Statistics in R

First steps

Joost van de Weijer and Dylan Glynn
Lund University

1. Installing R

R is freely available from www.r-project.org. On the site, one will find the CRAN server link where the various versions (LinuX, MacOSX, and Windows) can be downloaded. Follow the instructions to install R onto your personal computer.

2. Commands

Once R has been installed on your computer, you start it by double-clicking the program's icon. A window should open entitled "R Console". In this window, you will see some introductory text, followed by an empty line that starts with a “>” sign. This sign is called the prompt. Unlike many other programs, R does not have a graphical interface with drop-down menus or dialogue windows to do an analysis. Instead, you type directly in the console window what you want the program to do. The things that you type here are called commands or functions, many examples of which are given throughout this book. Commands can be simple or complex. An example of a command to read in a data file into R is given below. This command itself will not work since the location of the file is not specified, but it serves to explain the principle behind using command line.

```
mydata = read.table("dataframe.txt")
```

For many users new to R, working with commands is difficult at first. Commands have a very strict syntax. If they are not entered entirely correctly, they do not work. The source of the error can be small (a missing comma, for instance) and the resulting error message is usually not very helpful. Furthermore, many commands exist, which makes it difficult to remember them. Nevertheless, writing and memorizing commands gets easier through practice. A helpful strategy is to keep and maintain a personal collection of command examples that you have used before and which are useful for the kinds of analysis that you do.

Let us examine the above command line, identifying its constituent elements. The central element is the word `read.table`. This is one of many commands that are built in into R. And it is used for importing external data. We will see some more examples of the `read.table` command below. A command, such as `read.table`, is a way of telling the program to do something. What the program needs to do is specified by the keyword, here importing data. Next, note that `read.table` is followed by an opening parenthesis and, a bit further on at the end, a closing parenthesis. The text within the parentheses is the name of a data file, enclosed in quotation marks. Very often, the action that is specified by the command is performed 'on something'; here, the data file which is stored on the computer. This part is called the argument of the command. Note that the name of the file is put in parentheses.

Then notice the part to the left of the `read.table` keyword in the example above. The result of the action performed by the command is saved internally in R under the name `mydata`. Practically, this means that the content of the external data file is copied, and saved in R under a new name. This name is more or less arbitrarily chosen. There are some restrictions on the choice of names, but almost anything will do, as long as it starts with a letter and does not contain any special characters. The name could as well have been `olddata`, `sunday2112`, `pilotstudy`, or `xxx`. It is up to you to choose the name, choose something short and easy to remember.

Things that are saved in R are usually referred to as 'objects'. Once you have created an object, it will be available until you quit the program, or until you delete it yourself. An object can be a single number, a series of numbers, a complete data file, a graph, the output of an analysis, and so on. These are all stored in the so-called workspace. In order to see a listing of what is in the workspace, you can type the command `ls()`.

2. The data file

Creating a data file that will be loaded into R is an important step and one that often leads to confusion when first learning. The idea is to take your data, from whatever their source, and put them in a plain text file. This step needs extreme care since text files can often include hidden formatting and other information that will prevent the data being loaded. Typically, your data are held in a spreadsheet file (such as those produced by MS Word's Excel, Open Office Calc and iWork's Numbers) or a database (such as those produced by MS Word's Access, Open Office's Base, and Apple's FileMaker). When just beginning, the tabular layout of a spreadsheet is arguably the easiest. This is because when you display the data in a spreadsheet, you can easily see whether all cells are complete, whether the columns are aligned properly, how many cases there are, etc.

If you are using a spreadsheet, once the results look good, you can copy and paste the data directly into a plain text file, making sure to use a text editor that strips it of formatting. It is important that the file itself does not contain any 'formatting'. If the file contains formatting or invisible markup, R will not be able to read the file. There are many text-editing programs that can contain hidden formatting (such as WordPad for Windows and TextEdit for MacOSX). However, other text editors will automatically 'strip' any formatting, hidden or otherwise (such as NotePad in Windows and TextWrangler in MacOSX). If the data are contained in a database, you need to export the data to a plain text file. This option is also available in the spreadsheet programs. Exporting your data will avoid the problem of hidden formatting and mark up.

