because they essentially require the viewer to repeat the analysis for themselves. Instead, results should be presented to others using displays that highlight and emphasize the analyst's findings to make sure that the intended message is received. This distinction between *exploratory* and *explanatory* displays is discussed further in Chapter 2 on graphics in *R* and in Chapter 6 on crafting data stories (i.e., explaining your findings), but most of the emphasis in this book is on exploratory graphical tools to help us obtain these results.

The second point to note here is that the utility of any graphical display can depend strongly on exactly what is plotted, as illustrated in Fig. 1.1. This issue has two components: the mechanics of how a subset of data is displayed, and the choice of what goes into that data subset. While both of these aspects are important, the second is far more important than the first. Specifically, it is important to note that the form in which data arrives may not be the most useful for analysis. To illustrate, Fig. 1.1 shows two sets of plots, both constructed

```
library(MASS)
library(car)
par(mfrow=c(2,2))
truehist(mammals$brain)
truehist(log(mammals$brain))
qqPlot(mammals$brain)
title("Normal QQ-plot")
qqPlot(log(mammals$brain))
title("Normal QQ-plot")
```

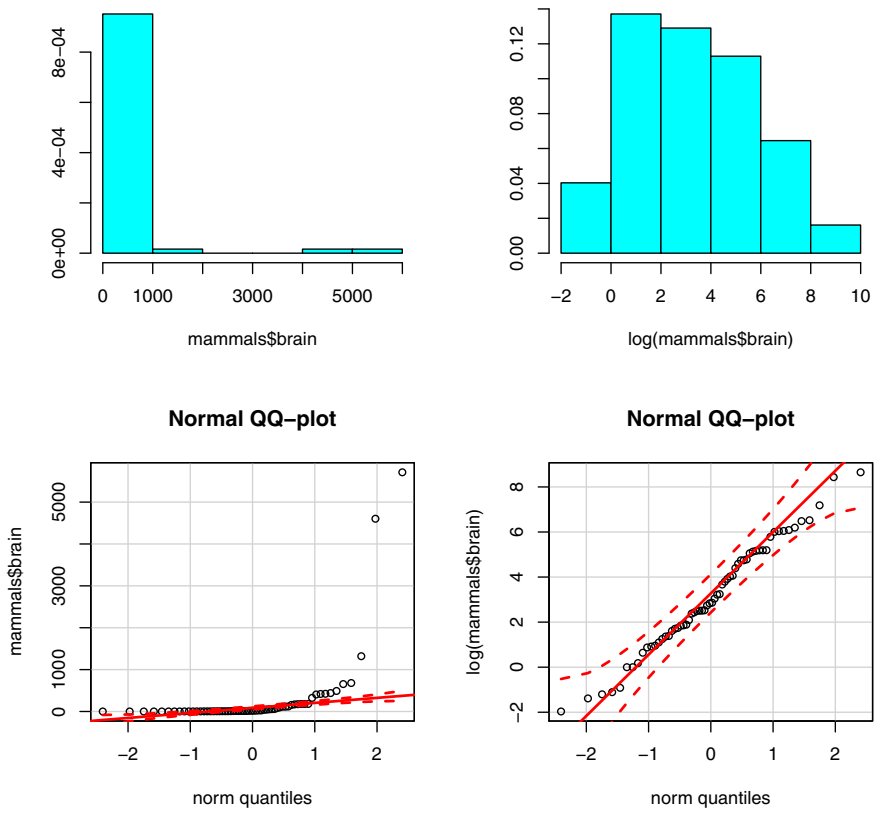


Figure 1.1: Two pairs of characterizations of the brain weight data from the `mammals` data frame: histograms and normal QQ-plots constructed from the raw data (left-hand plots), and from log-transformed data (right-hand plots).

from the `brain` element of the `mammals` dataset from the `MASS` package that lists body weights and brain weights for 62 different animals. This data frame is discussed further in Chapter 3, along with the characterizations presented

here, which are histograms (top two plots) and normal QQ-plots (bottom two plots). In both cases, these plots are attempting to tell us something about the distribution of data values, and the point of this example is that the extent to which these plots are informative depends strongly on how we prepare the data from which they are constructed. Here, the left-hand pair of plots were generated from the raw data values and they are much less informative than the right-hand pair of plots, which were generated from log-transformed data. In particular, these plots suggest that the log-transformed data exhibits a roughly Gaussian distribution, further suggesting that working with the log of brain weight may be more useful than working with the raw data values. This example is revisited and discussed in much more detail in Chapter 3, but the point here is that exactly what we plot—e.g., raw data values vs. log-transformed data values—sometimes matters a lot more than how we plot it.

Since it is one of the main themes of this book, a much more extensive introduction to exploratory data analysis is given in Chapter 3. Three key points to note here are, first, that exploratory data analysis makes extensive use of graphical tools, for the reasons outlined above. Consequently, the wide and growing variety of graphical methods available in *R* makes it a particularly suitable environment for exploratory analysis. Second, exploratory analysis often involves characterizing many different variables and/or data sources, and comparing these characterizations. This motivates the widespread use of simple and well-known summary statistics like means, medians, and standard deviations, along with other, less well-known characterizations like the MAD scale estimate introduced in Chapter 3. Finally, third, an extremely important aspect of exploratory data analysis is the search for "unusual" or "anomalous" features in a dataset. The notion of an *outlier* is introduced briefly in Sec. 1.3, but a more detailed discussion of this and other data anomalies is deferred until Chapter 3, where techniques for detecting these anomalies are also discussed.

1.2.3 Computers, software, and R

To use *R*—or any other data analysis environment—involves three basic tasks:

1. Make the data you want to analyze available to the analysis software;
2. Perform the analysis;
3. Make the results of the analysis available to those who need them.

In this chapter, all of the data examples come from built-in data frames in *R*, which are extremely convenient for teaching or learning *R*, but in real data analysis applications, making the data available for analysis can require significant effort. Chapter 4 focuses on this problem, but to understand its nature and significance, it is necessary to understand something about how computer systems are organized, and this is the subject of the next section. Related issues arise when we attempt to make analysis results available for others, and these issues are also covered in Chapter 4. Most of the book is devoted to various aspects of step (2) above—performing the analysis—and the second section below

briefly addresses the question of "why use *R* and not something else?" Finally, since this is a book about using *R* to analyze data, some key details about the structure of the *R* language are presented in the third section below.

General structure of a computing environment

In his book, *Introduction to Data Technologies* [56, pp. 211–214], Paul Murrell describes the general structure of a computing environment in terms of the following six components:

1. the *CPU* or *central processing unit* is the basic hardware that does all of the computing;
2. the *RAM* or *random access memory* is the *internal* memory where the CPU stores and retrieves results;
3. the *keyboard* is the standard interface that allows the user to submit requests to the computer system;
4. the *screen* is the graphical display terminal that allows the user to see the results generated by the computer system;
5. the *mass storage*, typically a "hard disk," is the *external* memory where data and results can be stored permanently;
6. the *network* is an external connection to the outside world, including the Internet but also possibly an *intranet* of other computers, along with peripheral devices like printers.

Three important distinctions between internal storage (i.e., RAM) and external storage (i.e., mass storage) are, first, that RAM is typically several orders of magnitude faster to access than mass storage; second, that RAM is *volatile*—i.e., the contents are lost when the power is turned off—while mass storage is not; and, third, that mass storage can accommodate much larger volumes of data than RAM can. (As a specific example, the computer being used to prepare this book has 4GB of installed RAM and just over 100 times as much disk storage.) A practical consequence is that both the data we want to analyze and any results we want to save need to end up in mass storage so they are not lost when the computer power is turned off. Chapter 4 is devoted to a detailed discussion of some of the ways we can move data into and out of mass storage.

These differences between RAM and mass storage are particularly relevant to *R* since most *R* functions require all data—both the raw data and the internal storage required to keep any temporary, intermediate results—to fit in RAM. This makes the computations faster, but it limits the size of the datasets you can work with in most cases to something less than the total installed RAM on your computer. *In some applications, this restriction represents a serious limitation on R's applicability.* This limitation is recognized within the R community and continuing efforts are being made to improve the situation.

Closely associated with the CPU is the *operating system*, which is the software that runs the computer system, making useful activity possible. That is, the operating system coordinates the different components, establishing and managing file systems that allow datasets to be stored, located, modified, or deleted; providing user access to programs like *R*; providing the support infrastructure required so these programs can interact with network resources, etc. In addition to the general computing infrastructure provided by the operating system, to analyze data it is necessary to have programs like *R* and possibly others (e.g., database programs). Further, these programs must be compatible with the operating system: on popular desktops and enterprise servers, this is usually not a problem, although it can become a problem for older operating systems. For example, Section 2.2 of the *R FAQ* document available from the *R* "Help" tab notes that "support for Mac OS Classic ended with R 1.7.1."

With the growth of the Internet as a data source, it is becoming increasingly important to be able to retrieve and process data from it. Unfortunately, this involves a number of issues that are well beyond the scope of this book (e.g., parsing HTML to extract data stored in web pages). A brief introduction to the key ideas with some simple examples is given in Chapter 4, but for those needing a more thorough treatment, Murrell's book is highly recommended [56].

Data analysis software

A key element of the data analysis chain (acquire → analyze → explain) described earlier is the choice of data analysis software. Since there are a number of possibilities here, why *R*? One reason is that *R* is a free, open-source language, available for most popular operating systems. In contrast, commercially supported packages must be purchased, in some cases for a lot of money.

Another reason to use *R* in preference to other data analysis platforms is the enormous range of analysis methods supported by *R's* growing universe of add-on packages. These packages support analysis methods from many branches of statistics (e.g., traditional statistical methods like ANOVA, ordinary least squares regression, and *t*-tests, Bayesian methods, and robust statistical procedures), machine learning (e.g., random forests, neural networks, and boosted trees), and other applications like text analysis. This availability of methods is important because it greatly expands the range of data exploration and analysis approaches that can be considered. For example, if you wanted to use the multivariate outlier detection method described in Chapter 9 based on the MCD covariance estimator in another framework—e.g., Microsoft Excel—you would have to first build these analysis tools yourself, and then test them thoroughly to make sure they are really doing what you want. All of this takes time and effort just to be able to get to the point of actually analyzing your data.

Finally, a third reason to adopt *R* is its growing popularity, undoubtedly fueled by the reasons just described, but which is also likely to promote the continued growth of new capabilities. A survey of programming language popularity by the Institute of Electrical and Electronics Engineers (IEEE) has been taken for the last several years, and a summary of the results as of July 18,

2017, was available from the website:

http://spectrum.ieee.org/computing/software/the-2017-top-ten-programming-languages

The top six programming languages on this list were, in descending order: Python, C, Java, C++, C#, and R. Note that the top five of these are general-purpose languages, all suitable for at least two of the four programming environments considered in the survey: web, mobile, desktop/enterprise, and embedded. In contrast, *R* is a specialized data analysis language that is only suitable for the desktop/enterprise environment. The next data analysis language in this list was the commercial package MATLAB®, ranked 15th.

The structure of R

The *R* programming language basically consists of three components:

- a set of *base R packages*, a required collection of programs that support language infrastructure and basic statistics and data analysis functions;
- a set of *recommended packages*, automatically included in almost all *R* installations (the `MASS` package used in this chapter belongs to this set);
- a very large and growing set of *optional add-on packages*, available through the Comprehensive R Archive Network (CRAN).

