05-natural-language-processing

October 21, 2021

Run in Google Colab

1 Natural Language Processing: Tools and Ideas

1.1 Introduction

Natural language processing (NLP) is a field of machine learning in which computers analyze, understand, and derive meaning from human language in a smart and useful way. By utilizing NLP, developers can organize and structure knowledge to perform tasks such as automatic summarization, translation, named entity recognition, relationship extraction, sentiment analysis, speech recognition, and topic segmentation.

NLP is characterized as a difficult problem in computer science. Human language is rarely precise, or plainly spoken. To understand human language is to understand not only the words, but the concepts and how they're linked together to create meaning. Despite language being one of the easiest things for the human mind to learn, the ambiguity of language is what makes natural language processing a difficult problem for computers to master.

1.2 The NLTK package

What is NLTK? The Natural Language Toolkit, or more commonly NLTK, is a suite of libraries and programs for symbolic and statistical natural language processing (NLP) for English written in the Python programming language. It was developed by Steven Bird and Edward Loper in the Department of Computer and Information Science at the University of Pennsylvania.NLTK includes graphical demonstrations and sample data. It is accompanied by a book that explains the underlying concepts behind the language processing tasks supported by the toolkit, plus a text book available also on line here

NLTK is intended to support research and teaching in NLP or closely related areas, including empirical linguistics, cognitive science, artificial intelligence, information retrieval, and machine learning.NLTK has been used successfully as a teaching tool, as an individual study tool, and as a platform for prototyping and building research systems. There are 32 universities in the US and 25 countries using NLTK in their courses. NLTK supports classification, tokenization, stemming, tagging, parsing, and semantic reasoning functionalities.

The latest version is NLTK 3.3. It can be used by students, researchers, and industrialists. It is an Open Source and free library. It is available for Windows, Mac OS, and Linux.

You can install nltk using pip installer if it is not installed in your Python installation. To test the installation:

- Open your Python IDE or the CLI interface (whichever you use normally)
- Type `import nltk` and press enter if no message of missing nltk is shown then nltk is installed on your computer.

After installation, nltk also provides test datasets to work within Natural Language Processing. You can download it by using the following commands in Python:

[1]:
```
import nltk
#nltk.download()
```

1.2.1 Using NLTK Corpus

You can use the NLTK Text Corpora which is a vast repository for a large body of text called as a Corpus which can be used while you are working with Natural Language Processing (NLP) with Python. There are many different types of corpora available that you can use with varying types of projects, for example, a selection of free electronic books, web and chat text and news documents on different genres.

In the online book site you can find everything you need to known to access the NLTK Corpus, in particular you can start from Chapter 2 - Accessing Text Corpora and Lexical Resources.

Here is an example of how you can use a corpora

[2]:
```
nltk.corpus.gutenberg.fileids()
```

[2]:
```
['austen-emma.txt',
 'austen-persuasion.txt',
 'austen-sense.txt',
 'bible-kjv.txt',
 'blake-poems.txt',
 'bryant-stories.txt',
 'burgess-busterbrown.txt',
 'carroll-alice.txt',
 'chesterton-ball.txt',
 'chesterton-brown.txt',
 'chesterton-thursday.txt',
 'edgeworth-parents.txt',
 'melville-moby_dick.txt',
 'milton-paradise.txt',
 'shakespeare-caesar.txt',
 'shakespeare-hamlet.txt',
 'shakespeare-macbeth.txt',
 'whitman-leaves.txt']
```

1.2.2 Loading Your Own Corpus

If you have your own collection of text files that you would like to access using the above methods, you can easily load them with the help of NLTK's `PlaintextCorpusReader`. Check the location of your files on your file system; in the following example, we have taken this to be the directory `C:\Corpus\EBA`. Whatever the location, set this to be the value of corpus_root.

The second parameter of the PlaintextCorpusReader initializer can be a list of fileids, like [‘a.txt’, ‘test/b.txt’], or a pattern that matches all fileids, like ’[abc]/.*.txt’

[3]:
```
import nltk
from nltk.corpus import PlaintextCorpusReader

corpus_root = '.\corpus\EBA'
corpus_list = PlaintextCorpusReader(corpus_root, '.*', encoding='latin-1')
corpus_list.fileids()
```

[3]:
```
['Final Guidelines on Accounting for Expected Credit Losses (EBA-
GL-2017-06).txt',
 'Final Guidelines on the management of interest rate risk arising from non-
trading activities.txt',
 'Final Report on Guidelines on LGD estimates under downturn conditions.txt',
 'Final Report on Guidelines on default definition (EBA-GL-2016-07).txt',
 'Final Report on Guidelines on uniform disclosure of IFRS9 transitional
arrangements (EBA-GL-2018-01).txt',
 'Final report on updated GL Funding Plans (EBA 9.12.2019).txt',
 'Final report on updated GL Funding Plans_(EBA-GL-2019-05)_09122019.txt']
```

Let’s pick out one of these text (for example the guideline on the interest rate), give it a short name, irrbb_w.

