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

Damian Trilling
Anne Kroon

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

September 28, 2021

Today

- Bottom-up vs. top-down
  - Approaches to working with text
- Natural Language Processing
  - Better tokenization
  - Stopword and punctuation removal
  - Stemming and lemmatization
  - ngrams
- Advanced NLP
  - Parsing sentences
- ACA using regular expressions
  - What is a regexp?
  - Using a regexp in Python

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

Boumans2016

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

```
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)]
```

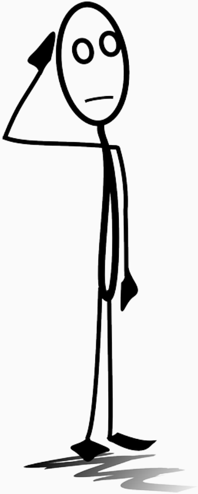
A simple top-down approach

```
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".

Bottom-up vs. top-down

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

Use tab completion for more!

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?

parse sentences How can identify and encode grammatical functions of tokens?

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.', '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.

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

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

```
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.

Removing punctuation

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

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

```
from string import punctuation
doc = "Today is @Toni's Birthday!!!"
"".join([w for w in doc if w not in punctuation])
```

```
'Today is Tonis Birthday'
```

Natural Language Processing

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', '!']]
```

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]))
```

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]))
```

Natural Language Processing

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?)**

Advanced NLP

Process and/or enrich

Advanced NLP

We did a lot of BOW (and some POS-tagging), but we can get more

- Named Entity Recognition (NER) to get names of people, organizations, ...
- Dependency Parsing to find out exact relationships ⇒ nltk, Stanford, FROG, Spacy

Advanced NLP

Parsing sentences

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 using NLTK

Tokenize a sentence, and “tag” the tokenized sentence:

```
tokens = nltk.word_tokenize(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 using NLTK

Tokenize a sentence, and “tag” the tokenized sentence:

```
tokens = nltk.word_tokenize(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]`!

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('Een 38-jarige vrouw uit Zeist en twee mannen moeten 24
    maanden de cel in voor de gecordineerde oplichting van Rabobank-
    klanten.')
for ent in doc.ents:
    print(ent.text,ent.label_)
```

returns:

```
Zeist LOC
Rabobank ORG
```

More NLP

http://nlp.stanford.edu http://spacy.io http://nltk.org
https://www.clips.uantwerpen.be/pattern

Main takeaway

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

Regular expressions

Automated content analysis using 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

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

Regex example

- Let's say we wanted to remove everything but words from a tweet
- We could do 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 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 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?

See the table in the book!

Regular expressions

Using a regexp in Python

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"

Use the less-known but more powerful module `regex` instead to support all dialects used in the book

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"

Use the less-known but more powerful module `regex` instead to support all dialects used in the book

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 (f"Your points: {}result.group(1)}, 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))
```

Practice yourself!

Let's take some time to write some regular expressions. Write a script that

- extracts URLS form a list of strings
- removes everything that is not a letter or number from a list of strings

(first develop it for a single string, then scale up)

More tips: http://www.pyregex.com/

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]
```

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.

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.