‘From Statistics to Data Mining’ Computer Lab Session n° 2: Introduction to R (2/2)

Master 1 COSI / CPS^2 Saint-Étienne, France

Fabrice Muhlenbach

Laboratoire Hubert Curien, UMR CNRS 5516
Université Jean Monnet de Saint-Étienne
18 rue du Professeur Benoît Lauras
42000 SAINT-ÉTIENNE, FRANCE
http://perso.univ-st-etienne.fr/muhlfabr/

Outcome

The goal of this lab is to discover a more user-friendly working environment with R by using an IDE (*RStudio*), to use the control structures of R programming language, and to produce graphics.

1 Working with an IDE

1.1 Why an Additional R Graphical User Interface?

For longer programs (called scripts) there is too much code to write all at once at the command prompt. Furthermore, for longer scripts it is convenient to be able to only modify a certain piece of the script and run it again in R. Programs called script editors are specially designed to aid the communication and code writing process. They have all sorts of helpful features including R syntax highlighting, automatic code completion, delimiter matching, and dynamic help on the R functions as they are being written. Even more, they often have all of the text editing features of programs like Microsoft Word. Lastly, most script editors are fully customizable in the sense that the user can customize the appearance of the interface to choose what colors to display, when to display them, and how to display them.

Most of these programs are “Graphical User Interfaces” (GUIs), softwares with an interface in which the user communicates with R by way of points-and-clicks in a menu of some sort. Several projects are aiming to build an easier to use GUI for R.

1.2 Examples of R GUIs

Rcmdr

The *Rcmdr* (“R Commander”) project is an R package that provides an alternative GUI for R. You can install it as an R package. It provides some buttons for loading data, and menu items for many common R functions.

Rkward

Rkward is a slick GUI front-end for R. It provides a palette and menu-driven UI for analysis, data editing tools, and an IDE for R code development.

Tinn-R/Sciviews-K

Tinn-R ("Tinn is not Notepad") is completely free and has all of the above mentioned options and more. It is simple enough to use that the user can virtually begin working with it immediately after installation. But *Tinn-R* proper is only available for Microsoft Windows operating systems. If you are on MacOS or Linux, a comparable alternative is Sci-Views – Komodo Edit.

Others R GUIs

Many other GUIs are available: *ESS* ("Emacs Speaks Statistics") for *Emacs* editor, *JGR* (read "Jaguar"), *Rattle* for data mining, *RExcel* for using R inside *Microsoft Excel* (Heiberger and Neuwirth, 2009) (but the package "RExcelInstaller" was now removed from the CRAN repository), *Npp2R* for using R with *Notepad++* editor, …

1.3 RStudio Environment

RStudio is a free and open source integrated development environment (IDE) for R. **RStudio** is available in two editions: **RStudio Desktop**, where the program is run locally as a regular desktop application; and **RStudio Server**, which allows accessing *RStudio* using a web browser while it is running on a remote Linux server. Prepackaged distributions of **RStudio Desktop** are available for Microsoft Windows, Mac OS X, and Linux. *RStudio* is written in the C++ programming language and uses the Qt framework for its graphical user interface.

If *RStudio* is not installed on your computer, download it here and install it on your personal folder (drive "U:").

With *RStudio*, the working space is divided in 4 windows:

- Top left: R script editor
- Top right: workspace and history
- Bottom left: basic R console
- Bottom right: Files, Plots, Packages and Help.

The frame in the upper right contains your workspace as well of a history of the commands that you've previously entered. Any plots that you generate will show up in the region in the lower right corner.

The frame on the left is where the action happens. It's called the console. Every time you launch *RStudio*, it will have the same text at the top of the console telling you the version that you're running. Below that information is the prompt. As its name suggests, this prompt is really a request, a request for a command. Initially, interacting with R is all about typing commands and interpreting the output. These commands and their syntax have evolved over decades and now provide what many users feel is a fairly natural way to access data and organize, describe and invoke statistical computations. To get you started, enter the following command at the R prompt.

Why RStudio?

