Introduction to NLP

NLP is Natural Language Processing. It is a field of AI that deals with how to program a computer to process natural language data. NLP helps computers to perform the following tasks:

- Automatic Texts Summarization
- Language to Language Translation
- Named Entity Recognition
- Relationship Extraction
- Sentiment Analysis
- Speech Recognition
- Text Classification

NLP Tasks

To do NLP we need to execute three main tasks: lexical analysis, syntactic analysis and semantic analysis:

- **Lexical Analysis -** these analysis deal with text structure. The text gets broken down to paragraphs, sentences, and single words.
- **Syntactic Analysis** (also called parsing) - it involves analyzing sentences for grammar and determining how words relate to each other.
- **Semantic Analysis** deals with extracting the meanings from text. It also maps syntactic structures to check whether they make sense.

NLP Tools

There are many NLP libraries, here are most [popular and known:

- **Apache OpenNLP**- Machine Learning toolkit; allows for tokenizers, sentence segmentation, part-of-speech tagging, chunking, parsing, named entity extraction, and more.
- **MALLET-** Java library for latent Dirichlet allocation, clustering, topic modeling, information extraction, document classification, and more.
- **NLTK - Natural Language ToolKit –** Python based toolkit with modules for processing text, classifying, tokenizing, stemming, parsing, tagging, and more.
- **Stanford NLP Suite-** Tools for part-of-speech tagging, named entity recognizer, sentiment analysis, conference resolution system, and more.
- **spaCy –** comprehensive library of NLP tools with convenient Python API. See the next section.

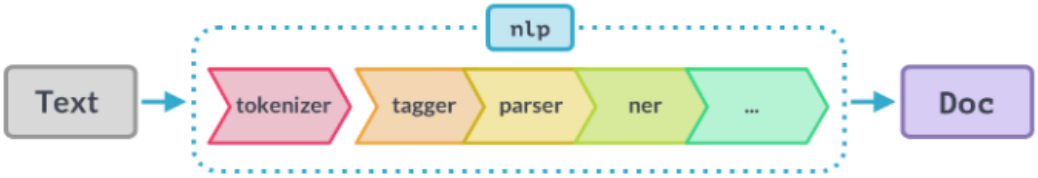
NLP with spaCy

spaCy is an industry standard NLP library with abilities to solve most NLP tasks with great speed, accuracy, and performance. spaCy supports more than 50 languages and new languages are being added. For English spaCy has three *language models* (pipelines) that differ in sizes and functionality:

- **en_core_web_sm:** English model, 11 MB size
- **en_core_web_md:** English model, with GloVe vectors trained on Common Crawl, 90 MB size
- **en_core_web_lg:** English model, trained on OntoNotes, with GloVe vectors trained on Common Crawl, 789 MB size

In this tutorial, we will use the medium model – en_core_web_md:
nlp = spacy.load('en_web_md')

It is a convention to name loaded language model object as *nlp.* This object is is a pipeline of several text pre-processing blocks turning the input text into the *doc* object: tokenizer, tagger, parser, ner, and so on…



When we work with spaCy the first step is to pass a text to *nlp:*

```
# load a  model
nlp = spacy.load('en_web_md')

# define a text
txt = 'This is just a meaningless text'

# create the doc object
doc = nlp(txt)
print(f"text: {doc.text}")
[OUT]:
text: This is just a meaningless text
```

or

```
doc.text('I love London very much!')
print(len(doc))
[OUT]:
6 # including '!'
```

The name *doc* is also a convention for an object for results of a text processed by *nlp*. The *doc* object contains tokens – results of splitting sentences into words and punctuation. A single token can be a word, a punctuation or a noun chunk, etc. The *doc* object is an iterator:

```
for token in doc[:3]:
    print(token)

[OUT]:
I
love
London
```

spaCy NER - Named Entity Recognition

One of the most common tasks in NLP is recognizing named entities. Named Entities are the words or groups of words that represent information about common things: persons, locations, brands, organizations, etc. These entities have specific names. The following sentence: “Donald Trump meet the chairman of Telsa in New York city” contains the following named entities:

- “Donald Trump” – a person
- “Tesla” – an organization
- “New York city” – a location

spaCy makes NER very easy: get the *doc* object from a text and extract it’s attribute *ents.* See the example spacy-ner.py:

```
txt = """ Apple Inc. announced on Monday that it will open a new office in San Francisco next month. CEO Tim Cook confirmed that the company plans to invest $2 billion in research and development. The announcement comes after a record-breaking quarter where Apple reported revenues of $81.43 billion, making it the highest earning tech company of 2023. Meanwhile, Microsoft Corporation, headquartered in Redmond, Washington, continues to expand its cloud services globally.
"""

nlp = spacy.load("en_core_web_md")

doc = nlp(txt)

for ent in doc.ents:
    print(f"{ent.text:<20}{ent.label_:<20}")
```

```
[OUT]:
Apple Inc.          ORG
Monday              DATE
San Francisco       GPE
next month          DATE
Tim Cook            PERSON
$2 billion          MONEY
Apple               ORG
$81.43 billion      MONEY
2023                CARDINAL
Microsoft Corporation   ORG
Redmond             GPE
Washington          GPE
```

Tim Cook is recognized as a PERSON, while San-Francisco, Redmond and Washington are GPEs - Geo-Political entities. We can use the spacy.explain() function to get the meaning of spaCy NER labels:

```
print(spacy.explain("GPE"))
[OUT]:
'Countries, cities, states'

print(spacy.explain("CARDINAL"))
[OUT]:
Numerals that do not fall under another type
```

spaCy Phrase Matching

Rule-based matching is another tool in spaCy's toolkit, which helps to find words and phrases in the text using user-defined rules. The spaCy matcher uses the text patterns and lexical properties of the word, such as POS tags, dependency tags, lemma, etc. Let's take a phrase and see what can we do with spaCy matcher

```
import spacy
nlp = spacy.load('en_core_web_md')

# Import spaCy Matcher
from spacy.matcher import Matcher
```

```
# Initialize the matcher with the spaCy vocabulary
matcher = Matcher(nlp.vocab)

doc = nlp("Smart people drink green tea at the morning")

# Define rule
pattern = [{'TEXT': 'green'}, {'TEXT': 'tea'}]

# Add rule, heed that patter must be passed as a list
matcher.add('rule_tea', [pattern])

matches = matcher(doc)
print(matches)
[OUT]:
matches: [(2748636944490998126, 3, 5)]

# Let's make it usable - extract matched text
for match_id, start, end in matches:
   # Get the matched span
   matched_span = doc[start:end]
   print(matched_span.text)
[OUT]:
green tea
```

The found matches has three elements: the first element, '*2748636944490998126*', is the match ID, the second and third elements are the positions of the matched tokens. Let's see another use case of the spaCy matcher. Consider the two sentences below:

1. I can read this book
2. I will book my ticket

Let's check whether a sentence contains the word '*book*' in it or not. It seems easy doesn't it? But there is a restriction: we have to find the word '*book*' only if it has been used as a noun. In the first sentence above, '*book*' has been used as a noun and in the second sentence, it has been used as a verb. Omni powerful regular

expressions would not help us much in this case. But the spaCy matcher that uses text token's POS property, should be able to extract the pattern from the first sentence only:

```
t1 = 'I am going read this book'
t2 = 'I was going to book that room'

pattern = [{'TEXT': 'book', 'POS': 'NOUN'}]
matcher.add('book_rule', [pattern])

for t in [t1, t2]:
   d = nlp(t)
   m = matcher(d)
   if len(m) > 0:
      for match_id, start, end in m:
         # Get the matched span
         matched_span = d[start:end]
         print(f"the match is '{matched_span.text}' from the text '{t}'")
[OUT]:
the match is 'book' from the text 'I read this book'
```

The matcher has found the pattern in the first text!

Word Vectors and Semantic Similarity

One more day-to-day NLP use case is finding semantic similarity. Similarity is the measurement of a difference between two vectors that represent the texts. In simple terms, similarity is the measure of how different or alike two text objects are. It is measured in the range 0 to 1. This score in the range of [0, 1] is called the similarity score. The less score is the more similar objects are.

Text Similarity

Look at the following sentences: "the glass is empty" and "there is nothing in the glass". For a human, it is obvious that these two mean the same thing even being written in different ways. Can we make a machine to understand that? Obviously, we need somehow translate these sentences into set of numbers and this would let us calculate a similarity. We will consider some effective methods of machine

text representation (text embedding) and after that see how to calculate similarity. This translation is called embedding.

