

Big Data and Automated Content Analysis (12EC)

Week 6: »Processing textual data«

Wednesday

Damian Trilling

d.c.trilling@uva.nl, @damian0604

March 16, 2022

UvA RM Communication Science

Today

Basic string operations

Regular expressions

What is a regexp?

Using a regexp in Python

The bag-of-words (BOW) model

General idea

A cleaner BOW representation

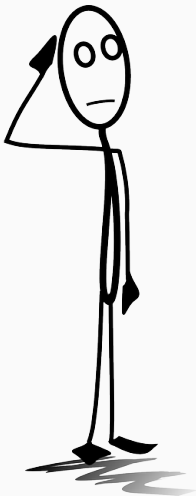
Better tokenization

Stopword removal

Pruning

Stemming and lemmatization

The order of preprocessing steps



Everything clear from last week?

Main points from last week

I assume that by now, everybody knows:

- the relationship between “traditional” statistics and machine learning;
- how to run unsupervised models with scikit-learn;
- how to run supervised models with scikit-learn.

This week, we will get a general overview of working with textual data. Combining the knowledge from this week with last week gives you all blocks you need to do cool automated content analyses – which we will do a lot next block.

Basic string operations

Working with strings

1. string methods that every string has (`"hello".upper()`)
2. functions that take a string as input (`len("hello")`)
3. pandas column string methods
(`df["somecolumn"].str.upper()`)
4. applying string methods or functions to a pandas column
(`df["somecolumn"].apply(len)` or
`df["somecolumn"].apply(lambda x: x.upper())`)

6

Two examples says even more:

```
1 from string import punctuation
2
3 # punctuation is just the string '!"#$%&'()*+,-./:;<=>@[|~`{|}~'
4
5 text = "This is a test! Let's get rid (of) punct&"
6
7 # we make a list of each character in the text but only if it is not
8 # a punctuation sign. The, we join the elements of the list directly
9 # to each other without anything between it ("")
10 cleantext = "".join([c for c in text if c not in punctuation])
```

8


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

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

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


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:

```
1 df['newcolumnwithresults'] = 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.

Regular expressions

Regular expressions

What is a regexp?

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

What else is possible?

See the table in the book!

Regular expressions

Using a regexp in Python

Example 1: Counting actors

```
1 import re, csv
2 from glob import glob
3 counts1=[]
4 counts2=[]
5 filenames = glob("/home/damian/articles/*.txt")
6
7 for fn in filenames:
8     with open(fn) as fi:
9         artikel = fi.read()
10        artikel = artikel.replace('\n', ' ')
11
12        ↪ counts1.append(len(re.findall('Israel.*(minister|politician.*|[Aa]utl
13        counts2.append(len(re.findall('[Pp]alest',artikel)))
14
15 output=zip(filenames, counts1, counts2)
16 with open("results.csv", mode='w',encoding="utf-8") as fo:
17     writer = csv.writer(fo)
18     writer.writerows(output)
```


Example 2: Parsing semi-structured data

If your data look like this, you can loop over the lines and use regular expressions to extract the info you need!

```
1           All Rights Reserved
2
3           2 of 200 DOCUMENTS
4
5           De Telegraaf
6
7           21 maart 2014 vrijdag
8
9 Brussel bereikt akkoord aanpak probleebanken;
10 ECB krijgt meer in melk te brokkelen
11
12 SECTION: Finance; Blz. 24
13 LENGTH: 660 woorden
14
15 BRUSSEL Europa heeft gisteren op de valreep een akkoord bereikt
16 over een saneringsfonds voor banken. Daarmee staat de laatste
```

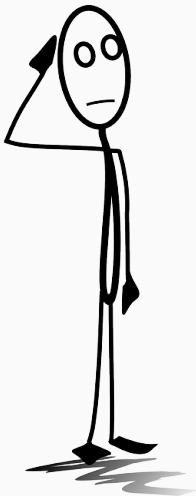

The BOW

The BOW

General idea



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



But are all terms equally important?

The BOW

A cleaner BOW representation

Room for improvement

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

pruning How can we remove unnecessary words?

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

OK, good enough, perfect?

.split()

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

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

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

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
 - BUT: Assumes lists of *sentences*.
- Solution: Build an own (combined) tokenizer (next slide)!

OK, good enough, perfect?