[4]:
```
id = "Final Guidelines on the management of interest rate risk arising from␣
 ↪non-trading activities.txt"
irrbb_w = corpus_list.words(id)
```

For more information about corpora readers see also *Bengfort B. et al. “Applied Text Analysis with Python” O’Reilly (2018) Chapter 2.*

1.3 Text Preprocessing

In this section I want to go over some important NLP concepts and show code examples on how to apply them on text data.

Once you are sure that all documents loaded properly, go on to **preprocess your texts**. This step allows you to **remove numbers, dealing with capitalization, common words, punctuation, and otherwise prepare your texts for analysis**

Data cleansing, though tedious, is perhaps the most important step in text analysis. As we will see, dirty data can play havoc with the results. Furthermore, as we will also see, data cleaning is invariably an iterative process as there are always problems that are overlooked the first time around.

Removing punctuation: Your computer cannot actually read. Punctuation and other special characters only look like more words to your computer and Python. Use the following to methods to remove them from your text.

[10]:
```
sentence = "Clear and effective communication is very important to us. Our␣
 ↪monetary policy becomes \
           more effective when our decisions are better understood. The media␣
 ↪play an important \
           role in this process and help keep us accountable to the European␣
 ↪public."
tokens = sentence.split()
# remove all tokens that are not alphabetic
words = [word for word in tokens if word.isalpha()]
print(words)
```

```
['Clear', 'and', 'effective', 'communication', 'is', 'very', 'important', 'to',
'Our', 'monetary', 'policy', 'becomes', 'more', 'effective', 'when', 'our',
'decisions', 'are', 'better', 'The', 'media', 'play', 'an', 'important', 'role',
'in', 'this', 'process', 'and', 'help', 'keep', 'us', 'accountable', 'to',
'the', 'European']
```

Note that this result is not completely correct. Why?

Converting to lowercase: As before, we want a word to appear exactly the same every time it appears. Since many languages **are** case sensitive, "Text" is not equal to "text" – and hence the rationale for converting to a standard case. We therefore change everything to lowercase. This is also a good time to check for any other special symbols that may need to be removed manually.

[17]:
```
# all lowercase
words = [word.lower() for word in words]
print(words)
```

```
['clear', 'and', 'effective', 'communication', 'is', 'very', 'important', 'to',
'us', 'our', 'monetary', 'policy', 'becomes', 'more', 'effective', 'when',
'our', 'decisions', 'are', 'better', 'understood', 'the', 'media', 'play', 'an',
'important', 'role', 'in', 'this', 'process', 'and', 'help', 'keep', 'us',
'accountable', 'to', 'the', 'european', 'public']
```

Removing numbers: Text analysts are typically not interested in numbers since these do not usually contribute to the meaning of the text. **However, this may not always be so. For example, it is definitely not the case if one is interested in getting a count of the number of times a particular year appears in a corpus**.

[19]:
```
# remove numbers
words = [w for w in words if not w.isdigit()]
```

Removing "Stop Words" (common words)

In every text, there are a lot of common, and uninteresting words that we generically call Stop WOrds. Stop words are words that may not carry any valuable information, like articles ("the"), conjunctions ("and"), or propositions ("with"). Why would you want to remove them? Because finding out that "the" and "a" are the most common words in your dataset doesn't tell you much about the data. Such words are frequent by their nature, and will confound your analysis if they remain in the text.

NLP Python libraries like NLTK usually come with an in-built stopword list which you can easily import.

[21]:
```
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

example_sent = "This is a sample sentence, showing off the stop words␣
 ↪filtration."
stop_words = set(stopwords.words('english'))
word_tokens = word_tokenize(example_sent)

filtered_sentence = [w for w in word_tokens if not w in stop_words]
print(filtered_sentence)
```

```
['This', 'sample', 'sentence', ',', 'showing', 'stop', 'words', 'filtration',
'.']
```

Note, that in some cases stop words matter. For example, in identifying negative reviews or recommendations. People will use stop words like "no" and "not" in negative reviews: "I will not buy this product again. I saw no benefits in using it".

Removing particular words:

The NLTK stopword list, however, only has around 200 stopwords. The stopword list which one can commonly use for text analysis may contains almost 600 words. So if you find that a particular word or two appear in the output, but are not of value to your particular analysis, you can remove them, specifically, from the text.

1.3.1 Tokenization

Tokenization is a process of splitting a text object into smaller units which are also called tokens. Examples of tokens can be words, numbers, engrams, or even symbols. Single words are called unigrams, two words bi-grams, and three words tri-grams.

The most commonly used tokenization process is White-space Tokenization.

