**MODULE 1:**

**INTRODUCTION TO R**

**Introduction to R:**

R is a programming language and a software system for computations and graphics. According to the R FAQ, it consists of a language plus a run-time environment with graphics, a debugger, access to certain system functions, and the ability to run programs stored in script files.” R was originally developed in 1992 by Ross Ihaka and Robert Gentleman at the University of Auckland in New Zealand. The R language is a “dialect” of the S language2 , which was developed (mainly) by John Chambers at Bell Laboratories. This software is currently maintained by the R Core Team, which consists of more than a dozen people, and includes Ihaka, Gentleman, and Chambers.

Many other people have contributed code to R since it was first released. R is open source; the source code for R is available under the GNU General Public License, meaning that users can modify, copy, and redistribute the software or derivatives, as long as the modified source code is made available. The software is regularly updated, but changes are usually not major.

**Finding and installing R**

The R Core Team maintains a network of servers that contains installation files and documentation on R, called the Comprehensive R Archive Network, or CRAN. You can access it through http: //cran.r-project.org/, or a Google search for CRAN R. R is available for Windows, Mac, and Unix–like operating systems. Installation files and instructions can be downloaded from the CRAN site by selecting one of the download links at the top. Although the graphical user interfaces (GUIs) and their menus differ across systems (if present at all), the R commands do not. R has been extended by users, and thousands of add-on packages (referred to just as “packages” by R users), which are modules of R functions and possibly data, usually related to a particular purpose, are available for free online.

**How to run R**

R operates in two modes: interactive and batch. The one typically used is interactive mode. In this mode, you type in commands, R displays results, you type in more commands, and so on. On the other hand, batch mode does not require interaction with the user. It’s useful for production jobs, such as when a program must be run periodically, say once per day, because you can automate the process.

**R Sessions and Functions**

**R Sessions:**

* An R session refers to the period of time during which R is open and active.
* When you start R, it initiates a new session, allowing you to interact with the R environment.
* During an R session, you can execute commands, load datasets, create plots, define functions, and perform various data analysis tasks.
* R sessions can be terminated by quitting R or closing the R console or IDE.

**Functions in R:**

* In R, a function is a block of code that performs a specific task and returns a result.
* Functions in R can be built-in functions provided by R packages or user-defined functions created by the user.
* Built-in functions cover a wide range of tasks, such as mathematical operations, data manipulation, statistical analysis, and plotting.
* User-defined functions allow users to create custom functions tailored to their specific needs.
* Functions in R follow a syntax where the function name is followed by parentheses containing optional arguments.
* To use a function in R, you typically call it by its name and provide the required arguments within the parentheses.

**Examples of R Functions:**

mean(): Calculates the mean of a numeric vector.

plot(): Creates a plot based on the provided data.

read.csv(): Reads data from a CSV file into a data frame.

lm(): Fits a linear regression model to the data.

print(): Prints the output to the console.

sum(): Calculates the sum of elements in a vector.

apply(): Applies a function to rows or columns of a matrix or data frame.

**Defining Custom Functions:**

* Users can define their own functions in R using the function() keyword followed by the function name and optional arguments.
* The body of the function contains the code to be executed, and the function may return a value using the return() statement.
* Custom functions allow users to encapsulate repetitive tasks, improve code readability, and promote code reusability.

**Basic Math**

The functions available in R for manipulating data are too many to be listed here. One can find all the basic mathematical functions (log, exp, log10, log2, sin, cos, tan, asin, acos, atan, abs, sqrt, . . .), special functions (gamma, digamma, beta, besselI, . . .), as well as diverse functions useful in statistics. Some of these functions are listed in the following table.

|  |  |
| --- | --- |
| Function | Description |
| sum(x) | sum of the elements of x |
| prod(x) | product of the elements of x |
| max(x) | maximum of the elements of x |
| min(x) | minimum of the elements of x |
| which.max(x) | returns the index of the greatest element of x |
| which.min(x) | returns the index of the smallest element of x |
| range(x) | id. than c(min(x), max(x)) |
| length(x) | number of elements in x |
| mean(x) | mean of the elements of x |
| median(x) | median of the elements of x |
| var(x) or cov(x) | variance of the elements of x (calculated on n − 1); if x is a matrix or a data frame, the variance-covariance matrix is calculated |
| cor(x) | correlation matrix of x if it is a matrix or a data frame (1 if x is a vector) |
| var(x, y) or cov(x, y) | covariance between x and y, or between the columns of x and those of y if they are matrices or data frames |
| cor(x, y) | linear correlation between x and y, or correlation matrix if they are matrices or data frames |

These functions return a single value (thus a vector of length one), except range which returns a vector of length two, and var, cov, and cor which may return a matrix.

**Variables**

A variable that is visible only within a function body is said to be local to that function. In oddcount(), k and n are local variables. They disappear after the function returns:

> oddcount(c(1,2,3,7,9))

[1] 4

> n

Error: object 'n' not found

It’s very important to note that the formal parameters in an R function are local variables. Suppose we make the following function call:

> z <- c(2,6,7)

> oddcount(z)

Now suppose that the code of oddcount() changes x. Then z would not change. After the call to oddcount(), z would have the same value as before. To evaluate a function call, R copies each actual argument to the corresponding local parameter variable, and changes to that variable are not visible outside the function.

Variables created outside functions are global and are available within functions as well. Here’s an example:

> f <- function(x) return(x+y)

> y <- 3

> f(5)

[1] 8

Here y is a global variable. A global variable can be written to from within a function by using R’s superassignment operator, <<-.

**Default Arguments**R also makes frequent use of default arguments. Consider a function definition like this:

> g <- function(x,y=2,z=T) { ... }

Data types

Understanding the different types of data in R can be a bit confusing. Use of the class, mode, or even typeof functions to check data types can be helpful (or, in some cases, confusing). The following discussion will attempt to accurately describe what I think you need to know46 . The simplest data structure in R is a vector, which is a simply an ordered collection of elements. All of the data objects we have worked with so far are vectors. There are several kinds of vectors, which differ only in the type of data they contain–we’ll use them to demonstrate the different types of data in R. There are four common types of data that can be held in vectors47: numeric, integer, character, and logical.

**Numeric data**

x <- 10.2

x

# [1] 10.2

Integer data

y <- 1:5

y

# [1] 1 2 3 4 5

**Character data**

name <- "Johnny Appleseed"

name

# [1] "Johnny Appleseed"

Any time character data are entered directly into a console, you must surround individual elements with quotes. Otherwise, R will look for an object.

name <- Johnny

Either single or double quotes can be used in R49. When character data are read into R from a file, quotes are not necessary50 . Logical data contain only three values: TRUE, FALSE, or NA (NA indicates a missing value–more on this later)

a <- TRUE

a

# [1] TRUE

Note that there are no quotes around logical values (quotes would make them character data). R will return logical data for any relational expression submitted to it.

4 < 2

# [1] FALSE

or

b <- 4 < 2

b

# [1] FALSE

There are several functions that can be used to identify data type. In most cases, class or mode is the best bet. The class (i.e., the value returned by class) of an R object is meant to be “the official public view” [3, p 141] and it determines how generic functions operate on an object52. This is the same “class” that is referred to above in the discussion on generic functions.

class(x)

# [1] "numeric"

class(y)

# [1] "integer"

Technically what these output mean is that the objects are: a numeric vector, an integer vector, a character vector, and a logical vector, respectively. For more complex objects, class doesn’t return info on whether data are numeric, character, etc., but describes only the structure of the object. Unfortunately, this approach can be a bit confusing. The mode and class functions return identical values for the three examples above, but this is not so for more complex objects, where mode gives a description of the way an object is stored.

mode(x)

# [1] "numeric"

mode(name)

# [1] "character"

mode(a)

# [1] "logical"

**Vectors**

In many programming languages, vector variables are considered different from scalars, which are single-number variables. Consider the following C code, for example:

int x;

int y[3];

This requests the compiler to allocate space for a single integer named

x and a three-element integer array (C terminology analogous to R’s vector

type) named y. But in R, numbers are actually considered one-element vectors, and there is really no such thing as a scalar.

R variable types are called modes. Recall from Chapter 1 that all elements in a vector must have the same mode, which can be integer, numeric

(floating-point number), character (string), logical (Boolean), complex,

and so on. If you need your program code to check the mode of a variable x,

you can query it by the call typeof(x).

Unlike vector indices in ALGOL-family languages, such as C and Python,

vector indices in R begin at 1This requests the compiler to allocate space for a single integer named x and a three-element integer array (C terminology analogous to R’s vector type) named y. But in R, numbers are actually considered one-element vectors, and there is really no such thing as a scalar. R variable types are called modes. Recall from Chapter 1 that all elements in a vector must have the same mode, which can be integer, numeric (floating-point number), character (string), logical (Boolean), complex, and so on. If you need your program code to check the mode of a variable x, you can query it by the call typeof(x). Unlike vector indices in ALGOL-family languages, such as C and Python, vector indices in R begin at 1.

**Adding and Deleting Vector Elements**

Vectors are stored like arrays in C, contiguously, and thus you cannot insert or delete elements—something you may be used to if you are a Python programmer. The size of a vector is determined at its creation, so if you wish to add or delete elements, you’ll need to reassign the vector. For example, let’s add an element to the middle of a four-element vector:

> x <- c(88,5,12,13)

> x <- c(x[1:3],168,x[4]) # insert 168 before the 13

> x

[1] 88 5 12 168 13

Here, we created a four-element vector and assigned it to x. To insert a new number 168 between the third and fourth elements, we strung together the first three elements of x, then the 168, then the fourth element of x. This creates a new five-element vector, leaving x intact for the time being. We then assigned that new vector to x. In the result, it appears as if we had actually changed the vector stored in x, but really we created a new vector and stored that vector in x. This difference may seem subtle, but it has implications. For instance, in some cases, it may restrict the potential for fast performance in R.

**Obtaining the Length of a Vector**

You can obtain the length of a vector by using the length() function:

> x <- c(1,2,4)

> length(x)

[1] 3

In this example, we already know the length of x, so there really is no need to query it. But in writing general function code, you’ll often need to know the lengths of vector arguments. For instance, suppose that we wish to have a function that determines the index of the first 1 value in the function’s vector argument (assuming we are sure there is such a value). Here is one (not necessarily efficient) way we could write the code:

first1 <- function(x) {

for (i in 1:length(x)) {

if (x[i] == 1) break # break out of loop

}

return(i)

}

Without the length() function, we would have needed to add a second argument to first1(), say naming it n, to specify the length of x. Note that in this case, writing the loop as follows won’t work:

for (n in x)

The problem with this approach is that it doesn’t allow us to retrieve the index of the desired element. Thus, we need an explicit loop, which in turn requires calculating the length of x. One more point about that loop: For careful coding, you should worry that length(x) might be 0. In such a case, look what happens to the expression 1:length(x) in our for loop:

> x <- c()

> x

NULL

> length(x)

[1] 0

> 1:length(x)

[1] 1 0

Our variable i in this loop takes on the value 1, then 0, which is certainly not what we want if the vector x is empty.

**Advanced Data Structures**

R has a variety of data structures. Here, we will sketch some of the most frequently used structures to give you an overview of R before we dive into the details. This way, you can at least get started with some meaningful examples, even if the full story behind them must wait.

**Vectors, the R Workhorse**

The vector type is really the heart of R. It’s hard to imagine R code, or even an interactive R session, that doesn’t involve vectors. The elements of a vector must all have the same mode, or data type. You can have a vector consisting of three character strings (of mode character) or three integer elements (of mode integer), but not a vector with one integer element and two character string elements.

**Scalars**

Scalars, or individual numbers, do not really exist in R. As mentioned earlier, what appear to be individual numbers are actually one-element vectors. Consider the following:

> x <- 8

> x

[1] 8

Recall that the [1] here signifies that the following row of numbers begins with element 1 of a vector—in this case, x[1]. So you can see that R was indeed treating x as a vector, albeit a vector with just one element.

**Character Strings**

Character strings are actually single-element vectors of mode character, (rather than mode numeric):

> x <- c(5,12,13)

> x

[1] 5 12 13

> length(x)

[1] 3

> mode(x)

[1] "numeric"

> y <- "abc"

> y

[1] "abc"

> length(y)

[1] 1

> mode(y)

[1] "character"

> z <- c("abc","29 88")

> length(z)

[1] 2

> mode(z)

[1] "character"

**Data Frames**

On a technical level, a data frame is a list, with the components of that list being equal-length vectors. Actually, R does allow the components to be other types of objects, including other data frames. This gives us heterogeneous–data analogs of arrays in our analogy. But this use of data frames is rare in practice, and in this book, we will assume all components of a data frame are vectors. In this chapter, we’ll present quite a few data frame examples, so you can become familiar with their variety of uses in R.

**Creating Data Frames**

To begin, let’s take another look at our simple data frame example

> kids <- c("Jack","Jill")

> ages <- c(12,10)

> d <- data.frame(kids,ages,stringsAsFactors=FALSE)

> d # matrix-like viewpoint

kids ages

1 Jack 12

2 Jill 10

The first two arguments in the call to data.frame() are clear: We wish to produce a data frame from our two vectors: kids and ages. However, that third argument, stringsAsFactors=FALSE requires more comment. If the named argument stringsAsFactors is not specified, then by default, stringsAsFactors will be TRUE. (You can also use options() to arrange the opposite default.) This means that if we create a data frame from a character vector—in this case, kids—R will convert that vector to a factor. Because our work with character data will typically be with vectors rather than factors, we’ll set stringsAsFactors to FALSE.

**Accessing Data Frames**

Now that we have a data frame, let’s explore a bit. Since d is a list, we can access it as such via component index values or component names:

> d[[1]]

[1] "Jack" "Jill"

> d$kids

[1] "Jack" "Jill"But we can treat it in a matrix-like fashion as well. For example, we can view column 1:

> d[,1]

[1] "Jack" "Jill"

This matrix-like quality is also seen when we take d apart using str():

> str(d)

'data.frame': 2 obs. of 2 variables:

$ kids: chr "Jack" "Jill"

$ ages: num 12 10

R tells us here that d consists of two observations—our two rows—that store data on two variables—our two columns.

Consider three ways to access the first column of our data frame above: d[[1]], d[,1], and d$kids. Of these, the third would generally considered to be clearer and, more importantly, safer than the first two. This better identifies the column and makes it less likely that you will reference the wrong column. But in writing general code—say writing R packages—matrix-like notation d[,1] is needed, and it is especially handy if you are extracting subdata frames.

**Extended Example: Regression Analysis of Exam Grades Continued**

Recall our course examination data set in Section 1.5. There, we didn’t have a header, but for this example we do, and the first few records in the file now are as follows:

"Exam 1" "Exam 2" Quiz

2.0 3.3 4.0

3.3 2.0 3.7

4.0 4.0 4.0

2.3 0.0 3.3

2.3 1.0 3.3

3.3 3.7 4.0

As you can see, each line contains the three test scores for one student. This is the classic two-dimensional file notion, like that alluded to in the preceding output of str(). Here, each line in our file contains the data for one observation in a statistical data set. The idea of a data frame is to encapsulate such data, along with variable names, into one object. Notice that we have separated the fields here by spaces. Other delimiters may be specified, notably commas for comma-separated value (CSV) files (as you’ll see in Section 5.2.5). The variable names specified in the first record must be separated by the same delimiter as used for the data, which is spaces in this case. If the names themselves contain embedded spaces, as we have here, they must be quoted. We read in the file as before, but in this case we state that there is a header record:

examsquiz <- read.table("exams",header=TRUE)

**Lists:**

As with vectors and matrices, one common operation with lists is indexing. List indexing is similar to vector and matrix indexing but with some major differences. And like matrices, lists have an analog for the apply() function. We’ll discuss these and other list topics, including ways to take lists apart, which often comes in handy.

**Creating Lists**

Technically, a list is a vector. Ordinary vectors—those of the type we’ve been using so far in this book—are termed atomic vectors, since their components cannot be broken down into smaller components. In contrast, lists are referred to as recursive vectors. For our first look at lists, let’s consider an employee database. For each employee, we wish to store the name, salary, and a Boolean indicating union membership. Since we have three different modes here—character, numeric, and logical—it’s a perfect place for using lists. Our entire database might then be a list of lists, or some other kind of list such as a data frame, though we won’t pursue that here. We could create a list to represent our employee, Joe, this way:

j <- list(name="Joe", salary=55000, union=T)

We could print out j, either in full or by component:

> j

$name

[1] "Joe"

$salary

[1] 55000

$union

[1] TRUE

Actually, the component names—called tags in the R literature—such as salary are optional. We could alternatively do this:

> jalt <- list("Joe", 55000, T)

> jalt

[[1]]

[1] "Joe"

[[2]]

[1] 55000

[[3]]

[1] TRUE

However, it is generally considered clearer and less error-prone to use names instead of numeric indices. Names of list components can be abbreviated to whatever extent is possible without causing ambiguity:

> j$sal

[1] 55000

Since lists are vectors, they can be created via vector():

> z <- vector(mode="list")

> z[["abc"]] <- 3

> z

$abc

[1] 3

**General List Operations**

Now that you’ve seen a simple example of creating a list, let’s look at how to access and work with lists.

**List Indexing**

You can access a list component in several different ways:

> j$salary

[1] 55000

> j[["salary"]]

[1] 55000

> j[[2]]

[1] 55000

We can refer to list components by their numerical indices, treating the list as a vector. However, note that in this case, we use double brackets instead of single ones. So, there are three ways to access an individual component c of a list lst and return it in the data type of c:

lst$c

• lst[["c"]]

• lst[[i]], where i is the index of c within lst

Each of these is useful in different contexts, as you will see in subsequent examples. But note the qualifying phrase, “return it in the data type of c.” An alternative to the second and third techniques listed is to use single brackets rather than double brackets:

lst["c"]

lst[i], where i is the index of c within lst

Both single-bracket and double-bracket indexing access list elements in vector-index fashion. But there is an important difference from ordinary (atomic) vector indexing. If single brackets [ ] are used, the result is another list—a sublist of the original. For instance, continuing the preceding example, we have this:

> j[1:2]

$name

[1] "Joe"

$salary

[1] 55000

> j2 <- j[2]

> j2

$salary

[1] 55000

> class(j2)

[1] "list"

> str(j2)

List of 1

$ salary: num 55000The subsetting operation returned another list consisting of the first two components of the original list j. Note that the word returned makes sense here, since index brackets are functions. This is similar to other cases you’ve seen for operators that do not at first appear to be functions, such as +. By contrast, you can use double brackets [[ ]] for referencing only a single component, with the result having the type of that component.

> j[[1:2]]

Error in j[[1:2]] : subscript out of bounds

> j2a <- j[[2]]

> j2a

[1] 55000

> class(j2a)

[1] "numeric"

**Adding and Deleting List Elements**

The operations of adding and deleting list elements arise in a surprising number of contexts. This is especially true for data structures in which lists form the foundation, such as data frames and R classes. New components can be added after a list is created.

> z <- list(a="abc",b=12)

> z

$a

[1] "abc"$b

[1] 12

> z$c <- "sailing" # add a c component

> # did c really get added?

> z

$a

[1] "abc"

$b

[1] 12

$c

[1] "sailing"

**Matrices and Arrays:**

Matrices are special cases of a more general R type of object: arrays. Arrays can be multidimensional. For example, a three-dimensional array would consist of rows, columns, and layers, not just rows and columns as in the matrix case. Most of this chapter will concern matrices, but we will briefly discuss higher-dimensional arrays in the final section. Much of R’s power comes from the various operations you can perform on matrices. We’ll cover these operations in this chapter, especially those analogous to vector subsetting and vectorization.

**Creating Matrices**

Matrix row and column subscripts begin with 1. For example, the upper-left corner of the matrix a is denoted a[1,1]. The internal storage of a matrix is in column-major order, meaning that first all of column 1 is stored, then all of column 2, and so on.

One way to create a matrix is by using the matrix() function:

> y <- matrix(c(1,2,3,4),nrow=2,ncol=2)

> y

[,1] [,2]

[1,] 1 3

[2,] 2 4

Here, we concatenate what we intend as the first column, the numbers 1 and 2, with what we intend as the second column, 3 and 4. So, our data is (1,2,3,4). Next, we specify the number of rows and columns. The fact that R uses column-major order then determines where these four numbers are put within the matrix. Since we specified the matrix entries in the preceding example, and there were four of them, we did not need to specify both ncol and nrow; just nrow or ncol would have been enough. Having four elements in all, in two rows, implies two columns:

> y <- matrix(c(1,2,3,4),nrow=2)

> y

[,1] [,2]

[1,] 1 3

[2,] 2 4

Note that when we then print out y, R shows us its notation for rows and columns. For instance, [,2] means the entirety of column 2, as can be seen in this check:

> y[,2]

[1] 3 4

Another way to build y is to specify elements individually:

> y <- matrix(nrow=2,ncol=2)

> y[1,1] <- 1

> y[2,1] <- 2

> y[1,2] <- 3

> y[2,2] <- 4

> y

[,1] [,2]

[1,] 1 3

[2,] 2 4

**General Matrix Operations**

Now that we’ve covered the basics of creating a matrix, we’ll look at some common operations performed with matrices. These include performing linear algebra operations, matrix indexing, and matrix filtering.

**Performing Linear Algebra Operations on Matrices**

You can perform various linear algebra operations on matrices, such as matrix multiplication, matrix scalar multiplication, and matrix addition. Using y from the preceding example, here is how to perform those three operations:

>y%\*% y # mathematical matrix multiplication

[,1] [,2]

[1,] 7 15

[2,]10 22

> 3\*y # mathematical multiplication of matrix by scalar

[,1] [,2]

[1,] 3 9

[2,] 6 12

> y+y # mathematical matrix addition

[,1] [,2]

[1,] 2 6

[2,] 4 8

**Matrix Indexing**

The same operations we discussed for vectors in Section 2.4.2 apply to matrices as well. Here’s an example:

> z

[,1] [,2] [,3]

[1,] 1 1 1

[2,] 2 1 0

[3,] 3 0 1

[4,] 4 0 0

> z[,2:3]

[,1] [,2]

[1,] 1 1

[2,] 1 0

[3,] 0 1

[4,] 0 0

Here, we requested the submatrix of z consisting of all elements with column numbers 2 and 3 and any row number. This extracts the second and third columns. Here’s an example of extracting rows instead of columns:

> y

[,1] [,2]

[1,]11 12

[2,]21 22

[3,]31 32

> y[2:3,]

[,1] [,2]

[1,]21 22

[2,]31 32

> y[2:3,2]

[1] 22 32

**Higher-Dimensional Arrays**

In a statistical context, a typical matrix in R has rows corresponding to observations, say on various people, and columns corresponding to variables, such as weight and blood pressure. The matrix is then a two-dimensional data structure. But suppose we also have data taken at different times, one data point per person per variable per time. Time then becomes the third dimension, in addition to rows and columns. In R, such data sets are called arrays. As a simple example, consider students and test scores. Say each test consists of two parts, so we record two scores for a student for each test. Now suppose that we have two tests, and to keep the example small, assume we have only three students. Here’s the data for the first test:

> firsttest

[,1] [,2]

[1,] 46 30

[2,] 21 25

[3,] 50 50

Student 1 had scores of 46 and 30 on the first test, student 2 scored 21 and 25, and so on. Here are the scores for the same students on the second test:

> secondtest

[,1] [,2]

[1,] 46 43

[2,] 41 35

[3,] 50 50

Now let’s put both tests into one data structure, which we’ll name tests. We’ll arrange it to have two “layers”—one layer per test—with three rows and two columns within each layer. We’ll store firsttest in the first layer and second test in the second. In layer 1, there will be three rows for the three students’ scores on the first test, with two columns per row for the two portions of a test. We use R’s array function to create the data structure:

> tests <- array(data=c(firsttest,secondtest),dim=c(3,2,2))

In the argument dim=c(3,2,2), we are specifying two layers (this is the second 2), each consisting of three rows and two columns. This then becomes an attribute of the data structure:

> attributes(tests)

$dim

[1] 3 2 2

Each element of tests now has three subscripts, rather than two as in the matrix case. The first subscript corresponds to the first element in the $dim vector, the second subscript corresponds to the second element in the vector, and so on. For instance, the score on the second portion of test 1 for student 3 is retrieved as follows:

> tests[3,2,1]

[1] 48

R’s print function for arrays displays the data layer by layer:

> tests

,,1

[,1] [,2]

[1,] 46 30

[2,] 21 25

[3,] 50 48

,,2

[,1] [,2]

[1,] 46 43

[2,] 41 35

[3,] 50 49

**Classes**

R is an object-oriented language. Objects are instances of classes. Classes are a bit more abstract than the data types you’ve met so far. Here, we’ll look briefly at the concept using R’s S3 classes. (The name stems from their use in the old S language, version 3, which was the inspiration for R.) Most of R is based on these classes, and they are exceedingly simple. Their instances are simply R lists but with an extra attribute: the class name. For example, we noted earlier that the (nongraphical) output of the hist() histogram function is a list with various components, such as break and count components. There was also an attribute, which specified the class of the list, namely histogram.

> print(hn)

$breaks

[1] 400 500 600 700 800 900 1000 1100 1200 1300 1400

$counts

[1] 1 0 5 20 25 19 12 11 6 1

...

...

attr(,"class")

[1] "histogram"

At this point, you might be wondering, “If S3 class objects are just lists, why do we need them?” The answer is that the classes are used by generic functions. A generic function stands for a family of functions, all serving a similar purpose but each appropriate to a specific class. A commonly used generic function is summary(). An R user who wants to use a statistical function, like hist(), but is unsure of how to deal with its output (which can be voluminous), can simply call summary() on the output, which is not just a list but an instance of an S3 class. The summary() function, in turn, is actually a family of summary-making functions, each handling objects of a particular class. When you call summary() on some output, R searches for a summary function appropriate to the class at hand and uses it to give a friendlier representation of the list. Thus, calling summary() on the output of hist() produces a summary tailored to that function, and calling summary() on the output of the lm() regression function produces a summary appropriate for that function.

