What is Natural Language Processing?

Computers and machines are great at working with tabular data or spreadsheets. However, as human beings generally communicate in words and sentences, not in the form of tables. Much information that humans speak or write is unstructured. So it is not very clear for computers to interpret such. In natural language processing (NLP), the goal is to make computers understand the unstructured text and retrieve meaningful pieces of information from it. Natural language Processing (NLP) is a subfield of **artificial intelligence**, in which its depth involves the interactions between computers and humans.

Applications of NLP:

- Machine Translation.
- Speech Recognition.
- Sentiment Analysis.
- Question Answering.
- Summarization of Text.
- Chatbot.
- Intelligent Systems.
- Text Classifications.
- Character Recognition.
- Spell Checking.
- Spam Detection.
- Autocomplete.
- Named Entity Recognition.
- Predictive Typing.

Understanding Natural Language Processing (NLP):

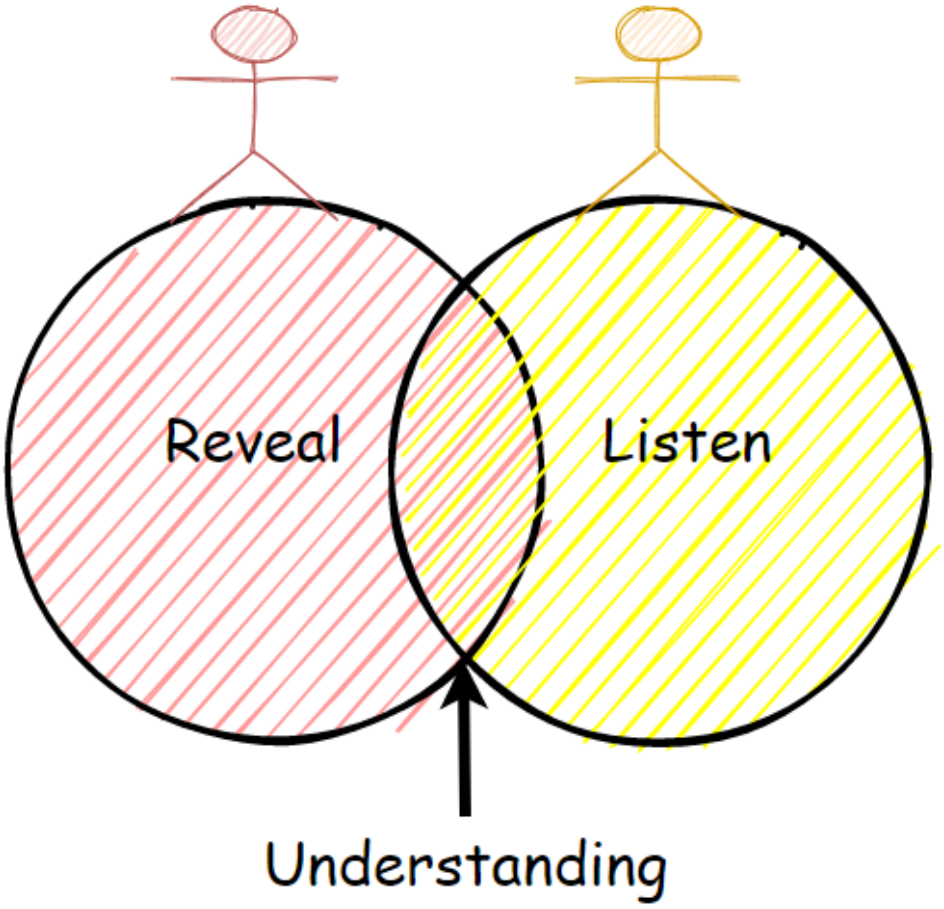


Figure 1: Revealing, listening, and understand.

We, as humans, perform natural language processing (NLP) considerably well, but even then, we are not perfect. We often misunderstand one thing for another, and we often interpret the same sentences or words differently.

For instance, consider the following sentence, we will try to understand its interpretation in many different ways:

Example 1:

```
I saw a man on a hill with a telescope.
```

Figure 2: NLP example sentence with the text: "I saw a man on a hill with a telescope."

These are some interpretations of the sentence shown above.

- There is a man on the hill, and I watched him with my telescope.
- There is a man on the hill, and he has a telescope.
- I'm on a hill, and I saw a man using my telescope.
- I'm on a hill, and I saw a man who has a telescope.
- There is a man on a hill, and I saw him something with my telescope.

Example 2:

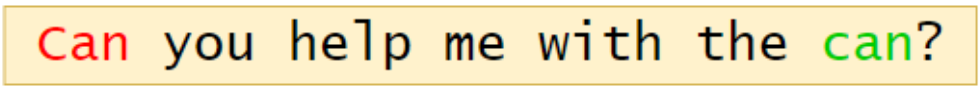


Figure 3: NLP example sentence with the text: "Can you help me with the can?"

In the sentence above, we can see that there are two "can" words, but both of them have different meanings. Here the first "can" word is used for question formation. The second "can" word at the end of the sentence is used to represent a container that holds food or liquid.

Hence, from the examples above, we can see that language processing is not "deterministic" (the same language has the same interpretations), and something suitable to one person might not be suitable to another. Therefore, Natural Language Processing (NLP) has a non-deterministic approach. In other words, Natural Language Processing can be used to create a new intelligent system that can understand how humans understand and interpret language in different situations.

Rule-based NLP vs. Statistical NLP:

Natural Language Processing is separated in two different approaches:

Rule-based Natural Language Processing:

It uses common sense reasoning for processing tasks. For instance, the freezing temperature can lead to death, or hot coffee can burn people's skin, along with other common sense reasoning tasks. However, this process can take much time, and it requires manual effort.

Statistical Natural Language Processing:

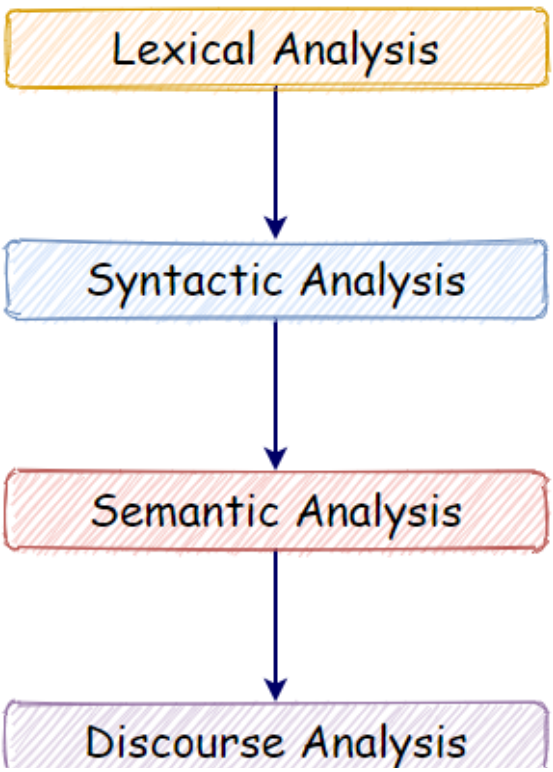
It uses large amounts of data and tries to derive conclusions from it. Statistical NLP uses machine learning algorithms to train NLP models. After successful training on large amounts of data, the trained model will have positive outcomes with deduction.

Comparison:

Rule Based NLP	Statistical NLP
+ 1) Flexible	+ 1) Easy to scale
+ 2) Easy to debug	+ 2) Learn by itself
+ 3) Doesn't require much training	+ 3) Fast development
+ 4) Understanding language	+ 4) High coverage
+ 5) High precision	
- 1) Requires skilled developers	- 1) Requires large amount of data
- 2) Slow parsing	- 2) Difficult to debug
- 3) Moderate coverage	- 3) No understanding of context

Figure 4: Rule-Based NLP vs. Statistical NLP.

Components of Natural Language Processing (NLP):



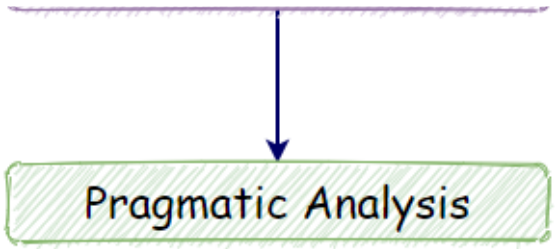


Figure 5: Components of Natural Language Processing (NLP).

a. Lexical Analysis:

With lexical analysis, we divide a whole chunk of text into paragraphs, sentences, and words. It involves identifying and analyzing words' structure.

b. Syntactic Analysis:

Syntactic analysis involves the analysis of words in a sentence for grammar and arranging words in a manner that shows the relationship among the words. For instance, the sentence "The shop goes to the house" does not pass.

c. Semantic Analysis:

Semantic analysis draws the exact meaning for the words, and it analyzes the text meaningfulness. Sentences such as "hot ice-cream" do not pass.

d. Disclosure Integration:

Disclosure integration takes into account the context of the text. It considers the meaning of the sentence before it ends. For example: "He works at Google." In this sentence, "he" must be referenced in the sentence before it.

e. Pragmatic Analysis:

Pragmatic analysis deals with overall communication and interpretation of language. It deals with deriving meaningful use of language in various situations.

> 📚 Check out an overview of machine learning algorithms for beginners with code examples in Python. 📚

Current challenges in NLP:

1. Breaking sentences into tokens.
2. Tagging parts of speech (POS).
3. Building an appropriate vocabulary.
4. Linking the components of a created vocabulary.
5. Understanding the context.
6. Extracting semantic meaning.
7. Named Entity Recognition (NER).
8. Transforming unstructured data into structured data.
9. Ambiguity in speech.

Easy to use NLP libraries:

a. NLTK (Natural Language Toolkit):

The NLTK Python framework is generally used as an education and research tool. It's not usually used on production applications. However, it can be used to build exciting programs due to its ease of use.

Features:

- Tokenization.
- Part Of Speech tagging (POS).
- Named Entity Recognition (NER).
- Classification.
- Sentiment analysis.
- Packages of chatbots.

Use-cases:

- Recommendation systems.

- Sentiment analysis.
- Building chatbots.

Pros	Cons
+ 1) Most well known and full NLP library	- 1) Difficult to learn and use
+ 2) Supports largest numbers of languages	- 2) Context of word is ignored
	- 3) Slow
	- 4) No neural network model

Figure 6: Pros and cons of using the NLTK framework.

b. spaCy:

spaCy is an open-source natural language processing Python library designed to be fast and production-ready. spaCy focuses on providing software for production usage.

Features:

- Tokenization.
- Part Of Speech tagging (POS).
- Named Entity Recognition (NER).
- Classification.
- Sentiment analysis.
- Dependency parsing.
- Word vectors.

Use-cases:

- Autocomplete and autocorrect.
- Analyzing reviews.
- Summarization.

Pros	Cons
+ 1) Fast	- 1) Less flexible
+ 2) Easy to learn and use	
+ 3) Use neural networks for training	

Figure 7: Pros and cons of the spaCy framework.

c. Gensim:

Gensim is an NLP Python framework generally used in topic modeling and similarity detection. It is not a general-purpose NLP library, but it handles tasks assigned to it very well.

