Basic R for corpus linguistics*

Gard B. Jenset
Department of Foreign Languages
University of Bergen

1 Introduction

This tutorial is very basic, and it is intended as a quick-and-dirty introduction to some of the possibilities that lie in R, R Development Core Team (2008). At the same time, you should be able to pick up sufficient skills to actually carry out some interesting tasks with R that might be relevant for a corpus-based master's thesis. For further explorations of R, the following resources are recommended: the R manual, which is available for free from the R webside, `http://cran.r-project.org/`, or in the form of a book: Venables, Smith, and the R Development Core Team (2004).

R is usually referred to as a statistics program. However, it is much more than just a statistics environment, and Stefan Th. Gries has called it "the Swiss Army knife of the corpus linguist." Basically, R is a computational environment which can do numerical computation, string processing with characters, and a lot more in addition. R can be programmed, although it is perfectly possible to use R and its packages for quite advanced processing without getting into programming. The R programming language is a free, open-source implementation of the S-plus programming language (which is the language used for writing scripts in the more recent versions of SPSS, in addition to being implemented in commercial stand-alone statistics applications).

1.1 Further reading

In addition to the R manual, there are a number of other resources available. Specifically linguistic applications of R (but with a special emphasis on statistics) can be found in Baayen (2008) and Johnson (2008). It is probably fair to say that Baayen's book is the most advanced of these two books, consequently it is also the most challenging for the novice, but also the most useful one for those looking at a career in linguistics. For a more general introduction to the possibilities in R, Spector (2008) is a very useful guide to working with data whereas Albert (2007) illustrates some of the really advanced things R is capable of, although the latter is definitely not a beginner's book. Also, recent statistics

*Handout for methods seminar in English linguistics, Fall 2008

books such as Gelman, Carlin, Stern, and Rubin (2004) and Gelman and Hill (2007) now regularly include R code. Everitt and Hothorn (2006) present a number of examples of useful statistical analyses and how to compute them in R.

1.2 Getting started with R

The R program can be downloaded from `http://cran.r-project.org/`. The installation process is described on the web site. To install additional packages, you need to check that you have administrator access to the computer you are working on – that is, you must be able to install software. The steps to install packages are as follows:

R code

- Open R, write `chooseCRANmirror()` and press enter.
- Choose a nearby mirror site from the list, for instance Austria.
- Write `install.packages(''NameOfPackage'')` and press enter.

In some cases a package will depend on other packages. R will then install these packages as well if they are not already installed on you computer. Note the quotation marks in the line above; `install.packages()` needs these to find the package you want to install. To see if a package is installed, write `library(NameofPackage)`. This tells R to make the package available, and needs to be done every time you wish to use the package in a new R session.[1] A list packages that are not included in the basic R setup, but useful for corpus linguistics: openNLP and openNLPmodels, gsubfn, languageR, zipfR, tm. See the R web site for further documentation on these packages, and an updated list of all packages available for downloading.

1.3 Basic R syntax

R works with 'vectors' which are 'placeholders' for real data: you may think of them as short-hand containers for whatever is on the right hand side of the assignment operator (either '=' or '<-'). The variable on the left hand side of the operator can consist of a single character or word-like sequences (but no spaces are allowed). Note that R is case-sensitive, so you need to keep in mind *exactly* what a vector is called, but what you call it is up to you. In computer science, word-like names of functions and variables traditionally start with a lowercase character, and when a new 'word' is begun, a capital letter is used. This is just a convention for better readability, as in `myNewCorpLingVariable`. To avoid excessive typing, it is also a good idea to keep variables short – R doesn't care if you call your variable '`x`' or '`oldEnglishWordOrderTableNumber3`' – but you may find it useful to give the vectors names that give a hint of their use.

[1] For frequently used packages, you can modify the R start-up scripts to automatically load the specified packages when R is started. This is sufficiently advanced to fall outside the scope of this tutorial.

A vector is empty until it is assigned a content. This can be done in two ways: using '=' or '<-', they mean the same. The following expression gives the value 5 to the vector c.

(1) `c <- 5` **R code**

Note that this does not produce an immediate output, c needs to be explicitly called to write its contents to the screen. To display the output, type `c` and press enter.

