Nagiza F. Samatova, William Hendrix, Arpan Chakraborty, John Jenkins, Kanchana Padmanabhan, Srinath Ravindran

Practical Graph Mining with R

List of Figures

List of Tables

Contents

1

An Introduction to R

Neil Shah

North Carolina State University

1.1 What is R?

R is a language for statistical computing and graphics. The R environment provides a multitude of built-in functions in the "base" package, many of which are commonly required for elementary data analysis (e.g., linear modeling, graph plotting, basic statistics). However, the beauty of R lies in its almost infinite expandability and versatility. Approximately 2,500 (and counting) packages have been developed for R by the active R community. These packages serve to augment R's natural capabilities in data analysis and often focus on developments in various scientific fields, as well as techniques used in highly specialized data analyses.

1.1.1 A Brief History

R is a GNU implementation of the *S* language, a popular statistical language developed at Bell Laboratories by John Chambers and his colleagues. While R is not *S* itself, much *S* code can run in R without any change. R itself was developed as an open-source alternative to the *S* environment. It was developed by Robert Gentlement and Ross Ihaka. The name R is partially a reference to the main authors of R as well as a play on the name *S*.

1.1.2 Common Uses of R

R is a uniting tool between all types of application scientists who deal with data management and analysis. It is used commonly in disciplines such as business modeling and projection, stock performance, biology, biochemistry, high-performance computing, parallel computing, statistics, and many more. Within each of these disciplines, R can also be applied to many different domain-specific tasks. Because of R's ease-of-use and simplicity, it has become a staple tool for many scientists to interface their work with software and augment possibilities and speed-up tasks. The R useR! conference is held each year to unite application scientists and industry representatives who work with R as a medium to enable them to perform tasks integral to their work and research.

If you have R installed on your system already, you can continue with this chapter as it is. However, if you have not yet installed R you should skip to the last section of this chapter, which covers installation routines, before continuing.

1.2 What Can R Do?

As a flexible language, R inherently provides many statistical functions involving inference tests, clustering, and curve fitting. Additionally, R can be put to use in many disciplines with the employment of various packages and libraries contributed by the R community. R is also equipped with excellent graphing and plotting methods to generate publication-quality images.

1.2.1 Data Manipulation

R provides many methods to work with data in a variety of ways. As a scripting language, R also has support for reading and writing data from files, creating data sequences, performing operations on data, summarizing data, and also casting data to other types. In the next few sections, we will address these various operations.

1.2.1.1 Reading from Files

The most common R function for reading data from files is the `read.table` command. This command reads data of a matrix or table type and reads the data (including any headers) into R as a data-frame type. Later in this chapter, we will see how to cast this data-frame into other, more commonly used types (such as matrices). The syntax of the `read.table` function is as follows:

```
1 > students = read.table(‘‘students.txt’’,header=TRUE)
2 > students
3   Name Age ID
4 1  Alex  16  1
5 2 Jacob  15  2
6 3   Sue  17  3
7 > students[1]
8   Name
9 1  Alex
10 2 Jacob
11 3   Sue
12 > students[‘‘Name’’]
13   Name
14 1  Alex
15 2 Jacob
16 3   Sue
```

Here we have a tab-delimited file "students.txt" containing the records of several high-school students. We use the `read.table` function to read the records from the file and store them as a data-frame type into the variable *students*. We use the "header=TRUE" (optional) argument to specify that we wish to keep the headings of the columns upon importing the data into R. We can refer to the imported data set by typing the name of the variable that references the data at the R prompt. As you can see, we can refer to a column in R by either referencing its index (here we use index 1 as the first column) or by its name (in this case, the name of the column containing student names is "Name").

The `scan` function can also be used to read data from files, but is far less common. However, you can read about `scan` in the R help pages, by typing in R's prompt:

```
1 > help(scan)
```

1.2.1.2 Writing to Files

It is useful to occasionally save your R session's output to a file. The best way to do this is to access the "Save to File" functionality from R's "File" menu. You may also look into the `sink` and `capture.output` functions to automate these processes, but these are used uncommonly.

1.2.1.3 Creating Data

Generating sequences of data is considerably straightforward in R. The most frequently used data types in R are vectors, matrices, lists, and data frames. One important fact to note is that in R, indexing of elements is done from array index 1—to those of you who have studied other programming or scripting languages, this will take some getting used to. However, the first element of any data type in R is accessed as `element[1]`, and not `element[0]`. This information will help you understand the examples presented in this section. We will cover the most frequently used data types in R with some depth in the next paragraphs.

Vectors are used in R as the most basic of data types, often used to store single-dimensional data. They can be of various sub-types, but most frequently are of type "numeric" in order to store integers or decimal numbers. Vectors can also store string data ("character" type vectors). The following code shows some rudimentary numeric and character data vector generation techniques:

```
1 > a = 1:10
2 > b = c(1:10)
3 > c = c(1,4,2,11,19)
```

```
> d = seq(0,10,by=2)
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> b
 [1]  1  2  3  4  5  6  7  8  9 10
> c
[1]  1  4  2 11 19
> d
[1]  0  2  4  6  8 10
> e = 10:1
> f = c(10:1)
> g = seq(10,0,by=-2)
> e
 [1] 10  9  8  7  6  5  4  3  2  1
> f
 [1] 10  9  8  7  6  5  4  3  2  1
> g
[1] 10  8  6  4  2  0
> h = ‘‘hello’’
> h
[1] ‘‘hello’’
```

The first operator used here creates a vector consisting of the numbers from 1 to 10. The colon operator denotes a range, or series of values. The `c` function creates an identical vector, but is a cleaner and more acceptable way of performing the same operation. The `c` function also allows us to specify whichever numbers we want to put in a vector. In most cases, you should use `c` to create vectors or sequences of values. The `seq` function generates a sequence (and returns a value of type vector). In this case, we generate a sequence of numbers from 0 to 10 in steps of 2. In the next three examples, we can see that by specifying a range from a larger number to a smaller number, we can also create sequences in a decreasing fashion. We can also use negative values as step increments in the `seq` function. The last example involves creation of a character vector, or a string.

