

WZB



Wissenschaftszentrum Berlin
für Sozialforschung

R Tutorial at the WZB

08 - Text mining I

Markus Konrad

December 13, 2018

Today's schedule

1. Review of last week's tasks
2. Text as data
3. Text mining methods for the Social Sciences
4. Matrices and lists in R
5. Bag-of-words model
6. Practical text mining with the `tm` package (I)

Review of last week's tasks

Solution for tasks #7

now online on

https://wzbsocialsciencecenter.github.io/wzb_r_tutorial/

Text as data

Text as data

Natural language is context-dependent, loosely structured and often ambiguous. This makes extracting structured information hard.

Text mining (TM) or text analytics tries to uncover structured key information from natural language text.

Other important fields:

- **Natural language processing (NLP):** deals with understanding and generating natural language (Amazon Echo, Apple Siri, etc.)
- **Quantitative text analysis (QTA):** "[...] extracting quantitative information from [...] text for social scientific purposes [...]"
([Ken Benoit](#))

Key terms in TM: Text corpus

Text material is compiled to a **corpus**. This is the data base for TM contains a set of **documents**. Each document has:

1. A unique name
2. Its raw text (machine-readable but unprocessed text)
3. Additional variables used during analysis, e.g. author, date, etc.
4. Meta data (variables not used during analysis, e.g. source)

Documents can be anything: news articles, scientific papers, twitter posts, books, paragraphs of books, speeches, etc.

Usually, you don't mix different sorts of text within a corpus.

Key terms in TM: Tokens/terms

A **token** is the lexical unit you work with during your analysis. This can be phrases, **words**, symbols, characters, etc.

→ ~ unit of measurement in your TM project.

Even if you initially use words as lexical unit, a tokenized and processed word might not be a lexicographically correct word anymore.

Example that employs stemming and lower-case transformation:

"I argued with him" → ["i", "argu", "with", "he"]

Tokens are also called **terms**.

Text mining applications

What can you find out with text mining? A few key methods often employed in the Soc. Sciences:

1. Simple & weighted word frequency comparisons

Count the words that occur in each document, calculate proportions, compare.

Weighted frequencies: Increase importance of document-specific words, reduce importance of very common words

→ key concept: term frequency - inverse document frequency (tf-idf).

1. Simple & weighted word frequency comparisons



2. Word co-occurrence and correlation

[illegible]

WZB 
Wissenschaftszentrum Berlin
für Sozialforschung

Text mining applications

3. Document classification

Approach:

1. Train a machine learning model with labelled documents
2. Evaluate model performance (estimate accuracy using held-out labelled data)
3. Classify unlabelled documents (prediction)

Examples:

- binary classification (spam / not spam, hate-speech / not hate-speech, ...)
- multiclass classification (predefined political categories, style categories, ...)

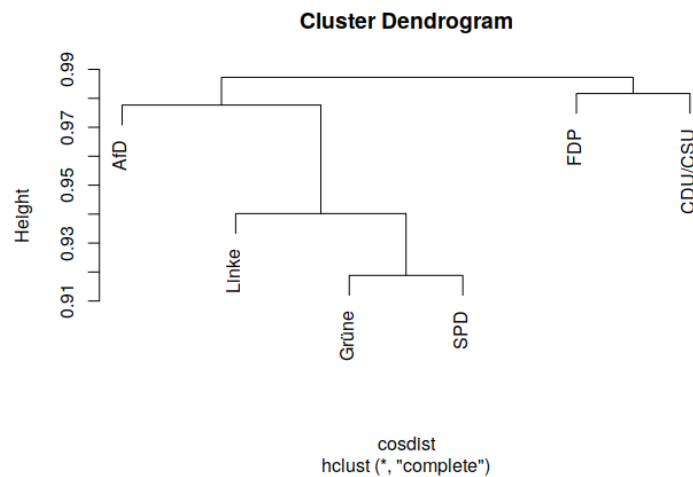
Text mining applications

4. Document similarity and clustering

How similar is document A as compared to document B?

Mostly used with word frequencies → compare (weighted) word usage between documents.

Once you have similarity scores for documents, you can cluster them.



Hierarchical clusters of party manifestos for Bundestag election 2017

Text mining applications

5. Term similarity and edit-distances

Term similarity work on the level of terms and their (phonetic, lexicographic, etc.) similarity. Edit-distances are often used to measure the editing difference between two terms or two documents A, B (how many editing steps to you need to come from A to B?).

