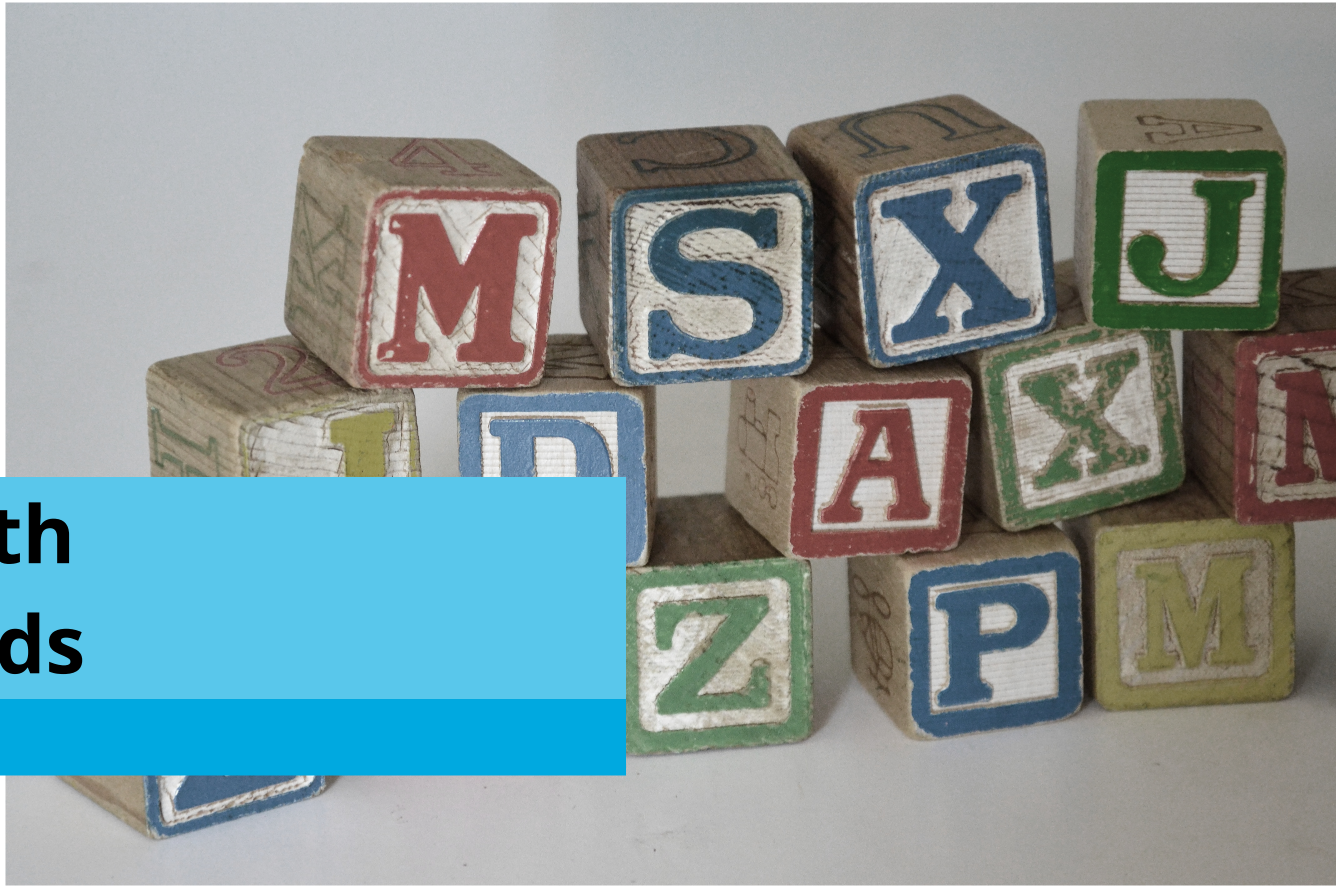
Universität Konstanz



Studying News Use with Computational Methods

Text Preprocessing in R

Julian Unkel
University of Konstanz

2021/06/14

Agenda

After having collected texts, we now need to bring them into a form with which statistical language models are able to work. In practice, this means reducing our text to features (i.e., predictor variables) used in statistical models represented by numbers.

Such **preprocessing** steps includes ways to transform text in numbers, clean, edit and reduce the amount of features, and cast them into the data structures ready for text models.

Our agenda today:

- Text representation in R
  - Terminology
  - The `tidytext` approach
  - The `quanteda` approach
- Basic text preprocessing
  - Tokenization
  - Document-feature matrices
  - Feature reduction: Trimming, stemming, stopword removal
- Advanced text preprocessing & annotation
  - Lemmatization
  - Part-of-speech tagging
  - Named entity recognition
  - Weighting
  - Word embeddings

Terminology

Terminology

In automated content analysis, we are analysing **corpora** of **documents**:

- **Document**: A single item of text as the basic unit of analysis, for example an article, a social media post, or a speech transcript
- **Corpus**: A structured collection of documents

Documents consist of a text string and, optionally, additional meta information:

- **Feature**: Any document property or characteristic used in the models. Think of features as predictors or explanatory/independent variables. For example, counts of individual words, but also any kind of document meta information may be used as a feature.
- **Token**: Any meaningful unit (sub-string) of a text (string). For example, we may *tokenize* a document into words, sentences, letters, etc.
- **n-gram**: Any contiguous sequence of tokens. Thus, 1-grams (*unigrams*) may be single words ("It's, "peanut", "butter", "jelly", "time"), 2-grams (*bigrams*) sequences of two words ("It's peanut" "peanut butter", "butter jelly", "jelly time"), etc.

