**Natural Language Processing: NLP**

This course walks you through various Natural Language Processing techniques using Python and NLTK.

**By the end of this course, you will be able to:**

* Perform various cleaning and pre-processing operations on textual data using NLTK
* Work with the various elements of text data such as Tokens, N-grams, Lemma and Frequency Distributions
* Annotate text with part of speech(POS) using in-built and custom POS taggers
* Work with various lexical resources provided with NLTK to preprocess your data
* Build applications such as Spam detector, Topic Modeler, Chatbot, Sentiment Analyzer using NLP and Machine Learning techniques

**WHY NLP?**

Natural Language Processing has always been a key tenet of Artificial Intelligence (AI). With increase in adoption of AI, systems to automate sophisticated tasks are being built. Some of these examples are described below.

# Diagnosing rare form of cancer

At the University of Tokyo's Institute of Medical Science, doctors used artificial intelligence to successfully diagnose a rare type of leukemia. The patient was a female in her mid 60s and was initially diagnosed with acute myeloid leukemia. She underwent chemotherapy which attacked her cancer cells. However, her recovery was unusually slow which puzzled the doctors.

The doctors then used an AI system which cross-referenced the patient’s genetic data with tens of millions of oncology papers and diagnosed the cancer as rare secondary leukemia caused by myelodysplastic syndromes. The machine was able to do this in 10 mins, while it would have taken human scientists about 2 weeks to do the same.

# Settling an insurance claim within 3 seconds

Lemonade built a bot - AI Jim, which interacts with the claimant in real time and understands the nature and severity of the claim. It assesses the likelihood of the claim being fraudulent, and even nudges people to be more honest by incorporating years of behavioural economics research into its conversations. This system settled an insurance claim within 3 seconds by running 18 algorithms.

# Automating customer service tasks

KLM Royal Dutch Airlines fly to 163 destination worldwide, operate 200+ aircrafts, and annually ferry 30M + passengers.

The airline wanted to create “a new entry point” for customers – one that provide opportunities for conversational interactions using voice and text.  They created BB (Blue bot) –  a chat bot that helps customers manage flight bookings though conversational interfaces.

In the first month of launch, the around 1.7 million messages have been exchanged between that bot and 500,000 people.

In all of these examples, the systems are meant to understand the natural language used by human beings. To elaborate:

* The system that diagnosed cancer had to go through millions of text documents written by humans in english/other languages in varied styles and a vast vocabulary
* Blue Bot not only interprets the queries of a customer which is typically in the natural language, but also generate appropriate responses in natural language.
* Having seen examples of where AI systems need to understand and process natural language, let us now demystify what natural language and natural language processing means.

# Natural language

In neuropsychology, linguistics, and the philosophy of language, a natural language or ordinary language is any language that has evolved naturally in humans through use and repetition without conscious planning or premeditation[1].

In contrast to artificial languages like Python, C, Java, etc. natural languages like English, Portuguese, French, etc. have evolved over time and use and its difficult to express them in strict formal rules.

# Natural Language Processing

* Natural Language processing(NLP) is an area of computer science and artificial intelligence concerned with the interactions between computers and human (natural) languages, in particular how to program computers to process and analyze large amounts of natural language data[2].
* Understanding natural language is informally considered as AI-hard/AI-complete[3] by several researches.

# Why is it hard?

* NLP is hard because natural languages evolved without a standard rule / logic. They were developed in response to evolution of human brain: in its ability to understand signs, voice and memory. With NLP, we are now trying to “discover rules” for a something (language) that evolved without rules.
* [1] Natural Language article on Wikipedia
* [2] Natural Language Processing article on Wikipedia
* [3] AI Complete article on Wikipedia

Let us now try to understand why NLP is considered hard using a few examples.

1. "There was not a single man at the party"
   * Does it mean that there were no men at the party? or
   * Does it mean that there was no one at the party?
   * Here does man refer to the gender "man" or "mankind"?
2. "The chicken is ready to eat"

* Does this mean that the bird (chicken) is ready to feed on some grains? or
* Does it mean that the meat is cooked well and is ready to be eaten by a human?

3. "Google is a great company." and "Google this word and find its meaning."

* Google is being used as a noun in the first statement and as a verb in the second.

4. The man saw a girl with a telescope.

* Did the man use a telescope to see the girl? or
* Did the man see a girl who was holding a telescope?

Natural language is full of **ambiguities.**Ambiguity can be referred to as the ability of having more than one meaning or being understood in more than one way. This is a primary reason why NLP is considered hard. Another reason why NLP is hard is because it deals with the extraction of knowledge from unstructured data.

## **Understanding Textual Data**

# Elements of Text

Let us now understand various elements of textual data and see how we can extract these using the NLTK library.

In the subsequent pages, we shall discuss the following elements of text:

* Hierarchy of Text
* Tokens
* Vocabulary
* Punctuation
* Part of speech
* Root of a word
* Base of a word
* Stop words

# Hierarchy of Text

Text is a collection of meaningful sentences. Each sentence in turn comprises many words.

Consider the text "India is a republic nation. We are proud Indians". This text contains 2 sentences and 9 words.

1. import nltk
2. from nltk import \*
3. sent = "India is a republic nation. We are proud Indians"

Texts are represented in Python in the form lists:  ['India is a republic nation. We are proud Indians']

 We can use slicing, indexing and we can also find length of these lists.

1. print(len(sent)) *#Prints the number of characters*
2. print(sent[0:5]) *#Prints 'India'*
3. print(sent[11:19]) *#Prints 'republic'*

# Tokens

A meaningful unit of text is a token. Words are usually considered as tokens in NLP. The process of breaking a text based on the token is called tokenization.

1. print(nltk.word\_tokenize(sent))
2. *#Prints list of words ['India', 'is', 'a', 'republic', 'nation', '.', 'We', 'are', 'proud', 'Indians']*

# Vocabulary

The vocabulary of a text is the set of all unique tokens used in it.

1. tokens = nltk.word\_tokenize(sent)
2. vocab = sorted(set(tokens))
3. print(vocab) *#Prints ['.', 'India', 'Indians', 'We', 'a', 'are', 'country', 'is', 'proud', 'republic']*

# Punctuation

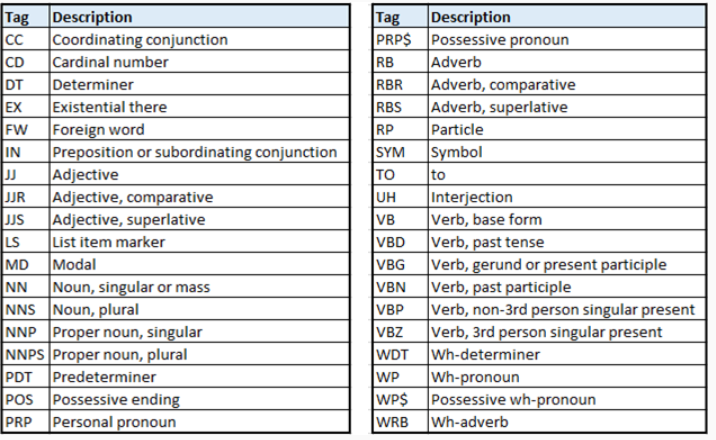
Punctuation refers to symbols used  to separate sentences and their elements and to clarify meaning.

1. from string import punctuation
2. vocab\_wo\_punct=[]
3. for i in vocab:
4. if i not in punctuation:
5. vocab\_wo\_punct.append(i)
6. print(vocab\_wo\_punct) *#Prints ['India', 'Indians', 'We', 'a', 'are', 'country', 'is', 'proud', 'republic']*

# Part of Speech

Part of speech(POS) refers to the category to which a word is assigned based on its function. You may recall that the English language has 8 parts of speech - noun, verb, adjective, adverb, pronoun, determiner, preposition, conjunction, and interjection.

Different POS taggers are available that classify words into POS. A popular one is the Penn treebank, which has the following parts of speech.



The below code demonstrates POS tagging on text

1. from nltk import pos\_tag
2. pos\_list = pos\_tag(vocab\_wo\_punct)
3. print(pos\_list)
4. """ Prints [('India', 'NNP'), ('Indians', 'NNPS'), ('We', 'PRP'), ('a', 'DT'),
5. ('are', 'VBP'), ('country', 'NN'), ('is', 'VBZ'), ('proud', 'JJ'),
6. ('republic', 'JJ'), ('India', 'NNP')] """

# Root of a word - Stemming

Stemming is a technique used to find the root form of a word. In the root form a word is devoid of any affixes (suffixes and prefixes)

1. from nltk.stem.snowball import SnowballStemmer
2. stemObj = SnowballStemmer("english")
3. stemObj.stem("Studying") *#Prints 'studi'*
4. stemmed\_vocab=[]
5. stemObj = SnowballStemmer("english")
6. for i in vocab\_no\_punct:
7. stemmed\_vocab.append(stemObj.stem(i))
8. print(stemmed\_vocab) *#Prints ['india', 'indian', 'we', 'a', 'are', 'countri', 'is', 'proud', 'republ']*

# Base of a word - Lemmatization

Lemmatization removes inflection and reduces the word to its base form

1. from nltk.stem.wordnet import WordNetLemmatizer
2. lemmaObj =  WordNetLemmatizer()
3. lemmaObj.lemmatize("went",pos='v')  *#Prints 'go'*

# Stop words

Stop words are typically the most commonly occurring words in text like 'the', 'and', 'is', etc.