The RStudio project currently provides most of the desired features for an IDE in a novel way, making it easier and more productive to use R (Verzani, 2011). Some highlights are:

- The main components of an IDE are all nicely integrated into a four-panel layout that includes a console for interactive R sessions, a tabbed source-code editor to organize a project's files, and panels with notebooks to organize less central components.
- The source-code editor is feature-rich and integrated with the built-in console.
- The console and source-code editor are tightly linked to R's internal help system through tab completion and the help page viewer component.
- Setting up different projects is a snap, and switching between them is even easier.
- *RStudio* provides many convenient and easy-to-use administrative tools for managing packages, the workspace, files, and more.
- The IDE is available for the three main operating systems and can be run through a web browser for remote access.
- *RStudio* is much easier to learn than *Emacs/ESS*, easier to configure and install than *Eclipse/StatET*, has a much better editor than *JGR*, is better organized than *Sciviews*, and unlike *Notepad++* and *RGui*, is available on more platforms than just Windows.

Tab Key Completion

In the console or the script editor, we can use the tab completion feature to assist us in filling missing values or finding candidates for many different settings.

Workspace Component

The Workspace component lists the objects in the project's global workspace. In the default panel layout, this component is in the upper-right notebook along with the Files component. If this panel isn't raised, we simply click on its tab (or perform a keyboard shortcut) to do so.

Using the Code Editor to Write R Scripts

Most of the time, the commands we need are a bit too long to type in a correct way at the command line (the console). We will instead use a script file so we can freely edit our commands. *RStudio* makes it easy to evaluate lines from a script file in the console. In addition, with the aid of syntax highlighting and automatic code formatting, we can quickly identify common errors before evaluation.

Workspace Browser

The Workspace browser appears by default in the notebook in the upper-right pane of the GUI. The browser lists the objects in the global workspace, organized by type of value: Data, Values, Functions.

Clicking on a value will initiate an action to edit or view the object.

The Workspace browser has a few other features available through its toolbar for manipulating the workspace:

- Previously saved workspaces (or the default one) can be loaded through a dialog invoked by the Load toolbar button.

- The current workspace can be saved either as the default workspace (.RData file) or to an alternate file.
- The entire workspace can be cleared through the Clear All toolbar button. To delete single items, one can use the rm function through the console.

Example: Use the function “Workspace | Import Dataset | From Text File...” for opening the file housing.data.txt seen in the previous lab session.

The Help Page Viewer

R is enhanced by external code organized into packages. Packages are a structured means for the functions and objects that extend R. Part of this organization extends to documentation. R has a few ways of documenting itself in a package. For packages on CRAN, compliance is checked automatically. Each exported function should be documented in a help page. This page typically contains a description of the function, a description of its arguments, additional information potentially of use to the user, and, optionally, some example code. Multiple functions may share the same documentation file. In addition to examples for a function, a package may contain demos to illustrate key functionality and vignettes (longer documents describing the package and what it provides).

2 R Programming Language

2.1 Functions

Curly braces are used to evaluate a series of expressions (separated by new lines or semicolons) and return only the last expression:

```
{expression_1; expression_2; ... expression_n}
```

Often, curly braces are used to group a set of operations in the body of a function:

```
> f <- function() {x <- 1; y <- 2; x + y}
> f()
[1] 3
```

2.2 R Control Structure

Nearly every operation in R can be written as a function, but it isn’t always convenient to do so. Therefore, R provides special syntax that you can use in common program structures. We’ve already described two important sets of constructions: operators and grouping brackets. This section describes a few other key language structures and explains what they do.