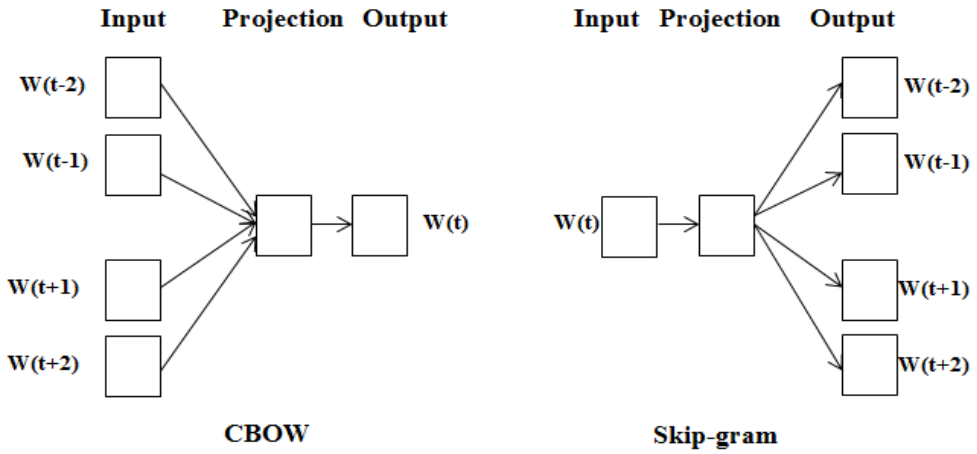
One-Hot Encoding

The simple way to represent words is the *one-hot encoding* method. We create a vector with the size of the total number of unique words in the whole text. Each unique word has a unique feature and will be represented by 1s with 0s everywhere else:

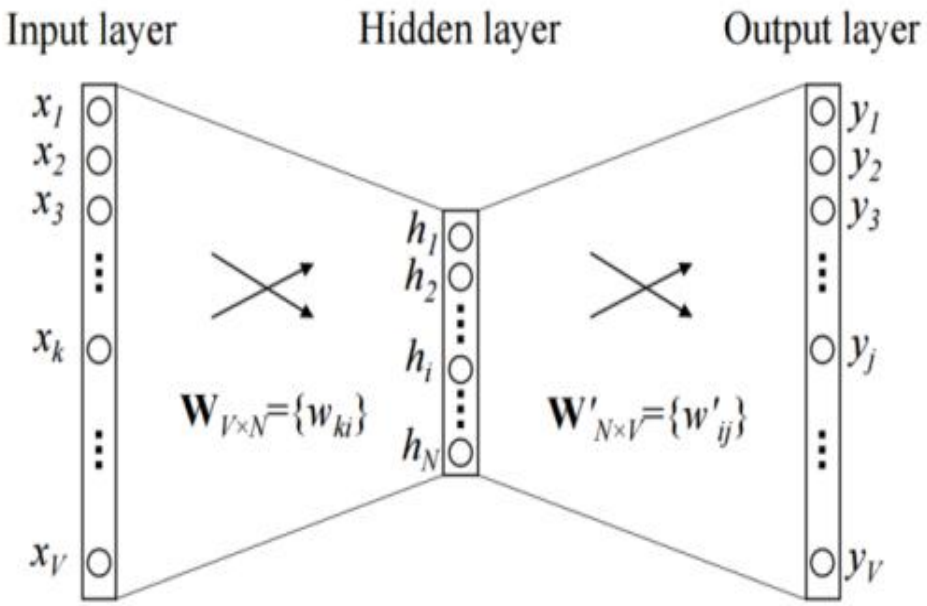
Atlanta =	**[1, 0, 0, 0, 0 ..0]**
Chicago =	**[0, 1, 0, 0, 0 ..0]**
Los-Angeles =	**[0, 0, 1, 0, 0 ..0]**
New York =	**[0, 0, 0, 1, 0 ..0]**
Orlando =	**[0, 0, 0, 0, 1 ..0]**

Obviously, this is a very "generous" approach to memory but that is where everything began. Another way is called the "bag of words" representation: each word is represented by its count instead of 1. Both approaches create huge, sparse vectors and contain no information about semantics. Let's skip historical research and cut to the chase, considering word2vec embedding.

Word2vec is a method for forming embeddings based upon a pre-trained two-layer neural network. It uses some big enough set of texts which serves as an input and outputs a set of feature vectors that represent words in that set. The set is called a corpus. Below is an illustration of how it is prepared.



Continuous bag-of-words takes each word as the input and tries to predict the word corresponding to the context. A context means the surrounding words.



For the details of this method, see the article on arxiv.org

spaCy calculates semantic similarity using word2vec embedding. This embedding is available in the medium and large models:

```
doc1 = nlp(t1)
doc2 = nlp(t2)
print(f"similarity: {doc1.similarity(doc2)}")
[OUT]:
similarity: 0.8309948794221409
```

All doc objects have this similarity() method. The similarity is calculated as a vector distance between using word2vec vectors, representing words in a text.

Check the folder **'work_with_spacy'** for more examples.