CHAPTER 1

Introduction to R statistical environment

Why R?

If you work in the field of biodata analysis, or if you are interested in getting a bioinformatics job, you can see a large number of related job advertisements targeting young professionals. There is one common topic coming back in those ads: they demand "a high degree of familiarity with R/Bioconductor." (Here, I am quoting an actual recent ad from Monster.com.)

Besides, when we have to create and analyze a large amount of data during our bio-researcher career, sooner or later we realize that simple approaches using spread sheets (aka the Excel part of MS Office) are not flexible anymore to fulfill the needs of our projects. In these situations, we start to look for dedicated statistical software tools, and soon we encounter the countless alternatives from which we can choose. The R statistical environment is one among the possibilities.

With the exponential spread of high-throughput experimental methods, including microarray and next-generation sequencing (NGS)-based experiments, the skills related to large-scale analysis of data from biological experiments have higher and higher value. R and Bioconductor offer a free and flexible tool-set for these types of analyses; therefore, many research groups and companies select it as their data analysis platform.

R is an open-source software licensed under the GNU General Public License (GPL). This has an advantage that **you can install R for free** on your desktop computer, regardless of whether you use Windows, Mac OS X, or a Linux distribution.

Introducing all the features of R thoroughly at a general level exceeds the scope and purpose of this book, which is to focus on molecular biology-specific applications. For those who are interested in a deeper introduction into R itself, it is suggested reading the book *R for Beginners* by Emmanuel Paradis as a reference guide. It is an excellent general guide, which can be found online (Paradis 2005). In the course, we use more biology-oriented examples to illustrate the most important topics. The other recommended book for this chapter is *R in a Nutshell* by Joseph Adler (2012).

Molecular Data Analysis Using R, First Edition. Csaba Ortutay and Zsuzsanna Ortutay.

Companion website: www.wiley.com/go/ortutay/molecular_data_analysis_r

Installing R

The first task of analyzing data with R is to install R on the computer. There is a nice discussion on the bioinformatics blogs about why people so seldom use their knowledge acquired on short bioinformatics courses. One of the main considerations points out that it is because the greatest challenge is to install the software in question.

There are plenty of available information on the web about how to install R, but the most authentic source is the website of the R project itself. In this page, the official documentation, installer, and other related links from the developers of R themselves are collected. The first step is to navigate to the download section of the page and find the mirror pages closest to the location of the user.

However, there are some differences in the installation process depending on the operating system of the computer in use. Windows users should find the **Windows installer** to their system from the download pages. It is useful to check for the **base** installer, not the contributed libraries. In the case of a **Linux** distribution, R can be installed via the package manager. Several Linux distributions provide R (and many R libraries) as a part of their repositories. This way, the package manager can take care of the updates. **Mac OS X** users and Apple fans can find the pkg file containing the R framework, 64-bit graphical user interface (GUI) (R.app) and Tcl/Tk 8.6.0 X11 libraries for installing the R base systems on their computer. Brave **users of other UNIX systems** (i.e., FreeBSD or OpenBSD) can use R, but they should compile it from the source. This is not a beginner topic. In the case of a computer owned by a company, university, or library, the installation of R (just like many other programs) requires most often superuser rights.

Interacting with R

The interface of R is somewhat different from other software used for statistics, such as SPSS, S-plus, Prism, or MS Excel (which is **not** a statistical software tool!). There are neither icons nor sophisticated menus to perform analyses. Instead, commands should be typed in the appropriate place of R called the "command prompt". It is marked with >. In this book, the commands for typing into the prompt are marked by fixed-width (monospaced) fonts:

```
> citation()
```

After typing in a command (and hitting Enter), the results turn up either under the command or, in case of graphics, in a separate window. If the result of a command is nothing, the string NULL appears as a result. Mistyping or making an error in the parameters of a command leads to an error message with some information about what was wrong.

```
> c()
NULL
> a * 5
Error: object 'a' not found
```