In contrast with vectors, matrices are frequently used to hold multi-dimensional data. The following code shows how to create matrices using R:

```
> m1 = matrix(c(1:10),nrow=2,ncol=5,byrow=FALSE)
> m1
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

```
> m2 = matrix(c(1:10),nrow=2,ncol=5,byrow=TRUE)
> m2
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10
```

Matrices are created using the `matrix` function. The arguments most commonly passed include the vector of values to be used, the number of rows, and the number of columns. The "byrow" argument takes a boolean value to determine whether the matrix should be created in a row-by-row or column-by-column fashion. Additionally, the functions `nrow` and `ncol` can be passed as an argument to return a single length vector respectively containing the number of rows or columns the matrix has. Matrix multiplication can be done using the `%*%` operator; for example, `A %*% B` will return the matrix multiplication of two compatible matrices, A and B.

Lists in R are collections of other objects, known as *components*. These components need not be of the same type, so a single list can contain a character vector (string), numeric vector, and a matrix all at once as three separate components. The following code shows a simple example of how to create a list object:

```
> list1 = list(name=''Jacob'',age=22, luckynumbers=c(3,7,9))
> list1
$name
[1] "Jacob"

$age
[1] 22

$luckynumbers
[1] 3 7 9
```

Lists are created using the `list` function. The arguments passed to the function are the components that are to be added to the list. Components can be accessed from the list by name or by numerical index; for example, the *age* component can be accessed as either `list1$age` or as `list1[2]`. Unlike vectors, lists have two indexing operators: '[]' and '[[]]'. The former operator is the general indexing operator—using this operator with a list will result in selecting a sublist from the parent list. For example, the code `list1[2]` will return a list with a single component called *age*. The latter operator, '[[]]' is a list-specific indexing operator that allows specific selection of list components outside of list context. For example, the code `list1[[2]]` will simply return a numeric vector with the value of *age*.

Data frames are essentially lists with the class *data.frame*. The components of data frames must be vectors, matrices, lists, or other data frames. Each component added to the data frame must be "compatible" in having an equal number of rows, or elements (in case of a vector). The data frame will have as many individual elements as the sum of all its individual components. Since data frames are still of type list, they have the characteristic of being able to hold multiple data types together. A simple example of creating a data frame is given below.

```
> a = c(1,2,3)
> b = matrix(c(4:15),nrow=3,ncol=4)
> p = c(``R'',``is'',``awesome'')
> df = data.frame(a,b,p)
> a
[1] 1 2 3
> b
     [,1] [,2] [,3] [,4]
[1,]    4    7   10   13
[2,]    5    8   11   14
[3,]    6    9   12   15
> p
[1] ``R''       ``is''      ``awesome''
> df
  a X1 X2 X3 X4       p
1 1  4  7 10 13       R
2 2  5  8 11 14      is
3 3  6  9 12 15 awesome
```

Data frames are created using the `data.frame` function. The arguments passed to the function, like a list, are the components that are to be added to the data frame. A data frame for purposes of accessing data can be considered a matrix with columns of different attributes. As you can see in the above example, it is generally displayed in a similar fashion to a matrix. Data frames have a similar indexing scheme to lists. The code `df[1]` returns a data frame with data from vector *a*. However, the code `df[[1]]` returns the first component of the data frame (vector *a*) outside of the data frame context.

1.2.1.4 Operating on Data

R provides very convenient methods to modify existing data. Common operations on vectors and matrices are all built into R. These operations are fairly intuitive, some of which are shown in the following code segment:

```
> a = c(1:10)
> a
 [1]  1  2  3  4  5  6  7  8  9 10
> a*2
 [1]  2  4  6  8 10 12 14 16 18 20
> a/2
 [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> a^3
 [1]    1    8   27   64  125  216  343  512  729 1000
> a/a
 [1] 1 1 1 1 1 1 1 1 1 1
> sum(a)
[1] 55
> b = matrix(c(1:4),nrow=2,ncol=2,byrow=TRUE)
> b
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> b*2
     [,1] [,2]
[1,]    2    4
[2,]    6    8
> b/2
     [,1] [,2]
[1,]  0.5    1
[2,]  1.5    2
> b/b
     [,1] [,2]
[1,]    1    1
[2,]    1    1
> b %*% b
     [,1] [,2]
[1,]    7   10
[2,]   15   22
```

In the above code block, we can see that R allows all sorts of operations on vectors, including scalar multiplication and division, as well as vector multiplication. In regards to matrices, scalar multiplication and division are also allowed, in addition to the useful matrix multiplication operation (denoted by operator "%*%").

Additionally, R provides very convenient logical row, column and subset selections from data objects. This capacity allows easy access to "slices" or samples of large data sets. The indexing operator '[]' can be used for selecting not only individual elements from vectors, lists, ma-

trices, or data frames but also a wealth of tasks. The following examples will highlight some of these tasks and their means of use. While the examples here deal with numeric vectors and matrices, the uses of the indexing operator can be extended to other data types effectively as well.

