INDIVIDUAL ASSIGNMENT

CT107-3-3-TXSA

TEXT ANALYTICS AND SENTIMENT ANALYSIS

CSSE___CT107-3-3-TXSA-L-3___2022-09-05

HAND OUT DATE: 19 SEPTEMBER 2022

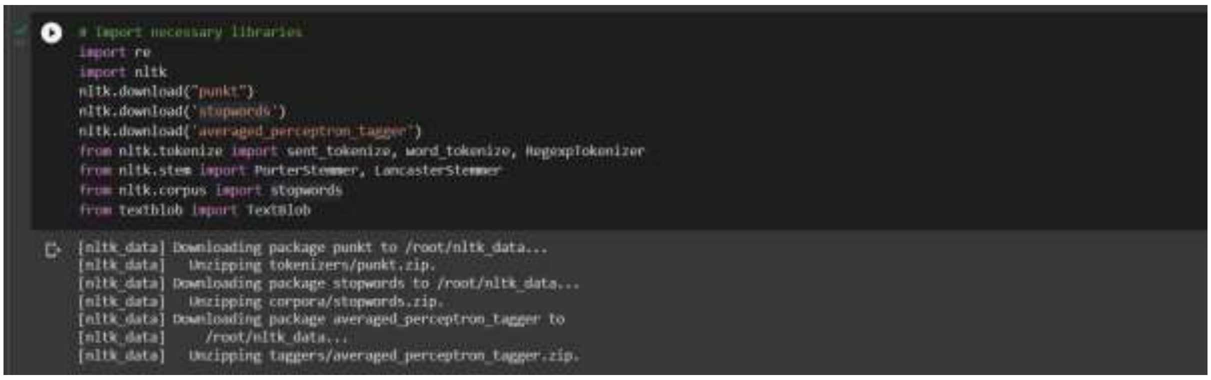
HAND IN DATE: 16 DECEMBER 2022

WEIGHTAGE: 25%

TP Number	**Student Name**
TP060810	Sia De Long

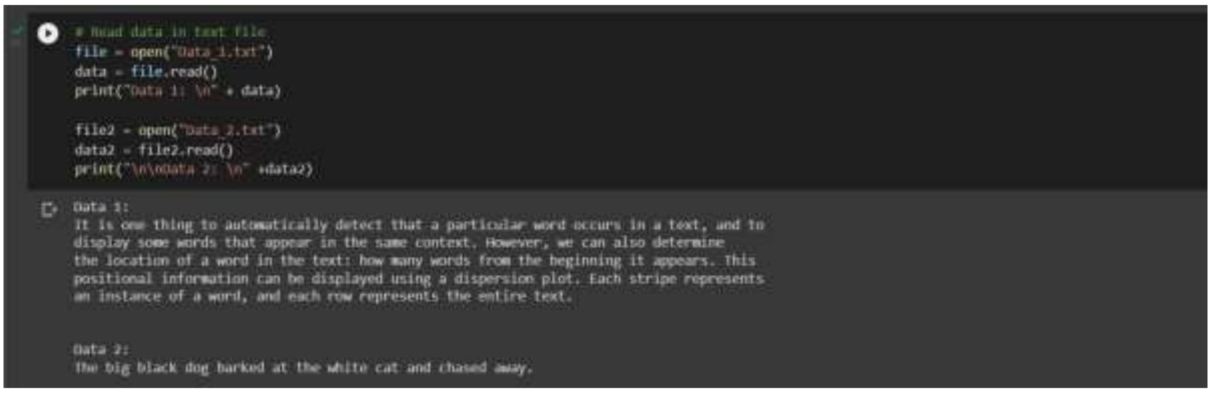
Table of Contents

Libraries Import and Data Reading



```python
# Import necessary libraries
import re
import nltk
nltk.download("punkt")
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')
from nltk.tokenize import sent_tokenize, word_tokenize, RegexpTokenizer
from nltk.stem import PorterStemmer, LancasterStemmer
from nltk.corpus import stopwords
from textblob import TextBlob
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
```

