Data preprocessing

- **Tokenization - splitting the text into tokens (minimal meaningful units); we tokenize a *sentence* into *words* or an *article* into *sentences***

- **POS - tagging – assigning a part of speech to each word (noun, adjective etc.)**

- **Stemming - the process of reducing a word to its stem (ex.: *walking* is reduced to *walk*)**

- **Lemmatization - similar to stemming except it operates also by including the knowledge about the word's context (ex.: *good* and *better*)**

- **NER - Named Entity Recognition – labels a sequence of words that represents the names of things (ex.: a company, a street, a person's name)**

- **Parsing - a parser analyzes the grammar of a text in order to extract or determine its structure**

The difference between stemming and lemmatization

The aim of both processes is the same: reducing the inflectional forms of each word into a common base or root. However, these two methods are not exactly the same.

- **Stemming algorithms work by cutting off the end or the beginning of the word, taking into account a list of common prefixes and suffixes that can be found in an inflected word. This indiscriminate cutting can be successful in some occasions, but not always. Example: form – studies (the verb "study"); suffix – es; stem – studi**

- **Lemmatization takes into consideration the morphological analysis of the words. To do so, it is necessary to have detailed dictionaries which the algorithm can look through in order to link the form back to its lemma. Example:**
  form - studies; morphological information - third person, singular number, present tense of the verb *study*; lemma – study

Note:

1. **A lemma is the base form of all its inflectional forms (toate “formele flexionare” ale cuvantului), whereas a stem isn’t.**
2. **Regular dictionaries are lists of lemmas, not stems.**

   (Lema este forma din dictionar).

NLP tools and libraries

- **Core NLP (Stanford CoreNLP): contains tools for tagging, parsing, analyzing text; is written in JAVA; supports other languages besides English (Arabic, French, Chinese, German, Spanish etc.)**

- **spaCy: created for industrial strength NLP; provides most mentioned functionalities (POS-tagging, tokenization etc.); supports various languages (German, Spanish, Portuguese, French, Italian, Dutch etc.); the 2017 version has 92,6% accuracy**

- **genism: for semantic analysis (text mining); the author is a Czech researcher**

- **NLTK – Natural Language Toolkit – "the mother of all NLP libraries": developed since 2001; well maintained and modular; comes with big amount of data, corpora and trained modules; book:**

  nltk.org/book

NLTK

Tokenization

- **The process of breaking up text into smaller pieces (tokens)**
- **Token: can be a word or a sentence**
- **`import nltk` and see demo file**

Stop words

- **they are removed from the text**
- **usually they are language specific**
- **they are words like "then", "before", "between", the" etc.**
- **these are words that are filtered out before processing natural text**
- **removing them is part of data processing (a stop word can be dominant in a text and must be removed from the text because it doesn't import anything to what the text is about)**
- **with NLTK you import stop words corresponding to the language your text is written in (NLTK provides lists of stop words)**
- **punctuation must also be removed**
- **see demo file**

POS-tagging

- **the process of assigning part of speech tags (noun, verb, adjective etc.) to tokens (words)**

```
>>> import nltk
>>> from nltk import pos_tag
…
>>> text = nltk.word_tokenize("And now for something completely different")
>>>nltk.pos_tag(text)
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completely', 'RB'), ('different', 'JJ')]
```

where:
CC designates a coordinating conjunction
RB designates an adverb
IN designates a preposition
NN designates a noun
JJ designates an adjective

- **NLTK provides documentation for each tag, which can be queried using the tag**

```
nltk.help.upenn_tagset('RB')
```

Named Entity Recognition

- **NER is used to recognize entities in a text - like people, organisations, places etc.**
- **One must import named entity chunk and also a sentence tokenizer:**

```
from nltk import ne_chunk, sent_tokenize, word_tokenize
```

where:

sent_tokenize(text) - returns a sentence-tokenized copy of text, using NLTK's recommended sentence tokenizer

word_tokenize(text) - returns a tokenized copy of text, using NLTK's recommended word tokenizer

ne_chunk(tagged_tokens) - Use NLTK's currently recommended named entity chunker to chunk the given list of tagged tokens