From now on, we will omit the > prompt character from the code samples so you can just copy/paste the commands. Leaving R happens with the *quit()* function.

```
quit(save='no')
q()
```

Graphical interfaces and integrated development environment (IDE) integration

A command-line interface is enough for performing the practices. However, some prefer to have GUI. There are multiple choices depending on the operating system in use. The Windows and Mac versions of R starts with a very simple GUI, while Linux/UNIX versions start only with a command-line interface. The Java GUI for R is available for any platform capable of running Java, and it sports simple, functional menus to perform the most basic tasks related to an analysis (Helbig, Urbanek, and Fellows 2013).

For a more advanced GUI, one can experiment with RStudio or R Commander (Fox 2005). There are several plugins to integrate R into the best coding production tools, such as Emacs (with the Emacs Speaks Statistics add-on), Eclipse (by StatET for R), and many others.

Scripting and sourcing

Doing data analysis in R means typing in commands and experimenting with parameters suitable for the given set of data. At a later stage, the procedure will be repeated either on the same data with slight modifications in the course of the analysis, or on different data with the same analysis. For example, the analyzed data are submitted to publication, but the manuscript reviewers request slight modifications in the analysis. It means to repeat almost the entire process, but parameter *x* should be 0.6 instead of 0.5 as used earlier.

Scripts are used to register the steps of an analysis. Scripts are small text files containing the commands of the analysis one after the other, in the same order as are issued during the data processing. Traditionally, we use ".R" extension (instead of .txt) for these text files to mark that these are R script files. Script files are the solution for

1. archiving an analysis,
2. automate tasks that take much time to run.

Script files can easily be included into an analysis flow called "sourcing" (the term is borrowed from other computer script languages) by issuing

the *source()* command. For example, let's have the following script file my_first_script.R:

```
a<-rep(5,5)
b<-rnorm(5)
print(a)
print(b)
print(a*b)
```

Scripts can be created using any text editor (i.e., gedit, mcedit, Notepad) but not with a word processor software (i.e., MS Word, LibreOffice Writer, and iWork Pages) unless it is possible to save it as a text file, and not a .doc, .docx, .odt, or any other more complex formats. R should be led to the location where the saved file can be found:

```
setwd('/home/path/to/your/files') #on Linux/UNIX
setwd('/Users/User Name/Documents/FOLDER') #on Mac
setwd('c:/path/to/my/directory/') #on Windows
```

The working directory can be checked using the *getwd()* command.

Loading the script file in the working directory is simple:

```
source('my_first_script.R')
```

If the script is somewhere else, the path is desired:

```
source('/path/to/my_first_script.R')
```

The R history and the R environment file

When starting R first time, it creates two files to register what was done: the history and the environment file. If R was started from the command line, these files are saved in the directory where R was started. Launching R by an icon results in saving the history and the environment file to a default place.

The history file is a text file that saves all the commands issued in a session with R, while the environment file holds the data used during the session. It is worth saving these files for further use with the *savehistory(file = "/path/to/Rhistory")* and *save.image(file = "/path/to/RData")* commands. When exiting R by using the *q()* command, it asks whether you want to save these to the default places. Choosing this option leads to start R from the same directory next time, and it will also remember the past work and data.

Packages and package repositories

Statistics is a huge field, and many disciplines use it for their specific purposes. All of them have different needs, flavors, and data types specifically designed for their needs. It would be meaningless and hopeless to put everything into a single software. Honestly, the majority of the code would never be used as

a bioinformatician rarely uses statistics designed for particle physics, likewise a computational chemist rarely reads in data from gene expression microarrays.

To address this problem, R developers have decided to provide only a common framework and some basic functionality as part of the base installation, and subject-specific elements are organized into code bags called "packages." In reality, the base installation of R is not very useful for molecular data analysis. Its most useful part is that suitable packages can be found for most of the often applied analysis types.