The `tidytext` approach

The `tidytext` package was created as an Tidyverse extension to apply tidy data principles to textual data and text analysis. Thus, a corpus is represented as a tibble, with documents as rows and document variables as columns.

```
install.packages("tidytext")
library(tidytext)
```

The sample data `guardian_sample_100.rds` on ILIAS contains a sample of 100 each articles published by The Guardian in 2020 in the news and sports section, respectively, including several additional variables (e.g., publication date) already in this format. Load it now:

```
guardian_tibble <- readRDS("data/guardian_sample_100.rds")
```

The tidytext approach

```
guardian_tibble
```

```
## # A tibble: 200 x 6
##        id title           body            url             date                pillar
##     <int> <chr>           <chr>           <chr>           <dttm>              <chr>
##  1      1 Morrison’s ro~ Given the Coa~ https://www.t~ 2020-02-21 19:00:02 News
##  2      2 Truck drives ~ A fuel semi-t~ https://www.t~ 2020-06-01 00:24:15 News
##  3      3 Hong Kong blo~ Hong Kong has~ https://www.t~ 2020-05-19 08:53:48 News
##  4      4 Bernie Sander~ Bernie Sander~ https://www.t~ 2020-02-12 18:15:09 News
##  5      5 After years o~ “It’s hard to~ https://www.t~ 2020-03-30 16:30:28 News
##  6      6 Worrying won'~ The most anno~ https://www.t~ 2020-03-06 10:00:33 News
##  7      7 'Obamagate': ~ On his primet~ https://www.t~ 2020-05-14 17:29:48 News
##  8      8 Climate crisi~ The independe~ https://www.t~ 2020-11-06 19:00:49 News
##  9      9 Deportation o~ Deporting a N~ https://www.t~ 2020-11-24 17:23:40 News
## 10     10 Tom Hunt obit~ In many respe~ https://www.t~ 2020-06-07 09:51:40 News
## # ... with 190 more rows
```

The quanteda approach

The most prominent package for automated content analysis in R is called `quanteda` (*Qu*antitative *An*alysis of *Te*xtual *Da*ta):

```
install.packages("quanteda")
library(quanteda)
```

`quanteda` uses it's own data type called `corpus` to represent text corpora. The `corpus()` function can be used to create corpora from a variety of other data types, including dataframes/tibbles.

If creating a corpus from a dataframe/tibble, use the `docid_field` and `text_field` arguments to select id and text variables, respectively:

```
guardian_corpus <- corpus(guardian_tibble, docid_field = "id", text_field = "body")
```

The quanteda approach

```
guardian_corpus
```

```
## Corpus consisting of 200 documents and 4 docvars.
## 1 :
## "Given the Coalition’s unconscionable track record, it is ver..."
##
## 2 :
## "A fuel semi-truck drove into a George Floyd demonstration of..."
##
## 3 :
## "Hong Kong has in effect banned an annual vigil for the Tiana..."
##
## 4 :
## "Bernie Sanders won the New Hampshire primary on Tuesday nigh..."
##
## 5 :
## "“It’s hard to put into words,” Graeme McCrabb says of seeing..."
##
## 6 :
## "The most annoying question news anchors ask their correspond..."
##
## [ reached max_ndoc ... 194 more documents ]
```

The quanteda approach

Use `docvars()` to access the document variables:

```
as_tibble(docvars(guardian_corpus))
```

```
## # A tibble: 200 x 4
##    title                     url                        date                pillar
##    <chr>                     <chr>                      <dttm>              <chr>
##  1 Morrison’s roadmap to em~ https://www.theguardian~ 2020-02-21 19:00:02 News
##  2 Truck drives through cro~ https://www.theguardian~ 2020-06-01 00:24:15 News
##  3 Hong Kong blocks Tiananm~ https://www.theguardian~ 2020-05-19 08:53:48 News
##  4 Bernie Sanders wins New ~ https://www.theguardian~ 2020-02-12 18:15:09 News
##  5 After years of drought, ~ https://www.theguardian~ 2020-03-30 16:30:28 News
##  6 Worrying won't help: why~ https://www.theguardian~ 2020-03-06 10:00:33 News
##  7 'Obamagate': Fox News fo~ https://www.theguardian~ 2020-05-14 17:29:48 News
##  8 Climate crisis: more tha~ https://www.theguardian~ 2020-11-06 19:00:49 News
##  9 Deportation of man with ~ https://www.theguardian~ 2020-11-24 17:23:40 News
## 10 Tom Hunt obituary         https://www.theguardian~ 2020-06-07 09:51:40 News
## # ... with 190 more rows
```

Converting between tidytext and quanteda

Both packages work fine together. Use `tidytext::tidy()` at any time to convert `quanteda` objects to `tidytext`-style tibbles:

```
tidy(guardian_corpus)
```

```
## # A tibble: 200 x 5
##    text              title             url               date                pillar
##    <chr>             <chr>             <chr>             <dttm>              <chr>
##  1 Given the Coali~ Morrison’s road~ https://www.the~ 2020-02-21 19:00:02 News
##  2 A fuel semi-tru~ Truck drives th~ https://www.the~ 2020-06-01 00:24:15 News
##  3 Hong Kong has i~ Hong Kong block~ https://www.the~ 2020-05-19 08:53:48 News
##  4 Bernie Sanders ~ Bernie Sanders ~ https://www.the~ 2020-02-12 18:15:09 News
##  5 “It’s hard to p~ After years of ~ https://www.the~ 2020-03-30 16:30:28 News
##  6 The most annoyi~ Worrying won't ~ https://www.the~ 2020-03-06 10:00:33 News
##  7 On his primetim~ 'Obamagate': Fo~ https://www.the~ 2020-05-14 17:29:48 News
##  8 The independent~ Climate crisis:~ https://www.the~ 2020-11-06 19:00:49 News
##  9 Deporting a Nig~ Deportation of ~ https://www.the~ 2020-11-24 17:23:40 News
## 10 In many respect~ Tom Hunt obitua~ https://www.the~ 2020-06-07 09:51:40 News
## # ... with 190 more rows
```

Universität Konstanz

Basic text preprocessing

Tokenization

Tokenization describes the process of splitting texts into individual tokens, which can then be used as features in a text analysis model. In quantitative text analysis, we usually want to split into single words, which is thus the default option for most tokenizers.

In `tidytext`, the function is called `unnest_tokens()`. This creates a column with the single tokens and (by default) removes the text input column (in our case, `body`):

```
guardian_tibble_tokenized <- guardian_tibble %>%
  unnest_tokens(word, body)
```

The resulting tibble now has one row per word. Note that by default, this removes punctuation and converts all words to lowercase.

