Chapter 2
Getting Started with [R]; a Brief Introduction

Lianne Ippel

Abstract In this chapter we provide some basics into [R] that will get you started and provide you with tools to continue the development of your skills in doing analyses in [R].

2.1 Introduction

Many statistical software packages are out there (SPSS, SAS, STATA, Mplus, just to name a few), each of them has its pros and cons. However, they have one common disadvantage: they all come with a (rather expensive) price tag. In contrast, [R] is an open source programming environment and is freely downloadable. Additionally, it allows a researcher to conduct the analyses exactly in the way she wants the analyses done. Data analysis in [R] does require some programming skills, for which we provide a short introduction. For issues beyond this introduction, you will find many of your questions answered with a search in your favorite search engine.

Step one of doing your data cleaning and analysis in [R] is, of course, to get the program on your computer. You can download [R] from https://cran.r-project.org.[1] Choose the version matching your operating system, and you are good to go. For those who desire some point-and-click options in [R], several Integrated Development Environments such as 'R Studio' exist, although we do not go into them here.

A few remarks before you get started and confused: (1) [R] is a case sensitive language, so be careful with how you name your variables, $x1 \neq X1$. (2) Unlike other program languages such as *C*, which read the entire script at once before executing it, [R] reads one line at a time. [R] will continue reading until it has reached the end of

[1]Note that if you use [R] for your reports, do not forget to mention the version of [R] you used, because slightly different results may come from different versions of [R], or the packages you used. we used [R] 3.2.2, 64-bit.

L. Ippel (✉)
Department of Methodology and Statistics, Tilburg University,
Warandelaan 2, 5000 AB Tilburg, The Netherlands
e-mail: g.j.e.ippel@uvt.nl

© Springer International Publishing Switzerland 2016

J. Robertson and M. Kaptein (eds.), *Modern Statistical Methods for HCI*,
Human–Computer Interaction Series, DOI 10.1007/978-3-319-26633-6_2

a command or found something it does not understand, which will produce an error. (3) Errors tell you approximately where it went wrong. However be aware, the fact that [R] did not produce an error does not mean it has done what you intended to do, it just has done exactly what you told it to do. What [R] did and what you wanted to do, are not necessarily the same thing. In fortunate cases, [R] will produce a warning when something did not fully go according to plan. Whether or not to take these warnings seriously is dependent on the warning and what you were aiming at doing. However in unfortunate cases, [R] does not produce an error or a warning, while still not doing what you aimed to do. Therefore always check whether the results make sense. (4) If you want to prevent yourself from doing the exact same command over and over again, you won't use the *console* of [R], but rather use a *script* (File → New script). You run the script either by 'right-click → run', or 'ctrl + r'. Lastly (5) to prevent you from trying to read your code a week after you produced it and being clueless about what you tried to do: insert comments in your code. Doing so is easy: inserting a # will tell [R] to skip this line, giving you the opportunity to write down what the code should do.

The organization of this chapter is as follows. First we discuss the types of data [R] can deal with and how these data are handled in [R]. We mention some useful tools to get some insight in your data. In Sect. 2.2 we detail how you can write your own commands with the use of functions and how to incorporate code written by others. The final section might be the most important section, because it contains the [R] help manual and additional literature.

2.1.1 Data Types

[R] can handle both numerical and character input. The difference between the two is denoted by adding quotation mark(s) in case of characters, for instance:

```
> # an example of a numerical variable:
> x1 <- 10
> x1
 [1] 10
> x1 + x1
 [1] 20
>
> # an example of a string variable:
> x2 <- '10'
> x2
 [1] ''10''
> x2 + x2
 Error in x2 + x2 : non-numeric argument to binary
  operator
```

In the above example, $x1$ is a numerical variable containing a single value, 10, with which you can do calculations. However $x2$ is a string variable because we placed quotation marks around the input, instructing [R] it should treat it as characters, with which you obviously cannot do computations. [R] can do all computations (as long as it concerns numerical values) your simple calculator can do too. Besides the obvious commands ($+$, $-$, $*$, $/$, $\widehat{}$, sqrt(), exp()), $**$ can be used interchangeably for $\widehat{}$. A third data type we want to illustrate is 'boolean', a data type which is either TRUE or FALSE. Although booleans look like characters, you can do computations with them, as TRUE translates to 1 and FALSE to 0. This data type is generated when you use a logical expression, for instance to see if two variables are equal to each other:

```
> # logical expressions
> x1==10 # equal to
 [1] TRUE
> x1 <= 5 # smaller or equal
 [1] FALSE
> x1 > 5  # larger
  [1] TRUE
> x1 != 5 # unequal to
 [1] TRUE
> # an example of computations with booleans
 > (x1 == x1)+1 # TRUE +1 = 2
 [1]  2
```