NLTK provides a pre-defined set of stopwords for English, as shown

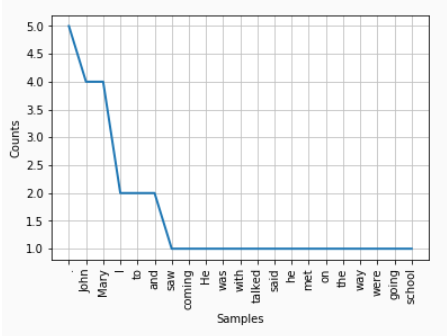
1. from nltk.corpus import stopwords
2. wo\_stop\_words = []
3. stop\_words\_set = set(stopwords.words("english"))
4. for i in vocab\_no\_punct:
5. if i not in stop\_words\_set:
6. wo\_stop\_words.append(i)
7. print(wo\_stop\_words) *#Prints ['India', 'Indians', 'We', 'country', 'proud', 'republic']*

**Distributions and n-grams**

# Frequency Distribution

Frequency distribution helps understand which words are commonly used and which are not. These can help refine stop words in a given text.

1. text="I saw John coming. He was with Mary. I talked to John and Mary. \
2. John said he met Mary on the way. John and Mary were going to school."
3. print(nltk.FreqDist(nltk.word\_tokenize(text)))
4. *#Prints FreqDist({'.': 5, 'Mary': 4, 'John': 4, 'I': 2, 'to': 2, 'and': 2, 'the': 1, 'was': 1, 'were': 1, 'school': 1, ...})*
5. nltk.FreqDist(text.split()).plot()



# Conditional Frequency Distribution

Conditional Frequency Distributions can help in identifying difference in usage of words in different text. For example, commonly used words in books/articles on "romance" genre could be different from words used in books/articles of "news" genre. An example with nltk library to get the conditional frequency distribution of words.

Here we use Brown corpus.

1. cfd = nltk.ConditionalFreqDist(
2. (a, b)
3. for a in brown.categories()
4. for b in brown.words(categories=genre))
5. genres\_list = ['romance','news','science\_fiction', 'humor','religion','hobbies']
6. modals\_list = ['may','could', 'can', 'must', 'will', 'might']
7. cfd.tabulate(conditions=genres\_list, samples=modals\_list)

Note: Currently we are using the default corpora provided by NLTK, towards the end of this course we can run a similar CFD on a corpora created by us.

# N-grams

N-gram is a contiguous sequence of n items from a given sample of text or speech. NLTK provides with methods to extract n-grams from text

1. from nltk import ngrams
2. *#use 2 for bigrams*
3. bigrams = ngrams(vocab\_no\_punct,2)
4. print(list(bigrams))
5. *#Prints [('India', 'Indians'), ('Indians', 'We'), ('We', 'a'), ('a', 'are'), ('are', 'country'),*
6. ('country', 'is'), ('is', 'proud'), ('proud', 'republic')]
7. *#use 3 for trigrams*
8. trigrams = ngrams(vocab\_no\_punct,3)
9. print(list(trigrams))
10. [('India', 'Indians', 'We'), ('Indians', 'We', 'a'), ('We', 'a', 'are'), ('a', 'are', 'country'),
11. ('are', 'country', 'is'), ('country', 'is', 'proud'), ('is', 'proud', 'republic')]

# Operations on Text

So far we you have seen various basic concepts of textual data. Let us now see how these are used together to pre-process text.

Pre-processing text typically involves at least some of the following operations:

* Trimming the text of unwanted spaces that occur at the beginning and end of the text
* Convert the text into either lower or uppercase
* Tokenize the text and determine its vocabulary
* Remove stop words from the given text
* Remove punctuation
* Normalize the text using stemming and/or lemmatization
* Create n-grams from text

Let us now use a [sample text file](http://10.177.157.28:3006/web-hosted/lex_auth_0125952437187788801499/assets/nlp_wikipedia_sample.txt) - Few paragraphs from the Wikipedia article on NLP and perform the different preprocessing operations.

Here is the initial setup of the code.

1. import nltk
2. import string
3. from nltk import word\_tokenize
4. from nltk import wordpunct\_tokenize
5. from nltk.corpus import stopwords
6. from string import punctuation
7. from nltk.stem.snowball import SnowballStemmer
8. from nltk.stem.wordnet import WordNetLemmatizer
9. from nltk import pos\_tag
10. *#read the file*
11. *#Note: use the correct path of the file depending on your environment*
12. file = open("nlp\_wikipedia\_sample.txt",'r')
13. text = ''
14. for i in file.readlines():
15. text += i
16. *# print(text)*

**Step1: Trimming the text of unwanted spaces**

Sometimes the text that we want to process and analyse may contain a few extra spaces at the beginning and end. We use the strip() method of Python to get rid of these unwanted spaces

1. *#remove trailing spaces*
2. trimmed\_text = text.strip()
3. *# print(trimmed\_text)*

**Step 2: Convert the text into either lower or uppercase**

In order to analyse text we often need to normalize it. Normalizing text includes various steps, one of which is bringing the text to a standard case - either lower or upper. Normalizing depends on the application we want to build. If we are trying to understand the sentiment of a given tweet, we may not convert the tweet to lowercase because uppercase is often used to emphasize sentiment i.e. "AWESOME" and "awesome" have different levels of emphasis.

To convert the the text into lower/upper case, we use the lower() and upper() methods of Python

1. converted\_text = trimmed\_text.lower()
2. *# print(converted\_text)*

**Step 3: Tokenize the text and determine its vocabulary**

As seen earlier, tokenization splits a sentence into its constituent words. NLTK provides different kinds of tokenizers based on different kind of text that you may encounter. For example, for twitter data NLTK provides the casual TweetTokenizer (nltk.tokenize.casual.TweetTokenizer) which handles commonly occurring smileys, hashtags, etc.

Visit nltk tokenize website to know more about tokenizers offered by nltk.

The below code shows the usage of word tokenizer and word punct tokenizer to get the tokens from the sample text

1. *#Tokenization using word tokenizer*
2. tokenized\_list = word\_tokenize(converted\_text)
3. *# print(tokenized\_list)*
4. *#Tokenization using word punct tokenizer*
5. punct\_tokenized\_list = wordpunct\_tokenize(converted\_text)
6. *# print(punct\_tokenized\_list)*
8. *#get vocabulary*
9. vocab\_set = set(tokenized\_list)
10. *# print(vocab\_list)*

**Step 4: Remove stop words from the text**

Removing stop words from text helps enrich the amount of useful information in the text. As seen earlier, the NLTK library provides pre-defined stop words in different natural languages. However, sometimes this may not be sufficient, we may need to add new words to the stop word list considering the domain of the text. We can determine which words are stop words by analyzing the results of word frequency and conditional frequency distributions.

The below code removes common English stop words from our sample text

1. *#remove stop words*
2. set\_wo\_stopwords = vocab\_set - set(stopwords.words("english"))
3. *# print(set\_wo\_stopwords)*

**Step 5: Remove punctuation**

Removing punctuation from text is again a normalization technique that is contextual to the application. Going back to the example of sentiment analysis of tweets, we may choose to not remove the punctuation, because the presence of some of the characters like '!', ':-)' , etc. convey a strong sentiment.

The below code demonstrates the removal of punctuation from the sample data

1. *#remove punctuation*
2. set\_wo\_punctuation = set\_wo\_stopwords - set(punctuation)
3. *# print(set\_wo\_punctuation)*

**Step 6: Normalize the text using stemming and/or lemmatization**

Stemming and Lemmatization result in the root/base form of words in text. This technique is particularly useful when building applications like a search engine.

NLTK provides different stemmers that can be used depending on the language and desired level of accuracy. You can read more about the available stemmers on nltk website under stem.

The below code demonstrates stemming using the SnowballStemmer

1. *#stemming*
2. stemmed\_list= []
3. stemObj = SnowballStemmer("english")
4. for i in set\_wo\_punctuation:
5. stemmed\_list.append(stemObj.stem(i))
6. *# print(stemmed\_list)*

WordnetLemmatizer is one the most commonly used lemmatizer for the English language. It is used to arrive at the base form of the word.

The below code demonstrates the usage of the WordnetLemmatizer on our sample data.

1. *#parts of speech tagging*
2. pos\_tag\_list = pos\_tag(set\_wo\_punctuation)
3. *# print(pos\_tag\_list)*
4. *#for getting parts of speech*
5. def parts\_of\_speech(pos):
6. if pos.startswith("N"):
7. return wordnet.NOUN
8. elif pos.startswith("J"):
9. return wordnet.ADJ
10. elif pos.startswith("V"):
11. return wordnet.VERB
12. elif pos.startswith("R"):
13. return wordnet.ADV
14. elif pos.startswith("S"):
15. return wordnet.ADJ\_SAT
16. else:
17. return ''
18. *#lemmatization*
19. lemma\_list = []
20. lemmaObj =  WordNetLemmatizer()
21. for word,pos in pos\_tag\_list:
22. get\_pos = parts\_of\_speech(pos)
23. if get\_pos != '':
24. lemma\_list.append(lemmaObj.lemmatize(word, pos = get\_pos))
25. else:
26. lemma\_list.append(word)
27. *# print(lemma\_list)*

**Step 7: Create n-grams from text**

In many applications of text analysis, tokens are not treated individually but based on how they occur together. For example, systems that automatically predict the word that you are going to type next need to look at tokens that are commonly used with one another.