```
> a = c(1,2,3,4,5)
> b = matrix(c(1:9),nrow=3,ncol=3)
> a
[1] 1 2 3 4 5
> b
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> a[3]
[1] 3
> a[-3]
[1] 1 2 4 5
> a[2:4]
[1] 2 3 4
> b[2,]
[1] 2 5 8
> b[,2]
[1] 4 5 6
> b[2,2]
[1] 5
```

In the first example shown, `a[3]` simply showcases what you have learned thus far about the indexing operator—it can be used to select individual elements. In this example, the third element in vector a is returned to the user. The next example is a bit more intriguing—when a minus is used in front of an index in the context of '[]', the element is *excluded* from the selection. The result of `a[-3]` is all the elements of vector a, with the exception of the element at index 3. The next example outlines the concept of range selection—R does not require you select only a single index, but allows you to select a series of indices just as easily. In this example, the 2^{nd} through 4^{th} elements of vector a are selected, as denoted by the range `2:4`.

The next few examples are applied to matrices, in order to outline the use of the indexing operator on multidimensional data objects. The first of the matrix examples shows that selections can be done row-wise; in this example specifically, the second row-vector of matrix b is selected. The next example is similar, but done column-wise; the second column-vector is selected. The last example shows how to access indi-

vidual elements from a matrix by specifying the row and column indices (first and second, respectively).

The last important means to operate on data which we will discuss in this introductory chapter is the use of control statements in R. Control statements primarily encompass conditional execution (the `if` expressions) and repetitive execution (`for` and `while` loops). Additionally, we will broach the topic of using the `apply` family of functions to emulate repetitive execution.

Conditional execution is essential in writing many useful Rscripts. Consider the (commonly encountered) case in which one desires his or her code to perform different tasks depending on the value of a variable. `if` expressions aim to address this problem of conditional execution. `if` statements have the form `if (cond) snippet1 else snippet2`, where the condition `cond` evaluates to a logical value. In the case that `cond` evaluates to be *TRUE*, then `snippet1` is executed; conversely, in the case that `cond` is *FALSE*, then `snippet2` is executed. `snippet2` can also be a group of statements, if surrounded by the left and right curly bracket symbols ({ and }) and separated with semicolons. A `condition` can be created using any of the logical operators introduced later in this chapter. The following example highlights the use of the `if` conditional.

```
1 > a = 2
2 > if(a < 5) x=1 else x=2
3 > x
4 [1] 1
5 > a = 7
6 > if(a < 5) x=1 else x=2
7 > x
8 [1] 2
```

Here, we run the same conditional `if` statement two times with the same conditions, except the difference in the value of the input. This difference causes the conditional to evaluate to *TRUE* in the first case and *FALSE* in the second. As a result, the value of the variable x differs after the execution of both conditional statements.

Repetitive execution is typically done using the `for` and `while` statements. Both of these statements simply repeat the nested code as long as their conditions hold true. The `for` loop is constructed like so: `for(name in expr1) expr2`, where name is a temporary naming convention for each item produced by the expression `expr1`. `expr2` is the code that is run for each of the items. Two examples of how to use the `for` loop are given below.

```
1 > for(i in c(1:10))
2 + print(i)
```

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
> x = list(c(1:3),c(4:6),c(7:9))
> x
[[1]]
[1] 1 2 3

[[2]]
[1] 4 5 6

[[3]]
[1] 7 8 9

> for(i in x)
+ print(mean(i))
[1] 2
[1] 5
[1] 8
```

The first example simply iterates over a numeric vector spanning from 1 to 10. For each of the items in this vector (numbers 1 through 10), a `print` command is issued to display the item. The numbers from 1 to 10 are printed as a result. The second example shows a slightly more complicated `for` loop which iterates over each item from a list of vectors and prints its mean. The result is the sequential output of the average values from the first, second and third vectors that composed the list object *x*.

The `while` loop is similar to the `for` loop. A `while` statement is constructed as `while(cond) expr1`, where the expression `expr1` executes as long as the condition `cond` holds true. An example of using the `while` statement is given below.

```
> a = 0
> while(a < 10)
+ + a = a+1+ print(a)+
[1] 1
```

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

In this example, the `while` construct is executed as long as the condition that $a < 10$ holds true. For as long as the condition is true, the value of a is incremented and printed. As a result, the values of a in each iteration of the `while` loop are printed sequentially.

In many cases, frequent loop usage can cause code to appear messy or perform poorly in terms of speed. The R `apply` family of functions serves to address these concerns by simplifying (and in many cases improving performance of) for repetitive execution.