Tokenization

```
guardian_tibble_tokenized
```

```
## # A tibble: 151,589 x 6
##       id title              url                date                pillar word
##    <int> <chr>              <chr>              <dttm>              <chr>  <chr>
##  1     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   given
##  2     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   the
##  3     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   coali~
##  4     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   uncon~
##  5     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   track
##  6     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   record
##  7     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   it
##  8     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   is
##  9     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   very
## 10     1 Morrison's roadma~ https://www.thegu~ 2020-02-21 19:00:02 News   very
## # ... with 151,579 more rows
```

Tokenization

In quanteda, we can use the `tokens()` function on our corpus object:

```
guardian_tokens <- guardian_corpus %>%
  tokens()
guardian_tokens
```

```
## Tokens consisting of 200 documents and 4 docvars.
## 1 :
##  [1] "Given"           "the"              "Coalition's"    "unconscionable"
##  [5] "track"           "record"           ","              "it"
##  [9] "is"              "very"             ","              "very"
## [ ... and 1,348 more ]
##
## 2 :
##  [1] "A"              "fuel"            "semi-truck"    "drove"
##  [5] "into"           "a"               "George"        "Floyd"
##  [9] "demonstration" "of"              "thousands"     "of"
## [ ... and 450 more ]
##
## 3 :
##  [1] "Hong"       "Kong"       "has"        "in"         "effect"     "banned"
##  [7] "an"         "annual"     "vigil"      "for"        "the"        "Tiananmen"
```

Tokenization

Note that `tokens()` by default does neither convert to lowercase nor removes punctuations. To get to the same result as with `tidytext`, we have to be more explicit:

```
guardian_tokens <- guardian_corpus %>%
  tokens(remove_punct = TRUE) %>%
  tokens_tolower()

guardian_tokens
```

```
## Tokens consisting of 200 documents and 4 docvars.
## 1 :
##  [1] "given"          "the"            "coalition's"    "unconscionable"
##  [5] "track"          "record"         "it"             "is"
##  [9] "very"           "very"           "hard"           "to"
## [ ... and 1,203 more ]
##
## 2 :
##  [1] "a"             "fuel"           "semi-truck"    "drove"
##  [5] "into"          "a"              "george"        "floyd"
##  [9] "demonstration" "of"             "thousands"     "of"
## [ ... and 409 more ]
##
```

Tokenization

Tokenization options depend on the project and research interest at hand. Both functions also allow you to use other tokenizers, for example from the `tokenizers` package. For example, when tokenizing tweets you may use special tokenizers that preserve hashtags, mentions, and URLs.

In practice, I find the following steps to be a good default options:

- Converting to lowercase
- Removing punctuation, numbers, symbols, URLs, and separators

```
guardian_tokens <- guardian_corpus %>%
  tokens(remove_punct = TRUE, remove_numbers = TRUE,
         remove_symbols = TRUE, remove_url = TRUE,
         remove_separators = TRUE) %>%
  tokens_tolower()
```

In the follwing, we will continue mainly with `quanteda`, but will return to `tidytext` later on.

Tokenization

We can now also create various n-grams of choice by using `tokens_ngrams()` on our tokens object. For example, for bigrams only:

```
guardian_tokens %>%
  tokens_ngrams(2)
```

```
## Tokens consisting of 200 documents and 4 docvars.
## 1 :
##  [1] "given_the"                     "the_coalition's"
##  [3] "coalition's_unconscionable" "unconscionable_track"
##  [5] "track_record"                  "record_it"
##  [7] "it_is"                         "is_very"
##  [9] "very_very"                     "very_hard"
## [11] "hard_to"                       "to_assume"
## [ ... and 1,201 more ]
##
## 2 :
##  [1] "a_fuel"               "fuel_semi-truck"      "semi-truck_drove"
##  [4] "drove_into"           "into_a"               "a_george"
##  [7] "george_floyd"         "floyd_demonstration" "demonstration_of"
## [10] "of_thousands"         "thousands_of"         "of_people"
## [ ... and 407 more ]
```

Tokenization

Unigrams, bigrams, and trigrams:

```
guardian_tokens %>%
  tokens_ngrams(1:3)
```

```
## Tokens consisting of 200 documents and 4 docvars.
## 1 :
##  [1] "given"          "the"            "coalition's"    "unconscionable"
##  [5] "track"          "record"         "it"             "is"
##  [9] "very"           "very"           "hard"           "to"
## [ ... and 3,627 more ]
##
## 2 :
##  [1] "a"             "fuel"           "semi-truck"     "drove"
##  [5] "into"          "a"              "george"         "floyd"
##  [9] "demonstration" "of"             "thousands"      "of"
## [ ... and 1,245 more ]
##
## 3 :
##  [1] "hong"       "kong"       "has"        "in"         "effect"     "banned"
##  [7] "an"         "annual"     "vigil"      "for"        "the"        "tiananmen"
## [ ... and 1,329 more ]
```

Document-feature matrices

The typical input for most quantitative text analysis methods is called a **DFM** (*D*ocument-*f*eature *m*atrix), with documents in rows and features in columns. Most often, we will use token (i.e., word) counts as features.

We can construct a DFM from our tokens object with `dfm()`:

```
guardian_dfm <- guardian_tokens %>%
  dfm()
```

Document-feature matrices

```
guardian_dfm
```

```
## Document-feature matrix of: 200 documents, 16,028 features (97.76% sparse) and 4 docvars.
##     features
## docs given the coalition's unconscionable track record it is very hard
##    1     4  66           1              1     1      1 20 22    2    1
##    2     0  40           0              0     0      0  4  0    0    0
##    3     0  22           0              0     0      0  3  5    1    0
##    4     0  48           0              0     0      1  5  7    0    0
##    5     0  53           0              0     0      0  8  4    0    1
##    6     0  43           0              0     0      1  8 21    2    1
## [ reached max_ndoc ... 194 more documents, reached max_nfeat ... 16,018 more features ]
```

- In total, our sample contains 16,489 different words (thus: 16,489 features and the same number of columns)
- The DFM's *sparsity* is the proportion of cells with a value of zero.
- Note that our docvars are still accessible via `docvars()`. As such, we could simply column-bind them to the matrix to use (some of) these variables as additional features, which may be particularly useful for (supervised) classification methods. However, in the following sessions, we will mainly work with word features only.

