ASIA PACIFIC UNIVERSITY
OF TECHNOLOGY & INNOVATION

INDIVIDUAL ASSIGNMENT

CT107-3-3-TXSA

TEXT ANALYTICS AND SENTIMENT ANALYSIS

CSDA__CT107-3-3-TXSA-L-5__2022-12-05

NAME	**: KEVIN AHMADIPUTRA**
TP NUMBER	**: TP058396**
HAND OUT DATE	**: 20 DECEMBER 2022**
HAND OUT DATE	**: 28 FEBRUARY 2023**
WEIGHTAGE	**: 25%**

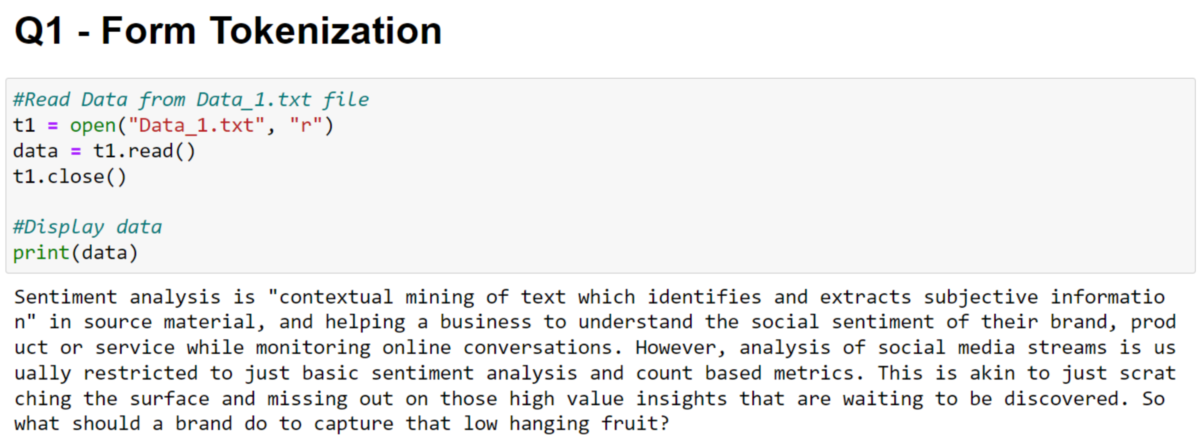
INSTRUCTIONS TO CANDIDATES:

1. Submit your assignment via Moodle.
2. Students are advised to underpin their answers with the use of references (cited using the 7th Edition of APA Referencing Style).
3. Late submission will be awarded zero (0) unless Extenuating Circumstances (EC) are upheld.
4. Cases of plagiarism will be penalized.
5. You must obtain 50% overall to pass this module.

Table of Contents

1 Tokenization

1.1 Demonstration of Sentence Segmentation.



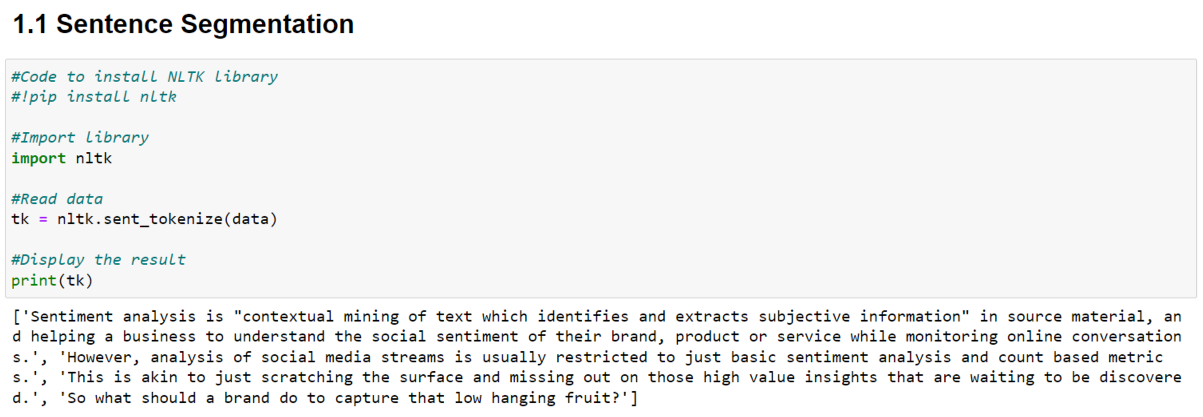
Q1 - Form Tokenization

```
#Read Data from Data_1.txt file
t1 = open("Data_1.txt", "r")
data = t1.read()
t1.close()

#Display data
print(data)
```

```
Sentiment analysis is "contextual mining of text which identifies and extracts subjective informatio
n" in source material, and helping a business to understand the social sentiment of their brand, prod
uct or service while monitoring online conversations. However, analysis of social media streams is us
ually restricted to just basic sentiment analysis and count based metrics. This is akin to just scrat
ching the surface and missing out on those high value insights that are waiting to be discovered. So
what should a brand do to capture that low hanging fruit?
```

Figure 1: Open & Read text file 1.

Figure 1 depicts codes that read a text file and assigns its sentences to a variable as a string. The code begins by opening the file and extracting its sentences. Next, it stores the text in the designated variable. Finally, the code closes the file and attempts to display the words contained within the variable. The text in this variable named data will be used for the next few sections as well.



1.1 Sentence Segmentation

```
#Code to install NLTK library
#!pip install nltk

#Import library
import nltk

#Read data
tk = nltk.sent_tokenize(data)

#Display the result
print(tk)
```

```
['Sentiment analysis is "contextual mining of text which identifies and extracts subjective information" in source material, an
d helping a business to understand the social sentiment of their brand, product or service while monitoring online conversation
s.', 'However, analysis of social media streams is usually restricted to just basic sentiment analysis and count based metric
s.', 'This is akin to just scratching the surface and missing out on those high value insights that are waiting to be discovere
d.', 'So what should a brand do to capture that low hanging fruit?']
```

Figure 2: Sentence Segmentation.

Sentence segmentation is the process in natural language processing that involves the division of the text into individual sentences that are usually necessary for various tasks, including text classification, summarization, and sentiment analysis (Baldwin, 2015).

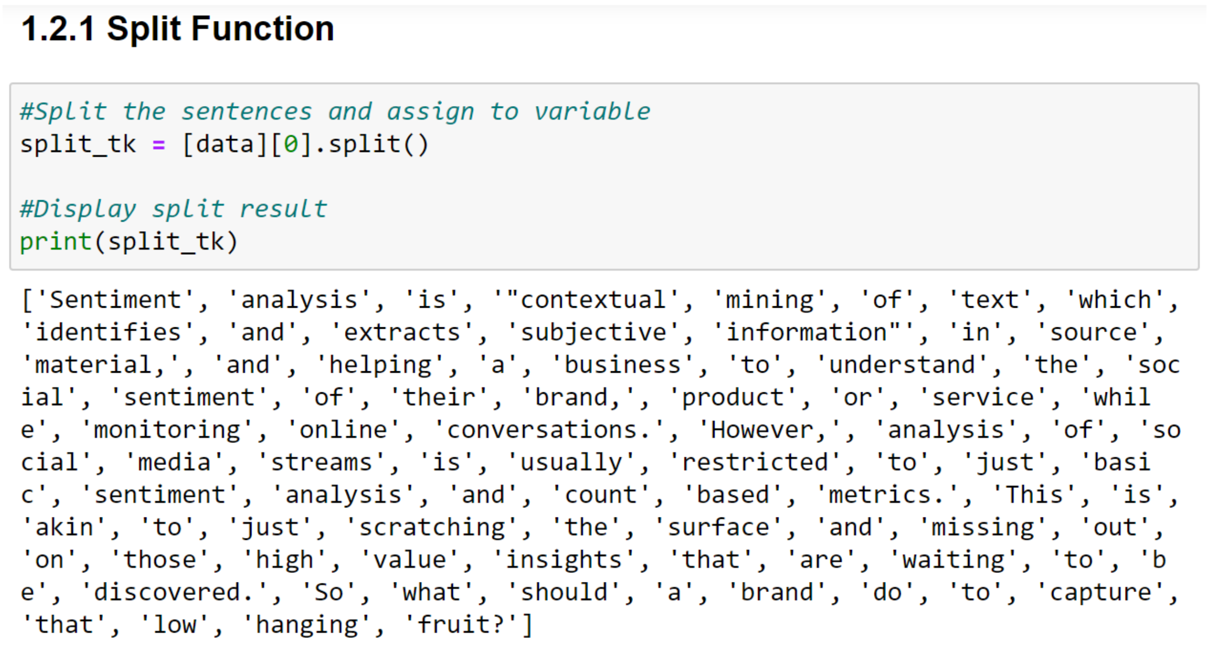
Figure 2 presents a code block that exemplifies the sentence segmentation process using the NLTK library. The first step is to install or import the library, followed by the use of the "nltk.sent_tokenize" function to extract individual sentences from the text. Those sentences are then assigned to a list variable and displayed by printing the variable. As shown in the figure, for the output, each sentence is separated and marked with a " sign at the start and end, indicating the completion of the sentence.