While there are many `apply` functions, the most frequently used are `apply` and `lapply` (as well as its close neighbor, `sapply`) . These functions allow one to perform repetitive execution of a function over elements of a matrix and a list respectively. These functions (along with the rest of the family) each have varying arguments, so it is suggested that the user run the `help` command on each variant indivdiually to read the detailed specifics on how to use each. Examples concerning the use of `apply` and `lapply` are given below.

```
> x = matrix(c(1:16),nrow=4,ncol=4,byrow=TRUE)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
> apply(x,1,max)
[1]  4  8 12 16
> apply(x,2,max)
[1] 13 14 15 16
> y = list(c(1:4),c(5:8),c(9:12),c(13:16))
> y
[[1]]
[1] 1 2 3 4

[[2]]
```

```
[1] 5 6 7 8

[[3]]
[1]  9 10 11 12

[[4]]
[1] 13 14 15 16

> lapply(y,max)
[[1]]
[1] 4

[[2]]
[1] 8

[[3]]
[1] 12

[[4]]
[1] 16
```

The first example shows two simple uses of the `apply` function. The `apply` function is used to apply a function (in this case, the `max` function) to the rows or columns of a matrix. The first arguments passed are input matrices. The second arguments passed to the `apply` functions are identifiers; 1 signifies that the function should be applied to each row of the matrix, whereas 2 signifies that the function should be applied to each column. Lastly, the third arguments are the functions to be executed on each of the rows or columns. The results of the two `apply` calls are two vectors containing the results of applying the `max` function to each of the "items" (rows and columns) from the matrix x.

The second example shows the use of the `lapply` function. The `lapply` function is used to apply a function (in this case, the `max` function) to each element of a list. The first argument passed to `lapply` is the input list. The second argument is the function to be executed on each element of the list. The result of the `lapply` call is a list containing the results of applying the `max` function to each of the "items" (in this case, vectors) from the list y. `sapply` does the exact same thing as `lapply`, but returns a vector containing the results instead of a list.

1.2.1.5 Summarizing Data

Summarizing a series of values is a common operation in a variety of disciplines. Computing basic statistics is useful in working with any large

data set. R provides summary functions for common statistical operations. For the slightly more advanced operations concerning standard deviation, variance, and more, we may import the stats package that comes preinstalled with the conventional R installation. The following functions show the proper usage of these summary functions. Additionally, note that we use the `library` function to load an installed library.

```
> data = c(3,17,4,13,91,55,10)
> min(data)
[1] 3
> max(data)
[1] 91
> mean(data)
[1] 27.57143
> median(data)
[1] 13
> summary(data)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   3.00    7.00   13.00   27.57   36.00   91.00
> library(stats)
> sd(data)
[1] 33.08503
> var(data)
[1] 1094.619
```

1.2.1.6 Typecasting Data

R provides an entire suite of functions that serve to effectively (and usually very accurately) cast data to other (compatible types). The functions create a new variable that is a forced version of the input variable. The following R code shows some of these functions:

```
> b = matrix(c(1:4),nrow=2,ncol=2,byrow=TRUE)
> b
     [,1] [,2]
[1,]    1    2
[2,]    3    4
> as.data.frame(b)
  V1 V2
1  1  2
2  3  4
> as.matrix(b)
     [,1] [,2]
```

```
[1,]    1    2
[2,]    3    4
> as.vector(b)
[1] 1 3 2 4
> as.list(b)
[[1]]
[1] 1

[[2]]
[1] 3

[[3]]
[1] 2

[[4]]
[1] 4
```

1.2.2 Calculation

R makes for a great calculator. It provides an easy pathway to simple tasks such as scalar, vector, and matrix addition and subtraction, multiplication, division, exponentiation, cumulative sums, and much more. Upon learning to use R, it becomes an exceptionally handy and easy-to-use tool for crunching numbers. For more advanced operations such as matrix multiplication, finding matrix inverses, eigenvalues and eigenvectors, there are a multitude of functions that are more than willing to do work for you at only a fraction of the time it would take for you to solve them yourself!

1.2.3 Graphical Display

R has a powerful ability to graph and plot data and generate publication-quality images. The most common graphical function you will likely use in R is the `plot` function, which simply plots a vector of y values against a vector of x values. The following is an example of the conventional use of this function:

```
> x = c(1,3,4,5,8)
> y = c(2,4,7,3,9)
> plot(x,y,type=‘‘l’’,col=‘‘black’’,
+ xlab=‘‘Values from X’’,ylab=‘‘Values from Y’’,
+ main=‘‘Plot of Y vs. X’’,font.main=2,font.lab=1,pch=19)
```

The plot command accepts several parameters, the most basic of which are the two value vectors that are to be plotted (x and y, in this case). Additionally, we have specified type="l" for a graph using lines. It is common practice to specify a graph color, and also specify the x-label, y-label, and main title. There are many more options to customize plotting. Type `help(plot)` at the R prompt to learn about these.

1.3 R Packages

In this textbook, we will be using a variety of packages that allow us to more easily perform certain tasks. It is critical to understand how to download and import packages in R. In the following sections, you will learn how to do both of these things.

1.3.1 Downloading a Package

You can download and install packages using both graphical and text-based interfaces. The means by which you can do so are explained in the next few sections.

1.3.1.1 Graphical User Interfaces

If you are using the R GUI in a Windows or OSX environment, the easiest way to download a package is to click on the "Packages" menu in R. From this menu, you should select the "Install package(s)..." item. You will be presented with a menu asking you to select the location of the mirror closest to you to download the package from. Upon selecting a mirror, you will be shown a list of all packages available for download on CRAN. By simply selecting and double-clicking any of these, the package will be downloaded to your hard drive. Additionally, package dependencies will also be downloaded and installed for you. For this reason, downloading packages using the inbuilt functionalities is very user-friendly. Downloaded packages are automatically unpacked and installed for you.