2.1.2 Storage

Besides different input (number vs. character), the input can be wrapped differently. You can think of these wrappings as different storages: they vary in size and flexibility. Depending on what you want to do with your data, one or the other storage can be more efficient. There are five different storage types: vector, matrix, array, data frame, and list. They all can deal with characters or numbers. In the first example ($x1$ and $x2$) both variables were vectors with only one element. Now we focus on larger vectors:

2.1.2.1 Vectors

We start with a vector, using command c(data):

```
> x3 <- c(1,2, NA)
> x3
 [1] 1  2  NA
```

In the example above, we created a variable $x3$, which is now a vector with three elements, two knowns and one missing.[2] If there is input missing, whether it is non response in your data collection or something else, you can instruct [R] to leave a blank space using NA (Not Available).

The command 'c()' is one of the ways to create a vector. Other common used commands for vectors are:

```
> # a vector containing a sequence, is created using
> # seq(from, to, by ):
> x4 <- seq(from = 1, to = 10, by = 2)
> x4
 [1] 1 3 5 7 9
>
> # note a sequence with interval of 1:
> x5 <- c(1:5)
> x5
 [1] 1  2  3  4  5
>
> # a vector containing series of repeated elements,
> # rep(data, times, each)
> # data = what should be repeated
> # times = number the data are repeated
> # each = number a single element is repeated
> x6 <- rep(c(1:2), times=3, each=2)
> x6
 [1] 1  1  2  2  1  1  2  2  1  1  2  2
```

You can also select separate elements from a vector, as follows.

```
> # select second element of vector x3
> x3[2]
 [1] 2
> # add fourth element to x3
> x3[4] <- 4
> x3
 [1] 1  2  NA  4
> # this also works:
> x3_new <- c(x3, 5)
> x3_new
 [1] 1  2   NA  4  5
```

[2]Be aware of how you name you variables, because you do not want to overwrite a command that already exist in [R]. Simply typing in the name you want to use for your object in the console will give insight in whether this name is already in use.

As you can see from the above example, there are often multiple ways which yield the exact same result. For small vectors, different ways of adding data and/or selecting elements do not really make a difference, however once you start working with large data sets (like data frames and lists) it pays to check for the fastest option.

2.1.2.2 Matrix

An extension of a vector is a matrix, which unlike a vector which is unidimensional, has two dimensions: rows and columns.

```
> x7 <- matrix(data = c(1:4), nrow = 2, ncol = 2)
> x7
      [,1] [,2]
 [1,]    1    3
 [2,]    2    4
```

Basically, what this command does is: you give [R] a vector and instruct [R] to break it down into the number of rows and columns. When the matrix has more cells than the number of elements in the provided vector, [R] will fill up the matrix, starting from the beginning, without error or warning!

```
> x8 <- matrix(data = c(1:3), nrow = 3, ncol = 2)
> x8
      [,1] [,2]
 [1,]    1    1
 [2,]    2    2
 [3,]    3    3
```

So we cannot repeat this too often, check whether [R] did what you wanted to do even, or maybe especially, when no error or warning was produced. One way to do so is check whether the elements are positioned the way you expected. Selecting or adding data from a matrix is very similar to a vector, with the minor difference of having to specify two dimensions. In line with algebraic rules, [R] will take the first input as row number and the second input as column number: [1,1] will select top left element, while [2,1] will select the element on the second row, first column. When you leave the first dimension open, [,1], the entire column is selected or when you leave the second dimension open, [1,], the entire row is selected.

```
> # select element on second row, first column of x7
> x7[2,1]
 [1]  2
> # select second row
> x7[2,]
```

```
 [1]  2  4
> # select second column
> x7[,2]
 [1]  3  4
> # add row to x7 using rbind(data, new_row):
> # rbind binds the new row to the data
> x7_new_row <- rbind(x7, c(1,2))
> x7_new_row
      [,1] [,2]
 [1,]    1    3
 [2,]    2    4
 [3,]    1    2
> # add column to x7 using cbind(data, new_column)
> # cbind binds the new column to the data
> x7_new_column <- cbind(x7, c(5,6))
> x7_new_column
      [,1] [,2] [,3]
 [1,]    1    3   5
 [2,]    2    4   6
```

