Lab 1 - Manipulating text data with R

Adrien Guille

03/10/2018

For this lab, we need two packages, text2vec, to compute the vocabulary and vectorize the corpus, and Matrix, to manipulate the sparse matrices generated with text2vec.

```
library(text2vec)
library(Matrix)
```

Part 1 - Analyzing a collection of movie reviews

1 - Load the corpus

The corpus is a collection of movie reviews written in English. First, we load the content of the CSV file into a data frame:

```
corpus <- read.csv('../Data/reviews.csv', stringsAsFactors=FALSE)
colnames(corpus)
```

```
## [1] "doc_id"    "text"      "sentiment"
```

There are three fields in this CSV: doc_id, text and sentiment. For this lab, we only need the content of the reviews. Let's look at the 50 first characters of the first review:

```
substr(corpus$text[1], 1, 50)
```

```
## [1] "plot : two teen couples go to a church party , dri"
```

Even though we don't care about the sentiment of the reviews for now, we can still look at how many positive reviews there are:

```
cat("There are", nrow(corpus), "reviews, out of which",
    nrow(corpus[which(corpus$sentiment=='pos'), ]), "are positive reviews.")
```

```
## There are 2000 reviews, out of which 1000 are positive reviews.
```

2 - Compute the vocabulary

We instanciate an iterator to transform the text into a sequence of lowercased unigrams and then compute the vocabulary:

```
iterator <- itoken(corpus$text,
                   preprocessor=tolower, # replace capital letters
                   tokenizer=word_tokenizer, # split the text into unigrams
                   progressbar=FALSE)
vocabulary <- create_vocabulary(iterator)
n_words <- nrow(vocabulary)
n_tokens <- sum(vocabulary$term_count)
cat("Number of word types:", n_words, "\nNumber of tokens:", n_tokens)
```

```
## Number of word types: 42392
## Number of tokens: 1309372
```

The vocabulary is a table; each row consist of a word (i.e. term), its overall frequency (i.e. term_count) and the number of documents it occurs in (i.e. doc_count):

```
head(vocabulary)
```

```
## Number of docs: 2000
## 0 stopwords:  ...
## ngram_min = 1; ngram_max = 1
## Vocabulary:
##           term term_count doc_count
## 1:       liken          1         1
## 2:  injections          1         1
## 3:  centrifuge          1         1
## 4: overkilling          1         1
## 5:     flossed          1         1
## 6:   artillary          1         1
```

Identify the 10 most common words

We sort the vocabulary in decreasing order w.r.t word frequency (i.e. term_count) and print the first 10 entries:

```
ordered_vocabulary <- vocabulary[order(-vocabulary$term_count), ]
head(ordered_vocabulary, 10)
```

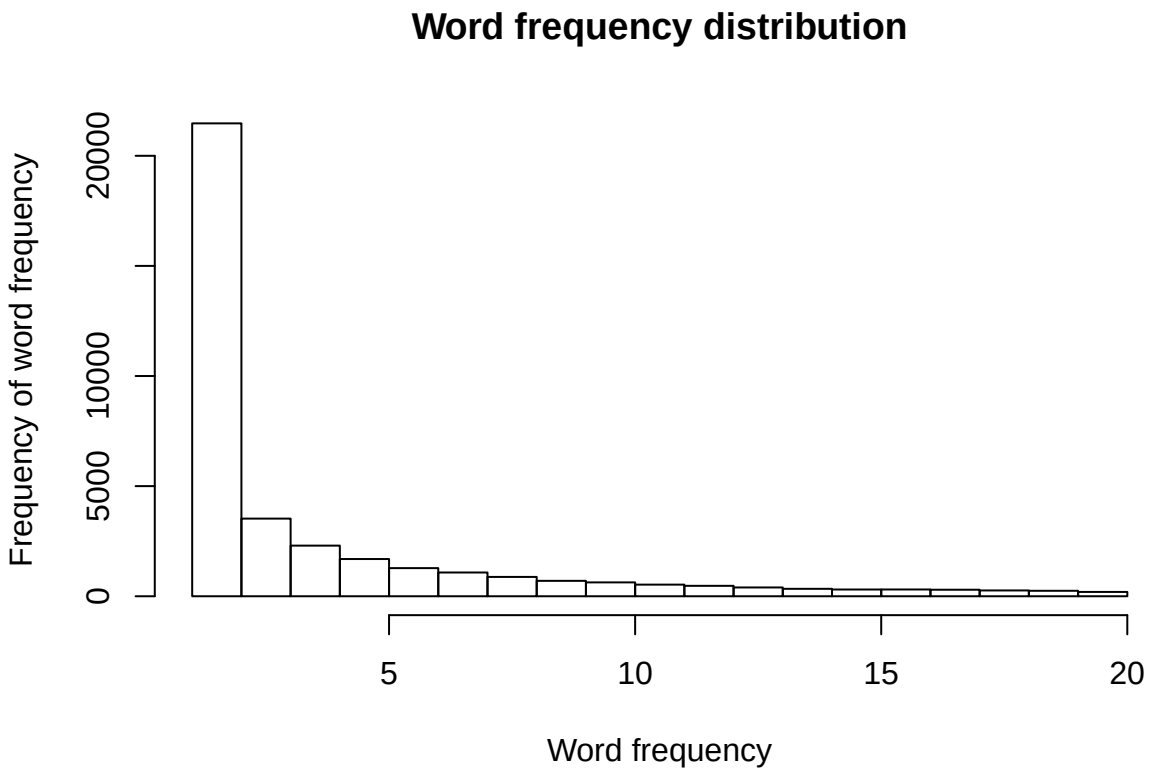
```
## Number of docs: 2000
## 0 stopwords:  ...
## ngram_min = 1; ngram_max = 1
## Vocabulary:
##     term term_count doc_count
##  1:  the      76562      1999
##  2:    a      38104      1996
##  3:  and      35576      1998
##  4:   of      34123      1998
##  5:   to      31937      1997
##  6:   is      25195      1995
##  7:   in      21821      1994
##  8: that      15129      1957
##  9:   it      12352      1935
## 10:   as      11378      1920
```