R packages are collected into so-called package repositories on the web. These places are dedicated to the maintenance and distribution of the packages. The concept is probably familiar to Linux users. R uses its own internal code called "package management system" to find, install, and update packages. There are two important package repositories, which are also used in this book: Comprehensive R Archive Network (CRAN) and Bioconductor.

Comprehensive R Archive Network

(R Core Team 2015) is a place for general-purpose packages, but many biology-related packages can be found here too. One can search packages related to the topic of interest (left side of the page, Software/packages/table of available packages, sorted by name) by keyword search. For example, if some biological sequences-related packages are required, searching (Ctrl+F) for the keyword "biological sequence" on this page will result in those soon.

Here, we introduce the **sequences** package (Gatto and Stojnic 2014). Clicking on the name of the package leads to a general information page. The most relevant documents here are the Vignettes (if they are available), providing a quick introduction to the package, and the reference manual that shows an extensive explanation for all the commands and datasets provided by the package.

Installing and managing CRAN packages is best done within R itself. Most GUIs provide some assistance for package management in the "Packages" menu. It is simple to install packages using the *install.packages()* command. Picking up the packages and their dependencies requires Internet access. The installation process can take much time if the selected package has many other packages to depend on.

```
install.packages("sequences")
```

On **Linux** *install.packages()* works properly if it is issued in an R session of the root, or if a library is specified as the package directory to write:

```
install.packages("sequences", lib="/home/mydir/Rpackages/")
```

The full list of available packages can be checked using *available.packages()*. This command lists all the packages compatible with the version and operating system on the computer in use. It often means many thousands of packages.

```
ap<-available.packages()
row.names(ap)
```

Loading a successfully installed package (e.g., the **sequences** package in the previous example) is done using the *library()* command (without quotation marks around the package name this time).

```
library(sequences)
```

Bioconductor

There is another R package repository dedicated mostly to the analysis of high-throughput data from molecular biology, called "Bioconductor" (Gentleman et al. 2004). It contains more than 1500 packages dedicated to this exciting field of bioinformatics. The packages are divided into three groups:

1. Software—This section contains the most interesting packages that can assist different kind of analysis. This sub-repository is roughly analog to CRAN in the sense that the packages here provide the statistical methods and procedures, such as microarray normalization functions or enrichment analysis approaches.
2. AnnotationData—Here is a collection of very important supporting information concerning genome, microarray platform, and database annotation. These packages are useful mostly as input data for other packages in the Software section.
3. ExperimentData—Prepared experimental data are available from here for further analysis. It is a good idea to test a new statistics or analysis approach on data from here first. This effort will assure that the code in use is compatible with the rest of Bioconductor's framework.

The packages are listed in a logical and hierarchical system, and it is relatively easy to find relevant packages for a certain type of analysis. For example, if mass spectrometry is in the focus of interest, the relevant packages can be found in the Software -> Assay Technologies -> Mass Spectrometry branch of the hierarchy; while in case of inferring networks from experimental data, the Software ->Bioinformatics -> Networks -> Network Inference branch should be checked. The vignette and the reference manual appears in a similar way on the dedicated page of the chosen package as it is in CRAN.

There is another, perhaps even more practical, way to find suitable packages from Bioconductor. There are complete recipes for more popular data analysis tasks in the Workflows section of the Bioconductor page Help menu, which not only shows the needed packages but also demonstrates how to use them.

Bioconductor uses its own package management system that works somewhat differently than the stock R system. It is based on a script titled *biocLite.R*, which can be sourced directly from the Internet:

```
source("http://bioconductor.org/biocLite.R")
```

This script contains everything needed for managing Bioconductor packages. For example, to install the **affy** package (Gautier et al. 2004), the *biocLite()* command should be called:

```
source("http://bioconductor.org/biocLite.R")
biocLite("affy")
```

This command processes the dependencies and installs everything in need. The **annotation and experimental data packages** tend to be **huge**, so high-speed Internet (or a lot of patience) and sufficient amount of disk space is needed to install them.