1.2 Demonstration of Word Tokenization.

Word tokenization is a fundamental technique in natural language processing that involves breaking a text into individual words or tokens (Manning, 2008). There is two python library used in this section which is re and NLTK.

The **re library** is used for word tokenization with regular expressions since it is a built-in Python library for working with regular expressions (regex), which are patterns used to manipulate text data. It can do some operations searching, replacing, and manipulating strings. Hence, regular expressions are able to find specific patterns, split strings, and replace text. On the other side, as a Python library for natural language processing (NLP), **NLTK library** provides a range of tools and resources which allow it to perform certain tasks such as tokenization, stemming, POS Tagging, etc.

1.2.1 Split Function.



1.2.1 Split Function

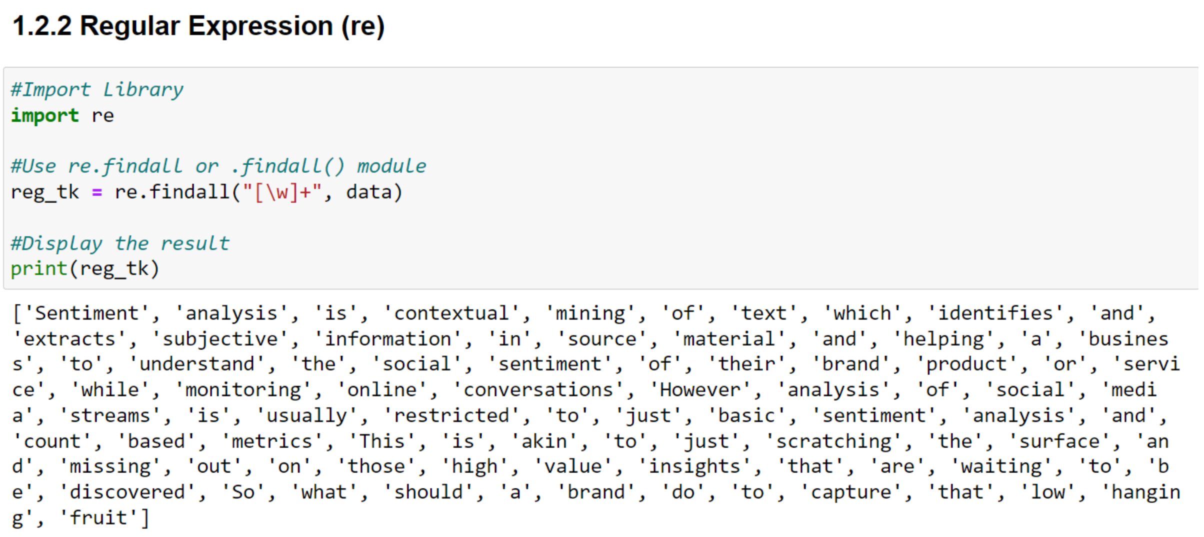
```
#Split the sentences and assign to variable
split_tk = [data][0].split()

#Display split result
print(split_tk)
```

```
['Sentiment', 'analysis', 'is', '"contextual', 'mining', 'of', 'text', 'which',
'identifies', 'and', 'extracts', 'subjective', 'information"', 'in', 'source',
'material,', 'and', 'helping', 'a', 'business', 'to', 'understand', 'the', 'soc
ial', 'sentiment', 'of', 'their', 'brand,', 'product', 'or', 'service', 'whil
e', 'monitoring', 'online', 'conversations.', 'However,', 'analysis', 'of', 'so
cial', 'media', 'streams', 'is', 'usually', 'restricted', 'to', 'just', 'basi
c', 'sentiment', 'analysis', 'and', 'count', 'based', 'metrics.', 'This', 'is',
'akin', 'to', 'just', 'scratching', 'the', 'surface', 'and', 'missing', 'out',
'on', 'those', 'high', 'value', 'insights', 'that', 'are', 'waiting', 'to', 'b
e', 'discovered.', 'So', 'what', 'should', 'a', 'brand', 'do', 'to', 'capture',
'that', 'low', 'hanging', 'fruit?']
```

Figure 3: Split Function – Word Tokenization.

The split function separates a string into words based on a specified delimiter. In this section, it uses space as the delimiter since none is explicitly defined. Consequently, the function captures each word in the text before every space as seen in Figure 3.

1.2.2 Regular Expression.



1.2.2 Regular Expression (re)

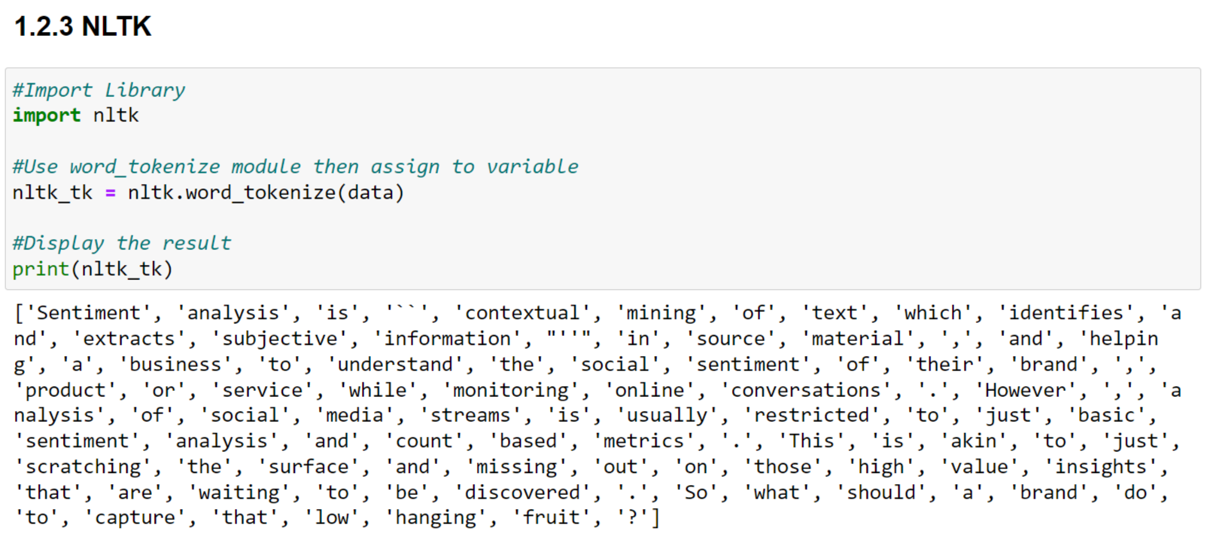
```
#Import Library
import re

#Use re.findall or .findall() module
reg_tk = re.findall("[\w]+", data)

#Display the result
print(reg_tk)
```

```
['Sentiment', 'analysis', 'is', 'contextual', 'mining', 'of', 'text', 'which', 'identifies', 'and',
'extracts', 'subjective', 'information', 'in', 'source', 'material', 'and', 'helping', 'a', 'busines
s', 'to', 'understand', 'the', 'social', 'sentiment', 'of', 'their', 'brand', 'product', 'or', 'servi
ce', 'while', 'monitoring', 'online', 'conversations', 'However', 'analysis', 'of', 'social', 'medi
a', 'streams', 'is', 'usually', 'restricted', 'to', 'just', 'basic', 'sentiment', 'analysis', 'and',
'count', 'based', 'metrics', 'This', 'is', 'akin', 'to', 'just', 'scratching', 'the', 'surface', 'an
d', 'missing', 'out', 'on', 'those', 'high', 'value', 'insights', 'that', 'are', 'waiting', 'to', 'b
e', 'discovered', 'So', 'what', 'should', 'a', 'brand', 'do', 'to', 'capture', 'that', 'low', 'hangin
g', 'fruit']
```

Figure 4: Regular Expression – Word Tokenization.