The plot() function is another generic function. You can use plot() on just about any R object. R will find an appropriate plotting function based on the object’s class. Classes are used to organize objects. Together with generic functions, they allow flexible code to be developed for handling a variety of different but related tasks.

**MODULE 2**

**R PROGRAMMING STRUCTURES**

**Control Statements**

Control statements in R look very similar to those of the ALGOL-descendant family languages mentioned above. Here, we’ll look at loops and if-else statements.

**Loops**

we defined the oddcount() function. In that function, the following line should have been instantly recognized by Python programmers:

for (n in x) {

It means that there will be one iteration of the loop for each component of the vector x, with n taking on the values of those components—in the first iteration, n = x[1]; in the second iteration, n = x[2]; and so on. For example, the following code uses this structure to output the square of every element in a vector:

> x <- c(5,12,13)

> for (n in x) print(n^2)

[1] 25

[1] 144

[1] 169

C-style looping with while and repeat is also available, complete with break, a statement that causes control to leave the loop. Here is an example that uses all three:

> i <- 1

> while (i <= 10) i <- i+4

> i

[1] 13

>

> i <- 1

> while(TRUE) { # similar loop to above

+ i <- i+4

+ if (i > 10) break

+ }

> i

[1] 13

>

> i <- 1

> repeat { # again similar

+ i <- i+4

+ if (i > 10) break

+ }

> i

[1] 13

In the first code snippet, the variable i took on the values 1, 5, 9, and 13 as the loop went through its iterations. In that last case, the condition i <= 10 failed, so the break took hold and we left the loop.

This code shows three different ways of accomplishing the same thing, with break playing a key role in the second and third ways. Note that repeat has no Boolean exit condition. You must use break (or something like return()). Of course, break can be used with for loops, too. Another useful statement is next, which instructs the interpreter to skip the remainder of the current iteration of the loop and proceed directly to the next one. This provides a way to avoid using complexly nested if-then else constructs, which can make the code confusing. Let’s take a look at an example that uses next.

sim <- function(nreps) {

commdata <- list()

commdata$countabsamecomm <- 0

for (rep in 1:nreps) {

commdata$whosleft <- 1:20

commdata$numabchosen <- 0

commdata <- choosecomm(commdata,5)

if (commdata$numabchosen > 0) next

commdata <- choosecomm(commdata,4)

if (commdata$numabchosen > 0) next

commdata <- choosecomm(commdata,3)

}

print(commdata$countabsamecomm/nreps)

}

There are next statements in lines 8 and 10. Let’s see how they work and how they improve on the alternatives. The two next statements occur within the loop that starts at line 4. Thus, when the if condition holds in line 8, lines 9 through 11 will be skipped, and control will transfer to line 4. The situation in line 10 is similar.

Because this simple example has just two levels, it’s not too bad. However, nested if statements can become confusing when you have more levels. The for construct works on any vector, regardless of mode. You can loop over a vector of filenames, for instance. Say we have a file named file1 with the following contents:

1 2 3 4 5 6

**Looping Over Nonvector Sets**

R does not directly support iteration over nonvector sets, but there are a couple of indirect yet easy ways to accomplish it:

* Use lapply(), assuming that the iterations of the loop are independent of each other, thus allowing them to be performed in any order.
* Use get(). As its name implies, this function takes as an argument a character string representing the name of some object and returns the object of that name. It sounds simple, but get() is a very powerful function.

Let’s look at an example of using get(). Say we have two matrices, u and v, containing statistical data, and we wish to apply R’s linear regression function lm() to each of them.

> u

[,1] [,2]

[1,] 1 1

[2,] 2 2

[3,] 3 4

> v

[,1] [,2]

[1,] 8 15

[2,] 12 10

[3,] 20 2

> for (m in c("u","v")) {

+ z <- get(m)

+ print(lm(z[,2] ~ z[,1]))

+ }

Call:

lm(formula = z[, 2] ~ z[, 1])

Coefficients:

(Intercept) z[, 1]

-0.6667 1.5000

Call:

lm(formula = z[, 2] ~ z[, 1])

Coefficients:

(Intercept) z[, 1]

23.286 -1.071

**if-else**

The syntax for if-else looks like this:

if (r == 4) {

x <- 1

} else {

x <- 3

y <- 4

}

It looks simple, but there is an important subtlety here. The if section consists of just a single statement:

x <- 1

So, you might guess that the braces around that statement are not necessary. However, they are indeed needed. The right brace before the else is used by the R parser to deduce that this is an if-else rather than just an if. In interactive mode, without braces, the parser would mistakenly think the latter and act accordingly, which is not what we want. An if-else statement works as a function call, and as such, it returns the last value assigned.

v <- if (cond) expression1 else expression2

This will set v to the result of expression1 or expression2, depending on whether cond is true. You can use this fact to compact your code. Here’s a simple example:

> x <- 2

> y <- if(x == 2) x else x+1

> y

[1] 2

> x <- 3

> y <- if(x == 2) x else x+1

> y

[1] 4

Without taking this tack, the code

y <- if(x == 2) x else x+1

would instead consist of the somewhat more cluttered

if(x == 2) y <- x else y <- x+1

In more complex examples, expression1 and/or expression2 could be function calls. On the other hand, you probably should not let compactness take priority over clarity

**Arithmetic and Boolean Operators and Values**

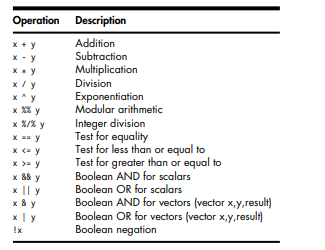


Table 7-1: Basic R Operators

Though R ostensibly has no scalar types, with scalars being treated as one-element vectors, we see the exception in Table 7-1: There are different Boolean operators for the scalar and vector cases. This may seem odd, but a simple example will demonstrate the need for such a distinction.

> x

[1] TRUE FALSE TRUE

> y

[1] TRUE TRUE FALSE

>x&y

[1] TRUE FALSE FALSE

> x[1] && y[1]

[1] TRUE

> x && y # looks at just the first elements of each vector

[1] TRUE

> if (x[1] && y[1]) print("both TRUE")

[1] "both TRUE"

> if (x & y) print("both TRUE")

[1] "both TRUE"

Warning message:

In if (x & y) print("both TRUE") :

the condition has length > 1 and only the first element will be used

**Default Values for Arguments**

> testscores <- read.table("exams",header=TRUE)

The argument header=TRUE tells R that we have a header line, so R should not count that first line in the file as data. This is an example of the use of named arguments. Here are the first few lines of the function

> read.table

function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",

row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA",

colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,

fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,

comment.char = "#", allowEscapes = FALSE, flush = FALSE,

stringsAsFactors = default.stringsAsFactors(), encoding = "unknown")

{

if (is.character(file)) {

file <- file(file, "r")

on.exit(close(file))

….

….

The second formal argument is named header. The = FALSE field means that this argument is optional, and if we don’t specify it, the default value will be FALSE. If we don’t want the default value, we must name the argument in our call:

> testscores <- read.table("exams",header=TRUE)

Hence the terminology named argument. Note, though, that because R uses lazy evaluation—it does not evaluate an expression until/unless it needs to—the named argument may not actually be used.

**Return Values**

The return value of a function can be any R object. Although the return value is often a list, it could even be another function. You can transmit a value back to the caller by explicitly calling return(). Without this call, the value of the last executed statement will be returned by default.

> oddcount

function(x) {

k <- 0 # assign 0 to k

for (n in x) {

if (n %% 2 == 1) k <- k+1 # %% is the modulo operator

}

return(k)

}

This function returns the count of odd numbers in the argument. We could slightly simplify the code by eliminating the call to return(). To do this, we evaluate the expression to be returned, k, as our last statement in the code:

oddcount <- function(x) {

k <- 0

pagebreak

for (n in x) {

if (n %% 2 == 1) k <- k+1

}

k

}

**Deciding Whether to Explicitly Call return()**

The prevailing R idiom is to avoid explicit calls to return(). One of the reasons cited for this approach is that calling that function lengthens execution time. However, unless the function is very short, the time saved is negligible, so this might not be the most compelling reason to refrain from using return(). But it usually isn’t needed nonetheless. Consider our second example from the preceding section:

oddcount <- function(x) {

k <- 0

for (n in x) {

if (n %% 2 == 1) k <- k+1

}

k

}

Here, we simply ended with a statement listing the expression to be returned—in this case, k. A call to return() wasn’t necessary. Code in this book usually does include a call to return(), for clarity for beginners, but it is customary to omit it. Good software design, however, should be mean that you can glance through a function’s code and immediately spot the various points at which control is returned to the caller. The easiest way to accomplish this is to use an explicit return() call in all lines in the middle of the code that cause a return. (You can still omit a return() call at the end of the function if you wish.)

**Returning Complex Objects**

Since the return value can be any R object, you can return complex objects. Here is an example of a function being returned:

> g

function() {

t <- function(x) return(x^2)

return(t)

}

> g()

function(x) return(x^2)

<environment: 0x8aafbc0>

**Functions Are Objects**

R functions are first-class objects (of the class "function", of course), meaning that they can be used for the most part just like other objects. This is seen in the syntax of function creation:

> g <- function(x) {

+ return(x+1)

+ }

Here, function() is a built-in R function whose job is to create functions! On the right-hand side, there are really two arguments to function(): The first is the formal argument list for the function we’re creating—here, just x—and the second is the body of that function—here, just the single statement return(x+1). That second argument must be of class "expression". So, the point is that the right-hand side creates a function object, which is then assigned to g. By the way, even the "{" is a function, as you can verify by typing this:

> ?"{"

Its job is the make a single unit of what could be several statements. These two arguments to function() can later be accessed via the R functions formals() and body(), as follows:

> formals(g)

$x

> body(g)

{

return(x + 1)

}

Recall that when using R in interactive mode, simply typing the name of an object results in printing that object to the screen. Functions are no exception, since they are objects just like anything else.

> g

function(x) {

return(x+1)

}

This is handy if you’re using a function that you wrote but which you’ve forgotten the details of. Printing out a function is also useful if you are not quite sure what an R library function does. By looking at the code, you may understand it better. For example, if you are not sure as to the exact behavior of the graphics function abline(), you could browse through its code to better understand how to use it.

> abline

function (a = NULL, b = NULL, h = NULL, v = NULL, reg = NULL,

coef = NULL, untf = FALSE, ...)

{

int\_abline <- function(a, b, h, v, untf, col = par("col"),

lty = par("lty"), lwd = par("lwd"), ...) .Internal(abline(a,

b, h, v, untf, col, lty, lwd, ...))

if (!is.null(reg)) {

if (!is.null(a))

warning("'a' is overridden by 'reg'")

a <- reg

}

if (is.object(a) || is.list(a)) {

p <- length(coefa <- as.vector(coef(a)))

...

...

If you wish to view a lengthy function in this way, run it through page():

> page(abline)

An alternative is to edit it using the edit() function.

Note, though, that some of R’s most fundamental built-in functions are written directly in C, and thus they are not viewable in this manner. Here’s an example:

> sum

function (..., na.rm = FALSE) .Primitive("sum")

Since functions are objects, you can also assign them, use them as arguments to other functions, and so on.

> f1 <- function(a,b) return(a+b)

> f2 <- function(a,b) return(a-b)

> f <- f1

> f(3,2)

[1] 5

> f <- f2

> f(3,2)

[1] 1

> g <- function(h,a,b) h(a,b)

> g(f1,3,2)

[1] 5

> g(f2,3,2)

[1] 1

And since functions are objects, you can loop through a list consisting of several functions. This would be useful, for instance, if you wished to write a loop to plot a number of functions on the same graph, as follows:

> g1 <- function(x) return(sin(x))

> g2 <- function(x) return(sqrt(x^2+1))

> g3 <- function(x) return(2\*x-1)

> plot(c(0,1),c(-1,1.5)) # prepare the graph, specifying X and Y ranges

> for (f in c(g1,g2,g3)) plot(f,0,1,add=T) # add plot to existing graph

The functions formals() and body() can even be used as replacement functions. We’ll discuss replacement functions in Section 7.10, but for now, consider how you could change the body of a function by assignment:

> g <- function(h,a,b) h(a,b)

> body(g) <- quote(2\*x + 3)

> g

function (x)

2 \* x+3

> g(3)

[1] 9

The reason quote() was needed is that technically, the body of a function has the class "call", which is the class produced by quote(). Without the call to quote(), R would try to evaluate the quantity 2\*x+3. So if x had been defined and equal to 3, for example, we would assign 9 to the body of g(), certainly not what we want. By the way, since \* and + are functions, as a language object, 2\*x+3 is indeed a call—in fact, it is one function call nested within another.

**No Pointers in R**

R does not have variables corresponding to pointers or references like those of, say, the C language. This can make programming more difficult in some cases. (As of this writing, the current version of R has an experimental feature called reference classes, which may reduce the difficulty.) For example, you cannot write a function that directly changes its arguments. In Python, for instance, you can do this:

>>> x = [13,5,12]

>>> x.sort()

>>> x

[5, 12, 13]Here, the value of x, the argument to sort(), changed. By contrast, here’s how it works in R:

> x <- c(13,5,12)

> sort(x)

[1] 5 12 13

> x

[1] 13 5 12

The argument to sort() does not change. If we do want x to change in this R code, the solution is to reassign the arguments:

> x <- sort(x)

> x

[1] 5 12 13

What if our function has several variables of output? A solution is to gather them together into a list, call the function with this list as an argument, have the function return the list, and then reassign to the original list. An example is the following function, which determines the indices of odd and even numbers in a vector of integers:

> oddsevens

function(v){

odds <- which(v %% 2 == 1)

evens <- which(v %% 2 == 1)

list(o=odds,e=evens)

}

In general, our function f() changes variables x and y. We might store them in a list lxy, which would then be our argument to f(). The code, both called and calling, might have a pattern like this:

f <- function(lxxyy) {

...

lxxyy$x <- ...

lxxyy$y <- ...

return(lxxyy)

}

# set x and y

lxy$x <- ...

lxy$y <- ...

lxy <- f(lxy)

# use new x and y

... <- lxy$x

... <- lxy$y

However, this may become unwieldy if your function will change many variables. It can be especially awkward if your variables, say x and y in the example, are themselves lists, resulting in a return value consisting of lists within a list. This can still be handled, but it makes the code more syntactically complex and harder to read. Alternatives include the use of global variables, and the new R reference classes mentioned earlier. Another class of applications in which lack of pointers causes difficulties is that of treelike data structures. C code normally makes heavy use of pointers for these kinds of structures. One solution for R is to revert to what was done in the “good old days” before C, when programmers formed their own “pointers” as vector indices.

**Recursion**

Once a mathematics PhD student whom I knew to be quite bright, but who had little programming background, sought my advice on how to write a certain function. I quickly said, “You don’t even need to tell me what the function is supposed to do. The answer is to use recursion.” Startled, he asked what recursion is. I advised him to read about the famous Towers of Hanoi problem. Sure enough, he returned the next day, reporting that he was able to solve his problem in just a few lines of code, using recursion. Obviously, recursion can be a powerful tool. Well then, what is it? A recursive function calls itself. If you have not encountered this concept before, it may sound odd, but the idea is actually simple. In rough terms, the idea is this:

To solve a problem of type X by writing a recursive function f():

1. Break the original problem of type X into one or more smaller problems of type X.
2. Within f(), call f() on each of the smaller problems.
3. Within f(), piece together the results of (b) to solve the original problem.

**A Quicksort Implementation**

A classic example is Quicksort, an algorithm used to sort a vector of numbers from smallest to largest. For instance, suppose we wish to sort the vector (5,4,12,13,3,8,88). We first compare everything to the first element, 5, to form two subvectors: one consisting of the elements less than 5 and the other consisting of the elements greater than or equal to 5. That gives us subvectors (4,3) and (12,13,8,88). We then call the function on the subvectors, returning (3,4) and (8,12,13,88). We string those together with the 5, yielding (3,4,5,8,12,13,88), as desired. R’s vector-filtering capability and its c() function make implementation of Quicksort quite easy.

qs <- function(x) {

if (length(x) <= 1) return(x)

pivot <- x[1]

therest <- x[-1]

sv1 <- therest[therest < pivot]

sv2 <- therest[therest >= pivot]

sv1 <- qs(sv1)

sv2 <- qs(sv2)

return(c(sv1,pivot,sv2))

}

Note carefully the termination condition:

if (length(x) <= 1) return(x)

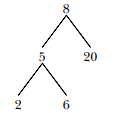
Without this, the function would keep calling itself repeatedly on empty vectors, executing forever. (Actually, the R interpreter would eventually refuse to go any further, but you get the idea.) Sounds like magic? Recursion certainly is an elegant way to solve many problems. But recursion has two potential drawbacks:

* It’s fairly abstract. I knew that the graduate student, as a fine mathematician, would take to recursion like a fish to water, because recursion is really just the inverse of proof by mathematical induction. But many programmers find it tough.
* Recursion is very lavish in its use of memory, which may be an issue in R if applied to large problems.

**Extended Example: A Binary Search Tree**

Treelike data structures are common in both computer science and statistics. In R, for example, the rpart library for a recursive partioning approach to regression and classification is very popular. Trees obviously have applications in genealogy, and more generally, graphs form the basis of analysis of social networks. However, there are real issues with tree structures in R, many of them related to the fact that R does not have pointer-style references, as discussed in Section 7.7. Indeed, for this reason and for performance purposes, a better option is often to write the core code in C with an R wrapper, as we’ll discuss in Chapter 15. Yet trees can be implemented in R itself, and if performance is not an issue, using this approach may be more convenient. For the sake of simplicity, our example here will be a binary search tree, a classic computer science data structure that has the following property: In each node of the tree, the value at the left link, if any, is less than or equal to that of the parent, while the value at the right link, if any, is greater than that of the parent.

Here is an example:



We’ve stored 8 in the root—that is, the head—of the tree. Its two child nodes contain 5 and 20, and the former itself has two child nodes, which store 2 and 6.

Note that the nature of binary search trees implies that at any node, all of the elements in the node’s left subtree are less than or equal to the value stored in this node, while the right subtree stores the elements that are larger than the value in this mode. In our example tree, where the root node contains 8, all of the values in the left subtree—5, 2 and 6—are less than 8, while 20 is greater than 8. If implemented in C, a tree node would be represented by a C struct, similar to an R list, whose contents are the stored value, a pointer to the left child, and a pointer to the right child. But since R lacks pointer variables, what can we do? Our solution is to go back to the basics. In the old prepointer days in FORTRAN, linked data structures were implemented in long arrays. A pointer, which in C is a memory address, was an array index instead. Specifically, we’ll represent each node by a row in a three-column matrix. The node’s stored value will be in the third element of that row, while the first and second elements will be the left and right links. For instance, if the first element in a row is 29, it means that this node’s left link points to the node stored in row 29 of the matrix. Remember that allocating space for a matrix in R is a time-consuming activity. In an effort to amortize the memory-allocation time, we allocate new space for a tree’s matrix several rows at a time, instead of row by row. The number of rows allocated each time will be given in the variable inc. As is common with tree traversal, we implement our algorithm with recursion.

Before discussing the code, let’s run through a quick session of tree building using its routines.

> x <- newtree(8,3)

> x

$mat

[,1] [,2] [,3]

[1,] NA NA 8

[2,] NA NA NA

[3,] NA NA NA

$nxt

[1] 2

$inc

[1] 3

> x <- ins(1,x,5)

> x

$mat

[,1] [,2] [,3]

[1,] 2 NA 8

[2,] NA NA 5

[3,] NA NA NA

$nxt

[1] 3

$inc

[1] 3

> x <- ins(1,x,6)

> x

$mat

[,1] [,2] [,3]

[1,] 2 NA 8

[2,] NA 3 5

[3,] NA NA 6

$nxt

[1] 4

$inc

[1] 3

> x <- ins(1,x,2)

> x

$mat

[,1] [,2] [,3]

[1,] 2 NA 8

[2,] 4 3 5

[3,] NA NA 6

[4,] NA NA 2

[5,] NA NA NA

[6,] NA NA NA

$nxt

[1] 5

$inc

[1] 3

> x <- ins(1,x,20)

> x

$mat

[,1] [,2] [,3]

[1,] 2 5 8

[2,] 4 3 5

[3,] NA NA 6

[4,] NA NA 2

[5,] NA NA 20

[6,] NA NA NA

$nxt

[1] 6

$inc

[1] 3

What happened here? First, the command containing our call newtree(8,3) creates a new tree, assigned to x, storing the number 8. The argument 3 specifies that we allocate storage room three rows at a time. The result is that the matrix component of the list x is now as follows:

[1,] NA NA 8

[2,] NA NA NA

[3,] NA NA NA

Three rows of storage are indeed allocated, and our data now consists just of the number 8. The two NA values in that first row indicate that this node of the tree currently has no children. We then make the call ins(1,x,5) to insert a second value, 5, into the tree x. The argument 1 specifies the root. In other words, the call says, “Insert 5 in the subtree of x whose root is in row 1.” Note that we need to reassign the return value of this call back to x. Again, this is due to the lack of pointer variables in R. The matrix now looks like this

[,1] [,2] [,3]

[1,] 2 NA 8

[2,] NA NA 5

[3,] NA NA NA

The element 2 means that the left link out of the node containing 8 is meant to point to row 2, where our new element 5 is stored. The session continues in this manner. Note that when our initial allotment of three rows is full, ins() allocates three new rows, for a total of six. In the end, the matrix is as follows:

**MODULE 3**

**DOING MATH AND SIMULATION IN R**

**Math Function**

R includes an extensive set of built-in math functions. Here is a partial list:

* exp(): Exponential function, base e
* log(): Natural logarithm
* log10(): Logarithm base 10
* sqrt(): Square root
* abs(): Absolute value
* sin(), cos(), and so on: Trig functions
* min() and max(): Minimum value and maximum value within a vector
* which.min() and which.max(): Index of the minimal element and maximal element of a vector
* pmin() and pmax(): Element-wise minima and maxima of several vectors
* sum() and prod(): Sum and product of the elements of a vector
* cumsum() and cumprod(): Cumulative sum and product of the elements of a vector
* round(), floor(), and ceiling(): Round to the closest integer, to the closest integer below, and to the closest integer above
* factorial(): Factorial function

**Extended Example: Calculating a Probability**

As our first example, we’ll work through calculating a probability using the prod() function. Suppose we have n independent events, and the i th event has the probability pi of occurring. What is the probability of exactly one of these events occurring? Suppose first that n = 3 and our events are named A, B, and C. Then we break down the computation as follows:

P(exactly one event occurs) =

P(A and not B and not C) +

P(not A and B and not C) +

P(not A and not B and C)

P(A and not B and not C) would be pA(1 − pB)(1 − pC ), and so on. For general n, that is calculated as follows:



(The i th term inside the sum is the probability that event i occurs and all the others do not occur.) Here’s code to compute this, with our probabilities pi contained in the vector p:

exactlyone <- function(p) {

notp <-1-p

tot <- 0.0

for (i in 1:length(p))

tot <- tot + p[i] \* prod(notp[-i])

return(tot)

}

How does it work? Well, the assignment

notp <-1-p

creates a vector of all the “not occur” probabilities 1 − pj , using recycling. The expression notp[-i] computes the product of all the elements of notp, except the i th—exactly what we need.

**Cumulative Sums and Products**

As mentioned, the functions cumsum() and cumprod() return cumulative sums and products.

> x <- c(12,5,13)

> cumsum(x)

[1] 12 17 30

> cumprod(x)

[1] 12 60 780

In x, the sum of the first element is 12, the sum of the first two elements is 17, and the sum of the first three elements is 30. The function cumprod() works the same way as cumsum(), but with the product instead of the sum.

**Minima and Maxima**

There is quite a difference between min() and pmin(). The former simply combines all its arguments into one long vector and returns the minimum value in that vector. In contrast, if pmin() is applied to two or more vectors, it returns a vector of the pair-wise minima, hence the name pmin.

Here’s an example:

> z

[,1] [,2]

[1,] 1 2

[2,] 5 3

[3,] 6 2

> min(z[,1],z[,2])

[1] 1

> pmin(z[,1],z[,2])

[1] 1 3 2

In the first case, min() computed the smallest value in (1,5,6,2,3,2). But the call to pmin() computed the smaller of 1 and 2, yielding 1; then the smaller of 5 and 3, which is 3; then finally the minimum of 6 and 2, giving 2. Thus, the call returned the vector (1,3,2).

You can use more than two arguments in pmin(), like this:

> pmin(z[1,],z[2,],z[3,])

[1] 1 2

The 1 in the output is the minimum of 1, 5, and 6, with a similar computation leading to the 2. The max() and pmax() functions act analogously to min() and pmin(). Function minimization / maximization can be done via nlm() and optim(). For example, let’s find the smallest value of f(x) = x2 − sin(x).

> nlm(function(x) return(x^2-sin(x)),8)

$minimum

[1] -0.2324656

$estimate

[1] 0.4501831

$gradient

[1] 4.024558e-09

$code

[1] 1

$iterations

[1] 5

Here, the minimum value was found to be approximately −0.23, occurring at x = 0.45. A Newton-Raphson method (a technique from numerical analysis for approximating roots) is used, running five iterations in this case. The second argument specifies the initial guess, which we set to be 8. (This second argument was picked pretty arbitrarily here, but in some problems, you may need to experiment to find a value that will lead to convergence.)

**Calculus**

R also has some calculus capabilities, including symbolic differentiation and numerical integration, as you can see in the following example.

> D(expression(exp(x^2)),"x") # derivative

exp(x^2) \* (2 \* x)

> integrate(function(x) x^2,0,1)

0.3333333 with absolute error < 3.7e-15

Here, R reported



And



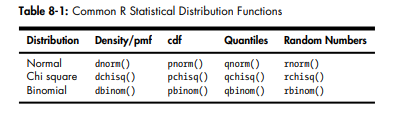
You can find R packages for differential equations (odesolve), for interfacing R with the Yacas symbolic math system (ryacas), and for other calculus operations. These packages, and thousands of others, are available from the Comprehensive R Archive Network (CRAN);

Functions for Statistical Distributions

R has functions available for most of the famous statistical distributions. Prefix the name as follows:

* With d for the density or probability mass function (pmf)
* With p for the cumulative distribution function (cdf)
* With q for quantiles
* With r for random number generation

The rest of the name indicates the distribution. Table 8-1 lists some common statistical distribution functions.