The below code demonstrates how bi-grams (n-grams where n=2) can be created on our sample text.

1. *#bigrams*
2. bigrams = ngrams(set\_wo\_punctuation,2)
3. *# print(list(bigrams))*

# Regular Expressions

R.E is the scalpel in the hand of an NLP practitioner

Having seen the different operations we can do on text using nltk lets see another special module called Regular Expressions usually called regex.

Regex provide us with an easy way to do various text processing methods like search, sub, findall.

Let us now see some simple steps for using regular expressions.

Keeping in mind we will need to import the module named "re".

1. import re

The syntax for doing any regex method would follow re."methodname"(r"some expression related to process",input\_string). The "some expression related to process" is called the raw string which has different types of expressions to go through and process the input\_string.

Let us now see some of these expressions and how to use them

| **Expression** | **How it works** |
| --- | --- |
| A-Z | Searches in the input string for characters that exist between A and Z |
| a-z | Searches in the input string for characters that exist between a and z |
| ? | Number of occurrences of the character preceding the ? can be 0 or 1 |
| . | Denotes any character either alphabet or number or special characters |
| + | Number of occurrences of the character preceding the + can be at least 1 or more |
| w | Denotes a set of alphanumeric characters(both upper and lower case) and '\_' |
| s | Denotes a set of all space related characters |
| ^ | Denotes a not character which means to not use the set succeeding ^ |

**Search**  
  
Search method searches for the string present in the r"" in the whole input string and returns a match object if there is a match else returns None. Search only returns the first match present in the string.

1. sent3 = "1947 was when India became independent."
2. print("Occurences of a-z: ",re.search(r"[a-z]+",sent3))
3. *#prints Occurences of a-z: <\_sre.SRE\_Match object; span=(5, 8), match='was'>*

The match at the string "was" is the only string according to search method because the space after it is not in our desired set. "+" indicates match the substring which has atleast one character.

1. sent3 = "1947 was when India became independent."
2. print("Occurences of 0-9: ",re.search(r"[0-9]+",sent3))
3. *#prints Occurences of 0-9: <\_sre.SRE\_Match object; span=(0, 4), match='1947'>*

The match at the string "1947" is the only string according to search method because our set only ranges from 0 to 9.

1. sent3 = "1947\_was when India became independent."
2. print("Occurences of w and space: ",re.search(r"[\w ]+",sent3))
3. *#prints Occurences of 0-9: <\_sre.SRE\_Match object; span=(0, 38), match='1947\_was when India became independent'>*

Our desired set is namely a-z, A-Z, 0-9, '\_' and a space character.

**Sub**

Sub is substitution of a substring with another string in the given input string. So understandably it takes three parameters.

First argument is the string to be removed, second argument is the resultant string and the last argument is the input string.

1. sent = "I like coffee"
2. print(re.sub(r"coffee","tea",sent))
3. *#prints I like tea*

**Findall**

Findall parses our input string from left to right and returns all the substrings matching with our raw string as a list.

1. sent = "I like coffee and coffee is amazing. coffee keeps me awake. coffee is bad"
2. print(re.findall(r"coffee",sent))
3. *#prints ['coffee', 'coffee', 'coffee', 'coffee']*
4. print(len(re.findall(r"coffee",sent)))
5. *#prints 4*

Go through python docs for finding more useful methods with re.

### **Text annotation using chunking and named entity recognition**

# Annotations

Consider the following [text](http://10.177.157.28:3006/web-hosted/auth/lex_auth_0126422253616578561056/assets/barack.txt)

Adding metadata to the text helps provide context to the NLP tasks. Such metadata is called **annotation**

For example, the outcome of annotation on the above article could be

(Barack Hussein Obama/Person (born August 4, 1961/Date of birth) is an American politician/Occupation who served as the 44th President/Designation of the United States/Country from January 20, 2009/Date of commencement, to January 20, 2017/Date of conclusion.)/Sentence1

(A member of the Democratic Party/Political party, he was the first African American to assume the presidency and previously served as a United States/Country Senator/Designation from Illinois/State (2005–2008).)/Sentence2

In this text, we have different types of entities, as listed below:

* Name of a person: Barrack Hussein Obama
* Country: United States
* Occupation: Politician
* Designation: President, Senator
* Date of birth: August 4, 1961
* Political Party: Democratic Party
* State: Illinois
* Date of commencement of Presidency: January 20, 2009
* Date of conclusion of Presidency: January 20, 2017.

In the annotated text, you may have observed that the following kinds of annotation are required:

* Sentence annotation - Identify the position of each sentence
* Word annotation - Identify the position of each word in the given text
* Person annotation - Identify the position of the person's name
* Place annotation - Identify the position of a geographical location
* Date annotation - Identify the position of a date,
* Many more...

Many such annotators need to be used in conjunction to extract relevant information from the text that is being processed.

After running the text through the annotation pipeline, we would get an output in a format similar to the below table

| **Element** | **Start position** | **End position** |
| --- | --- | --- |
| Sentence 1 | 0 | sent1\_end\_pos |
| Sentence 2 | sent1\_end\_pos + 1 | sent2\_end\_pos |
| Similarly for other sentences | | |
| Word 1 | 0 | word1\_end\_pos |
| Word 2 | word1\_end\_pos + 2 (including space) | word2\_end\_pos |
| Similarly for other words | | |
| Person 1 | person1\_start\_pos | person1\_end\_pos |
| Person 2 | person2\_start\_pos | person2\_end\_pos |
| Similarly for other people, and other named entities... | | |

# Chunking

The below code demonstrates the usage of nltk.ne\_chunk() method which works on a POS tagged list of the text.

1. from nltk import word\_tokenize
2. from nltk import pos\_tag
3. from nltk import ne\_chunk
4. barack = """Barack Hussein Obama (born August 4, 1961) is an American politician
5. who served as the 44th President of the United States from January 20, 2009, to January 20, 2017.
6. A member of the Democratic Party, he was the first African American to assume the presidency
7. and previously served as a United States Senator from Illinois (2005–2008)."""
8. tokenised\_barack = word\_tokenize(barack)
9. pos\_list = pos\_tag(tokenised\_barack)
10. print(ne\_chunk(pos\_list))

This code produces the following output

1. *# Please execute the previous code snippet on your computer to follow along*
2. (S
3. (PERSON Barack/NNP)
4. (PERSON Hussein/NNP Obama/NNP)
5. (/(
6. born/VBN
7. August/NNP
8. 4/CD
9. ,/,
10. 1961/CD
11. )/)
12. is/VBZ
13. an/DT
14. (GPE American/JJ)
15. politician/NN
16. ...
17. )

You can observe that the output contains only few of the desired annotations. To get better annotations, we can use other chunkers that use regular expressions.

**RegexpParser**

The below code demonstrates the usage of RegexpParser which can give a more desirable result in comparison to the default NE Chunker. Here we need to configure how entities are determined, i.e. what kind of POS combinations results in a specific named entity.

1. from nltk import RegexpParser
2. from nltk import word\_tokenize
3. from nltk import pos\_tag
4. barack = """Barack Hussein Obama II born August 4, 1961) is an American politician
5. who served as the 44th President of
6. the United States from January 20, 2009, to January 20, 2017.
7. A member of the Democratic Party, he was the
8. first African American to assume the presidency and previously
9. served as a United States Senator from Illinois (2005–2008)."""
10. grammar = r"""Place: {<NNP><NNPS>+}
11. Date: {<NNP><CD><,><CD>}
12. Person: {<NNP>+}
13. """
14. tokenised\_barack = word\_tokenize(barack)
15. pos\_list = pos\_tag(tokenised\_barack)
16. regParser = RegexpParser(grammar)
17. reg\_lines = regParser.parse(pos\_list)
18. print(reg\_lines)

The output is as shown below

1. *# Please run the above code snippet to get the complete output*
2. (S
3. (Person Barack/NNP Hussein/NNP Obama/NNP II/NNP)
4. born/VBD
5. (Date August/NNP 4/CD ,/, 1961/CD) )/)
6. is/VBZ
7. an/DT
8. American/JJ
9. politician/NN
10. who/WP
11. ...
12. (Person President/NNP)
13. of/IN
14. the/DT
15. (Place United/NNP States/NNPS)
16. ...
17. )

# Tagging

We have seen the POS tagger which identifies the part of speech of each word in a given sentence. In this sections we will dive deeper into the different models that can be used to do POS tagging.

Consider the below lines of code

1. sent1 = "The race officials refused to permit the team to race today"
2. print(pos\_tag(word\_tokenize(sent1)))

The output here is

1. [('The', 'DT'), ('race', 'NN'), ('officials', 'NNS'), ('refused', 'VBD'), ('to', 'TO'), ('permit', 'VB'),
2. ('the', 'DT'), ('team', 'NN'), ('to', 'TO'), ('race', 'NN'), ('today', 'NN')]

However, if you observe the statement and consult your knowledge of the English grammar, you will realize that the word 'race' here is being used as a noun in the first occurrence and as a verb in the second.

Similarly, can you identify where there is a possible mistake in the below code?