Firstly, import the regular expression library and then use the re.findall() function in order to split the text into words based on the specified pattern. In this example, the regular expression [\w]+ matches all the words in the text and captures only the letters or characters within them, ignoring any punctuation. As a result in Figure 4 above, each word is separated and stored in a list variable without any punctuation marks.

1.2.3 NLTK.



1.2.3 NLTK

```
#Import Library
import nltk

#Use word_tokenize module then assign to variable
nltk_tk = nltk.word_tokenize(data)

#Display the result
print(nltk_tk)
```

```
['Sentiment', 'analysis', 'is', '``', 'contextual', 'mining', 'of', 'text', 'which', 'identifies', 'a
nd', 'extracts', 'subjective', 'information', "''", 'in', 'source', 'material', ',', 'and', 'helpin
g', 'a', 'business', 'to', 'understand', 'the', 'social', 'sentiment', 'of', 'their', 'brand', ',',
'product', 'or', 'service', 'while', 'monitoring', 'online', 'conversations', '.', 'However', ',', 'a
nalysis', 'of', 'social', 'media', 'streams', 'is', 'usually', 'restricted', 'to', 'just', 'basic',
'sentiment', 'analysis', 'and', 'count', 'based', 'metrics', '.', 'This', 'is', 'akin', 'to', 'just',
'scratching', 'the', 'surface', 'and', 'missing', 'out', 'on', 'those', 'high', 'value', 'insights',
'that', 'are', 'waiting', 'to', 'be', 'discovered', '.', 'So', 'what', 'should', 'a', 'brand', 'do',
'to', 'capture', 'that', 'low', 'hanging', 'fruit', '?']
```

Figure 5: NLTK - Word Tokenization.

To perform word tokenization using the NLTK library, begin by importing the library and then use the nltk.word_tokenize() function. This method will tokenize the text by separating each word and punctuation, resulting in a list variable that contains all the words and punctuation marks found in the text.

1.3 The Differences between each Tokenisation Operation.

Tokenization operation involves breaking down a sentence or text into a list of separate words and the tokenization process typically consists of removing punctuation and other non-word characters and separating words based on whitespace. Based on the output of each word tokenization conducted, the output of the **split function** is a list variable containing the words as separate elements and this process does not maintain any punctuation marks or other non-alphanumeric characters. Meanwhile, the output of the **regular expression** is a list variable containing only alphanumeric characters based on the pattern specified. On the other hand, **NLTK** produces a list variable with each word and punctuation mark as separate elements. Hence, it will hold all the punctuation marks and non-alphanumeric characters found in the text.

1.4 Selection of the most Suitable Tokenization Operation.

NLTK is a preferred option for word tokenization in natural language processing due to its accuracy and advanced features. NLTK preserves all punctuation marks and other non-alphanumeric characters, making it suitable for tasks such as sentiment analysis or machine translation, where these characters provide crucial context for the analysis (Baccianella, 2010). Moreover, based on the output of each operation, NLTK produces the most accurate and better results compared to other techniques, as evident from Figure 5.

2 Form Word Stemming.

2.1 The Significance of Stemming in Text Analytics.

Stemming is a crucial technique in text analytics to improve the efficiency and accuracy of text analysis by removing or reducing the number of unique words to reduce the words to their base or root form. It also reduces the complexity of the text and eliminates redundancies, allowing the text to be analyzed more easily and accurately (Bostanci, 2016).

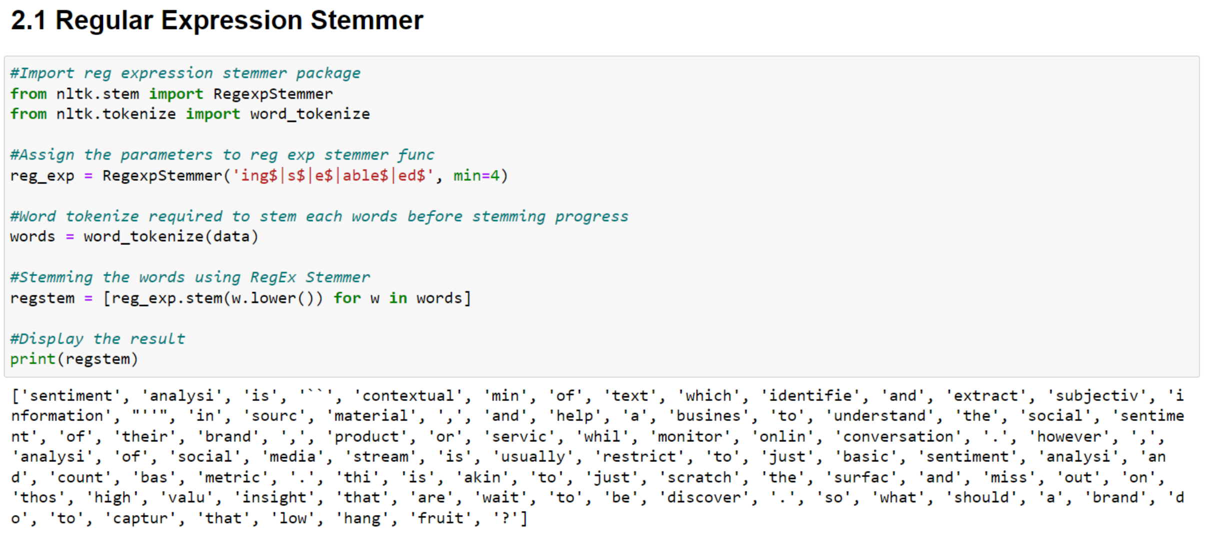
Furthermore, it also addresses issues such as data sparsity and can improve the performance of natural language processing algorithms like clustering and classification. In information retrieval applications, such as search engines, stemming helps to ensure that search queries are matched more comprehensively with indexed documents (Barker, 2021). In summary, stemming is a powerful tool in text analytics that can improve the accuracy, efficiency, and effectiveness of natural language processing tasks.

2.2 Differentiate between Regular Expression Stemmer and Porter Stemmer.

RegEx stemmer is rule-based and works by matching and replacing specific patterns of characters that aggressively reduce words to their root form using character pattern matching (Varghese, 2018). Hence, it may be resulting in a higher degree of word reduction than the Porter stemmer and may over-stemming the word, which means the word will lose meaning in some cases as shown in Figure 6. Meanwhile, the Porter stemmer is a dictionary-based approach that uses a set of predefined approach that preserves the base form of words to a greater extent but may result in under-stemming and the retention of irrelevant variations of the same word (Chaturvedi, 2020). This can be seen in the output of Porter Stemmer in Figure 7.

On the other hand, the main difference between the two algorithms is that the Regular Expression stemmer can handle irregular forms of words and is more effective for a range of languages, while the Porter stemmer cannot or is less effective for languages with irregular morphology. Additionally, the RegEx stemmer can be customized to handle specific rules for individual words, which makes it more accurate in certain cases (Abushawar, 2016).