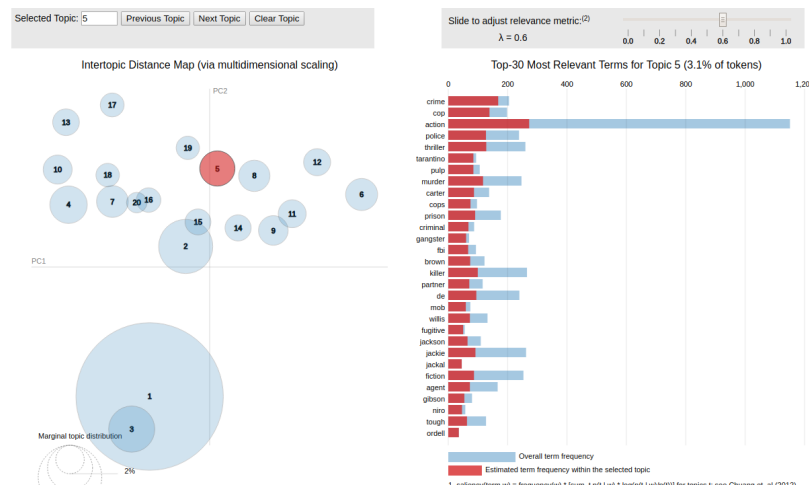
Example: Levenshtein distance between "kitten" and "sitting" is 3 edits.

Practical example: Measure how much drafts for a law changed over time.

Text mining applications

6. Topic modeling

Unsupervised machine learning approach to find latent topics in text corpora. Topics are distributions across words. Each document can be represented as a mixture of topics.



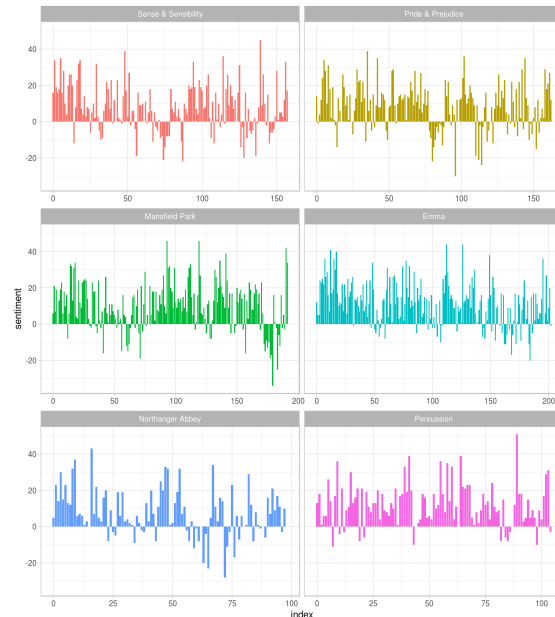
Practical example: Measure how the presence of certain topics changed over time in parliamentary debates; differences between parties, etc.

Text mining applications

7. Sentiment analysis

Also known as opinion mining. In its basic form, it tries to find out if the sentiment in a document is positive, neutral or negative by assigning a sentiment score.

This score can be estimated by using supervised machine learning approaches (using training data of already scored documents) or in a lexicon-based manner (adding up the individual sentiment scores for each word in the text).



source: [Silge & Robinson 2018: Text Mining with R](#)

Text mining applications

Named entity recognition: Find out company names, people's names, etc. in texts.

Gender (from name) prediction: Estimate the gender of a person (for example from a name).

... and much more

Text mining steps

TM consists of several steps, each of them applying a variety of methods:

1. Collection of text material into a corpus
2. Text processing (tokenization and normalization of the corpus)
3. Feature extraction (extracting structured information)
4. Modeling

Which steps and methods you apply depends on your material and the modeling approach.

Packages for TM in the R world

- [tm](#), Feinerer et al.
 - extensive set of tools for text mining
 - developed since 2008
- [tidytext](#), [Silge & Robinson 2018](#)
 - text preprocessing, topic modeling, sentiment analysis in the "tidyverse"
 - designed for English language text
- [quanteda](#), Benoit et al.
 - newly developed, extensive framework
 - also non-English texts

Specific methods:

- Topic modeling: [topicmodels](#), [lda](#), [stm](#)
- Text classification: [RTextTools](#)
- Word embeddings and similarities: [text2vec](#)

Matrices and lists in R

Matrices

The `matrix` structure stores data in a matrix with `m` rows and `n` columns. Each value must be of the same data type (type coercion rules apply).

To create a matrix, specify the data and its dimensions:

```
matrix(1:6, nrow = 2, ncol = 3)
```

```
##      [,1] [,2] [,3]  
## [1,]    1    3    5  
## [2,]    2    4    6
```

Matrices

The `matrix` structure stores data in a matrix with `m` rows and `n` columns. Each value must be of the same data type (type coercion rules apply).

To create a matrix, specify the data and its dimensions:

```
matrix(1:6, nrow = 3, ncol = 2)
```

```
##      [,1] [,2]  
## [1,]    1    4  
## [2,]    2    5  
## [3,]    3    6
```

Matrices

The `matrix` structure stores data in a matrix with `m` rows and `n` columns. Each value must be of the same data type (type coercion rules apply).