2.1.2.3 Data Frame

A data frame is a special case of a matrix. It also consists of two dimensions, with the cases stored in the rows and the variables in the columns. The example dataset on fruit-based smart watch performance provided as supplementary material is a data frame, which will be used throughout the book to explain multiple analyses. A data frame is created as follows:

```
> x9 <- data.frame(id=factor(c(1:3)), obs=c(10:12))
> x9
   id obs
 1  1  10
 2  2  11
 3  3  12
```

where the first column (without heading), denotes the row numbers, the second column is a nominal variable of the data frame (in this case labeled 'id'). Nominal variables can be created using the command 'factor()'. The third column is a numerical variable (labeled 'obs').

Because a data frame is a special case of a matrix, selecting and adding data can also be done similarly. However, because it is a special case and not exactly the same, it can also be done differently. The advantage of a data frame is that the columns have labels, which you can use to select elements or columns or add variables.

```
> # select element on second row, first variable of x9
> x9$id[2]
 [1] 2
 Levels: 1 2 3
> x9[2,1]
 [1] 2
 Levels: 1 2 3
> # or select a case with a particular number,
> # convenient when row numbers != id numbers:
> x9[x9$id==2,]
   id obs
 2  2  11
> # select variable
> x9$obs
 [1]  10  11  12
> # add variable
> x9$new_var <- c(20:22)
> x9
   id obs new_var
 1  1  11      20
 2  2  12      21
 3  3  13      22
```

2.1.2.4 Array

Next we turn to the two larger storages which have a flexible number of dimensions. First an 'array' which is an extension of a matrix, and can have many dimensions. An array with only two dimensions is equivalent to a matrix.

```
> # array(data, dim = c(rows, columns, slices, etc.))
> x10 <- array(data = c(1:4), dim = c(1, 3, 2))
> x10
 , , 1

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

 , , 2

      [,1] [,2] [,3]
 [1,]    4    1    2
```

Again we see that [R] fills up the array, starting from the beginning. Variable $x10$ has 1 row (first entry of the dim argument), 3 columns (second entry of the dim argument), and 2 slices (third entry of the dim argument). The only downside of an array is that you have to keep in mind how many dimensions you have and which dimension is which. Of course we could continue and add more dimensions but for the sake of clarity we stop at three dimensions. We skip adding and selecting elements from an array, because it is similar to a matrix.

2.1.2.5 List

The last storage is a 'list', which is different from the other storages in the sense that a list wraps storages. For instance, a list can contain a vector in the first cell, a data frame in the second, an other list in the third and so on. This makes a list very flexible and complicated at the same time. A list can be very convenient as an output of your analyses, storing all results in one place, however selecting one single number from a large list might end up challenging.

```
> # list(cell 1, cell 2, cell 3, etc.)
> x11 <- list(scalar=x1, vector=x3, array=x10, c(1,2))
> x11
 $scalar
 [1] 10

$vector
 [1]  1  2 NA

$array
 , , 1
      [,1] [,2] [,3]
 [1,]    1    2    3
 , , 2
      [,1] [,2] [,3]
 [1,]    4    1    2

[[4]]
[1] 1 2
```

Note that you can label the different cells, but it is not necessary to do so. You label the cells by adding the label before the object you stored in the cell: scalar = $x1$ labels the first cell as *scalar*. As mentioned above, selecting an element from a list is somewhat odd in the sense that adding or selecting an element depends on what you have stored in the list and whether you have labeled the different cells. If you have labeled the cells you can select the cell just like you would select a variable in a data frame, if you did not you select a cell using double squared brackets:

```
> # select second cell from list x11
> x11$vector
 [1]  1  2  NA
> x11[[2]]
[1]  1  2  NA
```

Selecting an element within a cell then depends on which storage is in the cell, and it would be repetition of the text above to illustrate how each of them work.