Loading of the installed packages happens in the same way as with CRAN packages:

```
library(affy)
```

Working with data

For a data analysis project, well, data are needed. It is a crucial question, how to load data into R, and that is often the second biggest challenge for a newbie bioinformatician. Several R tutorials start to explain this topic by introducing the *c()*, *edit()*, and *fix()* commands. These are commands and functions used to type in numbers and information in a tabular format. Also, these are the commands that are rarely used in a real-life project. The cause of this is simple: no-one type in the gene expression values of 40,000 gene probes for a few dozens of samples.

Most often data are loaded from files. Files might come out from databases, from measurement instruments, or from another software. Often data tables are assembled in MS Excel. MS Excel or other spreadsheet software can also export data tables as .csv files, which are easy to load to R. Depending on the operating system in use and the exact installation of R, there are multiple possibilities to read .xls files. The package **gdata** (Warnes et al. 2015) contains the *read.xls()* command, which can access the content of both .xls and .xlsx files:

```
library(gdata)
my.data<-read.xls("data_file.xlsx", sheet=1)
```

This code reads in a table from the first sheet in the .xlsx file into the *my.data* data frame. This is an excellent tool, but it requires the installation of Perl (a scripting language) on the computer. In Linux/Unix installations it is not a problem, but in Windows environments, it is not easy to solve. A universal solution of this problem is to read data from exported .csv files. This approach works for all platforms, and it does not require the installation of additional packages:

```
my.data<-read.csv("fdata_file.csv",sep="\t",row.names=1)
```

The first step is to prepare a data table using Excel or another spreadsheet software. The data are then exported to a .csv file called "tabular text file" or "comma-separated text file" in different software. It is important to specify the usage of a tab as a field separator (sep="\t") in the settings instead of a comma that is usually the default field separator for .csv files.

To handle the problem of transferring data from MS Excel to R, a new package called **readxl** has been released recently (Wickham 2015). The ultimate goal of this package is to provide accessibility to data saved in Excel files without further dependencies, and in an operating system-independent way.

There are many proprietary file formats coming out from different instruments. There are dedicated packages developed to read their content and load it in proper data structures in R for further analysis. For example, the *ReadAffy()* command from the **affy** package is designed to import Affymetrix GeneChip CEL files. Similarly, the *read.fasta()* command of **seqinr** package (Charif and Lobry 2007) or the *readFASTA()* command of **Biostrings** package (Pages et al. 2015) can import FASTA formatted sequence files.

This book has a dedicated support webpage. Here, all the R scripts and data are available to do all the practices discussed in the following chapters. As bonus material, the scripts used for generating the figures on these pages are also available from the same place.

Save the file furin_data.csv from the webpage of the book, and open it with a text editor. There are rows and columns of the data in the file. The first step for now is to set the exact path to the location of the file in the furin.file variable, and use the *read.csv()* command to read its command to the *my.data* variable. Checking the structure of the data happens by using the *str()* command.

```
furin.file<- '/path/to/your/file/furin_data.csv'
my.data<-read.csv(furin.file,sep="\t")
str(my.data)
```

Basic operations in R

All scripting languages provide simple ways to perform basic computational operations on data. R is not different in that sense. Certainly, the most basic things like arithmetic operations work as expected. For example, adding and multiplying with numbers the same way as in math classes:

```
4 + 7
6 * 2
```

Of course, R is not the most suitable choice if only a calculator is needed. R is used to store numbers, information, and data, and also to perform different tricks and calculations on those. For those, who know one or other programming

languages, it is clear that variables should be used. For the sake of those who are not familiar with these issues: variables are similar to labeled "shoe-boxes" containing data items. During an analysis the data items can be stored in these "shoe-boxes" instead of reading them from file for each operation. The arrow mark (or assignment operator) is used for loading any data, for example, a number, into these variables.

```
my.data <- 5
```