Most *R* installations have all of the base and recommended packages, with at least a few selected add-on packages. The advantage of this language structure is that it allows extensive customization: as of February 3, 2018, there were 12,086 packages available from CRAN, and new ones are added every day. These packages provide support for everything from rough and fuzzy set theory to the analysis of twitter tweets, so it is an extremely rare organization that actually needs *everything* CRAN has to offer. Allowing users to install only what they need avoids massive waste of computer resources.

Installing packages from CRAN is easy: the *R* graphical user interface (GUI) has a tab labeled "Packages." Clicking on this tab brings up a menu, and selecting "Install packages" from this menu brings up one or two other menus. If you have not used the "Install packages" option previously in your current *R* session, a menu appears asking you to select a CRAN mirror; these sites are locations throughout the world with servers that support CRAN downloads, so you should select one near you. Once you have done this, a second menu appears that lists all of the *R* packages available for download. Simply scroll down this list until you find the package you want, select it, and click the "OK" button at the bottom of the menu. This will cause the package you have selected to be downloaded from the CRAN mirror and installed on your machine, along with all other packages that are required to make your selected package work. For example, the `car` package used to generate Fig. 1.1 requires a number of other packages, including the quantile regression packge `quantreg`, which is automatically downloaded and installed when you install the `car` package.

It is important to note that *installing* an *R* package makes it available for you to use, but this does *not* "load" the package into your current *R* session. To do this, you must use the `library()` function, which works in two different ways. First, if you enter this function without any parameters—i.e., type "library()" at the *R* prompt—it brings up a new window that lists all of the packages that have been installed on your machine. To use any of these packages, it is necessary to use the `library()` command again, this time specifying the name of the package you want to use as a parameter. This is shown in the code appearing at the top of Fig. 1.1, where the `MASS` and `car` packages are loaded:

```
library(MASS)
library(car)
```

The first of these commands loads the `MASS` package, which contains the `mammals` data frame and the `truehist` function to generate histograms, and the second loads the `car` package, which contains the `qqPlot` function used to generate the normal QQ-plots shown in Fig. 1.1.

1.3 A representative R session

To give a clear view of the essential material covered in this book, the following paragraphs describe a simple but representative *R* analysis session, providing a few specific illustrations of what *R* can do. The general task is a typical preliminary data exploration: we are given an unfamiliar dataset and we begin by attempting to understand what is in it. In this particular case, the dataset is a built-in data example from *R*—one of many such examples included in the language—but the preliminary questions explored here are analogous to those we would ask in characterizing a dataset obtained from the Internet, from a data warehouse of customer data in a business application, or from a computerized data collection system in a scientific experiment or an industrial process monitoring application. Useful preliminary questions include:

1. How many records does this dataset contain?
2. How many fields (i.e., variables) are included in each record?
3. What kinds of variables are these? (e.g., real numbers, integers, categorical variables like "city" or "type," or something else?)
4. Are these variables always observed? (i.e., is missing data an issue? If so, how are missing values represented?)
5. Are the variables included in the dataset the ones we were expecting?
6. Are the values of these variables consistent with what we expect?
7. Do the variables in the dataset seem to exhibit the kinds of relationships we expect? (Indeed, what relationships do we expect, and why?)

The example presented here does not address all of these questions, but it does consider some of them and it shows how the *R* programming environment can be useful in both answering and refining these questions.

Assuming *R* has been installed on your machine (if not, see the discussion of installing *R* in Chapter 11), you begin an interactive session by clicking on the *R* icon. This brings up a window where you enter commands at the ">" prompt to tell *R* what you want to do. There is a toolbar at the top of this display with a number of tabs, including "Help" which provides links to a number of useful documents that will be discussed further in later parts of this book. Also, when you want to end your *R* session, type the command "q()" at the ">" prompt: this is the "quit" command, which terminates your *R* session. Note that the parentheses after "q" are important here: this tells *R* that you are calling a *function* that, in general, does something to the argument or arguments you pass it. In this case, the command takes no arguments, but failing to include the parentheses will cause *R* to search for an object (e.g., a vector or data frame) named "q" and, if it fails to find this, display an error message. Also, note that when you end your *R* session, you will be asked whether you want to save your workspace image: if you answer "yes," *R* will save a copy of all of the commands you used in your interactive session in the file `.Rhistory` in the current working directory, making this command history—but not the *R* objects created from these commands—available for your next *R* session.

Also, in contrast to some other languages—*SAS*® is a specific example—it is important to recognize that *R* is *case-sensitive:* commands and variables in lower-case, upper-case, or mixed-case are *not* the same in *R*. Thus, while a *SAS* procedure like `PROC FREQ` may be equivalently invoked as `proc freq` or `Proc Freq`, the *R* commands `qqplot` and `qqPlot` are *not* the same: `qqplot` is a function in the `stats` package that generates quantile-quantile plots comparing two empirical distributions, while `qqPlot` is a function in the `car` package that generates quantile-quantile plots comparing a data distribution with a theoretical reference distribution. While the tasks performed by these two functions are closely related, the details of what they generate are different, as are the details of their syntax. As a more immediate illustration of *R's* case-sensitivity, recall that the function `q()` "quits" your *R* session; in contrast, unless you define it yourself or load an optional package that defines it, the function `Q()` does not exist, and invoking it will generate an error message, something like this:

```
Q()

## Error in Q():  could not find function "Q"
```

The specific dataset considered in the following example is the `whiteside` data frame from the `MASS` package, one of the *recommended* packages included with almost all *R* installations, as noted in Sec. 1.2.3. Typing "??whiteside" at the ">" prompt performs a fuzzy search through the documentation for all packages available to your *R* session, bringing up a page with all approximate matches on the term. Clicking on the link labeled `MASS::whiteside` takes us to a documentation page with the following description:

> Mr Derek Whiteside of the UK Building Research Station recorded the weekly gas consumption and average external temperature at his own house in south-east England for two heating seasons, one of 26 weeks before, and one of 30 weeks after cavity-wall insulation was installed. The object of the exercise was to assess the effect of the insulation on gas consumption.

To analyze this dataset, it is necessary to first make it available by loading the `MASS` package with the `library()` function as described above:

```
library(MASS)
```

An *R* data frame is a rectangular array of *N* records—each represented as a row—with *M* fields per record, each representing a value of a particular variable for that record. This structure may be seen by applying the `head` function to the `whiteside` data frame, which displays its first few records:

```
head(whiteside)

##    Insul Temp Gas
## 1 Before -0.8 7.2
## 2 Before -0.7 6.9
## 3 Before  0.4 6.4
## 4 Before  2.5 6.0
## 5 Before  2.9 5.8
## 6 Before  3.2 5.8
```

More specifically, the first line lists the field names, while the next six lines show the values recorded in these fields for the first six records of the dataset. Recall from the discussion above that the `whiteside` data frame characterizes the weekly average heating gas consumption and the weekly average outside temperature for two successive winters, the first before Whiteside installed insulation in his house, and the second after. Thus, each record in this data frame represents one weekly observation, listing whether it was made before or after the insulation was installed (the `Insul` variable), the average outside temperature, and the average heating gas consumption.

A more detailed view of this data frame is provided by the `str` function, which returns structural characterizations of essentially any *R* object. Applied to the `whiteside` data frame, it returns the following information:

```
str(whiteside)

## 'data.frame': 56 obs. of  3 variables:
##  $ Insul: Factor w/ 2 levels "Before","After": 1 1 1 1 1 1 1 1 1 1 ...
##  $ Temp : num  -0.8 -0.7 0.4 2.5 2.9 3.2 3.6 3.9 4.2 4.3 ...
##  $ Gas  : num  7.2 6.9 6.4 6 5.8 5.8 5.6 4.7 5.8 5.2 ...
```

Here, the first line tells us that `whiteside` is a data frame, with 56 observations (rows or records) and 3 variables. The second line tells us that the first variable, `Insul`, is a *factor* variable with two levels: "Before" and "After." (Factors are

an important *R* data type used to represent categorical data, introduced briefly in the next paragraph.) The third and fourth lines tell us that `Temp` and `Gas` are numeric variables. Further, all lines except the first provide summaries of the first few (here, 10) values observed for each variable. For the numeric variables, these values are the same as those shown with the `head` command presented above, while for factors, `str` displays a numerical index indicating which of the possible levels of the variable is represented in each of the first 10 records.

Because factor variables are both very useful and somewhat more complex in their representation than numeric variables, it is worth a brief digression here to say a bit more about them. Essentially, factor variables in *R* are special vectors used to represent categorical variables, encoding them with two components: a level, corresponding to the value we see (e.g., "Before" and "After" for the factor `Insul` in the `whiteside` data frame), and an index that maps each element of the vector into the appropriate level:

```
x <- whiteside$Insul
str(x)

##  Factor w/ 2 levels "Before","After": 1 1 1 1 1 1 1 1 1 1 ...

x[2]

## [1] Before
## Levels: Before After
```

Here, the `str` characterization tells us how many levels the factor has and what the names of those levels are (i.e., two levels, named "Before" and "After"), but the values `str` displays are the indices instead of the levels (i.e., the first 10 records list the the first value, which is "Before"). *R* also supports character vectors and these could be used to represent categorical variables, but an important difference is that the levels defined for a factor variable represent its only possible values: attempting to introduce a new value into a factor variable fails, generating a missing value instead, with a warning. For example, if we attempted to change the second element of this factor variable from "Before" to "Unknown," we would get a warning about an invalid factor level and that the attempted assignment resulted in this element having the missing value `NA`. In contrast, if we convert `x` in this example to a character vector, the new value assignment attempted above now works:

```
x <- as.character(whiteside$Insul)
str(x)

##  chr [1:56] "Before" "Before" "Before" "Before" "Before" "Before" ...

x[2]

## [1] "Before"

x[2] <- "Unknown"
str(x)

##  chr [1:56] "Before" "Unknown" "Before" "Before" "Before" "Before" ...
```

In addition to `str` and `head`, the `summary` function can also provide much useful information about data frames and other *R* objects. In fact, `summary` is an example of a *generic* function in *R*, that can do different things depending on the attributes of the object we apply it to. Generic functions are discussed further in Chapters 2 and 7, but when the generic `summary` function is applied to a data frame like `whiteside`, it returns a relatively simple characterization of the values each variable can assume:

```
summary(whiteside)

##     Insul         Temp             Gas
##  Before:26   Min.   :-0.800   Min.   :1.300
##  After :30   1st Qu.: 3.050   1st Qu.:3.500
##              Median : 4.900   Median :3.950
##              Mean   : 4.875   Mean   :4.071
##              3rd Qu.: 7.125   3rd Qu.:4.625
##              Max.   :10.200   Max.   :7.200
```

This result may be viewed as a table with one column for each variable in the `whiteside` data frame—`Insul`, `Temp`, and `Gas`—with a column format that depends on the type of variable being characterized. For the two-level factor `Insul`, the `summary` result gives the number of times each possible level occurs: 26 records list the value "Before," while 30 list the value "After." For the numeric variables, the result consists of two components: one is the mean value—i.e., the average of the variable over all records in the dataset—while the other is *Tukey's five-number summary*, consisting of these five numbers:

1. the *sample minimum*, defined as the smallest value of x in the dataset;
2. the *lower quartile*, defined as the value x_L for which 25% of the data satisfies $x \leq x_L$ and the other 75% of the data satisfies $x > x_L$;
3. the *sample median*, defined as the "middle value" in the dataset, the value that 50% of the data values do not exceed and 50% do exceed;
4. the *upper quartile*, defined as the value x_U for which 75% of the data satisfies $x \leq x_U$ and the other 25% of the data satisfies $x > x_U$;
5. the *sample maximum*, defined as the largest value of x in the dataset.