2.2.1 Conditional Statements

Conditional statements take the form:

```
if (condition) true_expression else false_expression
```

2.2.2 Loops

There are three different looping constructs in R. Simplest is repeat, which just repeats the same expression:

```
repeat expression
```

To stop repeating the expression, you can use the keyword break. To skip to the next iteration in a loop, you can use the command next. As an example, the following R code prints out multiples of 5 up to 25:

```
> i <- 5
> repeat {if (i > 25) break else {print(i); i <- i + 5;}}
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

If you do not include a break command, the R code will be an infinite loop. (This can be useful for creating an interactive application.)

Another useful construction is while loops, which repeat an expression while a condition is true:

```
while (condition) expression
```

As a simple example, let's rewrite the example above using a while loop:

```
> i <- 5
> while (i <= 25) {print(i); i <- i + 5}
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

You can also use break and next inside while loops. The break statement is used to stop iterating through a loop. The next statement skips to the next loop iteration without evaluating the remaining expressions in the loop body. Finally, R provides for loops, which iterate through each item in a vector (or a list):

```
for (var in list) expression
```

Let's use the same example for a for loop:

```
> for (i in seq(from=5,to=25,by=5)) print(i)
[1] 5
[1] 10
[1] 15
[1] 20
[1] 25
```

You can also use break and next inside for loops. There are two important properties of looping statements to remember. First, results are not printed inside a loop unless you explicitly call the print function. For example:

```
> for (i in seq(from=5,to=25,by=5)) i
```

Second, the variable *var* that is set in a for loop is changed in the calling environment:

```
> i <- 1
> for (i in seq(from=5,to=25,by=5)) i
> i
[1] 25
```

Like conditional statements, the looping functions “repeat”, “while”, and “for” have type special, because expression is not necessarily evaluated.

If you’ve used modern programming languages like Java, you might be disappointed that R doesn’t provide *iterators* or *foreach* loops. Luckily, these mechanisms are available through add-on packages. (These packages were written by *Revolution Computing* and are available through CRAN.)

3 Graphics

With R, you can easily (Chang, 2012):

- creating a scatter plot
- creating a line graph
- creating a bar graph
- creating a histogram
- creating a box plot
- plotting a function curve
- saving your plot as a PDF file, an image or copying it to the clipboard...

3.1 Scatter Plot

For creating a scatter plot, add for example the data of the *mtcars* dataset (Motor Trend Car Road Tests) in the workspace data (the data was extracted from the 1974 *Motor Trend US magazine*, and comprises fuel consumption and 10 aspects of automobile design and performance for 32 automobiles) and print the summary of the data.

```
> data("mtcars")
> summary(mtcars)
```

Print the scatter plot of the fuel consumption of the cars on Y-axis (attribute *mpg*: (US) miles per gallon[1]) as a function of the weight on X-axis (attribute *wt*: Weight in lb/1000[2]).

```
> plot(mtcars$wt, mtcars$mpg)
```

On *RStudio*, you can easily zoom the scatter plot, and resize the window graph as you like.

To produce more attractive graphics, it is possible to install and load other packages, for example the function qplot of ggplot2 package.

```
> install.packages("ggplot2")
> library(ggplot2)
> qplot(mtcars$wt, mtcars$mpg)
```

If the two vectors are already in the same data frame, you can use the following syntax:

```
> qplot(wt, mpg, data=mtcars)
```

3.2 Line Graph

For creating a scatter plot, add for example the data of the *pressure* dataset (vapor pressure of mercury as a function of temperature) in the workspace data (data on the relation between temperature in degrees Celsius and vapor pressure of mercury in millimeters) and print the summary of the data.

```
> data("pressure")
> summary(pressure)
```

To make a line graph using plot(), pass it a vector of x values and a vector of y values, and use type="l":

```
> plot(pressure$temperature, pressure$pressure, type="l")
```

[1] 1 US miles / gallon = 0.425143707 kilometers per liter
[2] 1 US pound (lb) = 0.45359 kg

To add points and/or multiple lines, first call plot() for the first line, then add points with points() and additional lines with lines():

```
> plot(pressure$temperature, pressure$pressure, type="l")
> points(pressure$temperature, pressure$pressure)
> lines(pressure$temperature, pressure$pressure/2, col="red")
> points(pressure$temperature, pressure$pressure/2, col="red")
```

With ggplot2, you can get a similar result using qplot() with geom="line":

```
> qplot(pressure$temperature, pressure$pressure, geom="line")
```

And lines and points together:

```
> qplot(temperature, pressure, data=pressure, geom=c("line", "point"))
```