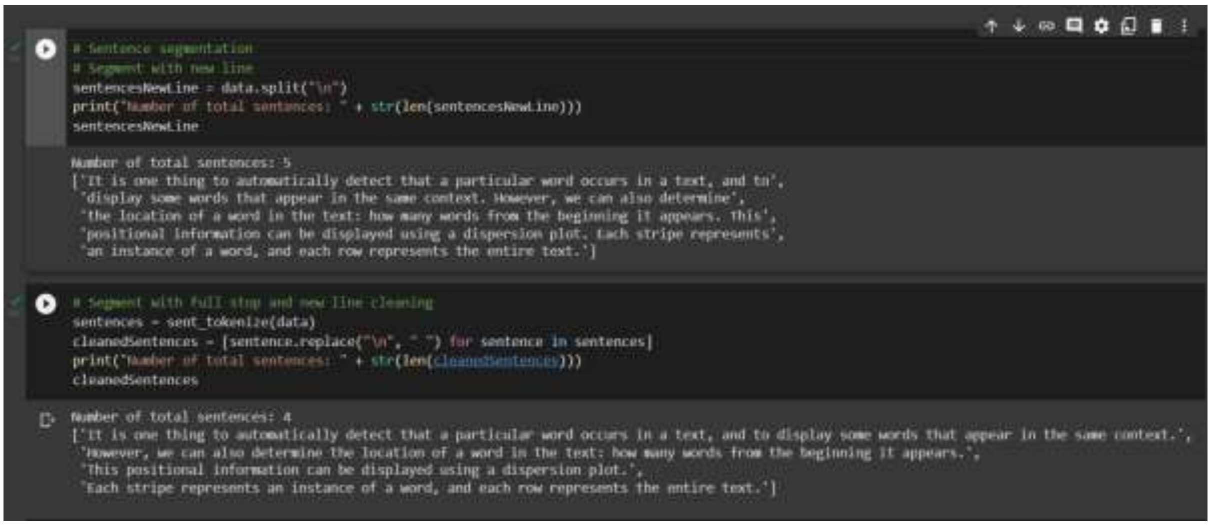


```python
# Read data in text file
file = open("Data_1.txt")
data = file.read()
print("Data 1: \n" + data)

file2 = open("Data_2.txt")
data2 = file2.read()
print("\n\nData 2: \n" +data2)
```

```
Data 1:
It is one thing to automatically detect that a particular word occurs in a text, and to
display some words that appear in the same context. However, we can also determine
the location of a word in the text: how many words from the beginning it appears. This
positional information can be displayed using a dispersion plot. Each stripe represents
an instance of a word, and each row represents the entire text.


Data 2:
The big black dog barked at the white cat and chased away.
```

1.0 Form Tokenization

1.1 Sentence Segmentation



```
# Sentence segmentation
# Segment with new line
sentencesNewLine = data.split("\n")
print("Number of total sentences: " + str(len(sentencesNewLine)))
sentencesNewLine
```