Now the number 5 is loaded into the *my.data* variable. The direction in which way the arrow points tells the story. For example, the result of an operation can be stored in a variable, and later on the data inside of the variable can be the subject of further operations.

```
my.data <- 5 + 3
my.other.data <- my.data * 2
```

Typing the name of the variable will show what is inside it:

```
>my.data
8
>my.other.data
16
```

The variables in R can store a great many types of different things like numbers, list of numbers, strings, sequences, data tables, data matrices, entire genomes, or multiple sequence alignments. Several operations have different meanings depending on what kind of data are applied to them. R is smart enough to figure out if a command has a different version specifically fit for a particular data type.

```
a <- 5
a + 3
b <- c(5,6,7)
b + 3
```

In the previous example, there are two very different variables: *a* and *b*. Variable *a* holds a single number (5), while variable *b* holds a vector of three numbers. The addition (+) operator guesses that applying it to a single number (variable *a*), it should add 3 to a single number. However, in case of vector (*b*), it will add 3 to all the numbers in the vector. This distinction is crucial, as the result of the first operation is a single value, while the result of the second one is a vector itself.

R has this kind of smart redundancy that is especially handy with the *plot()* command. There are several data types that are represented in graphs and figures, which are very often generated by the *plot()* command. Specific data types have their specific plots, and R packages are well prepared to draw different plots for them. Using the furin_data.csv file again as an example, different graphs can be

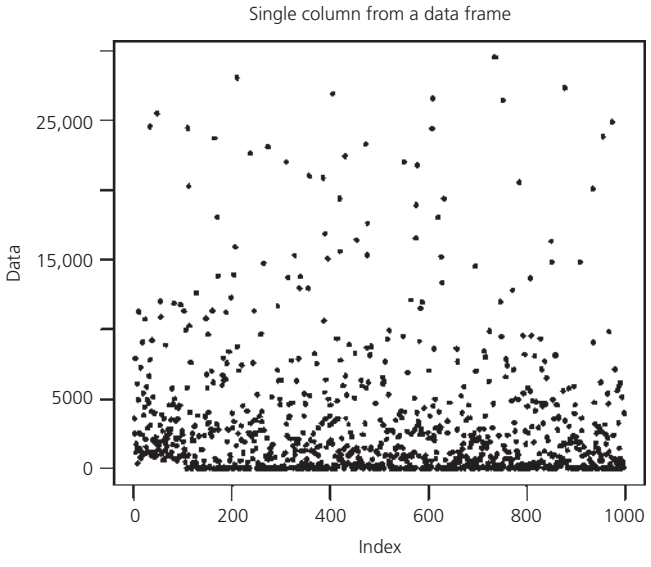

Figure 1.1 The *plot()* function produces a scatter plot when it is called on a column of a data frame. Unless specified differently, the values appear in their order in the data frame itself.

generated by of plotting one column of the data table (Figure 1.1), or all of them for checking their correlation (Figure 1.2):

```
furin.file<- '/path/to/your/file/furin_data.csv'
my.data<-read.csv(furin.file,sep="\t")
plot(my.data$Naive.KO.1)
plot(my.data)
```

When complex data tables and data structures are loaded or created, the variables contain specific types of data like vectors, lists, matrices, or data frames. Many commands need one of these data types since some operations make sense for particular data. For example, calculating the mean works with a vector of numbers but not with DNA sequences. Putting the data into an accurate format to be suitable for a certain command is a special point in working with R. This is the problem that causes a lot of headache for R newbies as well as longtime R users.

Some basics of graphics in R

A picture is worth a thousand words, as the old saying states. In data analysis context, it means that very often the goal of using R is to summarize the results of an experiment using plots and images. R has a versatile imaging facility that is a bit difficult at first. After taming it, however, it will help to generate publication-quality images with the possibility to adjust all their important aspects.