The data in the text file, whether taken from the database or a spreadsheet, will normally be in one of two formats. The first is the flat dataframe format, illustrated in Table 1:

Table 1. Example of a flat dataframe

verb	tense	person	figurativity
run	past	1	literal
run	past	1	metaphor
jog	future	2	metaphor
run	past	2	literal
skip	past	2	literal
jog	future	1	literal

If the columns are not perfectly straight, this does not necessarily mean there is a problem - remember, there is no formatting. Normally, the data will be tab delimited (the columns separated by a tab space). This is the default in most spreadsheets and databases. R will read any type of delimitation, but the commands below assume tab delimited data. The flat dataframe layout of the data normally results from the manual analysis of examples in a database or spreadsheet. In this format, the numbers of occurrences are not indicated, but instead each occurrence is listed in a large ‘flat’ file.

The second typical format for data is a numerical tabulation. We will refer to this as a cross-tabulation or a contingency table. In contrast to the flat dataframe, this format is a numerical summary of the data and is typically a result of manually counting occurrences or the results of questionnaires. However, the format is also generated by spreadsheet, database, and concordance programs as well as other corpus utilities. An example of a cross-tabulation is shown in Table 2.

Table 2. Example of the cross-tabulation of a contingency table

	future	past	1stPers	2ndPers	literal	metaphor
jog	2	0	1	1	1	1
run	0	3	2	1	2	1
skip	0	1	0	1	1	0

Using the first column in Table 1 as the row names, the data in Table 2 are the equivalent to those above. It is important to note that this table would look considerably different if we were to take a different column in Table 1 and use it to create the row names. This data format is sometimes referred to as a contingency table, frequency table, cross-tabulation, xtab, or pivot table.

3. Importing the data into R

3.1 Importing data from a flat dataframe

The most common command for importing data into R is `read.table()`, which we saw earlier in this text. Here we present this command again, but now with two small additions.

```
mydata = read.table("dataframe.txt", header=TRUE, sep="\t")
```

In this example, there are three arguments to the command rather than just one in the earlier example. The arguments have been separated by commas. The first argument, `"dataframe.txt"` is the filename, as we already saw above. The second argument, `header=TRUE`, indicates that the first line of the data file contains the variable names, as is the case in Table 1. If this argument had been omitted, then the fields in the first line of the data file would have been interpreted as values rather than as names, and the columns would have been labelled with automatically chosen names instead (`V1`, `V2`, etc.). The third argument, `sep="\t"`, indicates that the columns are separated by tabs. Had the columns been separated by commas instead, for instance, then the third argument would have been `sep=","`.

However, these commands will still not work – we have to add one further piece of information. We need to tell R where to find the data file. There are three possibilities:

```
mydata = read.table(file.choose(), header=TRUE, sep="\t")

mydata = read.table("clipboard", header=TRUE, sep="\t")

mydata = read.table("users/linguist/data/dataframe.txt”,
           header=TRUE, sep="\t")
```

In the first alternative, the name of the file has been replaced by `file.choose()`. This argument causes a dialogue window to open from which the data file can be chosen. With the second alternative, it is assumed that the data have been copied to the clipboard and R takes the data from there. In other words, this is similar to the copy and paste function common in applications such as MS Word and Excel. The third option tells R to go to a specific location and open the file. That location can be on the hard drive of a personal computer, on a server, or even on the Internet. The location is indicated as a "path", where slashes "/" represent folders. This location or path needs to be between inverted commas. In MacOSX and LinuX, the command given above indicates that the data file is in a folder called 'data', which is in a folder called 'linguist', which in turn is in a folder called 'users', which is on the boot drive of the system. In Windows, one needs to add the name of the disk or volume typically indicated by a lowercase letter followed by a colon, "c:" being the default label for a boot drive. Therefore, for Windows, the equivalent command line would look like this:

```
mydata = read.table
    ("C:users/linguist/data/dataframe.txt”,
```

```
    header=TRUE, sep="\t")
```

The object that is being created with `read.table()` is called `mydata`. It is important to check that the dataframe created from a data file has been properly imported into R. It is not uncommon that something goes astray during the process of importing, and not always does this result in an error message in R. Below, we provide you with some commands that can help you make sure that the newly imported data is in order. The simplest way of seeing what the data looks like in R is by typing the name of the object. We called our data object `mydata`.

```
mydata
```

This will display the contents of the entire object. However, if the object is a flat dataframe, this will result in the entire dataset being displayed, which will not be helpful. In the case of a flat dataframe, a better option is to look at the first few rows, using the command `head()`:

```
head(mydata)
```

This command will display the column names of the dataframe (these are the names that were in the first row of the data file; the headers in a spreadsheet and the cell labels in a database), followed by the first six rows. There is a corresponding command, `tail()`, that displays the last six rows of the dataframe.