We get the usual stop-words, which occur in almost all documents.

Plot the distribution of word frequency

For the sake of readability, we select the sub-vocabulary of words that occur at most 20 times, then plot the histogram of word frequency:

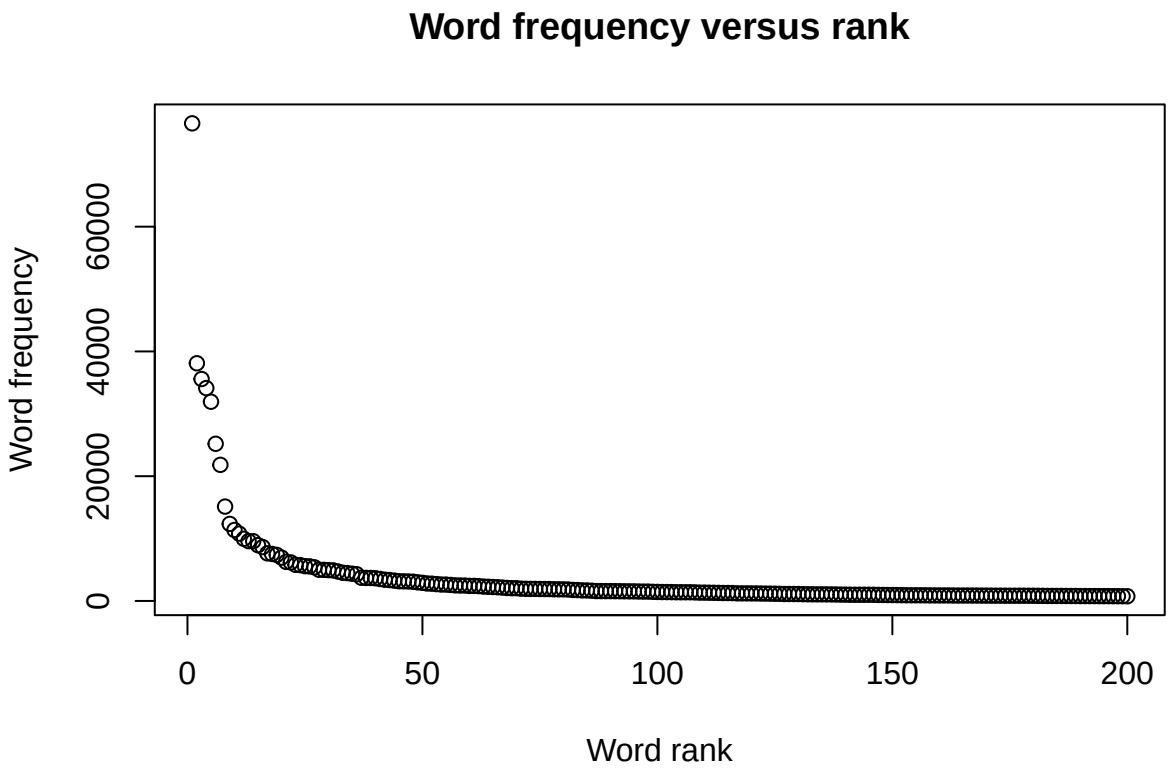
```
vocabulary_20 <- vocabulary[which(vocabulary$term_count <= 20), ]
histogram <- hist(vocabulary_20$term_count,
                  breaks=20,
                  main='Word frequency distribution',
                  xlab='Word frequency',
                  ylab='Frequency of word frequency')
```