1. sent2 = "That gentleman wants some water to water the plants"
2. print(pos\_tag(word\_tokenize(sent2)))
3. *# Output: [('That', 'DT'), ('gentleman', 'NN'), ('wants', 'VBZ'), ('some', 'DT'), ('water', 'NN'),*
4. ('to', 'TO'), ('water', 'NN'), ('the', 'DT'), ('plants', 'NNS')]

These mistakes in tagging are primarily because of how the taggers classify words and on what kind of data they have been trained.

Observe that the POS tagger gets the classification right for the below statement indicating that the error is not by default.

1. text = word\_tokenize("They refuse to permit us to obtain the refuse permit")
2. print(nltk.pos\_tag(text))
3. *# Prints [('They', 'PRP'), ('refuse', 'VBP'), ('to', 'TO'), ('permit', 'VB'), ('us', 'PRP'), ('to', 'TO'),*
4. ('obtain', 'VB'), ('the', 'DT'), ('refuse', 'NN'), ('permit', 'NN')]

NLTK provides different taggers that we can train and use in order to tag our unseen text data more efficiently. The taggers are:

* Default tagger
* Lookup taggers:
  + Unigram tagger - context independent tagging
  + Ngram tagger - context dependent tagging
* Regular Expression Tagger

We can also use a combination of these taggers to tag a sentence with the concept of backoff.

Note that in order to train a tagger, we need a corpus of tagged words.

# Default Tagger

The default tagger assigns the same tag to each token, this is considered the most naive tagger.

The below code demonstrates the usage of the default tagger on the Barack Obama article

1. *# importing a predefined corpus*
2. from nltk.corpus import brown
3. *# getting the most common tag in the brown corpus*
4. tags = [tag for (word,tag) in brown.tagged\_words()]
5. most\_common\_tag = nltk.FreqDist(tags).max()
6. print(most\_common\_tag)
7. *#Prints NN which means the most common POS is noun*
8. *# Using the most\_common\_tag as the input for DefaultTagger*
9. from nltk import DefaultTagger
10. default\_tagger  = DefaultTagger(most\_common\_tag)
11. def\_tagged\_barack = default\_tagger.tag(tokenised\_barack)
12. print(def\_tagged\_barack)
13. *#Prints [('Barack', 'NN'), ('Hussein', 'NN'), ('Obama', 'NN'), ('II', 'NN'), ('born', 'NN'),*
14. ('August', 'NN'), ('4', 'NN'), (',', 'NN'), ('1961', 'NN'), (')', 'NN'), ('is', 'NN'), ... ]

# Lookup Taggers

A NgramTagger tags a word based on the previous n words occurring in the text.

For example consider that we have only one tagged sentence in the training set as shown below

1. from nltk import word\_tokenize
2. sent1 = "the quick brown fox jumps over the lazy dog"
3. training\_tags = pos\_tag(word\_tokenize(sent1))
4. print(training\_tags)
5. """Prints [('the', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'VBZ'),
6. ('over', 'IN'), ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')] """
7. *# Now let us use these tags to train the NgramTagger*
8. ngram\_tagger = nltk.NgramTagger(n=2,train=[training\_tags]) *#Here when we set n=2, we are creating a bigram tagger*

Having provided the training data to the NgramTagger, we can now use it to tag a new sentence as shown below

1. sent2 = "the lazy dog was jumped over by the quick brown fox"
2. sent2\_tags = ngram\_tagger.tag(word\_tokenize(sent2))
3. print(sent2\_tags)

The output of the above code is

1. [('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN'), ('was', None), ('jumped', None),
2. ('over', None), ('by', None), ('the', None), ('quick', None), ('brown', None), ('fox', None)]

From the above output you can notice that many of the words have been tagged as None, which means that the tagger was unable to find the appropriate POS tag for the word in the given context.

In order to understand why this happened, we need to learn what context means for a NgramTagger.

# Context of an NGramTagger

Let us consider n=2 for a NGram Tagger and use that to understand the context.

The tagged training list [('the', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'VBZ'),   ('over', 'IN'), ('the', 'DT'), ('lazy', 'JJ'), ('dog', 'NN')] is converted to bigrams as below

1. print(list(nltk.ngrams(pos\_tag(word\_tokenize(sent1)),n=2)))
2. """Prints [(('the', 'DT'), ('quick', 'JJ')),
3. (('quick', 'JJ'), ('brown', 'NN')),
4. (('brown', 'NN'), ('fox', 'NN')),
5. (('fox', 'NN'), ('jumps', 'VBZ')),
6. (('jumps', 'VBZ'), ('over', 'IN')),
7. (('over', 'IN'), ('the', 'DT')),
8. (('the', 'DT'), ('lazy', 'JJ')),
9. (('lazy', 'JJ'), ('dog', 'NN'))] """

Now, when the Ngramtagger has to tag the new sentence "the lazy dog was jumped over by the quick brown fox", it converts this sentence into bigrams as well, as below

1. print(list(nltk.ngrams(word\_tokenize(sent2),n=2)))
2. """Prints [('the', 'lazy'),
3. ('lazy', 'dog'),
4. ('dog', 'was'),
5. ('was', 'jumped'),
6. ('jumped', 'over'),
7. ('over', 'by'),
8. ('by', 'the'),
9. ('the', 'quick'),
10. ('quick', 'brown'),
11. ('brown', 'fox')] """

 The NgramTagger then does a **lookup**for matching bigrams in the training data and uses that to tag the new data.

Since the pairs (the, lazy) and (lazy, dog) appear in the training data, the tagger is able to tag the words "the", "lazy" and "dog".

When it encounters the pair (dog, was) , this sequence was never present in the training data; so it assigns None to the word "was" and all other words succeeding it.

This looking up of occurrence of words in the sequence appearing in the training set can be considered as the **context.**

Therefore, we can now understand that a NgramTagger tags words that appear in context, and the context is defined by the window 'n' which is the number of tokens to consider together.

# Unigram tagger

UnigramTagger is a special case of NgramTagger where n=1. When n=1, then the NgramTagger has no context, i.e. each word is looked up independently in the training set. Therefore the UnigramTagger is also referred to as the context independent tagger.

The UnigramTagger performs a looks up the query word in the training data and assigns the most common tag associated with it.

The below code demonstrates the usage of a UnigramTagger.

1. barack = """Barack Hussein Obama II born August 4, 1961) is an American politician
2. who served as the 44th President of
3. the United States from January 20, 2009, to January 20, 2017.
4. A member of the Democratic Party, he was the
5. first African American to assume the presidency and previously
6. served as a United States Senator from Illinois (2005–2008)."""
7. bush = """George Walker Bush (born July 6, 1946) is an American politician who served as the 43rd President
8. of the United States from 2001 to 2009.
9. He had previously served as the 46th Governor of Texas from 1995 to 2000.
10. Bush was born New Haven, Connecticut, and grew up in Texas.
11. After graduating from Yale University in 1968 and Harvard Business School in 1975, he worked in the oil industry.
12. Bush married Laura Welch in 1977 and unsuccessfully ran for the U.S. House of Representatives shortly thereafter.
13. He later co-owned the Texas Rangers baseball team before defeating Ann Richards in the 1994 Texas gubernatorial election.
14. Bush was elected President of the United States in 2000 when he defeated Democratic incumbent
15. Vice President Al Gore after a close and controversial win that involved a stopped recount in Florida.
16. He became the fourth person to be elected president while receiving fewer popular votes than his opponent.
17. Bush is a member of a prominent political family and is the eldest son of Barbara and George H. W. Bush,
18. the 41st President of the United States.
19. He is only the second president to assume the nation's highest office after his father, following the footsteps
20. of John Adams and his son, John Quincy Adams.
21. His brother, Jeb Bush, a former Governor of Florida, was a candidate for the Republican presidential nomination
22. in the 2016 presidential election.
23. His paternal grandfather, Prescott Bush, was a U.S. Senator from Connecticut."""
24. pos\_tag\_barack = pos\_tag(word\_tokenize(barack))
25. pos\_tag\_bush = pos\_tag(word\_tokenize(bush))
26. trump = """Donald John Trump (born June 14, 1946) is the 45th and current President of the United States.
27. Before entering politics, he was a businessman and television personality.
28. Trump was born and raised in the New York City borough of Queens, and received an economics degree from the
29. Wharton School of the University of Pennsylvania.
30. He took charge of his family's real estate business in 1971, renamed it The Trump Organization, and expanded
31. it from Queens and Brooklyn into Manhattan.
32. The company built or renovated skyscrapers, hotels, casinos, and golf courses.
33. Trump later started various side ventures, including licensing his name for real estate and consumer products.
34. He managed the company until his 2017 inauguration.
35. He co-authored several books, including The Art of the Deal. He owned the Miss Universe and Miss USA beauty
36. pageants from 1996 to 2015, and he produced and hosted the reality television show The Apprentice from 2003 to 2015.
37. Forbes estimates his net worth to be $3.1 billion."""
38. unigram\_tag = nltk.UnigramTagger(train=[pos\_tag\_barack,pos\_tag\_bush])
39. trump\_tags = unigram\_tag.tag(word\_tokenize(trump))
40. print(trump\_tags)

After you run the above code, you can observe that tags for words in the Trump article that neither occurred in the article about Bush nor Obama are marked as None.

# Tagging pipeline and backoff

Having studied different taggers like DefaultTagger, RegexpTagger and  NgramTagger, let us now see how we can combine them together to automatically tag text data.