Features:

- Latent semantic analysis.
- Non-negative matrix factorization.
- TF-IDF.

Use-cases:

- Converting documents to vectors.
- Finding text similarity.
- Text summarization.

Pros	Cons
+ 1) Intuitive interface	- 1) Designed for unsupervised text models
+ 2) Scalable	- 2) Should be used with other libraries
+ 3) Implemented algorithms	

Figure 8: Pros and cons of the Gensim framework.

d. Pattern:

Pattern is an NLP Python framework with straightforward syntax. It's a powerful tool for scientific and non-scientific tasks. It is highly valuable to students.

Features:

- Tokenization.
- Part of Speech tagging.
- Named entity recognition.
- Parsing.
- Sentiment analysis.

Use-cases:

- Spelling correction.
- Search engine optimization.
- Sentiment analysis.

Pros	Cons
+ 1) Data mining	- 1) Not optimized with specific NLP tasks
+ 2) Network analysis and visualization	

Figure 9: Pros and cons of the Pattern framework.

e. TextBlob:

TextBlob is a Python library designed for processing textual data.

Features:

- Part-of-Speech tagging.
- Noun phrase extraction.
- Sentiment analysis.

- Classification.
- Language translation.
- Parsing.
- Wordnet integration.

Use-cases:

- Sentiment Analysis.
- Spelling Correction.
- Translation and Language Detection.

Pros	Cons
+ 1) Easy to use	- 1) Slow
+ 2) Intuitive interface to NLTK	- 2) No neural network model
+ 3) Provides language translation and detection	- 3) No integrated word vectors

Figure 10: Pros and cons of the TextBlob library.

For this tutorial, we are going to focus more on the NLTK library. Let's dig deeper into natural language processing by making some examples.

Exploring Features of NLTK:

a. Open the text file for processing:

First, we are going to open and read the file which we want to analyze.

```
#Open the text file :
text_file = open("Natural_Language_Processing_Text.txt")

#Read the data :
text = text_file.read()

#Datatype of the data read :
print (type(text))
print("\n")

#Print the text :
```

```
print(text)
print("\n")
#Length of the text :
print (len(text))
```

Figure 11: Small code snippet to open and read the text file and analyze it.

```
<class 'str'>


Once upon a time there was an old mother pig who had three little pigs and not enough food to feed them. So when they were old
enough, she sent them out into the world to seek their fortunes.

The first little pig was very lazy. He didn't want to work at all and he built his house out of straw. The second little pig wo
rked a little bit harder but he was somewhat lazy too and he built his house out of sticks. Then, they sang and danced and play
ed together the rest of the day.

The third little pig worked hard all day and built his house with bricks. It was a sturdy house complete with a fine fireplace
and chimney. It looked like it could withstand the strongest winds.


675
```

Figure 12: Text string file.

Next, notice that the data type of the text file read is a **String**. The number of characters in our text file is **675**.

b. Import required libraries:

For various data processing cases in NLP, we need to import some libraries. In this case, we are going to use NLTK for Natural Language Processing. We will use it to perform various operations on the text.

```
#Import required libraries :
import nltk
from nltk import sent_tokenize
from nltk import word_tokenize
```

Figure 13: Importing the required libraries.

c. Sentence tokenizing:

By tokenizing the text with `sent_tokenize( )` , we can get the text as sentences.

```
#Tokenize the text by sentences :
sentences = sent_tokenize(text)

#How many sentences are there? :
print (len(sentences))

#Print the sentences :
#print(sentences)
```

```
sentences
```

Figure 14: Using sent_tokenize( ) to tokenize the text as sentences.

```
9

['Once upon a time there was an old mother pig who had three little pigs and not enough food to feed them.',
 'So when they were old enough, she sent them out into the world to seek their fortunes.',
 'The first little pig was very lazy.',
 "He didn't want to work at all and he built his house out of straw.",
 'The second little pig worked a little bit harder but he was somewhat lazy too and he built his house out of sticks.',
 'Then, they sang and danced and played together the rest of the day.',
 'The third little pig worked hard all day and built his house with bricks.',
 'It was a sturdy house complete with a fine fireplace and chimney.',
 'It looked like it could withstand the strongest winds.']
```

Figure 15: Text sample data.

In the example above, we can see the entire text of our data is represented as sentences and also notice that the total number of sentences here is **9**.

d. Word tokenizing:

By tokenizing the text with `word_tokenize( )` , we can get the text as words.

```
#Tokenize the text with words :
words = word_tokenize(text)

#How many words are there? :
print (len(words))

#Print words :
print (words)
```

Figure 16: Using word_tokenize() to tokenize the text as words.

```
144
['Once', 'upon', 'a', 'time', 'there', 'was', 'an', 'old', 'mother', 'pig', 'who', 'had', 'three', 'little', 'pigs', 'and', 'no
t', 'enough', 'food', 'to', 'feed', 'them', '.', 'So', 'when', 'they', 'were', 'old', 'enough', ',', 'she', 'sent', 'them', 'ou
t', 'into', 'the', 'world', 'to', 'seek', 'their', 'fortunes', '.', 'The', 'first', 'little', 'pig', 'was', 'very', 'lazy',
'.', 'He', 'did', "n't", 'want', 'to', 'work', 'at', 'all', 'and', 'he', 'built', 'his', 'house', 'out', 'of', 'straw', '.', 'T
he', 'second', 'little', 'pig', 'worked', 'a', 'little', 'bit', 'harder', 'but', 'he', 'was', 'somewhat', 'lazy', 'too', 'and',
'he', 'built', 'his', 'house', 'out', 'of', 'sticks', '.', 'Then', ',', 'they', 'sang', 'and', 'danced', 'and', 'played', 'toge
ther', 'the', 'rest', 'of', 'the', 'day', '.', 'The', 'third', 'little', 'pig', 'worked', 'hard', 'all', 'day', 'and', 'built',
'his', 'house', 'with', 'bricks', '.', 'It', 'was', 'a', 'sturdy', 'house', 'complete', 'with', 'a', 'fine', 'fireplace', 'an
d', 'chimney', '.', 'It', 'looked', 'like', 'it', 'could', 'withstand', 'the', 'strongest', 'winds', '.']
```

Figure 17: Text sample data.

Next, we can see the entire text of our data is represented as words and also notice that the total number of words here is **144**.

e. Find the frequency distribution:

Let's find out the frequency of words in our text.

```
#Import required libraries :
from nltk.probability import FreqDist

#Find the frequency :
fdist = FreqDist(words)

#Print 10 most common words :
fdist.most_common(10)
```

Figure 18: Using FreqDist() to find the frequency of words in our sample text.

```
[('.', 9),
 ('and', 7),
 ('little', 5),
 ('a', 4),
 ('was', 4),
 ('pig', 4),
 ('the', 4),
 ('house', 4),
 ('to', 3),
 ('out', 3)]
```

Figure 19: Printing the ten most common words from the sample text.

Notice that the most used words are punctuation marks and stopwords. We will have to remove such words to analyze the actual text.

f. Plot the frequency graph:

Let's plot a graph to visualize the word distribution in our text.

```
#Plot the graph for fdist :
import matplotlib.pyplot as plt

fdist.plot(10)
```

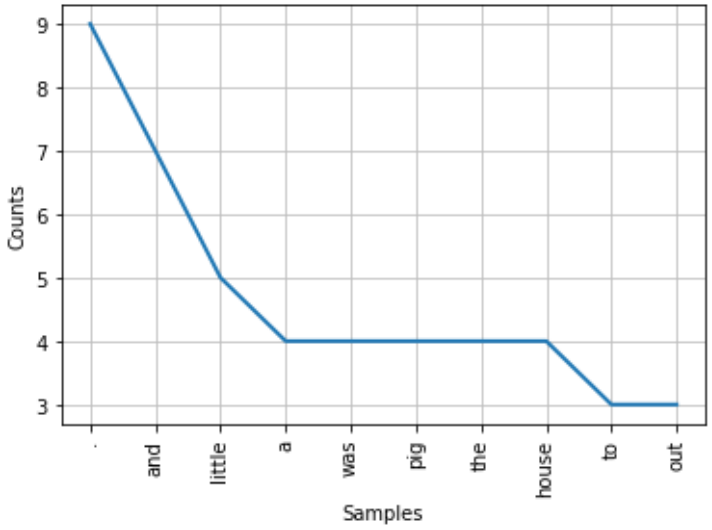


Figure 20: Plotting a graph to visualize the text distribution.

In the graph above, notice that a period "." is used nine times in our text. Analytically speaking, punctuation marks are not that important for natural language processing. Therefore, in the next step, we will be removing such punctuation marks.

g. Remove punctuation marks:

Next, we are going to remove the punctuation marks as they are not very useful for us. We are going to use `isalpha( )` method to separate the punctuation marks from the actual text. Also, we are going to make a new list called `words_no_punc`, which will store the words in lower case but exclude the punctuation marks.

```
#Empty list to store words:
words_no_punc = []

#Removing punctuation marks :
for w in words:
    if w.isalpha():
        words_no_punc.append(w.lower())

#Print the words without punctution marks :
print (words_no_punc)

print ("\n")

#Length :
print (len(words_no_punc))
```

Figure 21: Using the isalpha() method to separate the punctuation marks, along with creating a list under words_no_punc to separate words with no punctuation marks.

```
['once', 'upon', 'a', 'time', 'there', 'was', 'an', 'old', 'mother', 'pig', 'who', 'had', 'three', 'little', 'pigs', 'and', 'no
t', 'enough', 'food', 'to', 'feed', 'them', 'so', 'when', 'they', 'were', 'old', 'enough', 'she', 'sent', 'them', 'out', 'int
o', 'the', 'world', 'to', 'seek', 'their', 'fortunes', 'the', 'first', 'little', 'pig', 'was', 'very', 'lazy', 'he', 'did', 'wa
nt', 'to', 'work', 'at', 'all', 'and', 'he', 'built', 'his', 'house', 'out', 'of', 'straw', 'the', 'second', 'little', 'pig',
'worked', 'a', 'little', 'bit', 'harder', 'but', 'he', 'was', 'somewhat', 'lazy', 'too', 'and', 'he', 'built', 'his', 'house',
'out', 'of', 'sticks', 'then', 'they', 'sang', 'and', 'danced', 'and', 'played', 'together', 'the', 'rest', 'of', 'the', 'day',
'the', 'third', 'little', 'pig', 'worked', 'hard', 'all', 'day', 'and', 'built', 'his', 'house', 'with', 'bricks', 'it', 'was',
'a', 'sturdy', 'house', 'complete', 'with', 'a', 'fine', 'fireplace', 'and', 'chimney', 'it', 'looked', 'like', 'it', 'could',
'withstand', 'the', 'strongest', 'winds']


132
```

Figure 22: Text sample data.