2.1.3 Storage Descriptives

As mentioned above, it is very important to inspect the result [R] produced to check whether it actually did what you intended. Sometimes looking at the entire object at once is cumbersome. [R] has some tools that make a first inspection whether your object looks like what you expected:

- class(x, …): x is an arbitrary object. This function tells you which storage type is between the brackets
- str(x, …): x is an arbitrary object. This function tells you whether x contains strings or numbers
- summary(x, …): x is an arbitrary object. This function returns, depending on the kind of object you want the summary from, statistics such as number of cases, averages, standard deviations, etc.
- length(x): gives the length of the object x, usually x is a vector, but it works for other objects as well.
- dim(x): x is anything but a vector. The function gives the dimensions: first number of rows, then number of columns etc.
- nrow(x): x can be a vector, matrix, array or data frame. The function returns the number of rows
- ncol(x): like nrow, however this function returns columns

Note that some function have '…' and others do not; these dots imply that additional arguments could be included in the function.

2.2 Working with [R]

2.2.1 Writing Functions

Now you have some sense of how data looks like in [R], you of course want to do something with these data. The tools to work with are called functions. Without being explicit about it, we already came across many functions. All the commands we have used so far, whether it is to make a vector (c()), or to get some summary about an object (summary()), these are functions integrated in [R] already. You can,

and most likely will, write functions yourself as well. Writing functions is done as follows:

```
> test <- function(argument_1, argument_2, ...)
+ {
+    actions
+    return()
+ }
>
> result <- test(argument_1 = X, argument_2 = Y)
```

You define a function (in this case: test) and you provide the function with arguments. A (very simple) example could be

```
> multiplier <- function(input, times)
+ {
+    local_result<- input*times
+    return(local_result)
+ }
>
> global_result <- multiplier(input=5, times=2)
> global_result
 [1] 10
> # note local_result is defined within the function
> # therefore it doesn't exist globally:
> local_result
 Error: object'local_result' not found
```

Note that, although we indent the code, this is not required. Indentation improves readability, but it does not have any other function in [R].

Sometimes you only want your function to perform the action if a certain condition or conditions are satisfied, for instance if the result of the multiplier is zero, you might want to add 1:

```
> multiplier_not_zero<- function(input, times)
+ {
+   local_result<- input*times
+   if(local_result==0)
+   {
+     local_result<- 1
+   }
+   return(local_result)
+ }
> multiplier_not_zero(input = 3, times = 0)
 [1]  1
```

When there are multiple conditions you want to be satisfied before your function should run you can make combinations using the "and" and "or" operators which are written as "&" and "|" respectively. For example:

- | : if($x1 <= 1 | x1 >= 15$) { action }: or-statement
- & : if($x1 >= 1 \& x1 * x1 < 10$) { action }: and-statement

2.2.2 Data: Input and Output

2.2.2.1 Loading Data into [R]

Next, we discuss how you get your data in [R] to handle them. How to get your data into [R] depends on the format in which the data is stored. Most data formats (*.txt, *.RData, *.csv) can be easily included:

```
> # *.txt file: are there variable names: header=T
> # what separates different values? sep=''
> my_dat1 <- read.table('directory/filename.txt',
+ header, sep='')
> # *.RData file
> my_dat2 <- load('directory/filename.RData')
> # *.csv file
> my_dat3 <- read.csv('directory/filename.csv',
+ header, sep='')
```

For files with other extensions, you might need an additional package to load the data, for instance the package 'foreign' or 'Hmisc' for SPSS and SAS.

To prevent yourself from typing in the same directory repeatedly and having very long calls for your data, you can also set (and get) your working directory as follows:

```
> setwd('the/directory/you/want/to/work/in')
> getwd()
 [1] 'the/directory/you/are/working/in'
```

Knowing in which directory you are working saves you an elaborate search in all your files and folders when you saved an [R] object, which can be done as follows:

```
> # *.txt file
> write.table('directory/filename.txt', header,
+ sep='')
> # *.RData file
> save('directory/filename.RData')
> # *.csv file
>  write.csv('directory/filename.csv', header, sep='')
```

2.2.2.2 Simulate Data

When you do not have any data but you do want to practice with [R], or more likely, you want to test a new method it is useful to create data. [R] has plenty functions to generate (random) data. We put random between brackets because [R] will never provide you with fully random data, simply because your computer has a set of rules to create these data and therefore it cannot be completely random. Although this is an interesting topic, the point we want to make is that you can get the exact same 'random' data by fixing the begin point of the algorithm which creates the data using 'set.seed(number)'. For instance, if you want 3 draws from a normal distribution with mean = 10, and standard deviation = 2:

```
> set.seed(64568)
> rnorm(n=3, mean=10, sd=2)
[1] 10.468961  8.574238  9.754691
```