[22]:
```
sentence = "Clear and effective communication is very important to us. Our␣
 ↪monetary policy becomes more \
            effective when our decisions are better understood. The media play␣
 ↪an important role in this \
            process and help keep us accountable to the European public."
tokens = sentence.split()
print(tokens)
```

```
['Clear', 'and', 'effective', 'communication', 'is', 'very', 'important', 'to',
'us.', 'Our', 'monetary', 'policy', 'becomes', 'more', 'effective', 'when',
'our', 'decisions', 'are', 'better', 'understood.', 'The', 'media', 'play',
'an', 'important', 'role', 'in', 'this', 'process', 'and', 'help', 'keep', 'us',
'accountable', 'to', 'the', 'European', 'public.']
```

When dealing with a new dataset it's often helpful to extract the most common words to get an idea of what the data is about. You usually want to extract the most common unigrams first, but it can also be useful to extract n-grams with larger n to identify patterns. NLTK has in-built bigrams, trigrams and ngrams functions.

[23]:
```
from nltk.tokenize import word_tokenize
from nltk.util import ngrams, bigrams, trigrams

sen = "Clear and effective communication is very important to us. Our monetary␣
 ↪policy becomes more \
            effective when our decisions are better understood. The media play␣
 ↪an important role in this \
            process and help keep us accountable to the European public."
nltk_tokens = word_tokenize(sen) #using tokenize from NLKT and not split()␣
 ↪because split() does not take into account punctuation
```

[24]:
```
# remove all tokens that are not alphabetic
words = [word for word in nltk_tokens if word.isalpha()]
print(words)
```

```
['Clear', 'and', 'effective', 'communication', 'is', 'very', 'important', 'to',
'us', 'Our', 'monetary', 'policy', 'becomes', 'more', 'effective', 'when',
'our', 'decisions', 'are', 'better', 'understood', 'The', 'media', 'play', 'an',
'important', 'role', 'in', 'this', 'process', 'and', 'help', 'keep', 'us',
'accountable', 'to', 'the', 'European', 'public']
```

This is better!

[25]:
```
#splitting sentence into bigrams and trigrams
print(list(bigrams(nltk_tokens)))
print(list(trigrams(nltk_tokens)))
```

```
[('Clear', 'and'), ('and', 'effective'), ('effective', 'communication'),
('communication', 'is'), ('is', 'very'), ('very', 'important'), ('important',
'to'), ('to', 'us'), ('us', '.'), ('.', 'Our'), ('Our', 'monetary'),
('monetary', 'policy'), ('policy', 'becomes'), ('becomes', 'more'), ('more',
'effective'), ('effective', 'when'), ('when', 'our'), ('our', 'decisions'),
('decisions', 'are'), ('are', 'better'), ('better', 'understood'),
('understood', '.'), ('.', 'The'), ('The', 'media'), ('media', 'play'), ('play',
'an'), ('an', 'important'), ('important', 'role'), ('role', 'in'), ('in',
'this'), ('this', 'process'), ('process', 'and'), ('and', 'help'), ('help',
'keep'), ('keep', 'us'), ('us', 'accountable'), ('accountable', 'to'), ('to',
'the'), ('the', 'European'), ('European', 'public'), ('public', '.')]
[('Clear', 'and', 'effective'), ('and', 'effective', 'communication'),
('effective', 'communication', 'is'), ('communication', 'is', 'very'), ('is',
'very', 'important'), ('very', 'important', 'to'), ('important', 'to', 'us'),
('to', 'us', '.'), ('us', '.', 'Our'), ('.', 'Our', 'monetary'), ('Our',
'monetary', 'policy'), ('monetary', 'policy', 'becomes'), ('policy', 'becomes',
'more'), ('becomes', 'more', 'effective'), ('more', 'effective', 'when'),
```

```
('effective', 'when', 'our'), ('when', 'our', 'decisions'), ('our', 'decisions',
'are'), ('decisions', 'are', 'better'), ('are', 'better', 'understood'),
('better', 'understood', '.'), ('understood', '.', 'The'), ('.', 'The',
'media'), ('The', 'media', 'play'), ('media', 'play', 'an'), ('play', 'an',
'important'), ('an', 'important', 'role'), ('important', 'role', 'in'), ('role',
'in', 'this'), ('in', 'this', 'process'), ('this', 'process', 'and'),
('process', 'and', 'help'), ('and', 'help', 'keep'), ('help', 'keep', 'us'),
('keep', 'us', 'accountable'), ('us', 'accountable', 'to'), ('accountable',
'to', 'the'), ('to', 'the', 'European'), ('the', 'European', 'public'),
('European', 'public', '.')]
```

The sents() function divides the text up into sentencese, where each sentence is a list of words

[26]:
```
irrbb_s = corpus_list.sents(id)
irrbb_s
```

[26]:
```
[['GUIDELINES', 'ON', 'THE', 'MANAGEMENT', 'OF', 'INTEREST', 'RATE', 'RISK'],
 ['ARISING', 'FROM', 'NON', '-', 'TRADING', 'BOOK', 'ACTIVITIES'], …]
```

1.4 Lemmatisation and Stemming

Stemming and Lemmatization are Text Normalization (or sometimes called Word Normalization) techniques in the field of Natural Language Processing that are used to prepare text, words, and documents for further processing. Stemming and Lemmatization have been studied, and algorithms have been developed in Computer Science since the 1960's.

