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.

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 **Project Gutenberg** and select a book that you like. Check out Project Gutenberg's **Top 100** list.

2) To download the book go to the book's site 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).

3) Read the text into R using the `read_file()` function from the `readr` package. To use it, first load (if necessary, install the package first using `install.packages()`) the package using the `library()`-function. Use the `read_file()` function by providing it the full filepath to the text file as the (only) argument and assign the result to an object called, e.g., `text`. See example code below.

```
library(readr)

# example
text <- read_file(file = "folder/filename.txt")
```

```
# load packages
require(readr)

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

Note: Instead of taking the detour via your hard-drive, you can read the text directly from the internet by using the url of the text.

4) The resulting `text` object you should have of type `character` with length `1`. 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()` (e.g., `str_sub(text, 1, 1000)`) 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).

```
# str_sub example
str_sub(text, start = XX, end = XX)
```

```
require(stringr)

# evaluate text
typeof(text)
```

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

```
length(text)
```

```
## [1] 1
```

```
nchar(text)
```

```
## [1] 549811
```

```
str_sub(text, 1, 1000)
```

```
## [1] "The Project Gutenberg EBook of Grimms' Fairy Tales, by The Brothers Grimm\r\n\r\nThis eBook is
```

Tokenize text

1) Before the text can be *tokenized*, you must remove several header sections in the text 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 asterix symbols, e.g., `*** START OF THIS PROJECT GUTENBERG EBOOK THE GAMBLER ***`. Build a regular expression that identifies such lines. To do this, will will have to combine the escaped star symbol `\\*`, curly brackets `{3}` 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. Store the regular expression in an object called `regex`.

```
# Regex example - combine \\* {3} [:print:] +
regex <- "your_regex_code"
```

```
# Regex example - combine \\* {3} [:print:] +
regex <- '\\*{3}[:print:]*\\*{3}'
```

2) When you have defined your regular expression, use the regular expression in place of the `pattern`-argument in the `str_split()`-function to split the text into its sections. 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`.

```
# cut text into sections
text_split<- str_split(XX, pattern = XX)
```

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

3) Next, you have to extract from the `text_split` object the element that contains the actual text. To do this, you can use the fact that the actual text will be the section with the largest number of characters. Count the number of characters in each section by applying `nchar()` to vector of sections (i.e., `text_split[[1]]`).

```
# count number of characters per section
sections <- XX
nchar(sections)
```

```
# count number of characters per section
sections <- text_split[[1]]
nchar(sections)
```

```
## [1]    604 530064      4   1293  17629
```

4) Now that you know which element in `sections` has the largest number of characters, select that element using `sections[index]` and store it as `main_text`.

```
# select main text
main_text <- sections[XX]
```

```
# select main text
main_text <- sections[2]
```

5) As a last step before tokenizing change the entire text to lower case. This will later help in counting the words and is more efficiently performed prior to tokenizing. Simply apply `str_to_lower` to `main_text`.

```
# to lower case
main_text <- str_to_lower(main_text)
```

6) Now, use the `str_extract_all()` function to extract all individual text tokens, i.e., to tokenize the text. Create a regular expression consisting only of the `[:alpha:]` class and the plus `+` symbol. This defines a regular expression that identifies all sequences of alphabetic characters of length one and larger without any intermitting spaces or punctuation symbols. Use this regular expression in place of the `pattern` argument of the `str_extract_all()` function. Store the result in an object called `tokens`.

```
# tokenize
tokens = str_extract_all(main_text, 'your_regex_code')
```

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

7) `tokens` should, at this point, be a list of length one. Select the first element using `[[1]]` and overwrite the original `tokens` object so that `tokens` contains only the vector of tokens (not the list it was contained in).

```
# extract vector
tokens = tokens[[1]]
```

```
# extract vector
tokens = tokens[[1]]
```

Count tokens

1) Now its time to analyze the `tokens` vector. Begin 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. Apply `table()` to `tokens`.

```
# count tokens
table(tokens)
```

2) This was a long object, right? Store the resulting table in an object called `token_frequencies` and sort it using `sort(XX, decreaing = TRUE)` function.

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

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

3) Look at the first 100 entries in `token_frequencies` using `token_frequencies[1:100]`. What are the most frequent words?

4) Finally, create a tibble from `token_frequencies` using the `as_tibble()` function (from the `tibble` package - install and load) and call it `tokens_tbl`.