A second command that is useful in this regard is `summary()`. This command generates a numerical summary of the dataframe. This is also useful for spotting spelling mistakes and so forth in your analysis. Remember that R treats lowercase and uppercase letters as distinct items and does not ignore invisible characters such as a space or a tab. It is rare that a flat dataframe is without any mistakes.

```
summary(mydata)
```

A third command that provides information about a dataframe is the command `str()`. This command offers information about the structure of a dataframe. When applied to the data from Table 1, we receive the following output:

```
str(mydata)
```

```
'data.frame': 6 obs. of  4 variables:
$ verb        : Factor w/ 3 levels "jog","run","skip": 2 2 1 2 3 1
$ tense       : Factor w/ 2 levels "future","past": 2 2 1 2 2 1
$ person      : int  1 1 2 2 2 1
$ figurativity: Factor w/ 2 levels "literal","metaphor": 1 2 2 1 1 1
```

The first line of the output shows that `mydata` is a dataframe with six observations (rows) and four variables (columns). The next four lines show the four variables and three types of information about them. First of all, they show their names. Second they show what type of variables they are. Here, three of the four variables are labelled as `Factor`, which is the usual variable type for categorical variables. Additionally, the `str()` command tells you how many levels these variables have, that is, the number of values that these variables take. The variables *tense* and *figurativity* have two levels, while *verb* has three. The fourth variable, *person*, is labelled as `int`, which means that the values of that variable are whole numbers. The numbers `1` and `2` are labels for first and second person, respectively.

3.2 Importing cross-tabulations

If your data are in a cross-tabulated format, exemplified in Table 2, loading the data is a little different. There are two ways to load the data in this case. Firstly, one can repeat the command used above to load the data from a flat dataframe, but add the argument `row.names = 1`.

```
mydata = read.table(file.choose(), header=TRUE, sep="\t",
   row.names=1)
```

The read.table command is not designed for data in this format, so R does always not treat the data as it should (for example, the `str()` and `summary()` commands do not work). Nevertheless, for most purposes, loading the data in this way does not pose any problems. If you wish to see that the cross-tabulation has been correctly loaded, enter the object name:

```
mydata
```

This brings up the cross tabulation for inspection. A second, more orthodox, way to load a cross-tabulation is to use the command `read.ftable()`.

```
mydata = read.ftable(file.choose(), sep="\t")
```

This command tells R that the data are in the cross-tabulated format. Note that the `header=True` argument has been removed. The `read.ftable` command is sensitive to the actual layout of the data, especially to how the row and column names are placed in the text file. For this command to work, the name of the first row must be located on a previous line, independent from the column names. One must also remember to add a 'blank' first column beneath the first row-name. Table 3 exemplifies the layout.

Table 3. Layout for `read.ftable()` command

verb						
	future	past	1stPers	2ndPers	literal	metaphor
jog	2	0	1	1	1	1
run	0	3	2	1	2	1
skip	0	1	0	1	1	0

Both commands, `read.table` and `read.ftable()`, expect a blank line (a return carriage) at the end of the text file, beneath the table.

Transpose a contingency table

Sometimes it is useful to transpose a cross-tabulation. Transposition means that the entire data object is rotated by 90 degrees. That is, the rows become columns and the columns become the rows. For a flat dataframe, this is rarely useful, but for cross-tabulations, since the data are summarised as a given variable and its levels are relative to another variable (or variables and levels), inverting the table means that a statistical technique may actually examine the data from a different perspective. Transposition can be done in R using the command `t()`, as in the following example:

```
mydata2 = t(mydata)
```

The object `mydata2` is now an inverted or transposed version of the of the original cross-tabulation.

4. Making changes to data in R

It is not uncommon that information in a dataframe needs to be changed or that new information needs to be added. A possible way of doing this is to make changes directly in the data file and to re-load the data, but for various

reasons it might be desirable to keep the data file unchanged, and to do the modifications to the dataframe in R instead.

