A Practical Introduction to Machine Learning in Python

Day 2 - Tuesday

»Preparing for Analysis: From text to features«

Damian Trilling
Anne Kroon

d.c.trilling@uva.nl, @damian0604
a.c.kroon@uva.nl, @annekroon

Gesis

March 10, 2020

Today

Brief recap: Bottom-up and top-down approaches to computer-aided content analysis

	Methodological approach		
	Counting and Dictionary	*Supervised Machine Learning*	*Unsupervised Machine Learning*
Typical research interests and content features	visibility analysis sentiment analysis subjectivity analysis	frames topics gender bias	frames topics
Common statistical procedures	string comparisons counting	support vector machines naive Bayes	principal component analysis cluster analysis latent dirichlet allocation semantic network analysis

deductive → **inductive**

Some considerations

- Both can have a place in your workflow (e.g., bottom-up as first exploratory step)
- You have a clear theoretical expectation? Bottom-up makes little sense.
- But in any case: you need to transform your text into something "countable".

Some considerations

- Both can have a place in your workflow (e.g., bottom-up as first exploratory step)
- You have a clear theoretical expectation? Bottom-up makes little sense.
- But in any case: you need to transform your text into something "countable".

Some considerations

- Both can have a place in your workflow (e.g., bottom-up as first exploratory step)
- You have a clear theoretical expectation? Bottom-up makes little sense.
- But in any case: you need to transform your text into something "countable".

Bottom-up: Exploratory techniques to explore your data

Counting words

1. Split text into words ("tokenization")
2. Count words

Counting words

```
from collections import Counter

texts = ['This is the first text text text first', 'And another text
    yeah yeah']

tokenized_texts = [t.split() for t in texts]

c = Counter(tokenized_texts[0])
print(c.most_common(3)

c2 = Counter(tokenized_texts[1])
print(c2.most_common(3))
```

```
('text', 3), ('first', 2), ('This', 1)]
[('yeah', 2), ('And', 1), ('another', 1)]
```

What do we have to improve?

Counting words

```
from collections import Counter

texts = ['This is the first text text text first', 'And another text
    yeah yeah']

tokenized_texts = [t.split() for t in texts]

c = Counter(tokenized_texts[0])
print(c.most_common(3)

c2 = Counter(tokenized_texts[1])
print(c2.most_common(3))
```

('text', 3), ('first', 2), ('This', 1)]
[('yeah', 2), ('And', 1), ('another', 1)]

What do we have to improve?

Some preprocessing

(more about this later today)

lowercasing

```
texts2 = [t.lower() for t in texts]
```

removing punctuation (method 1)

```
texts3 = [t.replace('.','').replace(',','').replace('!','') for t in
    texts]
```

removing punctuation (method 2)

```
import string
trans = str.maketrans('', '', string.punctuation)
texts4 = [t.translate(trans) for t in texts]
```

Top-down: Regular expression

Regular Expressions: What and why?

What is a regexp?

- a *very* widespread way to describe patterns in strings
- Think of wildcards like $*$ or operators like `OR`, `AND` or `NOT` in search strings: a regexp does the same, but is *much* more powerful
- You can use them in many editors (!), in the Terminal, in STATA . . . and in Python

Regular Expressions: What and why?

What is a regexp?

- a *very* widespread way to describe patterns in strings
- Think of wildcards like `*` or operators like `OR`, `AND` or `NOT` in search strings: a regexp does the same, but is *much* more powerful
- You can use them in many editors (!), in the Terminal, in STATA . . . and in Python

Regular Expressions: What and why?

What is a regexp?