The below code demonstrates the usage of the three taggers that we have so far learnt

1. default\_tagger  = DefaultTagger('NN')
2. patterns = [
3. (r'.\*\'s$', 'NN$'),               *# possessive nouns*
4. (r'.\*es$', 'VBZ'),                *# 3rd singular present*
5. (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  *# cardinal numbers*
6. (r'[Aa][Nn][Dd]','CC'),           *# conjugate and*
7. (r'.\*ed$', 'VBD'),                *# simple past*
8. (r',',',')                       *# comma*
9. (r'.\*ould$', 'MD'),               *# modals*
10. (r'.\*ing$', 'VBG'),               *# gerunds*
11. (r'.\*s$', 'NNS'),                 *# plural nouns*
12. *# You will need to add several such rules to make an efficient tagger*
13. ]
14. regexp\_tagger = nltk.RegexpTagger(patterns,backoff=default\_tagger)
15. unigram\_tag = nltk.UnigramTagger(train=[pos\_tag\_barack,pos\_tag\_bush],backoff=regexp\_tagger)
16. trump\_tags = unigram\_tag.tag(word\_tokenize(trump))
17. print(trump\_tags)

In the above code, the UnigramTagger is first invoked to tag the tokens in the Trump article. Whichever words are tagged None by this UnigramTagger are then sent as **backoff** to the RegexpTagger. The RegexpTagger then tags the words based on the patterns rule it is fed. Any words that are still left untagged are then sent to the DefaultTagger as backoff.

## **Working with multiple documents**

# Corpus

So far we have been working with either a single text file or textual data that is dynamic and created during code execution. Let us now explore how we can work with multiple text documents.

A collection of documents is referred to as a corpus. NLTK provides several corpora out of the box.

The below code demonstrates how to load predefined corpora and use it.

1. from nltk.corpus import brown
2. *# brown corpus is a tagged corpus where each word in each file of the corpus is associated with its POS tag*
3. *#Display all the files that are there in the corpus*
4. print(brown.fileids())
5. *#Display the paragraphs, sentences and words in a specific file of the corpus*
6. print(brown.paras('ca01'))
7. print(brown.sents('ca01'))
8. print(brown.words('ca01'))
9. *#Display the POS tag for each word in a specific file of the corpus*
10. print(brown.tagged\_words('ca01'))
11. *#Loading the CoNLL2000 corpus which has 270k words that are tagged and chunked*
12. from nltk.corpus import conll2000
13. print(conll2000.fileids())
14. print(conll2000.words('train.txt'))
15. print(conll2000.chunked\_sents('train.txt'))
16. *# When you print the chunked sentences, observe how chunks of tagged words are tagged.*
17. *# For example, a chunk containing ('the' , 'DT') and ('pound', NN') is tagged as 'NP'*

In addition to the existing corpora, we can create our own corpora using the NLTK API. The below code demonstrates how we can create a corpora out of a collection of text files. For this example, we are using [3 text files - barack.txt, bush.txt and trump.txt](http://10.177.157.28:3006/web-hosted/lex_auth_0125952483347906561501/assets/text_docs.zip).

1. from nltk.corpus.reader.plaintext import PlaintextCorpusReader
2. path = "./text\_docs/"
3. president\_corpus = PlaintextCorpusReader(path,".\*")
4. *# Display the files in the corpus*
5. print(president\_corpus.fileids()) *#Prints ['barack.txt', 'bush.txt', 'trump.txt']*
6. *#Display the sentences in a specific file*
7. print(president\_corpus.sents('barack.txt'))
8. *#Display the sentences in all files of the corpus*
9. print(president\_corpus.sents())
10. *#Display the words in a specific file*
11. print(president\_corpus.words('barack.txt'))
12. *#Display the words in the corpus*
13. print(president\_corpus.words())

Note that the PlainTextCorpusReader automatically splits the text data into paragraphs, sentences and words using appropriate tokenizers.

Bag of words and TF-IDF models

**Vectorizing textual data**

Now that we have created clubbed multiple text documents into a corpus, let us explore concepts such as Bag-of-words and TF-IDF which can help us determine similarity between documents. These are models that represent text documents as vectors.

**Bag of words model**

Consider 2 documents containing one sentence each, as follows

* Document 1: India is a republic country. We are proud Indians.
* Document 2: The current Prime Minister of India is Shri. Narendra Modi.

In order to determine the similarity between these documents, we convert each of them into a vector of words and their counts. In the bag of words model, every word that occurs in the corpus (across all documents) is considered as a feature and document is represented as follows.

| **Document/Term** | **india** | **is** | **a** | **republic** | **country** | **we** | **are** | **proud** | **indians** | **the** | **current** | **prime** | **minister** | **of** | **shri** | **narendra** | **modi** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Document1** | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Document2** | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

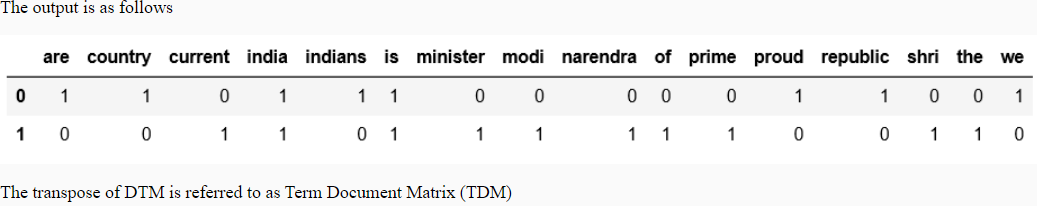
 Suppose that we considered 2 documents are similar if the words occurring them are the same, then we just need to evaluate how many words are common in between the two documents.

The matrix (table) that you see here is called as the Document Term Matrix (DTM) where each row represents a document and each column represents the term (token) in the document.

Observe that while creating the DTM, some pre-processing is done, i.e, all words are converted to lower case and punctuation is removed.

The below code demonstrates the computation of the DTM for the sample sentences using the CountVectorizer provided by scikit-learn

1. from sklearn.feature\_extraction.text import CountVectorizer
2. import pandas as pd
3. sent1 = "India is a republic country. We are proud Indians."
4. sent2 = "The current Prime Minister of India is Shri. Narendra Modi."
5. count\_vectorizer = CountVectorizer()
6. dtm  = count\_vectorizer.fit\_transform([sent1,sent2])
7. print(pd.DataFrame(data=dtm.toarray(), columns=count\_vectorizer.get\_feature\_names()))



The similarity between the two documents can now be found using commonly used distance metrics like Euclidean distance or cosine distance.

The below code demonstrates the use of cosine distance. The smaller this value, the more similar the two documents.

1. from scipy.spatial.distance import cosine
2. print(cosine(dtm[0].toarray(),dtm[1].toarray()))
3. *#Prints 0.7763932022500211 indicating that the two documents are not very similar*

# TF-IDF model

TF-IDF is an abbreviation for Term Frequency - Inverse Document Frequency

Here Term Frequency refers to the count of the occurrence of the term (token) within the document and Inverse Document Frequency refers to the count of the occurrence of a term across all documents.

In this model, each document is represented as a vector of TF-IDFs. This model results in giving a lower weightage to words that commonly occur across the corpus (typically stop-words)

The below code demonstrates the creation of TF-IDF vectors for the two sample documents

1. from sklearn.feature\_extraction.text import TfidfVectorizer
2. import pandas as pd
3. sent1 = "India is a republic country. We are proud Indians."
4. sent2 = "The current Prime Minister of India is Shri. Narendra Modi."
5. tfidf\_vectorizer = TfidfVectorizer()
6. tfidf\_vectors  = tfidf\_vectorizer.fit\_transform([sent1,sent2])
7. print(pd.DataFrame(data=tfidf\_vectors.toarray(),columns=tfidf\_vectorizer.get\_feature\_names()))
8. *#Exercise: Compute the cosine distance of these two documents using tfidf\_vectors*

Let us now determine the TF-IDF vectors and similarities between the 3 documents in our corpus (president\_corpus)

1. from nltk.corpus.reader.plaintext import PlaintextCorpusReader
2. from sklearn.feature\_extraction.text import TfidfVectorizer
3. path = "./text\_docs/"
4. president\_corpus = PlaintextCorpusReader(path,".\*",encoding="utf-8")
5. tf\_idf = TfidfVectorizer(input='filename')
6. files = [path+filename for filename in list(president\_corpus.fileids())]
7. tf\_idf\_matrix = tf\_idf.fit\_transform(raw\_documents=files)
8. barack = tf\_idf\_matrix.toarray()[0]
9. bush = tf\_idf\_matrix.toarray()[1]
10. trump = tf\_idf\_matrix.toarray()[2]
11. from scipy.spatial.distance import cosine
12. print("Distance between articles on Barack and Bush is:", cosine(barack,bush))
13. print("Distance between articles on Barack and Trump is:", cosine(barack,trump))
14. print("Distance between articles on Bush and Trump is:", cosine(bush,trump))

## **Lexical resources**

Wordlists

# Lexical resources

NLTK provides different word lists as lexical resources that can be used while processing textual data.

Here are some sample scenarios of using them

1. Detect spelling mistakes and unusual words in text
2. Remove stop words
3. Identify names of people in text (proper nouns)
4. Look up synonyms while rephrasing an article

Let us now use these lexical resources to achieve some of the above scenarios

# Detect unusual words in text

NLTK provides the vocabulary of all English words in the nltk.corpus.words module

Here is an example of detecting unusual words in some text by comparing the words in it to standard words in NLTK's wordlist.