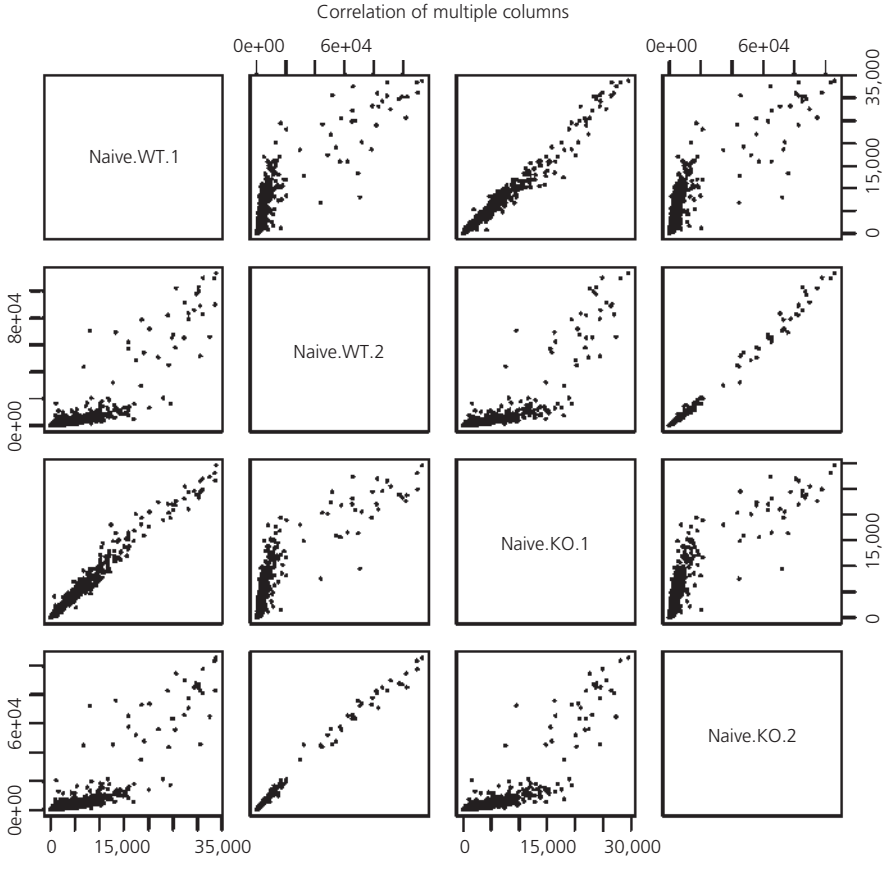

Figure 1.2 Calling *plot()* on multiple columns of a data frame results a correlogram among the columns of the data frame.

The topic is covered by shorter writings as well as several books. The main idea is simple, although it is somewhat unusual. Calling the *plot()* command (or one of its relatives) results in an image appearing on the screen. This image is not the one that is saved into a file! The secret behind the system is called "graphical device." Calling a *plot()* command in R sends the graphics to the actual graphical device that is open at the moment. The default is a window on the screen. However, any other devices can be opened, such as a pdf, a jpeg, or a tiff file. Those are other graphical tools for R, so the next *plot()* command will draw into those open devices. These file devices should be closed first, so R can take care of their proper formatting.

```
b<-rnorm(1000)
hist(b)
```

The first command here creates a vector of 1000 random, normally distributed numbers, and stores them in the b variable. The *hist()* command (a close relative

of *plot()*) shows the frequency distribution of those numbers, so the shape of the histogram is not really surprising. Now how to have the same in a jpeg file?

```
setwd('/home/path/to/your/files') #on Linux/UNIX
setwd('/Users/User Name/Documents/FOLDER') #on Mac
setwd('c:/path/to/my/directory/') #on Windows
jpeg(filename="my_first_plot.jpg")
hist(b)
dev.off()
```

After issuing these commands, a new jpeg file called "my_first_plot.jpg" appears in the working directory (Figure 1.3). It has the histogram similar to that seen before, but now it can be included in a manuscript or project report. Also, many aspects of the file itself can be adjusted, including its size and resolution.

