NLP - Assignment 1

In this assignment you will. . .

- download a book from the Gutenbrg project & read it into R.
- perform simple processing steps and tokenize the text.
- analyze the frequency distribution of tokens.

The goal of this assignment is to become familiar with. . .

- importing of text into R.
- regular expressions.
- tokenization.
- word frequency distributions.

Load text into R

The first task of this assignment consists of choosing a book of your liking and loading it into R.

1) Visit the website of Project Gutenberg and select a book that you like. Tipp: check out Project Gutenberg's Top 100 list. To download the book visit the book's website and select 'Textdatei UTF-8'. Depending on your browser (and your browser settings) this will either download the file directly or open up the text in the browser tab. In the latter case, you can use right-click on the text and select 'save as' (or comparable) to download the text as a text-file to your hard-drive (preferably your project folder).

2) Load text into R using the `read_file()` function from the `readr` package. To use it, first load (and, if necessary, also install) the package using the `library()`-function (`install.packages()`). Then, use the `read_file()` function with the path to the text file as the (only) argument and assign the result to an object called, e.g., `text`. Note: Instead of taking the detour via your hard-drive, you can also provide the url to the text to load the text straight from the internet. Now you should have a text object containing a `vector` of type `character` that has `length` one. Confirm this using `typeof()` and `length()`. You may also want to evaluate the number of characters using `nchar()` and inspect the first, say, 1000 characters using the `str_sub()` function from the `stringr` package (remember to install & load the package). The `str_sub()` function takes three arguments: 1) the text, 2) the starting character index (e.g., 1), 3) and the ending character index (e.g., 1000).

```r
# load packages
require(readr)
```

```
## Loading required package: readr
```

```r
require(stringr)
```

```
## Loading required package: stringr
```

```r
# load text
text <- read_file('pg2197.txt')
#text <- read_file('http://www.gutenberg.org/cache/epub/2197/pg2197.txt')

# evaluate object
typeof(text)
```

```
## [1] "character"
```

```r
length(text)
```

```
## [1] 1
```
```r
nchar(text)
```
```
## [1] 357761
```
```r
str_sub(text, 1, 1000)
```
```
## [1] "The Project Gutenberg EBook of The Gambler, by Fyodor Dostoyevsky\r\n\r\n\r\nThis eBook is for the u
```

Tokenize text

3) Before you can tokenize the text, i.e, split the text into its individual words, you must remove several text sections added by the Project Gutenberg containing information on the text and the license of use. The sections are separated by header lines with leading and trailing star symbols, e.g., *** START OF THIS PROJECT GUTENBERG EBOOK THE GAMBLER ***. Build a regular expression that identifies such lines using the escaped star symbol *, curly brackets {} to indicate the number of symbol repetitions, the print class [:print:] for every letter in-between star symbols, and the plus + to indicate the number of print repetitions. Then, use the regular expression to split the text into its sections. To do this, use the str_split()-function. This will return a list of length one, with the only element being a character vector containing the individual sections. Store the list in an object called, e.g., text_split. Next, you can use the fact that the main text will be (in all likelihood) the section with the largest number of characters. Count the number of characters in each section using nchar() by passing to the function the character vector of sections (i.e., the first element in the list selected using [[1]]). Now you can create a main_text-object by assigning to it the largest element in the section vector (selected using [index]).

```r
# cut text into sections
text_split = str_split(text, '\\*{3}[:print:]*\\*{3}')

# count characters per section
nchar(text_split[[1]])
```
```
## [1]    502 338120      4   1299  17639
```
```r
# extract main text
main_text = text_split[[1]][2]
```

4) As a last measure before tokenzing the text, transform the main_text-object to lower-case only using str_to_lower(). Now, conveniently apply the str_extract_all() function extract the individual text tokens. To do this, use the [:alpha:] class and the plus + symbol to extract all sequences of alphabetic characters of length one and larger. This will automatically disregard all non-alphabetic characters. The result is again a list of length one. Select the first element and store it as, e.g., tokens.

```r
# to lower
tokens = str_to_lower(main_text, '[:alpha:]+')

# tokenize
tokens = str_extract_all(main_text, '[:alpha:]+')
```