As an example, let’s simulate 1,000 chi-square variates with 2 degrees of freedom and find their mean.

> mean(rchisq(1000,df=2))

[1] 1.938179

The r in rchisq specifies that we wish to generate random numbers— in this case, from the chi-square distribution. As seen in this example, the first argument in the r-series functions is the number of random variates to generate.

These functions also have arguments specific to the given distribution families. In our example, we use the df argument for the chi-square family, indicating the number of degrees of freedom.

Let’s also compute the 95th percentile of the chi-square distribution with two degrees of freedom:

> qchisq(0.95,2)

[1] 5.991465

Here, we used q to indicate quantile—in this case, the 0.95 quantile, or the 95th percentile. The first argument in the d, p, and q series is actually a vector so that we can evaluate the density/pmf, cdf, or quantile function at multiple points. Let’s find both the 50th and 95th percentiles of the chi-square distribution with 2 degrees of freedom.

qchisq(c(0.5,0.95),df=2)

[1] 1.386294 5.991465

**Sorting**

Ordinary numerical sorting of a vector can be done with the sort() function, as in this example:

> x <- c(13,5,12,5)

> sort(x)

[1] 5 5 12 13

> x

[1] 13 5 12 5

Note that x itself did not change, in keeping with R’s functional language philosophy. If you want the indices of the sorted values in the original vector, use the order() function. Here’s an example:

> order(x)

[1] 2 4 3 1

This means that x[2] is the smallest value in x, x[4] is the second smallest, x[3] is the third smallest, and so on.

You can use order(), together with indexing, to sort data frames, like this:

> y

V1 V2

1 def 2

2 ab 5

3 zzzz 1

> r <- order(y$V2)

> r

[1] 3 1 2

> z <- y[r,]

> z

V1 V2

3 zzzz 1

1 def 2

2 ab 5

What happened here? We called order() on the second column of y, yielding a vector r, telling us where numbers should go if we want to sort them. The 3 in this vector tells us that x[3,2] is the smallest number in x[,2]; the 1 tells us that x[1,2] is the second smallest; and the 2 tells us that x[2,2] is the third smallest. We then use indexing to produce the frame sorted by column 2, storing it in z. You can use order() to sort according to character variables as well as numeric ones, as follows

> d

kids ages

1 Jack 12

2 Jill 10

3 Billy 13

> d[order(d$kids),]

kids ages

3 Billy 13

1 Jack 12

2 Jill 10

> d[order(d$ages),]

kids ages

2 Jill 10

1 Jack 12

3 Billy 13

A related function is rank(), which reports the rank of each element of a vector.

> x <- c(13,5,12,5)

> rank(x)

[1] 4.0 1.5 3.0 1.5

This says that 13 had rank 4 in x; that is, it is the fourth smallest. The value 5 appears twice in x, with those two being the first and second smallest, so the rank 1.5 is assigned to both. Optionally, other methods of handling ties can be specified.

**Linear Algebra Operations on Vectors and Matrices**

Multiplying a vector by a scalar works directly, as you saw earlier. Here’s another example:

> y

[1] 1 3 4 10

> 2\*y

[1] 2 6 8 20

If you wish to compute the inner product (or dot product) of two vectors, use crossprod(), like this:

> crossprod(1:3,c(5,12,13))

[,1]

[1,] 68

The function computed 1 · 5+2 · 12 + 3 · 13 = 68. Note that the name crossprod() is a misnomer, as the function does not compute the vector cross product. We’ll develop a function to compute real cross products in Section 8.4.1. For matrix multiplication in the mathematical sense, the operator to use is %\*%, not \*. For instance, here we compute the matrix product:



Here’s the code:

> a

[,1] [,2]

[1,] 1 2

[2,] 3 4

> b

[,1] [,2]

[1,] 1 -1

[2,] 0 1

>a%\*% b

[,1] [,2]

[1,] 1 1

[2,] 3 1The function solve() will solve systems of linear equations and even find matrix inverses. For example, let’s solve this system:

x1 + x2 = 2

−x1 + x2 = 4

Its matrix form is as follows:



Here’s the code:

> a <- matrix(c(1,1,-1,1),nrow=2,ncol=2)

> b <- c(2,4)

> solve(a,b)

[1] 3 1

> solve(a)

[,1] [,2]

[1,] 0.5 0.5

[2,] -0.5 0.5

In that second call to solve(), the lack of a second argument signifies that we simply wish to compute the inverse of the matrix. Here are a few other linear algebra functions:

* t(): Matrix transpose
* qr(): QR decomposition
* chol(): Cholesky decomposition
* det(): Determinant
* eigen(): Eigenvalues/eigenvectors
* diag(): Extracts the diagonal of a square matrix (useful for obtaining variances from a covariance matrix and for constructing a diagonal matrix).
* sweep(): Numerical analysis sweep operations

> m

[,1] [,2]

[1,] 1 2

[2,] 7 8

> dm <- diag(m)

> dm

[1] 1 8

> diag(dm)

[,1] [,2]

[1,] 1 0

[2,] 0 8

> diag(3)

[,1] [,2] [,3]

[1,] 1 0 0

[2,] 0 1 0

[3,] 0 0 1

**Extended Example: Vector Cross Product**

Let’s consider the issue of vector cross products. The definition is very simple: The cross product of vectors (x1, x2, x3) and (y1, y2, y3) in threedimensional space is a new three-dimensional vector, as shown in Equation 8.1.



This can be expressed compactly as the expansion along the top row of the determinant, as shown in Equation 8.2.



Here, the elements in the top row are merely placeholders. Don’t worry about this bit of pseudomath. The point is that the cross product vector can be computed as a sum of subdeterminants. For instance, the first component in Equation 8.1, x2y3 − x3y2, is easily seen to be the determinant of the submatrix obtained by deleting the first row and first column in Equation 8.2, as shown in Equation 8.3.



Our need to calculate subdeterminants—that is determinants of submatrices—fits perfectly with R, which excels at specifying submatrices. This suggests calling det() on the proper submatrices, as follows:

xprod <- function(x,y) {

m <- rbind(rep(NA,3),x,y)

xp <- vector(length=3)

for (i in 1:3)

xp[i] <- -(-1)^i \* det(m[2:3,-i])

return(xp)

}

Note that even R’s ability to specify values as NA came into play here to deal with the “placeholders” mentioned above. All this may seem like overkill. After all, it wouldn’t have been hard to code Equation 8.1 directly, without resorting to use of submatrices and determinants. But while that may be true in the three-dimensional case, the approach shown here is quite fruitful in the n-ary case, in n-dimensional space. The cross product there is defined as an n-by-n determinant of the form shown in Equation 8.1, and thus the preceding code generalizes perfectly.

**Extended Example: Finding Stationary Distributions of Markov Chains**

A Markov chain is a random process in which we move among various states, in a “memoryless” fashion, whose definition need not concern us here. The state could be the number of jobs in a queue, the number of items stored in inventory, and so on. We will assume the number of states to be finite. As a simple example, consider a game in which we toss a coin repeatedly and win a dollar whenever we accumulate three consecutive heads. Our state at any time i will the number of consecutive heads we have so far, so our state can be 0, 1, or 2. (When we get three heads in a row, our state reverts to 0.)

The central interest in Markov modeling is usually the long-run state distribution, meaning the long-run proportions of the time we are in each state. In our coin-toss game, we can use the code we’ll develop here to calculate that distribution, which turns out to have us at states 0, 1, and 2 in proportions 57.1%, 28.6%, and 14.3% of the time. Note that we win our dollar if we are in state 2 and toss a head, so 0.143 × 0.5 = 0.071 of our tosses will result in wins.

sult in wins. Since R vector and matrix indices start at 1 rather than 0, it will be convenient to relabel our states here as 1, 2, and 3 rather than 0, 1, and 2. For example, state 3 now means that we currently have two consecutive heads. Let pij denote the transition probability of moving from state i to state j during a time step. In the game example, for instance, p23 = 0.5, reflecting the fact that with probability 1/2, we will toss a head and thus move from having one consecutive head to two. On the other hand, if we toss a tail while we are in state 2, we go to state 1, meaning 0 consecutive heads; thus p21 = 0.5. We are interested in calculating the vector π = (π1, ..., πs), where πi is the long-run proportion of time spent at state i, over all states i. Let P denote the transition probability matrix whose i th row, j th column element is pij . Then it can be shown that π must satisfy Equation 8.4,



which is equivalent to Equation 8.5:



Here I is the identity matrix and P T denotes the transpose of P. Any single one of the equations in the system of Equation 8.5 is redundant. We thus eliminate one of them, by removing the last row of I−P in Equation 8.5. That also means removing the last 0 in the 0 vector on the right-hand side of Equation 8.5. But note that there is also the constraint shown in Equation 8.6.



In matrix terms, this is as follows:



where 1n is a vector of n 1s. So, in the modified version of Equation 8.5, we replace the removed row with a row of all 1s and, on the right-hand side, replace the removed 0 with a 1. We can then solve the system. All this can be computed with R’s solve() function, as follows:

findpi1 <- function(p) {

n <- nrow(p)

imp <- diag(n) - t(p)

imp[n,] <- rep(1,n)

rhs <- c(rep(0,n-1),1)

pivec <- solve(imp,rhs)

return(pivec)

}

Here are the main steps:

1. Calculate I − P T in line 3. Note again that diag(), when called with a scalar argument, returns the identity matrix of the size given by that argument.
2. Replace the last row of P with 1 values in line 4.
3. Set up the right-hand side vector in line 5.
4. Solve for π in line 6.

Another approach, using more advanced knowledge, is based on eigenvalues. Note from Equation 8.4 that π is a left eigenvector of P with eigenvalue 1. This suggests using R’s eigen() function, selecting the eigenvector corresponding to that eigenvalue. (A result from mathematics, the PerronFrobenius theorem, can be used to carefully justify this.) Since π is a left eigenvector, the argument in the call to eigen() must be P transpose rather than P. In addition, since an eigenvector is unique only up to scalar multiplication, we must deal with two issues regarding the eigenvector returned to us by eigen():

* It may have negative components. If so, we multiply by −1.
* It may not satisfy Equation 8.6. We remedy this by dividing by the length of the returned vector.

**Set Operations**

R includes some handy set operations, including these:

* union(x,y): Union of the sets x and y
* intersect(x,y): Intersection of the sets x and y
* setdiff(x,y): Set difference between x and y, consisting of all elements of x that are not in y
* setequal(x,y): Test for equality between x and y
* c %in% y: Membership, testing whether c is an element of the set y
* choose(n,k): Number of possible subsets of size k chosen from a set of size n

Here are some simple examples of using these functions:

> x <- c(1,2,5)

> y <- c(5,1,8,9)

> union(x,y)

[1] 1 2 5 8 9

> intersect(x,y)

[1] 1 5

> setdiff(x,y)

[1] 2

> setdiff(y,x)

[1] 8 9

> setequal(x,y)

[1] FALSE

> setequal(x,c(1,2,5))

[1] TRUE

> 2 %in% x

[1] TRUE

> 2 %in% y

[1] FALSE

> choose(5,2)

[1] 10

Recall from Section 7.12 that you can write your own binary operations. For instance, consider coding the symmetric difference between two sets— that is, all the elements belonging to exactly one of the two operand sets. Because the symmetric difference between sets x and y consists exactly of those elements in x but not y and vice versa, the code consists of easy calls to setdiff() and union(), as follows:

> symdiff

function(a,b) {

sdfxy <- setdiff(x,y)

sdfyx <- setdiff(y,x)

return(union(sdfxy,sdfyx))

}

**Input /out put**

R is not the tool you would choose for running an ATM, but it features a highly versatile array of I/O capabilities, as you will learn in this chapter. We’ll start with the basics of access to the keyboard and monitor, and then go into considerable detail on reading and writing files, including the navigation of file directories. Finally, we discuss R’s facilities for accessing the Internet.

**Accessing the Keyboard and Monitor**

R provides several functions for accesssing the keyboard and monitor. Here, we’ll look at the scan(), readline(), print(), and cat() functions.

**Using the scan() Function**

You can use scan() to read in a vector, whether numeric or character, from a file or the keyboard. With a little extra work, you can even read in data to form a list. Suppose we have files named z1.txt, z2.txt, z3.txt, and z4.txt. The z1.txt file contains the following:

123

4 5

6

The z2.txt file contents are as follows:

123

4.2 5

6

The z3.txt file contains this:

abc

de f

g

And finally, the z4.txt file has these contents:

abc

123 6

Y

Let’s see what we can do with these files using the scan() function.

> scan("z1.txt")

Read 4 items

[1] 123 4 5 6

> scan("z2.txt")

Read 4 items

[1] 123.0 4.2 5.0 6.0

> scan("z3.txt")

Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, :

scan() expected 'a real', got 'abc'

> scan("z3.txt",what="")

Read 4 items

[1] "abc" "de" "f" "g"

> scan("z4.txt",what="")

Read 4 items

[1] "abc" "123" "6" "y"

In the first call, we got a vector of four integers (though the mode is numeric). The second time, since one number was nonintegral, the others were shown as floating-point numbers, too. In the third case, we got an error. The scan() function has an optional argument named what, which specifies mode, defaulting to double mode. So, the nonnumeric contents of the file z3 produced an error. But we then tried again, with what="". This assigns a character string to what, indicating that we want character mode. (We could have set what to any character string.) The last call worked the same way. The first item was a character string, so it treated all the items that followed as strings too. Of course, in typical usage, we would assign the return value of scan() to a variable. Here’s an example:

> v <- scan("z1.txt")

By default, scan() assumes that the items of the vector are separated by whitespace, which includes blanks, carriage return/line feeds, and horizontal tabs. You can use the optional sep argument for other situations. As example, we can set sep to the newline character to read in each line as a string, as follows:

> x1 <- scan("z3.txt",what="")

Read 4 items

> x2 <- scan("z3.txt",what="",sep="\n")

Read 3 items

> x1

[1] "abc" "de" "f" "g"

> x2

[1] "abc" "de f" "g"

> x1[2]

[1] "de"

> x2[2]

[1] "de f"

In the first case, the strings "de" and "f" were assigned to separate elements of x1. But in the second case, we specified that elements of x2 were to be delineated by end-of-line characters, not spaces. Since "de" and "f" are on the same line, they are assigned together to x[2]. More sophisticated methods for reading files will be presented later in this chapter, such as methods to read in a file one line at a time. But if you want to read the entire file at once, scan() provides a quick solution.

You can use scan() to read from the keyboard by specifying an empty string for the filename:

> v <- scan("")

1: 12 5 13

4: 3 4 5

7: 8

8:

Read 7 items

> v

[1] 12 5 13 3 4 5 8

**Using the readline() Function**

If you want to read in a single line from the keyboard, readline() is very handy

> w <- readline()

abc de f

> w

[1] "abc de f"

Typically, readline() is called with its optional prompt, as follows:

> inits <- readline("type your initials: ")

type your initials: NM

> inits

[1] "NM"

**Printing to the Screen**

At the top level of interactive mode, you can print the value of a variable or expression by simply typing the variable name or expression. This won’t work if you need to print from within the body of a function. In that case, you can use the print() function, like this:

> x <- 1:3

> print(x^2)

[1] 1 4 9

Recall that print() is a generic function, so the actual function called will depend on the class of the object that is printed. If, for example, the argument is of class "table", then the print.table() function will be called.

It’s a little better to use cat() instead of print(), as the latter can print only one expression and its output is numbered, which may be a nuisance. Compare the results of the functions:

> print("abc")

[1] "abc"

> cat("abc\n")

Abc

Note that we needed to supply our own end-of-line character, "\n", in the call to cat(). Without it, our next call would continue to write to the same line. The arguments to cat() will be printed out with intervening spaces:

> x

[1] 1 2 3

> cat(x,"abc","de\n")

1 2 3 abc de

If you don’t want the spaces, set sep to the empty string "", as follows:

> cat(x,"abc","de\n",sep="")

123abcde

Any string can be used for sep. Here, we use the newline character:

> cat(x,"abc","de\n",sep="\n")

1

2

3

abc

de

You can even set sep to be a vector of strings, like this:

> x <- c(5,12,13,8,88)

> cat(x,sep=c(".",".",".","\n","\n"))

5.12.13.8

88

**Reading and Writing Files**

Now that we’ve covered the basics of I/O, let’s get to some more practical applications of reading and writing files. The following sections discuss reading data frames or matrices from files, working with text files, accessing files on remote machines, and getting file and directory information.

**Reading a Data Frame or Matrix from a File**

In Section 5.1.2, we discussed the use of the function read.table() to read in a data frame. As a quick review, suppose the file z looks like this:

name age

John 25

Mary 28

Jim 19

The first line contains an optional header, specifying column names. We could read the file this way:

> z <- read.table("z",header=TRUE)

> z

name age

1 John 25

2 Mary 28

3 Jim 19

Note that scan() would not work here, because our file has a mixture of numeric and character data (and a header). There appears to be no direct way of reading in a matrix from a file, but it can be done easily with other tools. A simple, quick way is to use scan() to read in the matrix row by row. You use the byrow option in the function matrix() to indicate that you are defining the elements of the matrix in a row-wise, rather than column-wise, manner. For instance, say the file x contains a 5-by-3 matrix, stored row-wise:

101

111

110

110

001

We can read it into a matrix this way:

> x <- matrix(scan("x"),nrow=5,byrow=TRUE)

This is fine for quick, one-time operations, but for generality, you can use read.table(), which returns a data frame, and then convert via as.matrix(). Here is a general method:

read.matrix <- function(filename) {

as.matrix(read.table(filename))

}

**Reading Text Files**

In computer literature, there is often a distinction made between text files and binary files. That distinction is somewhat misleading—every file is binary in the sense that it consists of 0s and 1s. Let’s take the term text file to mean a file that consists mainly of ASCII characters or coding for some other human language (such as GB for Chinese) and that uses newline characters to give humans the perception of lines. The latter aspect will turn out to be central here. Nontext files, such as JPEG images or executable program files, are generally called binary files. You can use readLines() to read in a text file, either one line at a time or in a single operation. For example, suppose we have a file z1 with the following contents:

John 25

Mary 28

Jim 19

We can read the file all at once, like this:

> z1 <- readLines("z1")

> z1

[1] "John 25" "Mary 28" "Jim 19"

Since each line is treated as a string, the return value here is a vector of strings—that is, a vector of character mode. There is one vector element for each line read, thus three elements here. Alternatively, we can read it in one line at a time. For this, we first need to create a connection, as described next.

**Writing to a File**

Given the statistical basis of R, file reads are probably much more common than writes. But writes are sometimes necessary, and this section will present methods for writing to files.

The function write.table() works very much like read.table(), except that it writes a data frame instead of reading one.

> kids <- c("Jack","Jill")

> ages <- c(12,10)

> d <- data.frame(kids,ages,stringsAsFactors=FALSE)

> d

kids ages

1 Jack 12

2 Jill 10

> write.table(d,"kds")

The file kds will now have these contents:

"kids" "ages"

"1" "Jack" 12

"2" "Jill" 10

In the case of writing a matrix to a file, just state that you do not want row or column names, as follows:

> write.table(xc,"xcnew",row.names=FALSE,col.names=FALSE)

The function cat() can also be used to write to a file, one part at a time. Here’s an example:

> cat("abc\n",file="u")

> cat("de\n",file="u",append=TRUE)

The first call to cat() creates the file u, consisting of one line with contents "abc". The second call appends a second line. Unlike the case of using the writeLines() function (which we’ll discuss next), the file is automatically saved after each operation. For instance, after the previous calls, the file will look like this:

abc

de

You can write multiple fields as well. So:

> cat(file="v",1,2,"xyz\n")

would produce a file v consisting of a single line:

1 2 xyz

You can also use writeLines(), the counterpart of readLines(). If you use a connection, you must specify "w" to indicate you are writing to the file, not reading from it:

> c <- file("www","w")

> writeLines(c("abc","de","f"),c)

> close(c)

The file www will be created with these contents:

abc

de

f

Note the need to proactively close the file.

**MODULE 4**

**GRAPHICS**

To begin, we’ll look at the foundational function for creating graphs: plot(). Then we’ll explore how to build a graph, from adding lines and points to attaching a legend.

**The Workhorse of R Base Graphics: The plot() Function**

The plot() function forms the foundation for much of R’s base graphing operations, serving as the vehicle for producing many different kinds of graphs.The plot() is a generic function, or a placeholder for a family of functions. The function that is actually called depends on the class of the object on which it is called. Let’s see what happens when we call plot() with an X vector and a Y vector, which are interpreted as a set of pairs in the (x,y) plane.

> plot(c(1,2,3), c(1,2,4))

This will cause a window to pop up, plotting the points (1,1), (2,2), and (3,4), as shown in Figure 12-1. As you can see, this is a very plain-Jane graph. We’ll discuss adding some of the fancy bells and whistles later in the chapter.

****

The plot() function works in stages, which means you can build up a graph in stages by issuing a series of commands. For example, as a base, we might first draw an empty graph, with only axes, like this:

> plot(c(-3,3), c(-1,5), type = "n", xlab="x", ylab="y")

This draws axes labeled x and y. The horizontal (x) axis ranges from −3 to 3. The vertical (y) axis ranges from −1 to 5. The argument type="n" means that there is nothing in the graph itself.

**Customizing Graphs**

You’ve seen how easy it is to build simple graphs in stages, starting with plot(). Now you can begin to enhance those graphs, using the many options R provides.

**Changing Character Sizes: The cex Option**

The cex (for character expand) function allows you to expand or shrink characters within a graph, which can be very useful. You can use it as a named parameter in various graphing functions. For instance, you may wish to draw the text “abc” at some point, say (2.5,4), in your graph but with a larger font, in order to call attention to this particular text. You could do this by typing the following:

text(2.5,4,"abc",cex = 1.5)

This prints the same text as in our earlier example but with characters 1.5 times the normal size.

**Changing the Range of Axes: The xlim and ylim Options**

You may wish to have the ranges on the x- and y-axes of your plot be broader or narrower than the default. This is especially useful if you will be displaying several curves in the same graph. You can adjust the axes by specifying the xlim and/or ylim parameters in your call to plot() or points(). For example, ylim=c(0,90000) specifies a range on the y-axis of 0 to 90,000. If you have several curves and do not specify xlim and/or ylim, you should draw the tallest curve first so there is room for all of them. Otherwise, R will fit the plot to the first one your draw and then cut off taller ones at the top! We took this approach earlier, when we plotted two density estimates on the same graph (Figures 12-3 and 12-4). Instead, we could have first found the highest values of the two density estimates. For d1, we find the following:

> d1

Call:

density.default(x = testscores$Exam1, from = 0, to = 100)

Data: testscores$Exam1 (39 obs.); Bandwidth 'bw' = 6.967

x y

Min. : 0 Min. :1.423e-07

1st Qu.: 25 1st Qu.:1.629e-03

Median : 50 Median :9.442e-03

Mean : 50 Mean :9.844e-03

3rd Qu.: 75 3rd Qu.:1.756e-02

Max. :100 Max. :2.156e-02

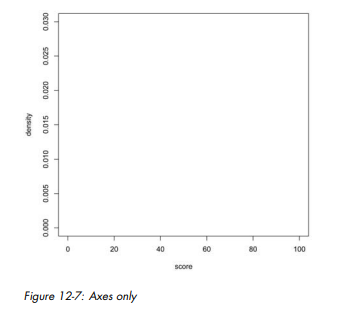
So, the largest y-value is 0.022. For d2, it was only 0.017. That means we should have plenty of room if we set ylim at 0.03. Here is how we could draw the two plots on the same picture:

> plot(c(0, 100), c(0, 0.03), type = "n", xlab="score", ylab="density")

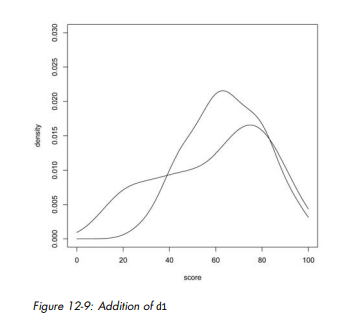
> lines(d2)

> lines(d1)

First we drew the bare-bones plot—just axes without innards, as shown in Figure 12-7. The first two arguments to plot() give xlim and ylim, so that the lower and upper limits on the Y axis will be 0 and 0.03. Calling lines() twice then fills in the graph, yielding Figures 12-8 and 12-9. (Either of the two lines() calls could come first, as we’ve left enough room.)

****

****

****

**Adding a Polygon: The polygon() Function**

You can use polygon() to draw arbitrary polygonal objects. For example, the following code draws the graph of the function f(x)=1 − e−x and then adds a rectangle that approximates the area under the curve from x = 1.2 to x = 1.4.

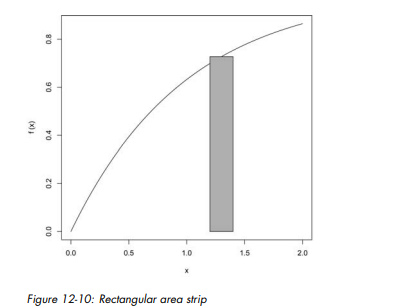
> f <- function(x) return(1-exp(-x))

> curve(f,0,2)

> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),col="gray")

The result is shown in Figure 12-10. In the call to polygon() here, the first argument is the set of x-coordinates for the rectangle, and the second argument specifies the y-coordinates. The third argument specifies that the rectangle in this case should be shaded in solid gray. As another example, we could use the density argument to fill the rectangle with striping. This call specifies 10 lines per inch:

> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),density=10)



**Smoothing Points: The lowess() and loess() Functions**

Just plotting a cloud of points, connected or not, may give you nothing but an uninformative mess. In many cases, it is better to smooth out the data by fitting a nonparametric regression estimator such as lowess(). Let’s do that for our test score data. We’ll plot the scores of exam 2 against those of exam 1:

> plot(testscores)

> lines(lowess(testscores))

The result is shown in Figure 12-11. A newer alternative to lowess() is loess(). The two functions are similar but have different defaults and other options. You need some advanced knowledge of statistics to appreciate the differences. Use whichever you find gives better smoothing.