2.3 Demonstration of Word Stemming.



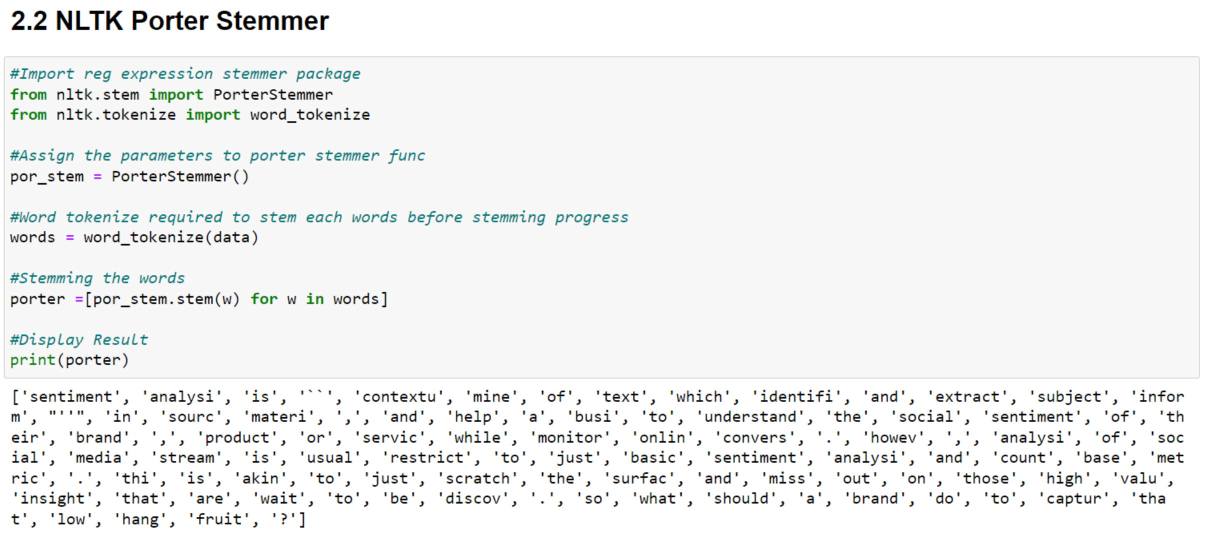
2.1 Regular Expression Stemmer

```
#Import reg expression stemmer package
from nltk.stem import RegexpStemmer
from nltk.tokenize import word_tokenize

#Assign the parameters to reg exp stemmer func
reg_exp = RegexpStemmer('ing$|s$|e$|able$|ed$', min=4)

#Word tokenize required to stem each words before stemming progress
words = word_tokenize(data)

#Stemming the words using RegEx Stemmer
regstem = [reg_exp.stem(w.lower()) for w in words]

#Display the result
print(regstem)
```

```
['sentiment', 'analysi', 'is', '``', 'contextual', 'min', 'of', 'text', 'which', 'identifie', 'and', 'extract', 'subjectiv', 'i
nformation', "''", 'in', 'sourc', 'material', ',', 'and', 'help', 'a', 'busines', 'to', 'understand', 'the', 'social', 'sentime
nt', 'of', 'their', 'brand', ',', 'product', 'or', 'servic', 'whil', 'monitor', 'onlin', 'conversation', '.', 'however', ',',
'analysi', 'of', 'social', 'media', 'stream', 'is', 'usually', 'restrict', 'to', 'just', 'basic', 'sentiment', 'analysi', 'an
d', 'count', 'bas', 'metric', '.', 'thi', 'is', 'akin', 'to', 'just', 'scratch', 'the', 'surfac', 'and', 'miss', 'out', 'on',
'thos', 'high', 'valu', 'insight', 'that', 'are', 'wait', 'to', 'be', 'discover', '.', 'so', 'what', 'should', 'a', 'brand', 'd
o', 'to', 'captur', 'that', 'low', 'hang', 'fruit', '?']
```

Figure 6: Regular Expression Stemmer.

RegexpStemmer is a stemming algorithm from nltk.stem module of the NLTK library that use RegEx to detect and remove suffixes from the words based on the assigned pattern. Figure 6 shows the step to do regular expression stemmer by importing the necessary packages and library first and then assigning the parameter to the stemmer function. Then, the word tokenize the text and the stemming to the tokenized words. As the result, some words stemmed from their base or root form.



2.2 NLTK Porter Stemmer

```
#Import reg expression stemmer package
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize

#Assign the parameters to porter stemmer func
por_stem = PorterStemmer()

#Word tokenize required to stem each words before stemming progress
words = word_tokenize(data)

#Stemming the words
porter =[por_stem.stem(w) for w in words]

#Display Result
print(porter)
```

```
['sentiment', 'analysi', 'is', '``', 'contextu', 'mine', 'of', 'text', 'which', 'identifi', 'and', 'extract', 'subject', 'infor
m', "''", 'in', 'sourc', 'materi', ',', 'and', 'help', 'a', 'busi', 'to', 'understand', 'the', 'social', 'sentiment', 'of', 'th
eir', 'brand', ',', 'product', 'or', 'servic', 'while', 'monitor', 'onlin', 'convers', '.', 'howev', ',', 'analysi', 'of', 'soc
ial', 'media', 'stream', 'is', 'usual', 'restrict', 'to', 'just', 'basic', 'sentiment', 'analysi', 'and', 'count', 'base', 'met
ric', '.', 'thi', 'is', 'akin', 'to', 'just', 'scratch', 'the', 'surfac', 'and', 'miss', 'out', 'on', 'those', 'high', 'valu',
'insight', 'that', 'are', 'wait', 'to', 'be', 'discov', '.', 'so', 'what', 'should', 'a', 'brand', 'do', 'to', 'captur', 'tha
t', 'low', 'hang', 'fruit', '?']
```

Figure 7: NLTK Porter Stemmer.

PorterStemmer is also an algorithm from nltk.stem module of the NLTK library. Unlike, RegEx, this algorithm is pre-defined and doesn't need a specific assigned pattern. In Figure 7, NLTK Porter Stemmer was utilized by importing necessary packages and libraries. Then, assigning parameters to the stemmer function, and tokenizing the text. Finally, the words then go through the stemming function which is resulting each word being reduced to its base form, although the output may differ from that of the RegEx stemmer.

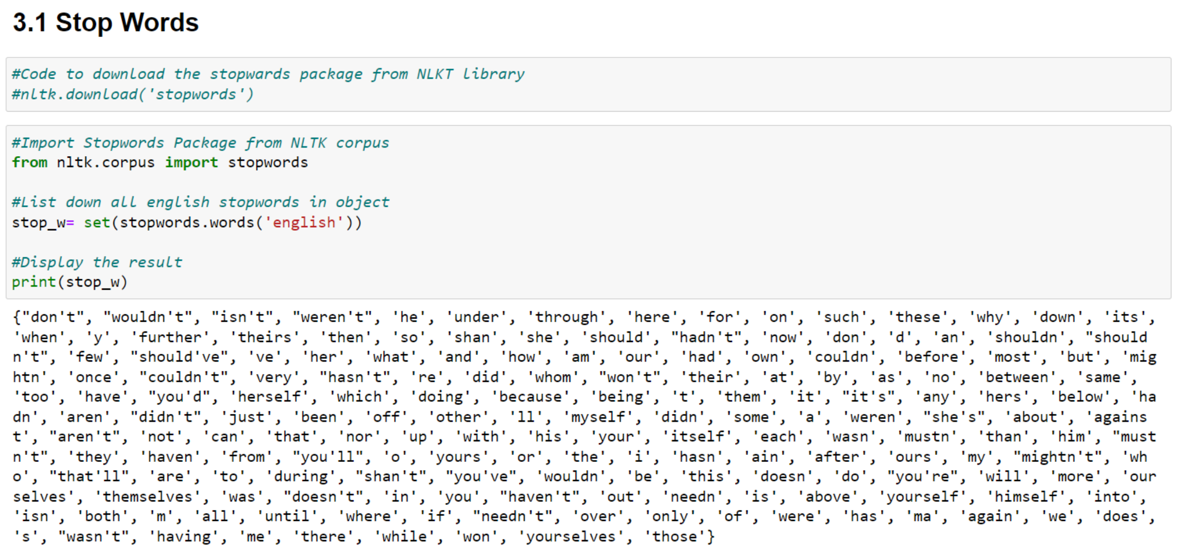
2.4 Selection of the Most Compatible Stemming Operation for Text Analytics.