As shown above, all the punctuation marks from our text are excluded. These can also cross-check with the number of words.

h. Plotting graph without punctuation marks:

```
#Frequency distribution :
fdist = FreqDist(words_no_punc)

fdist.most_common(10)
```

```
[('and', 7),
 ('the', 7),
 ('little', 5),
 ('a', 4),
 ('was', 4),
 ('pig', 4),
 ('he', 4),
 ('house', 4),
 ('to', 3),
 ('out', 3)]
```

Figure 23: Printing the ten most common words from the sample text.

```
#Plot the most common words on grpah:

fdist.plot(10)
```

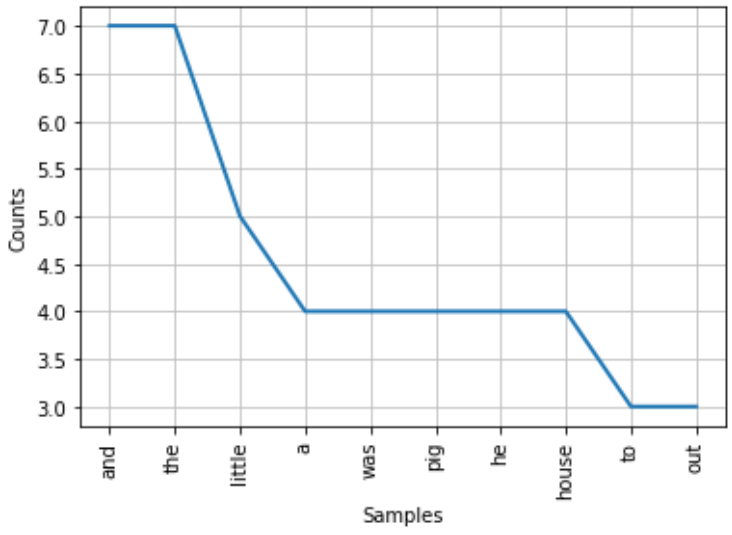


Figure 24: Plotting the graph without punctuation marks.

Notice that we still have many words that are not very useful in the analysis of our text file sample, such as “and,” “but,” “so,” and others. Next, we need to remove coordinating conjunctions.

i. List of stopwords:

```
from nltk.corpus import stopwords

#List of stopwords
stopwords = stopwords.words("english")
print(stopwords)
```

Figure 25: Importing the list of stopwords.

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'y
ourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself',
'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those',
'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'a
n', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'b
etween', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'of
f', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both',
'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very',
's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'ar
en', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "have
n't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should
n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

Figure 26: Text sample data.

j. Removing stopwords:

```
#Empty list to store clean words :
clean_words = []

for w in words_no_punc:
    if w not in stopwords:
        clean_words.append(w)

print(clean_words)
print("\n")
print(len(clean_words))
```

Figure 27: Cleaning the text sample data.

```
['upon', 'time', 'old', 'mother', 'pig', 'three', 'little', 'pigs', 'enough', 'food', 'feed', 'old', 'enough', 'sent', 'world',
'seek', 'fortunes', 'first', 'little', 'pig', 'lazy', 'want', 'work', 'built', 'house', 'straw', 'second', 'little', 'pig', 'wo
rked', 'little', 'bit', 'harder', 'somewhat', 'lazy', 'built', 'house', 'sticks', 'sang', 'danced', 'played', 'together', 'res
t', 'day', 'third', 'little', 'pig', 'worked', 'hard', 'day', 'built', 'house', 'bricks', 'sturdy', 'house', 'complete', 'fin
e', 'fireplace', 'chimney', 'looked', 'like', 'could', 'withstand', 'strongest', 'winds']


65
```

Figure 28: Cleaned data.

k. Final frequency distribution:

```
#Frequency distribution :
fdist = FreqDist(clean_words)

fdist.most_common(10)
```

```
[('little', 5),
 ('pig', 4),
 ('house', 4),
 ('built', 3),
 ('old', 2),
 ('enough', 2),
 ('lazy', 2),
 ('worked', 2),
 ('day', 2),
 ('upon', 1)]
```

Figure 29: Displaying the final frequency distribution of the most common words found.

```
#Plot the most common words on grpah:

fdist.plot(10)
```

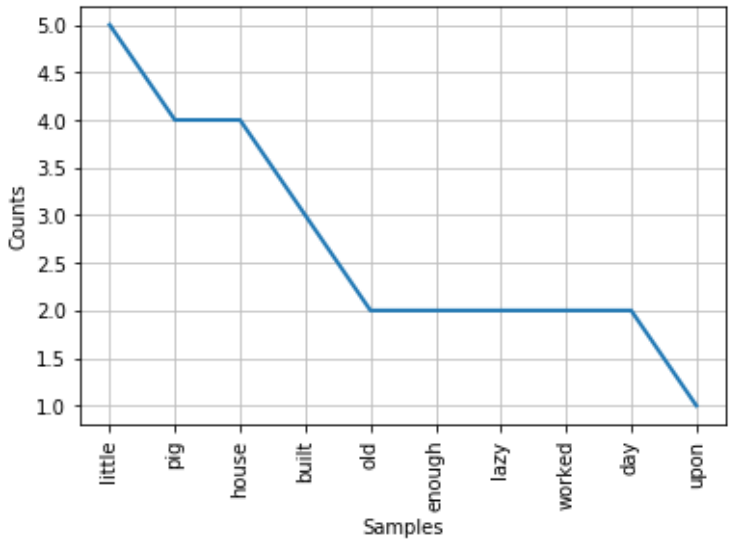


Figure 30: Visualization of the most common words found in the group.

As shown above, the final graph has many useful words that help us understand what our sample data is about, showing how essential it is to perform data cleaning on NLP.

Next, we will cover various topics in NLP with coding examples.

Word Cloud:

Word Cloud is a data visualization technique. In which words from a given text display on the main chart. In this technique, more frequent or essential words display in a larger and bolder font, while less frequent or essential words display in smaller or thinner

fonts. It is a beneficial technique in NLP that gives us a glance at what text should be analyzed.

Properties:

1. **font_path**: It specifies the path for the fonts we want to use.
2. **width**: It specifies the width of the canvas.
3. **height**: It specifies the height of the canvas.
4. **min_font_size**: It specifies the smallest font size to use.
5. **max_font_size:** It specifies the largest font size to use.
6. **font_step**: It specifies the step size for the font.
7. **max_words**: It specifies the maximum number of words on the word cloud.
8. **stopwords**: Our program will eliminate these words.
9. **background_color:** It specifies the background color for canvas.
10. **normalize_plurals**: It removes the trailing "s" from words.

Read the full documentation on WordCloud.

Word Cloud Python Implementation:

```
#Library to form wordcloud :
from wordcloud import WordCloud

#Library to plot the wordcloud :
import matplotlib.pyplot as plt

#Generating the wordcloud :
wordcloud = WordCloud().generate(text)

#Plot the wordcloud :
plt.figure(figsize = (12, 12))
plt.imshow(wordcloud)

#To remove the axis value :
plt.axis("off")
plt.show()
```

Figure 31: Python code implementation of the word cloud.

Figure 32: Word cloud example.

As shown in the graph above, the most frequent words display in larger fonts. The word cloud can be displayed in any shape or image.

For instance: In this case, we are going to use the following circle image, but we can use any shape or any image.

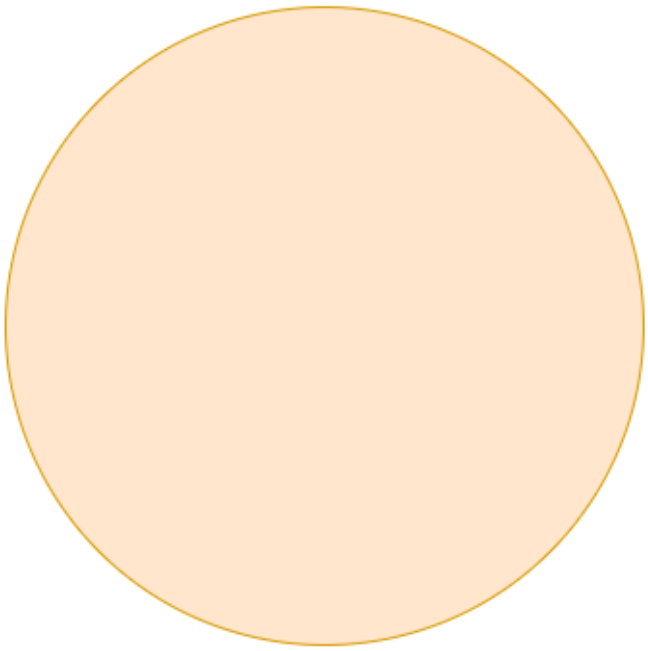


Figure 33: Circle image shape for our word cloud.

Word Cloud Python Implementation:

```
#Import required libraries :
import numpy as np
from PIL import Image
from wordcloud import WordCloud

#Here we are going to use a circle image as mask :
char_mask = np.array(Image.open("circle.png"))

#Generating wordcloud :
wordcloud = WordCloud(background_color="black",mask=char_mask).generate(text)

#Plot the wordcloud :
plt.figure(figsize = (8,8))
plt.imshow(wordcloud)

#To remove the axis value :
plt.axis("off")
plt.show()
```

Figure 34: Python code implementation of the word cloud.

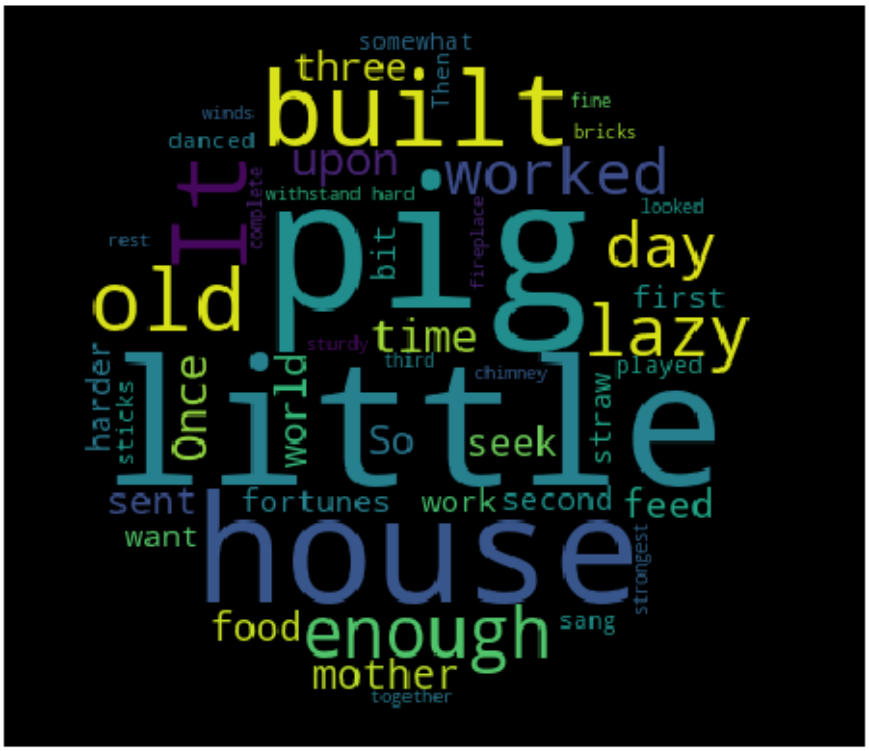


Figure 35: Word cloud with the circle shape.