Creating objects

To begin, we need to understand the principle of creating objects in R. Two synonymous signs are used for this, "=" and "<-". The signs can be used interchangeably. The item that is to the left of the sign is the new object, and it equates whatever is to the right. In the description above, we created an object `mydata` using this sign. This object is stored in the live memory of your computer and remains there until one quits R or one removes the object with the command `rm()`. For example, when changing your data in R, one option is to make a duplicate of your data. Of course, if something goes wrong, one can always re-load from the data file. Moreover, having too many data objects in R uses up precious memory. However, to help us understand how one creates objects, let us duplicate our data:

```
mydata.copy = mydata
```

Now there are two identical copies of the data in R, one labelled `mydata`, the other labelled `mydata.copy`. You do not normally use this functionality to duplicate data, but to create a new object; for example, the results of a statistical analysis, which you wish to store in order to plot or to run further analyses upon.

Working with large datasets or running many analyses can result in sluggish performance from your computer since all this information is stored in the active memory. Therefore, objects not in use should be removed. To remove an object, we type `rm()`. The following line removes the duplicated data.

```
rm(mydata.copy)
```

Changing a variable name

The name of one or more variables in a dataframe can be changed using the command `colnames()`. In order to do this, you must also know the number of the column with the name that needs to be changed. In the example dataframe, there are four columns. To see the names of all four columns, type:

```
colnames(mydata)
```

If we would want to change the name of a column, say the first one, type
```
colnames(mydata)[1] = "newname"
```

Changing a variable type
A common manipulation is to change the variable type. An example is when the levels of a categorical variable have been coded as numbers, as was the case with the variable *person* in Table 1. The levels of this variable have been coded as `1` and `2` for first and second person, respectively. Automatically, this variable was imported as an integer variable. As a consequence, certain commands that are appropriate for a categorical variable do not work (e.g., the `levels()` command) if no action is undertaken. More importantly, this variable is not numerical, it is categorical; the example sentence was either in the 1st person or the 2nd person. Letting R assume it is a numerical variable will lead to errors in any subsequent statistical analysis. The solution is to change the variable type for `person` from `integer` to `factor` using the `factor()` command:

```
mydata$person.fact = factor(mydata$person)
```

Note that we gave the transformed variable a new name (`person.fact`), which automatically created a new variable in the dataframe. While it is possible to keep the original name after transformation, there is a risk involved of inadvertently running the same transformation once again, and thus doubling the effect of the transformation. This risk is avoided by using a new name for the transformed variable.

Changing values in a dataframe
There are many situations where values need to be changed. These may be values that were not entered correctly in the first place, values that need to be rescaled, and so on. Here we illustrate how to change values in a dataframe by first adding a new variable to `mydata` that divides the verbs into two groups, and then changing some of the values of that new variable. Suppose, for instance, that we would like to add a variable to `mydata` that indicates verb type, and that the verbs *jog* and *run* belong to type `"A"` while *skip* belongs to type `"B"`. Here we do that in two steps. In the first step we create the new variable, call it `verbtype`, and assign it the value `"A"` for all cases in the dataset:

```
mydata$verbtype="A"
```

In the second step, we change the values of `verbtype` into `"B"`, but do that only for the cases where the variable `verb` equals *skip*. This can be done using square brackets (`[]`) notation. In R, square brackets are often used to identify the position of a value, as we saw above with the `colnames()` command. To find a value in a dataframe, we need to specify the row(s) that contain a specific value, and the column(s). In other words, we need to include two things within the square brackets: the `row(s)` and the `column(s)`. Within square brackets, the row is always specified first, followed by a comma, followed by the specification of the column. Schematically:

```
[which row(s)? , which column(s)?]
```

In our example, the changes need to apply only to the rows where the variable `verb` equals *skip*, and only in the column that contains the variable `verbtype`:

```
mydata[mydata$verb=="skip","verbtype"]
```

If we give this as a command to R, we get the value of the variable `verbtype` for the cases where the variable `verb` equals *skip*. Now this value is still `"A"`, but it needs to be changed to `"B"`. Setting the new value is easy:

```
mydata[mydata$verb=="skip","verbtype"]="B"
```

We can check the result by typing the name of the dataframe:

```
mydata
```

which shows:

```
  verb  tense person figurativity verbtype
1  run   past      1      literal        A
2  run   past      1     metaphor        A
3  jog future      2     metaphor        A
4  run   past      2      literal        A
5 skip   past      2      literal        B
6  jog future      1      literal        A
```

Learning to use the square brackets notation is very helpful when working with R. It is a flexible way of manipulating dataframes, or getting information out of a dataframe.