Another way to download packages involves downloading the R package from a browser-accessible webpage (most commonly a CRAN mirror). If you download a package to your local disk, you can select the "Install package(s) from local zip files...", and browse to the location of the downloaded zip file. Upon selecting this file, the package will be installed for you.

1.3.1.2 Command-Line Interfaces

If you are using R in a command-line OSX or *NIX environment, then you can install packages using CLI accessible methods. To install packages from local .tar.gz files, you can use the command $R_HOME/bin/R CMD INSTALL *filename* from within your terminal environment (not within the R environment). The package will be installed to the default R directory for packages.

If you wish to install a package directly from a CRAN mirror, you can start the R environment with $R_HOME/bin/R, and from within the R CLI, enter the following:

```
1 > install.packages(``packagename'');
```

1.3.2 Importing a Package

To import a package into your R session, you can simply enter the following:

```
1 > library(packagename);
```

1.3.3 Getting Help

To learn more about a package (version, title, author, etc.), you can use:

```
1 > library(help = packagename);
```

To find out which functions are inside a package, you can use:

```
1 > ls(package:packagename);
```

Lastly, to find information on how to use a function (which arguments to provide to the function, what type the function returns, etc.), you can use:

```
1 > help(functionname);
```

As a helpful note, press the "q" key to quit out of the help screens and get back to R.

1.4 Why use R?

You may be wondering whether R is indeed the best tool for this type of work. Rather, why not *SAS* or *S+* or even some other scripting environment with packages, such at *Matlab*? R has many benefits which make it the ideal candidate for our use. We will highlight these benefits briefly in the following sections.

1.4.1 Expansive Capabilities

The CRAN (Comprehensive R Archive Network) package repository currently features 2,482 packages for download. Almost all of these have functional versions for Windows, *NIX, and OSX. You may do some browsing on the R website *http://www.r-project.org/* to find the download mirror nearest to you, but *http://cran.revolution-computing.com/web/packages/index.html* shows a full list of the packages available on CRAN. The majority of these packages have excellent documentation and can be understood and employed with minimal user effort.

To install any of these packages, simply click on its name and download the archive corresponding to your operating system. Then, in your R environment, click on the "Packages" dropdown menu, and select the option to install a package from local files. Next, browse to and select the downloaded package, and R will do the rest. Additionally, you may employ the use of a library in an R session by typing library(*libname*) once installed. All libraries come with references, so be sure to use those to understand how to work with the library.

In addition to using packages that others have developed, R lets anyone and everyone develop packages for personal or public use! You can submit packages to CRAN (provided that they meet specified guidelines) and have them featured as well!

1.4.2 Proactive Community

R is all about the community. With more than 2 million users and thousands of contributors around the world, chances are you can find someone to answer your questions. Do a quick search online for "R help" and you'll find an assortment of mailing lists and forums where you can find answers to the most common questions, or ask your own. Check out *http://www.inside-R.org/*, a new community site for R sponsored by Revolution Analytics. It provides a great language reference, as well

as a lot of promotional information about R. Every so often, it has tips and tricks on how to perform certain tasks.

1.5 Common R functions

In the next few sections, you will find short descriptions and information about functions that you may commonly use when working with R. For a full reference, you should run the `help` command on any of these functions.

1.5.1 Data Manipulation

`read.table`
: The `read.table` command commonly accepts an input file and reads the data from the file into a variable. It accepts a number of other arguments, the most common of which denotes whether the file has a header line or not (header=TRUE or header=FALSE).

`c`
: The `c` command accepts a range or comma-delimited list of values and creates a vector with them.

`matrix`
: The `matrix` command accepts a vector or range of values, as well as several other arguments. The most common of these are the "nrow," "ncol," and "byrow" arguments. "nrow" specifies the number of rows in the matrix, whereas "ncol" specifies the number of columns. The "byrow" argument is TRUE when the data should be read in row-by-row, and false when the data is meant to be read column-by-column.

`seq`
: The `seq` command accepts a start and end value, as well as the "by" argument, which denotes step-size in the sequence. The `seq` command can also accept a start value larger than the end value, in which case it will create a decreasing sequence.

`as.data.frame`
: The `as.data.frame` command accepts an input and forces the values to take the composition of a data-frame.

`as.vector`
: The `as.vector` command accepts an input and forces the values to take the composition of a vector.

`as.list`
: The `as.list` command accepts an input and forces the values to take the composition of a list.

`as.matrix`
: The `as.matrix` command accepts an input and forces the values to take the composition of a matrix.

`apply` family
: The `apply` family of functions are used to execute a function repeatedly on items of a particular data type (list, matrix, etc.) The *apply* family is composed of *apply*, *lapply*, *sapply*, *vapply*, *mapply*, *rapply* and *tapply* . Run *help* on each of these to learn their specific use-cases.

1.5.2 Calculation

While there are no real functions for calculation, there are some common operators that one should know how to use in R.

1.5.2.1 Mathematical Operators

\+
: The "+," or plus sign in R functions the same way as it would elsewhere. It is used to add numbers, vectors, or matrices.

\-
: The "-," or minus sign in R functions the same way as it would elsewhere. It is used to subtract numbers, vectors, or matrices.

*
: The "*," or multiplication sign in R functions the same way it would elsewhere. It is used to multiply numbers, vectors, or matrices (component multiplication, not matrix multiplication).