- a *very* widespread way to describe patterns in strings
- Think of wildcards like `*` or operators like `OR`, `AND` or `NOT` in search strings: a regexp does the same, but is *much* more powerful
- You can use them in many editors (!), in the Terminal, in STATA ... and in Python

An example

Removing everything but words

- We wanted to remove everything but words from a tweet
- We did so by calling the `.replace()` method
- We could do this with a regular expression as well: `[^a-zA-Z]` would match anything that is not a letter

Basic regexp elements

Alternatives

`[TtFf]` matches either T or t or F or f

`Twitter|Facebook` matches either Twitter or Facebook

`.` matches any character

Repetition

`?` the expression before occurs 0 or 1 times

`*` the expression before occurs 0 or more times

`+` the expression before occurs 1 or more times

Basic regexp elements

Alternatives

`[TtFf]` matches either T or t or F or f

`Twitter|Facebook` matches either Twitter or Facebook

`.` matches any character

Repetition

`?` the expression before occurs 0 or 1 times

`*` the expression before occurs 0 or more times

`+` the expression before occurs 1 or more times

regexp quizz

Which words would be matched?

1. `[Pp]ython`
2. `[A-Z]+`
3. `RT ?:? @[a-zA-Z0-9]*`

regexp quizz

Which words would be matched?

1. `[Pp]ython`
2. `[A-Z]+`
3. `RT ?:? @[a-zA-Z0-9]*`

regexp quizz

Which words would be matched?

1. `[Pp]ython`
2. `[A-Z]+`
3. `RT ?:? @[a-zA-Z0-9]*`

What else is possible?

If you google `regexp` or `regular expression`, you'll get a bunch of useful overviews. The wikipedia page is not too bad, either.

How to use regular expressions in Python

The module re

`re.findall("[Tt]witter|[Ff]acebook",testo)` returns a list with all occurances of Twitter or Facebook in the string called `testo`

`re.findall("[0-9]+[a-zA-Z]+",testo)` returns a list with all words that start with one or more numbers followed by one or more letters in the string called `testo`

`re.sub("[Tt]witter|[Ff]acebook","a social medium",testo)` returns a string in which all all occurances of Twitter or Facebook are replaced by "a social medium"

How to use regular expressions in Python

The module re

`re.findall("[Tt]witter|[Ff]acebook",testo)` returns a list with all occurances of Twitter or Facebook in the string called `testo`

`re.findall("[0-9]+[a-zA-Z]+",testo)` returns a list with all words that start with one or more numbers followed by one or more letters in the string called `testo`

`re.sub("[Tt]witter|[Ff]acebook","a social medium",testo)` returns a string in which all all occurances of Twitter or Facebook are replaced by "a social medium"

How to use regular expressions in Python

The module re

`re.match(" +([0-9]+) of ([0-9]+) points",line)` returns `None` unless it *exactly* matches the string `line`. If it does, you can access the part between () with the `.group()` method.

Example:

```
line="            2 of 25 points"
result=re.match(" +([0-9]+) of ([0-9]+) points",line)
if result:
print ("Your points:",result.group(1))
print ("Maximum points:",result.group(2))
```

Your points: 2
Maximum points: 25

Possible applications

Data preprocessing

- Remove unwanted characters, words, . . .
- Identify *meaningful* bits of text: usernames, headlines, where an article starts, . . .
- filter (distinguish relevant from irrelevant cases)

Possible applications

Data analysis: Automated coding

- Actors
- Brands
- links or other markers that follow a regular pattern
- Numbers (!)

Example 1: Counting actors

```
import re, csv
from glob import glob
count1_list=[]
count2_list=[]
filename_list = glob("/home/damian/articles/*.txt")

for fn in filename_list:
  with open(fn) as fi:
    artikel = fi.read()
    artikel = artikel.replace('\n',' ')

    count1 = len(re.findall('Israel.*(minister|politician.*|[Aa]uthorit)',artikel))
    count2 = len(re.findall('[Pp]alest',artikel))

    count1_list.append(count1)
    count2_list.append(count2)

output=zip(filename_list,count1_list, count2_list)
with open("results.csv", mode='w',encoding="utf-8") as fo:
  writer = csv.writer(fo)
  writer.writerows(output)
```