As shown above, the word cloud is in the shape of a circle. As we mentioned before, we can use any shape or image to form a word cloud.

Word CloudAdvantages:

- They are fast.
- They are engaging.
- They are simple to understand.
- They are casual and visually appealing.

Word Cloud Disadvantages:

- They are non-perfect for non-clean data.
- They lack the context of words.

Stemming:

We use Stemming to normalize words. In English and many other languages, a single word can take multiple forms depending upon context used. For instance, the verb "study" can take many forms like "studies," "studying," "studied," and others, depending on its context. When we tokenize words, an interpreter considers these input words as different words even though their underlying meaning is the same. Moreover, as we know that NLP is about analyzing the meaning of content, to resolve this problem, we use stemming.

Stemming normalizes the word by truncating the word to its stem word. For example, the words "studies," "studied," "studying" will be reduced to **"studi,"** making all these word forms to refer to only one token. Notice that stemming may not give us a dictionary, grammatical word for a particular set of words.

Let's take an example:

a. Porter's Stemmer Example 1:

In the code snippet below, we show that all the words truncate to their stem words. However, notice that the stemmed word is not a dictionary word.

```
#Stemming Example :

#Import stemming library :
from nltk.stem import PorterStemmer

porter = PorterStemmer()
```

```
#Word-list for stemming :
word_list = ["Study","Studying","Studies","Studied"]

for w in word_list:
    print(porter.stem(w))
```

```
studi
studi
studi
studi
```

Figure 36: Code snippet showing a stemming example.

b. Porter's Stemmer Example 2:

In the code snippet below, many of the words after stemming did not end up being a recognizable dictionary word.

```
#Stemming Example :

#Import stemming library :
from nltk.stem import PorterStemmer

porter = PorterStemmer()

#Word-list for stemming :
word_list = ["studies","leaves","decreases","plays"]

for w in word_list:
    print(porter.stem(w))
```

```
studi
leav
decreas
play
```

Figure 37: Code snippet showing a stemming example.

c. SnowballStemmer:

SnowballStemmer generates the same output as porter stemmer, but it supports many more languages.

```
#Stemming Example :

#Import stemming library :
from nltk.stem import SnowballStemmer

snowball = SnowballStemmer("english")

#Word-list for stemming :
word_list = ["Study","Studying","Studies","Studied"]
```

```
for w in word_list:
    print(snowball.stem(w))
```

```
studi
studi
studi
studi
```

Figure 38: Code snippet showing an NLP stemming example.

d. Languages supported by snowball stemmer:

```
#Stemming Example :

#Import stemming library :
from nltk.stem import SnowballStemmer

#Print Languages supported :
SnowballStemmer.languages
```

```
('arabic',
 'danish',
 'dutch',
 'english',
 'finnish',
 'french',
 'german',
 'hungarian',
 'italian',
 'norwegian',
 'porter',
 'portuguese',
 'romanian',
 'russian',
 'spanish',
 'swedish')
```

Figure 39: Code snippet showing an NLP stemming example.

Various Stemming Algorithms:

a. Porter's Stemmer:

Porter's Stemmer
Advantage
It produces the best output and it has the lowest error rate.
Limitation

The reduced words are not always real words.

Figure 40: Porter's Stemmer NLP algorithm, pros, and cons.

b. Lovin's Stemmer:

Lovin's Stemmer
Advantage
It's fast and it can handle irregular plurals like foot → feet.
Limitation
It's very time consuming and it has a higher error rate.

Figure 41: Lovin's Stemmer NLP algorithm, pros, and cons.

c. Dawson's Stemmer:

Dawson's Stemmer
Advantage
It's fast and covers more suffixes.
Limitation
It's a bit complex to implement.

Figure 42: Dawson's Stemmer NLP algorithm, pros, and cons.

d. Krovetz Stemmer:

Krovetz Stemmer
Advantage
It's light in nature and used as a pre-stemmer to other stemmers.
Limitation
It is inefficient incase of larger documents.

Figure 43: Krovetz Stemmer NLP algorithm, pros, and cons.

e. Xerox Stemmer:

Xerox Stemmer
Advantage
It works fine with larger documents and the error rate is low.
Limitation
It may result in overstemming and it's language dependent.

Figure 44: Xerox Stemmer NLP algorithm, pros, and cons.

f. Snowball Stemmer:

Snowball Stemmer
Advantage
It supports more languages.
Limitation
The reduced words are not always real words.

Figure 45: Snowball Stemmer NLP algorithm, pros, and cons.

Lemmatization:

Lemmatization tries to achieve a similar base "stem" for a word. However, what makes it different is that it finds the dictionary word instead of truncating the original word. Stemming does not consider the context of the word. That is why it generates results faster, but it is less accurate than lemmatization.

If accuracy is not the project's final goal, then stemming is an appropriate approach. If higher accuracy is crucial and the project is not on a tight deadline, then the best option is amortization (Lemmatization has a lower processing speed, compared to stemming).

Lemmatization takes into account Part Of Speech (POS) values. Also, lemmatization may generate different outputs for different values of POS. We generally have four choices for POS:

Verb (v)
Examples : Study, Play, Learn, Am, Is, Are...
Noun (n)
Examples : Doctor, Engineer, Farm, Physiotherapist,Towards AI...
Adjective (a)
Examples : Beautiful, Elegant, Angry, Polite, Repulsive...
Adverb (r)
Examples : Badly, Slowly, Peacefully, Very, Extremely, Occasionally...

Figure 46: Part of Speech (POS) values in lemmatization.

Difference between Stemmer and Lemmatizer:

a. Stemming:

Notice how on stemming, the word “studies” gets truncated to “studi.”

```
from nltk.stem import PorterStemmer

stemmer = PorterStemmer()

print(stemmer.stem('studies'))
```

```
studi
```

Figure 47: Using stemming with the NLTK Python framework.

b. Lemmatizing:

During lemmatization, the word “studies” displays its dictionary word “study.”

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()

print(lemmatizer.lemmatize('studies'))
```

```
study
```

Figure 48: Using lemmatization with the NLTK Python framework.

Python Implementation:

a. A basic example demonstrating how a lemmatizer works

In the following example, we are taking the PoS tag as “verb,” and when we apply the lemmatization rules, it gives us dictionary words instead of truncating the original word:

```
from nltk import WordNetLemmatizer

lemma = WordNetLemmatizer()
word_list = ["Study","Studying","Studies","Studied"]

for w in word_list:
    print(lemma.lemmatize(w ,pos="v"))
```

```
Study
Studying
Studies
Studied
```

Figure 49: Simple lemmatization example with the NLTK framework.

b. Lemmatizer with default PoS value

The default value of PoS in lemmatization is a noun(n). In the following example, we can see that it’s generating dictionary words:

```
from nltk import WordNetLemmatizer

lemma = WordNetLemmatizer()
word_list = ["studies","leaves","decreases","plays"]

for w in word_list:
    print(lemma.lemmatize(w))
```

```
study
leaf
decrease
play
```

Figure 50: Using lemmatization to generate default values.

c. Another example demonstrating the power of lemmatizer

```
from nltk import WordNetLemmatizer

lemma = WordNetLemmatizer()
word_list = ["am","is","are","was","were"]
```

```
for w in word_list:
    print(lemma.lemmatize(w ,pos="v"))
```

```
be
be
be
be
be
```

Figure 51: Lemmatization of the words: ”am”, ”are”, ”is”, ”was”, ”were”

d. Lemmatizer with different POS values

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
print(lemmatizer.lemmatize('studying', pos="v"))
print(lemmatizer.lemmatize('studying', pos="n"))
print(lemmatizer.lemmatize('studying', pos="a"))
print(lemmatizer.lemmatize('studying', pos="r"))
```

```
study
studying
studying
studying
```

Figure 52: Lemmatization with different Part-of-Speech values.

Part of Speech Tagging (PoS tagging):

Why do we need Part of Speech (POS)?

Can you help me with the can?

Figure 53: Sentence example, ”can you help me with the can?”

Parts of speech(PoS) tagging is crucial for syntactic and semantic analysis. Therefore, for something like the sentence above, the word “can” has several semantic meanings. The first “can” is used for question formation. The second “can” at the end of the sentence is used to represent a container. The first “can” is a verb, and the second “can” is a noun. Giving the word a specific meaning allows the program to handle it correctly in both semantic and syntactic analysis.

Below, please find a list of Part of Speech (PoS) tags with their respective examples:

1. CC: Coordinating Conjunction

(CC) Coordinating Conjunction
Words
And, But, For, Nor, Or, So, Yet...
Example
You can eat your cake with a spoon or fork.

Figure 54: Coordinating conjunction example.

2. CD: Cardinal Digit

(CD) Cardinal Digit
Words
One, Two, Three, Four, Five, Six...
Example
There are five flowers in the flower vase.

Figure 55: Cardinal digit example.

3. DT: Determiner

(DT) Determiner
Definite Article
The
Indefinite Article
A, An
Demonstratives
This, That, These, Those
Pronouns and Possessive Determiners
My, Your, His, Her, It's, Our, Their
Quantifiers
Any, A few, A little, Much, Many, Most, Some
Numbers
One, Ten, Thirty

One, Ten, Thirty
Distributives
All, Both, Half, Either, Neither, Each, Every
Difference Words
Other, Another
Pre-determiners
Such, What, Rather, Quite

Figure 56: A determiner example.

4. EX: Existential There

(EX) Existential There
Words
There
Example
There is a pen on the desk.

Figure 57: Existential "there" example.

5. FW: Foreign Word

(FW) Foreign Words
Words
dolce, ersatz, esprit, quo, maitre
Example
They were selling ersatz products.

Figure 58: Foreign word example.

6. IN: Preposition / Subordinating Conjunction

(IN) Preposition / Subordinating Conjunction
Words
At, In, On, Above, Behind
Example
All speak at the same time.

Figure 59: Preposition/Subordinating conjunction.

7. JJ: Adjective

(JJ) Adjective
Words
New, Old, High, Special, Big, Local
Example
They live in a beautiful house.

Figure 60: Adjective example.

8. JJR: Adjective, Comparative

(JJR) Adjective, Comparative
Words
Newer, Bigger, Higher
Example
The new bird is angrier than Robin.

Figure 61: Adjective, comparative example.

9. JJS: Adjective, Superlative

(JJR) Adjective, Superlative

Words
Biggest, Darkest, Most
Example
Jupyter is the biggest planet in our solar system.

Figure 62:

10. LS: List Marker

(LS) List Marker
Words
1), 2), 3)...
Example
1) One 2) Two 3) Three

Figure 63: List marker example.

11. MD: Modal

(MD) Modal
Words
Could, Will, Should
Example
Should we go outside in this situation?

Figure 64:

12. NN: Noun, Singular

(NN) Noun , Singular
Words

Year, Home, Cost, Time, Education
Example
Why don't we stop by my house for coffee?

Figure 65: Noun, singular example.

13. NNS: Noun, Plural

(NNS) Noun , Plural
Words
Pens, Books, Hats
Example
There are a bunch of pens on the table.

Figure 66: Noun, plural example.