3.3 Bar Graph

For creating a bar graph, add for example the data of *BOD* dataset (it concerns the biochemical oxygen demand versus time in an evaluation of water quality) and print the summary of the data.

```
> data("BOD")
> summary(BOD)
```

To make a bar graph of values, use barplot() and pass it a vector of values for the height of each bar and (optionally) a vector of labels for each bar. If the vector has names for the elements, the names will automatically be used as labels:

```
> barplot(BOD$demand, names.arg=BOD$Time)
```

Sometimes a bar graph refers to a graph where the bars represent the count of cases in each category. This is similar to a histogram, but with a discrete instead of continuous x-axis. To generate the count of each unique value in a vector, use the table() function:

```
> table(mtcars$cyl)
```

We will see that there are 11 cases of the value 4, 7 cases of 6, and 14 cases of 8. Then simply pass the table to barplot() to generate the graph of counts:

```
> barplot(table(mtcars$cyl))
```

With the ggplot2 package, you can get a similar result using qplot(). To plot a bar graph of values, use geom="bar" and stat="identity".

```
> qplot(BOD$Time, BOD$demand, geom="bar", stat="identity")
```

We can convert the x variable to a factor, so that it is treated as discrete:

```
> qplot(factor(BOD$Time), BOD$demand, geom="bar", stat="identity")
```

3.4 Histogram

You want to view the distribution of one-dimensional data. To make a histogram, use hist() and pass it a vector of values:

```
> hist(mtcars$mpg)
```

You can specify approximate number of bins with breaks:

```
> hist(mtcars$mpg, breaks=10)
```

With the ggplot2 package, you can get a similar result using qplot():

```
> qplot(mpg, data=mtcars, binwidth=4)
```

3.5 Box Plots

A box plot (or a “box-and-whisker” plot) is a convenient way of graphically depicting groups of numerical data through their quartiles. Box plots may also have lines extending vertically from the boxes (whiskers) indicating variability outside the upper and lower quartiles. Outliers may be plotted as individual points.

If you want to create a box plot for comparing distributions, use plot() and pass it a factor of x values and a vector of y values. When x is a factor (as opposed to a numeric vector), it will automatically create a box plot.

Add for example the data of *ToothGrowth* dataset (it concerns the effect of vitamin C on tooth growth in Guinea pigs) and print the summary of the data.

```
> data("ToothGrowth")
> summary(ToothGrowth)
```

Then print the box plot:

```
> plot(ToothGrowth$supp, ToothGrowth$len)
```

If the two vectors are in the same data frame, you can also use formula syntax. With this syntax, you can combine two variables on the x-axis:

```
> boxplot(len ~ supp, data = ToothGrowth)
```

You can also put interaction of two variables on x-axis:

```
> boxplot(len ~ supp + dose, data = ToothGrowth)
```

With the ggplot2 package, you can get a similar result using qplot()), with geom="boxplot":

```
> qplot(ToothGrowth$supp, ToothGrowth$len, geom="boxplot")
```

It's also possible to make box plots for multiple variables, by combining the variables with interaction():

```
> qplot(interaction(ToothGrowth$supp, ToothGrowth$dose),
+       ToothGrowth$len, geom="boxplot")
```

3.6 Function Curve

To plot a function curve, use curve() and pass it an expression with the variable x. For example, to plot the function $y = x^3 - 5x$ in the interval $[-4; 4]$:

```
> curve(x^3 - 5*x, from=-4, to=4)
```

You can plot any function that takes a numeric vector as input and returns a numeric vector, including functions that you define yourself. Using add=TRUE will add a curve to the previously created plot:

```
> myfun <- function(xvar) { 1/(1 + exp(-xvar + 10)) }
> curve(myfun(x),from=0,to=20)
> # Add a line:
> curve(1-myfun(x), add = TRUE, col = "red")
```

With the ggplot2 package, you can get a similar result using qplot(), by using stat="function" and geom="line" and passing it a function that takes a numeric vector as input and returns a numeric vector:

```
> qplot(c(0,20), fun=myfun, stat="function", geom="line")
```

3.7 Pairwise Graph

For comparing the different attributes of a dataset, it is sometimes useful to produce a matrix of scatter-plots. This can be done in R with the pairs function.

