Teach teacher workshop » NLP «

Anne Kroon

a.c.kroon@uva.nl
@annekroon

July 1, 2021

Afdeling Communicatiewetenschap
Universiteit van Amsterdam

Today

- Bag-of-words
  - General idea
- clean BOW
  - Better tokenization
  - Stopword removal
  - Stemming and lemmatization
  - How further?

Bag-of-words

Bag-of-words

General idea

A text as a collections of word

Let us represent a string

```
t = "This this is is is a test test test"
```

like this:

```
from collections import Counter
print(Counter(t.split()))
```

```
Counter({'is': 3, 'test': 3, 'This': 1, 'this': 1, 'a': 1})
```

Compared to the original string, this representation

- is less repetitive
- preserves word frequencies
- but does *not* preserve word order
- can be interpreted as a vector to calculate with (!!!)

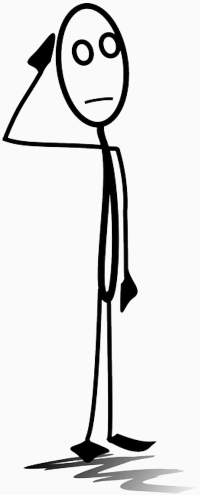
From vector to matrix

If we do this for multiple texts, we can arrange the vectors in a table.

t1 = "This this is is is a test test test"

t2 = "This is an example"

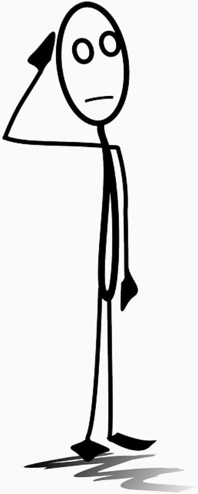
	a	an	example	is	this	This	test
t1	1	0	0	3	1	1	3
t2	0	1	1	1	0	1	0



What can you do with such a matrix?
Why would you want to represent a collection of texts in such a way?

The cell entries: raw counts versus tf·idf scores

- In the example, we entered simple counts (the "term frequency")



But are all terms equally important?

The cell entries: raw counts versus tf·idf scores

- In the example, we entered simple counts (the "term frequency")
- But does a word that occurs in almost all documents contain much information?
- And isn't the presence of a word that occurs in very few documents a pretty strong hint?
- **Solution: Weigh by *the number of documents in which the term occurs at least once) (the "document frequency")***

$\Rightarrow$ we multiply the "term frequency" (tf) by the inverse document frequency (idf)

(usually with some additional logarithmic transformation and normalization applied, see https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html)

tf·idf

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$$tf_{i,j} = \text{ number of occurrences of } i \text{ in } j$$
$$df_i = \text{ number of documents containing } i$$
$$N = \text{total number of documents}$$

Is tf·idf always better?

It depends.

- Ultimately, it's an empirical question which works better ($\rightarrow$ weeks on machine learning)
- In many scenarios, "discounting" too frequent words and "boosting" rare words makes a lot of sense (most frequent words in a text can be highly un-informative)
- Beauty of raw tf counts, though: interpretability + describes document in itself, not in relation to other documents

clean BOW

Room for improvement

tokenization How do we (best) split a sentence into tokens (terms, words)?

pruning How can we remove unneccessary words?

lemmatization How can we make sure that slight variations of the same word are not counted differently?

clean BOW

Better tokenization

OK, good enough, perfect?

.split()

- space → new word
- no further processing whatsoever
- thus, only works well if we do a preprocessing outselves (e.g., remove punctuation)

```
docs = ["This is a text", "I haven’t seen John’s derring-do. Second
    sentence!"]
tokens = [d.split() for d in docs]
```

```
[['This', 'is', 'a', 'text'], ['I', "haven't", 'seen', "John's", 'derring-do.', 'Second', '
    sentence!']]
```

OK, good enough, perfect?

Tokenizers from the NLTK pacakge

- multiple improved tokenizers that can be used instead of .split()
- e.g., Treebank tokenizer:
    - split standard contractions ("don't")
    - deals with punctuation

```
from nltk.tokenize import TreebankWordTokenizer
tokens = [TreebankWordTokenizer().tokenize(d) for d in docs]
```

```
[['This', 'is', 'a', 'text'], ['I', 'have', "n't", 'seen', 'John', "'s", 'derring-do.', 'Second
     ', 'sentence', '!']]
```

Notice the failure to split the `.` at the end of the first sentence in the second doc. That's because TreebankWordTokenizer expects *sentences* as input. See book for a solution.