14. NNP: Proper Noun, Singular

(NNP) Proper Noun , Singular
Words
April, Africa, Pratik
Example
April is one of the hottest months.

Figure 67: Proper noun, singular example.

15. NNPS: Proper Noun, Plural

(NNPS) Proper Noun , Plural
Words
Americans, Indians
Example
Most Americans are young at heart.

Figure 68: Proper noun, plural example.

16. PDT: Predeterminer

(PDT) Predeterminer
Words
All, Both, Half
Example
They all were accepted.

Figure 69: Predeterminer example.

17. POS: Possessive Endings

(POS) Possessive Endings
Words
Parent's, Pratik's
Example
It's Jan's assignment.

Figure 70: Possessive endings example.

18. PRP: Personal Pronoun

(PRP) Personal Pronoun
Words
I, He, She
Example
She is very brilliant.

Figure 71: Personal pronoun example.

19. PRP$: Possessive Pronoun

(PRP$) Possessive Pronoun
Words
My, His, Her, Hers, Yours
Example
This is my book.

Figure 72: Possessive pronoun example.

20. RB: Adverb

(RB) Adverb
Words
Really, Already, Still, Early, Now
Example
We are still waiting for them.

Figure 73: Adverb example.

21. RBR: Adverb, Comparative

(RBR) Adverb, Comparative
Words
Worse, Less, Better
Example
This is way better than we anticipated.

Figure 74: Adverb, comparative example.

22. RBS: Adverb, Superlative

(RBS) Adverb, Superlative
Words
Worst, Least, Best

Example
This is the best day ever.

Figure 75: Adverb, superlative example.

23. RP: Particle

(RP) Adverb, Superlative
Words
Aboard, About, Across, Along, At, Away
Example
They were about to leave.

Figure 76: Particle example.

24. TO: To

(TO) To
Words
To
Example
We were going to the store.

Figure 77: To example.

25. UH: Interjection

(UH) Interjection
Words
Amen, Huh, Howdy, Dammit, Heck, Anyways
Example
What the heck!

Figure 78: Interjection example.

26. VB: Verb, Base Form

(VB) Verb, Base Form
Words
Take, Play, Sit, Listen
Example
Let us play a game.

Figure 79: Verb, base form example.

27. VBD: Verb, Past Tense

(VBD) Verb, Past Tense
Words
Took, Studied, Played
Example
We studied for an hour.

Figure 80: Verb, past tense example.

28. VBG: Verb, Present Participle

(VBG) Verb, Present Participle
Words
Playing, Studying
Example
They have been playing since morning.

Figure 81: Verb, present participle example.

29. VBN: Verb, Past Participle

(VBN) Verb, Past Participle
Words

Taken, Languished, Dilapidated
Example
They were taken hostages.

Figure 82: Verb, past participle.

30. VBP: Verb, Present Tense, Not Third Person Singular

(VBP) Verb, Present Tense, Not 3rd Person
Words
Terminate, Appear, Cure
Example
We are going to terminate the contract.

Figure 83: Verb, present tense, not third-person singular.

31. VBZ: Verb, Present Tense, Third Person Singular

(VBP) Verb, Present Tense, 3rd Person
Words
Uses, Speaks, Slaps
Example
He generally speaks about sports.

Figure 84: Verb, present tense, third-person singular.

32. WDT: Wh — Determiner

(WDT) Wh - Determiner
Words
That, What, Whatever, Which, Whichever
Example
What is your name?

Figure 85: Determiner example.

33. WP: Wh — Pronoun

(WP) Wh - Pronoun
Words
Who, Whom, Which, What
Example
Who is that guy in the limo?

Figure 86: Pronoun example.

34. WP$: Possessive Wh — Pronoun

(WP$) Possessive Wh - Pronoun
Words
Whose
Example
Whose pen is this?

Figure 87: Possessive pronoun example.

35. WRB: Wh — Adverb

(WRB) Wh - Adverb
Words
How, However, Why, Where, When
Example
How is this possible?

Figure 88: Adverb example.

Python Implementation:

a. A simple example demonstrating PoS tagging.

```
#PoS tagging :
tag = nltk.pos_tag(["Studying","Study"])
print (tag)
```

```
[('Studying', 'VBG'), ('Study', 'NN')]
```

Figure 89: PoS tagging example.

b. A full example demonstrating the use of PoS tagging.

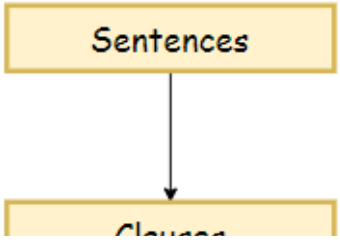
```
#PoS tagging example :

sentence = "A very beautiful young lady is walking on the beach"

#Tokenizing words :
tokenized_words = word_tokenize(sentence)

for words in tokenized_words:
    tagged_words = nltk.pos_tag(tokenized_words)

tagged_words
```

```
[('A', 'DT'),
 ('very', 'RB'),
 ('beautiful', 'JJ'),
 ('young', 'JJ'),
 ('lady', 'NN'),
 ('is', 'VBZ'),
 ('walking', 'VBG'),
 ('on', 'IN'),
 ('the', 'DT'),
 ('beach', 'NN')]
```

Figure 90: Full Python sample demonstrating PoS tagging.

Chunking:

Chunking means to extract meaningful phrases from unstructured text. By tokenizing a book into words, it's sometimes hard to infer meaningful information. It works on top of Part of Speech(PoS) tagging. Chunking takes PoS tags as input and provides chunks as output. Chunking literally means a group of words, which breaks simple text into phrases that are more meaningful than individual words.



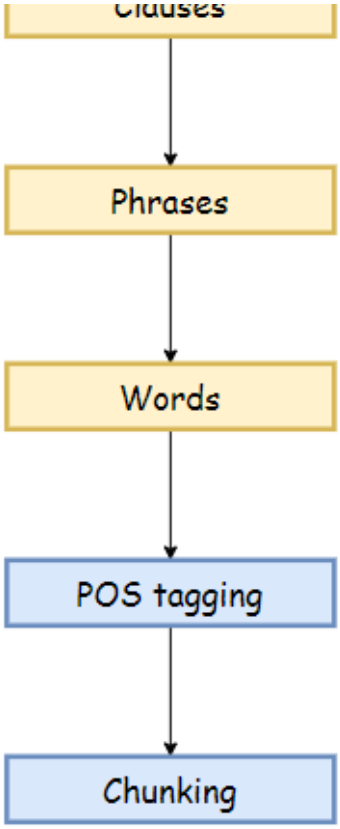


Figure 91: The chunking process in NLP.

Before working with an example, we need to know what phrases are? Meaningful groups of words are called phrases. There are five significant categories of phrases.

1. Noun Phrases (NP).
2. Verb Phrases (VP).
3. Adjective Phrases (ADJP).
4. Adverb Phrases (ADVP).
5. Prepositional Phrases (PP).

Phrase structure rules:

- S(Sentence) → NP VP.
- NP → {Determiner, Noun, Pronoun, Proper name}.
- VP → V (NP) (PP) (Adverb).
- PP → Pronoun (NP).

- AP → Adjective (PP).

Example:

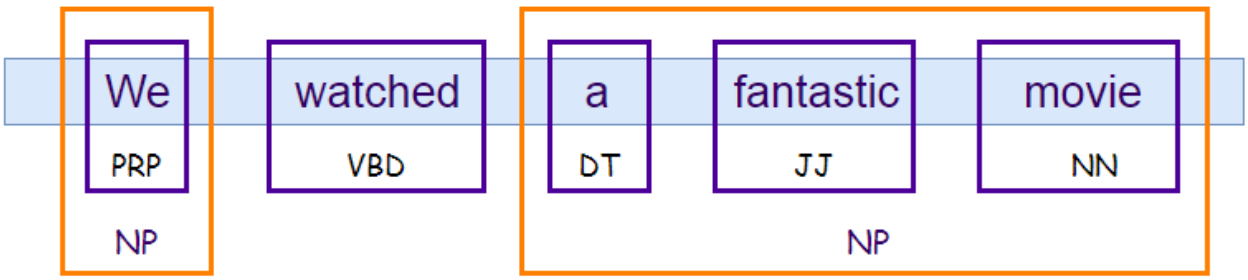


Figure 92: A chunking example in NLP.

Python Implementation:

In the following example, we will extract a noun phrase from the text. Before extracting it, we need to define what kind of noun phrase we are looking for, or in other words, we have to set the grammar for a noun phrase. In this case, we define a noun phrase by an optional determiner followed by adjectives and nouns. Then we can define other rules to extract some other phrases. Next, we are going to use `RegexpParser( )` to parse the grammar. Notice that we can also visualize the text with the `.draw( )` function.

```
#Extracting Noun Phrase from text :

# ? - optional character
# * - 0 or more repetations
grammar = "NP : {<DT>?<JJ>*<NN>} "

#Creating a parser :
parser = nltk.RegexpParser(grammar)

#Parsing text :
output = parser.parse(tagged_words)
print (output)

#To visualize :
output.draw()
```

```
(S
  A/DT
  very/RB
  (NP beautiful/JJ young/JJ lady/NN)
  is/VBZ
  walking/VBG
  on/IN
  (NP the/DT beach/NN))
```

Figure 93: Code snippet to extract noun phrases from a text file.

In this example, we can see that we have successfully extracted the noun phrase from the text.

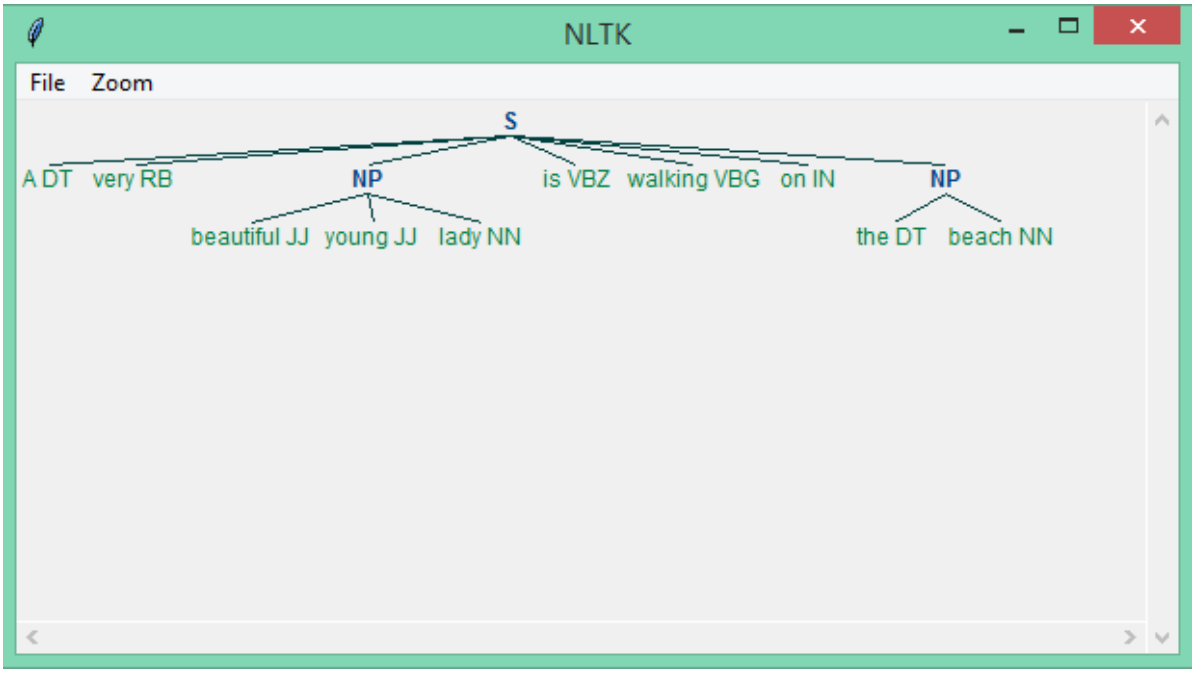


Figure 94: Successful extraction of the noun phrase from the input text.