You will see that if you insert this code that you will have the exact same three numbers. Besides generating data using 'rnorm()' you can get other distributional information about the normal distribution using either

- dnorm(x, mean, sd): x is a scalar, the function returns the density of normal distribution at x
- pnorm(x, mean, sd): like above, though this function returns the lower tail probability
- qnorm(x, mean, sd): opposite of the above one: x is the probability and the function returns the quantile belonging to probability x

Similar functions exist for many distributions. Because different distributions require different parameters, the arguments within the function differ, though the idea is similar.

2.2.3 For Loop

One of the tools which is common in many program languages is the for loop. The for loop in [R] is a simple function to go line by line through the data. Because it is a simple function it makes life easy when doing for instance simulations, however it also makes life slow. The for loop in [R] has the downfall that it can be rather slow when working with complex computations in combination with large datasets. In case of complex computations, you might want to look into the plyr package, which has some integrated functions which also perform computations on every line of data, though do it more quickly. For now, let us have a look at the for loop.

The function works as follows:

```
> # for loop example
> # if you want to store the result
> # you have to define the result outside the for loop:
> row_average <- c()
> for(i in 1 : nrow(x7))
+ {
+   row_average[i] <- mean(x7[i,])
+   print(row_average[i])
+ }
 [1]  2
 [1]  3
> row_average
 [1]  2  3
```

First note that you should store the result of the computation performed by the loop in a variable which will not be over-written. Second, objects defined within the for loop are accessible outside the local scope of the for loop. Apart from the fact that this is just a very simply example to illustrate for loops, the same result could have been obtained alternatively by

```
> (row_average <- rowMeans(x7))
 [1]  2  3
> # putting brackets around an assignment
> # tells [R] to print the object
>
> # example of one of plyr functions
> adply(.data=x7, .margins=1, .fun=mean)
   X1 V1
 1  1  2
 2  2  3
```

where the adply function returns a data frame with $X1$ the variable which indicates the row numbers and $V1$ being the averages per row.

2.2.4 Apply Function

An example of a function which is already more efficient but also more advanced than the for loop is the 'apply' function. It is incorporated in the [R] base, so no additional

packages are required. The function loops through either arrays or matrix-like objects (as long as it has more than one dimension). The function works as follows:

```
> # example of apply function
> # apply(array, margin, function)
> apply(x10, 1, mean)
[1] 2.166667 # mean of the 2 rows
> apply(x10, 2, sum)
[1] 5 3 5 # sum of each column
> apply(x10, 3, FUN=function(x)
+ {return(x*2)})
      [,1] [,2]
 [1,]    2    8
 [2,]    4    2
 [3,]    6    4
> # per slice the elements are multiplied by 2,
> # column 1 = slice 1, column 2 = slice 2
```

The apply function can deal with many preprogrammed functions. You can also write your own functions. When you get the hang of these functions, you also might want to look of variations of the apply function (among others):

- mapply(function, arguments): multiple argument version of apply,
- lapply(x, function): like apply, but it returns a list as result, of the same length as the array you put in,
- sapply(x, function): an easier function which does the same as lapply,
- tapply(x, margin, function): applies a function to each cell of a ragged array.

2.2.5 Common Used Functions

You do not have to program every single function you can think of yourself, because many of the simple descriptives are already included in [R], for instance:

- mean(x, …): x is a vector (other storages will be vectorized), the function returns arithmetic mean,
- sd(x, …): like the previous, returning the standard deviation,
- var(x, …): like the previous, returning the variance,
- cov(x, …): x consist of two dimensions, the function returns the covariance,
- corr(x, …): like the previous, returning the correlation,
- table(x, …): this function returns a frequency table
- max(x, …): x is an arbitrary object, the function returns highest value,
- min(x, …): like the previous, returning the lowest value.

among many more. Fill in your object of interest between the brackets and [R] returns the answer in no time. Besides these numerical descriptives of your data, [R] allows you to inspect your data graphically with different kind of plots:

- plot(x, …): add type='l' such that a line will connect the data points
- lines(x, …): add more lines to your plot
- points(x, …): add more data points to your plot
- hist(x, …): histogram
- boxplot(x, …)
- barplot(x, …)

Even many analyses are ready-to-use in [R], so no additional programming is required to do:

- anova(x, …): x containing the results returned by a model fitting function (e.g., lm or glm). The function can do both model testing as well as model comparisons.
- glm(formula, data, …): to fit generalized linear models
- lm(formula, data, …): to fit linear regression
- princomp(formula, data, …): to do principle component analysis
- t.test(x, …): the function wants at least one vector, additional arguments can be included to test one or two sided etc.

