A Practical Introduction to Machine Learning in Python
Day 2 - Tuesday Morning
»From text to features: Natural Language Processing«

Rupert Kiddle
Marieke van Hoof

r.t.kiddle@vu.nl
m.vanhoof@uva.nl

September 16, 2024

Today

- Bottom-up vs. top-down
- Basic string operations
  - A cleaner BOW representation
    - Better tokenization
    - Stopword removal
  - Stemming and lemmatization
  - ngrams
  - The order of preprocessing steps
  - How further?

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 ⇒ 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

Bottom-up vs. top-down

Bottom-up

- Count most frequently occurring words
- Maybe better: Count combinations of words ⇒ 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

```python
from collections import Counter

texts = ["I really really really love him, I do", "I hate him"]

for t in texts:
    print(Counter(t.split()).most_common(3))
```

```
[('really', 3), ('I', 2), ('love', 1)]
[('I', 1), ('hate', 1), ('him', 1)]
```

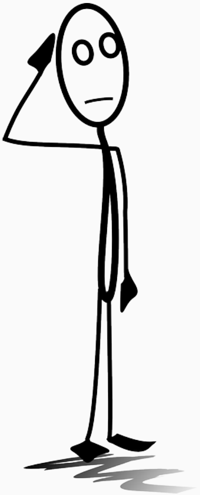
A simple top-down approach

```
texts = ["I really really really love him, I do", "I hate him"]
features = ['really', 'love', 'hate']

for t in texts:
    print(f"\nAnalyzing '{t}':")
    for f in features:
        print(f"{f} occurs {t.count(f)} times")
```

```
Analyzing 'I really really really love him, I do':
really occurs 3 times
love occurs 1 times
hate occurs 0 times

Analyzing 'I hate him':
really occurs 0 times
love occurs 0 times
hate occurs 1 times
```



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".

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".

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())`)

For today, we assume that our data are a list of strings – adapt accordingly for pandas.

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())`)

For today, we assume that our data are a list of strings – adapt accordingly for pandas.

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())`)

For today, we assume that our data are a list of strings – adapt accordingly for pandas.

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())`)

For today, we assume that our data are a list of strings – adapt accordingly for pandas.

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())`)

For today, we assume that our data are a list of strings – adapt accordingly for pandas.

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())`)

For today, we assume that our data are a list of strings – adapt accordingly for pandas.

An example says more than 1000 words. . .

Two examples says even more:

Combine both

The toolbox at a glance

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 with "a" replaced by "b"
- `.count("bla")` counts how often substring "bla" occurs
- `.isdigit()` true if only numbers

Use tab completion for more!

Basic string operations

A cleaner BOW representation

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?

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

OK, good enough, perfect?

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

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

```
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]
```

```
1  [['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.

Basic string operations

Stemming and lemmatization

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 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]
```

```
[['this', 'be', 'a', 'text'], ['-PRON-', 'have', 'not', 'see', 'John', "'s", '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:

```
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]
```

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

Basic string operations

ngrams

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?)**

Basic string operations

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

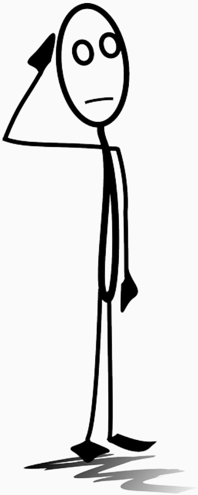
"Remove stopwords, punctuation etc. and store in a string with spaces"

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

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

Basic string operations

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

collocations *n*grams that appear more frequently than expected

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.

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 own 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]
```

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['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.

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.