> **Hint**: Although typing is quicker than pointing and clicking, it is still boring! Fortunately, R can remember what you type during a session. Use the up-arrow on the right hand side of your keyboard to recycle or edit commands.

2 Getting data into R

There are a number of ways to load your data into R, cf. (Venables et al., 2004, 47–50) and (Spector, 2008, 13–56). Below, we will consider three methods to construct table-like objects[2] in R and a generic method for reading text.

2.1 The `edit()` function

R is primarily based on a command-line interface, but the `edit()` function is an exception. This function will open a spread-sheet like window where you can manually enter values for each table cell, change the names of the columns, and decide data type for the columns. To use the function, type

(2) `xnew <- edit(data.frame())` **R code**

This code will create a new data-frame object, with empty cells. Closing the 'edit' window assigns the result to the vector. The edit function can also be used to make changes in existing tables, like this:

(3) `xmod <- edit(xnew)` **R code**

The code above makes no changes to the existing 'xnew' object, since all changes are stored in the 'xmod' object.

2.2 The `matrix()` function

A quick and easy way to create a table in R is to simply type:

(4) `tab <- matrix(c(45, 34, 67, 82), nrow = 2)` **R code**

[2]For really large tables, all the functions discussed here are relatively inefficient. Use the function `scan()` instead.

The `matrix()` function takes two arguments: the `c` operator (which 'combines' the values into a single object), and information on how to organize the table entries. It is possible to give R both the number of rows (`nrow`) and the number of columns (`ncol`), but R only needs one of the two to construct the table.

Note that the sequence of figures is not trivial: R assumes that you first list all the figures in the first column, then all the figures of the second column, etc, from top to bottom. Thus the output of the assignment above is the following table:

```
     [,1] [,2]
[1,]   45   67
[2,]   34   82
```

It is usually a good idea to check the output of such an assignment to make sure you got the table values sorted in the order you wanted.

2.3 The `read.table()` function

It is also possible to read an already existing table or data set into R. R can read different file formats, cf. 8.2 below. At the present we will look only at plain ASCII text (files with the .txt extension) which can be imported directly into R. The `read.table()` function creates a data frame object[3] and takes the following arguments: a file path, a logical argument `header = TRUE` which specifies that the first line of the table contains column names and that there are no row names, and finally an argument specifying the kind of character (e.g. tabs or commas) which separates the row values. It is recommended that you use the `read.table()` function for entering raw data with variables into R.

> **Hint**: Are your data stored in an MS Excel file? Use the Excel 'save as' option, and save the file as a tab delimited .txt file before loading the data into R with `read.table()`. You can also save it as a .csv file, and use the R function `read.csv()`.

It is often useful (unless you are good at remembering file paths) to use the `file.choose()` function as the first argument, which will let you search for a file on your computer. Note the empty parentheses in the `file.choose()` function:

(5)
```
c3 <- read.table(file.choose(), header = TRUE, sep = ',')
```
R code

In the example above, headers are included, and the separator sign is a comma (if the table is tab separated, use `sep = '\t'`).

[3] For the present purposes we will disregard the differences between a data frame and a matrix in R.

Hint: If you work with historical corpora and CorpusSearch, the easiest way to convert the output files to something readable by R is as follows: copy the output summary, and paste it into a text document. In MS Notepad (or a similar program), standardize the separator character to either tab or comma, using search and replace. Read into R using `read.table`, with `sep = ' ,' or sep = '\t'`.

2.4 The `readLines()` function

A running text can be read into a vector using the `readLines()` function. Like the previous function it takes as its first argument a file path, or the default file location function in R: `file.choose()`. The following will allow you to read in a text file, choosing the file from a Windows Explorer window:[4]

(6)
```
txt <- readLines(file.choose())
```
R code

Like the other functions in this section, it is recommended that you assign the result of the function to a vector, otherwise the text will simply be printed to the R window.

Note that there is a limit as to how much computer memory R has access to. Consequently, if you were to get hold of for instance a text file containing the entire British National Corpus with around 1 million words plus markup, an attempt to read this into R would probably result in the program running out of memory and crashing. For the purposes considered here, R has plenty of memory, but you can check this yourself typing `memory.size()` or `memory.limit()`. To read more about this, consult the R help files by typing `?memory.size` or `?memory.limit`.

