Introducing R for Corpus Linguistics

Gard B. Jenset

Tuesday, May 26, 2015

Table of Contents

1 Exploring R

The present document is intended to accompany an introductory session on how to use the R platform for corpus linguistics. However, it can also be used as a self-contained tutorial, e.g. for refreshing the material covered in the session.

As a short introduction, the tutorial does not claim to be comprehensive, and for further study a book such as "Quantitative Corpus Linguistics with R" (Gries 2009) is recommended.

The tutorial assumes that R has been installed on your compute and that you are able to find file paths to folders (and saving files), but assumes no particular computer insight beyond that.

Some brief references are made to regular expressions ("regex"). No knowledge of this is assumed, and the use of regular expressions is kept to a minimum. Some background knowledge is nevertheless useful:

- Regular expressions can be used to match a general pattern without knowing the exact value of the string identified
- Regular expressions use some reserved characters for special purposes, notably: `. * ? ! [] () ^ $`
- To refer to the actual character value in these cases, regex in R require two backslashes: `\\[`
- Regex are always enclosed in quotation marks in R
- A shorthand notation can be used to refer to special characters:
  - a word character: `\\w`
  - a non-word character: `\\W`
  - a word boundary: `\\b`

For further explanation and examples, see e.g. Gries (2009).

The code below was created on a Windows 7 computer using R version 3.1.2 (2014-10-31).

1.1 Example vectors

We start by typing data into R. Recall that a vector is a sequence of values, all of the same data type.

Create some example vectors, containing characters/strings, numbers and boolean values. Give them informative names. Use the `c()` function to concatenate individual data points into vectors.

```
affix = c("ly", "in", "re", "ity", "ation", "ee", "ism")
length = c(2, 2, 2, 3, 5, 2, 3)
native = c(TRUE, T, FALSE, F, F, F, F)
```

1.2 Find length of a vector

```
length(affix)

## [1] 7
```

1.3 Extract a subset

We can extract a subset of the vector with `[]` combined with numbers referring to the **index** of the element (numbered from 1 to the end of the vector).

The first 4 affixes are extracted like this:

```
affix[1:4]

## [1] "ly"  "in"  "re"  "ity"
```

The last two are extracted like this:

```
affix[6:7]

## [1] "ee"  "ism"
```

We can also use the generic functions `head()` and `tail()` for this. By default they return 6 values, but we can use the `n` argument to specify how many:

```
head (affix)

## [1] "ly"    "in"    "re"    "ity"   "ation" "ee"

tail (affix, n = 2)

## [1] "ee"  "ism"
```

We can also extract data based on a condition. The example below extracts affix length (in characters) where the length is greater than 2:

```
affix[ length > 2 ]

## [1] "ity"   "ation" "ism"
```

1.4 Combine into a data frame

Recall that a data frame is multidimensional data structure that can hold different data types in columns. Each column corresponds to a vector.

```
affix.df = data.frame(af = affix, len = length, native = native)
```

1.5 Inspect the data frame

This data frame has 7 rows. We can view it by typing the name:

```
affix.df

##      af len native
## 1     ly   2   TRUE
## 2     in   2   TRUE
## 3     re   2  FALSE
## 4    ity   3  FALSE
## 5 ation   5  FALSE
## 6     ee   2  FALSE
## 7    ism   3  FALSE
```

We can summarize it using the `summary()` function:

```
summary(affix.df)

##      af          len          native       
##  ation:1   Min.   :2.000   Mode :logical  
##  ee   :1   1st Qu.:2.000   FALSE:5        
##  in   :1   Median :2.000   TRUE :2        
##  ism  :1   Mean   :2.714   NA's :0        
##  ity  :1   3rd Qu.:3.000                  
##  ly   :1   Max.   :5.000                  
##  re   :1
```

To inspect the first 3 rows, we can use `head()` with `n = 3` (default value is 6):

```
head(affix.df, n = 3)

##   af len native
## 1 ly   2   TRUE
## 2 in   2   TRUE
## 3 re   2  FALSE
```

1.6 Data frame subset

We can also subset data using `[]` as in the example above. Since the data frame is **multidimensional** we can specify rows, columns, or a combination.