1. import nltk
2. sent1 = """Just forced myself to eat a slice. I'm really not hungry tho.
3. Mark is getting worried. He knows I'm sick when I turn down pizza. Lol"""
4. sent2 = "I call you later, don't have nw. If urgnt, sms me."
5. sent3 = "Watching a telugu movie..wat abt u?"
6. def find\_unusual\_words(text):
7. text\_vocab\_set = set(w.lower() for w in text if w.isalpha())
8. english\_vocab\_set = set(w.lower() for w in nltk.corpus.words.words())
9. unusual\_set = text\_vocab - english\_vocab
10. return sorted(unusual)
11. print(find\_unusual\_words(nltk.wordpunct\_tokenize(sent1)))
12. *#Prints ['knows', 'lol']*
13. print(unusual\_words(nltk.wordpunct\_tokenize(sent2)))
14. *#Prints ['nw', 'sms', 'urgnt']*
15. print(unusual\_words(nltk.wordpunct\_tokenize(sent3)))
16. *#Prints ['abt']*

# Detect possible spelling mistakes

If a word could not found in the word list, it is probable that it is a spelling mistake.

The below code, compares the unusual words with known words and suggests possible words based on edit distance. Edit distance is the measure of how similar or dissimilar two words are.

1. unusual\_words\_found = ['knows', 'lol', 'nw', 'sms', 'urgnt', 'abt']
2. from nltk.metrics import edit\_distance
3. possible\_suggestions={}
4. english\_vocab\_set = set(w.lower() for w in nltk.corpus.words.words())
5. for unusual\_word in unusual\_words\_found:
6. for word in english\_vocab\_set:
7. dist = edit\_distance(unusual\_word,word)
8. if dist<len(unusual\_word)/2:
9. if unusual\_word not in possible\_suggestions.keys():
10. possible\_suggestions[unusual\_word] = [word]
11. else:
12. possible\_suggestions[unusual\_word].append(word)
13. print(possible\_suggestions["lol"])

# Detect names of people in the text

The below code identifies is there is a known name in the text using nltk.corpus.names.

1. def names\_in\_text(text):
2. names = []
3. words\_set = set(i for i in text if i.isalpha())
4. male\_names = nltk.corpus.names.words('male.txt')
5. female\_names = nltk.corpus.names.words('female.txt')
6. for w in words\_set:
7. if i in male\_names or i in female\_names:
8. names.append(i)
9. return names
10. sent1 = "John and Mary go to the church every Sunday"
11. sent2 = "No man has ever seen the dark side of the Moon"
12. print(names\_in\_text(word\_tokenize(sent1)))
13. print(names\_in\_text(word\_tokenize(sent2)))

Note that this code only works on the names that are known in the corpus. You may considering trying a similar exercise with a corpus that you create containing names of people, places, organizations etc.

# WordNet

WordNet is a lexical database for English created at Princeton University. It contains over 150,000 words and 110,000 synonyms.

The below code demonstrates the various methods of wordnet package of nltk.

1. from nltk.corpus import wordnet as wn
2. *# Get all possible meanings of the word "dog*
3. print(wn.synsets("dog"))
4. *# Get all lemma names of "dog"*
5. print(dog.lemma\_names())
6. *#Get all hypernyms of "dog"*
7. print(wn.synset('dog.n.01').hypernyms())
8. *# A hypernym is the generic term where as a hyponym is a specific term*
9. *# For the word dog, the hypernyms are 'canine' and 'domestic\_animal'*
10. *#Get all hyponyms of "dog"*
11. print(wn.synset('dog.n.01').hyponyms())
12. *# some of hyponyms are "pug", "puppy", "lap\_dog", etc..*
13. *#Get the path similarity between to words - the method returns the shortest path in the taxonomy*
14. dog = wn.synset('dog.n.01')
15. cat = wn.synset('cat.n.01')
16. print(cat.path\_similarity(dog)) *#Returns a value between 0 and 1. The higher the number, higher the similarity in path*
17. *# wu and palmer similarity method.*
18. """ Produces similarity values based on their Least Common Subsumer (most specific ancestor node) and
19. the maximum depth in the taxonomy"""
20. cat.wup\_similarity(dog)
21. *# Get all synonyms of the word 'good'*
22. synonyms = []
23. for syn in wn.synsets("good"):
24. for word in syn.lemmas():
25. if word.name() != "good":
26. synonyms.append(word.name())
27. print(synonyms)
28. *# Get all antonyms of the word "good"*
29. antonyms = []
30. for syn in wn.synsets("good"):
31. for word in syn.lemmas():
32. if word.name() != "good" and word.antonyms() :
33. antonyms.append( word.antonyms()[0].name())
34. *# print(antonyms)*
35. *# Return the base form (morphy) of a word*
36. print(wn.morphy("working" , wn.VERB)) *#Prints "work"*
37. print(wn.morphy("denied" , wn.VERB)) *#Prints "deny"*
38. print(wn.morphy("abaci")) *#Prints "abacus"*

## 

The following examples show how NLP has made an impact in various fields

# Chatbots in healthcare

Chatbots like **Molly, Eva, Ginger, Replika, Florence, and Izzy** are widely used in healthcare. These bots provide the following support by

* Providing end to end suggestions to patients from their diagnosis to treatment options
* Managing medication
* Helping in emergency situations or with first aid
* Offering a solution for simpler medical issues

# Chatbots for mental health support

Bots like **Wysa and Woebot** are designed in such a way that they can provide support like a life coach. They are so good at asking right probing questions that can help the user to share their emotions and feelings after a hard day.

# Chatbots in banks

Banks like **Bank of America, JP Morgan Chase and Wells Fargo** have chatbots to provide financial advice and digital support.

Basic tasks like balance inquiry, bank account details, loan queries, etc. can be handled by these bots efficiently, so that the customer service representatives of the banks can invest their time for handling complex issues where human expertise is crucial.

# Chatbots for legal advice

Lawyers can use bots like **DonotPay, LISA, Ross, and BillyBot** to accelerate their work and provide better client experiences.

NLP is also playing an increasing role in areas like Legal research by

* Finding relevant information to make legal decisions
* Finding the relevance of documents to information requests
* Contract review to check if a contract is complete
* Document automation to generate routine legal documents
* Questions and answer dialogs to provide tailored legal advice

NLP is used across different domains for Brand Sentiment Analysis, Recruitment, and Optimizing call center operations.

# Brand sentiment analysis

Brand sentiment analysis helps in understanding the emotional tone of consumers via social posts in the public domain. Knowing the opinions and having a real-time view of the customer’s pulse is a critical element of brand marketing.

# Recruitment

Companies can use a semantic search to filter resumes that seem like a good fit for open positions.

# Call center operations

High volumes of consumer interaction create the need for a critical capability to prioritize which tasks to act upon first. Using voice to text, NLP and machine learning can quickly deliver insights to the most important customer inquiries.

# Other NLP applications

In Smart keyboards like Swiftkey, the software automatically completes your sentences by predicting the next word and corrects your spelling mistakes.

Applications like Grammarly can automatically correct your spelling and grammar and assists you in writing better essays or emails.

Ambiguities in Natural Language

# Lexical ambiguity

Take a look at the following sentences:

John bagged two silver medals.

Mary made a silver speech.

Roger’s worries had silvered his hair.

The word silver is used as a noun, an adjective, and a verb.

The word silver in isolation is mostly associated with the metal and considered as a noun. However, in other sentences, the context gives the word silver different meanings and also different parts of speech like adjectives and verbs. This ambiguity is called lexical ambiguity.

# Syntactic ambiguity

Take a look at the sentence given below

“Old men and women were taken to safe locations”

This sentence has a syntactic ambiguity where the scope of the adjective “old” needs to be resolved.

In this sentence, we may not know if the adjective applies only to men or to both men and women.

# Semantic ambiguity

Semantic ambiguity refers to ambiguity in the meaning.

For example, the sentence

“Alice loves her mother and so does Jacob.”

The ambiguity here is, we may not know if Jacob loves his own mother or Alice’s mother.

# Anaphoric ambiguity

In the below paragraph

“The horse ran up the hill. It was very steep. It soon got tired.”

In this paragraph, the pronoun ‘it’ is used to refer to the hill first and then to the horse. To interpret this sentence, we need to have knowledge of the world and context. These ambiguities are called anaphoric ambiguities.

# Pragmatic Ambiguity

The hardest kind of ambiguity to resolve is the pragmatic ambiguity. This kind of ambiguity arises from the inability to process the intention or sentiment or world belief.