Languages we speak and write are made up of several words often derived from one another. When a language contains words that are derived from another word as their use in the speech changes is called **Inflected Language**. Inflection is the modification of a word to express different grammatical categories such as tense, case, voice, aspect, person, number, gender, and mood. An inflection expresses one or more grammatical categories with a prefix, suffix or infix, or another internal modification such as a vowel change.

Typically a large corpus will contain many words that have a common root – for example: offer, offered and offering. Lemmatisation and stemming both refer to a process of reducing a word to its root. The difference is that stem might not be an actual word whereas, a lemma is an actual word. It's a handy tool if you want to avoid treating different forms of the same word as different words. Let's consider the following example:

- Lemmatising: considered, considering, consider → "consider"
- Stemming: considered, considering, consider → "consid"

Stemming and Lemmatization are widely used in tagging systems, indexing, SEOs, Web search results, and information retrieval. For example, searching for fish on Google will also result in fishes, fishing as fish is the stem of both words.

NLTK comes with many different in-built lemmatisers and stemmers, so just plug and play.

1.4.1 NLTK Stemming

[27]:
```
from nltk.stem import WordNetLemmatizer, PorterStemmer, SnowballStemmer

stemmer = PorterStemmer()
lemmatizer = WordNetLemmatizer()

word = "considering"

stemmed_word =  stemmer.stem(word)
lemmatised_word = lemmatizer.lemmatize(word)

print(stemmed_word)
print(lemmatised_word)
```

```
consid
considering
```

We can use any of the text of nltk corpora to make some test. For example:

[28]:
```
text_file=nltk.corpus.gutenberg.words('melville-moby_dick.txt')
my_lines_list=[]
for line in text_file:
    my_lines_list.append(line)
#my_lines_list
```

[29]:
```
stemmer=PorterStemmer()

def stemSentence(sentence):
    token_words=word_tokenize(sentence)
    token_words
    stem_sentence=[]
    for word in token_words:
        stem_sentence.append(stemmer.stem(word))
        stem_sentence.append(" ")
    return "".join(stem_sentence)
```

[30]:
```
stem_file=open("stemming_example.txt",mode="a+", encoding="utf-8")

for line in my_lines_list:
    stem_sentence=stemSentence(line)
    stem_file.write(stem_sentence)

stem_file.close()
```

Python nltk provides not only two English stemmers: PorterStemmer and LancasterStemmer but also a lot of non-English stemmers as part of SnowballStemmers, ISRIStemmer, RSLPSStemmer. Python NLTK included SnowballStemmers as a language to create to create non-English stemmers. One can program one's own language stemmer using snowball. Currently, it supports the following

languages: - Danish - Dutch - English - French - German - Hungarian - Italian - Norwegian - Porter - Portuguese - Romanian - Russian - Spanish - Swedish

ISRIStemmer is an Arabic stemmer and RSLPStemmer is stemmer for the Portuguese Language.

1.4.2 NLTK Lemmatization

As we have already said, lemmatization, unlike Stemming, reduces the inflected words properly ensuring that the root word belongs to the language. In Lemmatization root word is called Lemma. A lemma (plural lemmas or lemmata) is the canonical form, dictionary form, or citation form of a set of words.

For example, runs, running, ran are all forms of the word run, therefore run is the lemma of all these words. Because lemmatization returns an actual word of the language, it is used where it is necessary to get valid words.

Python NLTK provides WordNet Lemmatizer that uses the WordNet Database to lookup lemmas of words. WordNet is a lexical database for the English language, which was created by Princeton, and is part of the NLTK corpus.

[31]:
```
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer()

sentence = "He was running and eating at same time. He has bad habit of␣
 ↪swimming after playing long hours in the Sun."
punctuations="?:!.,;"
sentence_words = nltk.word_tokenize(sentence)
for word in sentence_words:
    if word in punctuations:
        sentence_words.remove(word)

sentence_words
print("{0:20}{1:20}".format("Word","Lemma"))
for word in sentence_words:
    print ("{0:20}{1:20}".format(word,wordnet_lemmatizer.lemmatize(word)))
```

```
Word                Lemma
He                  He
was                 wa
running             running
and                 and
eating              eating
at                  at
same                same
time                time
He                  He
has                 ha
bad                 bad
habit               habit
of                  of
```

```
swimming            swimming
after               after
playing             playing
long                long
hours               hour
in                  in
the                 the
Sun                 Sun
```

In the above output, you must be wondering that no actual root form has been given for any word, this is because they are given without context. You need to provide the context in which you want to lemmatize that is the parts-of-speech (POS). This is done by giving the value for pos parameter in `wordnet_lemmatizer.lemmatize`

[32]:
```
for word in sentence_words:
    print ("{0:20}{1:20}".format(word,wordnet_lemmatizer.lemmatize(word,␣
 ↪pos="v")))
```

```
He                  He
was                 be
running             run
and                 and
eating              eat
at                  at
same                same
time                time
He                  He
has                 have
bad                 bad
habit               habit
of                  of
swimming            swim
after               after
playing             play
long                long
hours               hours
in                  in
the                 the
Sun                 Sun
```

1.4.3 Stemming or lemmatization?

After going through this section, you may be asking yourself when should I use Stemming and when should I use Lemmatization? We have seen the following points:

Stemming and Lemmatization both generate the root form of the inflected words. The difference is that stem might not be an actual word whereas, lemma is an actual language word.