The syntax is: `[row index , column index]`

```
# first row
affix.df[ 1, ]

##   af len native
## 1 ly   2   TRUE

# the first 3 rows of the first column
affix.df[ 1:3, 1 ]

## [1] ly in re
## Levels: ation ee in ism ity ly re
```

1.7 Column operator

The data frame has a special syntax for accessing the column variables: the dollar sign $ followed by the name of the column.

If we want to refer to the length column by name (rather than by index) we can write:

```
affix.df$len

## [1] 2 2 2 3 5 2 3
```

1.8 Column operator subsetting

We can combine subsetting with the column operator to extract a subset of the data matching a particular condition:

```
affix.df [ affix.df$len > 2,  ]

##      af len native
## 4   ity   3  FALSE
## 5 ation   5  FALSE
## 7   ism   3  FALSE
```

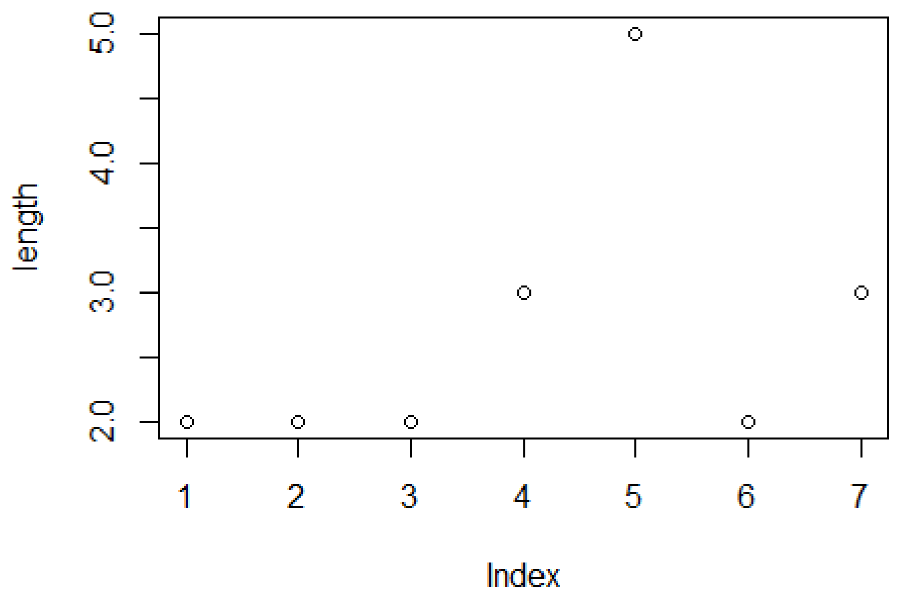
1.9 Basic plots

The advanced functionality for generating production quality plots is one of the strengths of R. Here are some basic types:

A simple scatter plot

This basic plot has **lots** of scope for tweaking and modification (`?plot`). We start by plotting the `length` data from above:

```
plot(length)
```