- **See demo file**

TF - IDF

TF: term frequency

IDF: Inverse Document Frequency

```
from math import log
from nltk.stem import PorterStemmer
from nltk.corpus import stopwords
import string
from nltk import corpus
import numpy
# importing the corpus into data variable
data = corpus.brown
# Using Porter Stemmer
stemmer = PorterStemmer()
# Building the list of stop words
# We will filter the tokens against it
stopwords = set(stopwords.words('english'))
stopwords = stopwords.union(string.punctuation)
# Limiting the number of files we will use
# uncomment the following line to use all of the
# corpus files
# fileIds = data.fileids()
fileids = data.fileids()[:30]
idf_matrix = []
dictionary = dict()
# total count f words in the corpus
words_count = 0
# total count of document in the corpus
documents_count = len(fileids)
# holds the words before and after filtering : Stemming
filtered = dict()
# to save the total counts of every word per file
frequencies = dict()
for fileid in fileids:
```

```
frequencies[fileid] = dict()
# filtering corpus
for fileid in fileids:
for word in data.words(fileid):
# Skipping if it is a stop word
if word in stopwords:
continue
# Before and after filtering
before_word = word
if before_word in filtered:
word = filtered[before_word]
else:
# stemming the word
word = stemmer.stem(word)
filtered[before_word] = word
if word in frequencies[fileid]:
frequencies[fileid][word] += 1
else:
frequencies[fileid][word] = 1
# saving all the words in a dictionary
if word not in dictionary:
dictionary[word] = words_count
words_count += 1
# Calculating TF #
# indexes of non zero values
tf_matrix = []
nonzeros = []
for fileid in fileids:
tf_vector = [0] * words_count
nonzeros_vec = []
```

```
for word in frequencies[fileid].keys():
f = frequencies[fileid][word]
tf_vector[dictionary[word]] = f
if f > 0:
nonzeros_vec.append(dictionary[word])
nonzeros.append(nonzeros_vec)
tf_matrix.append(tf_vector)
# Calculating IDF
idf_matrix = [0] * words_count
for fileid in fileids:
for word in frequencies[fileid].keys():
idf_matrix[dictionary[word]] += 1
# Calculating TF-IDF matrix#
tfidf = []
for i in range(documents_count):
vector = [0] * words_count
for j in nonzeros[i]:
tf_value = tf_matrix[i][j]
idf_value = idf_matrix[j]
tf_value = 1 + log(tf_value, 2)
idf_value = log(1 + documents_count/ float(idf_value), 2)
vector[j] = tf_value * idf_value
tfidf.append(vector)
print("------ Top 10 : Keywords per document -------")
for i in range(len(tfidf)):
print("--- Document : " + str(fileids[i]))
vector = tfidf[i]
sorted = numpy.argsort(vector)[::-1]
for ind in sorted[:15]:
stem = list(dictionary.keys())[list(dictionary.values()).index(ind)]
```

```
beforeStemming = list(filtered.keys())[list(filtered.values()).index(stem)]
print(beforeStemming + " -- " + str(vector[ind]))
```

TF = (number of times the term appears in a document) / (total number of terms in the document)

IDF = log(total number of documents)/(number of documents with the word in it)

TFIDF = TF * IDF

<u>Example</u>: We look at the Brown corpus, coming with NLTK, and we extract the key words

- **See the program TF-IDF.py on the next slide**
- **The code thereuses Porterstemmer from NLTK**
- **For the code to run smoothly you need to install the Python library called `numpy`**
- **The program initializes a list of stop words (for English, taken from NLTK)**

POS-Tagging and Grammars

POS-tagging forms the basis of any further syntactic analyses.

Exploring the In-Built Tagger

```
import nltk
#  Create a variable called simpleSentence and assign a
# string to it
simpleSentence = "Bangalore is the capital of Karnataka."
# Use the nltk built-in tokenizer function called
# "word_tokenize"
wordsInSentence = nltk.word_tokenize(simpleSentence)
# It breaks a given sentence into words and returns a Python
# list data type; we assign it to the variable wordsInSentence
#Display the given data structure (a Python list)
print(wordsInSentence)
```

```
['Bangalore', 'is', 'the', 'capital', 'of',
'Karnataka', '.']
```

```
# Invoke the nltk built-in tagger, called pos_tag, which takes
# the list of words and identifies the part of speech of each
# word in wordsInSentence
partsOfSpeechTags =
nltk.pos_tag(wordsInSentence)
# We see a list of tuples where each tuple has the tokenized
# word and the POS identifier
partsOfSpeechTags =
nltk.pos_tag(wordsInSentence)
# invoke the print function
print(partsOfSpeechTags)
```

```
[('Bangalore', 'NNP'), ('is', 'VBZ'), ('the', 'DT'), ('capital',
'NN'), ('of', 'IN'), ('Karnataka', 'NNP'), ('.', '.')]
```

Learning to Write Your Own Grammar

```
# import the string module into your own program
import string
#import the generate function from nltk parse.generate
# module
from nltk.parse.generate import generate
# the grammar can contain some production rules; the
# starting symbol is ROOT
productions = [
                "ROOT -> WORD",
                 "WORD -> NUMBER LETTER",
                 "WORD -> LETTER NUMBER",
               ]
# all production rules are converted to a string
grammarString = "\n".join(productions)
# a new grammar object
# we use the CFG.fromstring method which takes the
# grammarString variable that was just created
grammar=nltk.CFG.fromstring (grammarString)
```

```
# print the grammar
print(grammar)
```

Writing a Recursive CFG

(CFG = Context-Free Grammar)

```
productions = [
                    "ROOT->WORD"
                    "WORD->' '"
                    [
```

 # Retrieve the list of decimal digits as a list in the alphabets
 # variable

```
alphabets = list(string.digits)
```

 # Using the digits from 0 to 9 we add more productions to
 # our list

```
for alphabet in alphabets:
 productions.append("WORD->'{w}'WORD'{w}'".
 .format(w = alphabet))
```

 # concatenate all the rules

```
grammarString = "\n".join(productions)
```

 # create a new grammar object

```
grammar =
nltk.CFG.fromstring(grammarString)
print(grammar)
```

POS-tagging

- **the process of assigning part of speech tags (noun, verb, adjective etc.) to tokens (words)**

```
>>> import nltk
>>> from nltk import pos_tag
…
>>> text = nltk.word_tokenize("And now for something completely different")
>>>nltk.pos_tag(text)
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completely', 'RB'), ('different', 'JJ')]
```

where:
CC designates a coordinating conjunction
RB designates an adverb
IN designates a preposition
NN designates a noun
JJ designates an adjective

- **NLTK provides documentation for each tag, which can be queried using the tag**

```
nltk.help.upenn_tagset('RB')
```