Chinking:

Chinking excludes a part from our chunk. There are certain situations where we need to exclude a part of the text from the whole text or chunk. In complex extractions, it is possible that chunking can output unuseful data. In such case scenarios, we can use chinking to exclude some parts from that chunked text.
In the following example, we are going to take the whole string as a chunk, and then we are going to exclude adjectives from it by using chinking. We generally use chinking when we have a lot of unuseful data even after chunking. Hence, by using this method, we can easily set that apart, also to write chinking grammar, we have to use inverted curly braces, i.e.:

> } write chinking grammar here {

Python Implementation:

```
#Chinking example :
```

```
# * - 0 or more repetations
# + - 1 or more repetations

#Here we are taking the whole string and then
#excluding adjectives from that chunk.

grammar = r""" NP: {<.*>+}
               }<JJ>+{"""

#Creating parser :
parser = nltk.RegexpParser(grammar)

#parsing string :
output = parser.parse(tagged_words)
print(output)

#To visualize :
output.draw()
```

```
(S
  (NP A/DT very/RB)
  beautiful/JJ
  young/JJ
  (NP lady/NN is/VBZ walking/VBG on/IN the/DT beach/NN))
```

Figure 95: Chinking implementation with Python.

From the example above, we can see that adjectives separate from the other text.

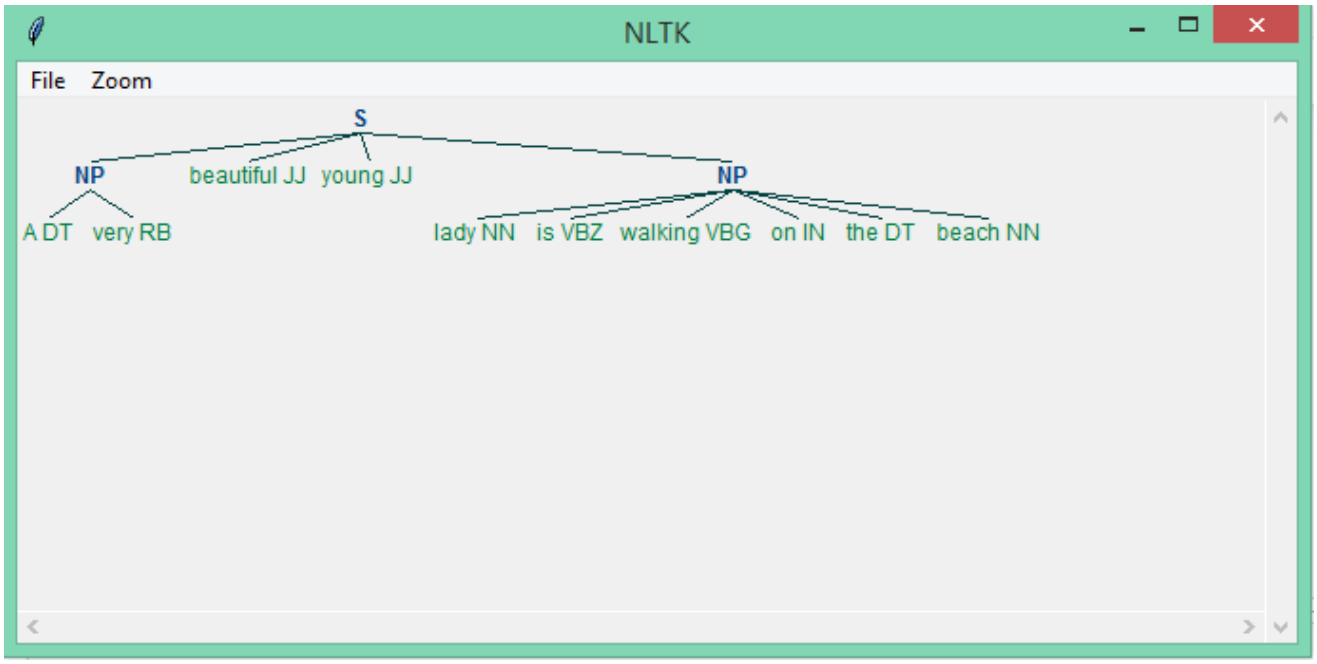


Figure 96: In this example, adjectives are excluded by using chinking.

Named Entity Recognition (NER):

Named entity recognition can automatically scan entire articles and pull out some fundamental entities like people, organizations, places, date, time, money, and GPE discussed in them.

Use-Cases:

1. Content classification for news channels.
2. Summarizing resumes.
3. Optimizing search engine algorithms.
4. Recommendation systems.
5. Customer support.

Commonly used types of named entity:

Named Entity Type	Example
ORGANIZATION	WHO
PERSON	President Obama
LOCATION	Mount Everest
DATE	2020-07-10
TIME	12:50 P.M.
MONEY	One Million Dollars
PERCENT	98.24%
FACILITY	Washington Monument
GPE	North West America

Figure 97: An example of commonly used types of named entity recognition (NER).

Python Implementation:

There are two options :

1. binary = True

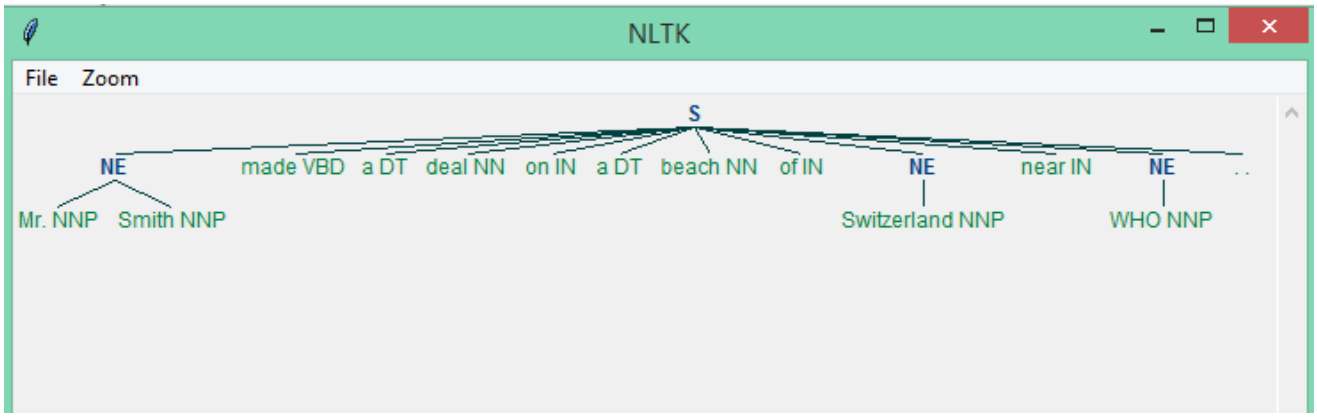
When the binary value is True, then it will only show whether a particular entity is named entity or not. It will not show any further details on it.

```
#Sentence for NER :
sentence = "Mr. Smith made a deal on a beach of Switzerland near WHO."

#Tokenizing words :
tokenized_words = word_tokenize(sentence)

#PoS tagging :
for w in tokenized_words:
    tagged_words = nltk.pos_tag(t_w)

#print (tagged_words)

#Named Entity Recognition :
N_E_R = nltk.ne_chunk(tagged_words,binary=True)

print(N_E_R)

#To visualize :
N_E_R.draw()
```

```
(S
  (NE Mr./NNP Smith/NNP)
  made/VBD
  a/DT
  deal/NN
  on/IN
  a/DT
  beach/NN
  of/IN
  (NE Switzerland/NNP)
  near/IN
  (NE WHO/NNP)
  ./.)
```

Figure 98: Python implementation when a binary value is True.

Our graph does not show what type of named entity it is. It only shows whether a particular word is named entity or not.



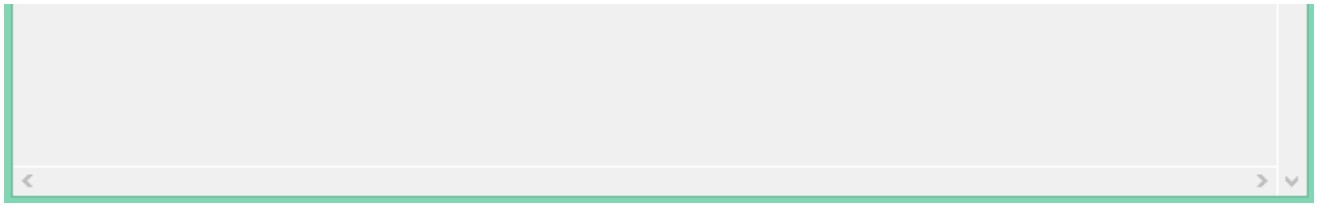


Figure 99: Graph example of when a binary value is True.

2. binary = False

When the binary value equals False, it shows in detail the type of named entities.

```
#Sentence for NER :
sentence = "Mr. Smith made a deal on a beach of Switzerland near WHO."

#Tokenizing words :
tokenized_words = word_tokenize(sentence)

#PoS tagging :
for w in tokenized_words:
    tagged_words = nltk.pos_tag(t_w)

#print (tagged_words)

#Named Entity Recognition :
N_E_R = nltk.ne_chunk(tagged_words,binary=False)
print(N_E_R)

#To visualize :
N_E_R.draw()
```

```
(S
  (PERSON Mr./NNP)
  (PERSON Smith/NNP)
  made/VBD
  a/DT
  deal/NN
  on/IN
  a/DT
  beach/NN
  of/IN
  (GPE Switzerland/NNP)
  near/IN
  (ORGANIZATION WHO/NNP)
  ./.)
```

Figure 100: Python implementation when a binary value is False.

Our graph now shows what type of named entity it is.