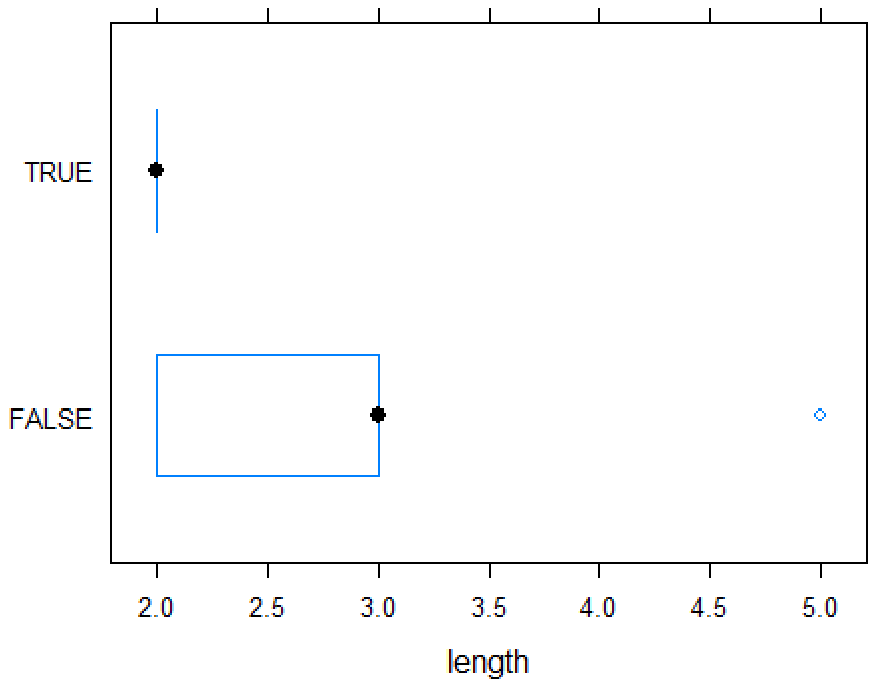


Note how the **values** (y-axis) are plotted against the **index number** (x-axis). The reason is that we only provided one variable to plot.

A box and whisker plot

Let's plot the length of the affix against its etymological history (native vs. non-native), using a box and whisker plot. The best result is obtained by loading the library `lattice` (included in R) and using the `bwplot()` function:

```
library(lattice)
bwplot(native ~ length)
```



The tilde operator (~) specifies that `native` should be plotted as a function of `length`.

Plotting data from a data frame

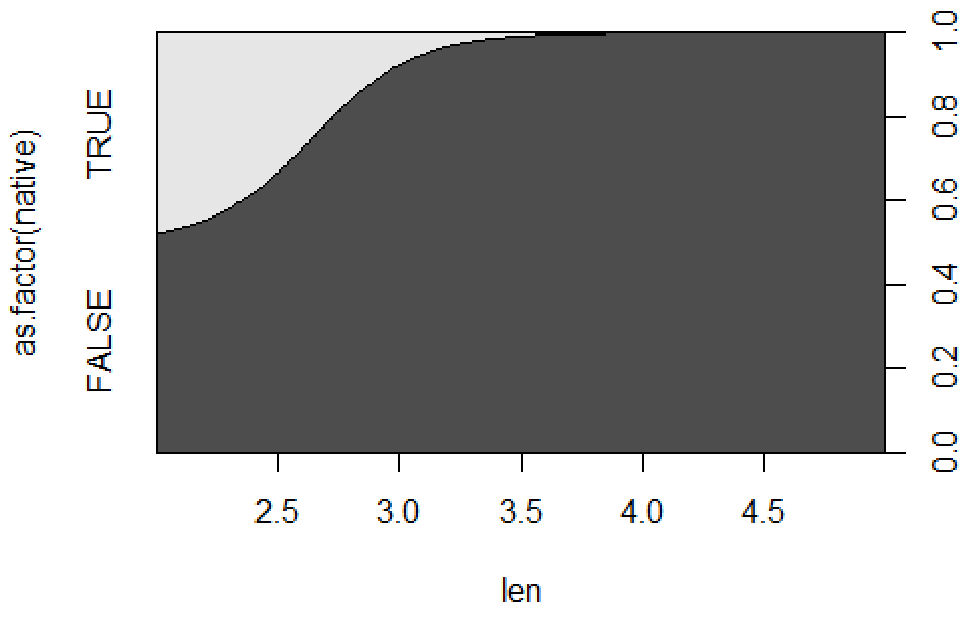
We can pick data to plot from a data frame. All R plot functions have an optional `data` argument where you can specify where to find the variables to be plotted (data frame column names). We can re-write the code above as:

```
bwplot( native ~ len, data = affix.df)
```

A conditional density plot

Another way of plotting the relationship between nativeness and length is to use a conditional density plot, which shows the relationship as changing probabilities:

```
cdplot( as.factor(native) ~ len, data = affix.df)
```



Note that the `as.factor()` function is necessary, since `native` has data of type boolean, whereas `cdplot()` expects a factor.

1.10 Saving plots

There are 3 basic ways to save a plot. Each allows you to control the format (e.g. PDF, PNG, JPG).

In R console

- Create the plot
- Click on the graphics device (window with plot)
- Go to `File` (upper left corner)
- Go to `Save as` and choose format

In RStudio

- Create the plot
- Go to the `Plots` tab on the right hand pane
- Click on the `Export` button and choose format

With R code

Recommended for reproducibility, but make sure you don't overwrite files you want to keep!

```
# specify where you want to save the plot, and the format
pdf("C:\\Users\\gbj\\Documents\\kurs\\grenoble_vår_2015\\img\\cdplot.pdf")
cdplot( as.factor(native) ~ len, data = affix.df)
dev.off()

## pdf
##   2
```

Notes:

- `dev.off()` is required to re-activate the "normal" plotting device in R.
- Make sure to change the file path to match your computer's folder structure.