Feature reduction

Even with as little as 200 articles, we have quite a large matrix with already several million cells. This can easily grow to several hundred million cells even with medium-sized corpora. This can both increase the computational load and make it harder to find meaningful, predictive features.

It is thus usually a good idea to reduce the dimensionality of the DFM by removing unnecessary features. The three most common steps are:

- *Trimming* by removing very uncommon and/or very common features because they usually have little to no discriminative or predictive value
- *Stemming* words to their word stem, so for example singular and plural forms of the same word are represented in the same feature
- Removing common functional words (*stopwords*) like conjunctions and articles because, again, they usually hhave little to no discriminative or predictive value

Feature reduction: Trimming

Use `dfm_trim()` to trim features from a DFM. You can trim both by term frequency (how often does the feature appear across all documents) and document frequency (in how many documents does the feature appear), and both by absolute and relative values (among others).

For example, the following code removes all features that appear less than 5 times across all documents and all features that appear in more than 75% of all documents:

```
guardian_dfm %>%
  dfm_trim(min_termfreq = 5, termfreq_type = "count",
           max_docfreq = 0.75, docfreq_type = "prop")
```

```
## Document-feature matrix of: 200 documents, 3,490 features (93.12% sparse) and 4 docvars.
##     features
## docs given track record very hard morrison government approach anything climate
##    1     4     1      1    2    1        4          8        1        1      11
##    2     0     0      0    0    0        0          0        0        0       0
##    3     0     0      0    1    0        0          0        0        0       0
##    4     0     0      1    0    0        0          0        0        0       0
##    5     0     0      0    0    1        0          0        0        0       0
##    6     0     0      1    2    1        0          0        0        0       0
## [ reached max_ndoc ... 194 more documents, reached max_nfeat ... 3,480 more features ]
```

Feature reduction: Stemming

Use `dfm_wordstem()` to conduct word stemming. This uses the Snowball stemming algorithm, which is currently available for 26 languages. Use the `language` argument to set the language (default: `"english"`), and `SnowballC::getStemLanguages()` to get a list of available languages.

```
guardian_dfm %>%
  dfm_wordstem(language = "english")
```

```
## Document-feature matrix of: 200 documents, 11,098 features (97.00% sparse) and 4 docvars.
##     features
## docs given the coalit unconscion track record it is veri hard
##    1     4  66      9          1     1      1 23 22    2    1
##    2     0  40      0          0     0      0  5  0    0    0
##    3     0  22      0          0     0      0  4  5    1    0
##    4     0  48      0          0     0      1  8  7    0    0
##    5     0  53      0          0     0      0 11  4    0    1
##    6     0  43      0          0     0      1 12 21    2    1
## [ reached max_ndoc ... 194 more documents, reached max_nfeat ... 11,088 more features ]
```

For example, note `coalit`, which has replaced `coalition's` (and all other forms of `coalition`) and now has count of 9.

Feature reduction: Stopword removal

Use `dfm_remove()` to remove any features given in a character vector from the DFM. Stopword removal is thus pretty much a form of trimming by providing a list of features to remove.

`quanteda` (via the `stopwords` package) provides several lists of stopwords for various languages. For example, the default list of english stopwords, also from the Snowball project, contains 175 stopwords:

```
stopwords("english")
```

```
##   [1] "i"          "me"         "my"         "myself"     "we"
##   [6] "our"        "ours"       "ourselves"  "you"        "your"
##  [11] "yours"      "yourself"   "yourselves" "he"         "him"
##  [16] "his"        "himself"    "she"        "her"        "hers"
##  [21] "herself"    "it"         "its"        "itself"     "they"
##  [26] "them"       "their"      "theirs"     "themselves" "what"
##  [31] "which"      "who"        "whom"       "this"       "that"
##  [36] "these"      "those"      "am"         "is"         "are"
##  [41] "was"        "were"       "be"         "been"       "being"
##  [46] "have"       "has"        "had"        "having"     "do"
##  [51] "does"       "did"        "doing"      "would"      "should"
##  [56] "could"      "ought"      "i'm"        "you're"     "he's"
##  [61] "she's"      "it's"       "we're"      "they're"    "i've"
##  [66] "you've"     "we've"      "they've"    "i'd"        "you'd"
```

Feature reduction: Stopword removal

For example, note that `the` is gone (and that the sparsity of the matrix actually increased) after stopword removal:

```
guardian_dfm %>%
  dfm_remove(stopwords("english"))
```

```
## Document-feature matrix of: 200 documents, 15,862 features (98.16% sparse) and 4 docvars.
##     features
## docs given coalition's unconscionable track record hard assume morrison
##    1     4           1              1     1      1    1      1        4
##    2     0           0              0     0      0    0      0        0
##    3     0           0              0     0      0    0      0        0
##    4     0           0              0     0      1    0      0        0
##    5     0           0              0     0      0    1      0        0
##    6     0           0              0     0      1    1      0        0
##     features
## docs government approach
##    1          8        1
##    2          0        0
##    3          0        0
##    4          0        0
##    5          0        0
```

Feature reduction

Some general tips on feature reduction:

- There is no single best way that applies to all projects.
- It is often advisable to try out different feature reduction steps and check their effects on the model outcome.
- However, a good, robust text model should also be somewhat resistant to small changes in feature reductions.
- Order matters! For example, when using both stemming and trimming, stem first, then trim.
- Stemming and stopword removal can also be applied to token objects before creating the DFM (with `tokens_wordstem()` and `tokens_remove()`, respectively).

Basic text preprocessing

Exercise 1: Preprocessing

`aoc_tweets.csv` (on ILIAS) contains 783 tweets by Alexandria Ocasio-Cortez made in 2021, as obtained by Twitter's Academic API.