Example 2: Which number has this Lexis Nexis article?

```
All Rights Reserved

2 of 200 DOCUMENTS

De Telegraaf

21 maart 2014 vrijdag

Brussel bereikt akkoord aanpak probleembanken;
ECB krijgt meer in melk te brokkelen

SECTION: Finance; Blz. 24
LENGTH: 660 woorden

BRUSSEL  Europa heeft gisteren op de valreep een akkoord bereikt
over een saneringsfonds voor banken. Daarmee staat de laatste
```

Example 2: Check the number of a lexis nexis article

```
All Rights Reserved

2 of 200 DOCUMENTS

De Telegraaf

21 maart 2014 vrijdag

Brussel bereikt akkoord aanpak probleembanken;
ECB krijgt meer in melk te brokkelen

SECTION: Finance; Blz. 24
LENGTH: 660 woorden

BRUSSEL  Europa heeft gisteren op de valreep een akkoord bereikt
over een saneringsfonds voor banken. Daarmee staat de laatste
```

```
for line in tekst:
matchObj=re.match(r" +([0-9]+) of ([0-9]+) DOCUMENTS",line)
if matchObj:
numberofarticle= int(matchObj.group(1))
totalnumberofarticles= int(matchObj.group(2))
```

Practice yourself!

`http://www.pyregex.com/`

Natural Language Processing

NLP: What and why?

What can we do?

- remove stopwords
- stemming
- parse sentences (advanced)

NLP: What and why?

What can we do?

- remove stopwords
- stemming
- parse sentences (advanced)

NLP: What and why?

What can we do?

- remove stopwords
- stemming
- parse sentences (advanced)

Natural Language Processing:
Stopword removal

The logic of the algorithm is very much related to the one of a simple sentiment analysis!

Natural Language Processing:
Stopword removal

The logic of the algorithm is very much related to the one of a simple sentiment analysis!

Stopword removal: What and why?

Why remove stopwords?

- If we want to identify key terms (e.g., by means of a word count), we are not interested in them
- If we want to calculate document similarity, it might be inflated
- If we want to make a word co-occurance graph, irrelevant information will dominate the picture

Stopword removal: How

```
testo='He gives her a beer and a cigarette.'
testonuovo=""
mystopwords=['and','the','a','or','he','she','him','her']
for verbo in testo.split():
  if verbo not in mystopwords:
    testonuovo=testonuovo+verbo+" "
```

What do we get if we do:

```
print (testonuovo)
```

Can you explain the algorithm?

We get:

```
>>> print (testonuovo)
'He gives beer cigarette. '
```

Why is "He" still in there?
How can we fix this?

Stopword removal

```
testo='He gives her a beer and a cigarette.'
testonuovo=""
mystopwords=['and','the','a','or','he','she','him','her']
for verbo in testo.split():
  if verbo.lower() not in mystopwords:
    testonuovo=testonuovo+verbo+" "
```

With *list comprehension* and the `.join()` method, you can achieve the same thing in one line:

```
tn2 = " ".join([w for w in testo.split() if w not in mystopwords])
```

This is more efficient and more "pythonic", but may be more difficult to debug (especially if it gets more complicated)

Stopword removal

```
testo=’He gives her a beer and a cigarette.’
testonuovo=""
mystopwords=[’and’,’the’,’a’,’or’,’he’,’she’,’him’,’her’]
for verbo in testo.split():
  if verbo.lower() not in mystopwords:
    testonuovo=testonuovo+verbo+" "
```

With *list comprehension* and the `.join()` method, you can achieve the same thing in one line:

```
tn2 = " ".join([w for w in testo.split() if w not in mystopwords])
```

This is more efficient and more “pythonic”, but may be more difficult to debug (especially if it gets more complicated)

Stopword removal

```
testo='He gives her a beer and a cigarette.'
testonuovo=""
mystopwords=['and','the','a','or','he','she','him','her']
for verbo in testo.split():
  if verbo.lower() not in mystopwords:
    testonuovo=testonuovo+verbo+" "
```

With *list comprehension* and the `.join()` method, you can achieve the same thing in one line:

```
tn2 = " ".join([w for w in testo.split() if w not in mystopwords])
```

This is more efficient and more “pythonic”, but may be more difficult to debug (especially if it gets more complicated)

When, why, and how do we pre-process?