Analyze frequency distribution

5) Now that you have a vector of tokens, you can begin analyzing the frequency distribution by counting the occurences of each word using table(). The table()-function will return a table object that you can treat, for the most part, like a named integer vector. To inspect the token frequencies sort them first using sort() with decreasing = TRUE so that the first positions in the table will be occupied by

the most frequent ones. This allows you to access the, say, 100 most frequent words using `[1:100]`
Note: `1:100` creates an integer vector with all values from 1 to 100. What are the most frequent words
and do they reveal much about the content?

```
# count tokens
token_frequencies = table(tokens)

# sort token table
token_frequencies = sort(token_frequencies, decreasing = TRUE)
```

6) In the previous task, you will have noticed that the most frequent tokens are words that are used in
any context and thus are not very uninformative with regard to the contents of your specific book. To
focus on words that are more relevant for the current content, try to remove 1) overly short words (e.g.,
words with less than five characters) and 2) stopwords. To prepare the former, create a logical vector
that indicates whether the tokens (tokens can be accessed via `names(tokens_frequencies)`) have less
than five characters (use `nchar()` and `<`). To prepare the latter, test which tokens are included in a
common stopword lists. A suitable stopword list can be obtained using the `stopwords()`-function from
`tm` package. If you evaluate a German book pass on `de` as the `kind` argument to access the German
stopword list. Now create a logical vector indicating whether tokens are included in the stopword list
using the is-element-of operator `%in%`. E.g., `c(1, 7) %in% c(1, 2, 3, 4, 5)` tests for each element in
the left-hand-side vector whether they are included in the right-hand-side vector, returning in this case
`c(TRUE, FALSE)`. Now you can combine the two logical vectors using the pipe `|` operator (represents
logical OR) to select cases where one or the other exclusion criterion is true and use the resulting logical
vector to extract relevant tokens. To do this negate the logical vector using `!`, which flips all `TRUE` to
`FALSE` and vice versa, and use it inside single brackets `[]`. Assign the result to a new object and inspect
again the first 100 elements. Do the most frequent words look more relevant now?

```
# load tm package
require(tm)
```

```
## Loading required package: tm
```

```
## Loading required package: NLP
```
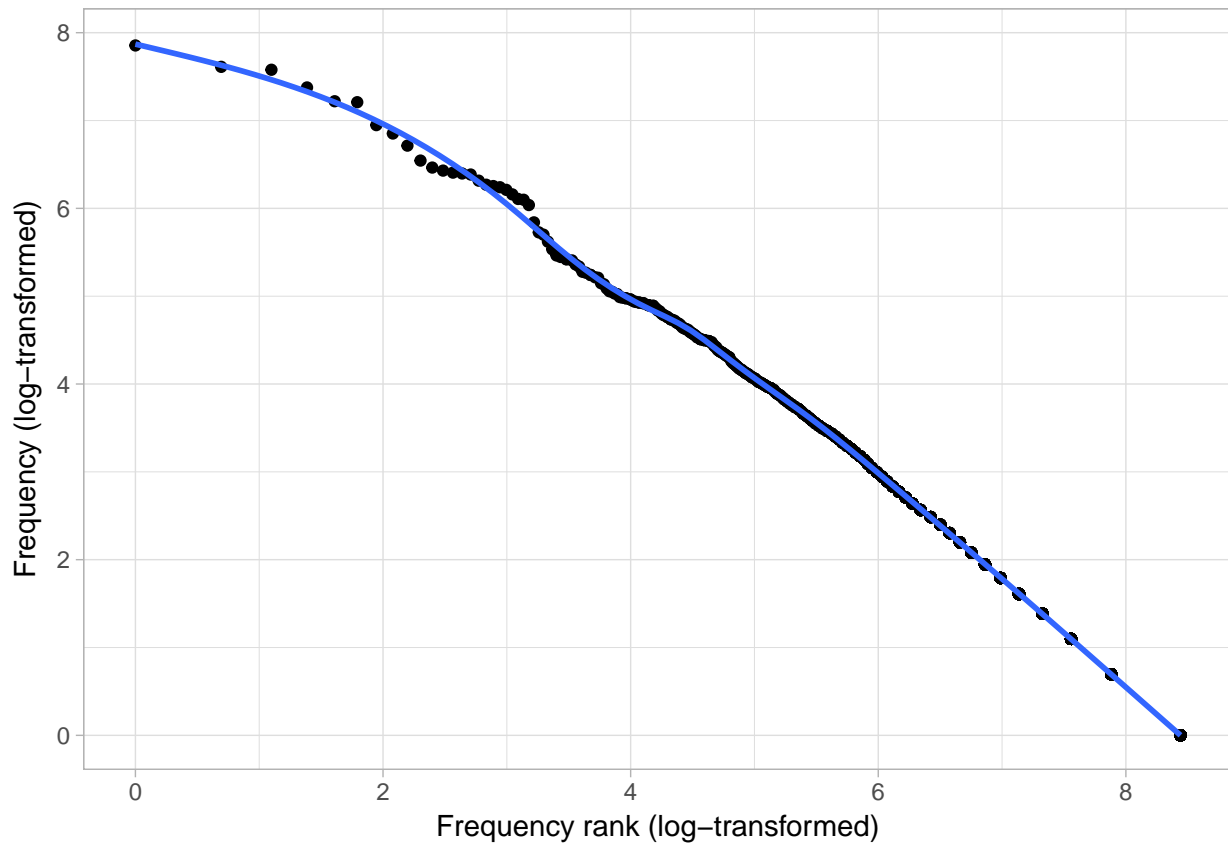
```
# create select variables
sel_by_length = nchar(names(token_frequencies)) < 5
sel_by_stopwords = names(token_frequencies) %in% stopwords('en')
sel = sel_by_length | sel_by_stopwords

# extract relevant tokens
token_frequencies_sel = token_frequencies[!sel]
```