OK, so we can tokenize with a list comprehension (and that's often a good idea!). But what if we want to *directly* get a DTM instead of lists of tokens?

OK, good enough, perfect?

scikit-learn's CountVectorizer (default settings)

- applies lowercasing
- deals with punctuation etc. itself
- minimum word length > 1
- more technically, tokenizes using this regular expression: `r"(?u)\b\w\w+\b"` [1]

```
from sklearn.feature_extraction.text import CountVectorizer
cv = CountVectorizer()
dtm_sparse = cv.fit_transform(docs)
```

[1] ?u = support unicode, \b = word boundary

OK, good enough, perfect?

CountVectorizer supports more

- stopword removal
- custom regular expression
- or even using an external tokenizer
- ngrams instead of unigrams

see
https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

Best of both worlds

Use the Count vectorizer with a NLTK-based external tokenizer! (see book)

clean BOW

Stopword removal

Stopword removal

What are stopwords?

- Very frequent words with little inherent meaning
- `the, a, he, she, ...`
- context-dependent: if you are interested in gender, `he` and `she` are no stopwords.
- Many existing lists as basis

When using the CountVectorizer, we can simply provide a stopword list.

But we can also remove stopwords "by hand" (next slide):

Stopword removal

```python
1  from nltk.corpus import stopwords
2  mystopwords = stopwords.words("english")
3  mystopwords.extend(["test", "this"])
4
5  def tokenize_clean(s, stoplist):
6      cleantokens = []
7      for w in TreebankWordTokenizer().tokenize(s):
8          if w.lower() not in stoplist:
9              cleantokens.append(w)
10     return cleantokens
11
12 tokens = [tokenize_clean(d, mystopwords) for d in docs]
```

```
[['text'], ["n't", 'seen', 'John', 'derring-do.', 'Second', 'sentence', '!']]
```

You can do more!

For instance, in line 8, you could add an or statement to also exclude punctuation.

CountVectorizer, only stopword removal

```
from sklearn.feature_extraction.text import CountVectorizer,
    TfidfVectorizer
myvectorizer = CountVectorizer(stop_words=mystopwords)
```

CountVectorizer, better tokenization, stopword removal (pay attention that stopword list uses same tokenization!):

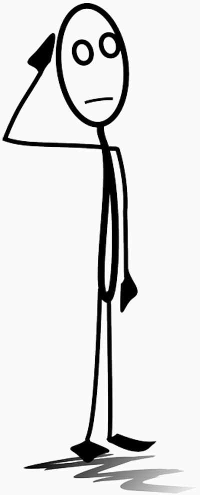
```
myvectorizer = CountVectorizer(tokenizer = TreebankWordTokenizer().
    tokenize, stop_words=mystopwords)
```

Additionally remove words that occur in more than 75% or less than $n = 2$ documents:

```
myvectorizer = CountVectorizer(tokenizer = TreebankWordTokenizer().
    tokenize, stop_words=mystopwords, max_df=.75, min_df=2)
```

All togehter: tf·idf, explicit stopword removal, pruning

```
myvectorizer = TfidfVectorizer(tokenizer = TreebankWordTokenizer().
    tokenize, stop_words=mystopwords, max_df=.75, min_df=2)
```



What is "best"? Which (combination of) techniques to use, and how to decide?

clean BOW

Stemming and lemmatization

Stemming and lemmatization

- Stemming: reduce words to its stem by removing last part (drinking → drink)
- Lemmatization: find word that you would need to look up in a dictionary (drinking → drink, but also went → go)
- stemming is simpler than lemmatization
- lemmatization often better

Example below: tokenization and lemmatization with spacy in one go:

```
import spacy
nlp = spacy.load('en') # potentially you need to install the language
    model first
lemmatized_tokens = [[token.lemma_ for token in nlp(doc)] for doc in
    docs]
```

clean BOW

How further?

Main takeaway

- It matters how you transform your text into numbers ("vectorization").
- Preprocessing matters, be able to make informed choices.
- Keep this in mind when we will discuss Machine Learning.

- Once you vectorized your texts, you can do all kinds of calculations (random example: get the cosine similarity between two texts)

More NLP

***n*-grams** Consider using *n*-grams instead of unigrams

POS-tagging grammatical function (“part-of-speach”) of tokens

NER named entity recognition (persons, organizations, locations)

More NLP

I **really** recommend looking into spacy (https://spacy.io) for advanced natural language processing, such as part-of-speech-tagging and named entity recogntion.