Natural Language Processing with NLTK and spacy

NLP: What and why?

Why do stemming?

- Because we do not want to distinguish between smoke, smoked, smoking, ...
- Typical preprocessing step (like stopword removal)

Stemming

(with NLTK, see Bird, S., Loper, E., & Klein, E. (2009). *Natural language processing with Python*. Sebastopol, CA: O'Reilly.)

```
from nltk.stem.snowball import SnowballStemmer
stemmer=SnowballStemmer("english")
frase="I am running while generously greeting my neighbors"
frasenuevo=""
for palabra in frase.split():
  frasenuevo=frasenuevo + stemmer.stem(palabra) + " "
```

If we now did `print(frasenuevo)`, it would return:

```
i am run while generous greet my neighbor
```

Stemming and stopword removal - let's combine them!

```
from nltk.stem.snowball import SnowballStemmer
from nltk.corpus import stopwords
stemmer=SnowballStemmer("english")
mystopwords = stopwords.words("english")
frase="I am running while generously greeting my neighbors"
frasenuevo=""
for palabra in frase.lower().split():
  if palabra not in mystopwords:
    frasenuevo=frasenuevo + stemmer.stem(palabra) + " "
```

Now, `print(frasenuevo)` returns:

```
run generous greet neighbor
```

Perfect!

Or:

```
print(" ".join([stemmer.stem(p) for p in frase.lower().split() if p not
     in mystopwords]))
```

In order to use `nltk.corpus.stopwords`, you have to download that module once. You can do so by typing the following in the Python console and selecting the appropriate package from the menu that pops up:

```
import nltk
nltk.download()
```

NB: Don't download everything, that's several GB.

Stemming and stopword removal - let's combine them!

```
from nltk.stem.snowball import SnowballStemmer
from nltk.corpus import stopwords
stemmer=SnowballStemmer("english")
mystopwords = stopwords.words("english")
frase="I am running while generously greeting my neighbors"
frasenuevo=""
for palabra in frase.lower().split():
  if palabra not in mystopwords:
    frasenuevo=frasenuevo + stemmer.stem(palabra) + " "
```

Now, `print(frasenuevo)` returns:

```
run generous greet neighbor
```

Perfect!
Or:

```
print(" ".join([stemmer.stem(p) for p in frase.lower().split() if p not
     in mystopwords]))
```

In order to use `nltk.corpus.stopwords`, you have to download that module once. You can do so by typing the following in the Python console and selecting the appropriate package from the menu that pops up:
`import nltk`
`nltk.download()`
NB: Don't download everything, that's several GB.

Stemming and stopword removal - let's combine them!

```
from nltk.stem.snowball import SnowballStemmer
from nltk.corpus import stopwords
stemmer=SnowballStemmer("english")
mystopwords = stopwords.words("english")
frase="I am running while generously greeting my neighbors"
frasenuevo=""
for palabra in frase.lower().split():
  if palabra not in mystopwords:
    frasenuevo=frasenuevo + stemmer.stem(palabra) + " "
```

Now, `print(frasenuevo)` returns:

```
run generous greet neighbor
```

Perfect!