NLTK Porter stemmer is generally better for text analytics due to its conservative approach that preserves the root form of words to a greater extent, helping to retain the meaning of the words. In contrast, the Regular Expression stemmer results in more aggressive stemming, potentially leading to the loss of meaning in some cases. In addition, based on the output of each stemmer in Figures 6 and 7, NLTK Porter Stemmer is a better choice for maintaining the meaning of the words.

3 Filter Stop Words & Punctuation.

3.1 Demonstration of Stop Words and Punctuation Removal.

3.1.1 Stop Words



3.1 Stop Words

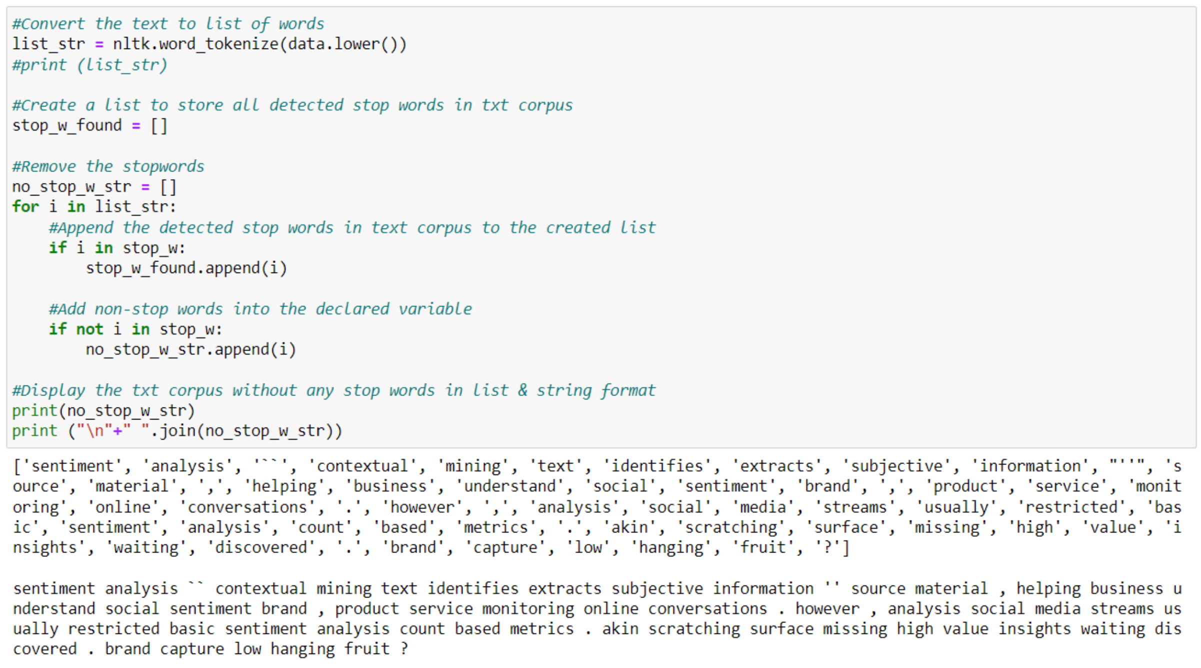
```
#Code to download the stopwards package from NLKT Library
#nltk.download('stopwords')
```

```
#Import Stopwords Package from NLTK corpus
from nltk.corpus import stopwords

#List down all english stopwords in object
stop_w= set(stopwords.words('english'))

#Display the result
print(stop_w)
```

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

Figure 8: Stop Words.

Steps to list and assign all English stop words are demonstrated in Figure 8 start with importing the library, and then the English language is set for the stop words to collect all English stop words. The stop words are then stored in a list variable. Finally, the list variable can be displayed to show all the English stop words contained within it.



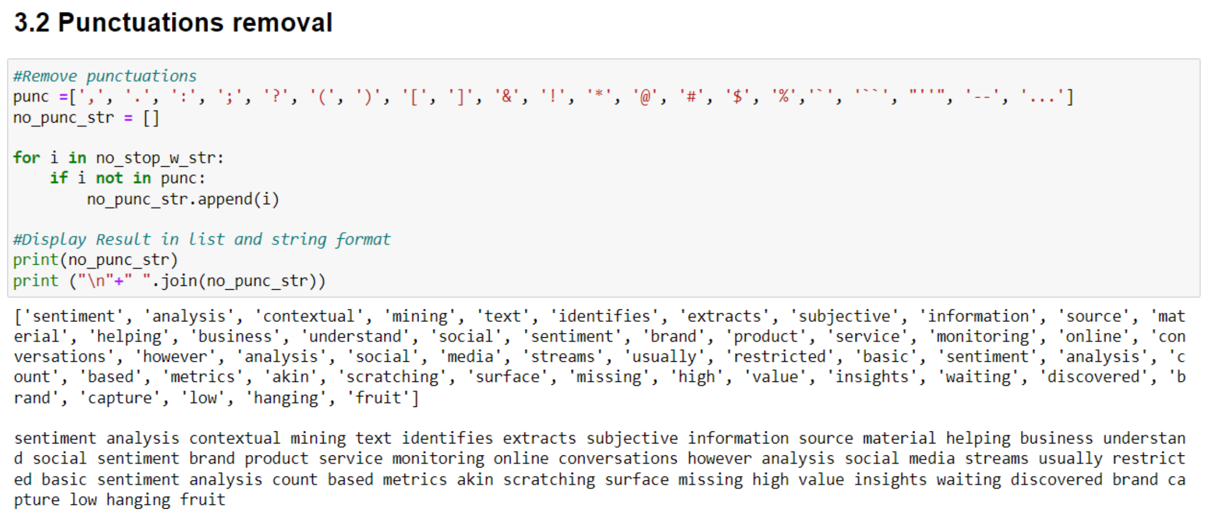
```
#Convert the text to list of words
list_str = nltk.word_tokenize(data.lower())
#print (list_str)

#Create a list to store all detected stop words in txt corpus
stop_w_found = []

#Remove the stopwords
no_stop_w_str = []
for i in list_str:
    #Append the detected stop words in text corpus to the created list
    if i in stop_w:
        stop_w_found.append(i)

    #Add non-stop words into the declared variable
    if not i in stop_w:
        no_stop_w_str.append(i)

#Display the txt corpus without any stop words in list & string format
print(no_stop_w_str)
print ("\n"+" ".join(no_stop_w_str))
```

```
['sentiment', 'analysis', '``', 'contextual', 'mining', 'text', 'identifies', 'extracts', 'subjective', 'information', "''", 's
ource', 'material', ',', 'helping', 'business', 'understand', 'social', 'sentiment', 'brand', ',', 'product', 'service', 'monit
oring', 'online', 'conversations', '.', 'however', ',', 'analysis', 'social', 'media', 'streams', 'usually', 'restricted', 'bas
ic', 'sentiment', 'analysis', 'count', 'based', 'metrics', '.', 'akin', 'scratching', 'surface', 'missing', 'high', 'value', 'i
nsights', 'waiting', 'discovered', '.', 'brand', 'capture', 'low', 'hanging', 'fruit', '?']

sentiment analysis `` contextual mining text identifies extracts subjective information '' source material , helping business u
nderstand social sentiment brand , product service monitoring online conversations . however , analysis social media streams us
ually restricted basic sentiment analysis count based metrics . akin scratching surface missing high value insights waiting dis
covered . brand capture low hanging fruit ?
```

Figure 9: Sentences with Removed Stop Words.

Figure 9 tells how to remove stop words from a given text and record the detected stop words to a declared variable. Initially, the text is converted into a list of words, and a list variable is defined to store the detected stop words. Then, using a loop and an if-else statement, the stop words in the text are removed and stored in the stop_w_found variable. Consequently, the cleaned text no longer includes any stop words.