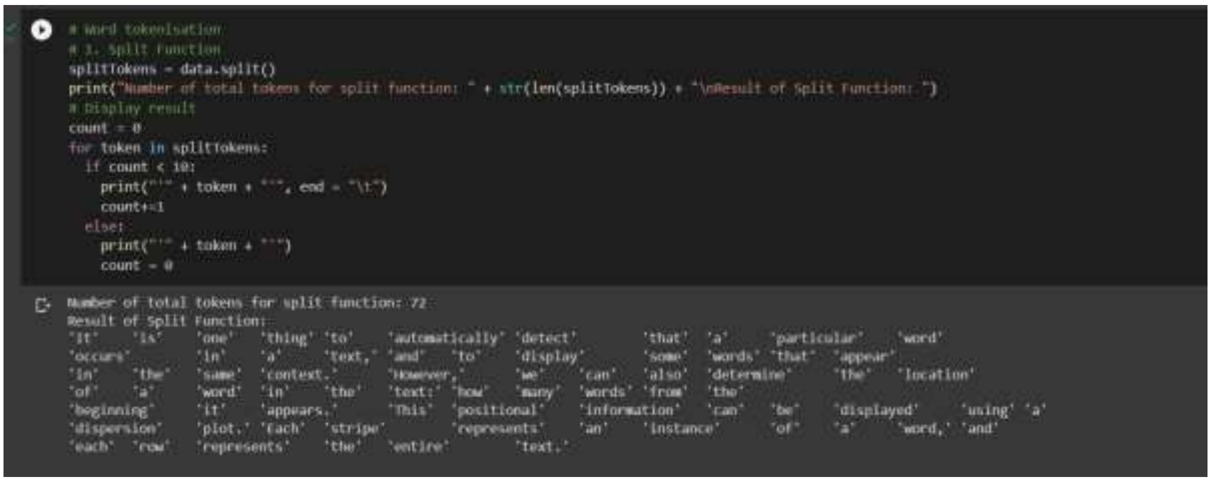
```
Number of total sentences: 5
['It is one thing to automatically detect that a particular word occurs in a text, and to',
 'display some words that appear in the same context. However, we can also determine',
 'the location of a word in the text: how many words from the beginning it appears. This',
 'positional information can be displayed using a dispersion plot. Each stripe represents',
 'an instance of a word, and each row represents the entire text.']
```

```
# Segment with full stop and new line cleaning
sentences = sent_tokenize(data)
cleanedSentences = [sentence.replace("\n", " ") for sentence in sentences]
print("Number of total sentences: " + str(len(cleanedSentences)))
cleanedSentences
```

```
Number of total sentences: 4
['It is one thing to automatically detect that a particular word occurs in a text, and to display some words that appear in the same context.',
 'However, we can also determine the location of a word in the text: how many words from the beginning it appears.',
 'This positional information can be displayed using a dispersion plot.',
 'Each stripe represents an instance of a word, and each row represents the entire text.']
```

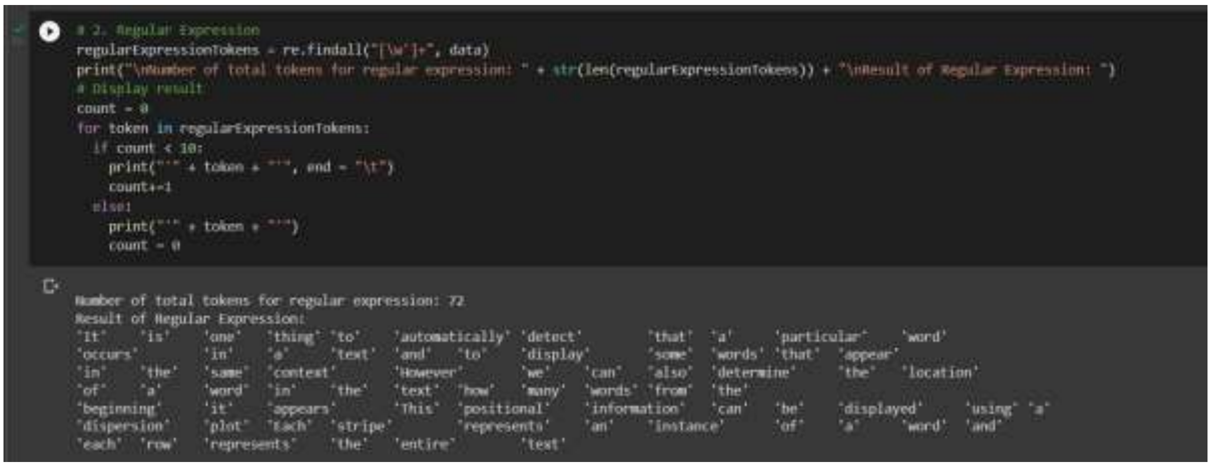
Since the data given in the file is not using a full stop punctuation as a line so there is two ways of segmenting the sentences where the first one will be using a new line as delimiter while the second one will be using a full stop as delimiter. The second one will be more appropriate as every sentence is complete with a full stop.