Stemming follows an algorithm with steps to perform on the words which makes it faster. Whereas, in lemmatization, you used WordNet corpus and a corpus for stop words as well to produce lemma

which makes it slower than stemming. You also had to define a parts-of-speech to obtain the correct lemma.

So when to use what! The above points show that if speed is focused then stemming should be used since lemmatizers scan a corpus which consumed time and processing. It depends on the application you are working on that decides if stemmers should be used or lemmatizers. If you are building a language application in which language is important you should use lemmatization as it uses a corpus to match root forms.

1.5 Part-of-Speech Tagging

1.5.1 What is POS Tagging?

The process of classifying words into their parts of speech and labeling them accordingly is known as part-of-speech tagging or POS-tagging, or simply tagging. The part of speech **explains how a word is used in a sentence**. There are eight main parts of speech:

- nouns,
- pronouns,
- adjectives,
- verbs,
- adverbs,
- prepositions,
- conjunctions
- interjections.

Examples:

- **Noun (N)** : Daniel, London, table, dog, teacher, pen, city
- **Verb (V)** : go, speak, run, eat, play, live, walk, have, like, are, is
- **Adjective (ADJ)** : big, happy, green, young, fun, crazy, three
- **Adverb (ADV)** : slowly, quietly, very, always, never, too, well, tomorrow
- **Preposition (P)** : at, on, in, from, with, near, between, about, under
- **Conjunction (CON)** : and, or, but, because, so, yet, unless, since, if
- **Pronoun (PRO)** : I, you, we, they, he, she, it, me, us, them, him, her, this
- **Interjection (INT)** : Ouch! Wow! Great! Help! Oh! Hey! Hi!

Most POS are divided into sub-classes. POS Tagging simply means labeling words with their appropriate Part-Of-Speech.

The collection of tags used for a particular task is known as a **Tagset**.

A part-of-speech tagger, or POS-tagger, processes a sequence of words, and attaches a part of speech tag to each word. Lets first run the below code and see what exactly are we talking about.

POS tagging is a **supervised learning solution** that uses features like the previous word, next word, is first letter capitalized etc. NLTK has a function to get pos tags and it works after tokenization process.

[33]:
```
sentence = "My name is Giovanni"
token = nltk.word_tokenize(sentence)
token
```

[33]:
```
['My', 'name', 'is', 'Giovanni']
```

[34]:
```
nltk.pos_tag(token)
```

[34]:
```
[('My', 'PRP$'), ('name', 'NN'), ('is', 'VBZ'), ('Giovanni', 'NNP')]
```

[8]:
```
# We can get more details about any POS tag using help funciton of NLTK as␣
 ↪follows.
nltk.help.upenn_tagset("PRP$")
```

```
PRP$: pronoun, possessive
    her his mine my our ours their thy your
```

[9]:
```
nltk.help.upenn_tagset("NN$")
```

```
NN: noun, common, singular or mass
    common-carrier cabbage knuckle-duster Casino afghan shed thermostat
    investment slide humour falloff slick wind hyena override subhumanity
    machinist …
```

[10]:
```
nltk.help.upenn_tagset("VBZ$")
```

```
VBZ: verb, present tense, 3rd person singular
    bases reconstructs marks mixes displeases seals carps weaves snatches
    slumps stretches authorizes smolders pictures emerges stockpiles
    seduces fizzes uses bolsters slaps speaks pleads …
```

[11]:
```
nltk.help.upenn_tagset("NNP$")
```

```
NNP: noun, proper, singular
    Motown Venneboerger Czestochwa Ranzer Conchita Trumplane Christos
    Oceanside Escobar Kreisler Sawyer Cougar Yvette Ervin ODI Darryl CTCA
    Shannon A.K.C. Meltex Liverpool …
```

The most popular tag set is Penn Treebank tagset. Most of the already trained taggers for English are trained on this tag set. To view the complete list, follow this link. Uncomment the below code to have the complete upenn tagset used in the nltk package.

