Texts to features

Max Callaghan

2022-09-15

Objectives

Methods

By the end of this session, you will be able to

- Turn a list of texts into document-feature matrix

Methods

By the end of this session, you will be able to

- Turn a list of texts into document-feature matrix
- Understand the choices you make to do this, and their implications

Methods

By the end of this session, you will be able to

- Turn a list of texts into document-feature matrix
- Understand the choices you make to do this, and their implications
- Use document-feature matrices to do things with texts

Definitions

You should also be familiar with the following definitions. We will rely on these throughout the course

- A **document**

Definitions

You should also be familiar with the following definitions. We will rely on these throughout the course

- A **document**
- A **corpus** (plural **corpora**)

Definitions

You should also be familiar with the following definitions. We will rely on these throughout the course

- A **document**
- A **corpus** (plural **corpora**)
- A **token**

Definitions

You should also be familiar with the following definitions. We will rely on these throughout the course

- A **document**
- A **corpus** (plural **corpora**)
- A **token**
- A **term**

Definitions

You should also be familiar with the following definitions. We will rely on these throughout the course

- A **document**
- A **corpus** (plural **corpora**)
- A **token**
- A **term**
- A **vocabulary**

Foundations

Creating a document feature matrix

1. Get some texts. A **corpus** is our collection of texts

Creating a document feature matrix

1. Get some texts. A **corpus** is our collection of texts
2. *split* / *merge* our text(s) into **documents**. A **document** is what we call an individual *arbitrary* unit of text. It could be chapters, paragraphs or sentences of a book; individual tweets, or twitter threads.

Creating a document feature matrix

1. Get some texts. A **corpus** is our collection of texts
2. *split* / *merge* our text(s) into **documents**. A **document** is what we call an individual *arbitrary* unit of text. It could be chapters, paragraphs or sentences of a book; individual tweets, or twitter threads.
3. *split* each **document** into **tokens**

Creating a document feature matrix

1. Get some texts. A **corpus** is our collection of texts
2. *split* / *merge* our text(s) into **documents**. A **document** is what we call an individual *arbitrary* unit of text. It could be chapters, paragraphs or sentences of a book; individual tweets, or twitter threads.
3. *split* each **document** into **tokens**
4. *Remove* unwanted **tokens**.

Creating a document feature matrix

1. Get some texts. A **corpus** is our collection of texts
2. *split* / *merge* our text(s) into **documents**. A **document** is what we call an individual *arbitrary* unit of text. It could be chapters, paragraphs or sentences of a book; individual tweets, or twitter threads.
3. *split* each **document** into **tokens**
4. *Remove* unwanted **tokens**.
5. *Map* tokens to a common form (**lemmatization** or **stemming**)

Creating a document feature matrix

1. Get some texts. A **corpus** is our collection of texts
2. *split* / *merge* our text(s) into **documents**. A **document** is what we call an individual *arbitrary* unit of text. It could be chapters, paragraphs or sentences of a book; individual tweets, or twitter threads.
3. *split* each **document** into **tokens**
4. *Remove* unwanted **tokens**.
5. *Map* tokens to a common form (**lemmatization** or **stemming**)
6. *Count* the occurrences of each **term** (and apply weighting if necessary)

Creating a document feature matrix

1. Get some texts. A **corpus** is our collection of texts
2. *split* / *merge* our text(s) into **documents**. A **document** is what we call an individual *arbitrary* unit of text. It could be chapters, paragraphs or sentences of a book; individual tweets, or twitter threads.
3. *split* each **document** into **tokens**
4. *Remove* unwanted **tokens**.
5. *Map* tokens to a common form (**lemmatization** or **stemming**)
6. *Count* the occurrences of each **term** (and apply weighting if necessary)

These steps are referred to as **preprocessing**. Choices we make here affect the resulting matrix.

What is a document feature matrix?

A **matrix** is a 2 dimensional array with m **rows**, and n **columns**.

In a document feature matrix, each **row** represents a **document**, and each **column** represents a **feature**

What is a document feature matrix?

A **matrix** is a 2 dimensional array with *m* **rows**, and *n* **columns**.

In a document feature matrix, each **row** represents a **document**, and each **column** represents a **feature**

	2022	adaptation	and	change	climate	compatible	current	goals	impacts	long	mitigation	not	of	pathways	state	system	term	the	vulnerability	with
System change not climate change	0	0	0	2	1	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
The Current State of the Climate	0	0	0	0	1	0	1	0	0	0	0	0	1	0	1	0	0	2	0	0
Mitigation pathways compatible with long-term goals	0	0	0	0	0	1	0	1	0	1	1	0	0	1	0	0	1	0	0	1
Climate Change 2022: Impacts, Adaptation and Vulnerability	1	1	1	1	1	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0

This is the **Bag of Words** model. What attributes of the texts are represented?

Feature matrix exercise!

Group exercise!

Form pairs. Each member of the group should come up with a short list of short documents.