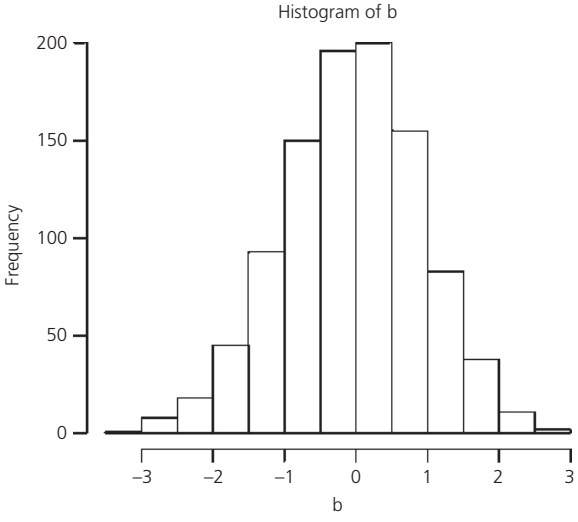

Figure 1.3 The frequency distribution of 1000 random numbers generated by the *hist()* function.

```
jpeg(filename="my_first_plot.jpg", width = 640, height = 480,
   res=100, quality=100)
```

The *pdf(), bmp(), png()*, and *tiff()* devices can be used in a very similar manner.

Getting help in R

As it was shown in the call of *jpeg(), read.csv()*, and other commands, different commands and functions can have many parameters and requirements. Often an explanation is desired about these parameters, or about the details of how a particular function works. R provides an excellent help system to assist in these issues. If one wants to know the suitable input data, the different parameters about a particular function (known by its name), either the

help() command can be used, or the function name should be appended to a question mark:

```
help("jpeg")
help("read.csv")
?jpeg
```

These commands provide all the necessary details. However, this approach has some limitations. First, it is extremely useful if the help page is open during experimenting with the different parameters of a command. Next, there are cases when the name of a function in not known, although the task to perform is clear (e.g., Student's t-test). In these situations, it is a good idea to look for the more advanced help system built into R. Calling the *help.start()* function will direct to a local page on the web browser. Here, the local manuals can be browsed that provide detailed information about the locally installed packages, and one can also perform searches in these pages.

```
help.start()
```

If one dives into the provided help pages, it can be noted that the result of this command is a huge amount of very detailed help supported by manuals and examples. After a little practice, it can be realized that almost all possible questions are already answered here. What remains for figuring out how to dig out those answers. The help facility of R can assist in this task.

In addition to this, there are two more education-oriented services built in R: the commands *example()* and *demo()*. The former is provided for the most important commands, while the latter is designed to demonstrate selected topics and how to perform them with the installed packages.

```
example(read.csv)
```

This command prints several lines of text with instructions and explanations about the *read.csv()* command.

```
demo()
```

This command will show the list of available demos coming from the installed packages on the computer in use. The base installation contains quite a few of them (e.g., about recursion and scoping). It is easy to call them and check the possibilities that R provides:

```
demo(recursion)
demo(image)
```

Files for practicing

furin_data.csv—This file is an excerpt of a gene expression microarray dataset. There are normalized gene expression values for different genes in the rows and

different samples in the columns. The file is in CSV format, but the filed separators are tabulator (tab) characters, which is denoted in R as "\t." The first row contains the column names. This file can be read into R using either the *read.csv()* function using the sep="\t" parameter, or by the *read.delim()* function.

Study exercises and questions

1. How to assign values to different variables in R?
2. What is a data frame ?
3. What are packages in R, and how to load a suitable package?
4. How to save a sequence of R commands to a file?
5. What are package repositories for R?
6. What is Bioconductor, and how it is related to R?
7. Give an example of assigning a vector to a variable.
8. How do we get a help on a function? List multiple ways.
9. If you had to name at least two useful R packages for bioinfomaticians, which ones would you choose? Provide a brief description of their functionality.
10. If you are plotting into a file, how to instruct R to close the graphic file?
11. In your opinion, what are the advantage/disadvantages of working with R-studio as compared to regular R-console?
12. Which are the commands to check and set your working directory?
13. Explain the difference between R history and R environment file?
14. How to install a certain package in R?
15. What happens if scripts are not saved as text files, but as .doc/.docx files? Why?
16. Describe how we can load a script that is located in the same place as the working directory and a script located outside of the working directory?

