4 | Computational Analyses of Language Use

Extract information from texts

Marco Petolicchio

November 2020

Prerequisites

- `Python` v.3 installed
- `NLTK` library installed and functioning
- Other libraries as `NumPY`, `SciPY`, and so on
- A coffee, tea, or something that you prefer.

The subsequent part will cover some different approaches over feature and information extraction. Note that these instruments are thought for language processing, while they can be used in different fields as computer vision, image recognition, genetic code analysis, and everything you want.

Let’s start :)

Bag of Words

- Bag-of-Word (BOW) model is widely used in feature extraction from texts.
- We can think that each document contains some words from a list.

Let’s imagine that we have two different sentences (inserted here with lowercase for preprocessing purposes):

```
s_1 = ”this sentence contains words.”
s_2 = ”also this one.”
```

The resultant text would be the union of the two (let’s omit the dot):

```
text = ”this sentence contains words also this one”
```

The BOW model permits to obtain quantitative information about the frequency of the items from a list. Now let's assign a binary value which represent the presence (1) or the absence (0) of the items in respect to each sentence:

```
t_1 = [1,1,1,1,0,1,0]
t_2 = [1,0,0,0,1,1,1]
```

For example now that we have a measure of the occurrencies, we could make a tool which tries to predict the genre of the text. BOW is used for spam-filtering purposes in Email, for example, where a recognition of linguistic pattern is made towards mathematical procedures.

The BOW model doesn't care about meaning, context, and so on: the only scope is to consider if a certain word appears in a known document.

Python code

```
import nltk

import matplotlib

import string

from nltk import *

from nltk import word_tokenize

from nltk.corpus import brown

from nltk.corpus import stopwords


def vectorize(tokens):

    vector=[]

    for word in filtered_words:

        vector.append(tokens.count(word))

    return vector
```

```
s_1 = ”this sentence contains words.”

s_2 = ”also this one.”

t_1 = word_tokenize(s_1)

t_2 = word_tokenize(s_2)

words = set(t_1 + t_2)

stop_words = stopwords.words('english')

filtered_words = [word for word in words if word not in stop_words]

v_1 = vectorize(t_1)

v_2 = vectorize(t_2)

print(v_1)

print(v_2)
```

N-Grams

In computational linguistics, a **n-gram** is a sequence of *n* elements from a given text. For example, we can consider phonemes, words, sequences as the bases of the computation.

N-Grams are useful for probabilistic purposes, as to guess the next item of a sequence. Also we can use n-grams to analyze the syntactic sequence, or the character succession, and so on.

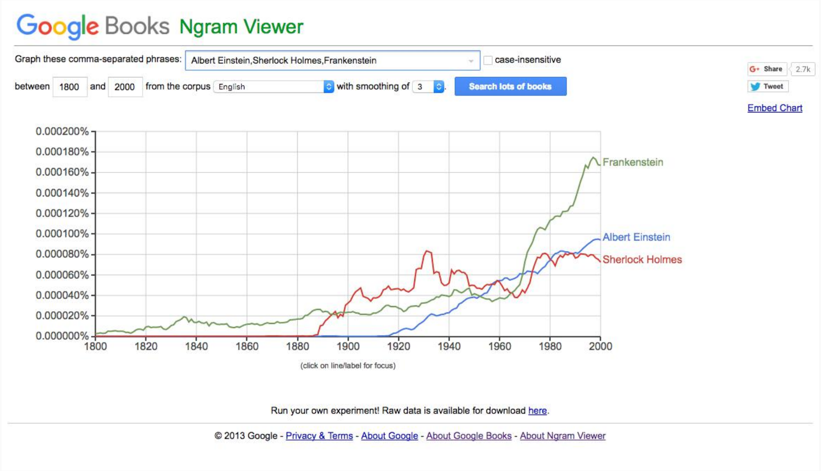
The pronunciation for the items follow the latin: unigram (1), bigram (2), trigram (3), etc.