This characterization has the advantage that it can be defined for any sequence of numbers and its complexity does not depend on how many numbers are in the sequence. In contrast, the complete table of counts for an L-level categorical variable consists of L numbers: for variables like `Insul` in the `whiteside` data frame, $L = 2$, so this characterization is simple. For a variable like "State" with 50 distinct levels (i.e., one for each state in the U.S.), this table has 50 entries. For this reason, the characterization returned by the `summary` function for categorical variables consists of the complete table if $L \leq 6$, but if $L > 6$, it lists only the five most frequently occurring levels, lumping all remaining levels into a single "other" category.

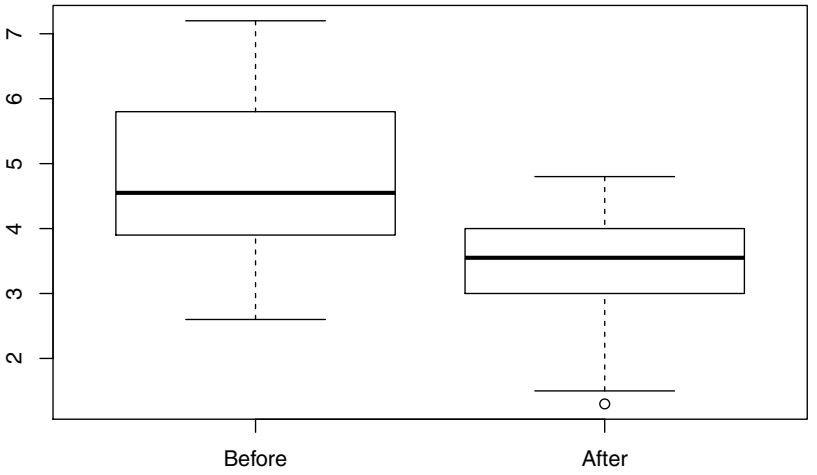


Figure 1.2: Side-by-side boxplot comparison of the "Before" and "After" subsets of the `Gas` values from the `whiteside` data frame.

An extremely useful graphical representation of Tukey's five-number summary is the *boxplot*, particularly useful in showing how the distribution of a numerical variable depends on subsets defined by the different levels of a factor. Fig. 1.2 shows a side-by-side boxplot summary of the `Gas` variable for subsets of the `whiteside` data frame defined by the `Insul` variable. This summary was generated by the following *R* command, which uses the *R formula interface* (i.e., `Gas ~ Insul`) to request boxplots of the ranges of variation of the `Gas` variable for each distinct level of the `Insul` factor:

```
boxplot(Gas ~ Insul, data = whiteside)
```

The left-hand plot—above the x-axis label "Before"—illustrates the boxplot in its simplest form: the short horizontal lines at the bottom and top of the plot correspond to the sample minimum and maximum, respectively; the wider, heavier line in the middle of the plot represents the median; and the lines at the top and bottom of the "box" in the plot correspond to the upper and lower quartiles. The "After" boxplot also illustrates a common variation on the "basic" boxplot based strictly on Tukey's five-number summary. Specifically, at the bottom of this boxplot—below the "sample minimum" horizontal line—is a single open circle, representing an *outlier*, a data value that appears inconsistent with the majority of the data (here, "unusually small"). In this boxplot, the

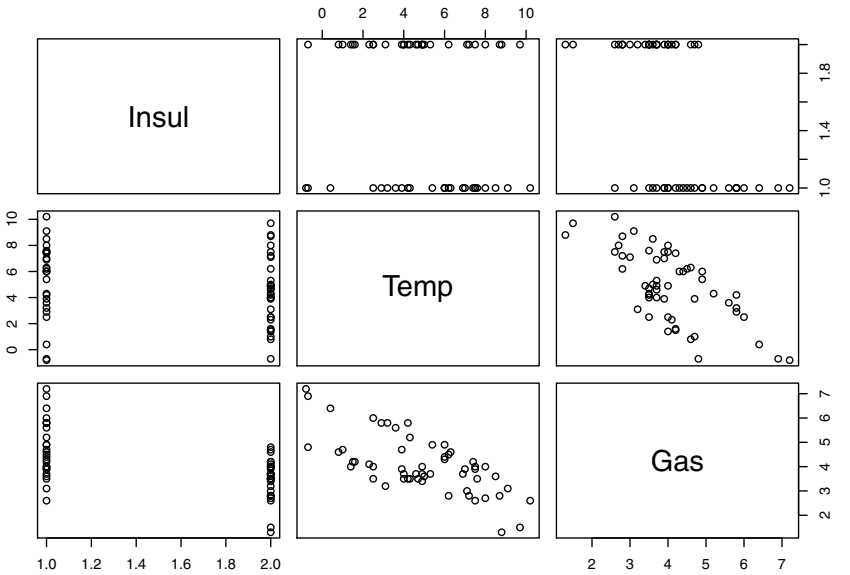


Figure 1.3: The 3×3 plot array generated by `plot(whiteside)`.

bottom horizontal line does not represent the sample minimum, but the "smallest non-outlying value" where the determination of what values are "outlying" versus "non-outlying" is made using a simple rule discussed in Chapter 3.

Fig. 1.3 shows the results of applying the `plot` function to the `whiteside` data frame. Like `summary`, the `plot` function is also generic, producing a result that depends on the nature of the object to which it is applied. Applied to a data frame, `plot` generates a matrix of *scatterplots*, showing how each variable relates to the others. More specifically, the diagonal elements of this plot array identify the variable that defines the x-axis in all of the other plots in that column of the array and the y-axis in all of the other plots in that row of the array. Here, the two scatterplots involving `Temp` and `Gas` are simply plots of the numerical values of one variable against the other. The four plots involving the factor variable `Insul` have a very different appearance, however: in these plots, the two levels of this variable ("Before" and "After") are represented by their numerical codes, 1 and 2. Using these numerical codes provides a basis for including factor variables in a scatterplot array like the one shown here, although the result is often of limited utility. Here, one point worth noting is that the plots involving `Insul` and `Gas` do show that the `Gas` values are generally smaller when `Insul` has its second value. In fact, this level corresponds to "After" and this difference reflects the important detail that less heating gas was consumed after insulation was installed in the house than before.

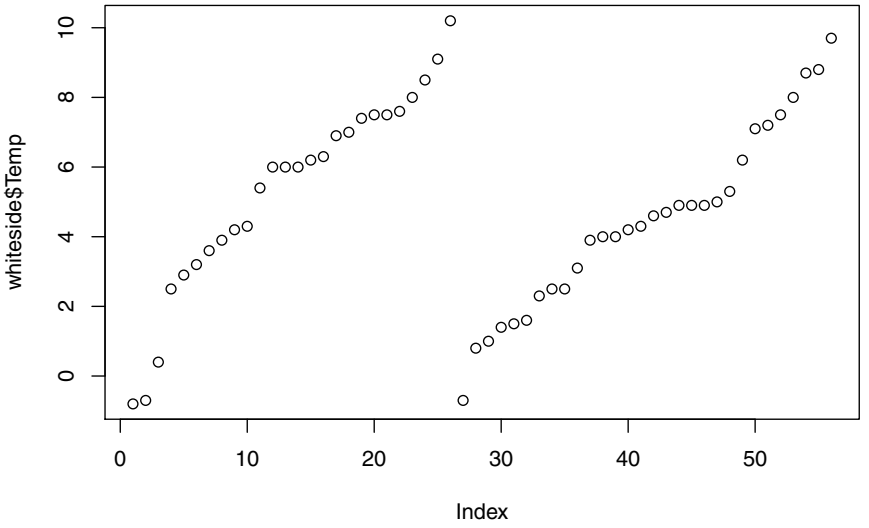


Figure 1.4: The result of `plot(whiteside$Temp)`.

In Fig. 1.4, applying `plot` to the `Temp` variable from the `whiteside` data frame shows how `Temp` varies with its record number in the data frame. Here, these values appear in two groups—one of 26 points, followed by another of 30 points—but within each group, they appear in ascending order. From the data description presented earlier, we might expect these values to represent average weekly winter temperatures recorded in successive weeks during the two heating seasons characterized in the dataset. Instead, these observations have been ordered from coldest to warmest within each heating season. While such unexpected structure often makes no difference, it sometimes does; the key point here is that plotting the data can reveal it.

Fig. 1.5 shows the result of applying the `plot` function to the factor variable `Insul`, which gives us a *barplot*, showing how many times each possible value for this categorical variable appears in the data frame. In marked contrast to this plot, note that Fig. 1.3 used the numerical level representation for `Insul`: "Before" corresponds to the first level of the variable—represented as 1 in the plot—while "After" corresponds to the second level of the variable, represented as 2 in the plot. This was necessary so that the `plot` function could present scatterplots of the "value" of each variable against the corresponding "value" of every other variable. Again, these plots emphasize that `plot` is a generic function, whose result depends on the type of *R* object plotted.

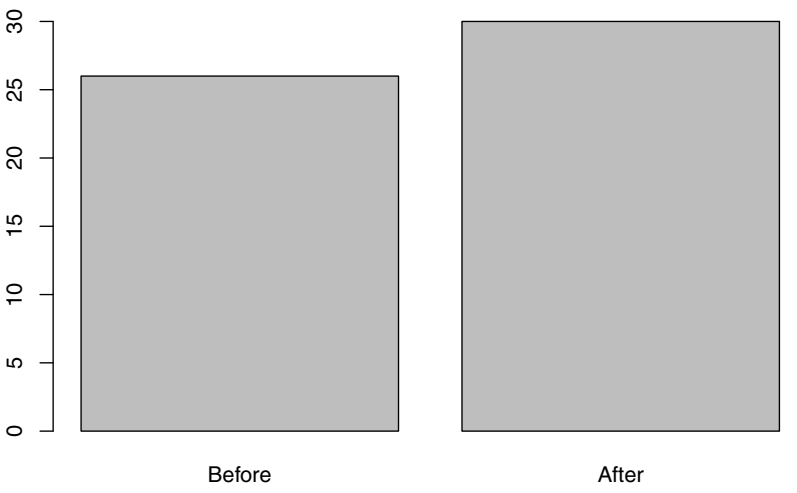


Figure 1.5: The result of `plot(whiteside$Insul)`.

The rest of this section considers some refinements of the scatterplot between weekly average heating gas consumption and average outside temperature appearing in the three-by-three plot array in Fig. 1.3. The intent is to give a "preview of coming attractions," illustrating some of the ideas and techniques that will be discussed in detail in subsequent chapters.