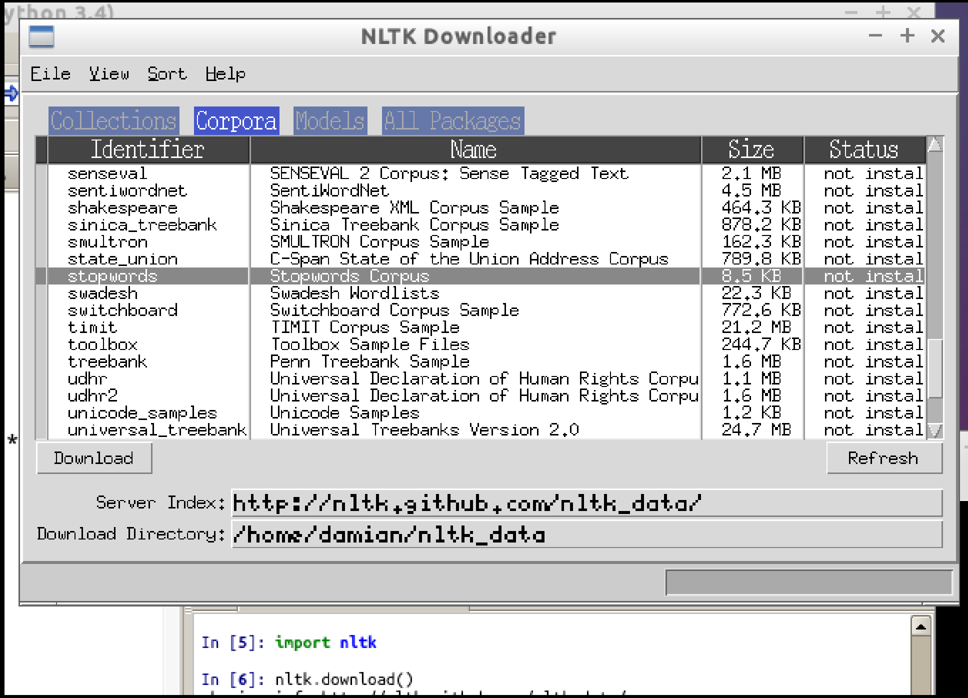
Or:

```
print(" ".join([stemmer.stem(p) for p in frase.lower().split() if p not
     in mystopwords]))
```

In order to use `nltk.corpus.stopwords`, you have to download that module once. You can do so by typing the following in the Python console and selecting the appropriate package from the menu that pops up:

```
import nltk
nltk.download()
```

NB: Don't download everything, that's several GB.


NLTK Downloader
File View Sort Help
Collections Corpora Models All Packages
Identifier Name Size Status
senseval SENSEVAL 2 Corpus: Sense Tagged Text 2.1 MB not instal
sentiwordnet SentiWordNet 4.5 MB not instal
shakespeare Shakespeare XML Corpus Sample 464.3 KB not instal
sinica_treebank Sinica Treebank Corpus Sample 878.2 KB not instal
smultron SMULTRON Corpus Sample 162.3 KB not instal
state_union C-Span State of the Union Address Corpus 789.8 KB not instal
stopwords Stopwords Corpus 8.5 KB not instal
swadesh Swadesh Wordlists 22.3 KB not instal
switchboard Switchboard Corpus Sample 772.6 KB not instal
timit TIMIT Corpus Sample 21.2 MB not instal
toolbox Toolbox Sample Files 244.7 KB not instal
treebank Penn Treebank Sample 1.6 MB not instal
udhr Universal Declaration of Human Rights Corpu 1.1 MB not instal
udhr2 Universal Declaration of Human Rights Corpu 1.6 MB not instal
unicode_samples Unicode Samples 1.2 KB not instal
universal_treebank Universal Treebanks Version 2.0 24.7 MB not instal
Download
Refresh
Server Index: http://nltk.github.com/nltk_data/
Download Directory: /home/damian/nltk_data
In [5]: import nltk
In [6]: nltk.download()

Instead of just looking at single words (unigrams), we can also use adjacent words (bigrams).

ngrams

```
import nltk
texts = ['This is the first text text text first', 'And another text yeah yeah']
texts_bigrams = [["_".join(tup) for tup in nltk.ngrams(t.split(),2)] for t in texts]
print(texts_bigrams)
```

```
[['This_is', 'is_the', 'the_first', 'first_text', 'text_text', 'text_text', 'text_first'], ['And_another', 'another_text', 'text_yeah', 'yeah_yeah']]
```

Typically, we would combine both. What do you think? Why is this useful? (and what may be drawbacks?)

ngrams

```
import nltk
texts = ['This is the first text text text first', 'And another text
    yeah yeah']
texts_bigrams = [["_".join(tup) for tup in nltk.ngrams(t.split(),2)] for
     t in texts]
print(texts_bigrams)
```

```
[['This_is', 'is_the', 'the_first', 'first_text',
'text_text', 'text_text', 'text_first'],
['And_another', 'another_text', 'text_yeah',
'yeah_yeah']]
```

Typically, we would combine both. **What do you think? Why is this useful? (and what may be drawbacks?)**

NLP: What and why?