1.2 Word Tokenisation



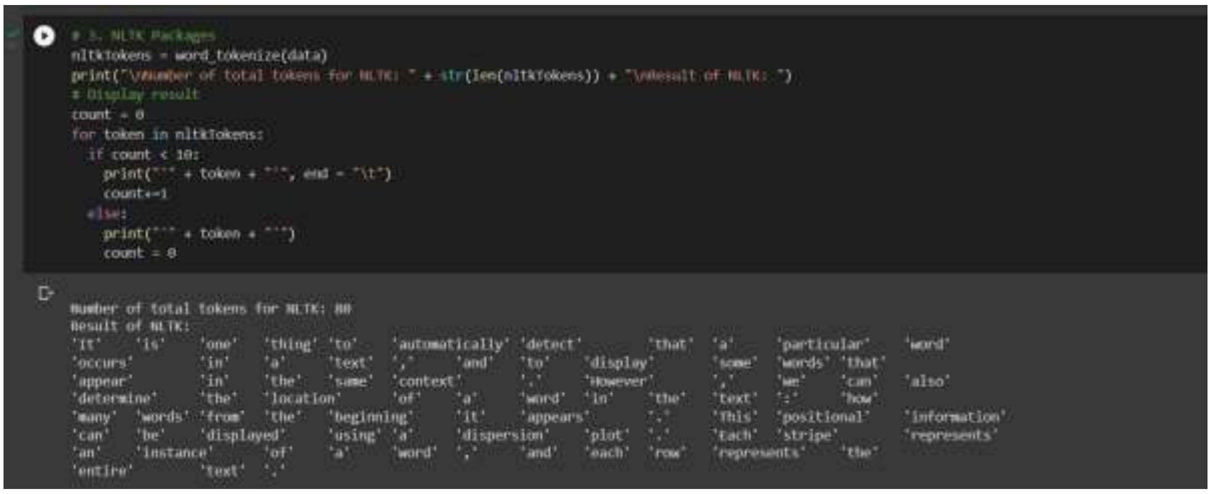
```
# Word tokenisation
# 1. Split Function
splitTokens = data.split()
print("Number of total tokens for split function: " + str(len(splitTokens)) + "\nResult of Split Function: ")
# Display result
count = 0
for token in splitTokens:
  if count < 10:
    print("'" + token + "'", end = "\t")
    count+=1
  else:
    print("'" + token + "'")
    count = 0
```

```
Number of total tokens for split function: 72
Result of Split Function:
'It'    'is'    'one'   'thing' 'to'    'automatically' 'detect'        'that'  'a'     'particular'    'word'
'occurs'        'in'    'a'     'text,' 'and'   'to'    'display'       'some'  'words' 'that'  'appear'
'in'    'the'   'same'  'context.'      'However,'      'we'    'can'   'also'  'determine'     'the'   'location'
'of'    'a'     'word'  'in'    'the'   'text:' 'how'   'many'  'words' 'from'  'the'
'beginning'     'it'    'appears.'      'This'  'positional'    'information'   'can'   'be'    'displayed'     'using' 'a'
'dispersion'    'plot.' 'Each'  'stripe'        'represents'    'an'    'instance'      'of'    'a'     'word,' 'and'
'each'  'row'   'represents'    'the'   'entire'        'text.'
```



```
# 2. Regular Expression
regularExpressionTokens = re.findall("[\w']+", data)
print("\nNumber of total tokens for regular expression: " + str(len(regularExpressionTokens)) + "\nResult of Regular Expression: ")
# Display result
count = 0
for token in regularExpressionTokens:
  if count < 10:
    print("'" + token + "'", end = "\t")
    count+=1
  else:
    print("'" + token + "'")
    count = 0
```

```
Number of total tokens for regular expression: 72
Result of Regular Expression:
'It'    'is'    'one'   'thing' 'to'    'automatically' 'detect'        'that'  'a'     'particular'    'word'
'occurs'        'in'    'a'     'text'  'and'   'to'    'display'       'some'  'words' 'that'  'appear'
'in'    'the'   'same'  'context'       'However'       'we'    'can'   'also'  'determine'     'the'   'location'
'of'    'a'     'word'  'in'    'the'   'text'  'how'   'many'  'words' 'from'  'the'
'beginning'     'it'    'appears'       'This'  'positional'    'information'   'can'   'be'    'displayed'     'using' 'a'
'dispersion'    'plot'  'Each'  'stripe'        'represents'    'an'    'instance'      'of'    'a'     'word'  'and'
'each'  'row'   'represents'    'the'   'entire'        'text'
```