The first of these extensions is Fig. 1.6, which plots `Gas` versus `Temp` with different symbols for the two heating seasons (i.e., "Before" and "After"). The following *R* code generates this plot, using open triangles for the "Before" data and solid circles for the "After" data:

```
plot(whiteside$Temp, whiteside$Gas, pch=c(6,16)[whiteside$Insul])
```

The approach used here to make the plotting symbol depend on the `Insul` value for each point is described in Chapter 2, which gives a detailed discussion of generating and refining graphical displays in *R*. Here, the key point is that using different plotting symbols for the "Before" and "After" points in this example highlights the fact that the relationship between heating gas consumption and outside temperature is substantially different for these two collections of points, as we would expect from the original description of the dataset. Another important point is that generating this plot with different symbols for the two sets of data points is not difficult.

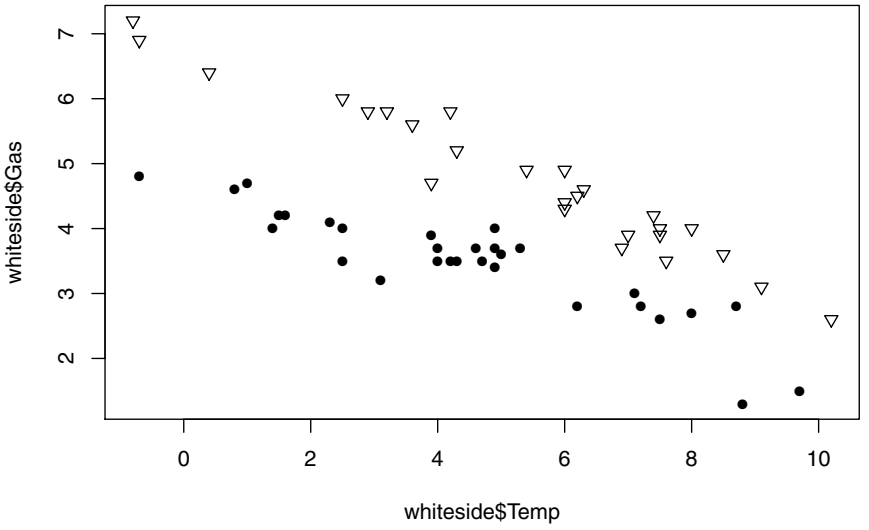


Figure 1.6: Scatterplot of `Gas` versus `Temp` from the `whiteside` data frame, with distinct point shapes for the "Before" and "After" data subsets.

Fig. 1.7 shows a simple but extremely useful modification of Fig. 1.6: the inclusion of a legend that tells us what the different point shapes mean. This is also quite easy to do, using the `legend` function, which can be used to put a box anywhere we like on the plot, displaying the point shapes we used together with descriptive text to tell us what each shape means. The *R* code used to add this legend is shown in Fig. 1.7.

The last example considered here adds two *reference lines* to the plot shown in Fig. 1.7. These lines are generated using the *R* function `lm`, which fits *linear regression models*, discussed in detail in Chapter 5. These models represent the simplest type of *predictive model*, a topic discussed more generally in Chapter 10 where other classes of predictive models are introduced. The basic idea is to construct a mathematical model that predicts a response variable from one or more other, related variables. In the `whiteside` data example considered here, these models predict the weekly average heating gas consumed as a linear function of the measured outside temperature. To obtain two reference lines, one model is fit for each of the data subsets defined by the two values of the `Insul` variable. Alternatively, we could obtain the same results by fitting a single linear regression model to the dataset, using both the `Temp` and `Insul` variables as predictors. This alternative approach is illustrated in Chapter 5 where this example is revisited.

```
plot(whiteside$Temp, whiteside$Gas, pch=c(6,16)[whiteside$Insul])
legend(x="topright",legend=c("Insul = Before","Insul = After"), pch=c(6,16))
```

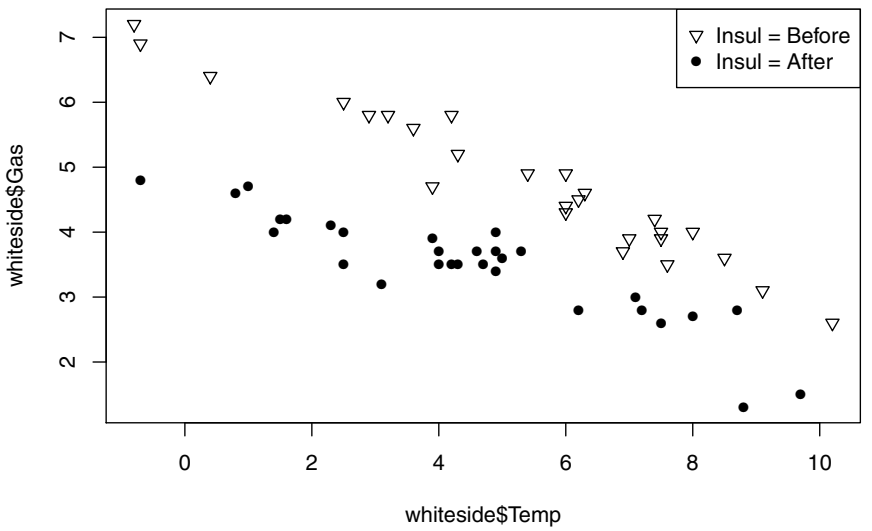


Figure 1.7: Scatterplot from Fig. 1.6 with a legend added to identify the two data subsets represented with different point shapes.

Fig. 1.8 is the same as Fig. 1.7, but with these reference lines added. As with the different plotting points, these lines are drawn with different line types. The *R* code listed at the top of Fig. 1.8 first re-generates the previous plot, then fits the two regression models just described, and finally draws in the lines determined by these two models. Specifically, the dashed "Before" line is obtained by fitting one model to only the "Before" points and the solid "After" line is obtained by fitting a second model to only the "After" points.

1.4 Organization of this book

This book is organized as two parts. The first focuses on analyzing data in an interactive *R* session, while the second introduces the fundamentals of *R* programming, emphasizing the development of custom functions since this is the aspect of programming that most *R* users find particularly useful. The second part also presents more advanced treatments of topics introduced in the first, including text analysis, a second look at exploratory data analysis, and an introduction to some more advanced aspects of predictive modeling.

```
plot(whiteside$Temp, whiteside$Gas, pch=c(6,16)[whiteside$Insul])
legend(x="topright",legend=c("Insul = Before","Insul = After"), pch=c(6,16))
Model1 <- lm(Gas ~ Temp, data = whiteside, subset = which(Insul == "Before"))
Model2 <- lm(Gas ~ Temp, data = whiteside, subset = which(Insul == "After"))
abline(Model1, lty=2)
abline(Model2)
```

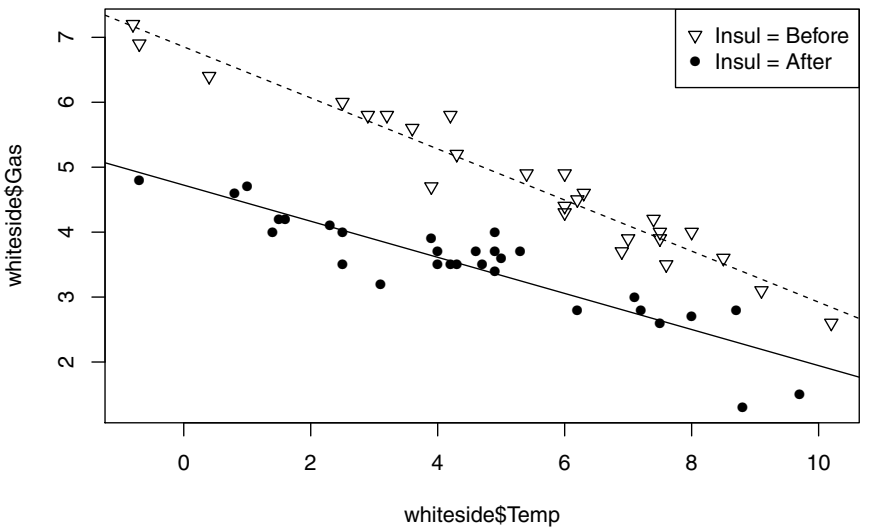


Figure 1.8: Scatterplot from Fig. 1.7 with linear regression lines added, representing the relationships between `Gas` and `Temp` for each data subset.

More specifically, the first part of this book consists of the first seven chapters, including this one. As noted, one of the great strengths of *R* is its variety of powerful data visualization procedures, and Chapter 2 provides a detailed introduction to several of these. This subject is introduced first because it provides those with little or no prior *R* experience a particularly useful set of tools that they can use right away. Specific topics include both basic plotting tools and some simple customizations that can make these plots much more effective. In fact, *R* supports several different graphics environments which, unfortunately, don't all play well together. The most important distinction is that between *base graphics*—the primary focus of Chapter 2—and the alternative *grid graphics* system, offering greater flexibility at the expense of being somewhat harder to use. While base graphics are used for most of the plots in this book, a number of important *R* packages use grid graphics, including the increasingly popular `ggplot2` package. As a consequence, some of the things we might want to do—e.g., add reference lines or put several different plots into a single array—can

fail if we attempt to use base graphics constructs with plots generated by an *R* package based on grid graphics. For this reason, it is important to be aware of the different graphics systems available in *R*, even if we work primarily with base graphics as we do in this book. Since *R* supports color graphics, two sets of color figures are included in this book, the first collected as Chapter 2.8 and the second collected as Chapter 9.10 in the second part of the book.

Chapter 3 introduces the basic notions of exploratory data analysis (EDA), focusing on specific techniques and their implementation in *R*. Topics include descriptive statistics like the mean and standard deviation, essential graphical tools like scatterplots and histograms, an overview of data anomalies (including brief discussions of different types, why they are too important to ignore, and a few of the things we can do about them), techniques for assessing or visualizing relationships between variables, and some simple summaries that are useful in characterizing large datasets. This chapter is one of two devoted to EDA, the second being Chapter 9 in the second part of the book, which introduces some more advanced concepts and techniques.

The introductory *R* session example presented in Sec. 1.3 was based on the `whiteside` data frame, an internal *R* dataset included in the `MASS` package. One of the great conveniences in learning *R* is the fact that so many datasets are available as built-in data objects. Conversely, for *R* to be useful in real-world applications, it is obviously necessary to be able to bring the data we want to analyze into our interactive *R* session. This can be done in a number of different ways, and the focus of Chapter 4 is on the features available for bringing external data into our *R* session and writing it out to be available for other applications. This latter capability is crucial since, as emphasized in Sec. 1.2.3, everything within our active *R* session exists in RAM, which is volatile and disappears forever when we exit this session; to preserve our work, we need to save it to a file. Specific topics discussed in Chapter 4 include data file types, some of *R's* commands for managing external files (e.g., finding them, moving them, copying or deleting them), some of the built-in procedures *R* provides to help us find and import data from the Internet, and a brief introduction to the important topic of *databases*, the primary tool for storing and managing data in businesses and other large organizations.

Chapter 5 is the first of two chapters introducing the subject of *predictive modeling*, the other being Chapter 10 in the second part of the book. Predictive modeling is perhaps most simply described as the art of developing mathematical models—i.e., equations—that predict a *response variable* from one or more *covariates* or *predictor variables*. Applications of this idea are extremely widespread, ranging from the estimation of the probability that a college baseball player will go on to have a successful career in the major leagues described in Michael Lewis' popular book *Moneyball* [51], to the development of mathematical models for industrial process control to predict end-use properties that are difficult or impossible to measure directly from easily measured variables like temperatures and pressures. The simplest illustration of predictive modeling is the problem of fitting a straight line to the points in a two-dimensional scatterplot; both because it is relatively simple and because a number of important

practical problems can be re-cast into exactly this form, Chapter 5 begins with a detailed treatment of this problem. From there, more general *linear regression* problems are discussed in detail, including the problem of *overfitting* and how to protect ourselves from it, the use of multiple predictors, the incorporation of categorical variables, how to include interactions and transformations in a linear regression model, and a brief introduction to robust techniques that are resistant to the potentially damaging effects of outliers.