**Graphing Explicit Functions**

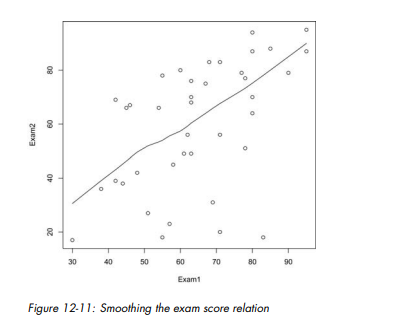
Say you want to plot the function g(t)=(t 2 + 1)0.5 for t between 0 and 5. You could use the following R code:

g <- function(t) { return (t^2+1)^0.5 } # define g()

x <- seq(0,5,length=10000) # x = [0.0004, 0.0008, 0.0012,..., 5]

y <- g(x) # y = [g(0.0004), g(0.0008), g(0.0012), ..., g(5)]

plot(x,y,type="l")



But you could avoid some work by using the curve() function, which basically uses the same method:

> curve((x^2+1)^0.5,0,5)

If you are adding this curve to an existing plot, use the add argument:

> curve((x^2+1)^0.5,0,5,add=T)

The optional argument n has the default value 101, meaning that the function will be evaluated at 101 equally spaced points in the specified range of x. Use just enough points for visual smoothness. If you find 101 is not enough, experiment with higher values of n. You can also use plot(), as follows:

> f <- function(x) return((x^2+1)^0.5)

> plot(f,0,5) # the argument must be a function name

Here, the call plot() leads to calling plot.function(), the implementation of the generic plot() function for the function class. Again, the approach is your choice; use whichever one you prefer.

**Extended Example: Magnifying a Portion of a Curve**

After you use curve() to graph a function, you may want to “zoom in” on one portion of the curve. You could do this by simply calling curve() again on the same function but with a restricted x range. But suppose you wish to display the original plot and the close-up one in the same picture. Here, we will develop a function, which we’ll name inset(), to do this. In order to avoid redoing the work that curve() did in plotting the original graph, we will modify its code slightly to save that work, via a return value. We can do this by taking advantage of the fact that you can easily inspect the code of R functions written in R (as opposed to the fundamental R functions written in C), as follows:

> curve

function (expr, from = NULL, to = NULL, n = 101, add = FALSE,

type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)

{

sexpr <- substitute(expr)

if (is.name(sexpr)) {

# ...lots of lines omitted here...

x <- if (lg != "" && "x" %in% strsplit(lg, NULL)[[1]]) {

if (any(c(from, to) <= 0))

stop("'from' and 'to' must be > 0 with log=\"x\"")

exp(seq.int(log(from), log(to), length.out = n))

}

else seq.int(from, to, length.out = n)

y <- eval(expr, envir = list(x = x), enclos = parent.frame())

if (add)

lines(x, y, type = type, ...)

else plot(x, y, type = type, ylab = ylab, xlim = xlim, log = lg, ...)

}

The code forms vectors x and y, consisting of the x- and y-coordinates of the curve to be plotted, at n equally spaced points in the range of x. Since we’ll make use of those in inset(), let’s modify this code to return x and y. Here’s the modified version, which we’ve named crv():

> crv

function (expr, from = NULL, to = NULL, n = 101, add = FALSE,

type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)

{

sexpr <- substitute(expr)

if (is.name(sexpr)) {

# ...lots of lines omitted here...

x <- if (lg != "" && "x" %in% strsplit(lg, NULL)[[1]]) {

if (any(c(from, to) <= 0))

stop("'from' and 'to' must be > 0 with log=\"x\"")

exp(seq.int(log(from), log(to), length.out = n))

}

else seq.int(from, to, length.out = n)

y <- eval(expr, envir = list(x = x), enclos = parent.frame())

if (add)

lines(x, y, type = type, ...)

else plot(x, y, type = type, ylab = ylab, xlim = xlim, log = lg, ...)

return(list(x=x,y=y)) # this is the only modification

}

**Saving Graphs to Files**

Let’s open a file:

> pdf("d12.pdf")

This opens the file d12.pdf. We now have two devices open, as we can confirm:

> dev.list()

X11 pdf

2 3

The screen is named X11 when R runs on Linux. (It’s named windows on Windows systems.) It is device number 2 here. Our PDF file is device number 3. Our active device is the PDF file:

> dev.cur()

pdf

3

All graphics output will now go to this file instead of to the screen. But what if we wish to save what’s already on the screen?

**Saving the Displayed Graph**

One way to save the graph currently displayed on the screen is to reestablish the screen as the current device and then copy it to the PDF device, which is 3 in our example, as follows:

> dev.set(2)

X11

2

> dev.copy(which=3)

pdf

3

But actually, it is best to set up a PDF device as shown earlier and then rerun whatever analyses led to the current screen. This is because the copy operation can result in distortions due to mismatches between screen devices and file devices.

**Closing an R Graphics Device**

Note that the PDF file we create is not usable until we close it, which we do as follows:

> dev.set(3)

pdf

3

> dev.off()

X11

2

You can also close the device by exiting R, if you’re finished working with it. But in future versions of R, this behavior may not exist, so it’s probably better to proactively close.

**Module 5**

**PROBABILITY DISTRIBUTIONS**

Being a statistical programming language, R easily handles all the basic necessities of statistics, including drawing random numbers and calculating distribution values (the focus of this chapter), means, variances, maxmima and minima, correlation and t-tests. Probability distributions lie at the heart of statistics, so naturally R provides numerous functions for making use of them. These include functions for generating random numbers and calculating the distribution and quantile.

**Normal Distribution**

Perhaps the most famous, and most used, statistical distribution is the normal distribution, sometimes referred to as the Gaussian distribution, which is defined as



where μ is the mean and σ the standard deviation. This is the famous bell curve that describes so many phenomena in life. To draw random numbers from the normal distribution use the rnorm function, which optionally allows the specification of the mean and standard deviation.

> # 10 draws from the standard 0-1 normal distribution

> rnorm(n=10)

[1] 0.4385627 1.1969098 1.0130680 0.0053413 -0.6086422 -1.5829601

[7] 0.9106169 -1.9663997 1.0108341 0.1931879

> # 10 draws from the 100-20 distribution

> rnorm(n=10, mean=100, sd=20)

[1] 114.99418 121.15465 95.35524 95.73121 86.45346 106.73548

[7] 104.05061 113.61679 101.40346 61.48190The density (the probability of a particular value) for the normal distribution is calculated using dnorm.

> randNorm10 <- rnorm(10)

> randNorm10

[1] 1.9125749 -0.5822831 0.5553026 -2.3583206 0.7638454 1.1312883

[7] -0.1721544 1.8832073 0.5361347 -1.2932703

> dnorm(randNorm10)

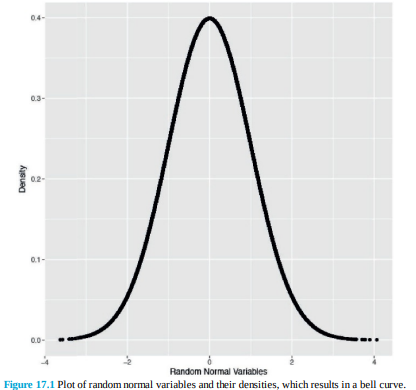
[1] 0.06406161 0.33673288 0.34194033 0.02472905 0.29799802 0.21037889

[7] 0.39307411 0.06773357 0.34553589 0.17287050

> dnorm(c(-1, 0, 1))

[1] 0.2419707 0.3989423 0.2419707

dnorm returns the probability of a specific number occuring. While it is technically mathematically impossible to find the exact probability of a number from a continuous distribution, this is an estimate of the probability. Like with rnorm, a mean and standard deviation can be specified for dnorm. To see this visually we generate a number of normal random variables, calculate their distributions and then plot them. This should result in a nicely shaped bell curve, as seen in Figure 17.1.



> # generate the normal variables

> randNorm <- rnorm(30000)

> # calcualte their distributions

> randDensity <- dnorm(randNorm)

> # load ggplot2

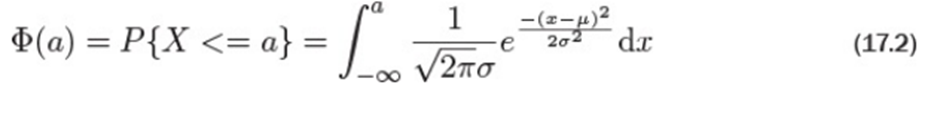
> library(ggplot2)

> # plot them

> ggplot(data.frame(x=randNorm, y=randDensity)) + aes(x=x, y=y) +

+ geom\_point() + labs(x="Random Normal Variables", y="Density")

Similarly, pnorm calculates the distribution of the normal distribution, that is, the cumulative probability that a given number, or smaller number, occurs. This is defined as



> pnorm(randNorm10)

[1] 0.972098753 0.280188016 0.710656152 0.009178915 0.777520317

[6] 0.871033114 0.431658071 0.970163858 0.704067283 0.097958799

> pnorm(c(-3, 0, 3))

[1] 0.001349898 0.500000000 0.998650102

> pnorm(-1)

[1] 0.1586553

By default this is left-tailed. To find the probability that the variable falls between two points, we must calculate the two probabilities and subtract them from each other.

> pnorm(1) - pnorm(0)

[1] 0.3413447

> pnorm(1) - pnorm(-1)

[1] 0.6826895

This probability is represented by the area under the curve and illustrated in Figure 17.2, which is drawn by the following code.



> # a few things happen with this first line of code

> # the idea is to build a ggplot2 object that we can build upon later

> # that is why it is saved to p

> # we take randNorm and randDensity and put them into a data.frame

> # we declare the x and y aes outside of any other function

> # this just gives more flexibility

> # we add lines with geom\_line()

> # x- and y-axis labels with labs(x="x", y="Density")

> p <- ggplot(data.frame(x=randNorm, y=randDensity)) + aes(x=x, y=y) +

+ geom\_line() + labs(x="x", y="Density")

>

> # plotting p will print a nice distribution

> # to create a shaded area under the curve we first calculate that area

> # generate a sequence of numbers going from the far left to -1

> neg1Seq <- seq(from=min(randNorm), to=-1, by=.1)

>

> # build a data.frame of that sequence as x

> # the distribution values for that sequence as y

> lessThanNeg1 <- data.frame(x=neg1Seq, y=dnorm(neg1Seq))

>

> head(lessThanNeg1)

x y

1 -4.164144 6.847894e-05

2 -4.064144 1.033313e-04

3 -3.964144 1.543704e-04

4 -3.864144 2.283248e-04

5 -3.764144 3.343484e-04

6 -3.664144 4.847329e-04

> # combine this with endpoints at the far left and far right

> # the height is 0

> lessThanNeg1 <- rbind(c(min(randNorm), 0),

+ lessThanNeg1,

+ c(max(lessThanNeg1$x), 0))

>

> # use that shaded region as a polygon

> p + geom\_polygon(data=lessThanNeg1, aes(x=x, y=y))

>

> # create a similar sequence going from -1 to 1

> neg1Pos1Seq <- seq(from=-1, to=1, by=.1)

>

> # build a data.frame of that sequence as x

> # the distribution values for that sequence as y

> neg1To1 <- data.frame(x=neg1Pos1Seq, y=dnorm(neg1Pos1Seq))

>

> head(neg1To1)

x y

1 -1.0 0.2419707

2 -0.9 0.2660852

3 -0.8 0.2896916

4 -0.7 0.3122539

5 -0.6 0.3332246

6 -0.5 0.3520653

> # combine this with endpoints at the far left and far right

> # the height is 0

> neg1To1 <- rbind(c(min(neg1To1$x), 0),

+ neg1To1,

+ c(max(neg1To1$x), 0))

>

> # use that shaded region as a polygon

> p + geom\_polygon(data=neg1To1, aes(x=x, y=y))

The distribution has a non-decreasing shape, as shown in Figure 17.3. The information displayed here is the same as in Figure 17.2 but it is shown differently. Instead of the cumulative probability being shown as a shaded region, it is displayed as a single point along the y-axis.



> randProb <- pnorm(randNorm)

> ggplot(data.frame(x=randNorm, y=randProb)) + aes(x=x, y=y) +

+ geom\_point() + labs(x="Random Normal Variables", y="Probability")

The opposite of pnorm is qnorm. Given a cumulative probability it returns the quantile.

> randNorm10

[1] 1.9125749 -0.5822831 0.5553026 -2.3583206 0.7638454 1.1312883

[7] -0.1721544 1.8832073 0.5361347 -1.2932703

> qnorm(pnorm(randNorm10))

[1] 1.9125749 -0.5822831 0.5553026 -2.3583206 0.7638454 1.1312883

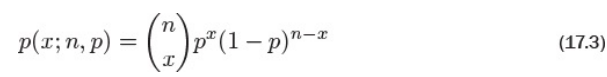
[7] -0.1721544 1.8832073 0.5361347 -1.2932703

> all.equal(randNorm10, qnorm(pnorm(randNorm10)))

[1] TRUE

**Binomial Distribution**

Like the normal distribution, the binomial distribution is well represented in R. Its probability mass function is



Where



and n is the number of trials and p is the probability of success of a trial. The mean is np and the variance is np(1 − p). When n = 1 this reduces to the Bernoulli distribution. Generating random numbers from the binomial distribution is not simply generating random numbers but rather generating the number of successes of independent trials. To simulate the number of successes out of ten trials with probability 0.4 of success, we run rbinom with n=1 (only one run of the trials), size=10 (trial size of 10) and prob=0.4 (probability of success is 0.4).

> rbinom(n=1, size=10, prob=.4)

[1] 1

That is to say that ten trials were conducted, each with 0.4 probability of success, and the number generated is the number that succeeded. As this is random, different numbers will be generated each time. By setting n to anything greater than 1, R will generate the number of successes for each of the n sets of size trials.

> rbinom(n=1, size=10, prob=.4)

[1] 3

> rbinom(n=5, size=10, prob=.4)

[1] 4 3 5 2 5

> rbinom(n=10, size=10, prob=.4)

[1] 5 2 7 4 7 3 2 3 3 3

Setting size to 1 turns the numbers into a Bernoulli random variable, which can take on only the value 1 (success) or 0 (failure).

> rbinom(n=1, size=1, prob=.4)

[1] 0

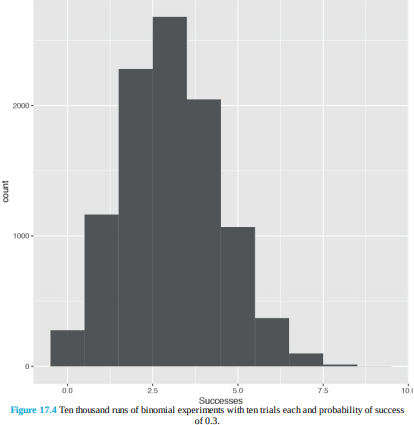
> rbinom(n=5, size=1, prob=.4)

[1] 1 1 0 0 0

> rbinom(n=10, size=1, prob=.4)

[1] 0 0 0 0 0 0 0 1 0 0

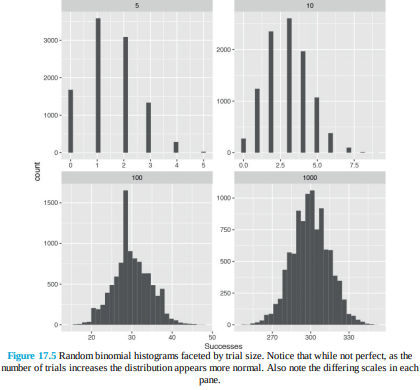
To visualize the binomial distribution, we randomly generate 10,000 experiments, each with 10 trials and 0.3 probability of success. This is seen in Figure 17.4, which shows that the most common number of successes is 3, as expected.



> binomData <- data.frame(Successes=rbinom(n=10000, size=10, prob=.3))

> ggplot(binomData, aes(x=Successes)) + geom\_histogram(binwidth=1)

To see how the binomial distribution is well approximated by the normal distribution as the number of trials grows large, we run similar experiments with differing numbers of trials and graph the results, as shown in Figure 17.5.



> # create a data.frame with Successes being the 10,000 random draws

> # Size equals 5 for all 10,000 rows

> binom5 <- data.frame(Successes=rbinom(n=10000, size=5, prob=.3), Size=5)

> dim(binom5)

[1] 10000 2

> head(binom5)

Successes Size

1 2 5

2 1 5

3 2 5

4 1 5

5 2 5

6 2 5

> # similar as before, still 10,000 rows

> # numbers are drawn from a distribution with a different size

> # Size now equals 10 for all 10,000 rows

> binom10 <- data.frame(Successes=rbinom(n=10000, size=10, prob=.3), Size=10)

> dim(binom10)

[1] 10000 2

> head(binom10)

Successes Size

1 2 10

2 2 10

3 1 10

4 2 10

5 4 10

6 1 10

> binom100 <- data.frame(Successes=rbinom(n=10000, size=100, prob=.3), Size=100)

>

> binom1000 <- data.frame(Successes=rbinom(n=10000, size=1000, prob=.3), Size=1000)

>

> # combine them all into one data.frame

> binomAll <- rbind(binom5, binom10, binom100, binom1000)

> dim(binomAll)

[1] 40000 2

> head(binomAll, 10)

Successes Size

1 2 5

2 1 5

3 2 5

4 1 5

5 2 5

6 2 5

7 1 5

8 1 5

9 2 5

10 1 5

> tail(binomAll, 10)

Successes Size

39991 288 1000

39992 289 1000

39993 297 1000

39994 327 1000

39995 336 1000

39996 290 1000

39997 310 1000

39998 328 1000

39999 281 1000

40000 307 1000

> # build the plot

> # histograms only need an x aesthetic

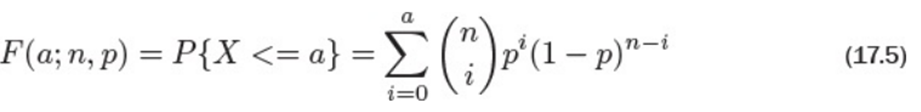
> # it is faceted (broken up) based on the values of Size

> # these are 5, 10, 100, 1000

> ggplot(binomAll, aes(x=Successes)) + geom\_histogram() +

+ facet\_wrap(~ Size, scales="free")

The cumulative distribution function is



where n and p are the number of trials and the probability of success, respectively, as before. Similar to the normal distribution functions, dbinom and pbinom provide the density (probability of an exact value) and distribution (cumulative probability), respectively, for the binomial distribution.

> # probability of 3 successes out of 10

> dbinom(x=3, size=10, prob=.3)

[1] 0.2668279

> # probability of 3 or fewer successes out of 10

> pbinom(q=3, size=10, prob=.3)

[1] 0.6496107

> # both functions can be vectorized

> dbinom(x=1:10, size=10, prob=.3)

[1] 0.1210608210 0.2334744405 0.2668279320 0.2001209490 0.1029193452

[6] 0.0367569090 0.0090016920 0.0014467005 0.0001377810 0.0000059049

> pbinom(q=1:10, size=10, prob=.3)

[1] 0.1493083 0.3827828 0.6496107 0.8497317 0.9526510 0.9894079

[7] 0.9984096 0.9998563 0.9999941 1.0000000Given a certain probability, qbinom returns the quantile, which for this distribution is the number of successes.

> qbinom(p=.3, size=10, prob=.3)

[1] 2

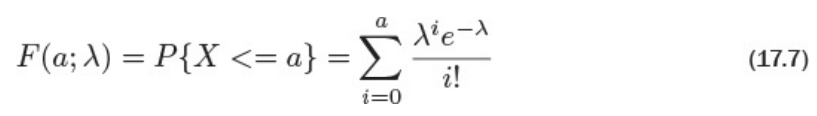
> qbinom(p=c(.3, .35, .4, .5, .6), size=10, prob=.3)

[1] 2 2 3 3 3

**Poisson Distribution**

Another popular distribution is the Poisson distribution, which is for count data. Its probability mass function is





and the cumulative distribution is where λ is both the mean and variance. To generate random counts, the density, the distribution and quantiles use rpois, dpois, ppois and qpois, respectively. As λ grows large the Poisson distribution begins to resemble the normal distribution. To see this we will simulate 10,000 draws from the Poisson distribution and plot their histograms to see the shape.

> # generate 10,000 random counts from 5 different Poisson distributions

> pois1 <- rpois(n=10000, lambda=1)

> pois2 <- rpois(n=10000, lambda=2)

> pois5 <- rpois(n=10000, lambda=5)

> pois10 <- rpois(n=10000, lambda=10)

> pois20 <- rpois(n=10000, lambda=20)

> pois <- data.frame(Lambda.1=pois1, Lambda.2=pois2,

+ Lambda.5=pois5, Lambda.10=pois10, Lambda.20=pois20)

> # load reshape2 package to melt the data to make it easier to plot

> library(reshape2)

> # melt the data into a long format

> pois <- melt(data=pois, variable.name="Lambda", value.name="x")

> # load the stringr package to help clean up the new column name

> library(stringr)

> # clean up the Lambda to just show the value for that lambda

> pois$Lambda <- as.factor(as.numeric(str\_extract(string=pois$Lambda,

+ pattern="\\d+")))

> head(pois)

Lambda x

1 1 1

2 1 1

3 1 1

4 1 2

5 1 2

6 1 0

> tail(pois)

Lambda x

49995 20 22

49996 20 15

49997 20 24

49998 20 23

49999 20 20

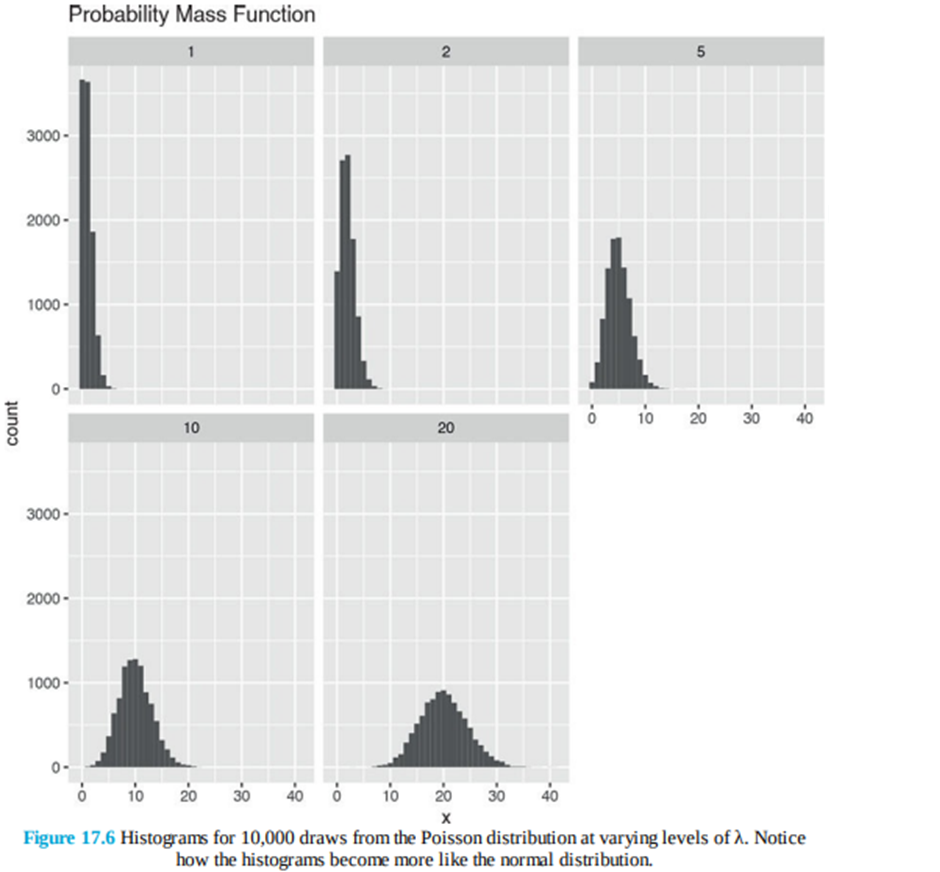
50000 20 23

Now we will plot a separate histogram for each value of λ, as shown in Figure 17.6.