3.1.2 Punctuations Removal


3.2 Punctuations removal

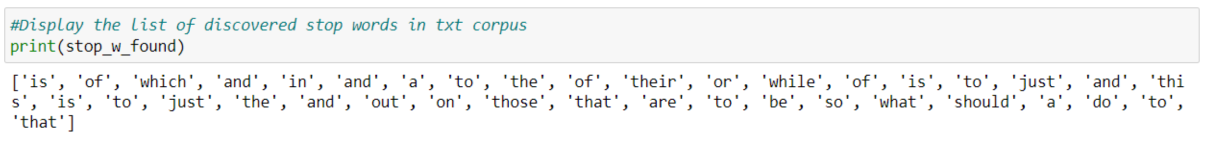
```
#Remove punctuations
punc =[',', '.', ':', ';', '?', '(', ')', '[', ']', '&', '!', '*', '@', '#', '$', '%','`', '``', "''", '--', '...']
no_punc_str = []

for i in no_stop_w_str:
    if i not in punc:
        no_punc_str.append(i)

#Display Result in list and string format
print(no_punc_str)
print ("\n"+" ".join(no_punc_str))
```

```
['sentiment', 'analysis', 'contextual', 'mining', 'text', 'identifies', 'extracts', 'subjective', 'information', 'source', 'mat
erial', 'helping', 'business', 'understand', 'social', 'sentiment', 'brand', 'product', 'service', 'monitoring', 'online', 'con
versations', 'however', 'analysis', 'social', 'media', 'streams', 'usually', 'restricted', 'basic', 'sentiment', 'analysis', 'c
ount', 'based', 'metrics', 'akin', 'scratching', 'surface', 'missing', 'high', 'value', 'insights', 'waiting', 'discovered', 'b
rand', 'capture', 'low', 'hanging', 'fruit']

sentiment analysis contextual mining text identifies extracts subjective information source material helping business understan
d social sentiment brand product service monitoring online conversations however analysis social media streams usually restrict
ed basic sentiment analysis count based metrics akin scratching surface missing high value insights waiting discovered brand ca
pture low hanging fruit
```

Figure 10: Punctuations Removal.

The code section in Figure 10 demonstrates how to remove all punctuations inside the given text starting by declaring a variable that contains a list of punctuation. Then utilize the loop function to remove all punctuation inside the given text by iterating through each word and removing any punctuation mark found. This process results in a clean version of the given text without any punctuation marks.

3.2 Report the Stop Words Collected.


```
#Display the list of discovered stop words in txt corpus
print(stop_w_found)
```

```
['is', 'of', 'which', 'and', 'in', 'and', 'a', 'to', 'the', 'of', 'their', 'or', 'while', 'of', 'is', 'to', 'just', 'and', 'thi
s', 'is', 'to', 'just', 'the', 'and', 'out', 'on', 'those', 'that', 'are', 'to', 'be', 'so', 'what', 'should', 'a', 'do', 'to',
'that']
```

Figure 11: Detected Stop Words.

Figure 11 displays the code to print all the stop words detected in the given text, stored in the variable stop_w_found. The output shows the list of stop words found, which can be helpful in understanding which words were removed from the text during the cleaning process.

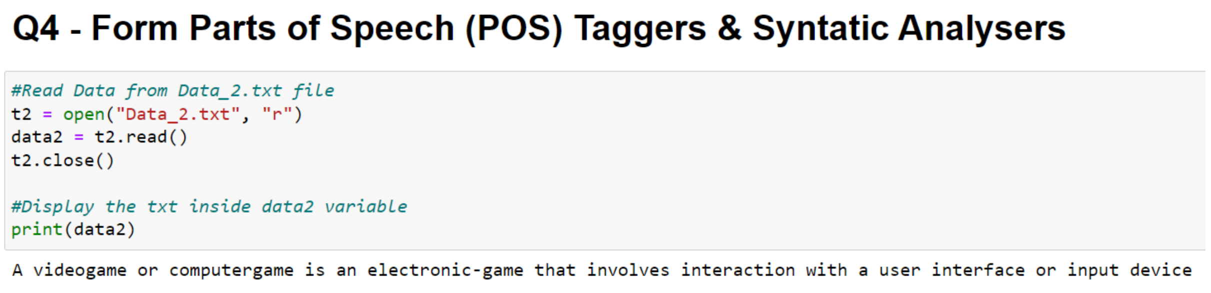
3.3 The Significance of Filtering the Stop Words & Punctuations.

First of all, stop words and punctuation are commonly found in natural language text, but they do not add much meaning to the text and can even introduce noise and inconsistencies. Removing stop words and punctuation is crucial in text analytics as it eliminates noise and inconsistencies in the data, leading to improved accuracy and efficiency in data analysis (Alotaibi, 2020).

This process also makes the data more concise and easier to analyze, resulting in better interpretability of results. Additionally, removing stop words and punctuation can improve the consistency of data, thus enhancing the performance of algorithms such as text classification. As shown in Figures 9 and 10, the text looks cleaner and better without stop words and punctuation, highlighting the importance of this step in text analytics.

4 Form Path of Speech (POS) Taggers & Syntactic Analysers.

4.1 Demonstration of POS Tagging.



Q4 - Form Parts of Speech (POS) Taggers & Syntatic Analysers

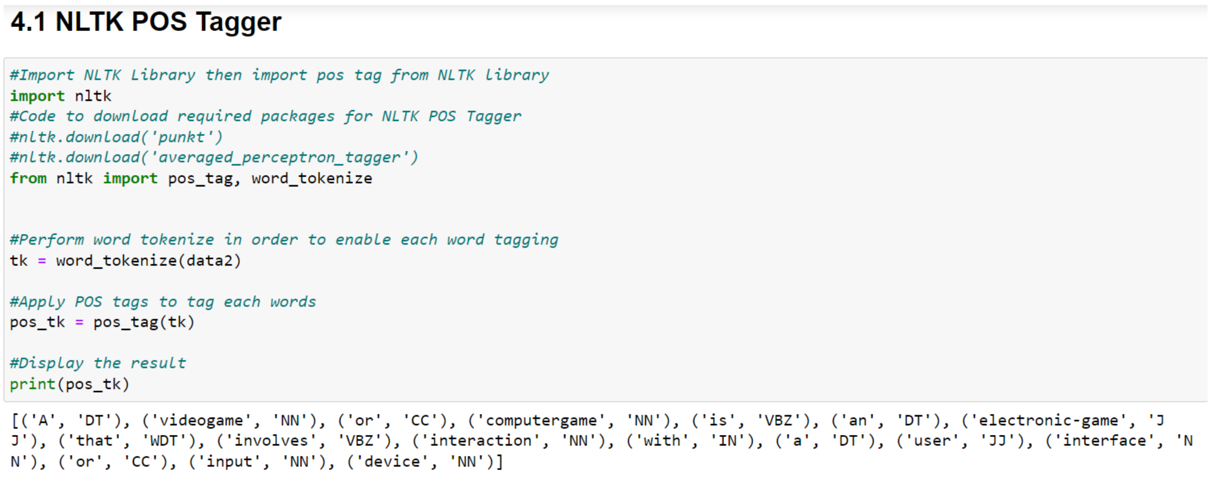
```
#Read Data from Data_2.txt file
t2 = open("Data_2.txt", "r")
data2 = t2.read()
t2.close()

#Display the txt inside data2 variable
print(data2)
```

```
A videogame or computergame is an electronic-game that involves interaction with a user interface or input device
```

Figure 12: Open & Read text file 2.

Figure 12 shows a code snippet that reads a text file and assigns its sentences to a string variable. The code starts by opening the file and extracting its sentences. It then saves the text in a designated variable. Finally, it closes the file and displays the words within the variable.