When we analyze data, we are typically attempting to understand or predict something that is of interest to others, which means we need to show them what we have found. Chapter 6 is concerned with the art of crafting *data stories* to meet this need. Two key details are, first, that different audiences have different needs, and second, that most audiences want a summary of what we have done and found, and not a complete account with all details, including wrong turns and loose ends. The chapter concludes with three examples of moderate-length data stories that summarize what was analyzed and why, and what was found without going into all of the gory details of how we got there (some of these details are important for the readers of this book even if they don't belong in the data story; these details are covered in other chapters).

The second part of this book consists of Chapters 7 through 11, introducing the topics of *R* programming, the analysis of text data, second looks at exploratory data analysis and predictive modeling, and the challenges of organizing our work. Specifically, Chapter 7 introduces the topic of writing programs in *R*. Readers with programming experience in other languages may want to skip or skim the first part of this chapter, but the *R*-specific details should be useful to anyone without a lot of prior *R* programming experience. As noted in the Preface, this book assumes no prior programming experience, so this chapter starts simply and proceeds slowly. It begins with the question of why we should learn to program in *R* rather than just rely on canned procedures, and continues through essential details of both the structure of the language (e.g., data types like vectors, data frames, and lists; control structures like for loops and if statements; and functions in *R*), and the mechanics of developing programs (e.g., editing programs, the importance of comments, and the art of debugging). The chapter concludes with five programming examples, worked out in detail, based on the recognition that many of us learn much by studying and modifying code examples that are known to work.

Text data analysis requires specialized techniques, beyond those covered in most statistics and data analysis texts, which are designed to work with numerical or simple categorical variables. Most of this book is also concerned with these techniques, but Chapter 8 provides an introduction to the issues that arise in analyzing text data and some of the techniques developed to address them. One key issue is that, to serve as a basis for useful data analysis, our original text data must be converted into a relevant set of numbers, to which either general or highly text-specific quantitative analysis procedures may be applied. Typically, the analysis of text data involves first breaking it up into relevant chunks (e.g., words or short word sequences), which can then be counted, forming the basis for constructing specialized data structures like *term-document matrices*,

to which various types of quantitative analysis procedures may then be applied. Many of the techniques required to do this type of analysis are provided by the *R* packages `tm` and `quanteda`, which are introduced and demonstrated in the discussion presented here. Another key issue in analyzing text data is the importance of *preprocessing* to address issues like inconsistencies in capitalization and punctuation, and the removal of numbers, special symbols, and non-informative *stopwords* like "a" or "the." Text analysis packages like `tm` and `quanteda` include functions to perform these operations, but many of them can also be handled using low-level string handling functions like `grep`, `gsub`, and `strsplit` that are available in base *R*. Both because these functions are often extremely useful adjuncts to specialized text analysis packages and because they represent an easy way of introducing some important text analysis concepts, these functions are also treated in some detail in Chapter 8. Also, these functions—along with a number of others in *R*—are based on *regular expressions*, which can be extremely useful but also extremely confusing to those who have not seen them before; Chapter 8 includes an introduction to regular expressions.

Chapter 9 provides a second look at exploratory data analysis, building on the ideas presented in Chapter 3 and providing more detailed discussions of some of the topics introduced there. For example, Chapter 3 introduces the idea of using random variables and probability distributions to model uncertainty in data, along with some standard random variable characterizations like the mean and standard deviation. The basis for this discussion is the popular Gaussian distribution, but this distribution is only one of many and it is not always appropriate. Chapter 9 introduces some alternatives, with examples to show why they are sometimes necessary in practice. Other topics introduced in Chapter 9 include confidence intervals and statistical significance, association measures that summarize the relationship between variables of different types, multivariate outliers and their impact on standard association measures, and a number of useful graphical tools that build on these ideas. Since color greatly enhances the utility of some of these tools, the second group of color figures follows, as Chapter 9.10.

Following this second look at exploratory data analysis, Chapter 10 builds on the discussion of linear regression models presented in Chapter 5, introducing a range of extensions, including *logistic regression* for binary responses (e.g., the *Moneyball* problem: estimate the probability of having a successful major league career, given college baseball statistics), more general approaches to these *binary classification problems* like decision trees, and a gentle introduction to the increasingly popular arena of machine learning models like random forests and boosted trees. Because predictive modeling is a vast subject, the treatment presented here is by no means complete, but Chapters 5 and 10 should provide a useful introduction and serve as a practical starting point for those wishing to learn more.

Finally, Chapter 11 introduces the larger, broader issues of "managing stuff": data files, *R* code that we have developed, analysis results, and even the *R* packages we are using and their versions. Initially, this may not seem either very interesting or very important, but over time, our view is likely to change.

In particular, as we get further into a data analysis effort, the data sources we are working with change (e.g., we obtain newer data, better data, or simply additional data), our intermediate analysis results accumulate ("first, I looked at the relationship between Variable A and Variable B, which everybody said was critically important, but the results didn't seem to support that, so next I looked at the relationship between Variables A and C, which looked much more promising, and then somebody suggested I consider Variables D and E, ..."), and different people need different summaries of our results. Often, these components accumulate rapidly enough that it may take a significant amount of time and effort to dig up what we need to either explain exactly what we did before or to re-do our analysis with a "simple" modification ("Can you drop the records associated with the Florida stores from your analysis, and oh yeah, use the 2009 through 2012 data instead of the 2008 through 2013 data? Thanks."). The purpose of Chapter 11 is to introduce some simple ideas and tools available in *R* to help in dealing with these issues before our analytical life becomes complicated enough to make some of them extremely painful.

1.5 Exercises

1: Section 1.2.2 considered the `mammals` data frame from the `MASS` package, giving body weights and brain weights for 62 animals. Discussions in later chapters will consider the `Animals2` data frame from the `robustbase` package which gives the same characterizations for a slightly different set of animals. In both cases, the row names for these data frames identify these animals, and the objective of this exercise is to examine the differences between the animals characterized in these data frames:

   1a. The `rownames` function returns a vector of row names for a data frame, and the `intersect` function computes the intersection of two sets, returning a vector of their common elements. Using these functions, construct and display the vector `commonAnimals` of animal names common to both data frames. How many animals are included in this set?

   1b. The `setdiff` function returns a vector of elements contained in one set but not the other: `setdiff(A, B)` returns a vector of elements in set `A` that are not in set `B`. Use this function to display the animals present in `mammals` that are not present in `Animals2`.

   1c. Use the `setdiff` function to display the animals present in `Animals2` that are not present in `mammals`.

   1d. Can you give a simple characterization of these differences between these sets of animals?

2: Figure 1.1 in the text used the `qqPlot` function from the `car` package to show that the log of the `brain` variable (brain weights) from the `mammals` data frame in the `MASS` package was reasonably consistent with a Gaussian

distribution. Generate the corresponding plot for the brain weights from the `Animals2` data frame from the `robustbase` package. Does the same conclusion hold for these brain weights?

3: As discussed at the end of Section 1.2.3, calling the `library` function with no arguments brings up a new window that displays a list of the *R* packages that have been previously installed and are thus available for our use by calling `library` again with one of these package names. Alternatively, the results returned by the `library` function when it is called without arguments can be assigned to an *R* data object. The purpose of this exercise is to explore the structure of this object:

   3a. Assign the return value from the `library()` call without arguments to the *R* object `libReturn`;

   3b. This *R* object is a named list: using the `str` function, determine how many elements this object has and the names of those elements;

   3c. One of these elements is a character array that provides the information normally displayed in the pop-up window: what are the names of the columns of this matrix, and how many rows does it have?

4: The beginning of Section 1.3 poses seven questions that are often useful to ask about a new dataset. The last three of these questions deal with our expectations and therefore cannot be answered by strictly computational methods, but the first four can be:

   4a. For the `cabbages` dataset from the `MASS` package, refer back to these questions and use the `str` function to answer the first three of them.

   4b. The combination of functions `length(which(is.na(x)))` returns the number of missing elements of the vector `x`. Use this combination to answer the fourth question: how many missing values does each variable in `cabbages` exhibit?

5: The generic `summary` function was introduced in Section 1.3, where it was applied to the `whiteside` data frame. While the results returned by this function do not directly address all of the first four preliminary exploration questions considered in Exercise 4, this function is extremely useful in cases where we do have missing data. One such example is the `Chile` data frame from the `car` package. Use this function to answer the following question: how many missing observations are associated with each variable in the `Chile` data frame?

6: As noted in the discussion in Section 1.2.2, the Gaussian distribution is often assumed as a reasonable approximation to describe how numerical variables are distributed over their ranges of possible values. This assumption is not always reasonable, but as illustrated in the lower plots in Figure 1.1, the `qqPlot` function from the `car` package can be used as an informal graphical test of the reasonableness of this assumption.

6a. Apply the `qqPlot` function to the `HeadWt` variable from the `cabbages` data frame: does the Gaussian assumption appear reasonable here?

6b. Does this assumption appear reasonable for the `VitC` variable?

7: The example presented in Section 1.3 used the `boxplot` function with the formula interface to compare the range of heating gas values (`Gas`) for the two different levels of the `Insul` variable. Use this function to answer the following two questions:

7a. The `Cult` variable exhibits two distinct values, representing different cabbage cultivars: does there appear to be a difference in cabbage head weights (`HeadWt`) between these cultivars?

7b. Does there appear to be a difference in vitamin C contents (`VitC`) between these cultivars?

8: One of the points emphasized throughout this book is the utility of *scatterplots*, i.e., plots of one variable against another. Using the `plot` function, generate a scatterplot of the vitamin C content (`VitC`) versus the head weight (`HeadWt`) from the `cabbages` dataset.

9: Another topic discussed in this book is *predictive modeling*, which uses mathematical models to predict one variable from another. The `lm` function was used to generate reference lines shown in Figure 1.8 for two subsets of the `whiteside` data from the `MASS` package. As a preview of the results discussed in Chapter 5, this problem asks you to use the `lm` function to build a model that predicts `VitC` from `HeadWt`. Refer back to the code included with Figure 1.8, noting that the `subset` argument is not needed here (i.e., you need only the formula expression and the `data` argument). Specifically:

9a. Use the `lm` function to build a model that predicts `VitC` from `HeadWt`, saving the result as `cabbageModel`.

9b. Apply the `summary` function to `cabbageModel` to obtain a detailed description of this predictive model. Don't worry for now about the details: the interpretation of these summary results will be discussed in Chapter 5.

10: Closely related to both scatterplots and linear regression analysis is the *product-moment correlation coefficient*, introduced in Chapter 9. This coefficient is a numerical measure of the tendency for the variations in one variable to track those of another variable: positive values indicate that increases in one variable are associated with increases in the other, while negative values indicate that increases in one variable are associated with decreases in the other. The correlation between `x` and `y` is computed using the `cor` function as `cor(x,y)`. Use this function to compute the correlation between `HeadWt` and `VitC` from the `cabbages` data frame: do these characteristics vary together or in opposite directions? Is this consistent with your results from Exercise 8?