For example, in the below conversation,

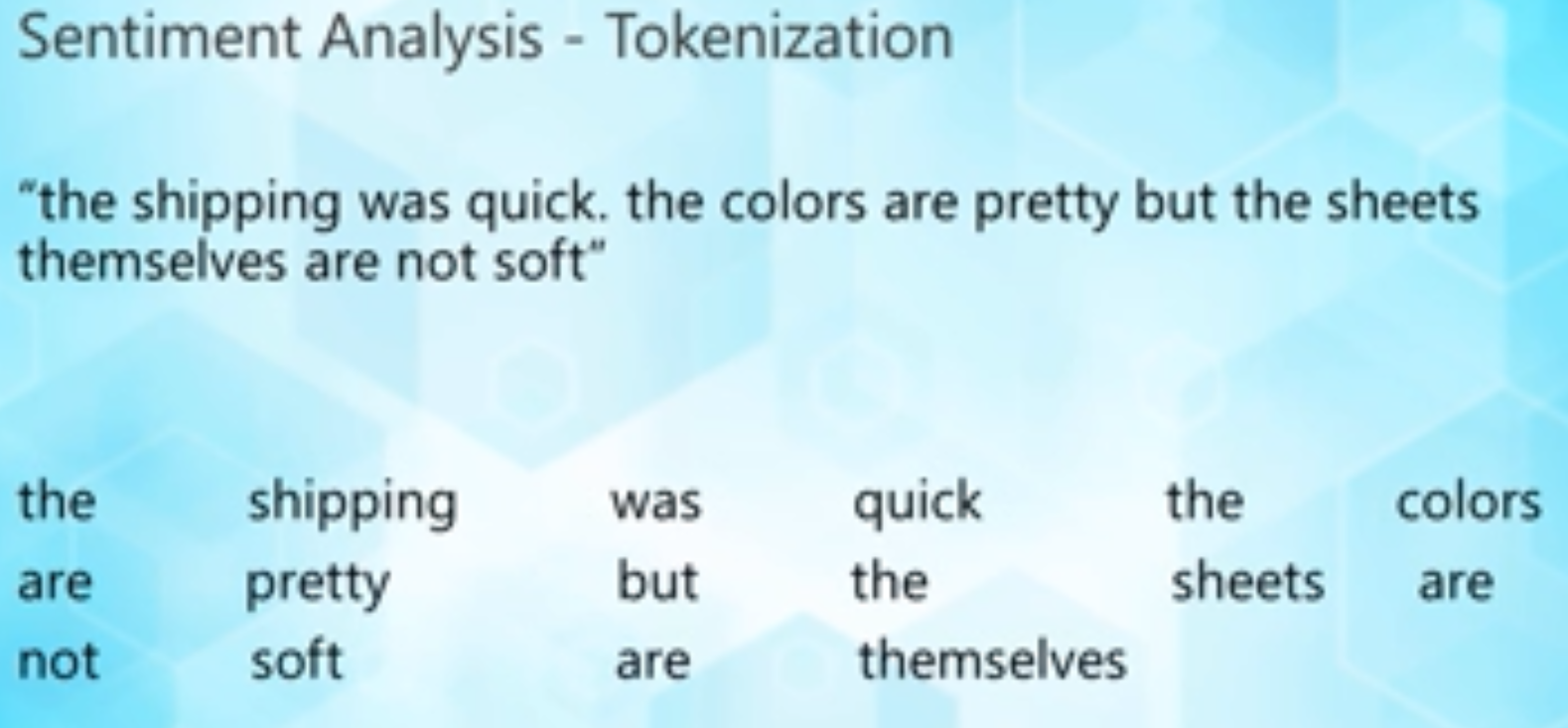
My wife said: "Please go to the store and buy a carton of milk and if they have eggs, get six."

I came back with 6 cartons of milk

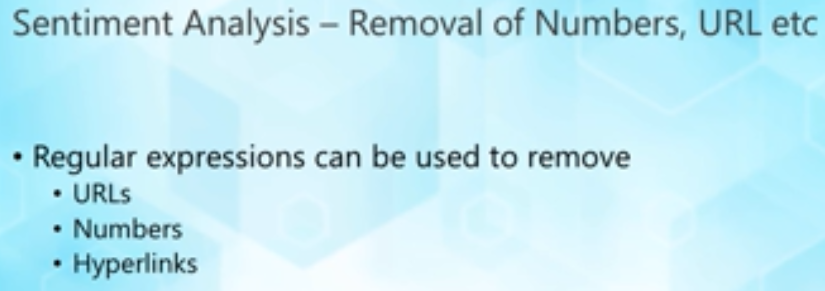
She said, "why did you buy six cartons of milk?"

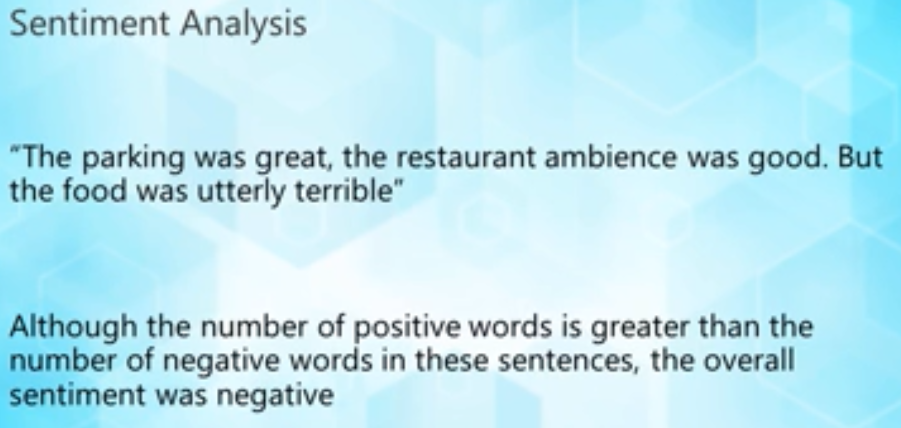
I replied, "They had eggs"

As you can see here, the ambiguity is in understanding the intention of the speaker.









Having seen a few popular applications of NLP like speech recognition, machine translation and sentiment analysis let us now take a look at the various techniques that are commonly used in building these systems.

Speech to text conversion (Automatic Speech Recognition)

This task is required in systems that usually have a voice user interface. The primary responsibility of systems that are built for this task is to accurately convert the audio waves (speech) into text representations. Several speech recognition systems use a combination of acoustic models that correlate audio waves to phonemes (how different words sound) and language models that are statistical distributions over word sequence used to provide context and distinguish between words/phrases that could sound similar (like hear vs here).

Speech recognition is a broad area of study and research by itself and recent use of machine learning techniques (especially deep learning) have contributed to more reliable and accurate speech recognition systems

**Handling/processing text**

Text processing comprises several techniques including tokenization, stop word removal, part of speech tagging, stemming, lemmatization, etc. Lets look at a few of these techniques:

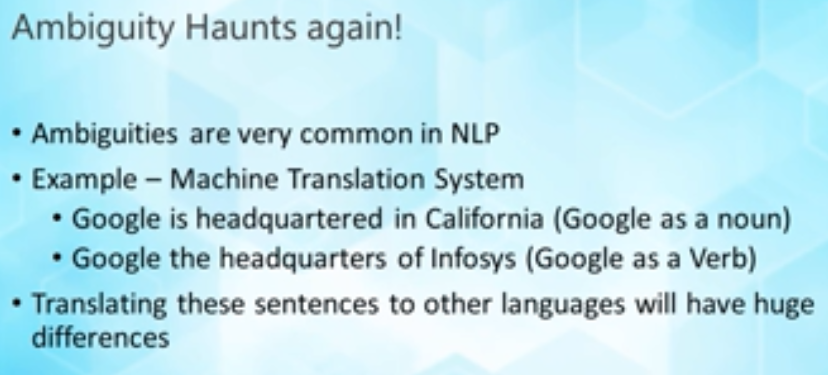
* **Tokenization**  
  In tokenization, the whole text is split into *tokens.*Sometimes we can assume that a token is a word, however that is not always necessary. During the process of tokenization, one may choose to also remove special characters appearing in punctuation or otherwise. Each token then can be used for several tasks in NLP. For example, if you were to create a translator you can construct a token based translation (although this may not work as good as you want a translator to).   
  1. Example:
  2. input - "He did not try to navigate after the first bold flight, for the reaction had taken something out of his soul."
  3. ouptut - ['He', 'did', 'not', 'try', 'to', 'navigate', 'after', 'the', 'first', 'bold', 'flight', ',', 'for', 'the', 'reaction', 'had', 'taken', 'something', 'out', 'of', 'his', 'soul', '.']

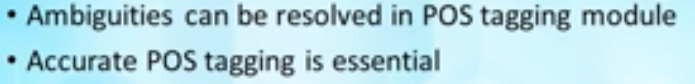
* **Stop word removal**  
  Stop words are often considered to be words that frequently appear across the entire corpus of text used for analysis. For example, in English the words  'the', 'and', 'to', etc. are quite common. Stop word removal is not necessary fro all applications. For example, we might remove stop words when we are clustering similar documents but we might choose to retain them when we are doing machine translation.  
  1. Example:
  2. input - ['He', 'did', 'not', 'try', 'to', 'navigate', 'after', 'the', 'first', 'bold', 'flight', ',', 'for', 'the', 'reaction', 'had', 'taken', 'something', 'out', 'of', 'his', 'soul', '.']
  3. output - ['try', 'navigate', 'first', 'bold', 'flight', ',', 'reaction', 'taken', 'something', 'soul', '.']