/
: The "/," or division sign in R functions the same way it would elsewhere. It is used to multiply numbers, vectors, or matrices (component division).

**

The "**" operator in R functions as exponentiation. The value preceding the operator is the base, whereas the value succeeding the operator is the power.

%%

The "%%" sign is the modulus operator in R which returns the remainder of the division of two values. The value preceding the operator is divided by the value succeeding the operator, and the remainder is returned.

%*%

This is the matrix multiplication sign. It should be preceded and succeeded by a matrix. This operation will return a resultant matrix which is the product of the matrix multiplication action.

~

The "~" operator is used to define a relational "model" in R. The general format for a model is $y \sim x$. This model is a simple linear regression of y on x. More complex models can also be built; for example, $y \sim x + z$ creates the multiple regression model of y on x and z. Polynomial models can also be built as follows: $y \sim$ *poly(x,3)* for polynomial regression of y on x of degree 3. In order to find the coefficients of such a model object, one can use function `lm` with the model as the argument.

1.5.2.2 Logical Operators

< or >

The "<" and ">" operators function as less than and greater than signs. These are used in logical calculations (with numbers) and return TRUE or FALSE values.

<= or >=

The "<=" and ">=" operators function as less than or equal to and greater than or equal to signs. They are used in logical calculations (with numbers) and return TRUE or FALSE values.

!=

The "!=" operators function as a 'not equal to' sign. It tests for equality between two numbers or variables and returns TRUE or FALSE values.

!

The "!" operator functions as a 'not' sign. It negates the value of the variable it precedes.

1.5.3 Graphical Display

`plot`
The `plot` function accepts vectors for x and y coordinates and plots them in a graphics window. The function accepts many more arguments (far too many to list here): the most commonly used arguments are the *xlab* and *ylab* arguments that are assigned strings for axis labels, the *main* and *sub* arguments that are assigned strings for main and secondary plot titles, and the *type* argument that is assigned a single letter controlling the type of plot produced (the most commonly used are "p" for plotting individual points (default), "l" for plotting lines, and "b" for a combination of both "p" and "l").

`title`
The `title` function adds a title(s) to the currently active plot. It accepts a single required argument for main plot title, and an optional secondary argument for a sub-title.

`pairs`
The `pairs` function accepts a numeric matrix or data frame and produces a series of pairwise plots where every column of the input is plotted against every other column of the input.

`points`
The `points` function accepts vectors for x and y coordinates and plots the specified points in a graphics window.

`abline`
The `abline` function accepts the slope and intercept of the line to be drawn (a and b, respectively) and draws the line on a graphics window. The function $abline(lm(y \sim x))$ can be used to draw a regression line between vectors y and x (containing the y and x coordinates of concern).

`text`
The `text` function adds text to a plot. It accepts arguments x, y, and a third argument for the text to be placed at point (x, y) of the active plot.

`hist`
The `hist` function accepts a vector of values and plots a histogram with the values on the x-axis and their frequencies on the y-axis.

`x11`
The `x11` function creates a new graphics window so multiple plots are not overwritten in the same window. It is used commonly between generating plots.

1.5.4 Probability Distributions

`summary`
The `summary` function summarizes some of the basic statistics of various fitting models provided as arguments. Most commonly, it is used on univariate data (vectors), and returns the global minimum, 1^{st} quartile, median, mean, 3^{rd} quartile, and maximum of the data.

`fivenum`
The `fivenum` function accepts numerical data as the input argument and returns the minimum, 1^{st} quartile, median, 3^{rd} quartile, and maximum of the data.

`density`
The `density` function is commonly used in conjunction with the `plot` function to produce density plots of univariate data. It can be used as *plot(density(x))*, where x is some univariate vector of data.

`ecdf`
The `ecdf` function is commonly used in conjunction with the `plot` function to produce the empirical cumulative distribution plot of univariate data. It can be used as *plot(ecdf(x))*, where x is some univariate vector of data.

`boxplot`
The `boxplot` function generates comparative boxplots of as many input vectors provided. This is especially useful for comparing multiple samples of data.

`t.test`
The `t.test` function performs the classical Student's t-test on two samples of data. It accepts many arguments, the most basic of which are the two vectors of data samples, and the *var.equal* boolean variable that specifies equal variances or unequal variances of the samples. In case of unequal variances, Welch's t-test is used instead of the classical t-test.

`var.test`
The `var.test` function performs the F-test to compare variances of two samples of data. It accepts the sample vectors as input arguments.

1.5.5 Writing your own R Functions

Like most other programming languages, R provides users the capacity to create their own functions. In R, functions have their own object

type (called "function"). Functions are typically blocks of code that allow performing a certain task repeatedly on multiple data *inputs* that yield varying data *outputs*; these outputs are typically called "return values." Learning how to write functions will drastically increase your productivity while working in R.