[12]:
```
#nltk.download('tagsets')
#print(nltk.help.upenn_tagset())
```

1.5.2 Example 1 - Reading the Newspaper

[13]:
```

```

```
article = "Democrats took exceedingly narrow control of the Senate on Wednesday␣
 ↪after winning both runoff elections in Georgia, granting them control of␣
 ↪Congress and the White House for the first time since 2011. Democrat Jon␣
 ↪Ossoff defeated Republican David Perdue, according to The Associated Press,␣
 ↪making him the youngest member of the U.S. Senate and the first Jewish␣
 ↪senator from Georgia. Earlier Raphael Warnock, a pastor from Atlanta,␣
 ↪defeated GOP Sen. Kelly Loeffler after a bitter campaign. Warnock becomes␣
 ↪the first Black Democrat elected to the Senate from a Southern state.The␣
 ↪Senate will now be split 50-50 between the two parties, giving Vice␣
 ↪President-elect Kamala Harris the tiebreaking vote."
print(article)
```

```
Democrats took exceedingly narrow control of the Senate on Wednesday after
winning both runoff elections in Georgia, granting them control of Congress and
the White House for the first time since 2011. Democrat Jon Ossoff defeated
Republican David Perdue, according to The Associated Press, making him the
youngest member of the U.S. Senate and the first Jewish senator from Georgia.
Earlier Raphael Warnock, a pastor from Atlanta, defeated GOP Sen. Kelly Loeffler
after a bitter campaign. Warnock becomes the first Black Democrat elected to the
Senate from a Southern state.The Senate will now be split 50-50 between the two
parties, giving Vice President-elect Kamala Harris the tiebreaking vote.
```

At a first glance as "human beings", we can see that in the extract there are different names of people and also of organizations; although it is possible that the names are completely new to us, we can recognize without a shadow of a doubt that "Kamala Harris" is the name of a person of female gender, just as "Associated Press" could be the name of an organization or a conference. Now, we want the same result to be produced by our system. Like? Through a function that takes the sentence as input and is able to tokenize the content and extract the grammatical types.

We can use the function ***pos_tag*** of the ***NLTK package***; it is in fact able to take an array of words and assign them a label that determines the type of content that each of them represents at the grammatical level.

[14]:
```
tagged_tokens = nltk.word_tokenize(article)
tagged_tokens = nltk.pos_tag(tagged_tokens)
print(tagged_tokens)
```

```
[('Democrats', 'NNPS'), ('took', 'VBD'), ('exceedingly', 'RB'), ('narrow',
'JJ'), ('control', 'NN'), ('of', 'IN'), ('the', 'DT'), ('Senate', 'NNP'), ('on',
'IN'), ('Wednesday', 'NNP'), ('after', 'IN'), ('winning', 'VBG'), ('both',
'DT'), ('runoff', 'NN'), ('elections', 'NNS'), ('in', 'IN'), ('Georgia', 'NNP'),
(',', ','), ('granting', 'VBG'), ('them', 'PRP'), ('control', 'NN'), ('of',
'IN'), ('Congress', 'NNP'), ('and', 'CC'), ('the', 'DT'), ('White', 'NNP'),
('House', 'NNP'), ('for', 'IN'), ('the', 'DT'), ('first', 'JJ'), ('time', 'NN'),
('since', 'IN'), ('2011', 'CD'), ('.', '.'), ('Democrat', 'NNP'), ('Jon',
'NNP'), ('Ossoff', 'NNP'), ('defeated', 'VBD'), ('Republican', 'NNP'), ('David',
'NNP'), ('Perdue', 'NNP'), (',', ','), ('according', 'VBG'), ('to', 'TO'),
('The', 'DT'), ('Associated', 'NNP'), ('Press', 'NNP'), (',', ','), ('making',
'VBG'), ('him', 'PRP'), ('the', 'DT'), ('youngest', 'JJS'), ('member', 'NN'),
```

```
('of', 'IN'), ('the', 'DT'), ('U.S.', 'NNP'), ('Senate', 'NNP'), ('and', 'CC'),
('the', 'DT'), ('first', 'JJ'), ('Jewish', 'JJ'), ('senator', 'NN'), ('from',
'IN'), ('Georgia', 'NNP'), ('.', '.'), ('Earlier', 'RBR'), ('Raphael', 'NNP'),
('Warnock', 'NNP'), (',', ','), ('a', 'DT'), ('pastor', 'NN'), ('from', 'IN'),
('Atlanta', 'NNP'), (',', ','), ('defeated', 'VBD'), ('GOP', 'NNP'), ('Sen.',
'NNP'), ('Kelly', 'NNP'), ('Loeffler', 'NNP'), ('after', 'IN'), ('a', 'DT'),
('bitter', 'JJ'), ('campaign', 'NN'), ('.', '.'), ('Warnock', 'NNP'),
('becomes', 'VBZ'), ('the', 'DT'), ('first', 'JJ'), ('Black', 'NNP'),
('Democrat', 'NNP'), ('elected', 'VBD'), ('to', 'TO'), ('the', 'DT'), ('Senate',
'NNP'), ('from', 'IN'), ('a', 'DT'), ('Southern', 'NNP'), ('state.The', 'NN'),
('Senate', 'NNP'), ('will', 'MD'), ('now', 'RB'), ('be', 'VB'), ('split',
'VBN'), ('50-50', 'JJ'), ('between', 'IN'), ('the', 'DT'), ('two', 'CD'),
('parties', 'NNS'), (',', ','), ('giving', 'VBG'), ('Vice', 'NNP'), ('President-
elect', 'NNP'), ('Kamala', 'NNP'), ('Harris', 'NNP'), ('the', 'DT'),
('tiebreaking', 'JJ'), ('vote', 'NN'), ('.', '.')]
```

As we have already seen, at a first glance, everything appears very confusing, but in reality each of those acronyms has a very specific meaning.