Creating a subset of the dataframe
It happens frequently that certain cases need to be excluded from an analysis. These may be cases containing missing values, outliers, or cases with incorrect values. In R, there are several ways of creating a subset of a dataframe. Here we present one of them, namely using the `subset()` command. Suppose, by way of illustration, that we would like to exclude all cases for the verb *to skip* from `mydata`. We could establish this by telling R

```
subset(mydata,mydata$verb!="skip")
```

where the operator `!=` means *is not equal to*. Note furthermore, that the quotes around the word `"skip"` are obligatory.

If, on the other hand, we would like to restrict the dataset to the verb *skip* only, then we would write:

```
subset(mydata,mydata$verb=="skip")
```

In this second example of the `subset()` command, pay attention to the double == sign. This is also obligatory, or the command will not work.

Merging dataframes
Two dataframes can be merged in two ways, either the second dataframe is appended below the first one, or it is added next to it. In the first merge, we add new cases to the first dataframe; in the second we add new variables. Merging two dataframes is easy, as long as the two dataframes match. If we append one dataframe to the other one, both dataframes need to have the same number of columns, and the column names need to be identical. The command for appending one dataframe to another one is `rbind()` ('row bind'):

```
rbind(dataframe1,dataframe2)
```

If we want to add the second dataframe to the right of the first one, and the two dataframes have the same number of rows, we can use the `cbind()` command ('column bind'):

```
cbind(dataframe1,dataframe2)
```

If two dataframes do not match completely, the commands `rbind()` and `cbind()` produce errors. An alternative, for partly matched dataframes, is the command `merge()`. This command allows you to specify which rows or columns in the first dataframe are to be matched with those in the second.

Reordering the levels of a factor
The levels of a categorical variable may be shown using the command `levels()`. To display the levels of the variable `verb`, for instance, type:

```
levels(mydata$verb)
```

The output shows the three verbs, which, by default, have been ordered alphabetically, i.e., *jog*, *run* and *skip*. This same order would also be applied when the verbs were to be displayed in a graph or a table, or when choosing one of the verbs as a reference to which the others are being compared. In these cases, the default order may need to be changed into something else. One way of establishing this is by adding an option to the `factor()` command. Suppose, for instance, that the three verbs were to be reordered as *skip*, *jog* and *run* instead, we could write:

```
mydata$verb = factor(mydata$verb,levels=c("skip","jog","run"))
```

The effect of this command is that the verbs will now be displayed in the specified order and that the verb that comes first serves as a reference in a regression analysis. If only the reference level needs to be set without affecting the order of the levels, there is a second alternative, namely the `relevel()` command. If, for instance, we would like to keep the original alphabetical order, but make the verb *run* the reference level of the three, then we could write:

```
mydata$verb = relevel(mydata$verb,ref="run")
```

Counting frequencies
The most basic command for obtaining frequency information from a dataframe was mentioned above. The command `summary ()` is the first port of call for examining frequencies in a dataframe:

```
summary(mydata)
```

This will display a summary of the frequencies for each level of each variable in the dataframe. However, it can only list the frequencies of up to 7 levels. This means that if a variable has many different tags, features or levels, we need to look specifically at the variable itself. For this, we use the `table ()` command. When applied to the variable `verb` in `mydata`, this will be:

```
table(mydata$verb)
```

This will list all the levels in the variable verb and their frequencies.

5. Converting data formats

We have seen that there are two data formats, the dataframe and the cross-tabulation. The data file shown in Table 1 is an example of a so-called flat or raw dataframe. Every row represents a single case. Often this is the format of a data file that is imported into R. It shows the data as they were collected by the researcher. However, we often want to display the frequency of occurrences in a flat dataframe, in which case we need to convert it to a contingency table or cross-tabulation. Moreover, this numerical format is needed for most statistical analysis, such as chi-square tests, cluster analysis, or correspondence analysis.