Chapter 2

Graphics in R

It has been noted both that graphical data displays can be extremely useful in understanding what is in a dataset, and that one of *R's* strengths is its range of available graphical tools. Several of these tools were demonstrated in Chapter 1, but the focus there was on specific examples and what they can tell us. The focus of this chapter is on how to generate useful data displays, with the axes we want, the point sizes and shapes we want, the titles we want, and explanatory legends and other useful additions put where we want them.

2.1 Exploratory vs. explanatory graphics

In their book, Iliinsky and Steele [44] draw a distinction between *infographics* and *data visualizations*, describing an infographic as an aesthetically rich, manually drawn representation of a specific data source, in contrast to a data visualization that is algorithmically drawn and "often aesthetically barren (i.e., data is not decorated)," but much more easily regenerated or modified, and much richer in data details. Here, we are concerned with data visualizations, which Iliinksy and Steele further divide into *exploratory graphics*, designed to help us understand what is in a dataset, and *explanatory graphics*, designed to convey our findings to others. Both exploratory and explanatory graphics are relevant here, since this book is concerned with both exploratory data analysis and conveying our results to others, which is the primary objective of Chapter 6 on data stories.

In describing exploratory visualizations, Iliinsky and Steele note that they are appropriate when you are attempting to understand what is in a large collection of data, and they offer two key observations [44, p. 7]. The first is reminiscent of the working definition of exploratory data analysis from Persi Diaconis [21] offered in Chapter 1:

> When you need to get a sense of what's inside your data set, translating it into a visual medium can help you quickly identify its features, including interesting curves, lines, trends, and anomalous outliers.

Their second observation concerns the level of detail that is appropriate to exploratory analysis:

> Exploration is generally best done at a high level of granularity. There may be a whole lot of noise in your data, but if you oversimplify or strip out too much information, you could end up missing something important.

In contrast, the purpose of explanatory visualizations is to convey the analyst's conclusions about what is in the data to others. Consequently, it is important in explanatory visualizations to remove or minimize data details that might obscure the message. Iliinsky and Steele make the point this way [44, p. 8]:

> Whoever your audience is, the story you are trying to tell (or the answer you are trying to share) is *known to you at the outset,* and therefore you can design to specifically accommodate and highlight that story. In other words, you'll need to make certain *editorial decisions* about which information stays in, and which is distracting or irrelevant and should come out. This is a process of selecting focused data that will support the story you are trying to tell.

The following two examples illustrate some of the key differences between exploratory and explanatory visualizations. Both are based on the `UScereal` data frame from the `MASS` package, which describes 11 characteristics of 65 breakfast cereals available for sale in the U.S., based mostly on information taken from the package label required by the U.S. Food and Drug Administration.

Fig. 2.1 represents a graphical data display that is best suited for exploration, constructed by applying the `plot` function to the entire data frame:

```
plot(UScereal, las = 2)
```

Specifically, this figure shows a plot array—a useful construct discussed further in Sec. 2.6—with one scatterplot for each pair of variables in the data frame. The diagonal elements of this array list the name of the variable appearing in the x-axis of all plots in that column, and the y-axis of all plots in that row. Since there are 11 variables in this data frame, the result is an array of 110 plots, making the result visually daunting at first glance. Further, because there are so many plots included in this array, each one is so small that it is impossible to see much detail in any individual plot. Nevertheless, this array represents a useful tool for preliminary data exploration because it allows us to quickly scan the plots to see whether any strong relationships appear to exist between any of the variable pairs. Here, there appear to be strong relationships between `fat` and `calories` (row 2, column 4 or vice versa: row 4, column 2), between `carbo` and `calories` (row 2, column 7 and vice versa), and between `potassium` and `fibre` (row 6, column 10 and vice versa). In addition, this display makes it clear that certain variables—e.g., `shelf` and `vitamins`—exhibit only a few distinct values. While this information can all be obtained using a combination of other displays

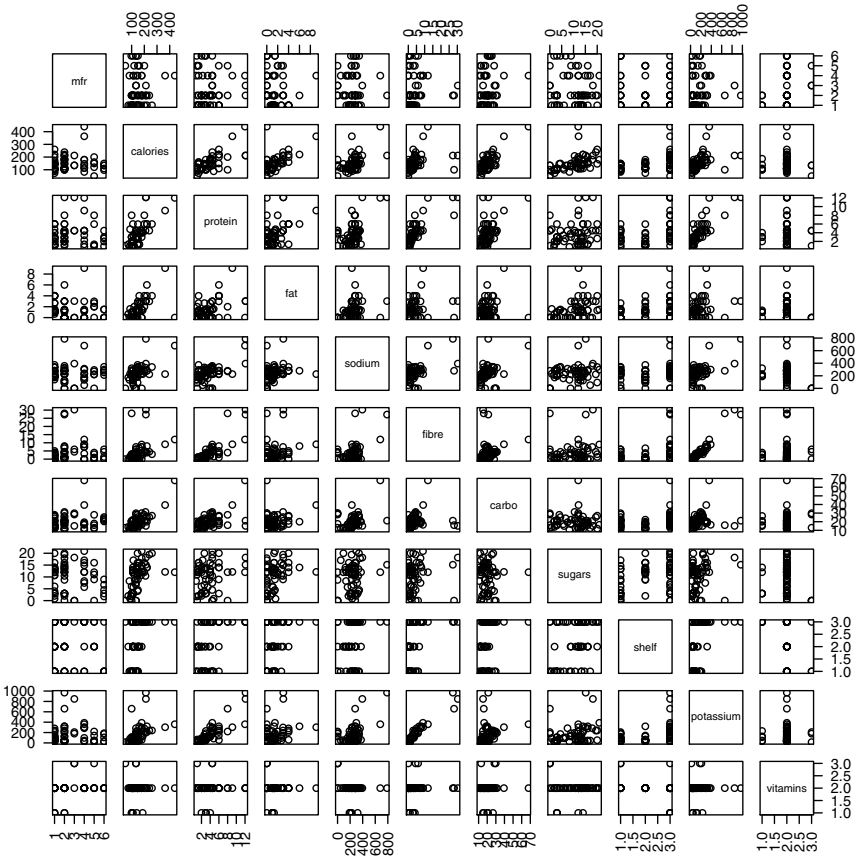


Figure 2.1: Array of pairwise scatterplots summarizing the UScereal data frame.

and/or nongraphical tools in *R*, this quick and simple plot array does provide a lot of useful preliminary information if we look at it carefully enough. That said, this plot array is *not* a good explanatory visualization because it contains far too much extraneous detail for any story we might wish to tell about any individual variable pair.

In contrast, Fig. 2.2 presents a much more detailed view of one of these scatterplots—that between the `calories` and `sugars` variables—augmented with a robust regression line emphasizing the general trend seen in most of this data, and with labels that explicitly identify two glaring outliers. Specifically, the dashed line in this plot represents the predictions of a robust linear regression model, generated using the `lmrob` function from the `robustbase` package discussed in Chapter 5. The key point here is that this dashed line highlights the trend our eye sees in the data if we ignore the two outlying points. These

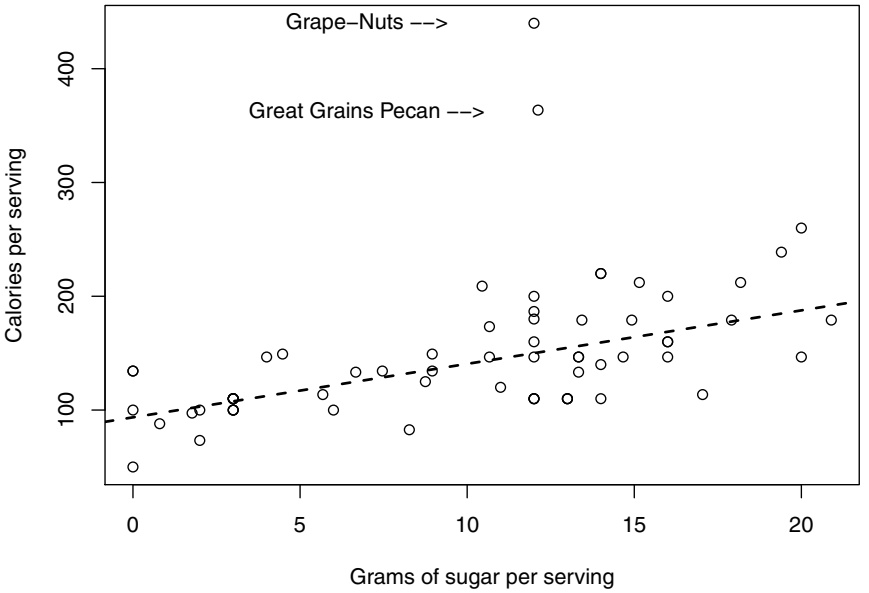


Figure 2.2: Annotated scatterplot showing the relationship between calories per serving and grams of sugar per serving from the UScereal data frame.

points correspond to the two cereals identified in the plot, representing much higher-calorie cereals than any of the others in the dataset. Fig. 2.2 tells a much more detailed story about the relationship between the variables `sugars` and `calories` from the `UScereal` data frame than we could infer from Fig. 2.1. Thus, Fig. 2.2 represents a much more effective way to describe or explain the relationship we have seen between `sugars` and `calories` to others.

2.2 Graphics systems in R

As noted, *R* supports many different graphical tools, but it is important to note that the underlying graphics systems on which these tools are built come in different flavors, and that tools built on different graphics systems generally don't play well together. The simplest of these systems is *base graphics*, described in Sec. 2.2.1 and used to create almost all of the graphical displays presented in this book. The other major graphics system in *R* is *grid graphics*, described in Sec. 2.2.2 and forming the basis for both *lattice graphics*, described in Sec. 2.2.3, and the `ggplot2` graphics package, described very briefly in Sec. 2.2.4.

Function	Object type	Nature of plot generated
`plot`	Many	Depends on the object type
`barplot`	Numeric	Bar plot (Sec. 2.5.2)
`boxplot`	Formula, numeric, or list	Boxplot summary (Chapter 3)
`hist`	Numeric	Histogram (Chapter 3)
`sunflowerplot`	Numeric + Numeric	Sunflower plot (Chapter 3)
`mosaicplot`	Formula or table	Mosaic plot (Chapter 3)
`symbols`	Multiple numeric	Bubbleplots, etc. (Sec. 2.5.3)

Table 2.1: A few of the more common base graphics functions.

2.2.1 Base graphics

The terms *base graphics* or *traditional graphics* [57] refer to the graphics system originally built into the *R* language. Because it is typically the default graphics system and is, in many respects, the easiest to learn, it is the primary graphics system used in this book, with exceptions noted when they arise. Probably the most common base graphics function is `plot`, described in detail in Sec. 2.3. As discussed in Chapter 1, this is a *generic function*, whose results depend on the type of *R* object we ask it to plot, an important concept discussed further in Sec. 2.3.2. Also, the detailed appearance of base graphics displays is partially controlled by a collection of 72 graphics parameters discussed further in Sec. 2.3.3. In addition, these displays can be further customized by using what Murrell calls *low-level* plotting functions [57] that add lines, points, text, and other details to an existing plot, discussed in Sec. 2.4.