References

Adler, J. 2012. *R in a Nutshell*. O'Reilly Media. http://shop.oreilly.com/product/0636920022008.do (accessed May 4, 2016).

Charif, D., and J. R. Lobry. 2007. "SeqinR 1.0-2: A Contributed Package to the R Project for Statistical Computing Devoted to Biological Sequences Retrieval and Analysis." In *Structural Approaches to Sequence Evolution: Molecules, Networks, Populations*, edited by U. Bastolla, M. Porto, H. E. Roman, and M. Vendruscolo, 207–32. Biological and Medical Physics, Biomedical Engineering. New York: Springer Verlag.

Fox, J. 2005. "The R Commander: A Basic Statistics Graphical User Interface to R." *Journal of Statistical Software* 14 (9): 1–42.

Gatto, L., and R. Stojnic. 2014. *Sequences: Generic and Biological Sequences*. http://CRAN.R-project.org/package=sequences (accessed May 4, 2016).

Gautier, L., L. Cope, B. M. Bolstad, and R. A. Irizarry. 2004. "Affy-Analysis of Affymetrix GeneChip Data at the Probe Level." *Bioinformatics* 20 (3): 307–15. doi:http://dx.doi.org/10.1093/bioinformatics/btg405.

Gentleman, R. C., V. J. Carey, D. M. Bates et al. 2004. "Bioconductor: Open Software Development for Computational Biology and Bioinformatics." *Genome Biology* 5: R80.

Helbig, M., S. Urbanek, and I. Fellows. 2013. *JGR: JGR - Java GUI for R*. http://CRAN.R-project.org/package=JGR (accessed May 4, 2016).

Pages, H., P. Aboyoun, R. Gentleman, and S. DebRoy. 2015. *Biostrings: String Objects Representing Biological Sequences, and Matching Algorithms*.

Paradis, E. 2005. *R for Beginners*. http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf (accessed May 4, 2016).

R Core Team. 2015. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. http://www.R-project.org/ (accessed May 4, 2016).

Warnes, G. R., B. Bolker, G. Gorjanc, G. Grothendieck, A. Korosec, T. Lumley, D. MacQueen, A. Magnusson, J. Rogers et al. 2015. *Gdata: Various R Programming Tools for Data Manipulation*. http://CRAN.R-project.org/package=gdata (accessed May 4, 2016).

Wickham, H. 2015. *Readxl: Read Excel Files*. http://CRAN.R-project.org/package=readxl (accessed May 4, 2016).

Webliography

http://www.r-project.org/—The official website of the R project.

http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf—An excellent general guide from Emmanuel Paradis: R for Beginners.

http://shop.oreilly.com/product/9780596801717.do—R in a Nutshell, a recommended book from Joseph Adler.

http://cran.r-project.org/mirrors.html—The download section of R webpage.

https://rforge.net/JGR/—The Java GUI for R.

http://www.rstudio.com/—The homepage of Rstudio, a GUI for R.

http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/—The homepage of R commander, a GUI for R.

http://www.sciviews.org/—The homepage of SciViews-R, a GUI for R.

http://stat.ethz.ch/ESS/—Emacs Speaks Statistics, an add-on package for Emacs text editor.

http://www.vim.org/scripts/script.php?script_id=2628—Nvim-R: Plugin to Vim text editor with R.

http://www.walware.de/goto/statet—StatET: a plugin for integrating R into Eclipse.

http://cran.r-project.org/index.html—CRAN, the general package repository for R.

http://www.bioconductor.org/—R package repository dedicated mostly to the analysis of high-throughput data from molecular biology.