```
require(tibble)

# to tibble
token_tbl = as_tibble(token_frequencies)
```

Zipf's law

1) Now that you know the, evaluate Zipf's law. First, create a new variable in `token_tbl` called `rank` containing the token's rank using the `rank()` function. Important: Don't forget the minus as we want to assign the smallest rank to the largest n.

```
# to tibble
token_tbl$XX = rank(-token_tbl$XX)
```

```
# to tibble
token_tbl$rank = rank(-token_tbl$n)
```

2) Plot the relationship between the tokens frequency (`n`) and its `rank` using the code below. Does it look liek Zipf's law holds here?

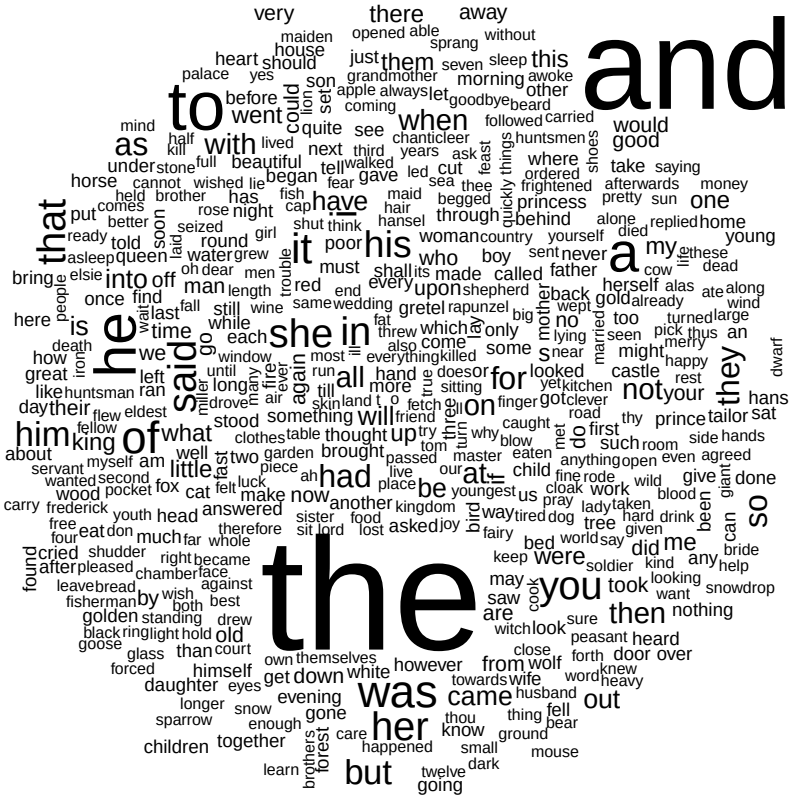
```
# load ggplot package
require(ggplot2)

# plot Zipfian
ggplot(data = token_tbl,
       mapping = aes(x = log(rank), y = log(n))) +
  geom_point() + theme_light() + geom_smooth() +
  labs(x = 'Frequency rank (log-transformed)',
       y = 'Frequency (log-transformed)')
```

Wordcloud

1) Word frequencies can also be used to learn more about the contents of a document, such as the book you are analysing. The idea is that the most frequent words should characterize what the book is about. A nice way to illustrate this is via a wordcloud. Use the code below (you will have to install `wordcloud` first).

```
require(wordcloud)

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

# plot word cloud
wordcloud(token_tbl$XX[1:500],
          freq = token_tbl$XX[1:500],
          scale = c(5, .5))
```

```
require(wordcloud)

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

# plot word cloud
wordcloud(token_tbl$tokens[1:500],
          freq = token_tbl$n[1:500],
          scale = c(5, .5))
```



2) You probably noticed that the most important words were mostly uninformative. To address this probem a typical approach is to remove socalled stopwords, which don't carry a lot of meaning. Create a new tibble without stopwords using the code below (you will have to install `tidytext` first) and create a new word cloud. Note: the variable `tokens` has been renamed to `word`.

```
require(tidytext)

# create tibble without stopwords
token_tbl_nostop <- token_tbl %>%
  rename(word = tokens) %>%
  anti_join(get_stopwords('en'), by = c('word'))
```

```
# plot word cloud
wordcloud(token_tbl_nostop$word[1:500],
          freq = token_tbl$n[1:500],
          scale = c(5, 1))
```

BONUS: Word length x Frequency

1) 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')
```