To create a matrix, specify the data and its dimensions:

```
# fill data in rowwise order  
matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE)
```

```
##      [,1] [,2] [,3]  
## [1,]    1    2    3  
## [2,]    4    5    6
```

Indexing matrices

```
(A <- matrix(1:6, nrow = 2, ncol = 3, byrow = TRUE))
```

```
##      [,1] [,2] [,3]  
## [1,]    1    2    3  
## [2,]    4    5    6
```

The same indexing rules as for data frames apply. Individual cells are selected by `[row index, column index]`:

```
A[2, 3]
```

```
## [1] 6
```

Rows are selected by `[row index,]`:

```
A[2, ]
```

```
## [1] 4 5 6
```

Columns are selected by `[, column index]`:

```
A[, 3]
```

```
## [1] 3 6
```


Matrix operations

A

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Matrix **B** with dimensions 3x3:

```
(B <- matrix(rep(1:3, 3), nrow = 3, ncol = 3, byrow = TRUE))
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    1    2    3
## [3,]    1    2    3
```

Matrix multiplication:

A %*% B

```
##      [,1] [,2] [,3]
## [1,]    6   12   18
## [2,]   15   30   45
```

Matrix operations

A

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Matrix C with same dimensions as A:

```
(C <- matrix(6:1, nrow = 2, ncol = 3, byrow = TRUE))
```

```
##      [,1] [,2] [,3]
## [1,]    6    5    4
## [2,]    3    2    1
```

Matrix addition:

A + C

```
##      [,1] [,2] [,3]
## [1,]    7    7    7
## [2,]    7    7    7
```

Matrix operations

A

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Matrix C with same dimensions as A:

```
(C <- matrix(6:1, nrow = 2, ncol = 3, byrow = TRUE))
```

```
##      [,1] [,2] [,3]
## [1,]    6    5    4
## [2,]    3    2    1
```

Element-wise multiplication:

A * C

```
##      [,1] [,2] [,3]
## [1,]    6   10   12
## [2,]   12   10    6
```

Matrix operations

A

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

Rowwise normalization of A:

rowSums(A)

```
## [1]  6 15
```

A / rowSums(A)

```
##      [,1] [,2] [,3]
## [1,] 0.1666667 0.3333333 0.5
## [2,] 0.2666667 0.3333333 0.4
```

Transpose:

t(A)

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Row and column names for matrix

As with data frames, row names and column names can optionally be set via `rownames()` and `colnames()`:

```
A
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
rownames(A) <- c('row1', 'row2')
colnames(A) <- c('col1', 'col2', 'col3')
A
```

```
##      col1 col2 col3
## row1     1     2     3
## row2     4     5     6
```

```
A['row2',]
```

```
## col1 col2 col3
##     4     5     6
```

Lists

In contrast to vectors and matrices, lists can contain elements of different types:

```
list(1:3, 'abc', 3.1415, c(FALSE, TRUE, TRUE, FALSE))
```

```
## [[1]]  
## [1] 1 2 3  
##  
## [[2]]  
## [1] "abc"  
##  
## [[3]]  
## [1] 3.1415  
##  
## [[4]]  
## [1] FALSE TRUE TRUE FALSE
```

Lists

You can think of a list as arbitrary "key-value" data structure. For each unique "key" (i.e. index), a list can hold a value of arbitrary type, even another list.

```
l <- list(a = 1:3, b = 'abc', c = 3.1415,  
          d = c(FALSE, TRUE, TRUE, FALSE),  
          e = list(1, 2, 3))  
str(l)
```

```
## List of 5  
## $ a: int [1:3] 1 2 3  
## $ b: chr "abc"  
## $ c: num 3.14  
## $ d: logi [1:4] FALSE TRUE TRUE FALSE  
## $ e:List of 3  
## ..$ : num 1  
## ..$ : num 2  
## ..$ : num 3
```

Indexing a list

If no key is given, the default keys are set as 1 to N:

```
(l <- list(1:3, 'abc', 3.1415, c(FALSE, TRUE, TRUE, FALSE)))
```

```
## [[1]]  
## [1] 1 2 3  
##  
## [[2]]  
## [1] "abc"  
##  
## [[3]]  
## [1] 3.1415  
##  
## [[4]]  
## [1] FALSE TRUE TRUE FALSE
```

Indexing with single square brackets **always results in a new list** (here, containing only a single element):

```
l[4]
```

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


Indexing a list

If no key is given, the default keys are set as 1 to N:

```
(l <- list(1:3, 'abc', 3.1415, c(FALSE, TRUE, TRUE, FALSE)))
```

```
## [[1]]  
## [1] 1 2 3  
##  
## [[2]]  
## [1] "abc"  
##  
## [[3]]  
## [1] 3.1415  
##  
## [[4]]  
## [1] FALSE TRUE TRUE FALSE
```