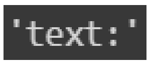


```
# 3. NLTK Packages
nltkTokens = word_tokenize(data)
print("\nNumber of total tokens for NLTK: " + str(len(nltkTokens)) + "\nResult of NLTK: ")
# Display result
count = 0
for token in nltkTokens:
  if count < 10:
    print("'" + token + "'", end = "\t")
    count+=1
  else:
    print("'" + token + "'")
    count = 0
```

```
Number of total tokens for NLTK: 88
Result of NLTK:
'It'    'is'    'one'   'thing' 'to'    'automatically' 'detect'        'that'  'a'     'particular'    'word'
'occurs'        'in'    'a'     'text'  ','     'and'   'to'    'display'       'some'  'words' 'that'
'appear'        'in'    'the'   'same'  'context'       '.'     'However'       ','     'we'    'can'   'also'
'determine'     'the'   'location'      'of'    'a'     'word'  'in'    'the'   'text'  ':'     'how'
'many'  'words' 'from'  'the'   'beginning'     'it'    'appears'       '.'     'This'  'positional'    'information'
'can'   'be'    'displayed'     'using' 'a'     'dispersion'    'plot'  '.'     'Each'  'stripe'        'represents'
'an'    'instance'      'of'    'a'     'word'  ','     'and'   'each'  'row'   'represents'    'the'
'entire'        'text'  '.'
```

1.3 Differences of The Tokenisation Operations

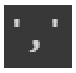
First and foremost, the split function tokenisation operation does not require any library imported in order to execute. It enables the data analyser to determine the delimiter in the split bracket, while it will consider white space as delimiter in default which is used in the source code. This operation will be simpler but the token it produced will not remove the punctuation and it will stick to the previous token and form as a single token which is shown as the figure below as an example.



```
'text:'
```

Moving on, the regular expression tokenisation operation does require the library named “re” in order to execute. It also enables the data analyser to determine the delimiter but in a more complex and details way by using regular expression symbol or notation while calling the function “findall”. This operation will be more complex depends on the regular expression that need to be defined depend on situation but the token it produced remove all the punctuation due to the fact that expression “[\w']+” was given to the function, so clean tokens is produced from this operation.

Last but not least, the NLTK tokenisation operation does require the library named “nltk” with “punkt” package downloaded, then import “word_tokenize” from “nltk.tokenize” in order to execute. It does enable the data analyser to determine the delimiter if the tokenizer is defined such as using “RegexpTokenizer”, but no tokenizer is defined in this operation and it will execute in default which is using white space as delimiter. This operation is also simple but the token it produced will not remove the punctuation and it will also be treated as one of the tokens shown as the figure below as an example.



```
','
```