Hint: To get help on a specific function in R, write `?functionName`, and R will open a window with help files. If the function resides in a package you have downloaded, you may have to load the package with the `library()` function first.

2.5 Saving data

The results of these operations can be saved with the function `write.table()`. The function takes as its input a file path or the Windows file locator and a matrix, as shown below:

(7)
```
write.table(c3, file.choose())
```
R code

with the matrix as the first argument, and the file location as the second.

[4] This procedure should work in Apple computers as well.

It is also possible to save the vectors you have created for later use, as explained in (Spector, 2008, 35–36). Assume we have the vectors `x`, `y` and `tab`. Typing

(8) `save(x, tab, file='fileName.rda')` **R code**

This means that when you open R next time, writing `load(fileName.rda)` will allow you to access the vectors in this file.

3 Creating tables from data: `xtabs()`

In the previous section, we looked at ways of creating tables with aggregated values. This is fine for performing simple tests on existing values, but it is not very realistic in the context of a large empirical investigation. The functions like `edit()` and `matrix()` above assume that you have already counted occurrences in your data set, something which should be avoided at all cost, because it is unnecessarily time consuming and because it is prone to human error. Instead, we would like R to count the raw data for us, a procedure which is both faster and more reliable.

In fact, R is an excellent tool for data analysis and the best way of creating and working with tables is to load your raw data directly into R. Then, tables can be created as the need arises using the function `xtabs()`. First, use `read.table()` to read the tab separated test data set 'medata.txt' into R. Inspect the data set visually and then take a look at a summary of the data using the following R function:

(9) `summary(test)` **R code**

Next, use `xtabs()` to view specific combinations of variables in table form, such as the number of clause patterns occurring main or sub clauses:

(10) `test.xtabs = xtabs(~Pattern + Clause, data = test)` **R code**

In `xtabs()`, the tilde (∼) operator specifies a formula with nothing to the left of the tilde.[5] The arguments following the tilde are the factors that we want to include in the analysis (compare them with the column names in the data set), the final factor specifies the data. The vector `test.xtabs` can now be used as an argument to a statistics function, such as `chisq.test()` or `fisher.test()`. It is also possible to create tables with three factors:

(11) `test.xtabs = xtabs(~Pattern + Clause + Type, data = test)` **R code**

This gives a three-way classification of the variables, presented in two tables. A similar effect of a table with several levels can be achieved using `ftabs()`, like this:

(12) `test.ftabs = ftabs(test)` **R code**

[5] See Baayen (2008, 13) for a full explanation.

Note that aggregated data can be reshaped in a simple way using the `rep()` function:

(13) `rep(data, data$Freq)`

The code above will take every row in a table, and repeat its contents as many times as specified by the integer in the column called 'Freq'. For more advanced reshaping, check out the `reshape` package.

4 Working with tables

Below, some basic operations on tables are exemplified. This is a comprehensive subject, as R has a number of useful functions for working with tables, and attempting to cover them all here would be futile. For instance, there are a number of different ways to compute the sums of table rows or columns, using either `rowsum()`, `colSums()`, or `tapply()`. For a more in-depth treatment of what can be done with tables see e.g. Baayen (2008, 4–17).

Once the data have been read into R, we can start working with it. Consider the table we created earlier, called 'tab'.

```
     [,1] [,2]
[1,]   45   67
[2,]   34   82
```

`tab` can now be given as input to various statistical tests. It is also very simple to transform the values in `tab` into proportions.

4.1 Relative frequencies and percentages

To make relative frequencies, type:

(14) `tRel <- tab/sum(tab)` **R code**

this will give a table like the one below:

```
          [,1]      [,2]
[1,] 0.1973684 0.2938596
[2,] 0.1491228 0.3596491
```

Since percentages are only relative frequencies multiplied by 100, this is easily computed like this:

(15) `tPer <- tab/sum(tab)*100` **R code**

which gives the following output:

```
          [,1]     [,2]
[1,] 19.73684 29.38596
[2,] 14.91228 35.96491
```

The procedures above for relative frequencies and percentages are taken from (Baayen, 2008, 14–15). The R tables can be saved as explained above, or simply be copied and pasted into your favorite text processor.