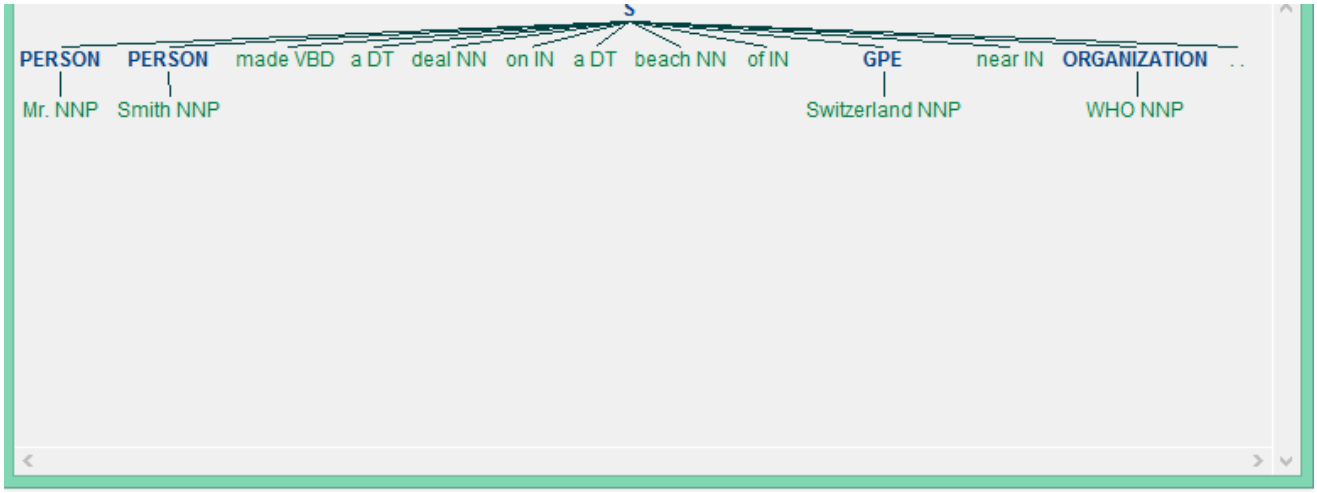


Figure 101: Graph showing the type of named entities when a binary value equals false.

WordNet:

Wordnet is a lexical database for the English language. Wordnet is a part of the NLTK corpus. We can use Wordnet to find meanings of words, synonyms, antonyms, and many other words.

a. We can check how many different definitions of a word are available in Wordnet.

```
from nltk.corpus import wordnet

for words in wordnet.synsets("Fun"):
    print(words)
```

```
Synset('fun.n.01')
Synset('fun.n.02')
Synset('fun.n.03')
Synset('playfulness.n.02')
```

Figure 102: Checking word definitions with Wordnet using the NLTK framework.

b. We can also check the meaning of those different definitions.

```
#How many differnt meanings :
for words in wordnet.synsets("Fun"):
    for lemma in words.lemmas():
        print(lemma)
    print("\n")
```

```
Lemma('fun.n.01.fun')
```

```
Lemma('fun.n.01.merriment')
Lemma('fun.n.01.playfulness')


Lemma('fun.n.02.fun')
Lemma('fun.n.02.play')
Lemma('fun.n.02.sport')


Lemma('fun.n.03.fun')


Lemma('playfulness.n.02.playfulness')
Lemma('playfulness.n.02.fun')
```

Figure 103: Gathering the meaning of the different definitions by using Wordnet.

c. All details for a word.

```
word = wordnet.synsets("Play")[0]

#Checking name :
print(word.name())

#Checking definition :
print(word.definition())

#Checking examples:
print(word.examples())
```

```
play.n.01
a dramatic work intended for performance by actors on a stage
['he wrote several plays but only one was produced on Broadway']
```

Figure: 104: Finding all the details for a specific word.

d. All details for all meanings of a word.

```
#Word meaning with definitions :
for words in wordnet.synsets("Fun"):
    print(words.name())
    print(words.definition())
    print(words.examples())

    for lemma in words.lemmas():
        print(lemma)
    print("\n")
```

```
fun.n.01
activities that are enjoyable or amusing
['I do it for the fun of it', 'he is fun to have around']
Lemma('fun.n.01.fun')
Lemma('fun.n.01.merriment')
Lemma('fun.n.01.playfulness')
```

```
Lemma('fun.n.01.playfulness')


fun.n.02
verbal wit or mockery (often at another's expense but not to be taken seriously)
['he became a figure of fun', 'he said it in sport']
Lemma('fun.n.02.fun')
Lemma('fun.n.02.play')
Lemma('fun.n.02.sport')


fun.n.03
violent and excited activity
['she asked for money and then the fun began', 'they began to fight like fun']
Lemma('fun.n.03.fun')


playfulness.n.02
a disposition to find (or make) causes for amusement
['her playfulness surprised me', 'he was fun to be with']
Lemma('playfulness.n.02.playfulness')
Lemma('playfulness.n.02.fun')
```

Figure 105: Finding all details for all the meanings of a specific word.

e. Hypernyms: Hypernyms gives us a more abstract term for a word.

```
word = wordnet.synsets("Play")[0]

#Find more abstract term :
print(word.hypernyms())
```

```
[Synset('dramatic_composition.n.01')]
```

Figure 106: Using Wordnet to find a hypernym.

f. Hyponyms: Hyponyms gives us a more specific term for a word.

```
word = wordnet.synsets("Play")[0]

#Find more specific term :
word.hyponyms()
```

```
[Synset('grand_guignol.n.01'),
 Synset('miracle_play.n.01'),
 Synset('morality_play.n.01'),
 Synset('mystery_play.n.01'),
 Synset('passion_play.n.01'),
 Synset('playlet.n.01'),
 Synset('satyr_play.n.01'),
 Synset('theater_of_the_absurd.n.01')]
```

Figure 107: Using Wordnet to find a hyponym.

g. Get a name only.

```
word = wordnet.synsets("Play")[0]

#Get only name :
print(word.lemmas()[0].name())
```

```
play
```

Figure 108: Finding only a name with Wordnet.

h. Synonyms.

```
#Finding synonyms :

#Empty list to store synonyms :
synonyms = []

for words in wordnet.synsets('Fun'):
    for lemma in words.lemmas():
        synonyms.append(lemma.name())

synonyms
```

```
['fun',
 'merriment',
 'playfulness',
 'fun',
 'play',
 'sport',
 'fun',
 'playfulness',
 'fun']
```

Figure 109: Finding synonyms with Wordnet.

i. Antonyms.

```
#Finding antonyms :

#Empty list to store antonyms :
antonyms = []

for words in wordnet.synsets('Natural'):
    for lemma in words.lemmas():
        if lemma.antonyms():
            antonyms.append(lemma.antonyms()[0].name())

#Print antonyms :
antonyms
```

```
['unnatural', 'artificial', 'supernatural', 'sharp']
```

Figure 110: Finding antonyms with Wordnet.

j. Synonyms and antonyms.

```
#Finding synonyms and antonyms :

#Empty lists to store synonyms/antonynms :
synonyms = []
antonyms = []

for words in wordnet.synsets('New'):
    for lemma in words.lemmas():
        synonyms.append(lemma.name())
        if lemma.antonyms():
            antonyms.append(lemma.antonyms()[0].name())

#Print lists :
print(synonyms)
print("\n")
print(antonyms)
```

```
['new', 'fresh', 'new', 'novel', 'raw', 'new', 'new', 'unexampled', 'new', 'new', 'newfangled', 'new', 'New', 'Modern', 'New',
'new', 'young', 'new', 'newly', 'freshly', 'fresh', 'new']


['old', 'worn']
```

Figure 111: Finding synonyms and antonyms code snippet with Wordnet.

k. Finding the similarity between words.

```
#Similarity in words :
word1 = wordnet.synsets("ship","n")[0]

word2 = wordnet.synsets("boat","n")[0]

#Check similarity :
print(word1.wup_similarity(word2))
```

```
0.9090909090909091
```

Figure 112: Finding the similarity ratio between words using Wordnet.

```
#Similarity in words :
word1 = wordnet.synsets("ship","n")[0]

word2 = wordnet.synsets("bike","n")[0]

#Check similarity :
print(word1.wup_similarity(word2))
```

```
0.6956521739130435
```

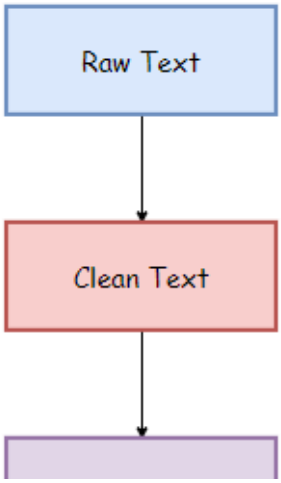
Figure 113: Finding the similarity ratio between words using Wordnet.

Bag of Words:

Figure 114: A representation of a bag of words.

What is the Bag-of-Words method?

It is a method of extracting essential features from row text so that we can use it for machine learning models. We call it **"Bag"** of words because we discard the order of occurrences of words. A bag of words model converts the raw text into words, and it also counts the frequency for the words in the text. In summary, a bag of words is a collection of words that represent a sentence along with the word count where the order of occurrences is not relevant.



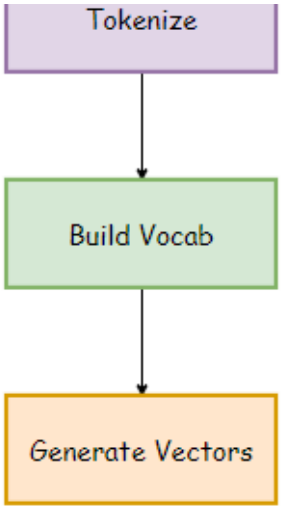


Figure 115: Structure of a bag of words.

1. **Raw Text:** This is the original text on which we want to perform analysis.
2. **Clean Text:** Since our raw text contains some unnecessary data like punctuation marks and stopwords, so we need to clean up our text. Clean text is the text after removing such words.
3. **Tokenize:** Tokenization represents the sentence as a group of tokens or words.
4. **Building Vocab:** It contains total words used in the text after removing unnecessary data.
5. **Generate Vocab:** It contains the words along with their frequencies in the sentences.

For instance:

Sentences:

1. Jim and Pam traveled by bus.
2. The train was late.
3. The flight was full. Traveling by flight is expensive.

a. Creating a basic structure:

Sentence 1	Sentence2	Sentence 3

Jim	The	The
and	train	flight
Pam	was	was
travelled	late	full
by		Travelling
the		by
bus		flight
		is
		expensive

Figure 116: Example of a basic structure for a bag of words.

b. Words with frequencies:

Sentence1	Count	Sentence2	Count	Sentence3	Count
Jim	1	The	1	The	1
and	1	train	1	flight	2
Pam	1	was	1	was	1
travelled	1	late	1	full	1
by	1			Travelling	1
the	1			by	1
bus	1			is	1
				expensive	1

Figure 117: Example of a basic structure for words with frequencies.

c. Combining all the words:

Sentence	Frequency
and	1
bus	1
by	2
expensive	1

flight	2
full	1
is	1
jim	1
late	1
pam	1
the	3
train	1
travelled	1
travelling	1
was	1

Figure 118: Combination of all the input words.

d. Final model:

	and	bus	by	expensive	flight	full	is	jim	Late	pam	The	train	travelled	travelling	was
S-1	1	1	1	0	0	0	0	1	0	1	1	0	1	0	0
S-2	0	0	0	0	0	0	0	0	1	0	1	1	0	0	1
S-3	0	0	1	1	2	1	1	0	0	0	1	0	0	1	1

Figure 119: The final model of our bag of words.