5.1 Creating a two-way contingency table

The `table()` command can also be applied to two columns in a flat dataframe, resulting in a two-dimensional contingency table. The contingency expresses the relation between the two variables, that is, how often the levels of the first variable co-occur with the levels of the second variable. If we were merely looking at the variables `verb` and `tense` (the first two columns in Table 1), then the following would create a contingency table:

```
table(mydata$verb,mydata$tense)
```

which displays the following output:

```
      future past
jog        2    0
run        0    3
skip       0    1
```

However, this kind of command is of limited use with complex dataframes since it is primarily for two-way contingency tables. In order to summarise a more complex dataframe, we need to 'stack' the columns. In other words, we make a string of two-way contingency tables, all relative to the same variable and put them together in a row.

5.2 Creating a stacked contingency table

The data in `mydata` are organised in more than two columns. That means that, in principle, multiple two-way contingency tables can be constructed from, displaying the relations between the variable in the first column and each of the subsequent variables. These contingency tables can be collected in one larger table with the first variable in the first column and the other variables in the columns to the right. In this case, we say that the contingency tables have been stacked next to each other. This is how Table 2, above was created. For explanatory purposes, it is represented here as Table 4.

Table 4. Data from Table 1 converted to a stacked contingency table

verb	future	past	1stPrs	2ndPrs	literal	metaphor
jog	2	0	1	1	1	1
run	0	3	2	1	2	1
skip	0	1	0	1	1	0

In R, one can create a stacked contingency table from a flat dataframe with the `cbind()` command that we also used earlier for combining dataframes. For instance,

```
cbind(table(mydata$verb,mydata$tense),
      table(mydata$verb,mydata$person))
```

combines the two contingency tables of verb by tense and verb by person. The result looks like this:

```
     future past 1 2
jog       2    0 1 1
run       0    3 2 1
skip      0    1 0 1
```

In this way, all combinations of columns can be stacked next to each other, yielding the results displayed in Table 2. In section 10, we provide a script that does this automatically for an entire dataframe, and which also labels the columns with appropriate names. The script is called `tablebind`. If you want to use this script, copy it into a text file on your computer and save the file as `tablebind.R`. In R, give the command:

```
source("tablebind.R")
```

This makes `tablebind()` available as a new R-command, which takes the flat dataframe as its argument. In order to run it, type:

```
tablebind(mydata)
```

This yields the output displayed in Table 2. Note that the first column in the dataframe is the one upon which the cross-tabulatation will be calculated. If you want a contingency table that calculates figurativity, for example, then the order of the columns in the original dataframe will have to be changed accordingly.

6. Making charts

In R, you can make almost any type of chart, and adjust the layout to the smallest detail. The variety of possibilities is too large to describe in this introduction. Here we provide just a few examples.

The generic command for making a chart in R is `plot()`. When this command is applied to two continuous variables, we get a scatter plot. When applied to a categorical variable, we get a barchart. If we apply this command to the variable `verb` in `mydata`, for instance, we get a chart that shows the frequencies of the verbs in the dataframe.

```
plot(mydata$verb)
```

The chart opens in a new window, called the Quartz-window in MacOSX (Figure 1). Sometimes, plots can be hidden behind the console. The window menu will allow you to bring the plot to the front.

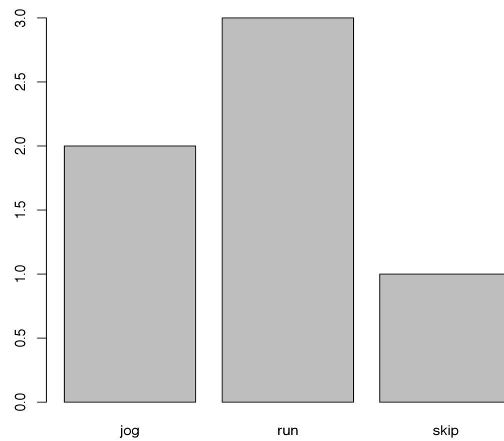

Figure 1. Barplot using `plot()`

There are several ways in which the layout of a chart can be changed. The first is to set the overall graphical parameters of the chart with the command `par()`. The command

```
par()
```

gives an overview of all the parameters and their settings. The background colour of a graph, for instance, is abbreviated as `bg`. To see the current setting for this parameter, type:

```
par("bg")
```

which gives you the default colour of the graph background. Each of these parameters can be modified. Changing the background colour to, say, lightgrey, is established by:

```
par(bg="lightgrey")
```

Modifications in the graphical parameters stay in effect for as long as the Quartz window is open. Once this window has been closed, the old settings are restored in a new window.