> **Hint**: MS Word does not handle tables very well. An easy solution is to copy the R table, paste it into MS Excel, and import the Excel table into Word. Alternatively, use LaTeX.

However, the tables above still look quite messy! There are way too many decimals – in corpus linguistics we rarely give percentages with more than one decimal, and often the sample size is such that we can safely dispense with decimals altogether – and rounding them off manually takes too much effort, so we get R to do it instead using `round`:

(16)
```
tPer2 <- round(tab/sum(tab), digits=2)*100
```
R code

which produces

```
      [,1]  [,2]
[1,]   20    29
[2,]   15    36
```

The `digits=2` argument to `round()` specifies the number of decimals, thus `digits=2` produces no decimals when converted to percent, 3 produces one decimal, etc.

We can also get marginal sums for rows and columns of the original table by using `addmargins()`:

(17)
```
tab2 = addmargins(tab, 1)
```
R code

The code above gives column sums, use '2' for row sums. To get both rows and columns, write:

(18)
```
tab3 = addmargins(tab, c(1,2))
```
R code

4.2 Extracting rows and columns

You may have noticed the rather cryptic row and column headings in the tables above, such as `[,1]`. These are indexes which can be used to point to a row, a column or a single cell. R will use 1, 2, ..., etc when no names are explicitly declared (but even when you give the rows or columns proper names, you can still access them with their numbers).

R interprets anything before the comma as having to do with rows, and anything after as having to do with columns. Thus, to get R to print only the first row of `tab` to the screen, write

(19)
```
tab[1,]
```
R code

Similarly, to get the second column, write

(20) `tab[,2]` **R code**

Predictably, the code to extract a single cell value of a table is the combination of its row and column number

(21) `tab[1,2]` **R code**

In a small table, this procedure is superfluous, but it can be of great help when working with large tables. Also, if you want to use parts of a table as input to a function, it is usually better to give references like this rather than typing the actual numbers in order to avoid errors.

To rename columns in a table, use `names()`:

(22) `names(tab)[1:2] = c('Variable 1', 'Variable 2')`

4.3 Other table operations

The function `table()` will give a summary list of the elements in the data set. The `subset()` function is a useful way to extract information:

(23) `subset(tt, tt$Freq == 40)`

(24) `subset(tt, tt$Freq > 80)`

(25) `subset(tt, tt$Freq == min(Freq))`

Note that the double equal sign ('==') is needed to compare equality between conditions. Remember, the '=' is an assignment operator.

4.4 Merging tables

If you want to combine data from two different tables into one, this can be achieved using the `merge()` function. The function is particularly useful if you want to combine material extracted from different tables into one table:

(26) `x <- test[,1]` **R code**

(27) `y <- test[,3:4]`

(28) `test2 <- merge(x,y)`

This will produce a new table `test2` containing columns 1, 3 and 4 from the original `test` table.

5 Processing text files

For translating characters into other characters, the functions `tolower(), toupper()` or `chartr()` will usually do the job. However, for more comprehensive cleanup,

the functions `sub()`, `gsub()` or `gsubfn()` should be used. However, these latter functions make use of regular expressions in pattern matching, which takes some time (and effort!) getting into, and their use is not covered here, see instead Spector (2008, 91–99). The advantage to this is the possibility for making standardized scripts that can make all the changes in one go, and be reused; but using 'search and replace' in a text editor will of course also get you there eventually.

In order to translate a text into lowercase, type the following:

(29)
```
lCtxt = tolower(txt)
```
R code

Alternatively, use

(30)
```
lCtxt = chartr([A-Z], [a-z], txt)
```

Note that `chartr()` is a general function, and more flexible than `tolower()`. It is for instance possible to change other characters than letters, or translate only certain letters or letter combinations.

6 Creating a sample

Sometimes it is useful to create a sample from the total amount of data, either because it is impractical to work with the full data set or because we want a random sample to use for statistical testing. To do this, R has a default function called `sample()`. For instance, to choose a random cell in the `tPer2` table above, write

(31)
```
rs <- sample(tPer2, size = 1, replace = FALSE)
```
R code

The `size` argument lets you specify the size of your sample, whereas `replace` lets you choose whether you want to sample with replacement or not. An optional argument `prob` lets you adjust the probabilities.