Swap document lists, and each make a feature matrix by hand

Practise

Now do this in R

```
library(quanteda)
texts <- c("System change not climate change","The Current State of the Climate")
dfmat <- texts %>%
  tokens() %>%
  dfm()
dfmat
```

```
## Document-feature matrix of: 2 documents, 8 features (43.75% sparse) and 0 docvars.
##        features
## docs    system change not climate the current state of
##   text1      1      2   1       1   0       0     0  0
##   text2      0      0   0       1   2       1     1  1
```

And in Python

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
texts = ["System change not climate change","The Current State of the Climate"]
vectorizer = CountVectorizer()
dfm = vectorizer.fit_transform(texts)
dfm
```

```
## <2x8 sparse matrix of type '<class 'numpy.int64'>'
##  with 9 stored elements in Compressed Sparse Row format>
```

Preprocessing choices

Preprocessing

Now we have seen how to make a document feature matrix with sensible defaults, let's explore *some* of the choices we can make along the way.

All of these choices are about *lowering* the *signal to noise ratio*, preferably without removing too much signal

Splitting/joining documents

A **document** is our single unit of analysis. For different applications, we may want this to be larger or smaller.

Consider the questions:

- Which party's manifesto mentions immigration the most?
- What topics co-occur with immigration in each party's manifesto?

We may exploit given (often hierarchical structures) of documents to do this, and we may at times need to do further joining or splitting ourselves

Check out `quanteda::corpus_reshape()` link and `nltk.sent_tokenizer` link for some help with this.

Tokenizing

We also have some choices about how we create tokens.

This mainly involves how we "clean" texts (check out the arguments of `?tokens` - or write a custom preprocessor to pass to `CountVectorizer`)

In different contexts, we may or may not want to keep punctuation, urls, or numbers

Stopwords

Stopwords are a words that are very common and therefore not that interesting. In *most* cases, we don't care how many times the word "the" appears in a document.

We can add a stopword remover to our pipe

```
library(quanteda)
texts <- c("System change not climate change","The Current State of the Climate")
dfmat <- texts %>%
  tokens() %>%
  tokens_remove(pattern=stopwords("en")) %>%
  dfm()
dfmat
```

```
## Document-feature matrix of: 2 documents, 5 features (40.00% sparse) and 0 docvars.
##        features
## docs    system change climate current state
##   text1      1      2       1       0     0
##   text2      0      0       1       1     1
```

In Python we can pass "english" or a list of stopwords to the `stop_words` parameter of our `CountVectorizer` instance.

Ngrams

By default we use single words (or **unigrams**) as our features. A **unigram** is an **n-gram** where $n = 1$, where an **n-gram** is a continuous sequence of items with length n. We can also have bigrams, trigrams, four-grams, or five-grams, or a combination of these.

```
library(quanteda)
texts <- c("System change not climate change","The Current State of the Climate")
dfmat <- texts %>%
  tokens() %>%
  tokens_ngrams(2) %>%
  dfm()
dfmat
```

```
## Document-feature matrix of: 2 documents, 9 features (50.00% sparse) and 0 docvars.
##        features
## docs    system_change change_not not_climate climate_change the_current
##   text1             1          1           1              1           0
##   text2             0          0           0              0           1
##        features
## docs    current_state state_of of_the the_climate
##   text1             0        0      0           0
##   text2             1        1      1           1
```

In python, we can set the `ngram_range` parameter of our CountVectorizer instance.

Stemming

If we are interested in the *subject* of a document, we may consider multiple forms of the same dictionary word (climate, climatic) as equivalents.

We can reduce all words to a common stem by **stemming**

```
library(quanteda)
texts <- c("System change not climate change","The Current State of the Climate")
dfmat <- texts %>%
  tokens() %>%
  tokens_wordstem() %>%
  dfm()
dfmat
```

```
## Document-feature matrix of: 2 documents, 8 features (43.75% sparse) and 0 docvars.
##        features
## docs    system chang not climat the current state of
##   text1      1     2   1      1   0       0     0  0
##   text2      0     0   0      1   2       1     1  1
```

Lemmatizing

Stemming works by chopping off the end of words. Lemmatization works by reducing words to their dictionary form (e.g. am -> be).

Doing this in R requires the `lexicon` package.