1.11 Saving data

Workspace

When you close the R application (either in the console or in RStudio), R will ask if you want to save your workspace. This is a `.Rdata` file containing all the objects you have created. This is useful, and can save some time when you work concurrently with many objects.

Data frame

Most of the time it is recommended to save your data as regular file, e.g. at .txt or .csv file. This is also a better format for sharing data.

To save the data frame you created, use `write.table()`. Only two arguments are required: what to save, and where to save it.

```
write.table(affix.df, "C:\\Users\\gbj\\Documents\\kurs\\grenoble_vår_2015\\da
ta\\data_frame_with_default_settings.txt")
```

However, we can control the output in more detail:

```
write.table(affix.df, "C:\\Users\\gbj\\Documents\\kurs\\grenoble_vår_2015\\da
ta\\data_frame_with_custom_settings.txt",
                            quote = FALSE, # switch off quotes around strings
                            row = FALSE, # don't save row numbers
                            sep = "\t" # specify tabulator separated fields
            )
```

Compare the two output files to see the difference!

Note: Make sure to change the file paths to match your computer's folder structure.

2 Reading data

Most of the time we want to read data that has been saved to the disc, rather than typing it into R. In this section you will see two ways of reading data:

- `read.table()` for tables / data frames
- `scan()` for running text

2.1 Reading saved data frames

A saved data frame can be read into R using `read.table()` like this:

```
new.df = read.table( "C:\\Users\\gbj\\Documents\\kurs\\grenoble_vår_2015\\dat
a\\data_frame_with_custom_settings.txt", header = TRUE, sep = "\t" )
```

As you can see, the syntax closely matches that of `write.table()`. `header = TRUE` specifies that the first row are to be treated as column names, and the `sep` argument tells R to divide up the data into rows and columns based on tabulators.

Note: Make sure to change the file path to match your computer's folder structure.

Some extra arguments you might need

These arguments are often needed for some data:

- `dec = ","` for setting the decimal separator in non-English environments
- `quote = "\""` whenever your data contains strings with apostrophes

In general, it's good to look closely at the documentation (`?read.table`).

2.2 Reading running text

In the sections below you will work with paragraphs of running, un-annotated text. For data in this format it's more convenient to use the `scan()` function.

The text is the beginning of the Wikipedia article on Alice in Wonderland (see Appendix). Copy the text, save it as `alice.txt`.

Read the text into R using the third code example below, substituting the file path with the path to the file on your computer.

```
# not really recommended: read from computer clipboard
# aw = scan("clipboard", what="char")

# better: opens a file dialogue box
# aw = scan(file.choose(), what="char")

# recommmended: specify the file path
aw = scan( file = "C:\\Users\\gbj\\Documents\\kurs\\grenoble_vår_2015\\data\\
alice.txt", what = "character")

# note the double backslashes: \\ and quotation marks
```

2.1 Summarize `alice.txt` data

Check that the length and type of data match the results below:

```
length(aw)

## [1] 453

class(aw)

## [1] "character"
```

2.2 Inspect beginning and end

Once the data is loaded into the R workspace, it is a good idea to check that it looks OK. One way to do this is to simply look at the beginning and the end of the data and see if they match your expectations.

Exercise

Refer back to section 1.3 above and extract the **first 7** and **last 5** values of the `aw` data. The output should look like this:

First seven

Alice's, Adventures, in, Wonderland, (commonly, shortened, to

Last five

Cat, and, the, Mad, Tea-Party.