7 Graphs and figures

One of the things that R does best is producing print quality graphical material. The sections below briefly introduce a few plot functions, but there are a multitude of methods with possibilities for adjustments in R, see Venables, Smith, and the R Development Core Team (2004) and help files for further details.

7.1 The `barplot()` and `histogram()` functions

With the Middle English test data set stored in `test`, we can make a simple barplot like this, cf. figure 1:

(32)
```
barplot(xtabs(~test$Pattern), xlab = 'Word order pattern',
col = 'gray')
```
R code

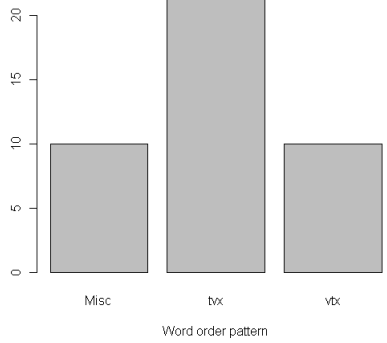

Figure 1: A simple bar plot.

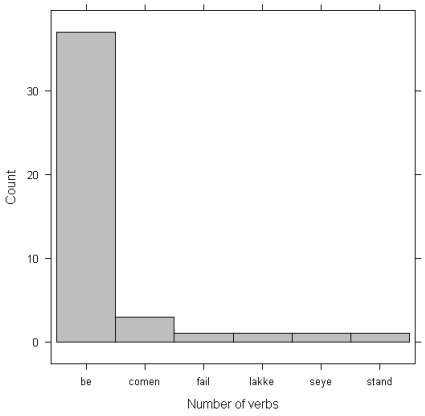

Figure 2: A histogram made with the `histogram()` function.

The arguments to `barplot()` is `xtabs()` – with an argument specifying the name of the object containing the data set, an operator, and the variable we are interested in (`test$Pattern`) – `xlab` which is the label for the x-axis and `col` which is the bar color (the default, `NULL`, gives bars with no color).

Alternatively, `histogram()` can be used, as shown below, giving the result in 2.

(33)
```
histogram(~test$Verb, type='count', xlab = 'Number of verbs',
col = 'gray')
```

7.2 The `assocplot()` function

Graphical elements are powerful devices to display properties of the data. A good example of this is the R function `assocplot()` which produces Cohen-Friendly plots, cf. Friendly (1995). The plot, as shown in figure 3, visualizes a two-way table. The variables will show up above or below the baseline expected frequency. The Cohen-Friendly plot provides a graphical representation of the contributions of the various cells to the final chi-square result of the table. For further details, see Friendly (1995), Cohen (1980) or the R help files (`?assocplot`).

7.3 The `plot()` function

To visualize more variables, we can use the generic `plot()` function, like this, illustrated in figure 4:

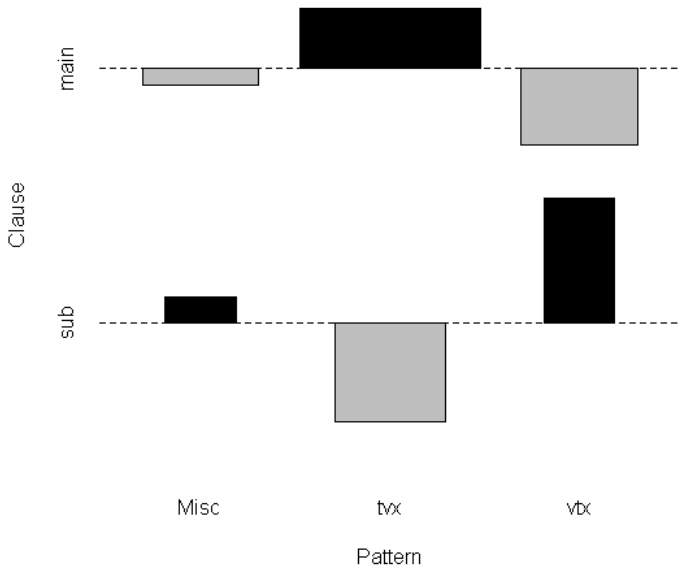

Figure 3: A Cohen-Friendly association plot.