- Load the tweets into R and create a quanteda corpus object
- Tokenize, convert to lowercase and remove punctuation, emojis, numbers, and URLs
- Create a DFM, remove english stopwords, the retweet indicator "RT", #hashtags and @mentions (hint: look at the `?dfm_remove()` documentation).
- Bonus points: Check the DFM and the most common features (`topfeatures()`) to identify further problematic features (and propose solutions)

Advanced text preprocessing & annotation

Using spacyr

Universität Konstanz

More sophisticated preprocessing requires pre-trained language models. The Python module spaCy provides several pre-trained models for a variety of languages, capable of such tasks as lemmatization, part-of-speech tagging, and named entity recognition.

We can make use of spaCy's preprocessing pipelines with the R package spacyr, but this still needs spaCy (and the language models) to be installed:

```
pip install spacy
python -m spacy download en_core_web_sm
```

After installing both spaCy and spacyr, we can then initialize the language models as follows:

```
library(spacyr)
spacy_initialize(model = "en_core_web_sm")
```

Lemmatization

Lemmatization, stemming's fancy sibling, groups inflected word forms to their common dictionary form, lemma. This means that, unlike with stemming, also irregular forms of words can be grouped together (i.e., `"is"` and `"are"` are both lemmatized as `"be"`). However, lemmatization thus also needs to identify the word meaning in a sentence, and as such requires full text (and not just a bag of words).

With a pre-trained model, the process itself is quite simple. We use `spacyr`'s `spacy_parse()` function on a text corpus and set `lemma = TRUE`:

```
guardian_lemma <- spacy_parse(guardian_corpus, lemma = TRUE,
                              pos = FALSE, entity = FALSE)
```

Lemmatization

```
guardian_lemma
```

```
## # A tibble: 175,395 x 5
##    doc_id sentence_id token_id token           lemma
##    <chr>        <int>    <int> <chr>           <chr>
##  1 1                1        1 Given           give
##  2 1                1        2 the             the
##  3 1                1        3 Coalition       Coalition
##  4 1                1        4 ’s              ’s
##  5 1                1        5 unconscionable unconscionable
##  6 1                1        6 track           track
##  7 1                1        7 record          record
##  8 1                1        8 ,               ,
##  9 1                1        9 it              it
## 10 1                1       10 is              be
## # ... with 175,385 more rows
```

For example, note that "`is`" in line 10 has been correctly lemmatized to "`be`".

Lemmatization

From here, we can continue with quanteda functions by converting the word list into a token object:

```
as.tokens(guardian_lemma, use_lemma = TRUE)
```

```
## Tokens consisting of 200 documents.
## 1 :
##  [1] "give"           "the"             "Coalition"     "’s"
##  [5] "unconscionable" "track"           "record"        ","
##  [9] "it"             "be"              "very"          ","
## [ ... and 1,379 more ]
##
## 2 :
##  [1] "a"             "fuel"           "semi"          "-"
##  [5] "truck"         "drive"          "into"          "a"
##  [9] "George"        "Floyd"          "demonstration" "of"
## [ ... and 462 more ]
##
## 3 :
##  [1] "Hong"      "Kong"      "have"      "in"        "effect"    "ban"
##  [7] "an"        "annual"    "vigil"     "for"       "the"       "Tiananmen"
## [ ... and 512 more ]
##
```

Part-of-speech tagging

Part-of-speech tagging (POS tagging) identifies the corresponding part of speech for each word, for example:

- `NOUN`: noun
- `VERB`: verb
- `ADJ`: adjective
- `PROPN`: proper noun

etc.

POS tagging is more relevant for more linguistics-focused models, but may still be useful for statistical approaches as well, for example for disambiguation purposes. We can again use `spacy_parse()`, this time setting `pos = TRUE`:

```
guardian_pos <- spacy_parse(guardian_corpus, lemma = FALSE,
                            pos = TRUE, entity = FALSE)
```

Part-of-speech tagging

```
guardian_pos
```

```
## # A tibble: 175,395 x 5
##    doc_id sentence_id token_id token          pos
##    <chr>        <int>    <int> <chr>          <chr>
##  1 1                1        1 Given          VERB
##  2 1                1        2 the            DET
##  3 1                1        3 Coalition      PROPN
##  4 1                1        4 ’s             PART
##  5 1                1        5 unconscionable ADJ
##  6 1                1        6 track          NOUN
##  7 1                1        7 record         NOUN
##  8 1                1        8 ,              PUNCT
##  9 1                1        9 it             PRON
## 10 1                1       10 is             AUX
## # ... with 175,385 more rows
```

Named entity recognition

Finally, we can use `spacyr` to extract named entities in a corpus. Named entities may include:

- `ORG`: organisations
- `PERSON`: persons
- `NORP`: nationalities, religious or political groups
- `GPE`: geo-political entities, i.e., countries, cities, states
- `FAC`: facilities like buildings, airports, etc.

and several more.

Again, we use `spacy_parse()`, this time setting `entity = TRUE`:

```
guardian_ner <- spacy_parse(guardian_corpus, lemma = FALSE,
                            pos = FALSE, entity = TRUE)
```

Named entity recognition

```
guardian_ner %>%
  filter(entity != "")
```

```
## # A tibble: 22,377 x 5
##    doc_id sentence_id token_id token     entity
##    <chr>        <int>    <int> <chr>     <chr>
##  1 1                1        2 the       ORG_B
##  2 1                1        3 Coalition ORG_I
##  3 1                1        4 ’s        ORG_I
##  4 1                1       18 Morrison  PERSON_B
##  5 1                3        9 Scott     PERSON_B
##  6 1                3       10 Morrison  PERSON_I
##  7 1                4        6 Weetbix   ORG_B
##  8 1                4       15 Morrison  PERSON_B
##  9 1                4       21 the       DATE_B
## 10 1                4       22 coming    DATE_I
## # ... with 22,367 more rows
```

Note that `_B` identifies the beginning of an entity, `_I` an "inside" part of an entity.