1.4 Most Suitable Tokenisation Operation

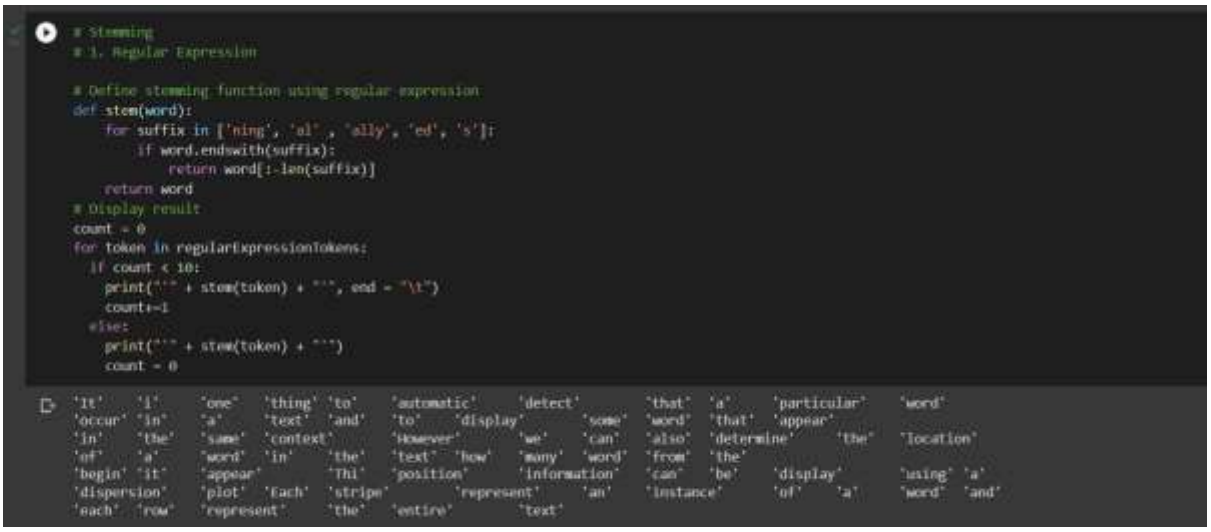
From the result of all tokenization operations, the most suitable operation would be NLTK tokenization operation because it is simple to fit every situation while it remains the punctuation and treat them as a single token which is very useful for sentiment analysis due to the fact that punctuation can express sentiments. However, the punctuations in the tokens can be removed if a data without punctuation is necessary for the analysis to treat every token equally.

2.0 Form Word Stemming

2.1 Importance of Stemming

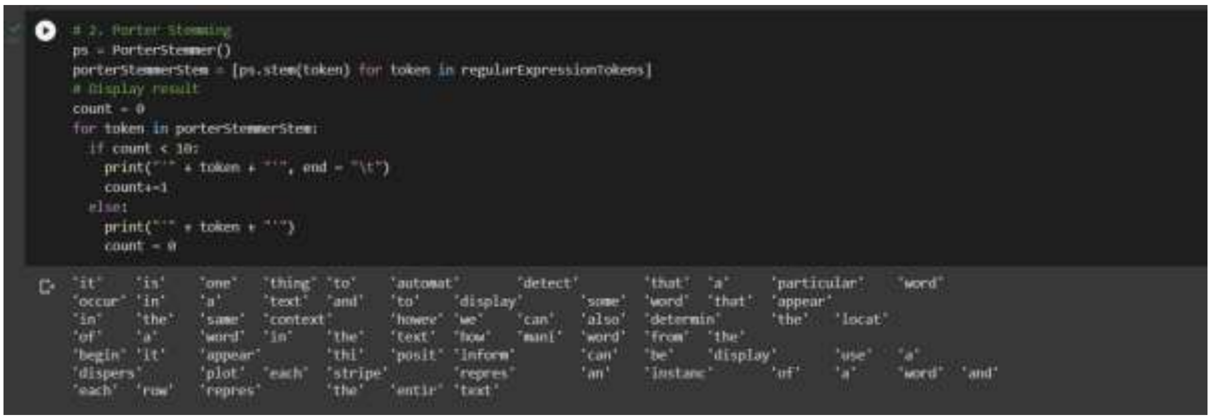
Stemming is important because it is a process that reducing a word to their root form called lemmas which mean will removing the prefix and subfix of the word. The computer or machine tends to be easier to understand these lemmas which help on natural language processing and analysis. Besides that, it will also help on search engine and information retrieval on reducing the number of results get missed out due to the system could not understand the form of word.