Table 2.1 lists a few of the more common base graphics functions, along with the types of *R* objects they can accept as plot data arguments and the types of plot they generate. As noted, the basic `plot` function listed there is generic, accepting many different *R* object types and generating many different types of plots as a result; examples illustrating the range of plots possible with this function are presented in Sec. 2.3. The other functions listed in Table 2.1 are less flexible in the range of object types they accept, but as examples presented in Sec. 2.5 and in Chapter 3 illustrate, these functions can be extremely useful in generating both exploratory and explanatory data visualizations.

2.2.2 Grid graphics

As Murrell notes [57], almost all *R* graphics functions are ultimately based on the *graphics engine* represented by the `grDevices` package, which supports the lowest-level interface between *R* and the devices that display our graphics, handling details like fonts, colors, and display formats. Traditional or base graphics

functions are based on the `graphics` package, while *grid graphics* is based on the `grid` package (which Murrell developed). Murrell gives the following description of this package [57, p. 18]:

> The **g**rid package provides a separate set of basic tools. It does not provide functions for drawing complete plots, so it is not often used directly to produce statistical plots. It is more common to use one of the graphics packages that are built on top of **g**rid, especially either the **l**attice package or the **g**gplot2 package.

In addition to these large graphics systems—described briefly in the following sections—it is important to note that certain data analysis packages in *R* are also built on grid graphics. A specific example is the `vcd` package, which provides a number of extremely useful visualization tools for categorical data. If we want to modify these plots—e.g., add annotations, construct multiple plot arrays, etc.—it is necessary to use grid graphics to do this.

While a detailed introduction to grid graphics is beyond the scope of this book, it is important to be aware of its existence, its general incompatibility with base graphics, and its basic structure. A key component of grid graphics is the *viewport*, which Murrell defines as "a power facility for defining regions" [57, p. 174]. To construct a graphical display using the `grid` package, the basic steps are these:

- create a viewport;
- put a collection of graphic objects in the viewport;
- render the viewport to obtain a graphical display.

A useful introduction to the `grid` package, with a number of very good examples, is Murrell's package vignette "**g**rid Graphics." The package is also supported by about a dozen other vignettes, giving more detailed discussions of different aspects of working with grid graphics, and additional details are given in Part II of his book [57].

Finally, it is important to note two other points. First, a potential source of confusion is the existence of the `grid` *function*, which is part of the *base graphics system* and *not* related to the `grid` package. The base graphics function `grid` adds a rectangular grid to an existing base graphics plot; refer to the results from `help(grid)` for details. The second important point is that Murrell has also developed the `gridBase` package which allows both grid and base graphics to be used together. For an introduction to what is possible and some preliminary ideas on how to use this package, refer to the `gridBase` package vignette, "Integrating Grid Graphics Output with Base Graphics Output."

2.2.3 Lattice graphics

As noted, one of the complete graphics systems in *R* that is based on the `grid` package is *lattice graphics*, implemented in the `lattice` package. This package

provides an alternative implementation of many of the standard plotting functions available in base graphics, including scatterplots, bar charts, boxplots, histograms, and QQ-plots. Two of the primary advantages of this package over base graphics are, first, that many prefer the lattice default options (e.g., colors, point shapes, spacing, and labels) over the corresponding base defaults [57, p. 123], and, second, that lattice graphics provides simple implementations of certain additional features. One example, illustrated in Fig. 2.3, is the *multipanel conditioning plot*, which shows how the relationship between two variables depends on a third categorical *conditioning variable*. Specifically, Fig. 2.3 shows six interrelated scatterplots, each describing the relationship between the variables `Horsepower` and `MPG.city` from the `Cars93` data frame in the `MASS` package, but only for a single level of the categorical conditioning variable `Cylinders`. The code to generate this plot is extremely simple:

```
library(lattice)
xyplot(MPG.city ~ Horsepower | Cylinders, data = Cars93)
```

The first line here loads the `lattice` package, which makes the scatterplot function `xyplot` available for use. The second line applies this function, which supports *R's* standard formula interface. Specifically, the first argument represents a three-component formula: the variable `MPG.city`, appearing to the left of the `~` symbol, represents the response variable to be plotted on the y-axes of all plots; the variable `Horsepower`, appearing to the right of this symbol but to the left of the symbol `|`, defines the x-axis in all of these plots; and the variable `Cylinders` that appears to the right of this symbol is the categorical conditioning variable. Thus, the `xyplot` function constructs one scatterplot of `MPG.city` versus `Horsepower` for each distinct value of `Cylinders` and displays them in the format shown in Fig. 2.3. The `data` parameter in the above function call specifies the data frame containing these variables.

Another useful feature of the `lattice` package is the `group` argument, which allows different groups within a dataset (e.g., distinct `Cylinders` values in the previous example) to be represented by different point shapes in a single scatterplot. In addition, a legend is automatically generated that identifies the groups and their associated plotting symbols. Nevertheless, these capabilities come at a price [57, p. 135]:

> One advantage of the lattice graphics system is that it can produce extremely sophisticated plots from relatively simple expressions, especially with its multipanel conditioning feature. However, the cost of this is that the task of adding simple annotations to a lattice plot, such as adding extra lines or text, is more complex compared to the same task in traditional graphics.

It is for this reason that this book focuses on traditional (i.e., base) graphics.

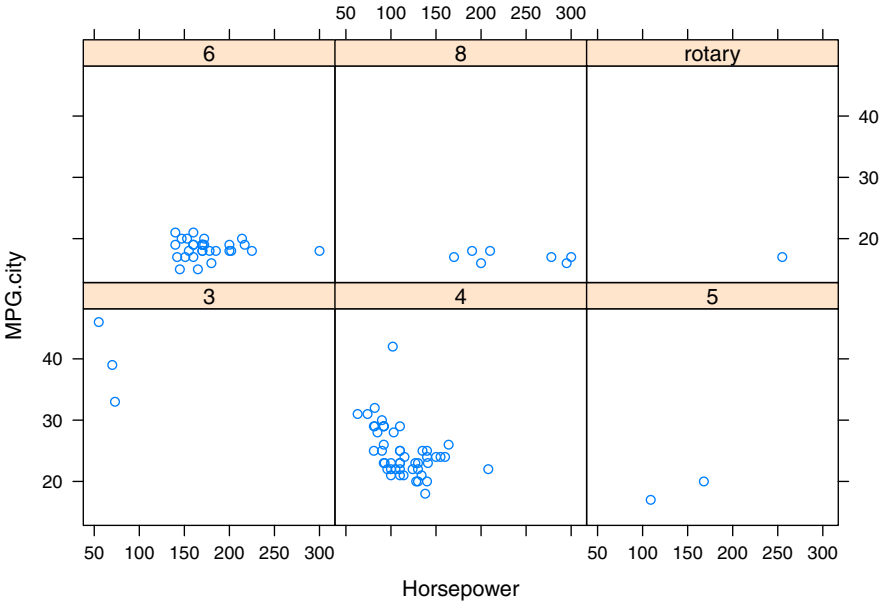


Figure 2.3: Lattice conditioning plot, showing the relationship between Horsepower and MPG.city, conditional on Cylinders, all from the Cars93 data frame.

2.2.4 The ggplot2 package

As noted, another *R* graphics system based on the `grid` package is `ggplot2` by Hadley Wickham, similar in some respects to lattice graphics, but with a fundamentally different basis and structure. Specifically, `ggplot2` is based on the *grammar of graphics*, a systematic approach to constructing graphical objects described in a book of the same title by Wilkinson [76]. Like lattice graphics, many of the default options for plots generated by `ggplot2` are based on research in human perception and are therefore preferred by many to the base graphics defaults [57, p. 21]. Also like lattice graphics, `ggplot2` provides better support for multipanel conditioning plots, and because this package is highly extensible and has become extremely popular, many extensions are available in the form of other *R* packages that provide significant additional capabilities beyond the `ggplot2` package itself. As with lattice graphics, however, one price paid for this additional flexibility is a steeper learning curve; another is the greater complexity of generating multiple plot arrays, which requires explicitly working with viewports in grid graphics.

Both because the learning curve is steeper and because some extremely useful tools like the `qqPlot` function from the `car` package, based on traditional graphics, are not available as built-in functions in `ggplot2`, this book uses base graphics instead of the more flexible `ggplot2` package. (Note that the basic

QQ-plot is available in `ggplot2` via the `qq` stat, but this option only displays the points in the plot: addition of the reference line, confidence intervals, and options for non-Gaussian distributions is all possible, but requires additional programming; in the `qqPlot` function, this is all included.) For a detailed introduction to the `ggplot2` package, good references are Hadley Wickham's book [73] and the chapter on grammar of graphics in Murrell's book [57, Ch. 5].

2.3 The plot function

Probably the most commonly used base graphics function is `plot`, which is a *generic* function, meaning that the nature of the plot it generates depends on the type of *R* object we pass to it. Most of the plots in Chapter 1 were generated using the `plot` function, usually augmented with some of the added details described in Sec. 2.4. Sec. 2.3.1 presents a collection of examples that illustrate the range of capabilities of the `plot` function, and Sec. 2.3.2 presents a brief but broader discussion of the concept of generic functions, giving some typical examples in *R* and their relationship to *S3 objects*. The essential idea is that an S3 object has certain defining characteristics, and generic functions with methods defined for a specific S3 object class can exploit those characteristics to return class-specific results. In the case of the generic `plot` function, this means that a command like "plot(x, y)" can generate a scatterplot if x and y are both numeric, a boxplot summary if x is categorical and y is numeric, or a mosaic plot if both variables are factors.

2.3.1 The flexibility of the plot function

The flexibility of the `plot` function was illustrated in the sample *R* session presented in Chapter 1, where results were shown for this function applied to a complete data frame, a numeric vector, a factor, and a pair of numeric variables: the same function returned an array of scatterplots, a plot of the numerical values in their order of appearance, a bar chart, and a scatterplot. In addition, this sample *R* session began by using the `boxplot` function to generate a boxplot summary of heating gas consumption both before and after the installation; this boxplot summary can also be generated by using the `plot` function:

```
plot(whiteside$Insul, whiteside$Gas)
```

Many of the modeling functions in *R* return an object of the type discussed in Sec. 2.3.2 (i.e., an S3 object), and special plot methods have frequently been developed for these objects. Fig. 2.4 provides an example, based on the class of *decision tree models* discussed in Chapter 10. This model predicts the average value of the heating gas consumption `Gas` in the `whiteside` data frame from the values of the other two variables, `Temp` and `Insul`. It is easily generated using the `rpart` package, with the following code:

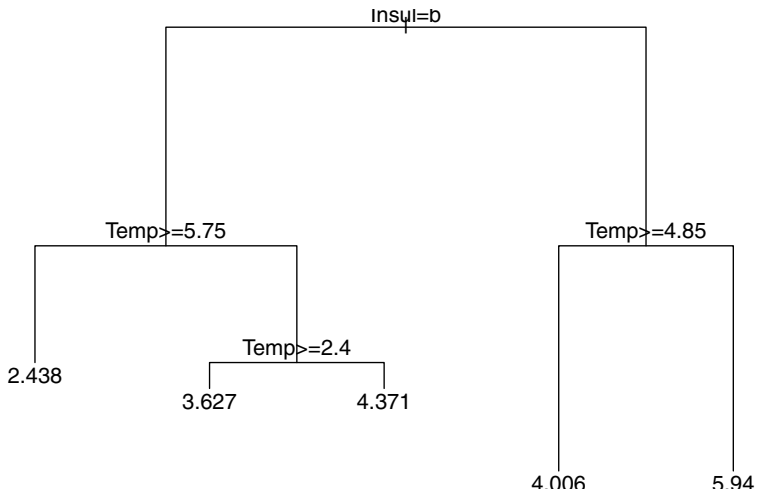


Figure 2.4: Plot of an rpart model built from the whiteside data frame.

```
library(rpart)
rpartModel <- rpart(Gas ~ ., data = whiteside)
```