> library(ggplot2)

> ggplot(pois, aes(x=x)) + geom\_histogram(binwidth=1) +

+ facet\_wrap(~ Lambda) + ggtitle("Probability Mass Function")



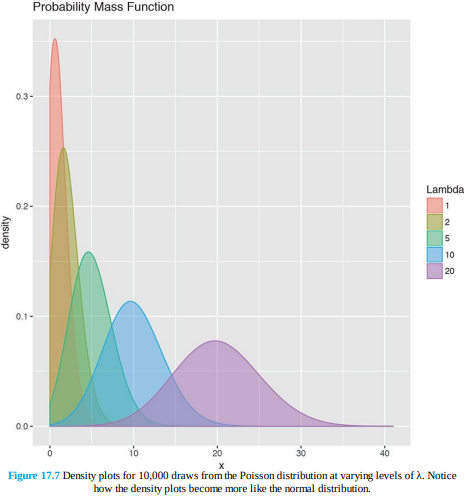
> ggplot(pois, aes(x=x)) +

+ geom\_density(aes(group=Lambda, color=Lambda, fill=Lambda),

+ adjust=4, alpha=1/2) +

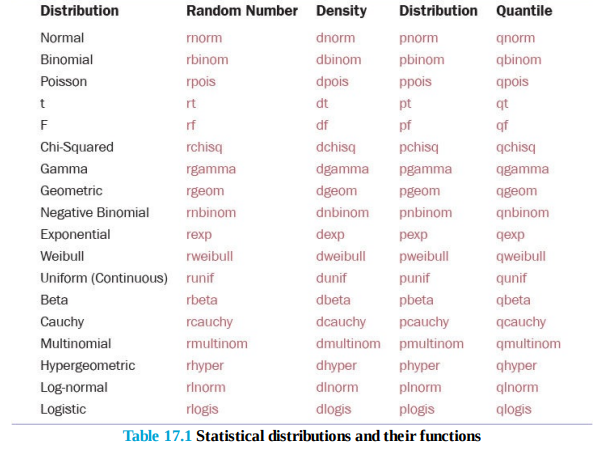
+ scale\_color\_discrete() + scale\_fill\_discrete() +

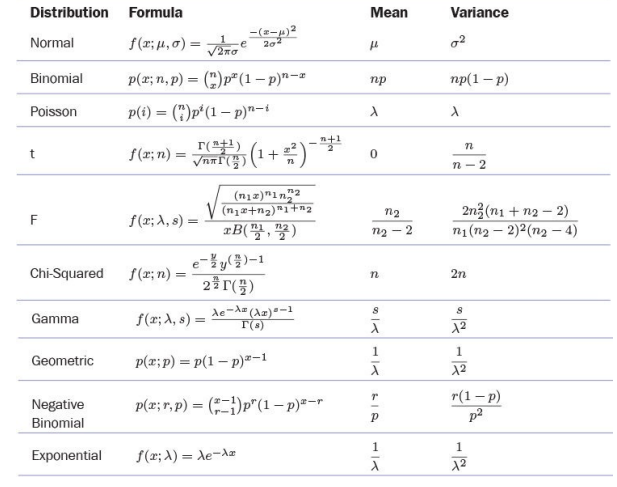
+ ggtitle("Probability Mass Function")

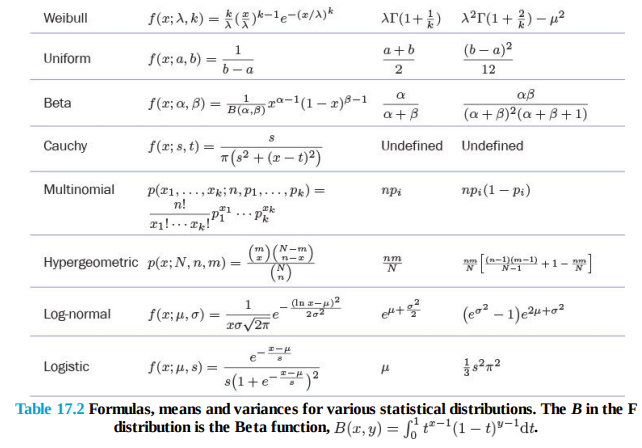


**Other Distributions**

R supports many distributions, some of which are very common, while others are quite obscure. They are listed in Table 17.1; the mathematical formulas, means and variances are in Table 17.2.







**Basic Statistics**

Some of the most common tools used in statistics are means, variances, correlations and t-tests. These are all well represented in R with easy-to-use functions such as mean, var, cor and t.test.

**Summary Statistics**

The first thing many people think of in relation to statistics is the average, or mean, as it is properly called. We start by looking at some simple numbers and later in the chapter play with bigger datasets. First we generate a random sampling of 100 numbers between 1 and 100.

> x <- sample(x=1:100, size=100, replace=TRUE)

> x

[1] 53 89 28 97 35 51 21 55 47 3 46 35 86 66 51 20 41 15 10 22 31

[22] 86 19 13 10 59 60 58 90 11 54 79 45 49 23 91 80 30 83 69 20 76

[43] 2 42 35 51 76 77 90 84 12 36 79 38 68 87 72 17 20 57 61 83 23

[64] 61 64 41 31 74 35 20 85 89 64 73 11 36 12 81 10 64 39 4 69 42

[85] 41 85 84 66 76 23 47 56 50 82 21 67 89 57 6 13sample uniformly draws size entries from x. Setting replace=TRUE means that the same number can be drawn multiple times.

Now that we have a vector of data we can calculate the mean.

> mean(x)

[1] 49.85

This is the simple arithmetic mean.



Simple enough. Because this is statistics, we need to consider cases where some data is missing. To create this we take x and randomly set 20 percent of the elements to NA.

> # copy x

> y <- x

> # choose a random 20 elements, using sample, to set to NA

> y[sample(x=1:100, size=20, replace=FALSE)] <- NA

> y

[1] 53 89 28 97 35 51 21 55 47 NA 46 35 86 NA NA NA 41 15 10 22 31

[22] NA 19 13 NA 59 60 NA 90 11 NA 79 45 NA 23 91 80 30 83 69 20 76

[43] 2 42 35 51 76 77 NA 84 NA 36 79 38 NA 87 72 17 20 57 61 83 NA

[64] 61 64 41 31 74 NA 20 NA 89 64 73 NA 36 12 NA 10 64 39 4 NA 42

[85] 41 85 84 66 76 23 47 56 50 82 21 67 NA NA 6 13

Using mean on y will return NA. This is because, by default, if mean encounters even one element that is NA it will return NA. This is to avoid providing misleading information.

> mean(y, na.rm=TRUE)

[1] 49.6

To calculate the weighted mean of a set of numbers, the function weighted.mean takes a vector of numbers and a vector of weights. It also has an optional argument, na.rm, to remove NAs before calculating; otherwise, a vector with NA values will return NA.

> grades <- c(95, 72, 87, 66)

> weights <- c(1/2, 1/4, 1/8, 1/8)

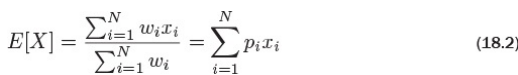
> mean(grades)

[1] 80

> weighted.mean(x=grades, w=weights)

[1] 84.625

The formula for weighted.mean is in Equation 18.2, which is the same as the expected value of a random variable.



Another vitally important metric is the variance, which is calculated with var.

> var(x)

[1] 724.5328

This calculates variance as



which can be verified in R.

> var(x)

[1] 724.5328

> sum((x - mean(x))^2) / (length(x) - 1)

[1] 724.5328

Standard deviation is the square root of variance and is calculated with sd. Like mean and var, sd has the na.rm argument to remove NAs before computation; otherwise, any NAs will cause the answer to be NA.

> sqrt(var(x))

[1] 26.91715

> sd(x)

[1] 26.91715

> sd(y)

[1] NA

> sd(y, na.rm=TRUE)

[1] 26.48506

Other commonly used functions for summary statistics are min, max and median. Of course, all of these also have na.rm arguments.

> min(x)

[1] 2

> max(x)

[1] 97

> median(x)

[1] 51

> min(y)

[1] NA

> min(y, na.rm=TRUE)

[1] 2The median, as calculated before, is the middle of an ordered set of numbers. For instance, the median of 5, 2, 1, 8 and 6 is 5. In the case when there are an even amount of numbers, the median is the mean of the middle two numbers. For 5, 1, 7, 4, 3, 8, 6 and 2, the median is 4.5. A helpful function that computes the mean, minimum, maximum and median is summary. There is no need to specify na.rm because if there are NAs, they are automatically removed and their count is included in the results.

> summary(x)

Min. 1st Qu. Median Mean 3rd Qu. Max.

2.00 23.00 51.00 49.85 74.50 97.00

> summary(y)

Min. 1st Qu. Median Mean 3rd Qu. Max. NA's

2.00 26.75 48.50 49.60 74.50 97.00 20

This summary also displayed the first and third quantiles. These can be computed using quantile.> # calculate the 25th and 75th quantile

> quantile(x, probs=c(.25, .75))

25% 75%

23.0 74.5

> # try the same on y

> quantile(y, probs=c(.25, .75))

Error in quantile.default(y, probs = c(0.25, 0.75)): missing values and NaN's

not allowed if 'na.rm' is FALSE

> # this time use na.rm=TRUE

> quantile(y, probs=c(.25, .75), na.rm=TRUE)

25% 75%

26.75 74.50

> # compute other quantiles

> quantile(x, probs=c(.1, .25, .5, .75, .99))

10% 25% 50% 75% 99%

12.00 23.00 51.00 74.50 91.06

Quantiles are numbers in a set where a certain percentage of the numbers are smaller than that quantile. For instance, of the numbers one through 200, the 75th quantile—the number that is larger than 75 percent of the numbers—is 150.25.

**Correlation and Covariance**

When dealing with more than one variable, we need to test their relationship with each other. Two simple, straightforward methods are correlation and covariance. To examine these concepts we look at the economics data from ggplot2.

> library(ggplot2)

> head(economics)

# A tibble: 6 × 8

date pce pop psavert uempmed unemploy year month

<date> <dbl> <int> <dbl> <dbl> <int> <dbl> <ord>

1 1967-07-01 507.4 198712 12.5 4.5 2944 1967 Jul

2 1967-08-01 510.5 198911 12.5 4.7 2945 1967 Aug

3 1967-09-01 516.3 199113 11.7 4.6 2958 1967 Sep

4 1967-10-01 512.9 199311 12.5 4.9 3143 1967 Oct

5 1967-11-01 518.1 199498 12.5 4.7 3066 1967 Nov

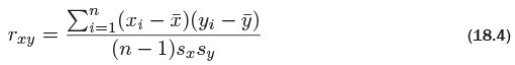
6 1967-12-01 525.8 199657 12.1 4.8 3018 1967 Dec

In the economics dataset, pce is personal consumption expenditures and psavert is the personal savings rate. We calculate their correlation using cor.

> cor(economics$pce, economics$psavert)

[1] -0.837069

This very low correlation makes sense because spending and saving are opposites of each other.



Correlation is defined as where and are the means of x and y, and sx and sy are the standard deviations of x and y. It can range between -1 and 1, with higher positive numbers meaning a closer relationship between the two variables, lower negative numbers meaning an inverse relationship and numbers near zero meaning no relationship. This can be easily checked by computing Equation 18.4.

> # use cor to calculate correlation

> cor(economics$pce, economics$psavert)

[1] -0.837069

> ## calculate each part of correlation

> xPart <- economics$pce - mean(economics$pce)

> yPart <- economics$psavert - mean(economics$psavert)

> nMinusOne <- (nrow(economics) - 1)

> xSD <- sd(economics$pce)

> ySD <- sd(economics$psavert)

> # use correlation formula

> sum(xPart \* yPart) / (nMinusOne \* xSD \* ySD)

[1] -0.837069

To compare multiple variables at once, use cor on a matrix (only for numeric variables).

> cor(economics[, c(2, 4:6)])

pce psavert uempmed unemploy

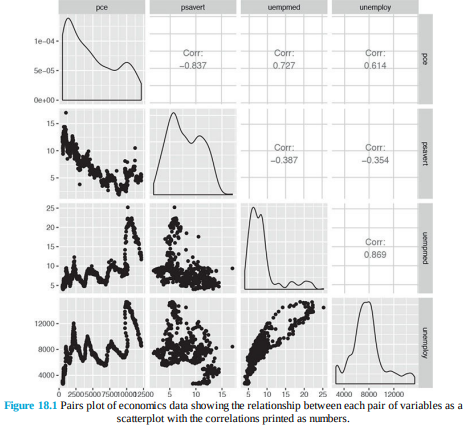
pce 1.0000000 -0.8370690 0.7273492 0.6139997

psavert -0.8370690 1.0000000 -0.3874159 -0.3540073

uempmed 0.7273492 -0.3874159 1.0000000 0.8694063

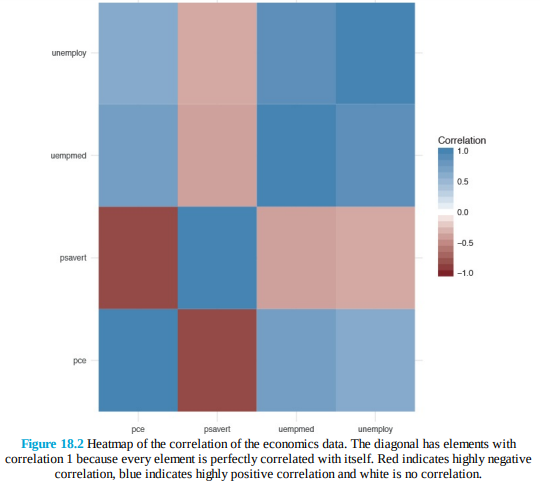
unemploy 0.6139997 -0.3540073 0.8694063 1.0000000

Because this is just a table of numbers, it would be helpful to also visualize the information using a plot. For this we use the ggpairs function from the GGally package (a collection of helpful plots built on ggplot2) shown in Figure 18.1. This shows a scatterplot of every variable in the data against every other variable. Loading GGally also loads the reshape package, which causes namespace issues with the newer reshape2 package. So rather than load GGally, we call its function using the :: operator, which allows access to functions within a package without loading it.



> GGally::ggpairs(economics[, c(2, 4:6)])

This is similar to a small multiples plot except that each pane has different x- and y-axes. While this shows the original data, it does not actually show the correlation. To show that we build a heatmap of the correlation numbers, as shown in Figure 18.2. High positive correlation indicates a positive relationship between the variables, high negative correlation indicates a negative relationship between the variables and near zero correlation indicates no strong relationship.



> # load the reshape package for melting the data

> library(reshape2)

> # load the scales package for some extra plotting features

> library(scales)

> # build the correlation matrix

> econCor <- cor(economics[, c(2, 4:6)])

> # melt it into the long format

> econMelt <- melt(econCor, varnames=c("x", "y"), value.name="Correlation")

> # order it according to the correlation

> econMelt <- econMelt[order(econMelt$Correlation), ]

> # display the melted data

> econMelt

x y Correlation

2 psavert pce -0.8370690

5 pce psavert -0.8370690

7 uempmed psavert -0.3874159

10 psavert uempmed -0.3874159

8 unemploy psavert -0.3540073

14 psavert unemploy -0.3540073

4 unemploy pce 0.6139997

13 pce unemploy 0.6139997

3 uempmed pce 0.7273492

9 pce uempmed 0.7273492

12 unemploy uempmed 0.8694063

15 uempmed unemploy 0.8694063

1 pce pce 1.0000000

6 psavert psavert 1.0000000

11 uempmed uempmed 1.0000000

16 unemploy unemploy 1.0000000

> ## plot it with ggplot

> # initialize the plot with x and y on the x and y axes

> ggplot(econMelt, aes(x=x, y=y)) +

+ # draw tiles filling the color based on Correlation

+ geom\_tile(aes(fill=Correlation)) +

+ # make the fill (color) scale a three color gradient with muted

+ # red for the low point, white for the middle and steel blue

+ # for the high point

+ # the guide should be a colorbar with no ticks, whose height is

+ # 10 lines

+ # limits indicates the scale should be filled from -1 to 1

+ scale\_fill\_gradient2(low=muted("red"), mid="white",

+ high="steelblue",

+ guide=guide\_colorbar(ticks=FALSE, barheight=10),

+ limits=c(-1, 1)) +

+ # use the minimal theme so there are no extras in the plot

+ theme\_minimal() +

+ # make the x and y labels blank

+ labs(x=NULL, y=NULL)

Missing data is just as much a problem with cor as it is with mean and var, but is dealt with differently because multiple columns are being considered simultaneously. Instead of specifying na.rm=TRUE to remove NA entries, one of “all.obs”, “complete.obs”, “pairwise.complete.obs”, “everything” or “na.or.complete” is used. To illustrate this we first make a five-column matrix where only the fourth and fifth columns have no NA values; the other columns have one or two NAs.

> m <- c(9, 9, NA, 3, NA, 5, 8, 1, 10, 4)

> n <- c(2, NA, 1, 6, 6, 4, 1, 1, 6, 7)

> p <- c(8, 4, 3, 9, 10, NA, 3, NA, 9, 9)

> q <- c(10, 10, 7, 8, 4, 2, 8, 5, 5, 2)

> r <- c(1, 9, 7, 6, 5, 6, 2, 7, 9, 10)

> # combine them together

> theMat <- cbind(m, n, p, q, r)

The first option for use is “everything”, which means that the entirety of all columns must be free of NAs; otherwise the result is NA. Running this should generate a matrix of all NAs except ones on the diagonal—because a vector is always perfectly correlated with itself—and between q and r. With the second option —“all.obs”—even a single NA in any column will cause an error.

> cor(theMat, use="everything")

m n p q r

m 1 NA NA NA NA

n NA 1 NA NA NA

p NA NA 1 NA NA

q NA NA NA 1.0000000 -0.4242958

r NA NA NA -0.4242958 1.0000000

> cor(theMat, use="all.obs")

Error in cor(theMat, use = "all.obs"): missing observations in cov/cor

The third and fourth options—“complete.obs” and “na.or.complete”—work similarly to each other in that they keep only rows where every entry is not NA. That means our matrix will be reduced to rows 1, 4, 7, 9 and 10, and then have its correlation computed. The difference is that “complete.obs” will return an error if not a single complete row can be found, while “na.or.complete” will return NA in that case.

> cor(theMat, use="complete.obs")

m n p q r

m 1.0000000 -0.5228840 -0.2893527 0.2974398 -0.3459470

n -0.5228840 1.0000000 0.8090195 -0.7448453 0.9350718

p -0.2893527 0.8090195 1.0000000 -0.3613720 0.6221470

q 0.2974398 -0.7448453 -0.3613720 1.0000000 -0.9059384

r -0.3459470 0.9350718 0.6221470 -0.9059384 1.0000000

> cor(theMat, use="na.or.complete")

m n p q r

m 1.0000000 -0.5228840 -0.2893527 0.2974398 -0.3459470

n -0.5228840 1.0000000 0.8090195 -0.7448453 0.9350718

p -0.2893527 0.8090195 1.0000000 -0.3613720 0.6221470

q 0.2974398 -0.7448453 -0.3613720 1.0000000 -0.9059384

r -0.3459470 0.9350718 0.6221470 -0.9059384 1.0000000

> # calculate the correlation just on complete rows

> cor(theMat[c(1, 4, 7, 9, 10), ])

m n p q r

m 1.0000000 -0.5228840 -0.2893527 0.2974398 -0.3459470

n -0.5228840 1.0000000 0.8090195 -0.7448453 0.9350718

p -0.2893527 0.8090195 1.0000000 -0.3613720 0.6221470

q 0.2974398 -0.7448453 -0.3613720 1.0000000 -0.9059384

r -0.3459470 0.9350718 0.6221470 -0.9059384 1.0000000

> # compare "complete.obs" and computing on select rows

> # should give the same result

> identical(cor(theMat, use="complete.obs"),

+ cor(theMat[c(1, 4, 7, 9, 10), ]))

[1] TRUEThe final option is “pairwise.complete”, which is much more inclusive. It compares two columns at a time and keeps rows—for those two columns—where neither entry is NA. This is essentially the same as computing the correlation between every combination of two columns with use set to “complete.obs”.

> # the entire correlation matrix

> cor(theMat, use="pairwise.complete.obs")

m n p q r

m 1.00000000 -0.02511812 -0.3965859 0.4622943 -0.2001722

n -0.02511812 1.00000000 0.8717389 -0.5070416 0.5332259

p -0.39658588 0.87173889 1.0000000 -0.5197292 0.1312506

q 0.46229434 -0.50704163 -0.5197292 1.0000000 -0.4242958

r -0.20017222 0.53322585 0.1312506 -0.4242958 1.0000000

> # compare the entries for m vs n to this matrix

> cor(theMat[, c("m", "n")], use="complete.obs")

m n

m 1.00000000 -0.02511812

n -0.02511812 1.00000000

> # compare the entries for m vs p to this matrix

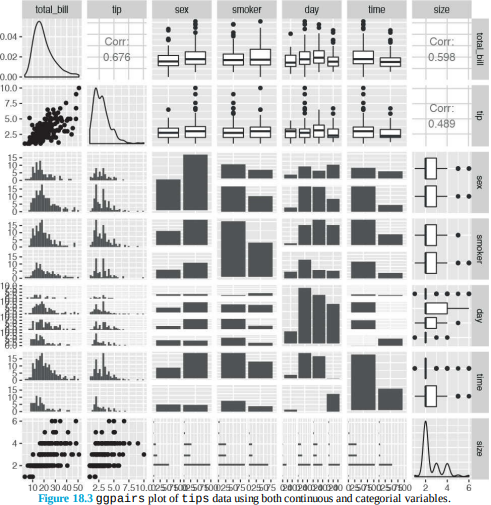
> cor(theMat[, c("m", "p")], use="complete.obs")

m p

m 1.0000000 -0.3965859

p -0.3965859 1.0000000

To see ggpairs in all its glory, look at tips data from the reshape2 package in Figure 18.3. This shows every pair of variables in relation to each other building either histograms, boxplots or scatterplots depending on the combination of continuous and discrete variables. While a data dump like this looks really nice, it is not always the most informative form of exploratory data analysis.



> data(tips, package="reshape2")

> head(tips)

total\_bill tip sex smoker day time size

1 16.99 1.01 Female No Sun Dinner 2

2 10.34 1.66 Male No Sun Dinner 3

3 21.01 3.50 Male No Sun Dinner 3

4 23.68 3.31 Male No Sun Dinner 2

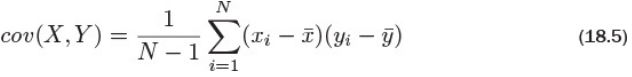
5 24.59 3.61 Female No Sun Dinner 4

6 25.29 4.71 Male No Sun Dinner 4

> GGally::ggpairs(tips)

No discussion of correlation would be complete without the old refrain, “Correlation does not mean causation.” In other words, just because two variables are correlated does not mean they have an effect on each other. This is exemplified in xkcd 1 comic number 552. There is even an R package, RXKCD, for downloading individual comics. Running the following code should generate a pleasant surprise.

> library(RXKCD) > getXKCD(which="552") Similar to correlation is covariance, which is like a variance between variables, its formula is in Equation 18.5. Notice the similarity to correlation in Equation 18.4 and variance in Equation 18.3.



The cov function works similarly to the cor function, with the same arguments for dealing with missing data. In fact, ?cor and ?cov pull up the same help menu.

> cov(economics$pce, economics$psavert)

[1] -9361.028

> cov(economics[, c(2, 4:6)])

pce psavert uempmed unemploy

pce 12811296.900 -9361.028324 10695.023873 5806187.162

psavert -9361.028 9.761835 -4.972622 -2922.162

uempmed 10695.024 -4.972622 16.876582 9436.074

unemploy 5806187.162 -2922.161618 9436.074287 6979955.661

> # check that cov and cor\*sd\*sd are the same

> identical(cov(economics$pce, economics$psavert),

+ cor(economics$pce, economics$psavert) \*

+ sd(economics$pce) \* sd(economics$psavert))

[1] TRUE

**T-Tests**

In traditional statistics classes, the t-test—invented by William Gosset while working at the Guinness brewery—is taught for conducting tests on the mean of data or for comparing two sets of data. To illustrate this we continue to use the tips data from Section 18.2.

> head(tips)

total\_bill tip sex smoker day time size

1 16.99 1.01 Female No Sun Dinner 2

2 10.34 1.66 Male No Sun Dinner 3

3 21.01 3.50 Male No Sun Dinner 3

4 23.68 3.31 Male No Sun Dinner 2

5 24.59 3.61 Female No Sun Dinner 4

6 25.29 4.71 Male No Sun Dinner 4

> # sex of the bill payer

> unique(tips$sex)

[1] Female Male

Levels: Female Male

> # day of the week

> unique(tips$day)

[1] Sun Sat Thur Fri

Levels: Fri Sat Sun Thur

**One-Sample T-Test**

First we conduct a one-sample t-test on whether the average tip is equal to $2.50. This test essentially calculates the mean of data and builds a confidence interval. If the value we are testing falls within that confidence interval, then we can conclude that it is the true value for the mean of the data; otherwise, we conclude that it is not the true mean.

> t.test(tips$tip, alternative="two.sided", mu=2.50)

One Sample t-test

data: tips$tip

t = 5.6253, df = 243, p-value = 5.08e-08

alternative hypothesis: true mean is not equal to 2.5

95 percent confidence interval:

2.823799 3.172758

sample estimates:

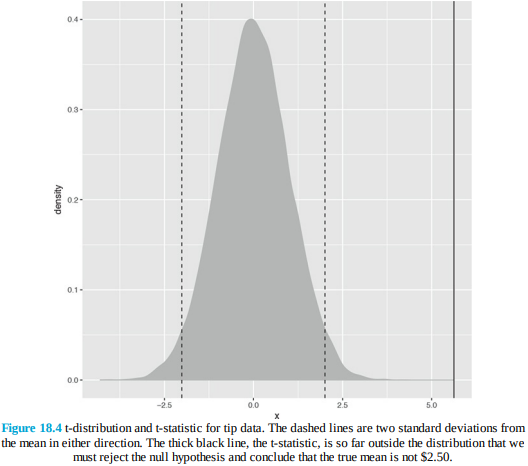
mean of x

2.998279

The output very nicely displays the setup and results of the hypothesis test of whether the mean is equal to $2.50. It prints the t-statistic, the degrees of freedom and p-value. It also provides the 95 percent confidence interval and mean for the variable of interest. The p-value indicates that the null hypothesis 2 should be rejected, and we conclude that the mean is not equal to $2.50. 2. The null hypothesis is what is considered to be true; in this case that the mean is equal to $2.50. We encountered a few new concepts here. The t-statistic is the ratio where the numerator is the difference between the estimated mean and the hypothesized mean and the denominator is the standard error of the estimated mean. It is defined in Equation 18.6.



Here, is the estimated mean, μ0 is the hypothesized mean and is the standard error of . 3 3. s is the standard deviation of the data and n is the number of observations. If the hypothesized mean is correct, then we expect the t-statistic to fall somewhere in the middle— about two standard deviations from the mean—of the t distribution. In Figure 18.4 we see that the thick black line, which represents the estimated mean, falls so far outside the distribution that we must conclude that the mean is not equal to $2.50.