NLTK POS Tags Examples are as Below:

Abbreviation

Meaning

CC

coordinating conjunction

CD

cardinal digit

DT

determiner

EX

existential there

FW

foreign word

IN

preposition/subordinating conjunction

JJ

This NLTK POS Tag is an adjective (large)

JJR

adjective, comparative (larger)

JJS

adjective, superlative (largest)

LS

list market

MD

modal (could, will)

NN

noun, singular (cat, tree)

NNS

noun plural (desks)

NNP

proper noun, singular (sarah)

NNPS

proper noun, plural (indians or americans)

PDT

predeterminer (all, both, half)

POS

possessive ending (parent ‘s)

PRP

personal pronoun (hers, herself, him,himself)

PRP$

possessive pronoun (her, his, mine, my, our)

RB

adverb (occasionally, swiftly)

RBR

adverb, comparative (greater)

RBS

adverb, superlative (biggest)

RP

particle (about)

TO

infinite marker (to)

UH

interjection (goodbye)

VB

verb (ask)

VBG

verb gerund (judging)

VBD

verb past tense (pleaded)

VBN

verb past participle (reunified)

VBP

verb, present tense not 3rd person singular(wrap)

VBZ

verb, present tense with 3rd person singular (bases)

WDT

wh-determiner (that, what)

WP

wh- pronoun (who)

WRB

wh- adverb (how)

1.5.3 Where we can use POST

Part-of-speech tags describe the characteristic structure of lexical terms within a sentence or text, therefore, we can use them for making assumptions about semantics. Other applications of POS tagging include:

- **Named Entity Recognition and Chunking (see notebook 4-1)**
- Co-reference Resolution
- Speech Recognition

1.5.4 Build your own POS Tagger

It is interesting to understand how to create your own POS tagger using a supervised learning method. This will also allow us to understand the role played by features in identifying individual parts of the text. Finally we will see how the context plays a fundamental role for a correct classification. For this exercise we focus out attenction on suffixes. We'll train a classifier to work out which suffixes are most informative. The training sample will be build from the Brown Corpus which is part of the NLTK package.

The Brown Corpus was the first million-word electronic corpus of English, created in 1961 at Brown University. This corpus contains text from 500 sources, and the sources have been categorized by genre, such as news, editorial, and so on. (for a complete list, see http://icame.uib.no/brown/bcm-los.html).

[12]:
```
from nltk.corpus import brown
```

Let's begin by finding out what the most common suffixes are. For this we are going to use a FreqDist() object from the NLTK package.

A frequency distribution counter records the number of times each outcome of an experiment has occurred. For example, a frequency distribution could be used to record the frequency of each word type in a document. Formally, a frequency distribution can be defined as a function mapping from each sample to the number of times that sample occurred as an outcome.

Frequency distributions are generally constructed by running a number of experiments, and incrementing the count for a sample every time it is an outcome of an experiment. For example, the following code will produce a frequency distribution of any n-gram with $1 \leq n \leq 3$ appearing at the end of a word

[13]:
```
suffix_fdist = nltk.FreqDist()

for word in brown.words():
    word = word.lower()
    suffix_fdist[word[-1:]] += 1   # single character at the end of the word
    suffix_fdist[word[-2:]] += 1   # bi-gram
    suffix_fdist[word[-3:]] += 1   # three-gram
```

[14]:
```
suffix_fdist
```

[14]:
```
FreqDist({'e': 202946, ',': 175002, '.': 152999, 's': 128722, 'd': 105687, 't':
94459, 'he': 92084, 'n': 87889, 'a': 74912, 'of': 72978, …})
```

[15]:
```
common_suffixes = [suffix for (suffix, count) in suffix_fdist.most_common(100)]
print(common_suffixes)
```

```
['e', ',', '.', 's', 'd', 't', 'he', 'n', 'a', 'of', 'the', 'y', 'r', 'to',
'in', 'f', 'o', 'ed', 'nd', 'is', 'on', 'l', 'g', 'and', 'ng', 'er', 'as',
'ing', 'h', 'at', 'es', 'or', 're', 'it', '``', 'an', "''", 'm', ';', 'i', 'ly',
'ion', 'en', 'al', '?', 'nt', 'be', 'hat', 'st', 'his', 'th', 'll', 'le', 'ce',
'by', 'ts', 'me', 've', "'", 'se', 'ut', 'was', 'for', 'ent', 'ch', 'k', 'w',
'ld', '`', 'rs', 'ted', 'ere', 'her', 'ne', 'ns', 'ith', 'ad', 'ry', ')', '(',
'te', '--', 'ay', 'ty', 'ot', 'p', 'nce', "'s", 'ter', 'om', 'ss', ':', 'we',
'are', 'c', 'ers', 'uld', 'had', 'so', 'ey']
```

Next, we'll define a feature extractor function which checks a given word for these suffixes. For this we are going to use the `endswith()` method of `String` class, that returns `True` if the string ends with the specified value, otherwise `False`.