For now, don't worry about the details of this model, its interpretation, or the `rpart` function used to obtain it: this example is discussed in detail in Chapter 10. The key point here is that the `rpart` function returns an S3 object of class "rpart," which the `plot` function has a method to support. Thus, we can execute the command `plot(rpartModel)` to obtain the plot shown in Fig. 2.4. Actually, the `plot` function only displays the tree structure of the model, without labels; to obtain the labels, we must also use `text`, another generic function with a method for rpart objects:

```
plot(rpartModel)
text(rpartModel)
```

The second model-based example is shown in Fig. 2.5 and it belongs to the class of *MOB models*, also discussed in Chapter 10. Like the `rpart` model just described, this model has a tree-based structure, but rather than generating a single numerical predicted value to each terminal node of the tree (i.e., each "leaf"), each terminal node contains a linear regression model that generates

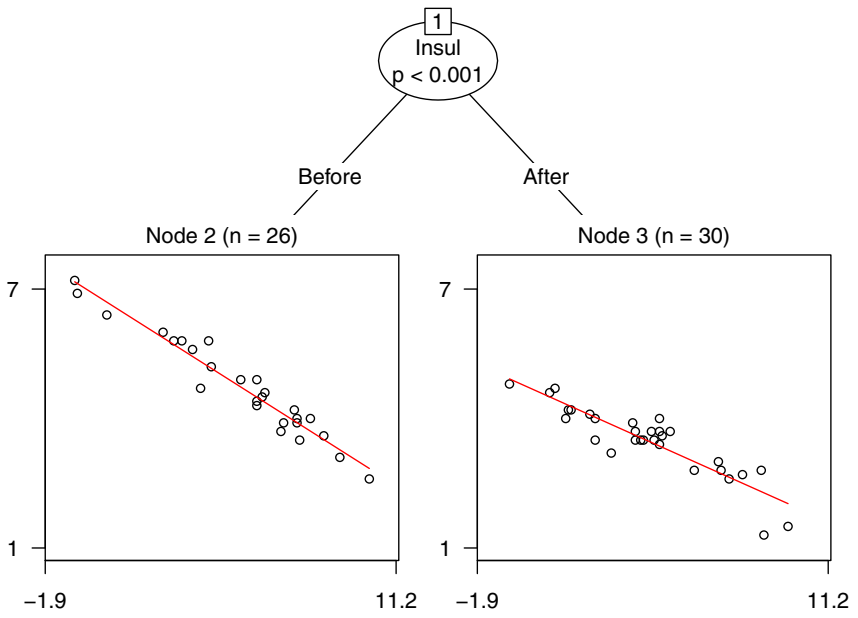


Figure 2.5: Plot of an MOB model built from the whiteside data frame.

predictions from other covariates. These models may be fit using the `lmtree` function from the `partykit` package, using very similar code to that used to generate the `rpart` model discussed above:

```
library(partykit)
MOBmodel <- lmtree(Gas ~ Temp | Insul, data = whiteside)
```

The resulting model object, `MOBmodel`, is an S3 object of class "lmtree" and when the `plot` function is applied to this object, we obtain the result shown in Fig. 2.5. In this case, the model is created using a three-part formula structure: the variable `Gas` appears to the left of the `~` symbol to indicate it is the response variable to be predicted; the variable `Temp` appears between this symbol and the `|` symbol to indicate it is the covariate used to predict the response variable in the models that appear at the terminal nodes of the tree; and the variable `Insul` that appears to the right of the `|` symbol is the partitioning variable used to build the tree. Since this partitioning variable is binary in this example, the resulting tree has two nodes, one corresponding to the "Before" data and the other corresponding to the "After" data. The structure of this model is clear from the plot: all records are assigned to one of these nodes, and a separate linear regression model that predicts `Gas` from `Temp` is built for each node. In

favorable cases—like this one—the MOB model class can be extremely effective in finding and exploiting strong heterogeneity in the underlying data, a point discussed further in Chapter 10 where this example is revisited.

The primary point of this discussion has been to illustrate the vast range of graphical results that can be generated using the generic `plot` function. The following section introduces the concepts of S3 object classes and their associated methods, the concepts that make this behavior possible. In fact, it is easy to define our own S3 object classes and construct methods for generic functions like `plot` or `summary` that make them generate specialized results for our object classes. This idea is discussed in detail in Chapter 7.

2.3.2 S3 classes and generic functions

The ability of functions like `plot` to behave very differently depending on the *type* of the *R* object we give it is a consequence of the language's *object-oriented* structure. Many programming languages are object-oriented, and in his book *Advanced R*, Hadley Wickham offers the following useful description of the minimal requirements for an object-oriented language [74, p. 99]:

> Central to any object-oriented system are the concepts of class and method. A **c**lass defines the behavior of **o**bjects by describing their attributes and their relationship to other classes. The class is also used when selecting **m**ethods, functions that behave differently depending on the class of their input. Classes are usually organized in a hierarchy: if a method does not exist for a child, then the parent's method is used instead; the child **i**nherits behavior from the parent.

Wickham further notes that *R* has three distinct object-oriented systems, based on three different object types: *S3 objects*, *S4 objects*, and *reference classes*. Since the S3 system is both the simplest of these three and the one we encounter the most often, it is useful to know something about S3 classes, particularly in understanding the behavior of *generic functions* like `plot`. Also, it is easy—and sometimes extremely useful—to define our own S3 objects and their associated methods, an important idea discussed further in Chapter 7.

The key feature of the S3 system is that methods belong to functions—specifically, generic functions—which is different from the object-oriented systems encountered in languages like *Python*, *Ruby*, or *Java*, where methods belong to objects. The newest object-oriented system in *R*—reference classes—has this more traditional structure, making it very different from the S3 system considered here. The S4 system is also significantly different from the S3 system—methods still belong to generic functions, but the structure is more formal; like the `lattice` and `ggplot2` graphics systems, the S4 system has greater flexibility, but a correspondingly steeper learning curve. Consequently, only S3 objects and their associated methods are considered here.

To see how this system works, consider the decision tree model, `rpartModel`, discussed in Sec. 2.3.1, built using the `rpart` package. The `class` function shows that the result is an S3 object of class "rpart":

```
class(rpartModel)

## [1] "rpart"
```

Fig. 2.4 was generated from this S3 object with the generic functions `plot` and `text`. In each case, the behavior of the function depends on the class of the object: for example, `text` is the same generic function as that used in Sec. 2.4.2 below to add annotation to scatterplots, but the two *methods* used with this generic function are different. Specifically, if we type "text" without trailing parentheses, we are asking *R* to display the function's code, which is:

```
text

## function (x, ...)
## UseMethod("text")
## <bytecode: 0x000000001599f1f8>
## <environment: namespace:graphics>
```

This result tells us that `text` is a function, taking one required argument (`x`) and allowing an unspecified number of optional arguments, via "..." (see Chapter 7 for a detailed discussion). The second line of this result tells us that this function is generic, having different methods associated with different object types. To see a list of these methods, use the `methods` function with the name of the generic function, i.e.:

```
methods("text")

## [1] text.default  text.formula* text.rpart*
## see '?methods' for accessing help and source code
```

In the examples presented in Sec. 2.4.2, the `text.default` method is used to add text to a scatterplot, while the `text.rpart` method was used to add the text labels to the `rpart` plot. The methods not marked with the asterisk (`*`) can be displayed directly as we did with the `text` function above, while those marked with the asterisk cannot; also, note that additional information is available if we type "`?methods`" and follow the instructions given there.

It is also possible to ask what methods are available for a given S3 object class. Again, we use the `class` function, but now we specify the desired class:

```
methods(class = "rpart")

##  [1] as.party    labels      meanvar     model.frame plot
##  [6] post        predict     print       prune       residuals
## [11] summary     text
## see '?methods' for accessing help and source code
```

We can see from this result that—based on the packages we currently have loaded in our *R* session—we have 12 methods available for S3 objects of class "rpart," including both the `plot` and `text` methods used above.

Finally, it is worth noting that some generic functions have *many* associated methods, and this number depends on what packages are loaded into our *R* session (specifically, many packages define new S3 objects and methods). For example, in the *R* environment used to develop this book, the `plot` function has 133 methods, and the `summary` function has 156.

2.3.3 Optional parameters for base graphics

It was noted earlier that there are 72 optional base graphics parameters that affect many of the base graphics plot functions. These parameters are set by the `par` function, which can also be called to return a named list with the current values for these parameters. The names are:

```
names(par())
```

```
##  [1] "xlog"      "ylog"      "adj"       "ann"       "ask"
##  [6] "bg"        "bty"       "cex"       "cex.axis"  "cex.lab"
## [11] "cex.main"  "cex.sub"   "cin"       "col"       "col.axis"
## [16] "col.lab"   "col.main"  "col.sub"   "cra"       "crt"
## [21] "csi"       "cxy"       "din"       "err"       "family"
## [26] "fg"        "fig"       "fin"       "font"      "font.axis"
## [31] "font.lab"  "font.main" "font.sub"  "lab"       "las"
## [36] "lend"      "lheight"   "ljoin"     "lmitre"    "lty"
## [41] "lwd"       "mai"       "mar"       "mex"       "mfcol"
## [46] "mfg"       "mfrow"     "mgp"       "mkh"       "new"
## [51] "oma"       "omd"       "omi"       "page"      "pch"
## [56] "pin"       "plt"       "ps"        "pty"       "smo"
## [61] "srt"       "tck"       "tcl"       "usr"       "xaxp"
## [66] "xaxs"      "xaxt"      "xpd"       "yaxp"      "yaxs"
## [71] "yaxt"      "ylbias"
```

Detailed descriptions of these parameters are available via the `help(par)` command, which notes that some of them are *read-only*, meaning that their values are fixed and cannot be modified (an example is `cin`, the default character size in inches). The following discussion does not attempt to discuss all of these parameters, only a few that are particularly useful. It is also worth noting that some of these parameters can be set in calls to certain base graphics functions (e.g., `plot`), while others can only be set through a call to the `par` function.

One of the most useful of the graphics parameters set by `par` is `mfrow`, a two-dimensional vector that sets up an array of plots; discussion of this parameter is deferred to Sec. 2.6 where the generation of plot arrays is covered in detail.

Several of these parameters come in closely related groups. One is the "cex-family" that specifies the extent to which text and symbols should be magnified relative to their default size. These parameters include:

- `cex` specifies the values for text and plotting symbols in the next plot generated, serving as a base for all of the other parameters in this group;
- `cex.axis` specifies the scaling of the axis annotations, relative to `cex`;
- `cex.lab` specifies the scaling of the axis labels, relative to `cex`;