4.1.1 NLTK POS Tagger.



4.1 NLTK POS Tagger

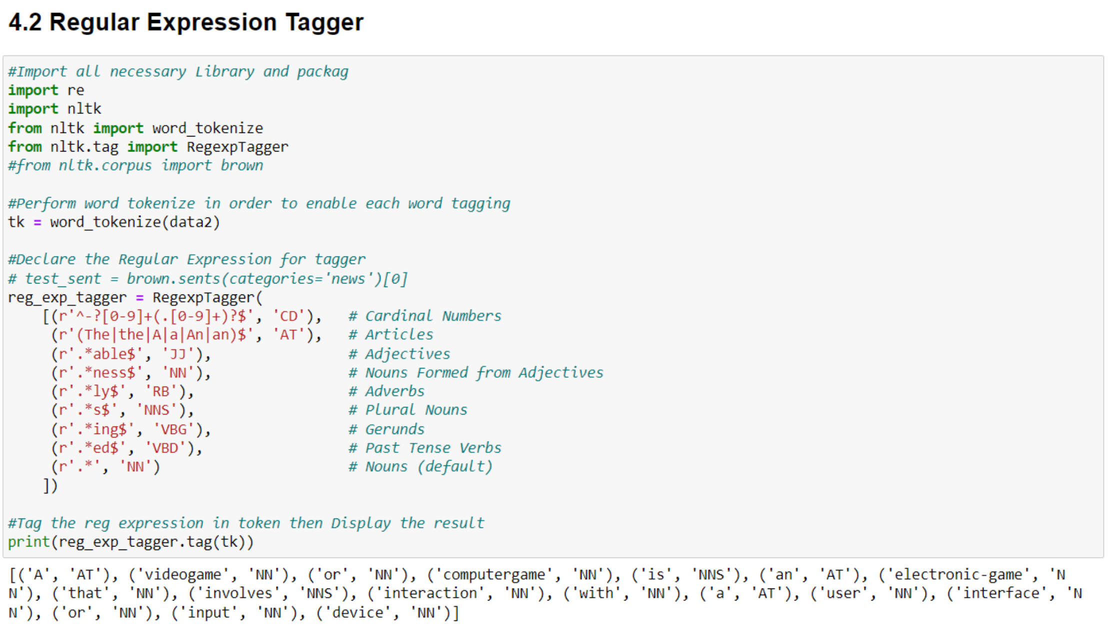
```
#Import NLTK Library then import pos tag from NLTK library
import nltk
#Code to download required packages for NLTK POS Tagger
#nltk.download('punkt')
#nltk.download('averaged_perceptron_tagger')
from nltk import pos_tag, word_tokenize


#Perform word tokenize in order to enable each word tagging
tk = word_tokenize(data2)

#Apply POS tags to tag each words
pos_tk = pos_tag(tk)

#Display the result
print(pos_tk)
```

```
[('A', 'DT'), ('videogame', 'NN'), ('or', 'CC'), ('computergame', 'NN'), ('is', 'VBZ'), ('an', 'DT'), ('electronic-game', 'J
J'), ('that', 'WDT'), ('involves', 'VBZ'), ('interaction', 'NN'), ('with', 'IN'), ('a', 'DT'), ('user', 'JJ'), ('interface', 'N
N'), ('or', 'CC'), ('input', 'NN'), ('device', 'NN')]
```

Figure 13: NLTK POS Tagger.

The NLTK POS Tagger process in Figure 13 begins with importing the necessary library, followed by tokenizing the given text and applying the POS tags to tag each word in the tokenized text using the pos_tag functions. It is an imported function from nltk library that is used to label the words as part of POS tagging. As a result, words from the text are tagged and labeled with their corresponding part of speech using the NLTK POS Tagger.

4.1.2 RegEx (Regular Expression) Tagger.



4.2 Regular Expression Tagger

```
#Import all necessary Library and packag
import re
import nltk
from nltk import word_tokenize
from nltk.tag import RegexpTagger
#from nltk.corpus import brown

#Perform word tokenize in order to enable each word tagging
tk = word_tokenize(data2)

#Declare the Regular Expression for tagger
# test_sent = brown.sents(categories='news')[0]
reg_exp_tagger = RegexpTagger(
    [(r'^-?[0-9]+(.[0-9]+)?$', 'CD'),   # Cardinal Numbers
     (r'(The|the|A|a|An|an)$', 'AT'),   # Articles
     (r'.*able$', 'JJ'),                # Adjectives
     (r'.*ness$', 'NN'),                # Nouns Formed from Adjectives
     (r'.*ly$', 'RB'),                  # Adverbs
     (r'.*s$', 'NNS'),                  # Plural Nouns
     (r'.*ing$', 'VBG'),                # Gerunds
     (r'.*ed$', 'VBD'),                 # Past Tense Verbs
     (r'.*', 'NN')                      # Nouns (default)
    ])

#Tag the reg expression in token then Display the result
print(reg_exp_tagger.tag(tk))
```

```
[('A', 'AT'), ('videogame', 'NN'), ('or', 'NN'), ('computergame', 'NN'), ('is', 'NNS'), ('an', 'AT'), ('electronic-game', 'N
N'), ('that', 'NN'), ('involves', 'NNS'), ('interaction', 'NN'), ('with', 'NN'), ('a', 'AT'), ('user', 'NN'), ('interface', 'N
N'), ('or', 'NN'), ('input', 'NN'), ('device', 'NN')]
```

Figure 14: Regular Expression Tagger.

The process of performing POS tagging using regular expressions is demonstrated in Figure 14. The first step is to import the NLTK and re-library then followed by tokenizing the given text. The regular expression tagger is then defined within a designated variable by using RegExpTagger which is an imported function from the nltk library that uses regular expression and declared pattern to tag the words. The regular expression is then used to tag each word from the tokenized text, resulting in each word being tagged with the RegEx Tagger.

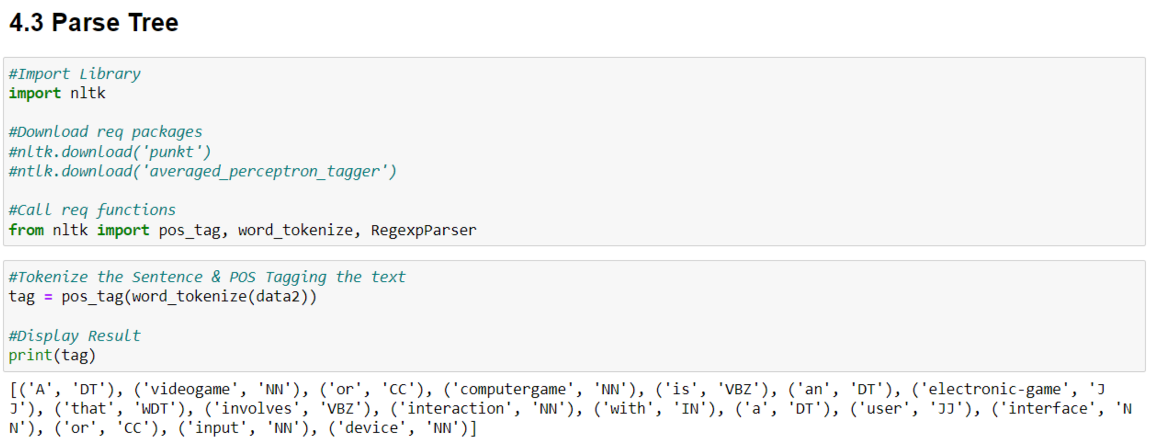
4.2 Differences of the POS Taggers Using the Output Obtained.

According to Figures 13 & 14, NLTK Tagger is more toward a statistical approach and is trained on a large corpus of annotated text. It also uses the built-in function or algorithm to learn the pattern in the text and assign the appropriate tags based on the context. Meanwhile, RegEx Tagger is a ruled-based approach and relies on the predefined pattern to tag each word. In addition, the RegEx tagger is quite flexible as per it can be customized to include specific knowledge and might be useful in certain cases while the NLTK Tagger cannot. Overall, NLTK Tagger can handle a wide range of text languages and genres compared to RegEx Tagger which is more suitable for specific languages or domains.