[16]:
```
word = "processing"
word.endswith("ing")
```

[16]:
```
True
```

[17]:
```
def pos_features(word):
    features = {}
    for suffix in common_suffixes:
        features['endswith({})'.format(suffix)] = word.lower().endswith(suffix)
    return features
```

[18]:
```
tagged_words = brown.tagged_words(categories='news')
```

[19]:
```
tagged_words[:10]
```

[19]:
```
[('The', 'AT'),
 ('Fulton', 'NP-TL'),
 ('County', 'NN-TL'),
 ('Grand', 'JJ-TL'),
 ('Jury', 'NN-TL'),
 ('said', 'VBD'),
 ('Friday', 'NR'),
 ('an', 'AT'),
 ('investigation', 'NN'),
 ('of', 'IN')]
```

[20]:
```
featuresets = [(pos_features(n), g) for (n,g) in tagged_words]
```

[21]:
```
size = int(len(featuresets) * 0.1)
train_set, test_set = featuresets[size:], featuresets[:size]
```

Now that we've defined our feature extractor, we can use it to train a new **decision tree** classifier

[22]:
```
classifier = nltk.DecisionTreeClassifier.train(train_set)
nltk.classify.accuracy(classifier, test_set)
```

[22]:
```
0.6270512182993535
```

Exploiting Context By augmenting the feature extraction function, we could modify this part-of-speech tagger to leverage a variety of other word-internal features, such as the length of the word, the number of syllables it contains, or its prefix. However, as long as the feature extractor just looks at the target word, we have no way to add features that depend on the context that the word appears in. But contextual features often provide powerful clues about the correct tag — for example, when tagging the word "fly," knowing that the previous word is "a" will allow us to determine that it is functioning as a noun, not a verb.

In order to accommodate features that depend on a word's context, we must revise the pattern that we used to define our feature extractor. Instead of just passing in the word to be tagged, we

will pass in a complete (untagged) sentence, along with the index of the target word.

[23]:
```
def pos_features_2(sentence, i):
    features = {"suffix(1)": sentence[i][-1:],
                "suffix(2)": sentence[i][-2:],
                "suffix(3)": sentence[i][-3:]}
    if i == 0:
        features["prev-word"] = "<START>"
    else:
        features["prev-word"] = sentence[i-1]
    return features
```

[34]:
```
tagged_sents = brown.tagged_sents(categories='news')
featuresets = []
for tagged_sent in tagged_sents:
    untagged_sent = nltk.tag.untag(tagged_sent)
    for i, (word, tag) in enumerate(tagged_sent):
        featuresets.append( (pos_features_2(untagged_sent, i), tag) )
```

[35]:
```
size = int(len(featuresets) * 0.1)
train_set, test_set = featuresets[size:], featuresets[:size]
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

[36]:
```
nltk.classify.accuracy(classifier, test_set)
```

[36]:
```
0.7891596220785678
```

It is clear that exploiting contextual features improves the performance of our part-of-speech tagger. For example, the classifier learns that a word is likely to be a noun if it comes immediately after the word "large". However, it is unable to learn the generalization that a word is probably a noun if it follows an adjective, because it doesn't have access to the previous word's part-of-speech tag. In general, simple classifiers always treat each input as independent from all other inputs. In many contexts, this makes perfect sense. For example, decisions about whether names tend to be male or female can be made on a case-by-case basis. However, there are often cases, such as part-of-speech tagging, where we are interested in solving classification problems that are closely related to one another. [rinviare al paragrafo di nltk per i sequence classificators]

1.6 spaCy

1.6.1 What is spaCy?

spaCy is another free, open-source library for advanced Natural Language Processing (NLP) in Python.

spaCy is designed specifically for production use and helps you build applications that process and "understand" large volumes of text. It can be used to build information extraction or natural language understanding systems. It is a fully optimized and is widely used in deep learning.

Both libraries (spaCy and NLTK) are excellent and very well managed and documented. Any of them could be a perfect choice, depending on the characteristics of your natural language processing

projects. NLTK is a string processing library: it takes strings as input and returns the resulting string as output; spaCy uses an object-oriented approach. As a result spaCy returns an object in the form of a document, with words or phrases. NLTK provides multiple algorithms for solving a problem, while spaCy only provides the best algorithm for solving a given problem.

We will not go into the details of this library in this introductory course. However we want to point out that the efficiency of the underlying deep learning models often make it the best choice for industrial applications.

We will use some of the trained pipelines in the last part of this seminar where we talk about extracting information from unstructured data.

1.7 References and Credits

Bird S. et al., "*Natural Language Processing with Python*" O'Reilly (2009)

Bengfort B. et al., "*Applied Text Analysis with Python*" O'Reilly (2018)