Functions are defined in the following manner: *fnname* ← *function(arg1, arg2, ...)* {*code here* }. Then, calling a function is as simple as entering its name and corresponding valid arguments into the R prompt. For example, entering *fnname(a,b)* in the R prompt would result in R running function *fnname* with variables *a* and *b*. Furthermore, in R, function arguments need not be restricted to a certain data type; this means that your functions can accept any type of arguments, as long as the arguments have compatibility with the code/expressions used inside the function. Consider the following example:

```
> add2 <- function(x,y)
+ + x + y+
> a = c(1,2,3,4,5)
> b = c(2,3,4,5,6)
> a
[1] 1 2 3 4 5
> b
[1] 2 3 4 5 6
> add2(a,b)
[1]  3  5  7  9 11
> d = matrix(c(1:9),nrow=3,ncol=3)
> f = matrix(c(10:18),nrow=3,ncol=3)
> d
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> f
     [,1] [,2] [,3]
[1,]   10   13   16
[2,]   11   14   17
[3,]   12   15   18
> add2(d,f)
     [,1] [,2] [,3]
[1,]   11   17   23
[2,]   13   19   25
[3,]   15   21   27
```

In this example, we defined a function *add2* that took two arguments

and returned the sum of the arguments as defined by the "+" operator. In both usecases of the function, we used arguments of different data types: the first usecase involved vectors, whereas the second usecase involved matrices. Since both vectors and matrices can be summed using the "+" operator, R happily accepted the function calls without complaints. However, had we used a third usecase where we tried to call *add2(a,d)*, we would have received an error, since the "+" operator does not allow summation of a vector with a matrix.

Lastly, this example also shows how R deals with function returns. Essentially, the result of the last statement of code in the function is used as the return value of the overall function. In order to group multiple objects into the return value of a single function, the most straightforward solution is to package the objects together appropriately using a vector or list object.

Additionally, functions can call other functions provided that the inner function calls use arguments or variables visible to the parent function. Consider the following example:

```
1  > statcalc <- function(x)
2  + {
3  + me = mean(x);
4  + med = median(x);
5  + stdv = sd(x);
6  + va = var(x);
7  + c(me,med,stdv,va);
8  + }
9  > a = c(1,2,3,4,5)
10 > a
11 [1] 1 2 3 4 5
12 > statcalc(a)
13 [1] 3.000000 3.000000 1.581139 2.500000
```

In this example, we created a function *statcalc* designed to compute the mean, median, standard deviation, and variance of an input. Specifically, *statcalc* creates 4 local variables, *me*, *med*, *stdv*, and *va* inside the function, and assigns them the values of the respective *mean*, *median*, *sd*, and *var* inbuilt R functions. Finally, *statcalc* creates a 4 element vector containing these values. Since the last statement of code here is the creation of the vector containing statistics of *a*, the return value of function *statcalc* is this vector. We provided *statcalc* with the input vector *a*, and received the return value of a vector containing the statistics we specificed in *statcalc*. By specifying the creation of this vector as the last line of our function, we were able to package together these 4 individual statistic variables into a single object for the function to return.

1.6 R Installation

You can run R on Windows, *NIX, and OSX environments. The installation processes for all of these are slighly different, but not very difficult.

1.6.1 Windows Systems

Installation of R on Windows systems is simple with a graphical interface. To install R on your Windows machine, go to *http://cran.revolution-computing.com/bin/windows/base/* and select the download appropriate for your system. Most users will likely want the 32-bit version of R (if you don't know whether your machine is 32-bit or 64-bit, be safe and use the 32-bit). Note that there is a link on the page to select a 64-bit version as well, so if your machine is running a 64-bit Windows OS, feel free to download that one as well and utilize all of your memory.

Upon downloading the executable installer file, run it, and follow the instructions to get your R installation up and running!

1.6.2 *NIX Systems

Installing R on a *NIX system is not as easy, but can be done without too much effort. You will want to navigate to *http://cran.revolution-computing.com/* and download the .tgz archive that contains the R installation files. Upon downloading the archive, you will want to startup a terminal session and navigate to the directory your archive was downloaded to. Next, you should untar your archive (by using "tar zxf *.tgz"). Upon navigating to the directory in which all your files were untarred, you can now enter the "configure," "make," and "make install" commands to fully install your R environment.

1.6.3 Mac OSX Systems

If you use OSX, then you will want to navigate to *http://cran.revolution-computing.com/bin/macosx/*, and download the .pkg file displayed on the webpage. You can then simply install that .pkg file and follow any instructions along the way. Upon doing so, you will have your R environment ready to go!

1.7 Exercises

1. ```
   > year=seq(1790,1950,10)
   > m.age=c(15.9,15.7,15.9,16.5,17.2,17.9,19.5,20.2,
   +20.6,21.6,22.9,23.8,24.9,26.1,27.1,29.5,30.4)
   ```

   The above code creates a vector named year with the sequence of years between 1790 and 1950 (with a step-size of 10) and another vector named m.age with the median age of residents of the United States.

   Calculate the mean, standard deviation, and variance of the data in m.age using R functions. Additionally, calculate the minimum, maximum, first, and third quartile values of age.

2. Using the data from Exercise 1, create a plot which has an x-label of "Year", y-label of "Median Age", plot *title* of "Plot of U.S. Resident Median Age by Decade from 1790-1950", and lines between points.

   HINT: Use help(plot) to find detailed information about the plot function and its arguments.

3. ```
   > x = c(1:10)
   > y=x^2
   ```

   The R code above simply creates 2 vectors, x and y, corresponding to values of the function $y = x^2$ in the domain [1,10]. Use R expressions to write the following:

   (a) Four plus the square of x.
   (b) The difference of y and x.
   (c) The sum of all elements in y.
   (d) A vector containing the product of each entry of x with the corresponding entry of y.
   (e) The first 5 elements of y (use the facts that a range of values can be written as a:b, and you can access elements of a vector with a call such as $v[a : b]$, where a and b are indices).