Using spacyr

Running spaCy language models consumes lots of memory, so remember to shut down those background processes after all annotation has been completed:

```
spacy_finalize()
```

Weighting

Universität Konstanz

Going back to DFM's and `quanteda`, we may improve model performance by weighting the features in the DFM. The general form to weight DFMs in `quanteda` is to use the function `dfm_weight()`. For example, to use the feature proportion per document instead of absolute frequencies, set `scheme = "prop"`:

```
guardian_dfm %>%
  dfm_weight(scheme = "prop")
```

```
## Document-feature matrix of: 200 documents, 16,028 features (97.76% sparse) and 4 docvars.
##     features
## docs       given        the  coalition's unconscionable        track
##    1 0.003294893 0.05436573 0.0008237232   0.0008237232 0.0008237232
##    2 0           0.09523810 0              0              0
##    3 0           0.04910714 0              0              0
##    4 0           0.05904059 0              0              0
##    5 0           0.09363958 0              0              0
##    6 0           0.04574468 0              0              0
##     features
## docs       record          it          is        very         hard
##    1 0.0008237232 0.016474465 0.018121911 0.001647446 0.0008237232
##    2 0            0.009523810 0           0           0
##    3 0            0.006696429 0.011160714 0.002232143 0
##    4 0.0012300123 0.006150062 0.008610086 0           0
```

Weighting

One common form of weighting is to use the **term frequency - inverse document frequency** (*tf-idf*) statistic. The tf-idf value increases with the number of times a word appears in a document, and is offset with the number of times the respective word appears in documents accross the whole corpus. As such, it seeks to reflect how important/distinctive a word is for a specific document.

In `quanteda`, use `dfm_tfidf()` to apply the tf-idf weighting to a DFM object. For example, note that `"the"` has a value of 0 even though it is the most frequent word in the first document:

```
guardian_dfm %>%
  dfm_tfidf()
```

```
## Document-feature matrix of: 200 documents, 16,028 features (97.76% sparse) and 4 docvars.
##     features
## docs    given the coalition's unconscionable    track    record        it
##    1 2.150408   0           2        2.30103 1.154902 0.8096683 0.7242435
##    2 0          0           0        0       0        0         0.1448487
##    3 0          0           0        0       0        0         0.1086365
##    4 0          0           0        0       0        0.8096683 0.1810609
##    5 0          0           0        0       0        0         0.2896974
##    6 0          0           0        0       0        0.8096683 0.2896974
##     features
```

Word embeddings

With DFMs, we follow the bag-of-words model, that is, we choose to ignore word order and all semantical and syntactical relationships between words (and this actually works fine for most purposes). However, alternative ways of representing text as numbers exist that seek to preserve such connections between lexical units.

With **word embeddings** (also *word vectors*), we encode words in high-dimensional numeric vectors so that words that have little distance in vector space are more similar in meaning than words that have larger distances.

There are complex, pre-trained word embedding models ready to use, such as `word2vec`, but for demonstration purposes, we will create our own word embeddings.

The following steps are based on chapter 5 of the great book "Supervised machine learning for text analysis in R" by Emil Hvitfeldt and Julia Silge.

Word embeddings

First, note that word embeddings are usually trained on a huge corpus of documents, which also makes them very computationally expensive. For demonstration purposes, let's use a larger sample of the Guardian corpus, containing 10,000 articles:

```
guardian_larger <- readRDS("temp/guardian_sample_2020.rds")
guardian_larger
```

```
## # A tibble: 10,000 x 6
##       id title            body             url              date                pillar
##    <int> <chr>            <chr>            <chr>            <dttm>              <chr>
##  1     1 We know this ~ There is a me~ https://www.t~ 2020-01-01 00:09:23 Opini~
##  2     2 Mariah Carey'~ Mariah Carey’~ https://www.t~ 2020-01-01 00:34:18 Arts
##  3     3 Australia wea~ Firefighters ~ https://www.t~ 2020-01-01 02:59:09 News
##  4     4 TV tonight: S~ Dracula 9pm, ~ https://www.t~ 2020-01-01 06:20:56 Arts
##  5     5 Shipping fuel~ Sulphur will ~ https://www.t~ 2020-01-01 07:00:58 News
##  6     6 Western Balka~ The European ~ https://www.t~ 2020-01-01 08:00:01 News
##  7     7 Welcome to th~ Australians f~ https://www.t~ 2020-01-01 08:50:00 News
##  8     8 The Power of ~ Without wishi~ https://www.t~ 2020-01-01 09:01:00 Arts
##  9     9 Top 10 books ~ The other nig~ https://www.t~ 2020-01-01 10:00:02 Arts
## 10    10 Three cities,~ There is more~ https://www.t~ 2020-01-01 10:57:37 Sport
## # ... with 9,990 more rows
```

Word embeddings

These 10,000 articles consist of 7,824,081 words in total, and 117,188 unique words:

```
guardian_larger_tokens <- guardian_larger %>%
    unnest_tokens(word, body)

guardian_larger_tokens %>%
    count(word, sort = TRUE)
```

```
## # A tibble: 117,188 x 2
##    word        n
##    <chr>   <int>
##  1 the    454164
##  2 to     226355
##  3 of     205956
##  4 and    197623
##  5 a      188465
##  6 in     159147
##  7 that    86125
##  8 is      77972
##  9 for     75943
## 10 on      67063
## # ... with 117,178 more rows
```

Word embeddings

First, we nest all documents into their own tibbles, as we want to compute word associations based on their co-occurence within the same document. For example, the row of the first document now contains a nested tibble containing all 698 words as single rows of said document:

```
guardian_nested <- guardian_larger_tokens %>%
    select(id, word) %>%
    nest(words = word)
guardian_nested
```

```
## # A tibble: 9,965 x 2
##        id words
##     <int> <list>
##  1      1 <tibble [698 x 1]>
##  2      2 <tibble [254 x 1]>
##  3      3 <tibble [462 x 1]>
##  4      4 <tibble [515 x 1]>
##  5      5 <tibble [933 x 1]>
##  6      6 <tibble [929 x 1]>
##  7      7 <tibble [437 x 1]>
##  8      8 <tibble [1,215 x 1]>
##  9      9 <tibble [1,204 x 1]>
## 10     10 <tibble [784 x 1]>
```

Word embeddings

One way to measure word associations is by looking at how often they appear together and how often they appear alone in predefined windows (i.e., sequences of n words). The *pointwise mutual information* (*PMI*) measures this association computing the logarithmn of the probability of finding two words together in a given window, and dividing this by the probability of finding each word alone. The higher the PMI, the more likely those two words appear together.