Why parse sentences?

- To find out what grammatical function words have
- and to get closer to the meaning.

Parsing a sentence

```
import nltk
sentence = "At eight o'clock on Thursday morning, Arthur didn't feel
     very good."
tokens = nltk.word_tokenize(sentence)
print (tokens)
```

`nltk.word_tokenize(sentence)` is similar to sentence.split(), but compare handling of punctuation and the `didn't` in the output:

```
['At', 'eight', "o'clock", 'on', 'Thursday', 'morning','Arthur', 'did',
     "n't", 'feel', 'very', 'good', '.']
```

Parsing a sentence

Now, as the next step, you can "tag" the tokenized sentence:

```
tagged = nltk.pos_tag(tokens)
print (tagged[0:6])
```

gives you the following:

```
[('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'),
('Thursday', 'NNP'), ('morning', 'NN')]
```

And you could get the word type of "morning" with tagged[5][1]!

Parsing a sentence

Now, as the next step, you can “tag” the tokenized sentence:

```
tagged = nltk.pos_tag(tokens)
print (tagged[0:6])
```

gives you the following:

```
[('At', 'IN'), ('eight', 'CD'), ("o'clock", 'JJ'), ('on', 'IN'),
('Thursday', 'NNP'), ('morning', 'NN')]
```

And you could get the word type of "morning" with `tagged[5][1]`!

More NLP

Look at `http://nltk.org`

More NLP

Look at `http://spacy.io`

Example: Named Entity Recognition with spacy

Terminal:

```
sudo pip3 install spacy
sudo python3 -m spacy download nl # or en, de, fr ....
```

Python:

```
import spacy
nlp = spacy.load('nl')
doc = nlp('De docent heet Damian, en hij geeft vandaag les. Daarnaast is
      hij een onderzoeker, net zoals Anne. Ze werken allebei op de UvA')
for ent in doc.ents:
print(ent.text,ent.label_)
```

returns:

```
Damian MISC
Anne PER
UvA LOC
```

Example: Lemmatization instead of stemming

In contrast to stemming, lemmatization actually gives you the words in the form in which you would look them up in a good old dictionary.

```
import spacy
nlp = spacy.load('en')
doc = nlp("I am running while generously greeting my neighbors")
lemmatized = " ".join([word.lemma_ for word in doc])
print(lemmatized)
```

returns:

```
-PRON- be run while generously greet -PRON- neighbor
```

More NLP

Look at `http://nlp.stanford.edu`

More NLP

Look at `https://www.clips.uantwerpen.be/pattern`

From text to feature: count vectorizers and tf-idf vectorizers

What is a vectorizer

- Transforms a list of texts into a sparse (!) matrix (of word frequencies)
- Vectorizer needs to be "fitted" to the training data (learn which words (features) exist in the dataset and assign them to columns in the matrix)
- Vectorizer can then be re-used to transform other datasets

What is a vectorizer

- Transforms a list of texts into a sparse (!) matrix (of word frequencies)
- Vectorizer needs to be "fitted" to the training data (learn which words (features) exist in the dataset and assign them to columns in the matrix)
- Vectorizer can then be re-used to transform other datasets

What is a vectorizer

- Transforms a list of texts into a sparse (!) matrix (of word frequencies)
- Vectorizer needs to be “fitted” to the training data (learn which words (features) exist in the dataset and assign them to columns in the matrix)
- Vectorizer can then be re-used to transform other datasets