```
1 import nltk
2 import regex
3
4 class MyTokenizer:
5     def tokenize(self, text):
6         tokenizer = nltk.tokenize.TreebankWordTokenizer()
7         result = []
8         word = r"\p{letter}"
9         for sent in nltk.sent_tokenize(text):
10             tokens = tokenizer.tokenize(sent)
11             tokens = [t for t in tokens
12                       if regex.search(word, t)]
13             result += tokens
14         return result
15
16 mytokenizer = MyTokenizer()
17 tokens = [mytokenizer.tokenize(d) for d in docs]
```

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



Can you (try to) explain the code?

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

```
1 from sklearn.feature_extraction.text import CountVectorizer
2 cv = CountVectorizer()
3 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 the custom NLTK-based external tokenizer we created before! `cv = CountVectorizer(tokenizer=mytokenizer.tokenize)`

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 the custom NLTK-based external tokenizer we created before! `cv = CountVectorizer(tokenizer=mytokenizer.tokenize)`

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 stopwords list.

But we can also remove stopwords “by hand” of course using either a for loop (like we did for punctuation removal) or by modifying the tokenizer (try it!).

General idea

- Idea behind both stopwords removal and tf-idf: too frequent words are uninformative
- (possible) downside stopwords 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

General idea

- Idea behind both stopwords removal and tf-idf: too frequent words are uninformative
- (possible) downside stopwords 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

General idea

- Idea behind both stopwords removal and tf-idf: too frequent words are uninformative
- (possible) downside stopwords 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

General idea

- Idea behind both stopwords removal and tf-idf: too frequent words are uninformative
- (possible) downside stopwords 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 stopwords removal

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

CountVectorizer, other tokenization, stopwords removal (pay attention that stopwords list uses same tokenization!):

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

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

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

All together: tf-idf, explicit stopwords removal, pruning

```
1 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?

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:

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

```
1 [['this', 'be', 'a', 'text'], ['-PRON-', 'have', 'not', 'see', 'John', "is", 'derring', '-', 'do',
  ', ', 'second', 'sentence', '!']]
```

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:

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

```
1 [['this', 'be', 'a', 'text'], ['-PRON-', 'have', 'not', 'see', 'John', "'s", 'derring', '-', 'do',
  ', .', 'second', 'sentence', '!']]
```

The BOW

The order of preprocessing steps

Option 1

Preprocessing only through Vectorizer

“Just use CountVectorizer or TfidfVectorizer with the appropriate options.”

- pro: No double work, efficient if your main goal is a sparse matrix (for ML?) anyway
- con: you cannot “see” the preprocessed texts

Option 2

Extensive preprocessing without Vectorizer

“Remove stopwords, punctuation etc. and store in a string with spaces”

```
1 cleaneddocs = [" ".join(re.findall(r"\w\w+", d)).lower() for d in docs]
2 cleaneddocswithoutstopwords = [" ".join([w for w in d.split() if w not
    in mystopwords]) for d in cleaneddocs]
```

```
1 ['this is text', 'haven seen john derring do second sentence']
2 ['text', 'seen john derring second sentence']
```

Yes, this list comprehension looks scary – you can make a more elaborate for loop instead

- pro: you can read (and store!) the preprocessed docs
- pro: even the most stupid vectorizer (or wordcloud tool) can split the resulting string later on
- con: potentially double work (for you *and* the computer)



How would you do it?

Sometimes, I go for Option 2 because

- I like to inspect a sample of the documents
- I can re-use the cleaned docs irrespective of the Vectorizer

But at other times, I opt of Option 1 instead because

- I want to systematically compare the effect of different choices in a machine learning pipeline (then I can simply vary the vectorizer instead of the data)
- I want to use techniques that are geared towards little or no preprocessing (deep learning)

The BOW

How further?

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 recognition.



Any questions?

Next steps

<https://github.com/uvacw/teaching-bdaca/blob/main/12ec-course/week06/exercises/>