3 - Plot word frequency versus rank

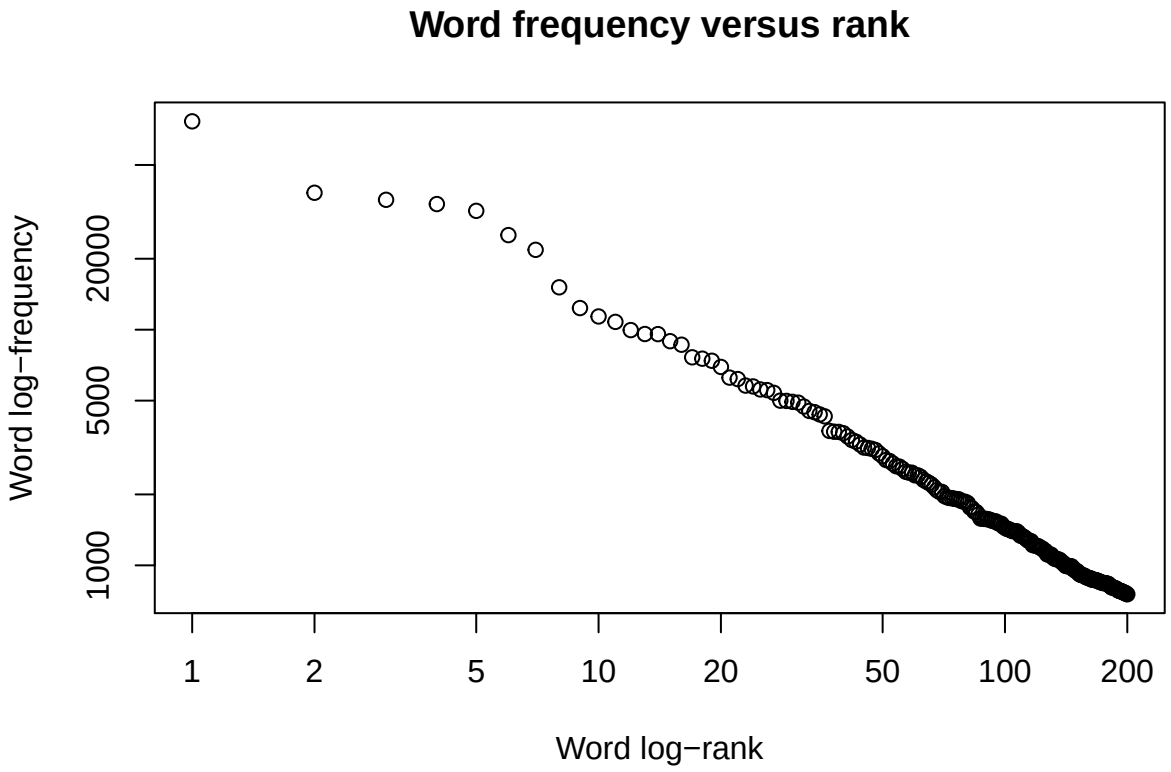
First, we plot word frequency versus word rank (i.e. position in the ordered vocabulary) for the 200 most frequent words:

```
frequency <- ordered_vocabulary$term_count[1:200]
plot(frequency,
     main='Word frequency versus rank',
     xlab='Word rank',
     ylab='Word frequency')
```



Then, we plot the same data with logarithmic axes. We observe kind of a straight-line, which is typical of power law relationships:

```
plot(frequency,
     main='Word frequency versus rank',
     xlab='Word log-rank',
     ylab='Word log-frequency',
     log='xy')
```



4 - Fit Zipf's law

Zipf's law models the relationship between the frequency of a word, f_r, and its rank, r:

$$f_r \simeq f_{\max} \frac{1}{r^{-k}}$$

In the log space, it becomes:

$$\log(f_r) \simeq \log(f_{\max}) + k \log(r)$$

Estimate the parameters of the power law

We estimate the parameters via least-square fitting, via the lm function:

```
log_frequency <- log(frequency)
log_rank <- log(c(1:200))
model <- lm(log_frequency ~ log_rank)
model$coefficients
```

```
## (Intercept)    log_rank
##  11.6419318  -0.9479958
```

We get a value of k close to -1, which is typical for English.