Add for example the data of the *iris* dataset (the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris) and print the summary of the data.

```
> data("iris")
> summary(iris)
```

Then print the pairwise graph of the graph with the 4 first continuous attributes:

```
> pairs(iris[1:4])
```

By the way, it is more interesting to study the effect of each attribute on the species of the iris. We can change the color of all iris species:

```
> pairs(iris[1:4], pch=21,
>       bg = c("red", "green3","blue")[unclass(iris$Species)])
```

3.8 Plot Titles

To have labels in a plot, just add the required arguments in the plot function:

- main: an overall title for the plot
- sub: a sub title for the plot
- xlab: a title for the X-axis
- ylab: a title for the Y-axis

Example:

```
> myfun <- function(xvar) { 1/(1 + exp(-xvar + 10)) }
> curve(myfun(x),
+     xlab = "activation",
+     ylab = "threshold␣function",
+     from=0, to=20)
```

Exercise

- Open the housing.data.txt file in R.
- Depending on the nature of each attribute, plot some interesting graphs. Reminder: the dataset concerns housing values in suburbs of Boston. What is interesting is to study the potential influence of the different variables on the *MEDV* attribute (median value of owner-occupied homes in $1000's).

4 Recommended Readings

The main literature for this section is:

- Braun and Murdoch (2007), "A First Course in Statistical Programming with R,"
  Chapters 1 to 4.
- Chambers (2008), "Software for Data Analysis: Programming with R."
- Chang (2012), "R Graphics Cookbook – Practical Recipes for Visualizing Data."
- Gardener (2012), "Beginning R: The Statistical Programming Language."
- Kerns (2010), "Introduction to Probability and Statistics Using R,"

  Chapter 2 "An Introduction to R."
- Maindonald and Braun (2010), "Data Analysis and Graphics Using R –an Example-Based Approach."
- Matloff (2011), "The Art of R Programming: A Tour of Statistical Software Design."
- Verzani (2011), "Getting Started with RStudio."
- **?**, "The R language definition."
- Venables et al. (2013), "An Introduction to R,"
  Chapter 9 "Grouping, loops and conditional execution,"
  Chapter 10 "Writing your own functions."
- Torgo (2011), "Data Mining with R: Learning with Case Studies"
- Zuur et al. (2009), "A Beginner's Guide to R."

References

Braun, W. J. and D. J. Murdoch (2007). *A First Course in Statistical Programming with R*. Cambridge University Press.

Chambers, J. M. (2008). *Software for Data Analysis: Programming with R*. Springer.

Chang, W. (2012). *R Graphics Cookbook – Practical Recipes for Visualizing Data*. O'Reilly.

Gardener, M. (2012). *Beginning R: The Statistical Programming Language*. John Wiley & Sons, Ltd.

Heiberger, R. M. and E. Neuwirth (2009). *R Through Excel: A Spreadsheet Interface for Statistics, Data Analysis, and Graphics*. Use R! Springer.

Kerns, G. J. (2010). *Introduction to Probability and Statistics Using R* (1^{st} ed.).

Maindonald, J. and W. J. Braun (2010). *Data Analysis and Graphics Using R –an Example-Based Approach* (3^{rd} ed.). Cambridge University Press.

Matloff, N. S. (2011). *The Art of R Programming: A Tour of Statistical Software Design*. San Francisco, CA: No Starch Press, Inc.

Torgo, L. (2011). *Data Mining with R: Learning with Case Studies*. Chapman & Hall / CRC.

Venables, W. N., D. M. Smith, and the R Core Team (2013). An introduction to R –notes on R: A programming environment for data analysis and graphics.
URL `http://cran.r-project.org/doc/manuals/R-intro.html`.

Verzani, J. (2011). *Getting Started with RStudio*. O'Reilly.

Zuur, A. F., E. N. Ieno, and E. Meesters (2009). *A Beginner's Guide to R*. Use R! Springer.