> ## build a t distribution

> randT <- rt(30000, df=NROW(tips)-1)

>

> # get t-statistic and other information

> tipTTest <- t.test(tips$tip, alternative="two.sided", mu=2.50)

>

> # plot it

> ggplot(data.frame(x=randT)) +

+ geom\_density(aes(x=x), fill="grey", color="grey") +

+ geom\_vline(xintercept=tipTTest$statistic) +

+ geom\_vline(xintercept=mean(randT) + c(-2, 2)\*sd(randT), linetype=2)

The p-value is an often misunderstood concept. Despite all the misinterpretations, a p-value is the probability, if the null hypothesis were correct, of getting as extreme, or more extreme, a result. It is a measure of how extreme the statistic—in this case, the estimated mean— is. If the statistic is too extreme, we conclude that the null hypothesis should be rejected. The main problem with p-values, however, is determining what should be considered too extreme. Ronald A. Fisher, the father of modern statistics, decided we should consider a p-value that is smaller than 0.10, 0.05 or 0.01 to be too extreme. While those p-values have been the standard for decades, they were arbitrarily chosen, leading some modern data scientists to question their usefulness. In this example, the p-value is 5.0799885 × 10 −8; this is smaller than 0.01, so we reject the null hypothesis. Degrees of freedom is another difficult concept to grasp but is pervasive throughout statistics. It represents the effective number of observations. Generally, the degrees of freedom for some statistic or distribution is the number of observations minus the number of parameters being estimated. In the case of the t distribution, one parameter, the standard error, is being estimated. In this example, there are nrow(tips)-1=243 degrees of freedom. Next we conduct a one-sided t-test to see if the mean is greater than $2.50.

> t.test(tips$tip, alternative="greater", mu=2.50)

One Sample t-test

data: tips$tip

t = 5.6253, df = 243, p-value = 2.54e-08

alternative hypothesis: true mean is greater than 2.5

95 percent confidence interval:

2.852023 Inf

sample estimates:

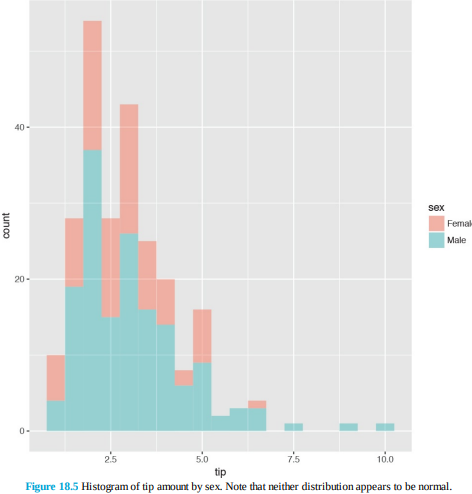
mean of x

2.998279

Once again, the p-value indicates that we should reject the null hypothesis and conclude that the mean is greater than $2.50, which coincides nicely with the confidence interval.

**Two-Sample T-Test**

More often than not the t-test is used for comparing two samples. Continuing with the tips data, we compare how female and male diners tip. Before running the t-test, however, we first need to check the variance of each sample. A traditional t-test requires both groups to have the same variance, whereas the Welch two-sample t-test can handle groups with differing variances. We explore this both numerically and visually in Figure 18.5.



> # first just compute the variance for each group

> # using the the formula interface

> # calculate the variance of tip for each level of sex

> aggregate(tip ~ sex, data=tips, var)

sex tip

1 Female 1.344428

2 Male 2.217424

> # now test for normality of tip distribution

> shapiro.test(tips$tip)

Shapiro-Wilk normality test

data: tips$tip

W = 0.89781, p-value = 8.2e-12

> shapiro.test(tips$tip[tips$sex == "Female"])

Shapiro-Wilk normality test

data: tips$tip[tips$sex == "Female"]

W = 0.95678, p-value = 0.005448

> shapiro.test(tips$tip[tips$sex == "Male"])

Shapiro-Wilk normality test

data: tips$tip[tips$sex == "Male"]

W = 0.87587, p-value = 3.708e-10

> # all the tests fail so inspect visually

> ggplot(tips, aes(x=tip, fill=sex)) +

+ geom\_histogram(binwidth=.5, alpha=1/2)

Since the data do not appear to be normally distributed, neither the standard F-test (via the var.test function) nor the Bartlett test (via the bartlett.test function) will suffice. So we use the nonparametric Ansari-Bradley test to examine the equality of variances.

> ansari.test(tip ~ sex, tips)

Ansari-Bradley test

data: tip by sex

AB = 5582.5, p-value = 0.376

alternative hypothesis: true ratio of scales is not equal to 1

This test indicates that the variances are equal, meaning we can use the standard two-sample t-test.

> # setting var.equal=TRUE runs a standard two sample t-test

> # var.equal=FALSE (the default) would run the Welch test

> t.test(tip ~ sex, data=tips, var.equal=TRUE)

Two Sample t-test

data: tip by sex

t = -1.3879, df = 242, p-value = 0.1665

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:

-0.6197558 0.1074167

sample estimates:

mean in group Female mean in group Male

2.833448 3.089618According to this test, the results were not significant, and we should conclude that female and male diners tip roughly equally. While all this statistical rigor is nice, a simple rule of thumb would be to see if the two means are within two standard deviations of each other.

> library(plyr)

> tipSummary <- ddply(tips, "sex", summarize,

+ tip.mean=mean(tip), tip.sd=sd(tip),

+ Lower=tip.mean - 2\*tip.sd/sqrt(NROW(tip)),

+ Upper=tip.mean + 2\*tip.sd/sqrt(NROW(tip)))

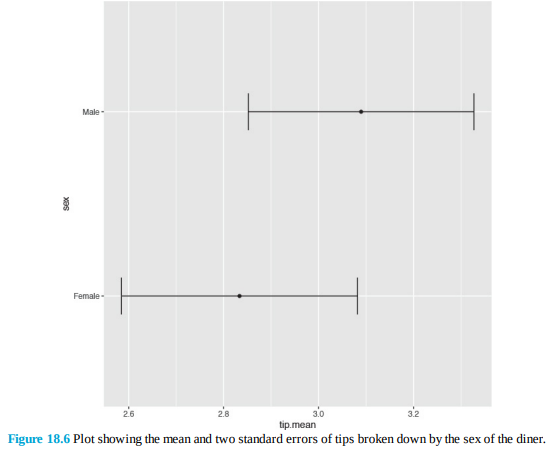
> tipSummary

sex tip.mean tip.sd Lower Upper

1 Female 2.833448 1.159495 2.584827 3.082070

2 Male 3.089618 1.489102 2.851931 3.327304

A lot happened in that code. First, ddply was used to split the data according to the levels of sex. It then applied the summarize function to each subset of the data. This function applied the indicated functions to the data, creating a new data.frame. As usual, we prefer visualizing the results rather than comparing numerical values. This requires reshaping the data a bit. The results, in Figure 18.6, clearly show the confidence intervals overlapping, suggesting that the means for the two sexes are roughly equivalent.

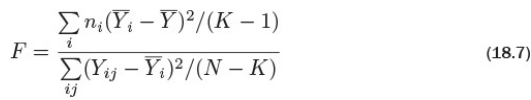


> ggplot(tipSummary, aes(x=tip.mean, y=sex)) + geom\_point() +

+ geom\_errorbarh(aes(xmin=Lower, xmax=Upper), height=.2)

**ANOVA**

After comparing two groups, the natural next step is comparing multiple groups. Every year, far too many students in introductory statistics classes are forced to learn the ANOVA (analysis of variance) test and memorize its formula, which is



where ni is the number of observations in group i, i is the mean of group i, is the overall mean, Yij is observation j in group i, N is the total number of observations and K is the number of groups. Not only is this a laborious formula that often turns off a lot of students from statistics; it is also a bit of an old-fashioned way of comparing groups. Even so, there is an R function—albeit rarely used—to conduct the ANOVA test. This also uses the formula interface where the left side is the variable of interest and the right side contains the variables that control grouping. To see this, we compare tips by day of the week, with levels Fri, Sat, Sun, Thur.

> tipAnova <- aov(tip ~ day - 1, tips)

In the formula the right side was day - 1. This might seem odd at first but will make more sense when comparing it to a call without -1.

> tipIntercept <- aov(tip ~ day, tips)

> tipAnova$coefficients

dayFri daySat daySun dayThur

2.734737 2.993103 3.255132 2.771452

> tipIntercept$coefficients

(Intercept) daySat daySun dayThur

2.73473684 0.25836661 0.52039474 0.03671477

Here we see that just using tip ~ day includes only Saturday, Sunday and Thursday, along with an intercept, while tip ~ day - 1 compares Friday, Saturday, Sunday and Thursday with no intercept. The importance of the intercept is made clear in Chapter 19, but for now it suffices that having no intercept makes the analysis more straightforward. The ANOVA tests whether any group is different from any other group but it does not specify which group is different. So printing a summary of the test just returns a single p-value.

> summary(tipAnova)

Df Sum Sq Mean Sq F value Pr(>F)

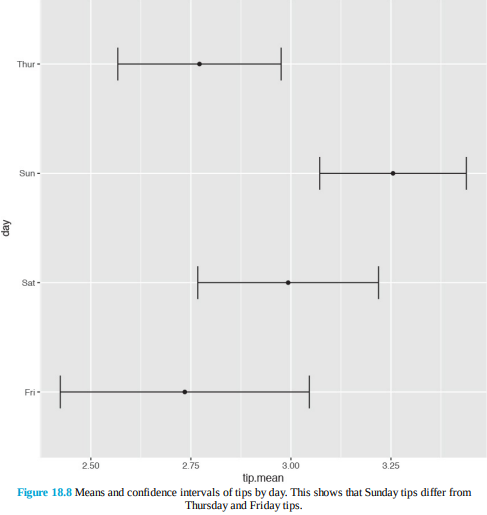
day 4 2203.0 550.8 290.1 <2e-16 \*\*\*

Residuals 240 455.7 1.9

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Since the test had a significant p-value, we would like to see which group differed from the others. The simplest way is to make a plot of the group means and confidence intervals and see which overlap. Figure 18.8 shows that tips on Sunday differ (just barely, at the 90 percent confidence level) from both Thursday and Friday



> tipsByDay <- ddply(tips, "day", plyr::summarize,

+ tip.mean=mean(tip), tip.sd=sd(tip),

+ Length=NROW(tip),

+ tfrac=qt(p=.90, df=Length-1),

+ Lower=tip.mean - tfrac\*tip.sd/sqrt(Length),

+ Upper=tip.mean + tfrac\*tip.sd/sqrt(Length)

+ )

>

> ggplot(tipsByDay, aes(x=tip.mean, y=day)) + geom\_point() +

+ geom\_errorbarh(aes(xmin=Lower, xmax=Upper), height=.3)

The use of NROW instead of nrow is to guarantee computation. Where nrow works only on data.frames and matrices, NROW returns the length of objects that have only one dimension.

> nrow(tips)

[1] 244

> NROW(tips)

[1] 244

> nrow(tips$tip)

NULL

> NROW(tips$tip)

[1] 244

To confirm the results from the ANOVA, individual t-tests could be run on each pair of groups. Traditional texts encourage adjusting the p-value to accommodate the multiple comparisons. However, some professors, including Andrew Gelman, suggest not worrying about adjustments for multiple comparisons. An alternative to the ANOVA is to fit a linear regression with one categorical variable and no intercept.

**Linear Models**

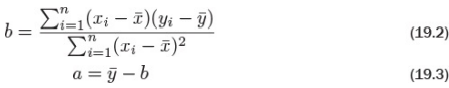
The workhorse of statistical analysis is the linear model, particularly regression. Originally invented by Francis Galton to study the relationships between parents and children, which he described as regressing to the mean, it has become one of the most widely used modelling techniques and has spawned other models such as generalized linear models, regression trees, penalized regression and many others. In this chapter we focus on simple and multiple regression.

**Simple Linear Regression**

In its simplest form regression is used to determine the relationship between two variables. That is, given one variable, it tells us what we can expect from the other variable. This powerful tool, which is frequently taught and can accomplish a great deal of analysis with minimal effort, is called simple linear regression. Before we go any further, we clarify some terminology. The outcome variable (what we are trying to predict) is called the response, and the input variable (what we are using to predict) is the predictor. Fields outside of statistics use other terms, such as measured variable, outcome variable and experimental variable for response, and covariate, feature and explanatory variable for predictor. Worst of all are the terms dependent (response) and independent (predictor) variables. These very names are misnomers. According to probability theory, if variable y is dependent on variable x, then variable x cannot be independent of variable y. So we stick with the terms response and predictor exclusively. The general idea behind simple linear regression is using the predictor to come up with some average value of the response. The relationship is defined as



Where



And



which is to say that there are normally distributed errors.

Equation 19.1 is essentially describing a straight line that goes through the data where a is the yintercept and b is the slope. This is illustrated using fathers’ and sons’ height data, which are plotted in Figure 19.1. In this case we are using the fathers’ heights as the predictor and the sons’ heights as the response. The blue line running through the points is the regression line and the gray band around it represents the uncertainty in the fit.

> data(father.son, package='UsingR')

> library(ggplot2)

> head(father.son)

fheight sheight

1 65.04851 59.77827

2 63.25094 63.21404

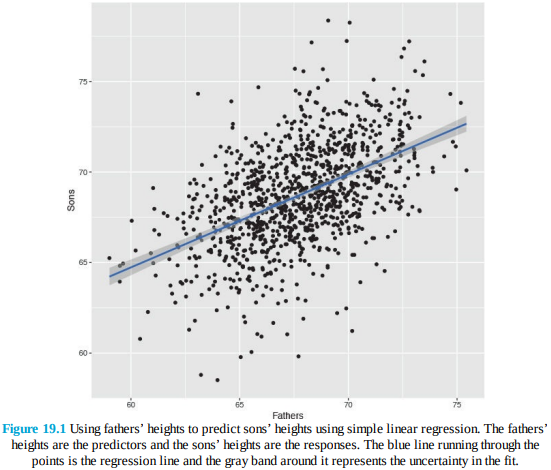
3 64.95532 63.34242

4 65.75250 62.79238

5 61.13723 64.28113

6 63.02254 64.24221

> ggplot(father.son, aes(x=fheight, y=sheight)) + geom\_point() + + geom\_smooth(method="lm") + labs(x="Fathers", y="Sons")



While that code generated a nice graph showing the results of the regression (generated with geom\_smooth(method=“lm”)), it did not actually make those results available to us. To actually calculate a regression, use the lm function.

> heightsLM <- lm(sheight ~ fheight, data=father.son)

> heightsLM

Call:

lm(formula = sheight ~ fheight, data = father.son)

Coefficients:

(Intercept) fheight

33.8866 0.5141

Here we once again see the formula notation that specifies to regress sheight (the response) on fheight (the predictor), using the father.son data, and adds the intercept term automatically. The results show coefficients for (Intercept) and fheight which is the slope for the fheight, predictor. The interpretation of this is that, for every extra inch of height in a father, we expect an extra half inch in height for his son. The intercept in this case does not make much sense because it represents the height of a son whose father had zero height, which obviously cannot exist in reality. While the point estimates for the coefficients are nice, they are not very helpful without the standard errors, which give the sense of uncertainty about the estimate and are similar to standard deviations. To quickly see a full report on the model, use summary.

> summary(heightsLM)

Call:

lm(formula = sheight ~ fheight, data = father.son)

Residuals:

Min 1Q Median 3Q Max

-8.8772 -1.5144 -0.0079 1.6285 8.9685

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 33.88660 1.83235 18.49 <2e-16 \*\*\*

fheight 0.51409 0.02705 19.01 <2e-16 \*\*\*

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.437 on 1076 degrees of freedom

Multiple R-squared: 0.2513,Adjusted R-squared: 0.2506

F-statistic: 361.2 on 1 and 1076 DF, p-value: < 2.2e-16

This prints out a lot more information about the model, including the standard errors, t-test values and p-values for the coefficients, the degrees of freedom, residual summary statistics (seen in more detail in Section 21.1) and the results of an F-test. This is all diagnostic information to check the fit of the model, and is covered in more detail in Section 19.2 about multiple regression.

**Multiple Regression**

The logical extension of simple linear regression is multiple regression, which allows for multiple predictors. The idea is still the same; we are still making predictions or inferences 1 on the response, but we now have more information in the form of multiple predictors. The math requires some matrix algebra but fortunately the lm function is used with very little extra effort.

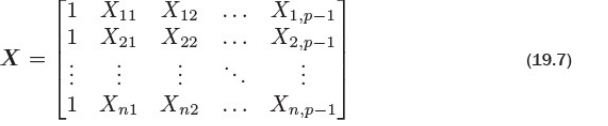
In this case the relationship between the response and the p predictors (p − 1 predictors and the intercept) is modeled as





where Y is the nx1 response vector

X is the nxp matrix (n rows and p − 1 predictors plus the intercept)



β is the px1 vector of coefficients (one for each predictor and intercept)





and ε is the nx1 vector of normally distributed errors with



which seems more complicated than simple regression but the algebra actually gets easier. The solution for the coefficients is simply written as in Equation 19.11.



To see this in action we use New York City condo evaluations for fiscal year 2011-2012, obtained through NYC Open Data. NYC Open Data is an initiative by New York City to make government more transparent and work better. It provides data on all manner of city services to the public for analysis, scrutiny and app building (through http://nycbigapps.com/). It has been surprisingly popular, spawning hundreds of mobile apps and being copied in other cities such as Chicago and Washington, DC.

Its Web site is at https://data.cityofnewyork.us/.

The original data were separated by borough with one file each for Manhattan, Brooklyn, Queens,

the Bronx and Staten Island, and contained extra information we will not be using. So we combined the five files into one, cleaned up the column names and posted it at

http://www.jaredlander.com/data/housing.csv. To access the data, either download it from that URLand use read.table on the now local file, or read it directly from the URL.

2.https://data.cityofnewyork.us/Finances/DOF-Condominium-Comparable-Rental-Income-Manhattan/dvzp-h4k9

3. https://data.cityofnewyork.us/Finances/DOF-Condominium-Comparable-Rental-Income-Brooklyn-/bss9-579f

4. https://data.cityofnewyork.us/Finances/DOF-Condominium-Comparable-Rental-Income-Queens-FY/jcih-dj9q

5. https://data.cityofnewyork.us/Property/DOF-Condominium-Comparable-Rental-Income-Bronx-FY-/3qfc-4tta

6. https://data.cityofnewyork.us/Finances/DOF-Condominium-Comparable-Rental-Income-Staten-Is/tkdy-59zg

> housing <- read.table("http://www.jaredlander.com/data/housing.csv", + sep = ",", header = TRUE, + stringsAsFactors = FALSE)

A few reminders about what that code does: sep specifies that commas were used to separate columns; header means the first row contains the column names; and stringsAsFactors leaves character columns as they are and does not convert them to factors, which speeds up loading time and also makes them easier to work with. Looking at the data, we see that we have a lot of columns and some bad names, so we should rename those.

> names(housing) <- c("Neighborhood", "Class", "Units", "YearBuilt",

+ "SqFt", "Income", "IncomePerSqFt", "Expense",

+ "ExpensePerSqFt", "NetIncome", "Value",

+ "ValuePerSqFt", "Boro")

> head(housing)

Neighborhood Class Units YearBuilt SqFt Income

1 FINANCIAL R9-CONDOMINIUM 42 1920 36500 1332615

2 FINANCIAL R4-CONDOMINIUM 78 1985 126420 6633257

3 FINANCIAL RR-CONDOMINIUM 500 NA 554174 17310000

4 FINANCIAL R4-CONDOMINIUM 282 1930 249076 11776313

5 TRIBECA R4-CONDOMINIUM 239 1985 219495 10004582

6 TRIBECA R4-CONDOMINIUM 133 1986 139719 5127687

IncomePerSqFt Expense ExpensePerSqFt NetIncome Value

1 36.51 342005 9.37 990610 7300000

2 52.47 1762295 13.94 4870962 30690000

3 31.24 3543000 6.39 13767000 90970000

4 47.28 2784670 11.18 8991643 67556006

5 45.58 2783197 12.68 7221385 54320996

6 36.70 1497788 10.72 3629899 26737996

ValuePerSqFt Boro

1 200.00 Manhattan

2 242.76 Manhattan

3 164.15 Manhattan

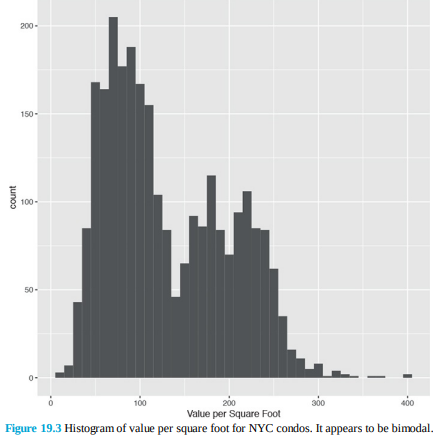
4 271.23 Manhattan

5 247.48 Manhattan

6 191.37 Manhattan

For these data the response is the value per square foot and the predictors are everything else. However, we ignore the income and expense variables, as they are actually just estimates based on an arcane requirement that condos be compared to rentals for valuation purposes. The first step is to visualize the data in some exploratory data analysis. The natural place to start is with a histogram of ValuePerSqFt, which is shown in Figure 19.3.

> ggplot(housing, aes(x=ValuePerSqFt)) + + geom\_histogram(binwidth=10) + labs(x="Value per Square Foot")



The bimodal nature of the histogram means there is something left to be explored. Mapping color to Boro in Figure 19.4a and faceting on Boro in Figure 19.4b reveal that Brooklyn and Queens make up one mode and Manhattan makes up the other, while there is not much data on the Bronx and Staten Island.

> ggplot(housing, aes(x=ValuePerSqFt, fill=Boro)) +

+ geom\_histogram(binwidth=10) + labs

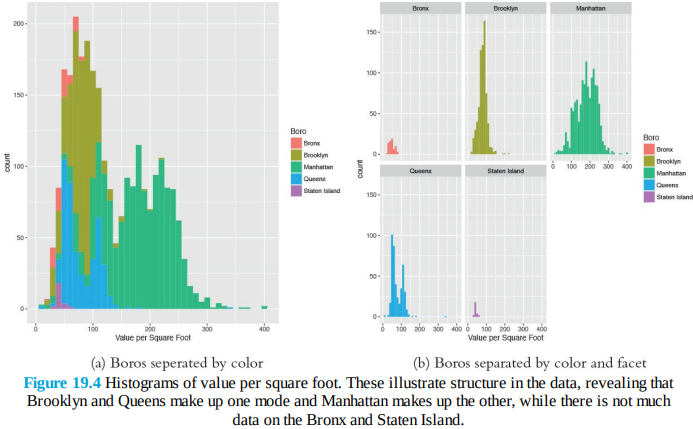
(x="Value per Square Foot")

> ggplot(housing, aes(x=ValuePerSqFt, fill=Boro)) +

+ geom\_histogram(binwidth=10) + labs

(x="Value per Square Foot") +

+ facet\_wrap(~Boro)



Next we should look at histograms for square footage and the number of units.

> ggplot(housing, aes(x=SqFt)) + geom\_histogram()

> ggplot(housing, aes(x=Units)) + geom\_histogram()

> ggplot(housing[housing$Units < 1000, ], aes(x=SqFt)) +

+ geom\_histogram()

> ggplot(housing[housing$Units < 1000, ], aes(x=Units)) +

+ geom\_histogram()

Figure 19.5 shows that there are quite a few buildings with an incredible number of units. Plotting scatterplots in Figure 19.6 of the value per square foot versus both number of units and square footage, with and without those outlying buildings, gives us an idea whether we can remove them from the analysis.

> ggplot(housing, aes(x=SqFt, y=ValuePerSqFt)) + geom\_point()

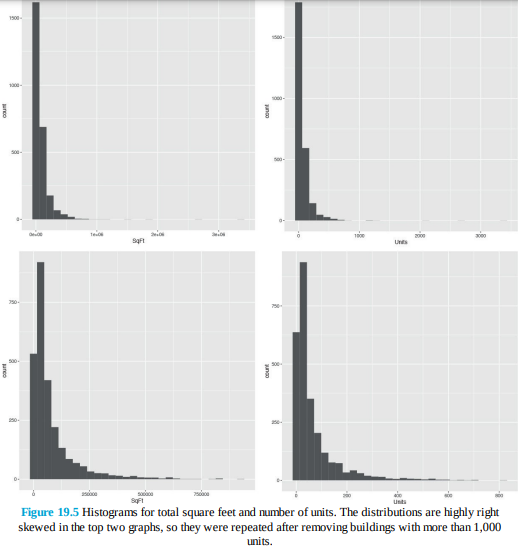
> ggplot(housing, aes(x=Units, y=ValuePerSqFt)) + geom\_point()

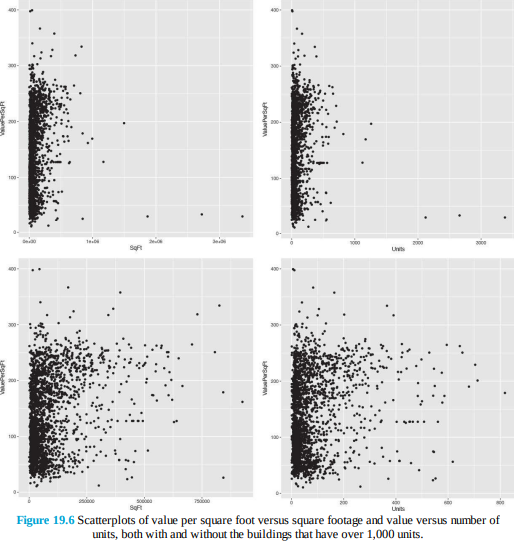
> ggplot(housing[housing$Units < 1000, ], aes(x=SqFt, y=ValuePerSqFt)) +

+ geom\_point()

> ggplot(housing[housing$Units < 1000, ], aes(x=Units, y=ValuePerSqFt)) +

+ geom\_point()





> # how many need to be removed?

> sum(housing$Units >= 1000)

[1] 6

> # remove them

> housing <- housing[housing$Units < 1000, ]

Even after we remove the outliers, it still seems like a log transformation of some data could be helpful. Figures 19.7 and 19.8 show that taking the log of square footage and number of units might prove helpful. It also shows what happens when taking the log of value.