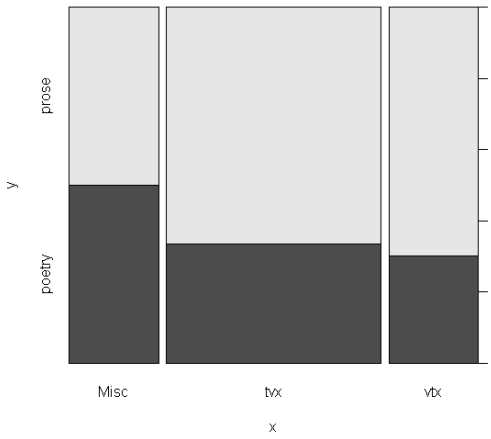

Figure 4: A figure produced with a simple `plot()` function.

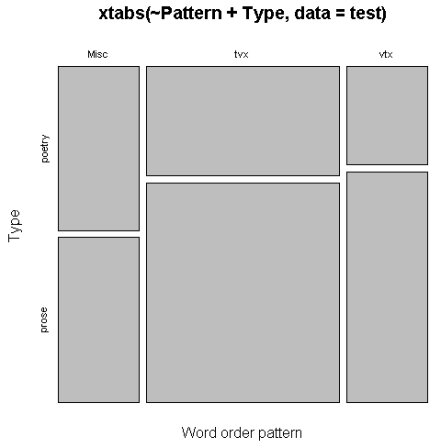

Figure 5: A mosaic plot made with `xtabs()`.

(34) `plot(test$Pattern, test$Type)` **R code**

It is also possible to send the `xtabs()` function to the generic `plot()`:

(35) `plot(xtabs(~Pattern + Type, data = test), xlab = 'Word order pattern')` **R code**

This produces a mosaic plot which looks a little different from the previous figure, cf. figure 5, but gives the same overview of the proportions of all variables in the data set. For a deeper interpretation of such plots, see Friendly (1995).

7.4 Saving graphs and figures

To save the R graphics you have created, simply right-click the image, and choose 'save as'. Alternatively, choose 'copy as bitmap' and paste it into your favorite image processor for further refinement.

> **Hint**: If you want decent graphics of printable quality, make sure to save the images as PNG (.png) files in the image processor program.

8 Examples of useful R packages

Below is a brief presentation of two useful R libraries for linguistic work, both available for download from the R web site. There is a huge number of user con-

tributed libraries available, and if you are looking for the a solution to a particular type of problem, the downloadable packages at `http://cran.r-project.org` is a good place to start.

8.1 The *openNLP* package

The package *openNLP* (along with its extension *openNLPmodels*), by Feinerer (2008), provides some neat, easy-to-use functions for English and Spanish. The package allows for sentence detection (based on punctuation), tokenization, and parts of speech tagging (based on a maximum entropy model). First, read the text data into a vector `txt` using `readLines`, as explained above:

(36) `txt <- readLines(file.choose())` **R code**

Now you can start working with the text. Load openNLP and openNLPmodels with the `library()` function. Then you can divide it into sentences using the sentence detection function:

(37) `txt2 <- sentDetect(txt, language = ''en'')` **R code**

To find how many sentences your text was divided into, use the standard R function

(38) `length(txt2)` **R code**

You can also find how many characters it contains with another standard function:

(39) `nchar(txt)` **R code**

Tagging and tokenization of the text is done with the following functions:

(40) POS tagging: `txt3 <- tagPOS(txt, language = ''en'')` **R code**

(41) tokenization: `txt4 <- tokenize(txt, language = ''en'')`

> **Hint**: The tokenization function in *openNLP* will treat punctuation marks as tokens. If you don't want this, try the standard R function `strsplit(txt, ' ')` and see what happens. Note that this means R will treat "bank" and "bank," (with and without punctuation) as different items. An alphabetic word list can then be created using the function `table` like this: `table(txt)`. For a list ordered by frequency, type `sort(table(txt))`. To check if Darwin used the word 'planet' in his text, type `any(grep('planet', txt))`. Also, check out `as.data.frame()` and `ftable()`.