Use double square brackets to get the actual element as vector:

```
l[[4]]
```

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

Indexing a list

We can explicitly define keys for a list:

```
l <- list(a = 1:3, b = 'abc', c = 3.1415,  
          d = c(FALSE, TRUE, TRUE, FALSE),  
          e = list(1, 2, 3))  
str(l)
```

```
## List of 5  
## $ a: int [1:3] 1 2 3  
## $ b: chr "abc"  
## $ c: num 3.14  
## $ d: logi [1:4] FALSE TRUE TRUE FALSE  
## $ e:List of 3  
## ..$ : num 1  
## ..$ : num 2  
## ..$ : num 3
```

The same rules for single and double square brackets apply:

```
l['d']
```

```
## $d  
## [1] FALSE TRUE TRUE FALSE
```

Indexing a list

We can explicitly define keys for a list:

```
l <- list(a = 1:3, b = 'abc', c = 3.1415,  
          d = c(FALSE, TRUE, TRUE, FALSE),  
          e = list(1, 2, 3))  
str(l)
```

```
## List of 5  
## $ a: int [1:3] 1 2 3  
## $ b: chr "abc"  
## $ c: num 3.14  
## $ d: logi [1:4] FALSE TRUE TRUE FALSE  
## $ e:List of 3  
## ..$ : num 1  
## ..$ : num 2  
## ..$ : num 3
```

The same rules for single and double square brackets apply:

```
l[['d']]
```

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

Indexing a list

We can explicitly define keys for a list:

```
l <- list(a = 1:3, b = 'abc', c = 3.1415,  
          d = c(FALSE, TRUE, TRUE, FALSE),  
          e = list(1, 2, 3))  
str(l)
```

```
## List of 5  
## $ a: int [1:3] 1 2 3  
## $ b: chr "abc"  
## $ c: num 3.14  
## $ d: logi [1:4] FALSE TRUE TRUE FALSE  
## $ e:List of 3  
## ..$ : num 1  
## ..$ : num 2  
## ..$ : num 3
```

A shortcut to access elements in a list by key is the dollar symbol:

```
l$d      # same as l[['d']]
```

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

Bag-of-words model

Bag-of-words model

Bag-of-words is a simple, but powerful representation of a text corpus.

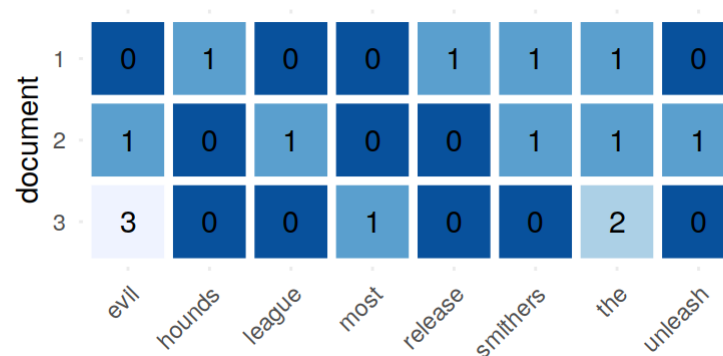
- each document in a corpus is "bag of its words"
 - store which words occur and how often do they occur
 - disregard grammar, word order
- basis for:
 - word frequency, co-occurrence
 - document similarity and clustering
 - topic modeling
 - text classification, etc.
- result is a document term matrix (DTM) (also: document feature matrix)

Bag-of-words model – example

Three documents:

doc_id	text
1	Smithers, release the hounds.
2	Smithers, unleash the League of Evil!
3	The evil Evil of the most Evil.

The resulting DTM with **normalized** words:



- rows are N_{docs} documents, columns are words, elements are counts
- unique words (terms) of all documents make up vocabulary of size N_{terms}
- column sums: overall occurrences per word; row sums: document length

Bag of words with n-grams

So far, we've used unigrams. Each word ("term") is counted individually.

We can also count **subsequent word combinations (n-grams)**. This counts n subsequent words for each word:

"Smithers, release the hounds."

→ as bigrams (2-grams):

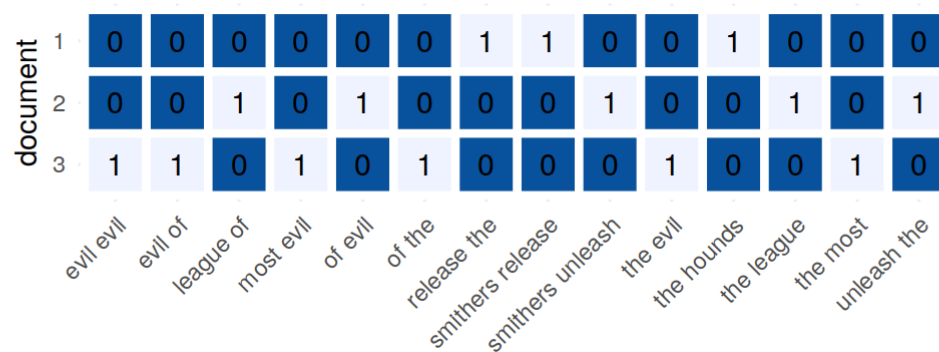
["smithers release", "release the", "the hounds"]

Bag of words with n-grams

Again, our example data:

doc_id	text
1	Smithers, release the hounds.
2	Smithers, unleash the League of Evil!
3	The evil Evil of the most Evil.