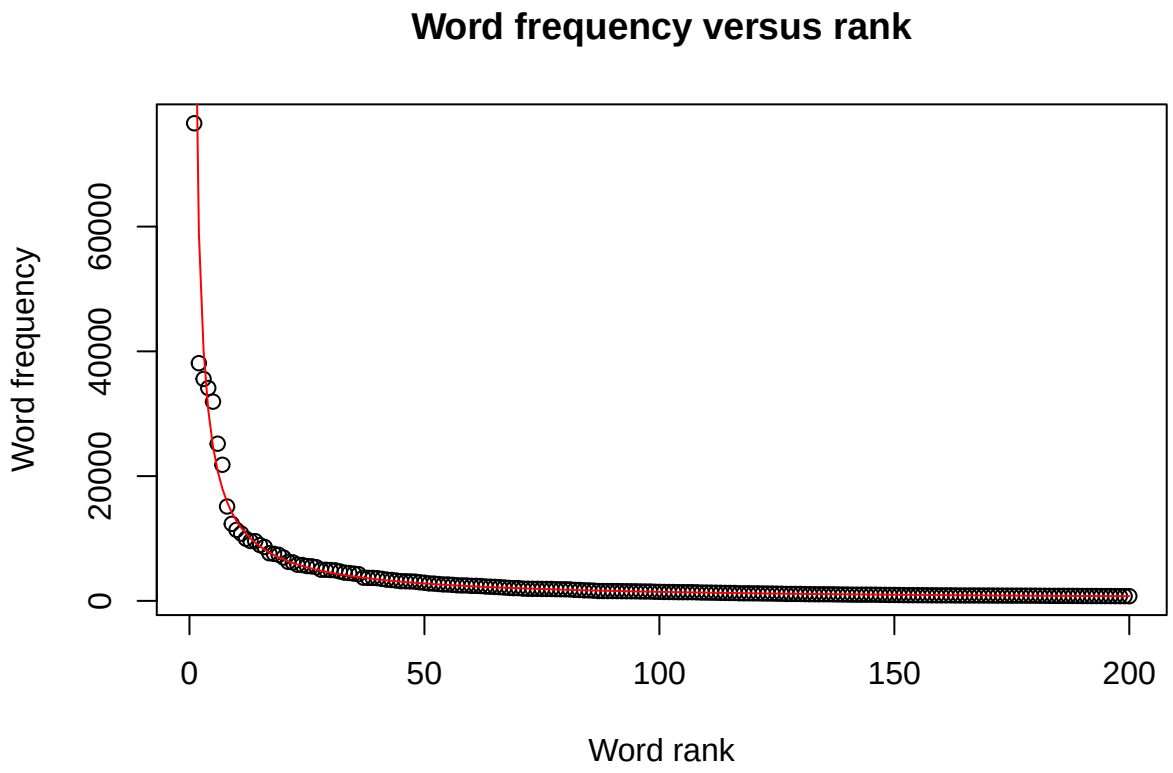
Plot the estimation of frequency vs. rank

We write a function that returns the estimation of log the frequency of a word according to this model, given its rank:

```
estimate_log_frequency <- function(r){
    return(as.numeric(model$coefficients[1])
           + as.numeric(model$coefficients[2]) * log(r))
}
```

We plot the estimation in the original space, thus we take the exponential of the estimation:

```
estimated_frequency <- exp(sapply(c(1:200), estimate_log_frequency))
plot(frequency,
     main='Word frequency versus rank',
     xlab='Word rank',
     ylab='Word frequency')
lines(estimated_frequency, col='red')
```



5 - Vectorize the corpus

Prune the vocabulary

In order to contain the dimension of the document-term matrix (i.e. dtm), we prune the vocabulary:

```
pruned_vocabulary <- prune_vocabulary(vocabulary,
                                      doc_proportion_max=0.5,
                                      term_count_min=10)
nrow(pruned_vocabulary)
```

```
## [1] 9389
```

Only 10 000 word types satisfy the pruning conditions.