Unless the functions above are assigned to a vector, the result will simply be displayed directly on the screen. However, the tagging and tokenization functions take as input a character vector containing either the original text or

a processed version (i.e. you can tokenize a tagged version of the original text), which means you can save some typing by assigning output to vectors.

8.2 The *foreign* package

In case you are working with data collected by someone else and stored in a particular format, the package called *foreign*, by core members, DebRoy, Bivand, and others: see COPYRIGHTS file in the sources. (2008), will allow you to read data of virtually any file format into R. The package handles data from spreadsheet software like relational data base files, and data files from statistics programs like SPSS, SAS and Stata. For instance, SPSS data files can be read using the `read.spss()` function. It is also possible to write data into files that these programs can read. This is quite useful, especially since these programs are not required to be installed on the computer you are working on.

> **Hint**: When using R or contributed packages, remember to cite them! Type `citation()` to see how to properly cite R, and `citation('nameOfPackage')` to see how to cite a specific library.

9 Some corpus resources

This is a totally subjective list of useful corpus resources.

9.1 Some contemporary English corpora

BNC
American corpus

The `http://corpus.byu.edu` page contains a number of useful corpora, including the 100 million word *British National Corpus* (BNC) and the 360 million-and-still-growing word *Brigham Young Corpus of American English*. These corpora are freely available, and are accessed through a standardized user-friendly web search interface. Parts of speech information only.

Further resources can be found at the AKSIS web page, `http://www.aksis.uib.no/`, and at the web page of the *English-Norwegian Parallel Corpus* at the University of Oslo, `http://www.hf.uio.no/ilos/forskning/forskningsprosjekter/enpc/`.

9.2 Historical corpora

Penn-Helsinki
York-Toronto

The Department of Foreign Languages has a limited number of licences for the *Penn-Helsinki parsed corpora of Early English*. These corpora contain information on parts of speech as well as syntactic information. There is also the *York-Toronto corpus of Old English* (YCOE), available from the Oxford Text Archive (`http://ota.ahds.ac.uk/`) free of charge (note that OTA itself contains huge amounts of electronic text, although not all of it is tagged or parsed). Like the Penn-Helsinki corpora, it is tagged and parsed. All these corpora are

searched with the CorpusSearch 2.0 program, a free program available from `http://corpussearch.sourceforge.net`.

9.3 A web-based alternative

Sketchengine

The *Sketchengine*, (`http://www.sketchengine.co.uk/`) can be used to make POS-annotated, web-based corpora – it is particularly useful for creating your own domain-specific corpus. You can create a fully functional free 30-day trial account, but the UiB (through AKSIS) also has a few licenses for this product.

References

Albert, J. (2007). *Bayesian computation with R*. New York: Springer.

Baayen, R. H. (2008). *Analyzing linguistic data: A practical introduction to statistics using R*. Cambridge: Cambridge Univeristy Press.

Cohen, A. (1980). On the graphical display of the significant components in a two-way contingency table. *Communications in Statistics – Theory and Methods A9*(10), 1025–1041.

core members, R., S. DebRoy, R. Bivand, and others: see COPYRIGHTS file in the sources. (2008). *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, dBase, ...*. R package version 0.8-26.

Everitt, B. S. and T. Hothorn (2006). *A handbook of statistical analyses using R*. Boca Raton, Fl.: Chapman & Hall/CRC.

Feinerer, I. (2008). *openNLP: openNLP Interface*. R package version 0.0-6.

Friendly, M. (1995). Conceptual and visual models for categorical data. *The American statistician 49*(2), 153–160.

Gelman, A., J. B. Carlin, H. S. Stern, and D. B. Rubin (2004). *Bayesian data analysis* (2^{nd} ed.). Boca Raton, FL.: Chapman & Hall/CRC.

Gelman, A. and J. Hill (2007). *Data analysis using regression and multilevel / hierarchical models*. Cambridge: Cambridge University Press.

Johnson, K. (2008). *Quantitative methods in linguistics*. Oxford: Blackwell Publishing.

R Development Core Team (2008). *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing.

Spector, P. (2008). *Data manipulation with R*. New York: Springer.

Venables, W., D. Smith, and the R Development Core Team (2004). *An introduction to R: Revised and updated*. Bristol: Network Theory.