4. Use the R `seq` function to write the following:

   (a) A sequence of odd numbers between 1 and 10.
   (b) A sequence of even numbers between 1 and 10.
   (c) A sequence of decreasing numbers from 10 to 1.

(d) A sequence of cubes of odd numbers between 1 and 10.

5. $(A) = \begin{pmatrix} 2 & 5 \\ 8 & 4 \end{pmatrix}$ $(B) = \begin{pmatrix} 7 & 6 \\ 2 & 3 \end{pmatrix}$

   Given the matrices above, use R to calculate the following (and provide the code used to do so):

   (a) The matrix sum.
   (b) The sum of the first matrix with 3 times the second matrix.
   (c) The determinant of **A**.
   (d) The matrix multiplication of **A** with **B**.
   (e) The transpose of **B** (use the `t` function).

6. ```
   1 > addfour = function(x){x+4}
   ```

   This is an R function. Functions accept arguments and return values in accordance with their purpose. In this example, we consider a function `addfour`, which accepts any data type x which supports addition with a scalar in R. This includes matrices, vectors, etc. Write your own simple function to find the cube of its input.

7. Download the random package and use a function in the package to create a 10x10 data frame with 100 random numbers between 1 and 25.
   ```

2

Solutions

Solutions to Chapter 1 Exercises

1.
```
> library(stats)
> summary(m.age)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  15.70   17.20   20.60   21.51   24.90   30.40
> sd(m.age)
[1] 4.83385
> var(m.age)
[1] 23.36610
```

2. Figure 2.1 displays the solution.

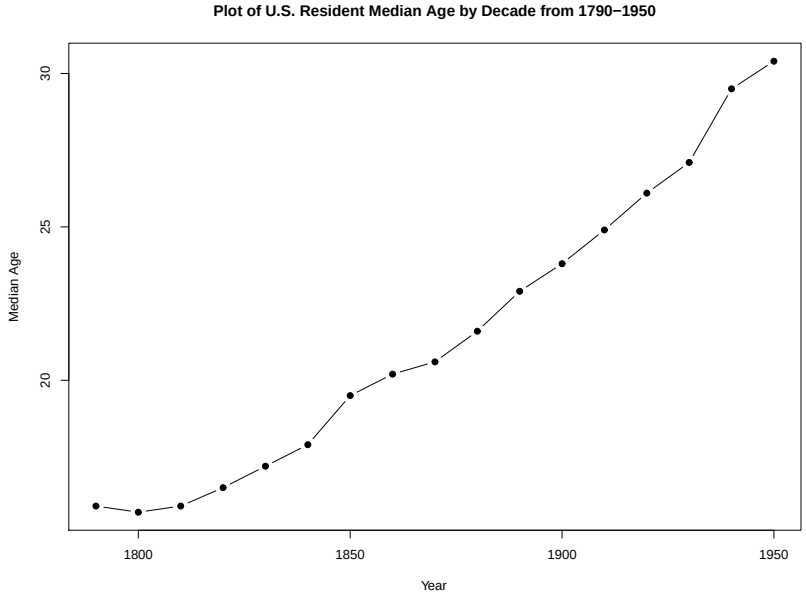

FIGURE 2.1: The output of the plot command plot(year,m.age,type="b",col="black",xlab="Year",ylab="Median Age",main="Plot of U.S. Resident Median Age by Decade from 1790-1950",font.main=2,font.lab=1,pch=19)

3. (a) $4 + y$
   (b) $y - x$
   (c) $sum(y)$

(d) $x * y$

(e) $y[1:5]$

4. (a) $seq(1, 10, by = 2)$

(b) $seq(2, 10, by = 2)$

(c) $seq(10, 1, by = -1)$

(d) $seq(1, 10, by = 2)\hat{}3$

5. (a) $\mathbf{A} + \mathbf{B} = \begin{pmatrix} 9 & 11 \\ 10 & 7 \end{pmatrix}$

(b) $\mathbf{A} + 3{*}\mathbf{B} = \begin{pmatrix} 23 & 23 \\ 14 & 13 \end{pmatrix}$

(c) $det(\mathbf{A})$ = -32

(d) $\mathbf{A}$ *%*%* $\mathbf{B} = \begin{pmatrix} 24 & 27 \\ 64 & 60 \end{pmatrix}$

(e) $t(\mathbf{B}) = \begin{pmatrix} 7 & 2 \\ 6 & 3 \end{pmatrix}$

6.
```
> cube = function(x){x^3}
```

7.
```
> randomNumbers(n=100,min=1,max=25,col=10)
      V1 V2 V3 V4 V5 V6 V7 V8 V9 V10
 [1,]  3 20  9 22 17  2 23 12  9  13
 [2,]  3  4 21  8 25 15 24 10  7   4
 [3,] 18 23 25  4 10 22  1 15  3   2
 [4,]  9  3 22  7 12  9 15 21 14  24
 [5,] 11  3 16 20 13  2  9  8 25   4
 [6,] 19  3 13  9  6 20  8 14 22   1
 [7,] 25  6  9 11 11 21 17 15 23   6
 [8,]  9 22 21 19  5 13 16  5  1   6
 [9,]  4 11  7 11 24 20  3  3 23  20
[10,] 15  9  5 16  4 13 11 16 19  14
```

Bibliography