Let’s see an example based on words (from the incipit of G.Orwell, 1984):

- `sample_text = ”It was a bright cold day in April, and the clocks were striking thirteen.”`
- `2-gr = ”It was, was a, a bright, bright cold, cold day, day in, in April, April and, and the, the clocks, clocks were, were striking, striking thirteen”`
- `3-gr = ”It was a, was a bright, a bright cold, bright cold day, cold day in, day in April, in April and, April and the, and the clocks, the clocks were, clocks were striking, were striking thirteen”`

Despite from BOW model, ngrams consist of ordered sets that conserve the ordering of the items.

In the recent years, Google developed a tool which aims to display ngrams to the user, who can select the corpus to search in:


Google Books Ngram Viewer
Graph these comma-separated phrases: Albert Einstein,Sherlock Holmes,Frankenstein
case-insensitive
between 1800 and 2000 from the corpus English with smoothing of 3.
Search lots of books
Share 2.7k
Tweet
Embed Chart
Frankenstein
Albert Einstein
Sherlock Holmes
0.000200%
0.000180%
0.000160%
0.000140%
0.000120%
0.000100%
0.000080%
0.000060%
0.000040%
0.000020%
0.000000%
1800
1820
1840
1860
1880
1900
1920
1940
1960
1980
2000
(click on line/label for focus)
Run your own experiment! Raw data is available for download here.
© 2013 Google - Privacy & Terms - About Google - About Google Books - About Ngram Viewer

Python code

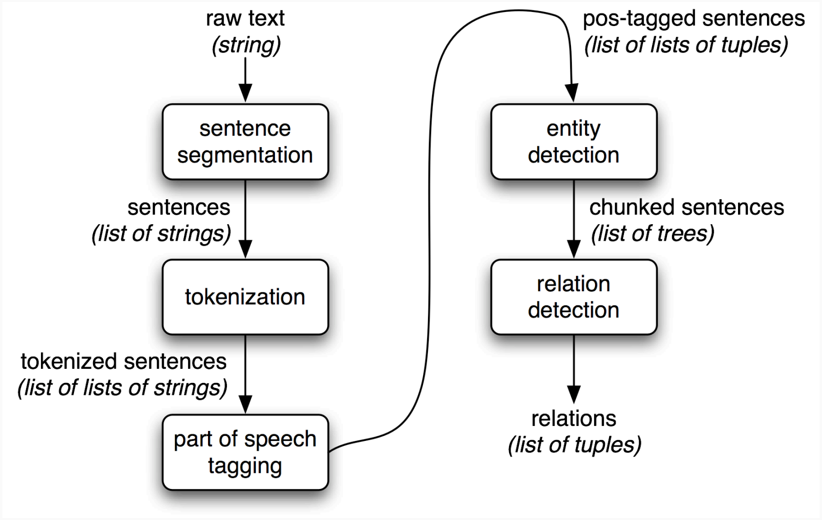
We can use NLTK to generate the n-grams:

```
import nltk
from nltk.util import ngrams


# Function to generate n-grams from sentences.
def extract_ngrams(data, num):
    n_grams = ngrams(nltk.word_tokenize(data), num)
    return [ ' '.join(grams) for grams in n_grams]


data = ”It was a bright cold day in April, and the clocks were striking thirteen.”


print(”2-gram: ”, extract_ngrams(data, 2))
print(”3-gram: ”, extract_ngrams(data, 3))
```

Named Entity recognition

Despite from the previous approaches, with Named Entity Recognition (NER) we enter into a more difficult field of Natural Language Processing, which involves either a semantical analysis than a syntactic and contextual one. Extracting particular information from texts is not trivial: it needs to be processed towards an actual set of data for some specific real-world usecases, as extracting geographical information, person, and so on. Imagine how many fields take advantage of it!

The way we can do NER needs a complex pipeline, which necessitates the cleaning of the corpus, the preprocessing of the texts, and then the tagging of the Parts-of-Speech, in order to obtain a set of tuples of the elements.



After that POS-tagging take place, we would have a similar perspective:

```
    ('It', 'PRP'),
    ('was', 'VBD'),
    ('a', 'DT'),
    ('bright', 'JJ'),
    ('cold', 'JJ'),
```

where each item is appended with the respective POS.

At this point we can join different word into *chunks* (as for example a complex DP as DET, Adj, NP).

For Python code see the NLTK textbook.

NER is one of the most promising way to extract information, e.g. in Q/A replies or Semantic Web queries.

In the next lesson we will see something deeper about classification and training models.