Construct the document-term matrix

We instanciate a vetorizer based on the pruned vocabulary and create the document-term matrix using the iterator we've defined previously:

```
vectorizer = vocab_vectorizer(pruned_vocabulary)
dtm = create_dtm(iterator, vectorizer)
```

6 - Implement cosine similarity

The cosine similarity between two documents, d_1 and d_2, is given by:

$$\text{cosine similarity}(d_1, d_2) = \frac{d_1 \cdot d_2}{||d_1||\ ||d_2||}$$

```
cosine_similarity <- function(d1, d2){
  dot_product <- d1 %*% d2
  norm_prod <- sqrt(sum(d1**2)) * sqrt(sum(d2**2))
  return(as.numeric(dot_product/norm_prod))
}
```

7 - Assess the impact of tf-idf weighting

We apply tf-idf weighting to the original document-term matrix:

```
tfidf <- TfIdf$new(smooth_idf = TRUE, sublinear_tf = TRUE)
tfidf_dtm <- tfidf$fit_transform(dtm)
```

We measure the cosine similarity between the first review and the next five ones in the corpus, using both document-term matrices:

```
raw_sim <- numeric(5)
tfidf_sim <- numeric(5)
for(i in 2:6){
  raw_sim[i-1] <- cosine_similarity(dtm[1, ], dtm[i, ])
  tfidf_sim[i-1] <- cosine_similarity(tfidf_dtm[1, ], tfidf_dtm[i, ])
}
```

We sort the five reviews w.r.t their similarity with the first review:

```
order(-raw_sim)
```

```
## [1] 2 1 5 3 4
```

```
order(-tfidf_sim)
```

```
## [1] 2 5 4 3 1
```

Even though the closest review is the same in both cases, the rest of the rankings are quite different.

Part 2 - Analyzing random text

1 - Generate random text following Li's procedure

We generate random text from an alphabet of 4 letters, plus the space character to delimit tokens. The maximum length of a word is fixed to 4 characters. We generate a biased random character sequence according to the probability distribution suggested by Li (p(a)=0.5, p(b)=0.13, p(c)=0.1, p(d)=0.07, p(space)=0.2):

```
M <- 4
alphabet <- letters[1:M]
alphabet[M+1] <- ' '
probabilities <- c(0.5, 0.13, 0.1, 0.07, 0.2)
random_text <- ""
max_word_length <- 4
current_word_length <- 0

generate_random_text <- function(n_char){
  for(i in 1:n_char){
      next_character <- sample(alphabet, 1, prob=probabilities)
      if(next_character == ' '){
          current_word_length <- 0
      }else{
          current_word_length <- current_word_length + 1
      }
      if(current_word_length > max_word_length){
          next_character <- ' '
          current_word_length <- 0
      }
      random_text <- paste(random_text, next_character, sep='')
  }
  return(random_text)
}
```

We generate a sequence of 200 characters to see what a random text looks like:

```
generate_random_text(200)
```

```
## [1] "da a  baaa caab abab  ad accb aa  aaa  aaaa ccc ba daac  c b d  ca aaaa ba  a  acaa  ca bcab aba
```

We generate a sequence of 10 000 characters and compute the vocabulary:

```
random_text <- generate_random_text(10**4)
iterator_r <- itoken(random_text,
                     tokenizer = word_tokenizer)
vocabulary_r <- create_vocabulary(iterator_r)
n_words_r <- nrow(vocabulary_r)
n_tokens_r <- sum(vocabulary_r$term_count)
cat("Number of distinct words: ",
    n_words_r,
    "\nNumber of tokens: ",
    n_tokens_r)
```

```
## Number of distinct words:  231
## Number of tokens:  2367
```

2 - Fit Zipf's law on this text

As in Part 1, we fit Zipf's law parameters in log-space:

```
frequency_r <- rev(vocabulary_r$term_count)[1:200]
log_frequency_r <- log(frequency_r)
log_rank_r <- log(c(1:200))
model_r <- lm(log_frequency_r ~ log_rank_r)
model_r$coefficients
```

```
## (Intercept)  log_rank_r
##    6.705998   -1.195138
```

Again, we get a value close to -1. Finally, we plot the estimation on top of the data:

```
estimate_log_frequency_r <- function(r){
    return(as.numeric(model_r$coefficients[1])
           + as.numeric(model_r$coefficients[2]) * log(r))
}
estimated_frequency_r <- exp(sapply(c(1:200), estimate_log_frequency_r))
plot(frequency_r,
     main='Word frequency versus rank (Random text)',
     xlab='Word rank',
     ylab='Word frequency')
lines(estimated_frequency_r, col='red')
```