Different vectorizers

1. CountVectorizer (=simple word counts)
2. TfidfVectorizer (word counts ("term frequency") weighted by number of documents in which the word occurs at all ("inverse document frequency"))

$$tfidf_{t,d} = tf_{t,d} \cdot idf_t$$

There are different ways to weigh the idf score. A common one is taking the logarithm:

$$idf_t = \log \frac{N}{n_t}$$

where N is the total number of documents and n_t is the number of documents containing term t

Different vectorizers

1. CountVectorizer (=simple word counts)
2. TfidfVectorizer (word counts ("term frequency") weighted by number of documents in which the word occurs at all ("inverse document frequency"))

$$tfidf_{t,d} = tf_{t,d} \cdot idf_t$$

There are different ways to weigh the idf score. A common one is taking the logarithm:

$$idf_t = \log \frac{N}{n_t}$$

where N is the total number of documents and n_t is the number of documents containing term t

Different vectorizers

1. CountVectorizer (=simple word counts)
2. TfidfVectorizer (word counts ("term frequency") weighted by number of documents in which the word occurs at all ("inverse document frequency"))

$$tfidf_{t,d} = tf_{t,d} \cdot idf_t$$

There are different ways to weigh the idf score. A common one is taking the logarithm:

$$idf_t = \log \frac{N}{n_t}$$

where N is the total number of documents and n_t is the number of documents containing term t

Different vectorizers

1. CountVectorizer (=simple word counts)
2. TfidfVectorizer (word counts ("term frequency") weighted by number of documents in which the word occurs at all ("inverse document frequency"))

$$tfidf_{t,d} = tf_{t,d} \cdot idf_t$$

There are different ways to weigh the idf score. A common one is taking the logarithm:

$$idf_t = \log \frac{N}{n_t}$$

where N is the total number of documents and n_t is the number of documents containing term t

Different vectorizer options

- Preprocessing (e.g., stopword removal)
- Remove words below a specific threshold ("occurring in less than $n = 5$ documents") $\Rightarrow$ spelling mistakes etc.
- Remove words above a specific threshold ("occuring in more than 50% of all documents) $\Rightarrow$ de-facto stopwords
- Not only to improve prediction, but also performance (can reduce number of features by a huge amount)

Using a scikit-learn vectorizer

```
from sklearn.feature_extraction.text import Couectorizer
texts = ['This is the first text text text first', 'And another text yeah yeah']
vec = CountVectorizer(texts)
vec.fit_transform(texts)

# if we want to see what it looks like
# DON'T DO THIS WITH LARGE MATRICES!
print(vec.get_feature_names())
print(vec.transform(texts).todense())
```

Summing up: From text to feature

Before we can do machine learning, we need to make features

- typically, (weighted) word frequencies (count vs tf·idf)
- normalization steps first (lowercasing, punctuation, (stemming/lemmatizing))
- potentially also other feature (e.g., named entities – or only specific word types)
- unigrams vs ngrams
- pruning (removing extremes)

Before we can do machine learning, we need to make features

- typically, (weighted) word frequencies (count vs tf·idf)
- normalization steps first (lowercasing, punctuation, (stemming/lemmatizing))
- potentially also other feature (e.g., named entities – or only specific word types)
- unigrams vs ngrams
- pruning (removing extremes)

Before we can do machine learning, we need to make features

- typically, (weighted) word frequencies (count vs tf·idf)
- normalization steps first (lowercasing, punctuation, (stemming/lemmatizing))
- potentially also other feature (e.g., named entities – or only specific word types)
- unigrams vs ngrams
- pruning (removing extremes)

Before we can do machine learning, we need to make features

- typically, (weighted) word frequencies (count vs tf·idf)
- normalization steps first (lowercasing, punctuation, (stemming/lemmatizing))
- potentially also other feature (e.g., named entities – or only specific word types)
- unigrams vs ngrams
- pruning (removing extremes)

Before we can do machine learning, we need to make features

- typically, (weighted) word frequencies (count vs tf·idf)
- normalization steps first (lowercasing, punctuation, (stemming/lemmatizing))
- potentially also other feature (e.g., named entities – or only specific word types)
- unigrams vs ngrams
- pruning (removing extremes)