2.3 Cleaning up

`scan()` will by default expect the data to be separated by white space. However, this is too simple an approach here. Words might be separated by white space, new line characters, or punctuation marks. Here is how to clean up the data further:

- combine all the elements together into one string
- use regular expressions to split on all non-words (including punctuation)
- combine the result into a vector

The elements of `aw` can be combined into one string separated by white space by using the `paste()` function:

```
aw.pasted = paste(aw, collapse = " ")
```

We can now use the `strsplit()` function for splitting strings based on a pattern that we specify (in this case: all non-word characters)

```
aw.split = strsplit(aw.pasted, "\\W")
# head(aw.split) # try this if you want to see the result
```

One problem remains: `strsplit()` returns a list of vectors. That is, instead of all the words as separate elements in a vector, we get a **list** with one element, and that element contains the words. We want a single vector of elements (which is what we started with). We can acheive this with the `unlist()` function which simplifies a list by turning its elements into a vector:

```
aw.clean = unlist(aw.split)
head(aw.clean, n = 3)

## [1] "Alice"      "s"          "Adventures"
```

3 Basic corpus linguistics

In this section you will explore some basic corpus linguistics tasks, using the Alice in Wonderland Wikipedia text as example.

These operations are also described in Stefan Gries' book "Quantitative Corpus Linguistics with R", although some of the code used to achieve the end results differs. This illustrates a nice property of R: there's more than one way of doing things.

3.1 A sorted word-frequency list

The most basic operation in corpus linguistics is probably to create a list of sorted word frequencies. Here's how to get word frequencies from `aw.clean`.

To make the word list case insensitive we apply the function `tolower()` to the words:

```
aw.lc = tolower(aw)
```

The `table()` function will count the number of times each word is found:

```
aw.table = table(aw.lc)
```

The `sort()` function will sort the frequencies:

```
aw.sorted = sort(aw.table)
head(aw.sorted)

## aw.lc
## (commonly      (the     (this         .       10,       13,
##         1         1         1         1         1         1
```

Note that the code above contains some unnecessary steps, for the purpose of showing how the operation of building a frequency list contains several stages. R functions can take the **output of other functions as input**, which allows us to nest the code like this:

```
aw.sorted.freq.list = sort( table( tolower(aw.clean) ), decreasing = T )
head(aw.sorted.freq.list)
```

```
##
##      the  in and  of   a
##   57  38  14  13  13  12
```

The `decreasing = TRUE` argument sorts from highest to lowest. Note how white space is also treated like a word. This is addressed further in section 3.4.

3.2 A simple concordance list for "the"

A basic functionality of corpus linguistics software packages is the ability to show words in context, as concordance lines. Below you will see how a very simple list of concordances can be created with R.

Here we use the `grep()` function which will find instances matching a regular expression. With the `ignore.case` argument we can omit turning all the words to lower case.

In the step below, we find the index (i.e. position) of all instances of "the":

```
the.ind = grep("\\bthe\\b", aw.clean, ignore.case=T)
head(the.ind)

## [1]  26  54  61  80  84 111
```

Note the regex pattern \\b for word boundary, to avoid matching "**the**n" or "**the**re".

With the index position information, we can identify all the instances of "the" using subsetting:

```
head( aw.clean[ the.ind ] )

## [1] "the" "The" "the" "the" "the" "the"
```

We can now identify what comes before and after, based on the index positions:

```
# concordances:
aw.conc = data.frame(before = aw.clean[ the.ind-1 ], keyword = aw.clean[the.i
nd], after = aw.clean[ the.ind+1 ])
head(aw.conc)

##   before keyword     after
## 1  under     the pseudonym
## 2            The      tale
## 3 giving     the     story
## 4     of     the      best
## 5     of     the  literary
## 6     in     the   fantasy
```

Excerise 1

Modify the code above to create a concordance list for "Alice".

Excerise 2

Think about how you would modify the code above to include more context around the keyword.

3.3 A frequency list of word-pairs

A slightly more complex task is to count how many times words occur together. This is the basis for identifying collocations (words occurring together more often than expected by chance). Finding collocations is not covered here, but let's see how we can get the counts of how many times two words occur together.

For this task we're using the `sapply()` function, which can take another function (in this case one written for this purposes with the `function()` construct) and applies it to all elements of a **list**.

```
# paste together word pairs. Note we're using list created above: aw.split
pairs = sapply( aw.split , function(x) paste( head(x,-1) , tail(x,-1) , sep =
" ") )
head(pairs)

##      [,1]
## [1,] "Alice s"
## [2,] "s Adventures"
## [3,] "Adventures in"
## [4,] "in Wonderland"
## [5,] "Wonderland "
## [6,] " commonly"

# count and sort
pairs.freq = sort( table ( unlist(pairs) ) , decreasing = T)
head(pairs.freq)

##
##            in the      The Liddell    of the the book
##         6         6          4         4         4         4
```

Note: Our function creates two lists whose elements are shifted with respect to each other, so that the first element of list A is the **first element** of the original data, whereas the first element of list B is the **second element** of the original data. Pasting them together gives us the first and the second elements as a unit.