2.2.6 Packages

When you want to do some analysis which is not included in [R] base, you either have to program it yourself or see if others have done it before you (and made it publicly available). If the latter is the case you can include this code as follows:

```
> install.packages('plyr')
```

which will install the plyr package, which we already discussed above. What happens next is a pop-up window to select the server it should download the package from. Select your country (or something close) to complete the installation of the package. In order to use the package you have to attach the package to your working environment as follows:

```
> library(plyr)
```

Each time you open [R] you do have to attach packages again, but you do not have to install them every time. Thus, you do not want to put 'install.packages()' in your script (but in the console), however you do want to include 'library()' in your script so when you run your entire script it will automatically load the packages.

2.2.6.1 Useful Packages

There are a lot of packages available, not all of them of very good quality. Here we shortly list packages that are useful and/or used throughout the book

- digest: allows users to easily compare arbitrary [R] objects by means of hash signatures
- directlabels: adds nice labels to (the more fancy) plots
- foreign: allows you to include data files from other software programs such as SPSS
- GGally: to make a matrix of plots produced by ggplot2
- ggplot2: to make pretty plots
- gridExtra: to arrange multiple grid-based plots on a page
- lattice: to make pretty plots
- lme4: to do multi-level analyses
- MASS: functions and datasets to support Venables and Ripley (2002)
- plyr: for faster for loops
- poLCA: to perform latent structure analysis
- psych: for multivariate analysis and scale construction using factor analysis, principal component analysis, cluster analysis and reliability analysis
- reshape2: to transform data between wide and long formats
- xtable: export tables to LaTeX or HTML

2.3 Mastering [R]

This is of course a very short introduction in [R] which allows you to do the very basics of data analysis and hopefully understand what the authors of the following chapters are doing. There is only one way to truly learn [R], which is hands on. Practice (eventually) makes perfect, so do not be discouraged when you are faced with many errors, warnings or unexpected results. There is an extensive help function in [R]: if you do not know how a function works, you can get information by putting it within the help function or, put one or two question marks before the name of the function.

```
> help(plot)
> ?plot
>??plot
```

One question mark will provide you with the web page with information about the particular function including examples. Two question marks will give you an overview of closely related topics. When these pages do not provided you with the information you need or understand, there is also a large community of [R] users, which have answered many questions at the many forums out there. Do not be afraid

to plug your error, warning, or problem in a search engine on the Internet, because you will be amazed about the amount of information, examples and ready-to-use solutions that is out there, whether you have beginner questions or more advanced problems.

2.3.1 Further Reading

Below we listed some readings, which are either focused on introducing and working with [R], or on statistics using [R], both have proven to be useful.

- http://cran.r-project.org/doc/manuals/R-intro.pdf: A users guide to [R] in which the topics covered in this chapter are discussed in more details, including some code to work with
- http://cran.r-project.org/doc/contrib/Torfs+Brauer-Short-R-Intro.pdf: This article has besides what is mentioned in this chapter, an additional overview of some integrated functions
- Discovering statistics using R by Field et al. (2012): This is more a stats book than a [R] manual, but it does what you expect: it explains statistics in [R]
- Bayesian computation in [R] by Albert (2009): Similar story to the above one, though dealing with Bayesian analysis
- Introduction to Applied Bayesian Statistics and Estimation for Social Scientist by Lynch (2007): different angle, similar in content to the above one

References

Albert J (2009) Bayesian computation with R, 2nd edn. Springer
Field A, Miles J, Field Z (2012) Discovering statistics using R. Sage, London
Lynch SM (2007) Introduction to applied Bayesian statistics and estimation for social scientist. Springer
Venables W, Ripley B (2002) Modern applied statistics with S, 4th edn. Springer

http://www.springer.com/978-3-319-26631-2

Modern Statistical Methods for HCI
Robertson, J.; Kaptein, M. (Eds.)
2016, XX, 348 p. 77 illus., Hardcover
ISBN: 978-3-319-26631-2