Computational Communication Science 2
Week 2 - Lecture
»Text as Data «

Anne Kroon

a.c.kroon@uva.nl, @annekroon

April, 2022

Digital Society Minor, University of Amsterdam

Today

The toolkit

- Bottom-up vs. top-down
- Approaches to working with text

Natural Language Processing

- Better tokenization
- Stopword and punctuation removal
- ngrams

From text to features: vectorizers

- General idea
- Pruning

From test to large-scale

The toolkit

The toolkit

Bottom-up vs. top-down

Automated content analysis can be either bottom-up (inductive, explorative, pattern recognition, . . .) or top-down (deductive, based on a-priori developed rules, . . .). Or in between.

The ACA toolbox

	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

Boumans and Trilling, 2016

Bottom-up vs. top-down

Bottom-up

- Count most frequently occurring words
- Maybe better: Count combinations of words $\Rightarrow$ Which words co-occur together?

We *don't* specify what to look for in advance

Top-down

- Count frequencies of pre-defined words
- Maybe better: patterns instead of words

We *do* specify what to look for in advance

A simple bottom-up approach

```
from collections import Counter
texts = ["Communication in the Digital Society is a very very complex phenomenon", "I like to study it"]
bottom_up = []
for t in texts:
    bottom_up.append(Counter(t.lower().split()).most_common(3))
print(bottom_up)
```

This results in:

```
[('very', 2), ('Communication', 1), ('in', 1)]
[('I', 1), ('like', 1), ('to', 1)]
```

Please note that you can also write this like:

```
bottom_up = [Counter(t.split()).most_common(3) for t  in texts]
```

- This is *exactly* the same, just shorter (and faster).
- You do *not* have to use list comprehensions, but it helps if you can read them.

A simple top-down approach

```python
texts = ["Communication in the Digital Society is a very very complex phenomenon", "I like to study it"]
features = ["communication", "digital", "study"]
for t in texts:
    print(f"\nAnalyzing '{t}':")
        for f in features:
            print(f"{f} occurs {t.lower().count(f)} times")
```

```
Analyzing 'Communication in the Digital Society is a very very complex phenomenon':
communication occurs 1 times
digital occurs 1 times
study occurs 0 times

Analyzing 'I like to study it':
communication occurs 0 times
digital occurs 0 times
study occurs 1 times
```

. . . save the results as a `list` as follows . . .

```python
top_down = [[t.lower().count(f) for f in features] for t in texts]
```

When would you use which approach?

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".

The toolkit

Approaches to working with text

The toolbox

Slicing

`mystring[2:5]` to get the characters with indices 2,3,4

String methods

- `.lower()` returns lowercased string
- `.strip()` returns string without whitespace at beginning and end
- `.find("bla")` returns index of position of substring “bla” or -1 if not found
- `.replace("a","b")` returns string where `"a"` is replaced by `"b"`
- `.count("bla")` counts how often substring “bla” occurs

Natural Language Processing

Natural Language Processing

NLP: What and why?

Preprocessing steps

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

pruning How can we remove unneccessary words/ punctuation?

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

Natural Language Processing

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.',
```

OK, good enough, perfect?

Tokenizers from the NLTK package

- 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.

Natural Language Processing

Stopword and punctuation removal

Stopword removal

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

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

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

```python
from nltk.corpus import stopwords
mystopwords = stopwords.words("english")
mystopwords.extend(["test", "this"])

tokens_without_stopwords = [[word for word in doc if word.lower() not in mystopwords] for doc in tokens]
```

```
[['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.

Removing punctuation

```
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer(r'\w+')
tokenizer.tokenize("Hi students, what's up!")
```

```
['Hi', 'students', 'what', 's', 'up']
```