We need some additional packages to efficiently compute this measures for all two-word associations in a given window in our corpus:

```
library(slider) # Creating sliding windows
library(widyr)  # Wide-matrix processing
library(furrr)  # Parallel computing
```

```
## Loading required package: future
```

Word embeddings

The aforementioned book provides a function to create word windows of a given size:

```
slide_windows <- function(tbl, window_size) {
    skipgrams <- slider::slide(
        tbl,
        ~.x,
        .after = window_size - 1,
        .step = 1,
        .complete = TRUE
    )

    safe_mutate <- safely(mutate)

    out <- map2(skipgrams,
                1:length(skipgrams),
                ~ safe_mutate(.x, window_id = .y))

    out %>%
        transpose() %>%
        pluck("result") %>%
        compact() %>%
        bind_rows()
}
```

Word embeddings

Let's create word windows of size 8. This is the computationally expensive part of the procedure and can easily run for several hours, depending on the size of the corpus, even on fairly powerful hardware.

Note that the window size is crucial here: a small window captures only close associations and thus focuses on functionally similar words, whereas larger windows capture more thematic information. However, the larger the window, the higher the computational load.

```
plan(multisession)  ## for parallel processing

guardian_windows <- guardian_nested %>%
    mutate(words = future_map(words, slide_windows, 8L))
```

Word embeddings

The result now contains *all* windows of 8 sequential words for each document. For example, the first document contains 698 words, and now 5,528 windows of 8 words:

```
guardian_windows
```

```
## # A tibble: 9,965 x 2
##        id words
##     <int> <list>
##  1      1 <tibble [5,528 x 2]>
##  2      2 <tibble [1,976 x 2]>
##  3      3 <tibble [3,640 x 2]>
##  4      4 <tibble [4,064 x 2]>
##  5      5 <tibble [7,408 x 2]>
##  6      6 <tibble [7,376 x 2]>
##  7      7 <tibble [3,440 x 2]>
##  8      8 <tibble [9,664 x 2]>
##  9      9 <tibble [9,576 x 2]>
## 10     10 <tibble [6,216 x 2]>
## # ... with 9,955 more rows
```

Word embeddings

Some more data transformations - we unnest our nested tibbles and unite the document and window id variables into one variable:

```
guardian_windows_united <- guardian_windows %>%
    unnest(words) %>%
    unite(window_id, id, window_id)
```

Word embeddings

Window 1_1 identifies the first 8 word window of the first document, window 1_2 the second 8 word window of the second document (note that this second window begins with the second word of the first window), etc.:

```
guardian_windows_united
```

```
## # A tibble: 62,034,608 x 2
##    window_id word
##    <chr>     <chr>
##  1 1_1       there
##  2 1_1       is
##  3 1_1       a
##  4 1_1       message
##  5 1_1       woven
##  6 1_1       into
##  7 1_1       everything
##  8 1_1       the
##  9 1_2       is
## 10 1_2       a
## # ... with 62,034,598 more rows
```

Word embeddings

We can now compute the *PMI* for each two-word association in a given window using `pairwise_pmi()`:

```
guardian_pmi <- guardian_windows_united %>%
    pairwise_pmi(item = word, feature = window_id)
```

Word embeddings

For each word, we now have the PMI of said word appearing with every other word in a 8-word window across the whole corpus. This results in 23,670,248 values for 117,188 unique words:

```
guardian_pmi
```

```
## # A tibble: 23,670,248 x 3
##    item1      item2    pmi
##    <chr>      <chr>  <dbl>
##  1 is         there  1.17
##  2 a          there  0.115
##  3 message    there -0.322
##  4 woven      there -1.25
##  5 into       there -0.849
##  6 everything there -0.125
##  7 the        there -0.588
##  8 prime      there -1.43
##  9 minister   there -1.01
## 10 says       there  0.284
## # ... with 23,670,238 more rows
```

Word embeddings

Universität Konstanz

Now for the actual word embeddings. To reduce the dimensionality of our word matrix (currently 23,670,248 cells), we apply a matrix factorization algorithmn called *singular value decomposition* (*SVD*) that factors our large initial matrix into a set of smaller matrices, the amount of which corresponds to the dimensionality of the vector space. This reduces the size of our initial word matrix to `n_unique_words * n_dimensions`:

```
guardian_word_vectors <- guardian_pmi %>%
    widely_svd(
        item1, item2, pmi,
        nv = 100, maxit = 1000
    )
```

```
guardian_word_vectors <- readRDS("temp/guardian_word_vectors.rds")
```

Word embeddings

Universität Konstanz

The resulting tibble now contains 100 values per word, giving the vector position of each word in each of the 100 dimensions. This way, we have reduced the size of our initial word matrix from 23,670,248 to 11,718,800 (117,188 unique words * 100 dimenions):

```
guardian_word_vectors
```

```
## # A tibble: 11,718,800 x 3
##    item1                 dimension      value
##    <chr>                     <int>      <dbl>
##  1 meyomesse                     1  0.000176
##  2 enoh                          1  0.000183
##  3 andrewwhitey                  1  0.0000581
##  4 www.andrewwhite.nyc           1  0.0000425
##  5 15,040                        1  0.000277
##  6 att                           1  0.000285
##  7 chetnamakan                   1  0.0000833
##  8 leonrestaurants               1  0.0000878
##  9 depaola                       1 -0.0000819
## 10 tomie                         1 -0.000113
## # ... with 11,718,790 more rows
```