The second way of adjusting the chart layout is to specify additional options in the plot command. These options make it possible, among other things, to change the scales of the axes, change the plotting symbols, add

labels, and add a title. The following example expands the limits of the Y-axis scale in the barchart from Figure 1 from 0 to 5, adds the label "Verb" to the X-axis, adds the label "Frequency" to the Y-axis, and adds the title “Barchart Example”:

```
plot(mydata$verb,
     ylim=c(0,5),
     xlab="Verb",
     ylab="Frequency",
     main="Barchart Example")
```

Note that the command has been written out on several lines. Finally, extra elements (legend, arrows, stars, text, lines) can be added separately.

Once the graph is finished, it can be saved as a file. In MacOSX, you simply save the file by choosing the menu “File” and then the option “Save”. The file is saved as a .pdf by default, giving optimal quality. Many applications, such as Preview or Adobe Acrobat have export options, which allow the file to then be saved under any format and at resolutions defined by the user. Under Windows, there is a list of options such as .pdf, .png, .jpg etc., under which the image file can be saved. Alternatively, the file can be written to a file directly from the Console window. The following three commands show an example of saving the barchart as a .pdf-file:

```
pdf("barchart.pdf")
plot(mydata$verb)
dev.off()
```

The same procedure can be followed for .jpg, .png and other file types. Vector formats such as .pdf offer the best quality, but certain versions of MS Word do not accept .pdf images or automatically convert them to poor quality bitmap images. In which case, saving the image as a high quality .png file is the best option. For .png, a resolution of at least 600 dpi is recommended, but more is better.

R possesses a rich inventory of possibilities for visualising results. For any command, if one types a `?` before the command and enters this, a screen will appear with a help file. This will give you the arguments that the command accepts, as well as an example of its use and hyperlinks to other related commands. Experiment by typing:

```
?plot
```

7. Working with scripts

More often than not, an analysis consists of a series of commands. The first command could read in the data, for example, the second command calculates some descriptive statistics, and the third command makes a plot of the results. In that case, it is a good idea to save them in a so-called script file. In order to make a script, select File > New Document from the R menu. A new window will open in which you can add commands, exactly as you typed them in the console window. An example is shown below:

```
mydata=read.table("datafile.txt")
str(mydata)
summary(mydata)
```

Commands can be run directly from the script window. Click once on the command that you want to run, then type command-Enter (Macintosh) or Control-r (Windows). The command and its output are then automatically displayed in the Console window.

There are several advantages of working with scripts, and we recommend that you routinely use them. One advantage is that you build up a collection of scripts with commands that you often use, and which you can use as examples for other scripts.

A second advantage is that scripts can contain personal comments. Comments can be anything from the date that you created the script, to the purpose of the script, to the explanation of a complex command. The following is an example of a very short script, which includes the example command that we saw above, with a comment added that gives information on the date the script was created:

```
# Example R-script, created December 2011
mydata=read.table("datafile.txt")
```

The first line starting with the hash-sign ("#") is the comment. The second line is the command. Comments have no effect if you run them, they are there only for additional information or explanation.

A third advantage is that script files allow you to write very long commands on multiple lines. Commands to make plots, typically, become very long. Writing them on separate lines (as was shown above in section 6) usually makes the structure of the command more transparent, and makes it easier, if there were something wrong with the command, to spot the error.

Finally, R offers the possibility of syntax colouring, which also can help with seeing the structure in R commands. Syntax colouring means that different parts of the script are shown in different colours. A specific colour is reserved for a comment, another colour for a command keyword, a third colour for numbers, and so on. For these reasons, script files can be a great help in learning to work with R, and we recommend that you routinely use them. Script files are normally saved on the computer with the extension `.R`.

8. Extending functionality with packages

When R is installed for the first time it contains many basic functions for doing analyses and making plots. These base functions can be complemented with other functions that are geared towards more special-purpose analyses, such as the ones described in this book. Many so-called packages exist that contain functions used within a specific discipline, or for producing specific types of plots, or doing specific types of analysis. An incredibly rich collection of packages for performing all kinds of statistical analyses exists, and this collection is constantly being improved upon and added to. The packages are small and download quickly.