Bigrams:



- advantage: captures more "context"
- disadvantage: captures lots of very rare word combinations

Tf-idf weighting

Problem with BoW: common (uninformative) words (e.g. "the, a, and, or, ...") that occur often in many documents overshadow more specific (potentially more interesting) words.

Solutions:

- use **stopword lists** → manual effort
- use a **weighting factor** that decreases the weight of uninformative words / increases the weight of specific words

Tf-idf weighting

Tf-idf (term frequency – inverse document frequency) is such a weighting factor.

For each term t in each document d in a corpus of all documents D , the tfidf weighting factor is calculated as product of two factors:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D)$$

- $\text{tf}(t, d)$: term frequency – measures how often a word t occurs in document d
- $\text{idf}(t, D)$: inverse document frequency – inverse of how common a word t is across all documents D in a corpus

There are different weighting variants for both factors.

Term frequency tf

Common variants:

- absolute word count of term t : $tf(t, d) = N_{d,t}$
- relative word frequency of t in a document d : $tf(t, d) = N_{d,t}/N_d$
 - prevents that documents with many words get higher weights than those with few words

Absolute term counts

document	evil	hounds	league	most	release	smithers	the	unr
1	0	1	0	0	1	1	1	
2	1	0	1	0	0	1	1	
3	3	0	0	1	0	0	2	

relative term frequencies

document	evil	hounds	league	most	release	smithers	the	unleash
1	0	0.25	0	0	0.25	0.25	0.25	0
2	0.2	0	0.2	0	0	0.2	0.2	0.2
3	0.5	0	0	0.17	0	0	0.33	0

Inverse document frequency

idf

Again, many variants. We'll use this one:

$$\text{idf}(t, D) = \log_2(1 + |D| / |d \in D : t \in d|)$$

- t : a term from our vocabulary of corpus D
- $|D|$: the number of documents in corpus D
- $|d \in D : t \in d|$: number of documents d in which t appears
- we assume that each t occurs at least once in D (otherwise a division by zero would be possible)
- we add 1 inside log in order to avoid an idf value of 0

Inverse document frequency

idf

Again, many variants. We'll use this one:

$$\text{idf}(t, D) = \log_2(1 + |D| / |d \in D : t \in d|)$$

	evil	hounds	league	most	release	smithers	the	unleash
document 1	0	1	0	0	1	1	1	0
document 2	1	0	1	0	0	1	1	1
document 3	3	0	0	1	0	0	2	0

Calculate $|d \in D : t \in d|$ (number of doc. d in which t appears) for all terms:

##	evil	hounds	league	most	release	smithers	the	unleash
##	2	1	1	1	1	2	3	1

Inverse document frequency

idf

Again, many variants. We'll use this one:

$$\text{idf}(t, D) = \log_2(1 + |D| / |d \in D : t \in d|)$$

document	evil	hounds	league	most	release	smithers	the	unleash
1	0	1	0	0	1	1	1	0
2	1	0	1	0	0	1	1	1
3	3	0	0	1	0	0	2	0

Plug-in to above formula and you get the idf for all terms:

##	evil	hounds	league	most	release	smithers	the	unleash
##	1.32	2.00	2.00	2.00	2.00	1.32	1.00	2.00

This factor is multiplied to each term frequency

→ the more common the word in the corpus, the lower its idf value

Why is idf logarithmically scaled?

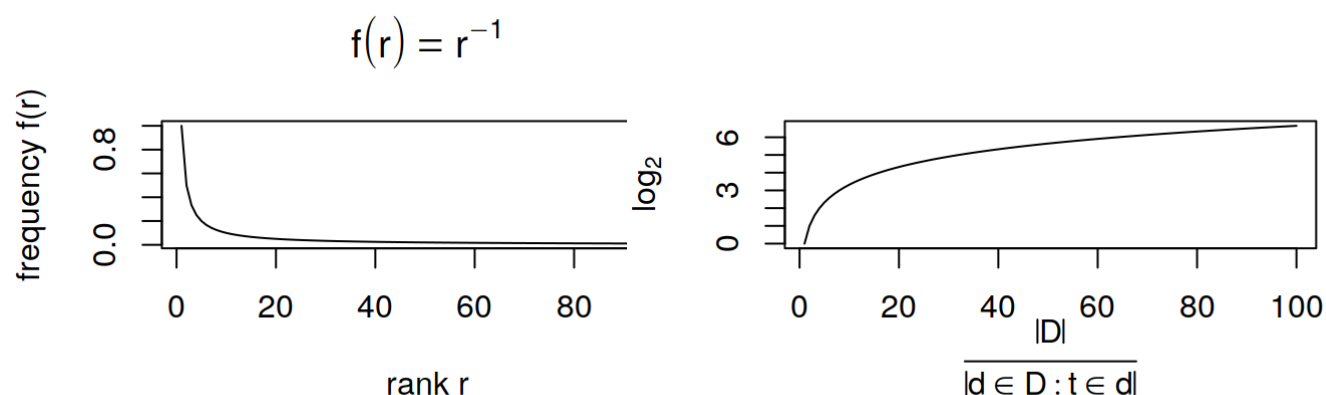
The distribution of words in a natural language text usually follows the "Zipfian distribution", which relates to Zipf's law:

Zipf's law states that the frequency that a word appears is inversely proportional to its rank. – [Silge & Robinson 2017](#)

$$\text{frequency} \propto r^{-1}$$

→ second most frequent word occurs half as often as the most frequent word; third most frequent word occurs a third of the time of the most frequent word, etc.

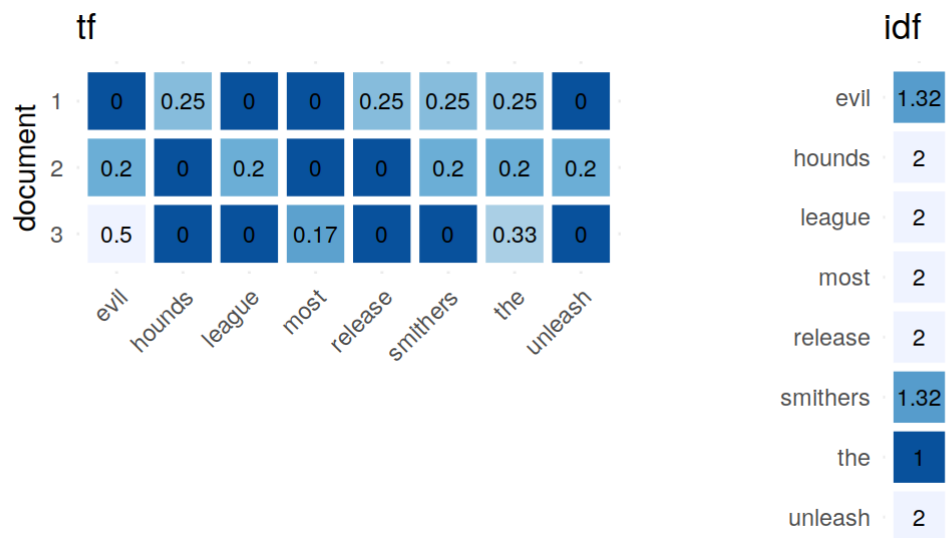
To account for that, we use logarithmical values:



Tf-idf weighting

Back to the initial formula:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D)$$

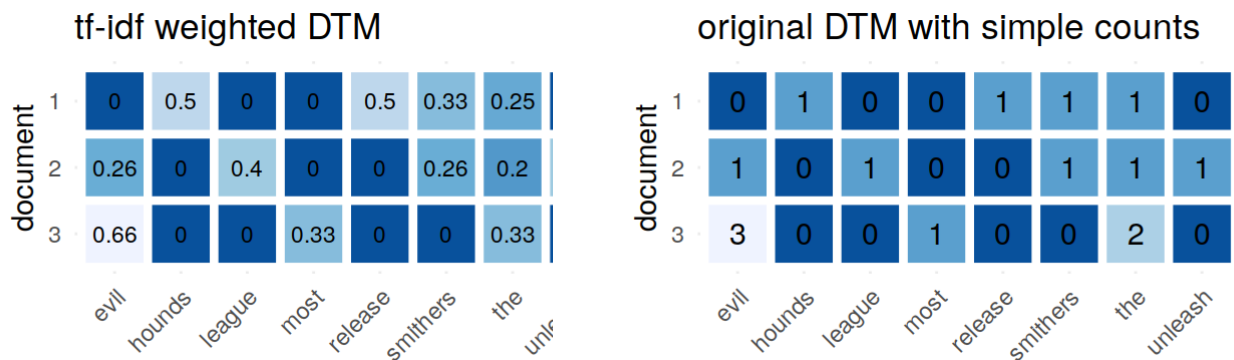


Tf-idf weighting

Back to the initial formula:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \cdot \text{idf}(t, D)$$

Result after matrix multiplication between tf and diagonal of idf:



→ uncommon (i.e. more specific) words get higher weight (e.g. "hounds" or "league")

Feature vectors

Once we have a DTM, we can consider each document as a **vector across terms** (each row in a DTM is a vector of size N_{terms}).

