Natural Language Processing

Natural languages: are the normal languages we use to communicate in everyday life, e.g. English, Chinese... etc., which -unlike languages like Python- have been naturally developed through time.

So, when we talk about Natural Language Processing (NLP), we mean how computers process these languages; or simply put, how to deal with text data.

Some of the most important examples of NLP are:

- **Sentiment Analysis**: deciding whether a statement carries a positive or a negative emotion.
- **Topic Modeling:** classifying different texts based on their topic.
- **Text Generation:** creating new texts that are similar to previously written ones.

To solve a problem using NLP, we first need to start by gathering raw data about the subject and putting it into one of the standard formats that are easily analyzed and processed by computers. **Different types of analysis require different data formats, some of which are:**

- **Corpus**

  A corpus is a collection of texts, in which we put our data in an organized table. We usually create this corpus using Pandas library with the DataFrame function.

Comedian	Transcript
Ali Wong	Hi. Hello! Welcome! Thank you! Thank you for coming. Hello! Hello.
Dave Chappelle	Thank you! Thank you very much! Thank you all. Oh, wow. That was exciting.
John Mulaney	All right, Petunia. Wish me luck out there. You will die on August 7th, 2037.
...	...

- **Document-term Matrix**

  For us to put the data in this form, we need to clean it, tokenize it, then put it into a matrix.

  For example:

  All right, Petunia. Wish me luck out there. You will die on August 7th, 2037.

  - **Cleaning**: by removing punctuation, numbers and write all letters in lowercase. (You can do this by using Regular expression (re) operations in Python)

    all right petunia wish me luck out there you will die on august

  - **Tokenization**: is to split text into smaller pieces. The most common token size is a word. It can also be a sentence.

    Now that every word is its own token, you can filter out words that have very little meaning (the, out, all…etc.) – these are called **stop words.**

After that, we end up with the essential words out of each sentence, each represented as its own token. This representation of text is called a **bag of words model.** It is a simple format that ignores order.

right *petunia* *wish* *luck* *die* *august*

- **Matrix**: we put the final words into a matrix. And the reason we need to do that is because we need to store this info for multiple documents.

Comedian	right	petunia	wish	me	luck	die	august	hello	thank	welcome	wow	exciting
John Mulaney	1	1	1	1	1	1	1	0	0	0	0	0
Ali Wong	0	0	0	0	0	0	0	3	2	0	0	0
Dave Chappelle	0	0	0	0	0	0	0	0	3	0	1	1
...												

We can easily make this matrix using CounterVectorizer() function in the scikit-learn library.

```
# We are going to create a document-term matrix using CountVectorizer, and exclude common English stop words
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer(stop_words='english')
data_cv = cv.fit_transform(data_clean.transcript)
data_dtm = pd.DataFrame(data_cv.toarray(), columns=cv.get_feature_names())
data_dtm.index = data_clean.index
data_dtm
```

Back to the NLP techniques:

- **Sentiment Analysis**

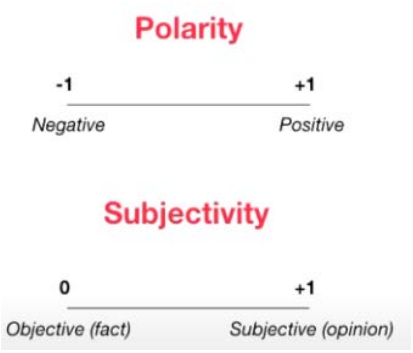
  For Sentiment Analysis, we should input our data as a corpus. The reason we do not use document-term matrix (bag of words format) here is because order matters, i.e. “great” = positive. “not great” = negative.