> # plot ValuePerSqFt against SqFt

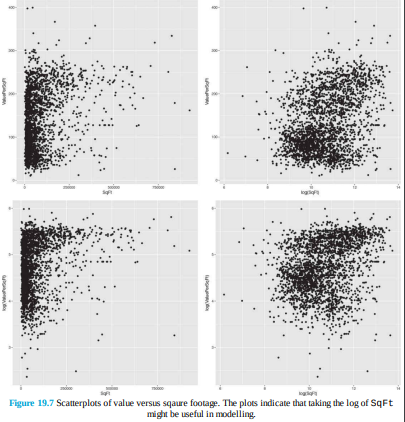
> ggplot(housing, aes(x=SqFt, y=ValuePerSqFt)) + geom\_point()

> ggplot(housing, aes(x=log(SqFt), y=ValuePerSqFt)) + geom\_point()

> ggplot(housing, aes(x=SqFt, y=log(ValuePerSqFt))) + geom\_point()

> ggplot(housing, aes(x=log(SqFt), y=log(ValuePerSqFt))) +

+ geom\_point()



> # plot ValuePerSqFt against Units

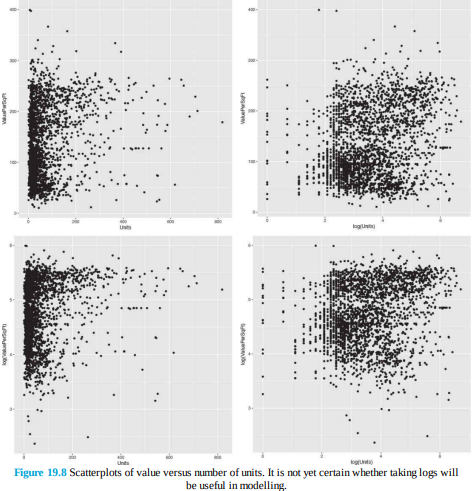
> ggplot(housing, aes(x=Units, y=ValuePerSqFt)) + geom\_point()

> ggplot(housing, aes(x=log(Units), y=ValuePerSqFt)) + geom\_point()

> ggplot(housing, aes(x=Units, y=log(ValuePerSqFt))) + geom\_point()

> ggplot(housing, aes(x=log(Units), y=log(ValuePerSqFt))) +

+ geom\_point()



Now that we have viewed our data a few different ways, it is time to start modelling. We already saw from Figure 19.4 that accounting for the different boroughs will be important and the various scatterplots indicated that Units and SqFt will be important as well. Fitting the model uses the formula interface in lm. Now that there are multiple predictors, we separate them on the right side of the formula using plus signs (+).

> house1 <- lm(ValuePerSqFt ~ Units + SqFt + Boro, data=housing)

> summary(house1)

Call:

lm(formula = ValuePerSqFt ~ Units + SqFt + Boro, data = housing)

Residuals:

Min 1Q Median 3Q Max

-168.458 -22.680 1.493 26.290 261.761

Coefficients:

Estimate Std. Error t value Pr(>|t|)

(Intercept) 4.430e+01 5.342e+00 8.293 < 2e-16 \*\*\*

Units -1.532e-01 2.421e-02 -6.330 2.88e-10 \*\*\*

SqFt 2.070e-04 2.129e-05 9.723 < 2e-16 \*\*\*

BoroBrooklyn 3.258e+01 5.561e+00 5.858 5.28e-09 \*\*\*

BoroManhattan 1.274e+02 5.459e+00 23.343 < 2e-16 \*\*\*

BoroQueens 3.011e+01 5.711e+00 5.272 1.46e-07 \*\*\*

BoroStaten Island -7.114e+00 1.001e+01 -0.711 0.477

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 43.2 on 2613 degrees of freedom

Multiple R-squared: 0.6034,Adjusted R-squared: 0.6025

F-statistic: 662.6 on 6 and 2613 DF, p-value: < 2.2e-16

The first thing to notice is that in some versions of R there is a message warning us that Boro was converted to a factor. This is because Boro was stored as a character, and for modelling purposes character data must be represented using indicator variables, which is how factors are treated inside modelling functions, as seen in Section 5.1.

The summary function prints out information about the model, including how the function was called, quantiles for the residuals, coefficient estimates, standard errors and p-values for each variable, and the degrees of freedom, p-value and F-statistic for the model. There is no coefficient for the Bronx because that is the baseline level of Boro, and all the other Boro coefficients are relative to that baseline. The coefficients represent the effect of the predictors on the response and the standard errors are the uncertainty in the estimation of the coefficients. The t value (t-statistic) and p-value for the coefficients are numerical measures of statistical significance, though these should be viewed with caution, as most modern data scientists do not like to look at the statistical significance of individual coefficients but rather judge the model as a whole as covered in Chapter 21. The model p-value and F-statistic are measures of its goodness of fit. The degrees of freedom for a regression are calculated as the number of observations minus the number of coefficients. In this example, there are nrow(housing) − length(coef(house1)) = 2613 degrees of freedom. A quick way to grab the coefficients from a model is to either use the coef function or get them from the model using the $ operator on the model object.

> house1$coefficients

(Intercept) Units SqFt

4.430325e+01 -1.532405e-01 2.069727e-04

BoroBrooklyn BoroManhattan BoroQueens

3.257554e+01 1.274259e+02 3.011000e+01

BoroStaten Island

-7.113688e+00

> coef(house1)

(Intercept) Units SqFt

4.430325e+01 -1.532405e-01 2.069727e-04

BoroBrooklyn BoroManhattan BoroQueens

3.257554e+01 1.274259e+02 3.011000e+01

BoroStaten Island

-7.113688e+00

> # works the same as coef

> coefficients(house1)

(Intercept) Units SqFt

4.430325e+01 -1.532405e-01 2.069727e-04

BoroBrooklyn BoroManhattan BoroQueens

3.257554e+01 1.274259e+02 3.011000e+01

BoroStaten Island

-7.113688e+00As a repeated theme, we prefer visualizations over tables of information, and a great way of visualizing regression results is a coefficient plot, like the one shown in Figure 19.2. Rather than build it from scratch, we use the convenient coefplot package that we wrote. Figure 19.9 shows the result, where each coefficient is plotted as a point with a thick line representing the one standard error confidence interval and a thin line representing the two standard error confidence interval. There is a vertical line indicating 0. In general, a good rule of thumb is that if the two standard error confidence interval does not contain 0, it is statistically significant.

> library(coefplot)

> coefplot(house1)

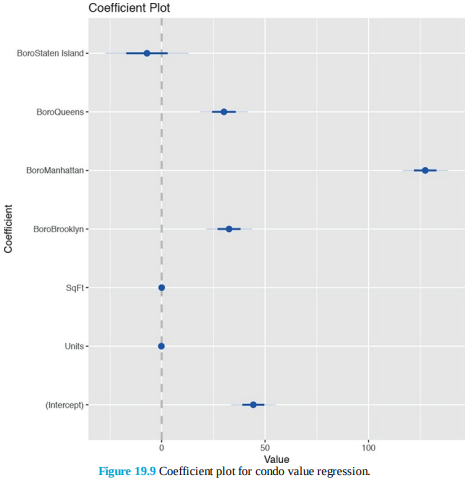


Figure 19.9 shows that, as expected, being located in Manhattan has the largest effect on value per square foot. Surprisingly, the number of units or square feet in a building has little effect on value. This is a model with purely additive terms. Interactions between variables can be equally powerful. To enter them in a formula, separate the desired variables with a \* instead of +. Doing so results in the individual variables plus the interaction term being included in the model. To include just the interaction term, and not the individual variables, use : instead. The results of interacting Units and SqFt are shown in Figure 19.10.

> house2 <- lm(ValuePerSqFt ~ Units \* SqFt + Boro, data=housing)

> house3 <- lm(ValuePerSqFt ~ Units : SqFt + Boro, data=housing)

> house2$coefficients

(Intercept) Units SqFt

4.093685e+01 -1.024579e-01 2.362293e-04

BoroBrooklyn BoroManhattan BoroQueens

3.394544e+01 1.272102e+02 3.040115e+01

BoroStaten Island Units:SqFt

-8.419682e+00 -1.809587e-07

> house3$coefficients

(Intercept) BoroBrooklyn BoroManhattan

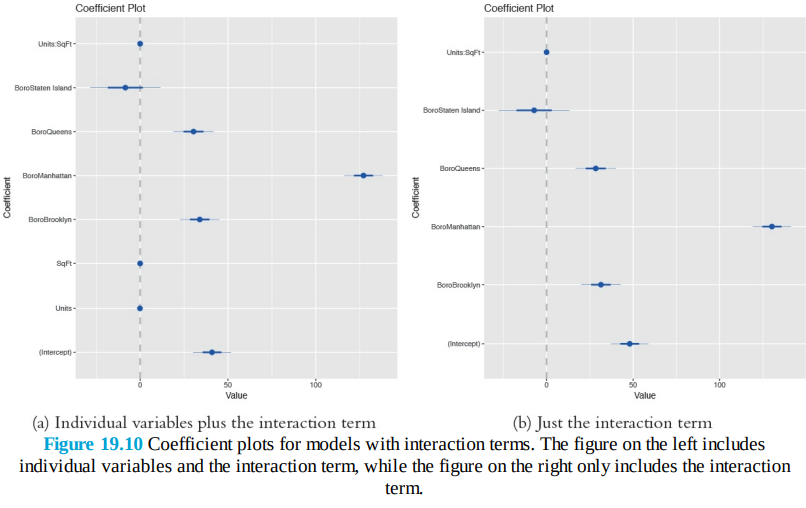
4.804972e+01 3.141208e+01 1.302084e+02

BoroQueens BoroStaten Island Units:SqFt

2.841669e+01 -7.199902e+00 1.088059e-07

> coefplot(house2)

> coefplot(house3)



If three variables all interact together, the resulting coefficients will be the three individual terms, three two-way interactions and one three-way interaction.

> house4 <- lm(ValuePerSqFt ~ SqFt\*Units\*Income, housing)

> house4$coefficients

(Intercept) SqFt Units

1.116433e+02 -1.694688e-03 7.142611e-03

Income SqFt:Units SqFt:Income

7.250830e-05 3.158094e-06 -5.129522e-11

Units:Income SqFt:Units:Income

-1.279236e-07 9.107312e-14

Interacting (from now on, unless otherwise specified, interacting will refer to the \* operator) a continuous variable like SqFt with a factor like Boro results in individual terms for the continuous variable and each non-baseline level of the factor plus an interaction term between the continuous variable and each non-baseline level of the factor. Interacting two (or more) factors yields terms for all the individual non-baseline levels in both factors and an interaction term for every combination of non-baseline levels of the factors.

> house5 <- lm(ValuePerSqFt ~ Class\*Boro, housing)

> house5$coefficients

(Intercept)

47.041481

ClassR4-CONDOMINIUM

4.023852

ClassR9-CONDOMINIUM

-2.838624

ClassRR-CONDOMINIUM

3.688519

BoroBrooklyn

27.627141

BoroManhattan

89.598397

BoroQueens

19.144780

BoroStaten Island

-9.203410

ClassR4-CONDOMINIUM:BoroBrooklyn

4.117977

ClassR9-CONDOMINIUM:BoroBrooklyn

2.660419

ClassRR-CONDOMINIUM:BoroBrooklyn

-25.607141

ClassR4-CONDOMINIUM:BoroManhattan

47.198900

ClassR9-CONDOMINIUM:BoroManhattan

33.479718

ClassRR-CONDOMINIUM:BoroManhattan

10.619231

ClassR4-CONDOMINIUM:BoroQueens

13.588293

ClassR9-CONDOMINIUM:BoroQueens

-9.830637

ClassRR-CONDOMINIUM:BoroQueens

34.675220

ClassR4-CONDOMINIUM:BoroStaten Island

NA

ClassR9-CONDOMINIUM:BoroStaten Island

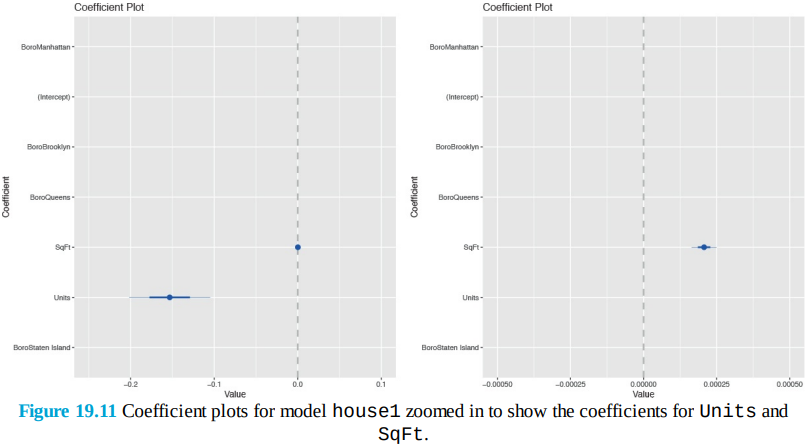
NA

ClassRR-CONDOMINIUM:BoroStaten Island

NANeither SqFt nor Units appear to be significant in any model when viewed in a coefficient plot. However, zooming in on the plot shows that the coefficients for Units and SqFt are non-zero as seen in Figure 19.11.

> coefplot(house1, sort='mag') + scale\_x\_continuous(limits=c(-.25, .1))

> coefplot(house1, sort='mag') + scale\_x\_continuous(limits=c(-.0005, .0005))



This is likely a scaling issue, as the indicator variables for Boro are on the scale of 0 and 1 while the range for Units is between 1 and 818 and SqFt is between 478 and 925,645. This can be resolved by standardizing, or scaling, the variables. This subtracts the mean and divides by the standard deviation. While the results of the model will mathematically be the same, the coefficients will have different values and different interpretations. Whereas before a coefficient was the change in the response corresponding to a one-unit increase in the predictor, the coefficientis now the change in the response corresponding to a one-standard-deviation increase in the predictor. Standardizing can be performed within the formula interface with the scale function.

> house1.b <- lm(ValuePerSqFt ~ scale(Units) + scale(SqFt) + Boro, + data=housing)

> coefplot(house1.b, sort='mag')

The coefficient plot in Figure 19.12 shows that for each change in the standard deviation of SqFt there is a change of 21.95 in ValuePerSqFt. We also see that Units has a negative impact. This implies that having fewer, but larger, units is beneficial to the value of a building.



Another good test is to include the ratio of Units and SqFt as a single variable. To simply divide one variable by another in a formula, the division must be wrapped in the I function.

> house6 <- lm(ValuePerSqFt ~ I(SqFt/Units) + Boro, housing)

> house6$coefficients

(Intercept) I(SqFt/Units) BoroBrooklyn

43.754838763 0.004017039 30.774343209

BoroManhattan BoroQueens BoroStaten Island

130.769502685 29.767922792 -6.134446417

The I function is used to preserve a mathematical relationship in a formula and prevent it from being interpreted according to formula rules. For instance, using (Units + SqFt)^2 in a formula is the same as using Units \* SqFt, whereas I(Units + SqFt)^2 will include the square of the sum of the two variables as a term in the formula.

> house7 <- lm(ValuePerSqFt ~ (Units + SqFt)^2, housing)

> house7$coefficients

(Intercept) Units SqFt Units:SqFt

1.070301e+02 -1.125194e-01 4.964623e-04 -5.159669e-07

> house8 <- lm(ValuePerSqFt ~ Units \* SqFt, housing)

> identical(house7$coefficients, house8$coefficients)

[1] TRUE

> house9 <- lm(ValuePerSqFt ~ I(Units + SqFt)^2, housing)

> house9$coefficients

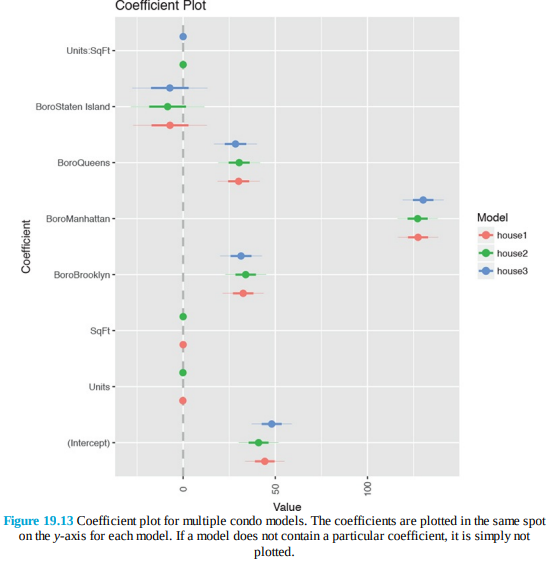
(Intercept) I(Units + SqFt)

1.147034e+02 2.107231e-04

We have fit numerous models from which we need to pick the “best” one. Model selection is discussed in Section 21.2. In the meantime, visualizing the coefficients from multiple models is a handy tool. Figure 19.13 shows a coefficient plot for models house1, house2 and house3.

> # also from the coefplot package

> multiplot(house1, house2, house3)



Regression is often used for prediction, which in R is enabled by the predict function. For this example, new data are available at <http://www.jaredlander.com/data/housingNew.csv>.

> housingNew <- read.table("http://www.jaredlander.com/data/housingNew.csv", + sep=",", header=TRUE, stringsAsFactors=FALSE)

Making the prediction can be as simple as calling predict, although caution must be used when dealing with factor predictors to ensure that they have the same levels as those used in building the model.

> # make prediction with new data and 95% confidence bounds

> housePredict <- predict(house1, newdata=housingNew, se.fit=TRUE,

+ interval="prediction", level=.95)

> # view predictions with upper and lower bounds based on standard errors

> head(housePredict$fit)

fit lwr upr

1 74.00645 -10.813887 158.8268

2 82.04988 -2.728506 166.8283

3 166.65975 81.808078 251.5114

4 169.00970 84.222648 253.7968

5 80.00129 -4.777303 164.7799

6 47.87795 -37.480170 133.2361

> # view the standard errors for the prediction

> head(housePredict$se.fit)

1 2 3 4 5 6

2.118509 1.624063 2.423006 1.737799 1.626923 5.318813

**Generalized Linear Models**

Not all data can be appropriately modeled with linear regression, because they are binomial (TRUE/FALSE), count data or some other form. To model these types of data, generalized linear models were developed. They are still modeled using a linear predictor, Xβ, but they are transformed using some link function. To the R user, fitting a generalized linear model requires barely any more effort than running a linear regression.

**Logistic Regression**

A very powerful and common model—especially in fields such as marketing and medicine—is logistic regression. The examples in this section will use the a subset of data from the 2010 American Community Survey (ACS) for New York State. 1 ACS data contain a lot of information, so we have made a subset of it with 22,745 rows and 18 columns available at <http://jaredlander.com/data/acs_ny.csv>.

> acs <- read.table("http://jaredlander.com/data/acs\_ny.csv", + sep="," , header=TRUE, stringsAsFactors=FALSE)

Logistic regression models are formulated as



where yi is the ith response and Xiβ is the linear predictor. The inverse logit function



transforms the continuous output from the linear predictor to fall between 0 and 1. This is the inverse of the link function. We now formulate a question that asks whether a household has an income greater than $150,000. To do this we need to create a new binary variable with TRUE for income above that mark and FALSE for income below.

> acs$Income <- with(acs, FamilyIncome >= 150000)

> library(ggplot2)

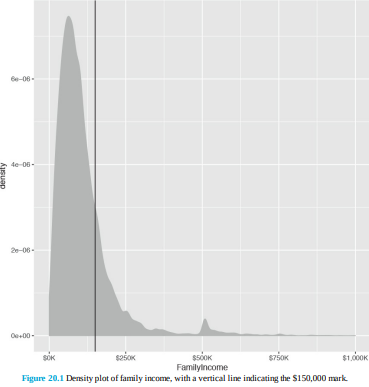
> library(useful)

> ggplot(acs, aes(x=FamilyIncome)) +

+ geom\_density(fill="grey", color="grey") +

+ geom\_vline(xintercept=150000) +

+ scale\_x\_continuous(label=multiple.dollar, limits=c(0, 1000000))



> head(acs)

Acres FamilyIncome FamilyType NumBedrooms NumChildren NumPeople

1 1-10 150 Married 4 1 3

2 1-10 180 Female Head 3 2 4

3 1-10 280 Female Head 4 0 2

4 1-10 330 Female Head 2 1 2

5 1-10 330 Male Head 3 1 2

6 1-10 480 Male Head 0 3 4

NumRooms NumUnits NumVehicles NumWorkers OwnRent

1 9 Single detached 1 0 Mortgage

2 6 Single detached 2 0 Rented

3 8 Single detached 3 1 Mortgage

4 4 Single detached 1 0 Rented

5 5 Single attached 1 0 Mortgage

6 1 Single detached 0 0 Rented

YearBuilt HouseCosts ElectricBill FoodStamp HeatingFuel Insurance

1 1950-1959 1800 90 No Gas 2500

2 Before 1939 850 90 No Oil 0

3 2000-2004 2600 260 No Oil 6600

4 1950-1959 1800 140 No Oil 0

5 Before 1939 860 150 No Gas 660

6 Before 1939 700 140 No Gas 0

Language Income

1 English FALSE

2 English FALSE

3 Other European FALSE

4 English FALSE

5 Spanish FALSE

6 English FALSE

Running a logistic regression is done very similarly to running a linear regression. It still uses the formula interface but the function is glm rather than lm (glm can actually fit linear regressions as well), and a few more options need to be set.

> income1 <- glm(Income ~ HouseCosts + NumWorkers + OwnRent +

+ NumBedrooms + FamilyType,

+ data=acs, family=binomial(link="logit"))

> summary(income1)

Call:

glm(formula = Income ~ HouseCosts + NumWorkers + OwnRent + NumBedrooms +

FamilyType, family = binomial(link = "logit"), data = acs)

Deviance Residuals:

Min 1Q Median 3Q Max

-2.8452 -0.6246 -0.4231 -0.1743 2.9503

Coefficients:

Estimate Std. Error z value Pr(>|z|)

(Intercept) -5.738e+00 1.185e-01 -48.421 <2e-16 \*\*\*

HouseCosts 7.398e-04 1.724e-05 42.908 <2e-16 \*\*\*

NumWorkers 5.611e-01 2.588e-02 21.684 <2e-16 \*\*\*

OwnRentOutright 1.772e+00 2.075e-01 8.541 <2e-16 \*\*\*

OwnRentRented -8.886e-01 1.002e-01 -8.872 <2e-16 \*\*\*

NumBedrooms 2.339e-01 1.683e-02 13.895 <2e-16 \*\*\*

FamilyTypeMale Head 3.336e-01 1.472e-01 2.266 0.0235 \*

FamilyTypeMarried 1.405e+00 8.704e-02 16.143 <2e-16 \*\*\*

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 22808 on 22744 degrees of freedom

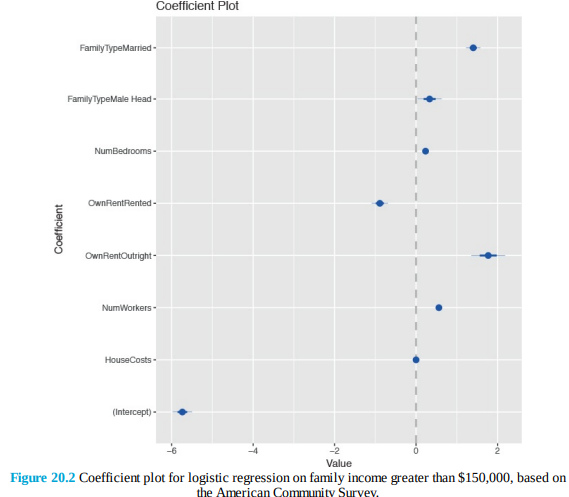
Residual deviance: 18073 on 22737 degrees of freedom

AIC: 18089

Number of Fisher Scoring iterations: 6

> library(coefplot)

> coefplot(income1)



The output from summary and coefplot for glm is similar to that of lm. There are coefficient estimates, standard errors, p-values—both overall and for the coefficients—and a measure of correctness, which in this case is the deviance and AIC. A general rule of thumb is that adding a variable (or a level of a factor) to a model should result in a drop in deviance of two; otherwise, the variable is not useful in the model. Interactions and all the other formula concepts work the same. Interpreting the coefficients from a logistic regression necessitates taking the inverse logit.

> invlogit <- function(x)

+ {

+ 1 / (1 + exp(-x))

+ }

> invlogit(income1$coefficients)

(Intercept) HouseCosts NumWorkers

0.003211572 0.500184950 0.636702036

OwnRentOutright OwnRentRented NumBedrooms

0.854753527 0.291408659 0.558200010

FamilyTypeMale Head FamilyTypeMarried

0.582624773 0.802983719**Poisson Regression**

Another popular member of the generalized linear models is Poisson regression, which, much like the Poisson distribution, is used for count data. Like all other generalized linear models, it is called using glm. To illustrate we continue using the ACS data with the number of children (NumChildren) as the response. The formulation for Poisson regression is



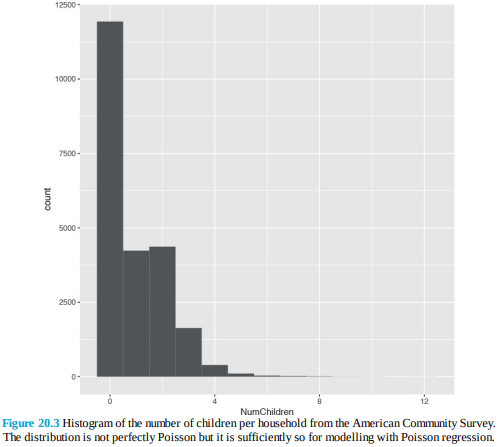
where yi is the ith response and



is the mean of the distribution for the ith observation. Before fitting a model, we look at the histogram of the number of children in each household.

> ggplot(acs, aes(x=NumChildren)) + geom\_histogram(binwidth=1)

While Figure 20.3 does not show data that have a perfect Poisson distribution, it is close enough to fit a good model. The coeficient plot is shown in Figure 20.4.