E.g. document #3 has the following term count vector:

```
##           [,1]
## evil         3
## hounds       0
## league       0
## most         1
## release      0
## smithers     0
## the          2
## unleash      0
```

In machine learning terminology this is a **feature vector**. We can use these features for example for document classification, **document similarity**, document clustering, etc.

Non-English text

Most packages, tutorials, etc. are designed for English language texts. When you work with other languages, you may need to apply other methods for text preprocessing. For example, working with German texts might require proper lemmatization to bring words from their inflected form to their base form (e.g. "geschlossen" → "schließen").

Practical text mining with the **tm** package

The tm package

- extensive set of tools for text mining in R
- developed since 2008 by Feinerer et al.

Resources to start:

- [package overview on CRAN](#)
- [Introduction to the tm Package](#)

I will demonstrate how to use the package to investigate word frequency and document similarity.

Creating a corpus

A corpus contains the raw text for each document (identified by a document ID).

The base class is **VCorpus** which can be initialized with a data source.

Read plain text files from a directory:

```
corpus <- VCorpus(DirSource('path/to/documents', encoding = 'UTF-8'),  
                  readerControl = list(language = 'de')) # default language
```

- **encoding** specifies the text format → important for special characters (like German umlauts)
- many file formats supported (Word documents, PDF documents, etc.)

Creating a corpus

A data frame can be converted to a corpus, too. It must contain at least the columns `doc_id`, `text`:

```
df_texts
##   doc_id  text                                     date
##   <chr>   <chr>                                     <chr>
## 1 Grüne   "A. EINLEITUNG\nLiebe Bürgerinnen und Bürger,... 2017...
## 2 Linke   "Die Zukunft, für die wir kämpfen: SOZIAL. GE... 2017...
## 3 SPD     "Es ist Zeit für mehr Gerechtigkeit!\n2017 is... 2017...
```

```
corpus <- VCorpus(DataframeSource(df_texts))
```


The English Europarl corpus

We load a sample of the [European Parliament Proceedings Parallel Corpus](#) with English texts. If you want to follow along, download ["08textmining-resource.zip"](#) from the tutorial website.

```
library(tm)

europarl <- VCorpus(DirSource('08textmining-resources/nltk_europarl'))
europarl

## <<VCorpus>>
## Metadata:  corpus specific: 0, document level (indexed): 0
## Content:  documents: 10
```

Inspecting a corpus

`inspect` returns information on corpora and documents:

```
inspect(europarl)
```

```
## <<VCorpus>>
## Metadata:  corpus specific: 0, document level (indexed): 0
## Content:  documents: 10
##
## [[1]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 145780
##
## [[2]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 554441
##
## [[3]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 228141
##
## [[4]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 559
##
## [[5]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 314931
##
## [[6]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 147766
##
## [[7]]
## <<PlainTextDocument>>
## Metadata:  7
## Content:  chars: 170580
```

Inspecting a corpus

Information for the fourth document:

```
inspect(europarl[[4]])
```

```
## <<PlainTextDocument>>
## Metadata: 7
## Content: chars: 559
##
##
## Adoption of the Minutes of the previous sitting Mr President , I simply w
## There was a terrorist attack this morning in Madrid .
## Someone planted a car bomb and one person has died .
## On behalf of my Group , I once again condemn these terrorist acts .
## Thank you , Mrs Fraga Estévez .
## We had heard about this regrettable incident .
## Unfortunately , the terrorist murderers are once again punishing Spanish
## I note your comments with particular keenness , as you may expect , given
## ( The Minutes were approved )
```

Inspecting a corpus

Get the raw text of a document with `content()`:

```
head(content(europarl[[1]]))
```

```
## [1] " "  
## [2] "Resumption of the session I declare resumed the session of the Europ  
## [3] "Although , as you will have seen , the dreaded ' millennium bug ' fa  
## [4] "You have requested a debate on this subject in the course of the nex  
## [5] "In the meantime , I should like to observe a minute ' s silence , as  
## [6] "Please rise , then , for this minute ' s silence ."
```

Text processing

We want to investigate word frequencies in our corpus. To count words, we need to transform raw text into a normalized sequence of tokens.

Why normalize text? Consider these documents:

1. "We can't explain what we don't know."
 2. "We cannot do that. We do not want that."
- instances of "We" and "we" shouldn't be counted separately → transform to lower case
 - instances of contracted and expanded words ("can't" and "cannot") shouldn't be counted separately → expand all contractions

Text processing

Text processing includes many steps and hence many decisions that have **big effect** on your results. Several possibilities will be shown here. If and how to apply them depends heavily on your data and your later analysis.

Can you think of an example, where unconditional lower case transformation is bad?