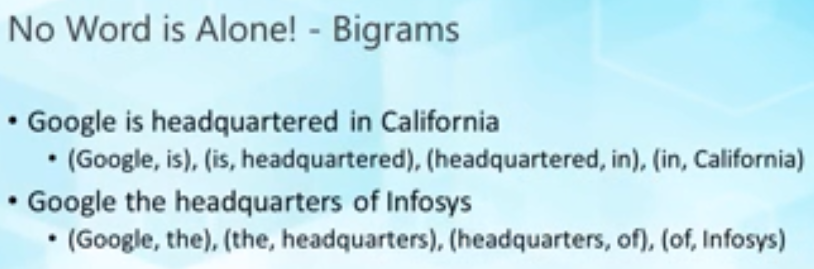
* **Part of speech (POS) tagging**  
  Part of speech tagging involves identifying the right form of the word (verb, noun, etc.). Identifying the correct part of speech being used is important to several NLP applications such as machine translation, text summarization, understanding meaning and context in speech, sentiment analysis etc.
  1. Example:
  2. input - ['And', 'from', 'their', 'high', 'summits', ',', 'one', 'by', 'one', ',', 'drop', 'everlasting', 'dews', '.']
  3. output - [('And', 'CC'),
  4. ('from', 'IN'),
  5. ('their', 'PRP$'),
  6. ('high', 'JJ'),
  7. ('summits', 'NNS'),
  8. (',', ','),
  9. ('one', 'CD'),
  10. ('by', 'IN'),
  11. ('one', 'CD'),
  12. (',', ','),
  13. ('drop', 'NN'),
  14. ('everlasting', 'VBG'),
  15. ('dews', 'NNS'),
  16. ('.', '.')]
  17. where,
  18. JJ: Adjective
  19. RB: Adverb
  20. CC: Conjunction
  21. DT: Determiner
  22. MD: Modal Verb
  23. NN: Noun
  24. CD: Numeral
  25. VR: Participle
  26. RP: Particle
  27. PO: Possessive Ending
  28. IN: Preposition
  29. PP: Pronoun
  30. SY: Symbol
  31. VB: Verb
  32. Refer to the Penn Treebank tagset for a complete list of tags

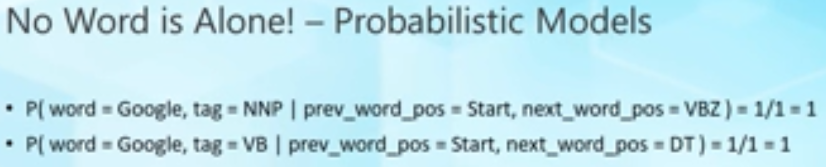
Several other steps such as standardizing the case of the text, removing URLs, numbers or special characters, normalize the data using stemming/lemmatization, etc.  are also involved in the text analysis pipeline.











# What is Deep Learning?

Deep learning is a class of machine learning algorithms based on artificial neural networks. Several kinds of deep learning architectures are in use today for solving a wide range of problems in areas such as computer vision, speech recognition, machine translation, image and video analysis, reinforcement learning and game AI. Deep learning architectures rely on stacking up several layers of neural networks where each layer learns to transform the input into abstract representations.

The key advantage that deep learning techniques offer in comparison to other machine learning techniques is their ability to automatically learn higher level features from the data. For example, if we were to use a classical machine learning algorithm from an application like spam detection, we would engineer several features like - length of email, contains URLs, contains unusual words, etc. ; however when using deep learning the layer of neural networks are able to learn such features by themselves requiring lesser manual feature engineering. The caveat however is that deep learning architectures are data and resource hungry and take a lot of time and resource while training.

Let us now look at some of the key advancements that deep learning has brought into the field of NLP

Some of the key advancements that were brought out by using deep learning techniques for NLP are listed below

# Distributed representations and word embedding

Distributed representations overcome limitation of manual feature representations. In these, we represent words (or tokens) as vectors (sequence of numbers) that can be used directly in machine learning for various subsequent tasks like sentiment analysis or machine translation. These vectors represent syntactic and semantic information of the tokens. A popular application of word embedding is in expressing semantic relationships among words e.g. 'man' - 'woman' = 'king' - 'queen' (word analogies). Commonly used word embedding models are word2vec and GloVe. Word embeddings have been used for various tasks in NLP such as POS tagging, named entity recognition and sentiment analysis.

# Use of Convolutional Neural Networks for text

Convolutional neural networks are neural networks that contain convolutional layers in them. These layer perform the convolution operation on the input signal. These layers apply a feature function on the input signal and help in extracting higher level abstractions which can effectively be used in various NLP tasks such as sentiment analysis, machine translation and question answering, speech recognition, etc.

# Use of Recurrent Neural Networks for NLP

Recurrent neural networks (RNNs) help build dynamic models that best capture variations in sequential and time based data which is often the nature of natural language. These networks by virtue of their structure have a memory component associated with them which helps maintain context over a long sequence of tokens/words. RNNs have been used in several NLP applications and a few notable RNN architectures like BERT, ElMo, etc. have performed better than previous benchmarks and achieved state of the art results.

The following table compiles the state of the art deep learning models for NLP based on the task (as of July 2019)

| **Task** | **Model** | **Performance score** |
| --- | --- | --- |
| POS Tagging | Deep Memory Network (DMN) | 97.56 % per token accuracy |
| Dependency Parsing | Deep fully-connected NN with features including POS | 94.30 unlabelled attachment score |
| Constituency Parsing | seq2seq learning with LSTM + attention | 93.5% F1 score |
| Named Entity Recognition | Dilated CNN with CRF | 90.54% F1 score |
| Semantic role labeling | Bidirectional LSTM with highway connections | 83.40% F1 score |
| Sentiment classification | Tree LSTm with refined word embedding | 90.3% accuracy |
| Machine translation | seq2seq with CNN | 41.29 BLEU score |

Let us now look at the capabilities of various software libraries and platforms available for building NLP systems

**Frameworks and libraries for NLP**

| **Library/Framework** | **Capabilities** |
| --- | --- |
| Natural Language Toolkit (Python) | Provides a suite of open source python modules, data sets, and tutorials. Some of the key features that NLTK provides out of the box include - tokenization, POS tagging, NER, sentiment analysis, corpora (data sets) |
| spaCy | Open source library for advanced NLP tasks written in Python and Cython. Some of the key features include - tokenization, NER, POS tagging, sentiment analysis, dependency parsing, word vectors for deep learning |
| Gensim | Open source vector space and topic modelling toolkit implemented in Python. Gensim is popular for its topic modelling, word vectors and corpora |
| Stanford NLP | Open source NLP package built by the Stanford NLp group. Originally built in Java and now also exposes a Python package. Stanford NLP is typically used to build an NLP pipeline comprising tasks such as tokenization, POS tagging, NER, word vectors for deep learning, and pre trained models for more than 50 languages. |
| Polyglot | A natural language pipeline that supports massive multilingual applications. Offers the following features in multiple languages:  Language detection (196 Languages)   * Tokenization (165 Languages) * Word Embeddings (137 Languages) * Sentiment Analysis (136 Languages) * Morphological analysis (135 Languages) * Transliteration (69 Languages) * Named Entity Recognition (40 Languages) * Part of Speech Tagging (16 Languages) |
| scikit-learn | One of the most popular open source machine learning library in Python. Offers a simple to use API to build and test supervised and unsupervised machine learning models |
| TensorFlow, Keras, PyTorch | Popular deep learning libraries that can be used to train deep neural networks for NLP tasks. Several open source code repositories for common NLP tasks such as sentiment analysis, machine translation, named entity recognition, word vectors etc. have been built using these libraries |

**Platforms for NLP**

In addition to several open source libraries, you can also take advantage of platforms built by several companies like Google, Amazon and Microsoft. Here is a brief description of the platforms that you can use:

**Platforms/Services provided by Amazon over Amazon Web Services**

* Amazon Comprehend: NLP service that sues machine learning to find insights in text. You can perform language detection, named entity recognition, sentiment analysis, POS tagging, and topic modelling using this service
* Amazon Lex: Service for building conversational interfaces using voice or text. Provides automatic speech recognition and natural language understanding capabilities
* Amazon Polly: Service that converts text into life like speech in 25+ languages
* Amazon Transcribe: Service with automatic speech recognition capabilities that converts speech to text. Can transcribe speech in real-time as well.
* Amazon Translate: A neural machine translation service that delivers fast, high quality and affordable machine translation.
* Amazon Textract: A service that automatically extracts text and data from scanned documents

**Platforms/Services provided by Google over Google Cloud Platform**

* Google Cloud Natural Language: Perform syntax analysis,  entity analysis, custom entity extraction, sentiment analysis, content classification, and build machine learning models on cloud
* Google Cloud Translation: Dynamically translate between multiple languages.
* Google Cloud Speech-to-Text API: Performs speech recognition across 120 different languages
* Google Cloud Text-to-Speech API: Synthesize natural sounding speech in 32 different voices across multiple languages and variants
* DialogFlow: Enables the end-to-end development of conversational interfaces (chatbots) across devices and platforms

**Platforms/Services provided by Microsoft over Microsoft Azure**

* Microsoft Cognitive Services - Speech Services: Automatic speech-to-text transcription, Natural text-to-speech, and real time speech translation
* Microsoft Cognitive Services - Text Analytics: Named Entitry Recognition (NER), Key Phrase Extraction, Sentiment Analysis
* Microsoft Cognitive Services - Translator Text: Automatic language detection and text translation
* Microsoft Cognitive Services - QnA Maker: QnA extraction from text, Create knowledge base from set of QnAs, Semantic matching for knowledge bases
* Microsoft Azure Bot Service: End-to-end development of intelligent, enterprise-grade chatbots
* Microsoft Cognitive Services - Language Understanding (LUIS): Build natural language understanding into your apps, bots and IoT devices