```
library(quanteda)
texts <- c("I am","you are", "she is")
dfmat <- texts %>%
  tokens() %>%
  tokens_replace(pattern = lexicon::hash_lemmas$token, replacement = lexicon::hash_lemmas$lemma) %>%
  dfm()
dfmat
```

```
## Document-feature matrix of: 3 documents, 4 features (50.00% sparse) and 0 docvars.
##        features
## docs    i be you she
##   text1 1  1   0   0
##   text2 0  1   1   0
##   text3 0  1   0   1
```

Stemming and Lemmatization can be achieved in Python by passing a custom tokenizer to your `CountVectorizer` link

TFIDF

In the **dfms** we have made so far, we assume that each feature is equally informative.

However, often the presence of an *uncommon* word will tell us more about a document than the presence of a word that appears in almost every other document.

To reflect this, we often apply a *weight* which penalizes words that appear in many documents in our corpus.

Formally, multiplying the count of each word in each document by the log of the number of documents in the corpus divided by the number of documents containing the word gives us the *term frequency inverse document frequency* of *tf-idf* for short.

TFIDF By hand

EXERCISE: Go back to your handmade document feature matrices. Turn these into tfidf matrices.

$tf(t, d) = f_{t,d}$

(The frequency of term t in document D)

$idf(t, D) = log \frac{N}{|\{d \in D : t \in d\}|}$

(The log of N, the number of documents in the corpus, divided by the length of the set $(|\{\ldots\}|)$ of documents that contain the term.)

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

TFIDF in practice

To use *term frequency inverse document frequency* weighting, we simply add `dfm_tfidf` to our pipe

```
library(quanteda)
texts <- c("I am","you are", "she is")
dfmat <- texts %>%
  tokens() %>%
  tokens_replace(pattern = lexicon::hash_lemmas$token, replacement = lexicon::hash_lemmas$lemma) %>%
  dfm() %>%
  dfm_tfidf()
dfmat
```

```
## Document-feature matrix of: 3 documents, 4 features (50.00% sparse) and 0 docvars.
##        features
## docs            i be       you       she
##   text1 0.4771213  0 0         0
##   text2 0          0 0.4771213 0
##   text3 0          0 0         0.4771213
```

In python, we use a `TfidfVectorizer` in place of a `CountVectorizer`

Using a document-feature matrix

Basic matrix operations

We can inspect the terms with the highest total scores with some basic matrix operations

```
texts <- c("System change not climate change","The Current State of the Climate")
dfmat <- texts %>% tokens() %>% dfm()
sums <- colSums(dfmat)
sums[order(sums, decreasing=TRUE)]
```

```
##   change climate     the   system     not current   state      of
##        2       2       2        1       1       1       1       1
```

```
import numpy as np
texts = ["System change not climate change","The Current State of the Climate"]
vectorizer = CountVectorizer()
dfm = vectorizer.fit_transform(texts)
counts = dfm.sum(axis=0).A1
order = np.argsort(counts)[::-1]
print(vectorizer.get_feature_names_out()[order])
```

```
## ['the' 'climate' 'change' 'system' 'state' 'of' 'not' 'current']
```

```
print(counts[order])
```

```
## [2 2 2 1 1 1 1 1]
```

Classification

The document feature matrix is often the input to other analyses, one of which might be to build a classifier that says whether a text belongs to class or classes of interest.

We can build our own very naive classifier that uses the scores in the **dfm** to predict an outcome. Let's take an toy example that predicts whether a paper title is about NLP.

```
texts <- c(
  "Poverty and inequality implications of carbon pricing",
  "Optimizing and Comparing Topic Models is Simple",
  "How to stop cities and companies causing planetary harm",
  "Contextualized Document Embeddings Improve Topic Coherence",
  "Optimal carbon taxation and horizontal equity"
)
dfmat <- texts %>%
  tokens() %>%
  tokens_wordstem() %>%
  dfm() %>%
  dfm_tfidf()
pred <- (-1 + dfmat[,"document"]*0.5 + dfmat[,"topic"]*3)@x
pred
```

```
## [1] -1.000000  0.193820 -1.000000  0.543305 -1.000000
```

Classification with python

We can do the same in python

```
texts = [
  "Poverty and inequality implications of carbon pricing",
  "Optimizing and Comparing Topic Models is Simple",
  "How to stop cities and companies causing planetary harm",
  "Contextualized Document Embeddings Improve Topic Coherence",
  "Optimal carbon taxation and horizontal equity"
]

vectorizer = TfidfVectorizer()
dfm = vectorizer.fit_transform(texts)
X = dfm.toarray()
vi = vectorizer.vocabulary_
pred = -1 + X[:,vi["document"]]*0.5 + X[:,vi["topic"]]*4

pred
```

```
## array([-1.        ,  0.32098105, -1.        ,  0.56790713, -1.        ])
```

Build your own classifiers!

Get into pairs again. One of you will be spammer, and the other will be a spam filterer.

The spammer starts by writing 5 email subject-like texts (using only standard English words) which are obviously spam or non-spam. The filterer must build a classifier (maximum 3 coefficients) which predicts the spam-ness of the texts.

On each round, the spammer can add 4 texts, and the spam filterer can add 1 coefficient (and edit the others as well as any pre-processing steps).

Keep track of the accuracy of your classifiers!

Outlook

Next week

Next week we'll be looking at a variety of text sources, and exploring how to acquire and use them. We'll cover scraping as well as using APIs.

Have a look at this explanation of how to use APIs in R as well as this recent paper on the relationship between temperatures and hate speech.