  We then use the python library ***TextBlob*** (built on nltk) to give us a sentiment score (how positive/negative) and a subjectivity score (how opinionated) for a sentence. It does that by finding all the words and phrases that it can assign a polarity and subjectivity to, and averages all of them together.

```
# The code for TextBlob's sentiment analysis

from textblob import TextBlob

TextBlob("I love PyOhio").sentiment
```



#Output:

```
Sentiment(polarity=0.5, subjectivity=0.6)
```

Example:

```
# Create quick lambda functions to find the polarity and subjectivity of each routine
# Terminal / Anaconda Navigator: conda install -c conda-forge textblob
from textblob import TextBlob

pol = lambda x: TextBlob(x).sentiment.polarity
sub = lambda x: TextBlob(x).sentiment.subjectivity

data['polarity'] = data['transcript'].apply(pol)
data['subjectivity'] = data['transcript'].apply(sub)
```

- **Topic Modeling**
  For Topic Modeling, we should input our data as a document-term matrix. As each topic will consist of a set of a words where order does not matter, so we are going to use the bag of words format.

  We then use ***gensim***, which is a Python toolkit for topic modeling. Gensim has different techniques to do topic modeling, the most popular of which is called **Latent Dirichlet Allocation** (LDA). For this to work, we need to specify the number of topics we think there are in our corpus, and how many iterations we want Gensim to go through our corpus for.

  This will output the top words in each topic. It is your job as a human to interpret this and see if the results make sense.

  Example:

```
# Create the gensim corpus
corpusna = matutils.Sparse2Corpus(scipy.sparse.csr_matrix(data_dtmna.transpose()))

# Create the vocabulary dictionary
id2wordna = dict((v, k) for k, v in cvna.vocabulary_.items())
```

```
# Let's start with 2 topics
ldana = models.LdaModel(corpus=corpusna, num_topics=2, id2word=id2wordna, passes=10)
ldana.print_topics()
```

```
# Let's try 3 topics
ldana = models.LdaModel(corpus=corpusna, num_topics=3, id2word=id2wordna, passes=10)
ldana.print_topics()
```

```
# Let's try 4 topics
ldana = models.LdaModel(corpus=corpusna, num_topics=4, id2word=id2wordna, passes=10)
ldana.print_topics()
```

- **Text Generation**
  For Text Generation, we should input our data as a corpus. As we want to preserve the order of the text, including the punctuation.

  We can perform this using ***Markov Chains***, which are a way of representing how systems change over time. The main concept behind Markov chains is that they are memoryless, meaning that the next state of a process only depends on the previous state, i.e. It takes every word as a state, and it looks at the next word thinking how likely it is going to be this word based on word number 1.

  To apply this, we create a dictionary for a corpus where the keys are the current state and the values are the options for the next state. And we write a function to randomly generate next terms.

  Example:

Build a Markov Chain Function

We are going to build a simple Markov chain function that creates a dictionary:

- The keys should be all of the words in the corpus
- The values should be a list of the words that follow the keys

```
from collections import defaultdict

def markov_chain(text):
    '''The input is a string of text and the output will be a dictionary with each word as
       a key and each value as the list of words that come after the key in the text.'''

    # Tokenize the text by word, though including punctuation
    words = text.split(' ')

    # Initialize a default dictionary to hold all of the words and next words
    m_dict = defaultdict(list)

    # Create a zipped list of all of the word pairs and put them in word: list of next words format
    for current_word, next_word in zip(words[0:-1], words[1:]):
        m_dict[current_word].append(next_word)

    # Convert the default dict back into a dictionary
    m_dict = dict(m_dict)
    return m_dict
```

```
# Create the dictionary for Ali's routine, take a look at it
ali_dict = markov_chain(ali_text)
```

Create a Text Generator

We're going to create a function that generates sentences. It will take two things as inputs:

- The dictionary you just created
- The number of words you want generated

Here are some examples of generated sentences:

> 'Shape right turn– I also takes so that she's got women all know that snail-trail.'

> 'Optimum level of early retirement, and be sure all the following Tuesday… because it's too.'

```
import random

def generate_sentence(chain, count=15):
    '''Input a dictionary in the format of key = current word, value = list of next words
       along with the number of words you would like to see in your generated sentence.'''

    # Capitalize the first word
    word1 = random.choice(list(chain.keys()))
    sentence = word1.capitalize()

    # Generate the second word from the value list. Set the new word as the first word. Repeat.
    for i in range(count-1):
        word2 = random.choice(chain[word1])
        word1 = word2
        sentence += ' ' + word2

    # End it with a period
    sentence += '.'
    return(sentence)
```

```
generate_sentence(ali_dict)
```