Text normalization

Normalization might involve some of the following steps:

- replace contractions ("shouldn't" → "should not")
- remove punctuation and special characters
- case conversion (usually to lower case)
- remove stopwords (extremely common words like "the, a, to, ...")
- correct spelling
- stemming / lemmatization

The order is important!

Text normalization with **tm**

Text normalization can be employed with "transformations" in **tm**.

Concept:

```
tm_map(<CORPUS>, content_transformer(<FUNCTION>), <OPTIONAL ARGS>)
```

- **<FUNCTION>** can be any function that takes a character vector, transforms it, and returns the result as character vector
- **<OPTIONAL ARGS>** are fixed arguments passed to **<FUNCTION>**
- **tm** comes with many predefined transformation functions like `removeWords`, `removePunctuation`, `stemDocuments`, ...

Text normalization with **tm**

A transformation pipeline applied to our corpus (only showing the first three documents):

Original documents:

```
##      name                                     text
## 1      1 Resumption of the session I declare resumed the se...
## 2      2 Adoption of the Minutes of the previous sitting Th...
## 3      3 Middle East peace process ( continuation ) The nex...
```

```
europarl <- tm_map(europarl, content_transformer(textclean::replace_contract
  tm_map(content_transformer(tolower)) %>%
  tm_map(removeNumbers) %>%
  tm_map(removeWords, stopwords('en')) %>%
  tm_map(removePunctuation) %>%
  tm_map(stripWhitespace))
```

After text normalization:

```
##      name                                     text
## 1      1 resumption session declare resumed session europea...
## 2      2 adoption minutes previous sitting minutes yesterda...
## 3      3 middle east peace process continuation next item c...
```

Creating a DTM

- `DocumentTermMatrix()` takes a corpus, tokenizes it, generates document term matrix (DTM)
- parameter `control`: adjust the transformation from corpus to DTM
 - here: allow words that are at least 2 characters long
 - by default, words with less than 3 characters would be removed

```
dtm <- DocumentTermMatrix(euoparl,
                           control = list(wordLengths = c(2, Inf)))
inspect(dtm)
```

```
## <<DocumentTermMatrix (documents: 10, terms: 14118)>>
## Non-/sparse entries: 42920/98260
## Sparsity           : 70%
## Maximal term length: 24
## Weighting          : term frequency (tf)
## Sample            :
##
##                Terms
## Docs            also can commission european  mr must parliament
## ep-00-01-17.en   82  46          130          93 128   53          79
## ep-00-01-18.en  306 200          692          477 356  316          258
## ep-00-01-19.en  132 107          104          187 157   99          104
## ep-00-01-21.en    0   0           0           0   1    0           0
## ep-00-02-02.en  188 118          194          298 220  157          191
## ep-00-02-03.en   69  59           36          146  73   68          101
## ep-00-02-14.en   80  63          126          132  86   75           91
## ep-00-02-15.en  312 255          562          449 365  375          216
## ep-00-02-16.en  293 183          260          556 360  179          212
## ep-00-02-17.en  185 142          184          336 307  215          116
##
##                Terms
## Docs            president union will
## ep-00-01-17.en      89    56    94
## ep-00-01-18.en     203   169   575
## ep-00-01-19.en      89   114   284
## ep-00-01-21.en       1     0     0
## ep-00-02-02.en     183   199   297
## ep-00-02-03.en      47    50   113
## ep-00-02-14.en      90    61   123
```

Creating a DTM

- a **tm** DTM is a **sparse matrix** → only values $\neq 0$ are stored → saves a lot of memory
- many values in a DTM are 0 for natural language texts → can you explain why?
- some functions in R can't work with sparse matrices → convert to an ordinary matrix then:

```
as.matrix(dtm)[,1:8] # cast to an ordinary matrix and see first 8 terms
```

```
##              Terms
## Docs          aan abandon abandoned abandoning abandonment abattoirs
## ep-00-01-17.en  0         0         0         0         0         0
## ep-00-01-18.en  0         1         4         0         0         0
## ep-00-01-19.en  0         1         1         1         0         0
## ep-00-01-21.en  0         0         0         0         0         0
## ep-00-02-02.en  0         0         0         0         0         0
## ep-00-02-03.en  0         1         6         0         0         0
## ep-00-02-14.en  0         0         1         0         0         0
## ep-00-02-15.en  1         0         1         0         0         1
## ep-00-02-16.en  0         0         0         0         0         0
## ep-00-02-17.en  0         1         6         0         1         0
##              Terms
## Docs          abb abbalsthom
## ep-00-01-17.en  0         0
## ep-00-01-18.en  3         0
## ep-00-01-19.en  0         0
## ep-00-01-21.en  0         0
## ep-00-02-02.en  0         0
## ep-00-02-03.en  0         0
## ep-00-02-14.en  0         0
## ep-00-02-15.en  0         0
## ep-00-02-16.en  0         0
## ep-00-02-17.en  0         7
```

To be continued in the
next session ...