Python Implementation:

```
#Import required libraries :
from sklearn.feature_extraction.text import CountVectorizer

#Text for analysis :
sentences = ["Jim and Pam travelled by the bus:",
             "The train was late",
             "The flight was full.Travelling by flight is expensive"]

#Create an object :
cv = CountVectorizer()

#Generating output for Bag of Words :
B_O_W = cv.fit_transform(sentences).toarray()

#Total words with their index in model :
```

```
print(cv.vocabulary_)
print("\n")

#Features :
print(cv.get_feature_names())
print("\n")

#Show the output :
print(B_O_W)
```

Figure 120: Python implementation code snippet of our bag of words.

```
{'jim': 7, 'and': 0, 'pam': 9, 'travelled': 12, 'by': 2, 'the': 10, 'bus': 1, 'train': 11, 'was': 14, 'late': 8, 'flight': 4,
'full': 5, 'travelling': 13, 'is': 6, 'expensive': 3}


['and', 'bus', 'by', 'expensive', 'flight', 'full', 'is', 'jim', 'late', 'pam', 'the', 'train', 'travelled', 'travelling', 'wa
s']


[[1 1 1 0 0 0 0 1 0 1 1 0 1 0 0]
 [0 0 0 0 0 0 0 0 1 0 1 1 0 0 1]
 [0 0 1 1 2 1 1 0 0 0 1 0 0 1 1]]
```

Figure 121: Output of our bag of words.

```
[[1 1 1 0 0 0 0 1 0 1 1 0 1 0 0]
 [0 0 0 0 0 0 0 0 1 0 1 1 0 0 1]
 [0 0 1 1 2 1 1 0 0 0 1 0 0 1 1]]
```

Figure 122: Output of our bag of words.

Applications:

1. Natural language processing.
2. Information retrieval from documents.
3. Classifications of documents.

Limitations:

1. **Semantic meaning**: It does not consider the semantic meaning of a word. It ignores the context in which the word is used.
2. **Vector size**: For large documents, the vector size increase, which may result in higher computational time.
3. **Preprocessing**: In preprocessing, we need to perform data cleansing before using it.

TF-IDF

TF-IDF stands for **Term Frequency — Inverse Document Frequency**, which is a scoring measure generally used in information retrieval (IR) and summarization. The TF-IDF score shows how important or relevant a term is in a given document.

The intuition behind TF and IDF:

If a particular word appears multiple times in a document, then it might have higher importance than the other words that appear fewer times (TF). At the same time, if a particular word appears many times in a document, but it is also present many times in some other documents, then maybe that word is frequent, so we cannot assign much importance to it. (IDF). For instance, we have a database of thousands of dog descriptions, and the user wants to search for **"a cute dog"** from our database. The job of our search engine would be to display the closest response to the user query. **How would a search engine do that?** The search engine will possibly use TF-IDF to calculate the score for all of our descriptions, and the result with the higher score will be displayed as a response to the user. Now, this is the case when there is no exact match for the user's query. If there is an exact match for the user query, then that result will be displayed first. Then, let's suppose there are four descriptions available in our database.

1. The furry dog.
2. A cute doggo.
3. A big dog.
4. The lovely doggo.

Notice that the first description contains 2 out of 3 words from our user query, and the second description contains 1 word from the query. The third description also contains 1 word, and the forth description contains no words from the user query. As we can sense that the closest answer to our query will be description number two, as it contains the essential word "**cute**" from the user's query, this is how TF-IDF calculates the value.

Notice that the term frequency values are the same for all of the sentences since none of the words in any sentences repeat in the same sentence. So, in this case, the value of TF will not be instrumental. Next, we are going to use IDF values to get the closest answer to the query. Notice that the word dog or doggo can appear in many many documents. Therefore, the IDF value is going to be very low. Eventually, the TF-IDF value will also be lower. However, if we check the word "cute" in the dog descriptions, then it will come up

relatively fewer times, so it increases the TF-IDF value. So the word “cute” has more discriminative power than “dog” or “doggo.” Then, our search engine will find the descriptions that have the word “cute” in it, and in the end, that is what the user was looking for.

Simply put, the higher the TF*IDF score, the rarer or unique or valuable the term and vice versa.

Now we are going to take a straightforward example and understand TF-IDF in more detail.

Example:

Sentence 1: This is the first document.

Sentence 2: This document is the second document.

TF: Term Frequency

$$TF = \frac{Frequency\ of\ the\ word\ in\ the\ sentence}{Total\ number\ of\ words\ in\ the\ sentence}$$

Figure 123: Calculation for the term frequency on TF-IDF.

a. Represent the words of the sentences in the table.

Sentence 1	Sentence 2
This	This
is	document
the	is
first	the
document	second
	document

Figure 124: Table representation of the sentences.

b. Displaying the frequency of words.

Sentence 1	TF	Sentence 2	TF
This	1	This	1
is	1	document	2
the	1	is	1
first	1	the	1
document	1	second	1

Figure 125: Table showing the frequency of words.

c. Calculating TF using a formula.

$$TF = \frac{Frequency\ of\ the\ word\ in\ the\ sentence}{Total\ number\ of\ words\ in\ the\ sentence}$$

Figure 126: Calculating TF.

Sentence 1	TF	Sentence 2	TF
This	1/5 = 0.20	This	1/6 = 0.166
is	1/5 = 0.20	document	1/3 = 0.33
the	1/5 = 0.20	is	1/6 = 0.166
first	1/5 = 0.20	the	1/6 = 0.166
document	1/5 = 0.20	second	1/6 = 0.166

Figure 127: Resulting TF.

IDF: Inverse Document Frequency

$$IDF = \frac{Total\ number\ of\ sentences}{Number\ of\ sentences\ containing\ that\ word}$$

Figure 128: Calculating the IDF.

d. Calculating IDF values from the formula.

Sentence 1	IDF	Sentence 2	IDF
This	2/2 = 1	This	2/2 = 1
is	2/2 = 1	document	2/2 = 1
the	2/2 = 1	is	2/2 = 1
first	2/1 = 2	the	2/2 = 1
document	2/2 = 1	second	2/1 = 2

Figure 129: Calculating IDF values from the formula.

e. Calculating TF-IDF.

TF-IDF is the multiplication of TF*IDF.

Word	TF-IDF 1	TF-IDF2
This	0.20*1 = 0.20	0.166*1 = 0.16
is	0.20*1 = 0.20	0.166*1 = 0.16
the	0.20*1 = 0.20	0.166*1 = 0.16
first	0.20*2 = 0.40	0*2 = 0
document	0.20*1 = 0.20	0.333*1 = 0.33
second	0*2 = 0	0.166*2 = 0.33

Figure 130: The resulting multiplication of TF-IDF.

In this case, notice that the import words that discriminate both the sentences are “first” in sentence-1 and “second” in sentence-2 as we can see, those words have a relatively higher value than other words.

However, there any many variations for smoothing out the values for large documents. The most common variation is to use a log value for TF-IDF. Let’s calculate the TF-IDF value again by using the new IDF value.

$$IDF = log\left(\frac{Total\ number\ of\ sentences}{Number\ of\ sentences\ containing\ that\ word}\right)$$

Figure 131: Using a log value for TF-IDF by using the new IDF value.

f. Calculating IDF value using log.

Word	IDF 1	IDF 2
This	log(1) = 0	log(1) = 0
is	log(1) = 0	log(1) = 0
the	log(1) = 0	log(1) = 0
first	log(2) =0.693	log(2) = 0.693
document	log(1) = 0	log(1) = 0
second	log(2) = 0.693	log(2) = 0.693

Figure 132: Calculating the IDF value using a log.

g. Calculating TF-IDF.

Word	TF-IDF 1	TF-IDF 2
This	0.20*log(1) = 0	0.166*log(1) = 0
is	0.20*log(1) = 0	0.333*log(1) = 0

the	0.20*log(1) = 0	0.166*log(1) = 0
first	0.20*log(2) =0.13	0*log(2) = 0
document	0.20*log(1) = 0	0.166*log(1) = 0
second	0*log(2) = 0	0.166*log(2) = 0.13

Figure 133: Calculating TF-IDF using a log.

As seen above, “first” and “second” values are important words that help us to distinguish between those two sentences.

Now that we saw the basics of TF-IDF. Next, we are going to use the sklearn library to implement TF-IDF in Python. A different formula calculates the actual output from our program. First, we will see an overview of our calculations and formulas, and then we will implement it in Python.

Actual Calculations:

a. Term Frequency (TF):

Sentence 1	TF	Sentence 2	TF
This	1	This	1
is	1	document	2
the	1	is	1
first	1	the	1
document	1	second	1

Figure 134: Actual calculation of TF.

b. Inverse Document Frequency (IDF):

$$IDF = log\left(\frac{Total\ number\ of\ sentences + 1}{Number\ of\ sentences\ containing\ that\ word + 1}\right)$$

Figure 135: Formula for the IDF.

Word	IDF 1	IDF 2
This	log(3/3) = 0	log(3/3) = 0
is	log(3/3) = 0	log(3/3) = 0
the	log(3/3) = 0	log(3/3) = 0
first	log(3/2) = 0.40	log(3/2) = 0.40
document	log(3/3) = 0	log(3/3) = 0
second	log(3/2) = 0.40	log(3/2) = 0.40

Figure 136: Applying a log to the IDF values.

c. Calculating final TF-IDF values:

$$TF - IDF = (TF) * (IDF + 1)$$

Figure 137: Calculating the final IDF values.

Word	TF-IDF 1	TF-IDF 2
This	1*(0+1) = 1	1*(0+1) = 1
is	1*(0+1) = 1	1*(0+1) = 1
the	1*(0+1) =1	1*(0+1) = 1
first	1*(0.40+1) = 1.40	0*(0.40+1) = 0
document	1*(0+1) = 1	2*(0+1) = 2
second	0*(0.40+1) = 0	1*(0.40+1) = 1.40

Figure 138: Final TF-IDF values.

Python Implementation:

```
#Import required libraries :
from sklearn.feature_extraction.text import TfidfVectorizer

#Sentences for analysis :
sentences = ['This is the first document','This document is the second document']

#Create an object :
vectorizer = TfidfVectorizer(norm = None)

#Generating output for TF_IDF :
X = vectorizer.fit_transform(sentences).toarray()

#Total words with their index in model :
print(vectorizer.vocabulary_)
print("\n")

#Features :
print(vectorizer.get_feature_names())
print("\n")

#Show the output :
print(X)
```

Figure 139: Python implementation of TF-IDF code snippet.

```
{'this': 5, 'is': 2, 'the': 4, 'first': 1, 'document': 0, 'second': 3}


['document', 'first', 'is', 'second', 'the', 'this']


[[1.         1.40546511 1.         0.         1.         1.        ]
 [2.         0.         1.         1.40546511 1.         1.        ]]
```

Figure 140: Final output.

Conclusion:

These are some of the basics for the exciting field of natural language processing (NLP). We hope you enjoyed reading this article and learned something new. Any suggestions or feedback is crucial to continue to improve. Please let us know in the comments if you have any.

. . .