3.4 A stopword list

If you are familiar with corpus linguistics, you may by now be asking yourself "what about stopwords?". That is, how interesting is it to know that function words like "the" or "a" (let alone a white space) are very frequent in a text?

The solution to this very relevant question is to create a stopword list. This can be used to create a subset of the original data by following this recipe:

- Identify the position of stopword in the data using `%in%`
- The negation operator `!` is used to find words not in the stopword list

This provides us with everything that is **not** in the stopword list:

```
# as simple stopword list:
stopw = c("", "a", "an", "the", "of", "in", "by", "to", "for")

# identify position of stop words:
stopw.ind = tolower(aw.clean) %in% stopw
head(stopw.ind)

## [1] FALSE FALSE FALSE  TRUE FALSE  TRUE

# compare with the data:
head(aw.clean)

## [1] "Alice"      "s"           "Adventures" "in"          "Wonderland"
## [6] ""

# sorted frequencies of words not in the stop list:
aw.clean_no_stopw = aw.clean[ !stopw.ind ]
head( sort( table ( aw.clean_no_stopw ), decreasing = T ) )

## aw.clean_no_stopw
##     and   Alice Dodgson      he Liddell   story
##      13      12       8       5       5       5
```

4 Processing text to get data

We typically want more than one variable (e.g. frequency) to investigate a hypothesis, since language is subject to multiple types of constraints (grammatical, phonological, cogntive, socio-pragmatic, etc.). To achieve this, we require relevant **measurements** on our objects of study (be they words, sentences or texts).

We can get such data from a number of sources:

- the text itself
- corpus meta-data
- corpus annotation (e.g. syntactic annotation or PoS tagging)
- External data (e.g. ratings relevant to word senses or sentences)

Since this tutorial only deals with un-annotated text, we will look into how we can autmatise some measurements for the sentences constituting the text.

4.1 Find sentence lengths

A simple measurement to obtain is the number words in each sentence. This is a numeric measurement, which can be created in different ways (and using different functions). The code below takes the white space delimited strings from earlier and splits them by sentence. We can then count the number of white spaces to get an approximation of the number of words.

```
# split text string into sentences based on full stop. Result is a list of se
ntences
aw.sent = strsplit(aw.pasted, "\\. ", perl=T)

# count white space by comparing number of characters with and without white
space.
# Add 1 to get number of words:
 sent_len = nchar(aw.sent[[1]])- nchar(gsub(" ", "", aw.sent[[1]])) + 1
sent_len

##  [1]  26  21  17  15  23 106  18  23  15  18  32  30  26  37  25  34
```

4.2 Find sentences containing "and"

Another type of measurement is using a boolean value to classify a sentence as either having, or not having, a feature. In the example below, each of the sentences in the data is classified as either containing the word "and" (`TRUE`), or not (`FALSE`).

```
contains_and = grepl("\\band\\b", aw.sent[[1]])
head(contains_and)

## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE
```

Note: The word boundary regex character (`\\b`) is used to distinguish "and" from "Wonderl**and**".

Exercise 1: create a data frame

Now it's time to combine the measurements.

Create a data frame called `alice.df`. It should have 16 rows, one for each sentence, and two columns: one of numeric values indicating the number of words in the sentence, and the other a boolean indicating if the sentence contains "and".

The output should look like this:

```
##    sLen   and
## 1    26 FALSE
## 2    21 FALSE
## 3    17 FALSE
## 4    15 FALSE
## 5    23  TRUE
## 6   106  TRUE
## 7    18  TRUE
## 8    23 FALSE
## 9    15  TRUE
## 10   18 FALSE
## 11   32  TRUE
## 12   30  TRUE
## 13   26 FALSE
## 14   37 FALSE
## 15   25 FALSE
## 16   34  TRUE
```

Exercise 2: Explore results with plots

Make a box and whiskers plot to see if sentences containing "and" have a higher number of words in them than sentences without. Refer back to section 1.9 to see how box and whisker plots are made, if necessary.