Natural Language Processing

ngrams

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

ngrams

```python
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?)**

Main takeaway

- Preprocessing matters, be able to make informed choices.
- Keep this in mind when moving to Machine Learning.

From text to features: vectorizers

From text to features: vectorizers

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:
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 (!!!)

Of course, still a lot of stuff to fine-tune. . . (for example, This/this)

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?

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

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

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

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

Is tf·idf always better?

It depends.

- Ultimately, it's an empirical question which works better ($\rightarrow$ 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

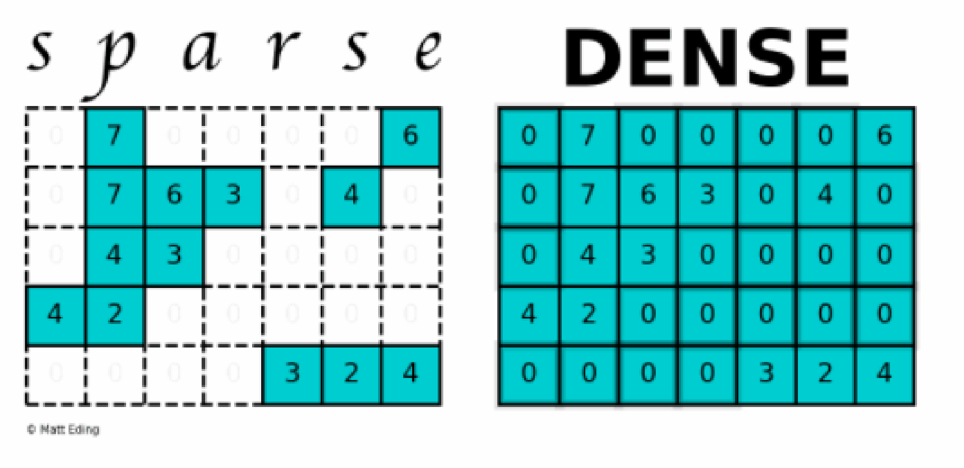
Different vectorizers

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”))

Internal representations

Sparse vs dense matrices

- $\rightarrow$ tens of thousands of columns (terms), and one row per document
- Filling all cells is inefficient *and* can make the matrix too large to fit in memory (!!!)
- Solution: store only non-zero values with their coordinates! (sparse matrix)
- dense matrix (or dataframes) not advisable, only for toy examples



https://matteding.github.io/2019/04/25/sparse-matrices/

We justed learned how to 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)

From text to features: vectorizers

Pruning

General idea

- Idea behind both stopword removal and tf·idf: too frequent words are uninformative
- (possible) downside stopword removal: a priori list, does not take empirical frequencies in dataset into account
- (possible) downside tf·idf: does not reduce number of features

Pruning: remove all features (tokens) that occur in less than X or more than X of the documents

CountVectorizer, only stopword removal

```python
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!):

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

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

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

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

```python
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?

From test to large-scale

General approach

1. Take a single string and test your idea

```
t = "This is a test test test."
print(t.count("test"))
```

2a. You'd assume it to return 3. If so, scale it up:

```
results = []
for t in listwithallmytexts:
    r = t.count("test")
    print(f"{t} contains the substring {r} times")
        results.append(r)
```

2b. If you *only* need to get the list of results, a list comprehension is more elegant:

```
results = [t.count("test") for t in listwithallmytexts]
```

General approach

Test on a single string, then make a for loop or list comprehension!

Own functions

If it gets more complex, you can write your ow= function and then use it in the list comprehension:

```
def mycleanup(t):
    # do sth with string t here, create new string t2
    return t2

results = [mycleanup(t) for t in allmytexts]
```

Pandas string methods as alternative

If you select column with strings from a pandas dataframe, pandas offers a collection of string methods (via `.str.`) that largely mirror standard Python string methods:

```
df['newcoloumnwithresults'] = df['columnwithtext'].str.count("bla")
```

To pandas or not to pandas for text?

Partly a matter of taste.

Not-too-large dataset with a lot of extra columns? Advanced statistical analysis planned? Sounds like pandas.

It's mainly a lot of text? Wanna do some machine learning later on anyway? It's large and (potentially) messy? Doesn't sound like pandas is a good idea.

Thank you!!

Thank you for your attention!

- Questions? Comments?

References

References

Boumans, Jelle W. and Damian Trilling (2016). "Taking stock of the toolkit: An overview of relevant autmated content analysis approaches and techniques for digital journalism scholars." In: *Digital Journalism* 4.1, pp. 8–23. ISSN: 2167-0811. DOI: 10.1080/21670811.2015.1096598.