4.3 Selection of Best POS Word Tagger for Text Analytics.

In this case, NLTK POS Tagger is more suitable since it is a pre-trained tagger that uses machine learning techniques and a large annotated corpus to tag words in a text. Hence, it is better in terms of accuracy, recall, and precision in POS tagging for the English language. On the other hand, Regular Expression Tagger is less effective due to allows because it requires manual specification of tag patterns which is only good in certain cases. Those conditions can be seen in Figures 13 and 14 where RegEx contains a set of a pre-defined patterns whereas the NLTK will automatically tag with one command. In conclusion, NLTK POS Tagger is generally better in most applications of text analytics, especially in situations where high accuracy is crucial.

4.4 Create Possible Parse Tree and Attach the Python Code.



4.3 Parse Tree

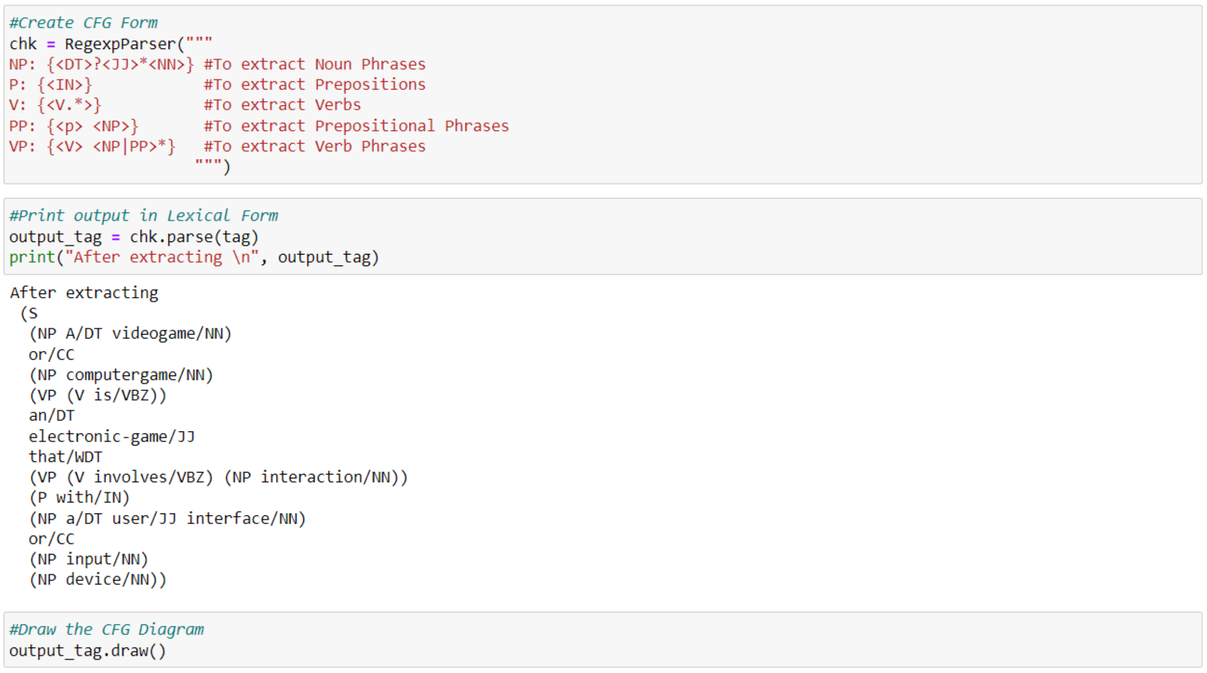
```
#Import Library
import nltk

#Download req packages
#nltk.download('punkt')
#ntlk.download('averaged_perceptron_tagger')

#Call req functions
from nltk import pos_tag, word_tokenize, RegexpParser
```

```
#Tokenize the Sentence & POS Tagging the text
tag = pos_tag(word_tokenize(data2))

#Display Result
print(tag)
```

```
[('A', 'DT'), ('videogame', 'NN'), ('or', 'CC'), ('computergame', 'NN'), ('is', 'VBZ'), ('an', 'DT'), ('electronic-game', 'J
J'), ('that', 'WDT'), ('involves', 'VBZ'), ('interaction', 'NN'), ('with', 'IN'), ('a', 'DT'), ('user', 'JJ'), ('interface', 'N
N'), ('or', 'CC'), ('input', 'NN'), ('device', 'NN')]
```

Figure 15: Parse Tree Code 1.

The code above shows how to create a parse tree from the given text by importing the required library and packages, then tokenizing and performing POS tagging on the given text.



```
#Create CFG Form
chk = RegexpParser("""
NP: {<DT>?<JJ>*<NN>} #To extract Noun Phrases
P: {<IN>}            #To extract Prepositions
V: {<V.*>}           #To extract Verbs
PP: {<p> <NP>}       #To extract Prepositional Phrases
VP: {<V> <NP|PP>*}   #To extract Verb Phrases
                    """)
```

```
#Print output in Lexical Form
output_tag = chk.parse(tag)
print("After extracting \n", output_tag)
```

```
After extracting
 (S
  (NP A/DT videogame/NN)
  or/CC
  (NP computergame/NN)
  (VP (V is/VBZ))
  an/DT
  electronic-game/JJ
  that/WDT
  (VP (V involves/VBZ) (NP interaction/NN))
  (P with/IN)
  (NP a/DT user/JJ interface/NN)
  or/CC
  (NP input/NN)
  (NP device/NN))
```

```
#Draw the CFG Diagram
output_tag.draw()
```

Figure 16: Parse Tree Code 2

In Figure 16, the code for creating a parse tree continues by creating a CFG form in the declared variable and then printing the output in lexical form. The next process is extracting the tagged word with CFG Form. Finally, draw the CFG Diagram in Parse tree form.

The parse tree generated in Figure 17 provides a visual representation of the text's syntactic structure, making it easier to analyze and interpret. Hence, it can help in understanding the syntactic structure of the text and can be used for further analysis.

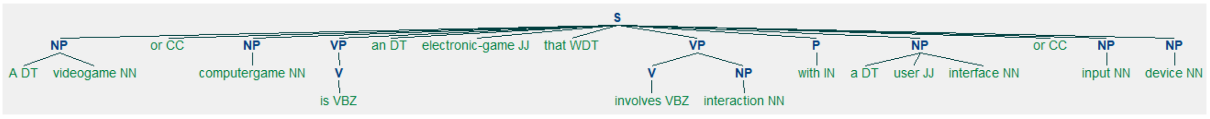


Figure 17: Parse Tree Output.

The given text's parse tree showcases the different parts of speech (POS) in each word and their relationships. It begins with a root sentence (S) composed of several phrases, including two noun phrases (NP) for "videogame" and "computergame" connected by "or". The following phrase (VP) has the verb "is". Subsequently, a determiner phrase (DT and JJ) depicts "electronic-game" type, and (WDT and VP) describes the character of the game, that it "involves interaction”. It continues with a preposition (P) "with" and a phrase (NP) describing the interface type used to interact with the game - "user interface" or "input device." Finally, the last phrase (CC and NP) connects the interface or input device type.

5 References

Abushawar, S. a. (2016). Rule-based Arabic stemmer. *Procedia Computer Science,*, 55-61.

Alotaibi, F. &. (2020). *Science Information.* thesai: https://doi.org/10.14569/ijacsa.2020.0110901

Baccianella, S. E. (2010). An enhanced lexical resource for sentiment analysis and opinion mining. *In LREC*, 2200-2204.

Baldwin, T. (2015). The language of information retrieval. *Pearson Australia*.

Barker, K. R. (2021). Information retrieval: searching in the 21st century. *Journal of the European Association for Health Information and Libraries,*, 2-5.

Bostanci, E. &. (2016). Text mining: Techniques, applications, and issues. *Journal of Economics, Finance and Accounting*, 1-20.

Chaturvedi, A. R. (2020). A Comparative Study of Stemming Algorithms in Text Mining. *Computing Methodologies and Communication*, 202-210.

Manning, C. D. (2008). Introduction to information retrieval. *Cambridge University Press*.

Varghese, A. S. (2018). Performance analysis of different stemming algorithms on Malayalam text. *Conference on Computational Intelligence & Communication Technology* , 1-6.