> children1 <- glm(NumChildren ~ FamilyIncome + FamilyType + OwnRent,

+ data=acs, family=poisson(link="log"))

> summary(children1)

Call:

glm(formula = NumChildren ~ FamilyIncome + FamilyType + OwnRent,

family = poisson(link = "log"), data = acs)

Deviance Residuals:

Min 1Q Median 3Q Max

-1.9950 -1.3235 -1.2045 0.9464 6.3781

Coefficients:

Estimate Std. Error z value Pr(>|z|)

(Intercept) -3.257e-01 2.103e-02 -15.491 < 2e-16 \*\*\*

FamilyIncome 5.420e-07 6.572e-08 8.247 < 2e-16 \*\*\*

FamilyTypeMale Head -6.298e-02 3.847e-02 -1.637 0.102

FamilyTypeMarried 1.440e-01 2.147e-02 6.707 1.98e-11 \*\*\*

OwnRentOutright -1.974e+00 2.292e-01 -8.611 < 2e-16 \*\*\*

OwnRentRented 4.086e-01 2.067e-02 19.773 < 2e-16 \*\*\*

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 35240 on 22744 degrees of freedom

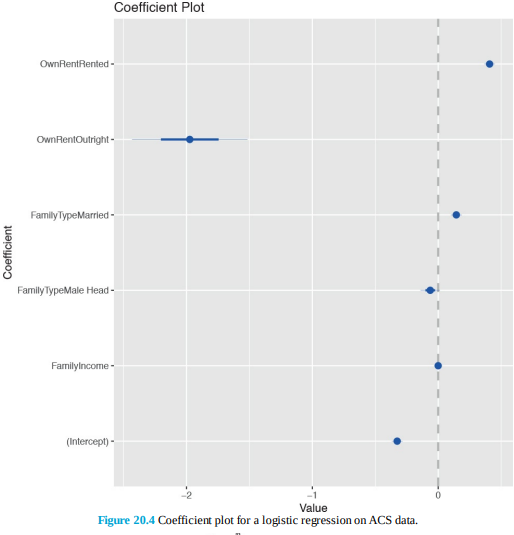
Residual deviance: 34643 on 22739 degrees of freedom

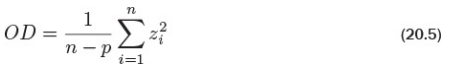
AIC: 61370

Number of Fisher Scoring iterations: 5

> coefplot(children1)

The output here is similar to that for logistic regression, and the same rule of thumb for deviance applies. A particular concern with Poisson regression is overdispersion, which means that the variability seen in the data is greater than is theorized by the Poisson distribution where the mean and variance are the same.





Overdispersion is defined as

Where



are the studentized residuals. Calculating overdispersion in R is as follows.

> # the standardized residuals

> z <- (acs$NumChildren - children1$fitted.values) /

+ sqrt(children1$fitted.values)

> # Overdispersion Factor

> sum(z^2) / children1$df.residual

[1] 1.469747

> # Overdispersion p-value

> pchisq(sum(z^2), children1$df.residual)

[1] 1

Generally an overdispersion ratio of 2 or greater indicates overdispersion. While this overdispersion ratio is less than 2, the p-value is 1, meaning that there is a statistically significant overdispersion. So we refit the model to account for the overdispersion using the quasipoisson family, which actually uses the negative binomial distribution.

> children2 <- glm(NumChildren ~ FamilyIncome + FamilyType + OwnRent, + data=acs, family=quasipoisson(link="log"))

> multiplot(children1, children2)

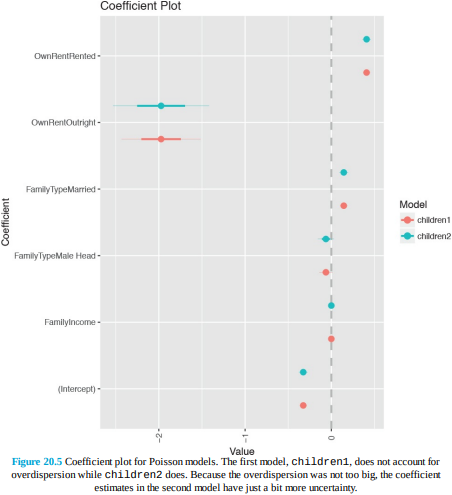


Figure 20.5 shows a coefficient plot for a model with that accounts for overdispersion and one that does not. Since the overdispersion was not very large, the second model adds just a little uncertainty to the coefficient estimates.

**Other Generalized Linear Models**

Other common generalized linear models supported by the glm function are gamma, inverse gaussian and quasibinomial. Different link functions can be supplied, such as the following: logit, probit, cauchit, log and cloglog for binomial; inverse, identity and log for gamma; log, identity and sqrt for Poisson; and 1/mu^2, inverse, identity and log for inverse gaussian. Multinomial regression, for classifying multiple categories, requires either running multiple logistic regressions (a tactic well supported in statistical literature) or using the polr function or the multinom function from the nnet package.

**Survival Analysis**

While not technically part of the family of generalized linear models, survival analysis is another important extension to regression. It has many applications, such as clinical medical trials, server failure times, number of accidents and time to death after a treatment or disease. Data used for survival analysis are different from most other data in that they are censored, meaning there is unknown information, typically about what happens to a subject after a given amount of time. For an example, we look at the bladder data from the survival package.

> library(survival)

> head(bladder)

id rx number size stop event enum

1 1 1 1 3 1 0 1

2 1 1 1 3 1 0 2

3 1 1 1 3 1 0 3

4 1 1 1 3 1 0 4

5 2 1 2 1 4 0 1

6 2 1 2 1 4 0 2

The columns of note are stop (when an event occurs or the patient leaves the study) and event (whether an event occurred at the time). Even if event is 0, we do not know if an event could have occurred later; this is why it is called censored. Making use of that structure requires the Surv function.

> # first look at a piece of the data

> bladder[100:105, ]

id rx number size stop event enum

100 25 1 2 1 12 1 4

101 26 1 1 3 12 1 1

102 26 1 1 3 15 1 2

103 26 1 1 3 24 1 3

104 26 1 1 3 31 0 4

105 27 1 1 2 32 0 1

> # now look at the response variable built by build.y

> survObject <- with(bladder[100:105, ], Surv(stop, event))

> # nicely printed form

> survObject

[1] 12 12 15 24 31+ 32+

> # see its matrix form

> survObject[, 1:2]

time status

[1,] 12 1

[2,] 12 1

[3,] 15 1

[4,] 24 1

[5,] 31 0

[6,] 32 0

This shows that for the first three rows where an event occurred, the time is known to be 12, whereas the bottom two rows had no event, so the time is censored because an event could have occurred afterward. Perhaps the most common modelling technique in survival analysis is using a Cox proportional hazards model, which in R is done with coxph. The model is fitted using the familiar formula interface supplied to coxph. The survfit function builds the survival curve that can then be plotted as shown in Figure 20.6. The survival curve shows the percentage of participants surviving at a given time. The summary is similar to other summaries but tailored to survival analysis.

> cox1 <- coxph(Surv(stop, event) ~ rx + number + size + enum,

+ data=bladder)

> summary(cox1)

Call:

coxph(formula = Surv(stop, event) ~ rx + number + size + enum,

data = bladder)

n= 340, number of events= 112

coef exp(coef) se(coef) z Pr(>|z|)

rx -0.59739 0.55024 0.20088 -2.974 0.00294 \*\*

number 0.21754 1.24301 0.04653 4.675 2.93e-06 \*\*\*

size -0.05677 0.94481 0.07091 -0.801 0.42333

enum -0.60385 0.54670 0.09401 -6.423 1.34e-10 \*\*\*

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

exp(coef) exp(-coef) lower .95 upper .95

rx 0.5502 1.8174 0.3712 0.8157

number 1.2430 0.8045 1.1347 1.3617

size 0.9448 1.0584 0.8222 1.0857

enum 0.5467 1.8291 0.4547 0.6573

Concordance= 0.753 (se = 0.029 )

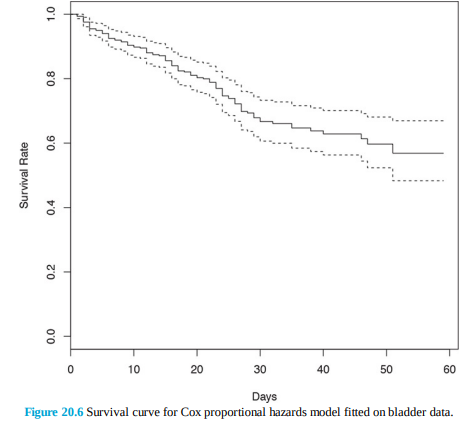
Rsquare= 0.179 (max possible= 0.971 )

Likelihood ratio test= 67.21 on 4 df, p=8.804e-14

Wald test = 64.73 on 4 df, p=2.932e-13

Score (logrank) test = 69.42 on 4 df, p=2.998e-14

> plot(survfit(cox1), xlab="Days", ylab="Survival Rate", conf.int=TRUE)



In this data, the rx variable indicates placebo versus treatment, which is a natural stratification of the patients. Passing rx to strata in the formula splits the data into two for analysis and will result in two survival curves like those in Figure 20.7.

> cox2 <- coxph(Surv(stop, event) ~ strata(rx) + number

+ + size + enum, data=bladder)

> summary(cox2)

Call:

coxph(formula = Surv(stop, event) ~ strata(rx) + number + size +

enum, data = bladder)

n= 340, number of events= 112

coef exp(coef) se(coef) z Pr(>|z|)

number 0.21371 1.23826 0.04648 4.598 4.27e-06 \*\*\*

size -0.05485 0.94662 0.07097 -0.773 0.44

enum -0.60695 0.54501 0.09408 -6.451 1.11e-10 \*\*\*

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

exp(coef) exp(-coef) lower .95 upper .95

number 1.2383 0.8076 1.1304 1.3564

size 0.9466 1.0564 0.8237 1.0879

enum 0.5450 1.8348 0.4532 0.6554

Concordance= 0.74 (se = 0.04 )

Rsquare= 0.166 (max possible= 0.954 )

Likelihood ratio test= 61.84 on 3 df, p=2.379e-13

Wald test = 60.04 on 3 df, p=5.751e-13

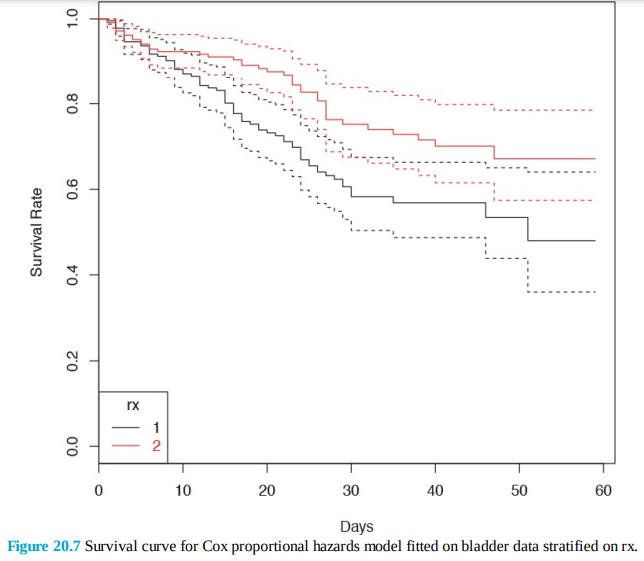
Score (logrank) test = 65.05 on 3 df, p=4.896e-14

> plot(survfit(cox2), xlab="Days", ylab="Survival Rate",

+ conf.int=TRUE, col=1:2)

> legend("bottomleft", legend=c(1, 2), lty=1, col=1:2,

+ text.col=1:2, title="rx")



As an aside, this was a relatively simple legend to produce but it took a lot more effort than it would with ggplot2. Testing the assumption of proportional hazards is done with cox.zph.

> cox.zph(cox1)

rho chisq p

rx 0.0299 0.0957 7.57e-01

number 0.0900 0.6945 4.05e-01

size -0.1383 2.3825 1.23e-01

enum 0.4934 27.2087 1.83e-07

GLOBAL NA 32.2101 1.73e-06

> cox.zph(cox2)

rho chisq p

number 0.0966 0.785 3.76e-01

size -0.1331 2.197 1.38e-01

enum 0.4972 27.237 1.80e-0

GLOBAL NA 32.101 4.98e-07

An Andersen-Gill analysis is similar to survival analysis, except it takes intervalized data and can handle multiple events, such as counting the number of emergency room visits as opposed to whether or not there is an emergency room visit. It is also performed using coxph, except an additional variable is passed to Surv, and the data must be clustered on an identification column (id) to keep track of multiple events. The corresponding survival curves are seen in Figure 20.8.

> head(bladder2)

id rx number size start stop event enum

1 1 1 1 3 0 1 0 1

2 2 1 2 1 0 4 0 1

3 3 1 1 1 0 7 0 1

4 4 1 5 1 0 10 0 1

5 5 1 4 1 0 6 1 1

6 5 1 4 1 6 10 0 2

> ag1 <- coxph(Surv(start, stop, event) ~ rx + number + size + enum +

+ cluster(id), data=bladder2)

> ag2 <- coxph(Surv(start, stop, event) ~ strata(rx) + number + size +

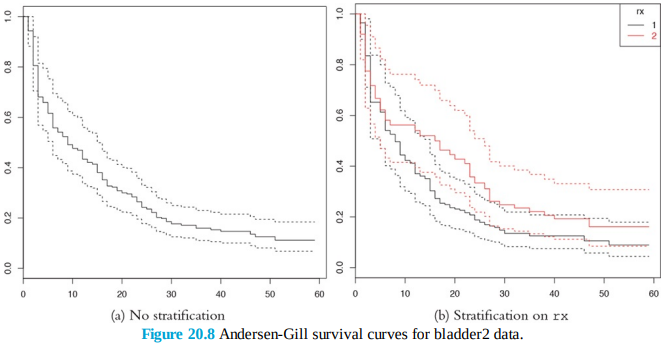
+ enum + cluster(id), data=bladder2)

> plot(survfit(ag1), conf.int=TRUE)

> plot(survfit(ag2), conf.int=TRUE, col=1:2)

> legend("topright", legend=c(1, 2), lty=1, col=1:2,

+ text.col=1:2, title="rx")



**Nonlinear Models**

A key tenet of linear models is a linear relationship, which is actually reflected in the coefficients, not the predictors. While this is a nice simplifying assumption, in reality nonlinearity often holds. Fortunately, modern computing makes fitting nonlinear models not much more difficult than fitting linear models. Typical implementations are nonlinear least squares, splines, decision trees and random forests and generalized additive models (GAMs).

**Splines**

A smoothing spline can be used to fit a smooth to data that exhibit nonlinear behavior and even make predictions on new data. A spline is a function f that is a linear combination of N functions (one for each unique data point) that are transformations of the variable x.



The goal is to find the function f that minimizes

where λ is the smoothing parameter. Small λs make for a rough smooth and large λs make for a smooth smooth. This is accomplished in R using smooth.spline. It returns a list of items where x holds the unique values of the data, y are the corresponding fitted values and df is the degrees of freedom used. We demonstrate with the diamonds data.

> data(diamonds)

> # fit with a few different degrees of freedom

> # the degrees of freedom must be greater than 1

> # but less than the number of unique x values in the data

> diaSpline1 <- smooth.spline(x=diamonds$carat, y=diamonds$price)

> diaSpline2 <- smooth.spline(x=diamonds$carat, y=diamonds$price,

+ df=2)

> diaSpline3 <- smooth.spline(x=diamonds$carat, y=diamonds$price,

+ df=10)

> diaSpline4 <- smooth.spline(x=diamonds$carat, y=diamonds$price,

+ df=20)

> diaSpline5 <- smooth.spline(x=diamonds$carat, y=diamonds$price,

+ df=50)

> diaSpline6 <- smooth.spline(x=diamonds$carat, y=diamonds$price,

+ df=100)

To plot these we extract the information from the objects, build a data.frame, and then add a new layer on top of the standard scatterplot of the diamonds data. Figure 23.3 shows this. Fewer degrees of freedom leads to straighter fits while higher degrees of freedom leads to more interpolating lines.

> get.spline.info <- function(object)

+ {

+ data.frame(x=object$x, y=object$y, df=object$df)

+ }

>

> library(plyr)

> # combine results into one data.frame

> splineDF <- ldply(list(diaSpline1, diaSpline2, diaSpline3, diaSpline4,

+ diaSpline5, diaSpline6), get.spline.info)

> head(splineDF)

x y df

1 0.20 361.9112 101.9053

2 0.21 397.1761 101.9053

3 0.22 437.9095 101.9053

4 0.23 479.9756 101.9053

5 0.24 517.0467 101.9053

6 0.25 542.2470 101.9053

> g <- ggplot(diamonds, aes(x=carat, y=price)) + geom\_point()

> g + geom\_line(data=splineDF,

+ aes(x=x, y=y, color=factor(round(df, 0)), group=df)) +

+ scale\_color\_discrete("Degrees of \nFreedom")



Making predictions on new data is done, as usual, with predict. Another type of spline is the basis spline, which creates new predictors based on transformations of the original predictors. The best basis spline is the natural cubic spline because it creates smooth transitions at interior breakpoints and forces linear behavior beyond the endpoints of the input data. A natural cubic spline with K breakpoints (knots) is made of K basis functions



Where



and ξ is the location of a knot and t+ denotes the positive part of t.

While the math may seem complicated, natural cubic splines are easily fitted using ns from the splines package. It takes a predictor variable and the number of new variables to return.

> library(splines)

> head(ns(diamonds$carat, df=1))

1

[1,] 0.00500073

[2,] 0.00166691

[3,] 0.00500073

[4,] 0.01500219

[5,] 0.01833601

[6,] 0.00666764

> head(ns(diamonds$carat, df=2))

1 2

[1,] 0.013777685 -0.007265289

[2,] 0.004593275 -0.002422504

[3,] 0.013777685 -0.007265289

[4,] 0.041275287 -0.021735857

[5,] 0.050408348 -0.026525299

[6,] 0.018367750 -0.009684459

> head(ns(diamonds$carat, df=3))

1 2 3

[1,] -0.03025012 0.06432178 -0.03404826

[2,] -0.01010308 0.02146773 -0.01136379

[3,] -0.03025012 0.06432178 -0.03404826

[4,] -0.08915435 0.19076693 -0.10098109

[5,] -0.10788271 0.23166685 -0.12263116

[6,] -0.04026453 0.08566738 -0.04534740

> head(ns(diamonds$carat, df=4))

1 2 3 4

[1,] 3.214286e-04 -0.04811737 0.10035562 -0.05223825

[2,] 1.190476e-05 -0.01611797 0.03361632 -0.01749835

[3,] 3.214286e-04 -0.04811737 0.10035562 -0.05223825

[4,] 8.678571e-03 -0.13796549 0.28774667 -0.14978118

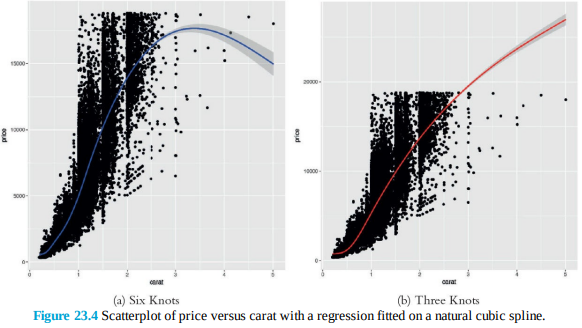
[5,] 1.584524e-02 -0.16428790 0.34264579 -0.17835789

[6,] 7.619048e-04 -0.06388053 0.13323194 -0.06935141

These new predictors can then be used in any model just like any other predictor. More knots means a more interpolating fit. Plotting the result of a natural cubic spline overlaid on data is easy with ggplot2. Figure 23.4a shows this for the diamonds data and six knots, and Figure 23.4b shows it with three knots. Notice that having six knots fits the data more smoothly.

> g + stat\_smooth(method="lm", formula=y ~ ns(x, 6), color="blue")

> g + stat\_smooth(method="lm", formula=y ~ ns(x, 3), color="red")



**Decision Trees**

A relatively modern technique for fitting nonlinear models is the decision tree. Decision trees work for both regression and classification by performing binary splits on the recursive predictors.

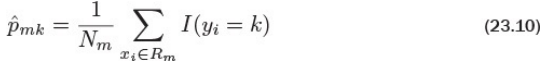
For regression trees, the predictors are partitioned into M regions R1 , R2 ,..., RM and the response y is



modeled as the average for a region with where



is the average y value for the region. The method for classification trees is similar. The predictors are partitioned into M regions and the proportion of each class in each of the regions, mk , is calculated as



where Nm is the number of items in region m and the summation counts the number of observations of class k in region m. Trees can be calculated with the rpart function in rpart. Like other modelling functions, it uses the formula interface but does not work with interactions.

> library(rpart)

> creditTree <- rpart(Credit ~ CreditAmount + Age +

+ CreditHistory + Employment, data=credit) Printing the object displays the tree in text

form.

> creditTree

n= 1000

node), split, n, loss, yval, (yprob)

\* denotes terminal node

1) root 1000 300 Good (0.7000000 0.3000000)

2) CreditHistory=Critical Account,Late Payment,Up To

Date 911 247 Good (0.7288694 0.2711306)

4) CreditAmount< 7760.5 846 211 Good (0.7505910 0.2494090) \*

5) CreditAmount>=7760.5 65 29 Bad (0.4461538 0.5538462)

10) Age>=29.5 40 17 Good (0.5750000 0.4250000)

20) Age< 38.5 19 4 Good (0.7894737 0.2105263) \*

21) Age>=38.5 21 8 Bad (0.3809524 0.6190476) \*

11) Age< 29.5 25 6 Bad (0.2400000 0.7600000) \*

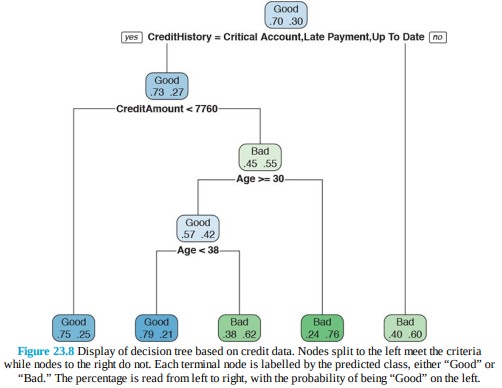
3) CreditHistory=All Paid,All Paid This Bank 89 36

Bad (0.4044944 0.5955056) \*The printed tree has one line per node. The first node is the root for all the data and shows that there are 1,000 observations of which 300 are considered “Bad.” The next level of indentation is the first split, which is on CreditHistory. One direction—where CreditHistory equals either “Critical Account,” “Late Payment” or “Up To Date”—contains 911 observations, of which 247 are considered “Bad.” This has a 73% probability of having good credit. The other direction—where CreditHistory equals either “All Paid” or “All Paid This Bank”—has a 60% probability of having bad credit. The next level of indentation represents the next split. Continuing to read the results this way could be laborious; plotting will be easier. Figure 23.8 shows the splits. Nodes split to the left meet the criteria while nodes to the right do not. Each terminal node is labelled by the predicted class, either “Good” or “Bad.” The percentage is read from left to right, with the probability of being “Good” on the left.

> library(rpart.plot)

> rpart.plot(creditTree, extra=4)

While trees are easy to interpret and fit data nicely, they tend to be unstable with high variance due to overfitting. A slight change in the training data can cause a significant difference in the model.



**Random Forests**

Random forests are a type of ensemble method. An ensemble method is a process in which numerous models are fitted, and the results are combined for stronger predictions. While this provides great predictions, inference and explainability are often limited. Random forests are composed of a number of decision trees where the included predictors and observations are chosen at random. The name comes from randomly building trees to make a forest. In the case of the credit data we will use CreditHistory, Purpose, Employment, Duration, Age and CreditAmount. Some trees will have just CreditHistory and Employment, another will have Purpose, Employment and Age, while another will have CreditHistory, Purpose, Employment and Age. All of these different trees cover all the bases and make for a random forest that should have strong predictive power. Fitting the random forest is done with randomForest from the randomForest package. Normally, randomForest can be used with a formula, but categorical variables must be stored as factors. To avoid having to convert the variables, we provide individual predictor and response matrices. This requirement for factor variables is due to the author’s (Andy Liaw) frustration with the formula interface. He even warned users “I will take the formula interface away.” We have seen, for this function, that using matrices is generally faster than formulas.

> library(randomForest)

> creditFormula <- Credit ~ CreditHistory + Purpose + Employment +

+ Duration + Age + CreditAmount - 1

> # we use all levels of the categorical variables since it is a tree

> creditX <- build.x(creditFormula, data=credit, contrasts=FALSE)

> creditY <- build.y(creditFormula, data=credit)

>

> # fit the random forest

> creditForest <- randomForest(x=creditX, y=creditY)

>

> creditForest

Call:

randomForest(x = creditX, y = creditY)

Type of random forest: classification

Number of trees: 500

No. of variables tried at each split: 4

OOB estimate of error rate: 27.4%

Confusion matrix:

Good Bad class.error

Good 644 56 0.0800000

Bad 218 82 0.7266667

The displayed information shows that 500 trees were built and four variables were assessed at each split; the confusion matrix shows that this is not exactly the best fit, and that there is room for improvement. Due to the similarity between boosted trees and random forests, it is possible to use xgboost to build a random forest by tweaking a few arguments. We fit 1000 trees in parallel (num\_parallel\_tree=1000) and set the row (subsample=0.5) and column (colsample\_bytree=0.5) sampling to be done at random.

> # build the response matrix

> creditY2 <- as.integer(relevel(creditY, ref='Bad')) - 1

> # Fit the random forest

> boostedForest <- xgboost(data=creditX, label=creditY2, max\_depth=4,

+ num\_parallel\_tree=1000,

+ subsample=0.5, colsample\_bytree=0.5,

+ nrounds=3, objective="binary:logistic")

[1] train-error:0.282000

[2] train-error:0.283000

[3] train-error:0.279000

In this case the error rate for the boosted-derived random forest is about the same as for the one fit by randomForest. Increasing the nrounds argument will improve the error rate, although it could also lead to overfitting. A nice benefit of using xgboost is that we can visualize the resulting random forest as a single tree as shown in Figure 23.11.

> xgb.plot.multi.trees(boostedForest, feature\_names=colnames(creditX))