Word embeddings

Where to go from here? For example, we can compute the cosine similarity between words to see how close/distant they are from each other (and thus, how closely they are related in our corpus) in the vector space. Luckily, the book also provides a helpful function for this:

```
nearest_neighbors <- function(df, token) {
    df %>%
        widely(
            ~ {
                y <- .[rep(token, nrow(.)), ]
                res <- rowSums(. * y) /
                    (sqrt(rowSums(. ^ 2)) * sqrt(sum(.[token, ] ^ 2)))

                matrix(res, ncol = 1, dimnames = list(x = names(res)))
            },
            sort = TRUE,
            sparse = TRUE
        )(item1, dimension, value) %>%
        select(-item2)
}
```

Word embeddings

Let's take a look at Joe Biden:

```
nearest_neighbors(guardian_word_vectors, "biden")
```

```
## # A tibble: 117,188 x 2
##    item1        value
##    <chr>        <dbl>
##  1 biden        1
##  2 biden's      0.905
##  3 presidential 0.882
##  4 sanders      0.871
##  5 democrats    0.831
##  6 republicans  0.825
##  7 buttigieg    0.791
##  8 joe          0.789
##  9 democratic   0.786
## 10 bernie       0.785
## # ... with 117,178 more rows
```

Word embeddings

Or at Jürgen Klopp:

```
nearest_neighbors(guardian_word_vectors, "klopp")
```

```
## # A tibble: 117,188 x 2
##    item1       value
##    <chr>       <dbl>
##  1 klopp       1
##  2 jürgen      0.917
##  3 mourinho    0.805
##  4 arteta      0.777
##  5 pep         0.767
##  6 lampard     0.764
##  7 liverpool's 0.759
##  8 solskjær    0.755
##  9 gunnar      0.750
## 10 ole         0.732
## # ... with 117,178 more rows
```

Word embeddings

Who is closer to Washington?

```
nearest_neighbors(guardian_word_vectors, "biden") %>%
  filter(item1 == "washington")
```

```
## # A tibble: 1 x 2
##   item1      value
##   <chr>      <dbl>
## 1 washington 0.413
```

```
nearest_neighbors(guardian_word_vectors, "klopp") %>%
  filter(item1 == "washington")
```

```
## # A tibble: 1 x 2
##   item1       value
##   <chr>       <dbl>
## 1 washington 0.0276
```

Universität Konstanz

Exercise solutions

Exercise solutions

Exercise 1: Preprocessing

We can use `read_csv()` (or the base R equivalent `read.csv`) to read in the CSV file:

```
aoc_tweets <- read_csv("data/aoc_tweets.csv",
                       col_types = c(id = col_character()))
```

Note that when using Twitter data (and other social media data), it is advisable to explicitly read numeric IDs as character, as longer numeric IDs may be too long for double precision.

Exercise solutions

We can use the Tweet `id` for the document id; the tweet text is stored in the `text` column:

```
aoc_corpus <- corpus(aoc_tweets, docid_field = "id", text_field = "text")
aoc_corpus
```

```
## Corpus consisting of 783 documents and 5 docvars.
## 1399487557151477764 :
## "RT @BernieSanders: Congratulations to Democrats in Texas for..."
##
## 1399487339043426309 :
## "Proud of you, @naomiosaka. https://t.co/ReCg1K33oA"
##
## 1398666730352922625 :
## "RT @nhannahjones: The only Tulsa commemoration I’m intereste..."
##
## 1398338093002932231 :
## "RT @RepJayapal: Mitch McConnell has already said that 100% o..."
##
## 1398319996179202050 :
## "RT @ninaturner: Big banks know that fossil fuels are causing..."
##
## 1398319597019971587 :
```

Exercise solutions

Hashtags and mentions are preserved by default. Emojis are symbols and are thus removed with `remove_symbols = TRUE`:

```
aoc_tokens <- aoc_corpus %>%
  tokens(remove_punct = TRUE,
         remove_symbols = TRUE,
         remove_numbers = TRUE,
         remove_url = TRUE) %>%
  tokens_tolower()
```

Exercise solutions

Remember that we lowercased all features, thus the retweet indicator "RT" can be removed with "rt":

```
aoc_dfm <- aoc_tokens %>%
  dfm() %>%
  dfm_remove(c("rt", stopwords("english"))) %>%
  dfm_remove(pattern = "@*") %>%
  dfm_remove(pattern = "#*")
aoc_dfm
```

```
## Document-feature matrix of: 783 documents, 4,173 features (99.65% sparse) and 5 docvars.
##                      features
## docs                  congratulations democrats texas protecting democracy
##   1399487557151477764               1         2     1          1         1
##   1399487339043426309               0         0     0          0         0
##   1398666730352922625               0         0     0          0         0
##   1398338093002932231               0         0     0          0         0
##   1398319996179202050               0         0     0          0         0
##   1398319597019971587               0         0     0          0         0
##                      features
## docs                  right vote see u.s proud
##   1399487557151477764     1    1   1   1     0
##   1399487339043426309     0    0   0   0     1
```

Exercise solutions

In the DFM, we can already see `"u.s"`, which is the result from tokenizing `"U.S."`. We may want to change this sequence of characters (`"U.S."`) to `"US"` beforehand, for example with `stringr::str_replace_all()`:

```
aoc_tweets %>%
  mutate(text = str_replace_all(text, "U\\.S\\.", "US"))
```

However, this would also create some ambiguity with `"us"`.

Exercise solutions

Let's look at the top features for further problems:

```
topfeatures(aoc_dfm)
```

```
##      amp   people      can     just        w congress  capitol      now 
##      161       97       89       68       61       58       54       50 
##       us      new 
##       50       50 
```

- `"amp"` is the remains of the HTML entity for the ampersand sign, `&`. This (changing some special characters to their HTML entity) is one annoying quirk of the Twitter API. We may just remove this "by hand" (`dfm_remove("amp")`). More generally, the most reliable option to deal with HTML entities is to decode them in the text beforehand `textutils::HTMLdecode()`.
- `"w"` is the remains of `"w/"` (abbreviation for "with"). We may want to remove this by hand, or, more conveniently, use `dfm_select(min_nchar = 2)` to remove all one-letter words from our DFM.

Thanks

Credits:

- Slides created with `xaringan`
- Title image by Susan Holt Simpson / Unsplash
- Coding cat gif by Memecandy/Giphy