If you want to use a package, you need to install it first, that is, you need to download it from the R-website and save it to your commuter. One way of installing a package is with the `install.packages()` command. The following example shows how to install the package called `ca`, for doing correspondence analysis described in chapter five:

```
install.packages("ca")
```

After installing, you still need to load the package before you can use it. Loading means that the functions inside the package become available in R. Loading a package can be done with the `library()` command:

```
library(ca)
```

Once you have loaded a package, it will be available until you quit R. At restart, you need to load again with the same command. A good way of preventing you from forgetting to load the necessary package(s) is to add the `library()` command to the R-script.

9. Going further

This short introduction is designed to help new users get started with R. There are many things to discover if you continue working with R, and it will probably take some time to get a good grasp of the kinds of commands, packages and graphs that are useful for the kind of analysis that you want to use R for. To conclude, we offer three pieces of advice that we found helpful on the way of becoming experienced R users.

First, as mentioned above, there is a built-in help function that shows information about the syntax of a command, some examples, and often links to other related commands. A call for help on a command is obtained with `help()` or with `?()`, placing the the command in parentheses.

Second, the Internet is a good place to search for help. There are numerous sites with blogs, tutorials, and user fora. Here you can find R code, pose questions, and see graphs. For the less experienced users, we can recommend the site Quick-R, which contains many clear examples. Furthermore, The R-website also offers a manual.

Finally, the number of books on statistics using R within many different disciplines grows steadily. Books for linguistic analysis are Baayen (2008), Johnson (2008) and Gries (2009a, 2009b). Focusing on graphics, Keen (2010) and Mittal (2001) are accessible to beginners and are relatively complete. Other introductory books include Crawley (2007), Everitt & Hothorn (2009), Maindonald & Braun (2010) and Adler (2010).

10. The tablebind-script

This script can be copied or carefully entered into R and saved as a function, explained in section 7. The script and the data file can be downloaded from XXXXXX.

```
tablebind=function(df.flat)
{
 if(ncol(df.flat)<3)
   return("Dataframe needs at least three columns.")

 for(i in 1:ncol(df.flat))
   df.flat[,i]=factor(df.flat[,i])

 df.stacked.ncol=0
 for(i in 2:ncol(df.flat))
   df.stacked.ncol=df.stacked.ncol+length(levels(df.flat[,i]))
```

```
 k=1
 df.stacked.colnames=rep("X",df.stacked.ncol)
 for(i in 2:ncol(df.flat))
   for(j in 1:length(levels(df.flat[,i])))
    {df.stacked.colnames[k]=
        paste(colnames(df.flat[i]), lev-
 els(df.flat[,i])[j],sep=".")
       k=k+1
    }

 df.stacked=table(df.flat[,1],df.flat[,2],exclude=c(NA),useNA="n
 o")
 for(i in 3:ncol(df.flat))
      df.stacked=cbind(df.stacked,table(df.flat[,1],df.flat[,i],
      exclude=c(NA),useNA="no"))

 df.stacked=data.frame(df.stacked)
 colnames(df.stacked)=df.stacked.colnames
 return(df.stacked)
}
```

References

Adler, Joseph. 2010. *R in a Nutshell. A desktop quick reference*. Sebastopol: O'Reilly Media.

Baayen, Harald. 2008. *Analyzing Linguistic Data: A practical introduction to statistics using R*. Cambridge: Cambridge University Press.

Crawley, Michael. 2007. *The R Book*. Chichester: John Wiley.

Dalgaard, Peter. 2008. *Introductory Statistics with R* (2nd ed.). Dordrecht: Srpinger.

Everitt, Brian S. & Ibrsten Hothorn. 2010. *A Handbook of Statistical Analyses Using R* (2nd ed.). Boca Raton: Taylor & Francis.

Gries, Stefan Th. 2009a. *Quantitative Corpus Linguistics with R: A practical introduction*. London: Routeledge.

Gries, Stefan Th. 2009b. *Statistics for Linguistics with R: A pratical introduction*. Berlin: Mouton de Gruyter.

Keen, Kevin. 2010. *Graphics for Statistics and Data Analysis with R*. Boca Raton: CRC Press.

Johnson, Keith. 2008. *Quantitative Methods in Linguistics*. Oxford: Blackwell.

Maindonald, John & John Braun. 2010. *Data Analysis and Graphics Using R* (3rd ed.). Cambridge: Cambridge University Press.

Mittal, Hrishi V. 2011. *R Graphs Cookbook*. Birmingham: Packt.