Is there really a systematic difference between sentences with and without and?

Excercise 3: Find sentences with parentheses

Add a third column to `alice.df`, specifying if the sentence contains a parenthesis. Model your code on the "and" example above. Your output should look like this:

```
##    sLen   and paren
## 1    26 FALSE  TRUE
## 2    21 FALSE FALSE
## 3    17 FALSE FALSE
## 4    15 FALSE FALSE
## 5    23  TRUE FALSE
## 6   106  TRUE  TRUE
## 7    18  TRUE FALSE
## 8    23 FALSE FALSE
## 9    15  TRUE FALSE
## 10   18 FALSE FALSE
## 11   32  TRUE FALSE
## 12   30  TRUE FALSE
## 13   26 FALSE FALSE
## 14   37 FALSE FALSE
## 15   25 FALSE FALSE
## 16   34  TRUE FALSE
```

4.3 Create cross tables

Based on the data frame `alice.df` we can create matrices / cross tables using the `xtabs()` function.

When used as in the example below, `xtabs()` will count the number of combinations of the variables separated by `+`. In this case, how many times "and" occurs in sentences with and without parentheses, and the number of sentences with neither.

```
and_paren = xtabs( ~ and + paren, data = alice.df)
and_paren
```

```
##        paren
## and     FALSE TRUE
##    FALSE      8     1
##    TRUE       6     1
```

However, this ignores the measurements we created earlier: the length of the sentences. We can include this information by including the column holding sentence length in the formula, in front of the tilde (~) character:

```
and_paren_len = xtabs( sLen ~ and + paren, data = alice.df)
and_paren_len
```

```
##         paren
## and      FALSE TRUE
##    FALSE   182   26
##    TRUE    152  106
```

The cells now show the **sum** of the length of the sentences, broken down by whether or not the sentence contains "and" or a parenthesis.

However, this is still not quite right: what exactly does the sum of the number of words in sentences in the cells refer to?

Instead, we calculate the mean sentence length for each category by dividing the second matrix by the first:

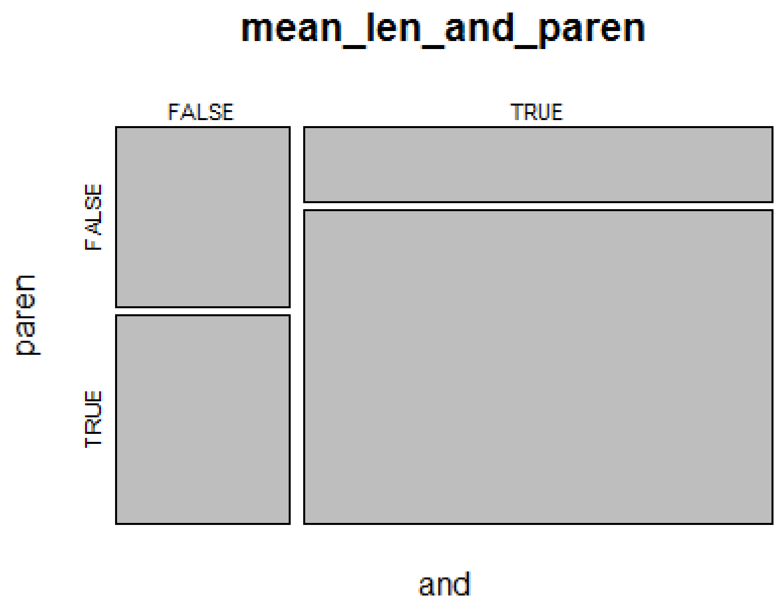
```
mean_len_and_paren = and_paren_len / and_paren
mean_len_and_paren
```

```
##         paren
## and          FALSE      TRUE
##    FALSE  22.75000  26.00000
##    TRUE   25.33333 106.00000
```