7) One way to represent frequency distributions is via word clouds. Plot a word cloud using the
`wordcloud()`-function from the `wordcloud` package (install & load). The function takes two ar-
guments: 1) the token names (i.e., `names(tokens_frequencies)`) and 2) the token frequencies. For
the second argument the function expects a numeric vector. Thus, you cannot provide the table of
frequencies directly, but you have to transform it first to `numeric` using `as.numeric()`. You may find
that not all words fit into the plot. Try plotting only, say, the first few hundred words and play around
with the `scale` argument (see `?wordcloud`). You will also want to set the plot margins to zero prior
to using the function using `par(mar = c(0, 0, 0, 0)`. When you are happy with the plot, save the
image using the *Export* function atop your plot window and post the plot on **twitter** using *#nlpbasel*.

```
# load wordcloud package
require(wordcloud)
```

```
## Loading required package: wordcloud
```

```
## Loading required package: RColorBrewer
```

```r
# set margins to zero
par(mar = c(0, 0, 0, 0))

# plot word cloud
wordcloud(names(token_frequencies_sel[1:100]),as.numeric(token_frequencies_sel[1:100]),
          scale = c(2, .1))
```


8) As the final task, evaluate Zipf's law by plotting the token frequency (y-axis) against its rank (x-axis). To retrieve the rank apply `rank()` to the negative of the token frequencies so that the most frequent token receives rank 1, the second most frequent token receives rank 2, and so on. You will see that the token frequency diminishes dramatically as the frequency ranks go up, to the extent that frequencies of larger ranks can hardly be distinguished. A better illustration is, thus, achieved by computing the `log()` of both variables. If the plot now shows a nearly linear relationship between token frequency and rank then this is consistent with Zipf's-law. To create this plot use `ggplot2`. Plot points using `geom_point()` and add a smooth line using `geom_smooth()`. If you like, also play around with the `theme`. When you are happy with the plot, save the image using the *Export* function atop your plot window and post the plot on **twitter** using *#nlpbasel*.

```r
# load ggplot package
require(ggplot2)
```

```
## Loading required package: ggplot2
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:NLP':
##
##     annotate
```

```r
# plot Zipfian
ggplot(mapping = aes(log(rank(-1 * c(token_frequencies))), log(c(token_frequencies)))) +
  geom_point() + theme_light() + geom_smooth() +
  labs(x = 'Frequency rank (log-transformed)',
       y = 'Frequency (log-transformed)')
```

```
## `geom_smooth()` using method = 'gam'
```

9) BONUS: Evaluate the relationship between word length and frequency. Is this in line with the information theoretic account of communication according to which the most frequent words are assigned the shortest codes?

```
# plot n characters by word frequency
ggplot(mapping = aes(log(c(token_frequencies)), nchar(names(token_frequencies)))) +
  geom_count() + theme_light() + geom_smooth() +
  labs(x = 'Frequency (log-transformed)',
       y = 'Number of characters')
```

```
## `geom_smooth()` using method = 'gam'
```