4.4 Visualize cross table results

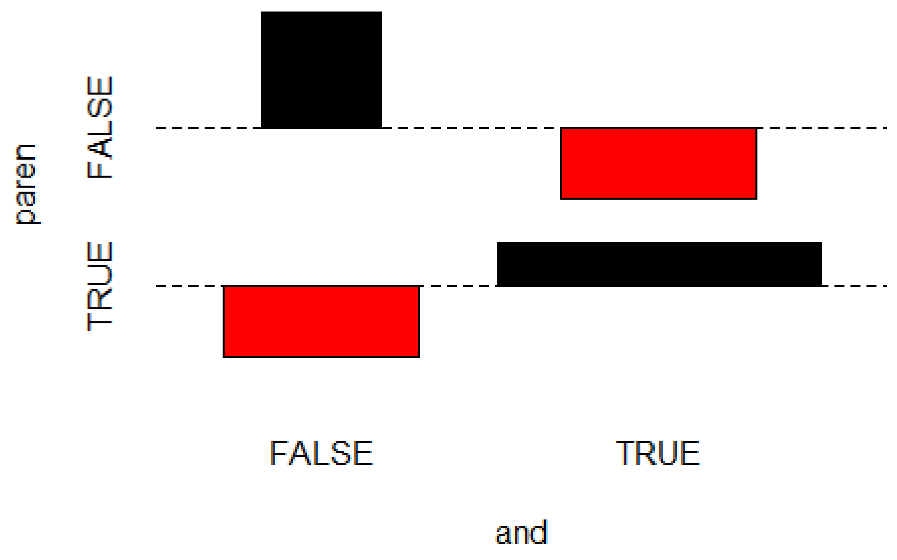
We can now visualize the matrix with the mean sentences lengths using different methods. The basic `plot()` function applied to a matrix creates a mosaic plot, where the size of the tile is proportional to the value of the corresponding cell.

```
plot(mean_len_and_paren)
```



An alternative is to the use the `assocplot()` function to produce a Cohen-Friendly plot, which shows the deviations (negative and positive) from the value expected under the Pearson's Chi-square test null-hypothesis:

```
assocplot(mean_len_and_paren)
```



Appendix

Introduction and background paragraphs from the Wikipedia article on Alice's Adventures in Wonderland, accessed May 20 2015.

Alice's Adventures in Wonderland (commonly shortened to Alice in Wonderland) is an 1865 novel written by English author Charles Lutwidge Dodgson under the pseudonym Lewis Carroll. It tells of a girl named Alice falling through a rabbit hole into a fantasy world populated by peculiar, anthropomorphic creatures. The tale plays with logic, giving the story lasting popularity with adults as well as with children. It is considered to be one of the best examples of the literary nonsense genre. Its narrative course and structure, characters and imagery have been enormously influential in both popular culture and literature, especially in the fantasy genre.

Alice was published in 1865, three years after Charles Lutwidge Dodgson and the Reverend Robinson Duckworth rowed in a boat, on 4 July 1862 (this popular date of the "golden afternoon" might be a confusion or even another Alice-tale, for that particular day was cool, cloudy and rainy), up the Isis with the three young daughters of Henry Liddell (the Vice-Chancellor of Oxford University and Dean of Christ Church): Lorina Charlotte Liddell (aged 13, born 1849, "Prima" in the book's prefatory verse); Alice Pleasance Liddell (aged 10, born 1852, "Secunda" in the prefatory verse); Edith Mary Liddell (aged 8, born 1853, "Tertia" in the prefatory verse).

The journey began at Folly Bridge near Oxford and ended five miles away in the village of Godstow. During the trip Dodgson told the girls a story that featured a bored little girl named Alice who goes looking for an adventure. The girls loved it, and Alice Liddell asked Dodgson to write it down for her. He began writing the manuscript of the story the next day, although that earliest version no longer exists. The girls and Dodgson took another boat trip a month later when he elaborated the plot to the story of Alice, and in November he began working on the manuscript in earnest.

To add the finishing touches he researched natural history for the animals presented in the book, and then had the book examined by other children-particularly the children of George MacDonald. He added his own illustrations but approached John Tenniel to illustrate the book for publication, telling him that the story had been well liked by children.

On 26 November 1864 he gave Alice the handwritten manuscript of Alice's Adventures Under Ground, with illustrations by Dodgson himself, dedicating it as "A Christmas Gift to a Dear Child in Memory of a Summer's Day". Some, including Martin Gardner, speculate there was an earlier version that was destroyed later by Dodgson when he wrote a more elaborate copy by hand.

But before Alice received her copy, Dodgson was already preparing it for publication and expanding the 15,500-word original to 27,500 words, most notably adding the episodes about the Cheshire Cat and the Mad Tea-Party.