2.2 Word Stemming



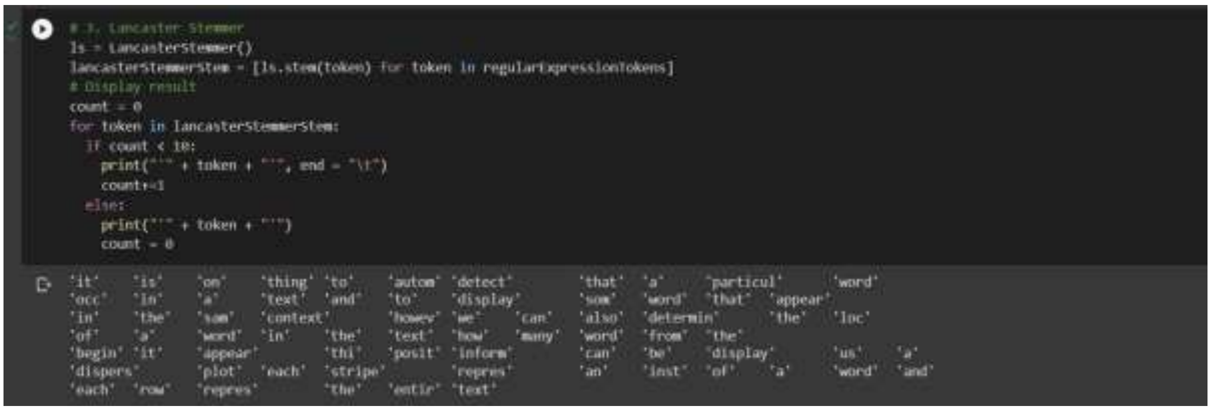
```python
# Stemming
# 1. Regular Expression

# Define stemming function using regular expression
def stem(word):
    for suffix in ['ning', 'al' , 'ally', 'ed', 's']:
        if word.endswith(suffix):
            return word[:-len(suffix)]
    return word
# Display result
count = 0
for token in regularExpressionTokens:
  if count < 10:
    print("'" + stem(token) + "'", end = "\t")
    count+=1
  else:
    print("'" + stem(token) + "'")
    count = 0
```

```
'It'    'i'     'one'   'thing' 'to'    'automatic'     'detect'        'that'  'a'     'particular'    'word'
'occur' 'in'    'a'     'text'  'and'   'to'    'display'       'some'  'word'  'that'  'appear'
'in'    'the'   'same'  'context'       'However'       'we'    'can'   'also'  'determine'     'the'   'location'
'of'    'a'     'word'  'in'    'the'   'text'  'how'   'many'  'word'  'from'  'the'
'begin' 'it'    'appear'        'Thi'   'position'      'information'   'can'   'be'    'display'       'using' 'a'
'dispersion'    'plot'  'Each'  'stripe'        'represent'     'an'    'instance'      'of'    'a'     'word'  'and'
'each'  'row'   'represent'     'the'   'entire'        'text'
```



```python
# 2. Porter Stemming
ps = PorterStemmer()
porterStemmerStem = [ps.stem(token) for token in regularExpressionTokens]
# Display result
count = 0
for token in porterStemmerStem:
  if count < 10:
    print("'" + token + "'", end = "\t")
    count+=1
  else:
    print("'" + token + "'")
    count = 0
```

```
'it'    'is'    'one'   'thing' 'to'    'automat'       'detect'        'that'  'a'     'particular'    'word'
'occur' 'in'    'a'     'text'  'and'   'to'    'display'       'some'  'word'  'that'  'appear'
'in'    'the'   'same'  'context'       'howev' 'we'    'can'   'also'  'determin'      'the'   'locat'
'of'    'a'     'word'  'in'    'the'   'text'  'how'   'mani'  'word'  'from'  'the'
'begin' 'it'    'appear'        'thi'   'posit' 'inform'        'can'   'be'    'display'       'use'   'a'
'dispers'       'plot'  'each'  'stripe'        'repres'        'an'    'instanc'       'of'    'a'     'word'  'and'
'each'  'row'   'repres'        'the'   'entir' 'text'
```



```python
# 3. Lancaster Stemmer
ls = LancasterStemmer()
lancasterStemmerStem = [ls.stem(token) for token in regularExpressionTokens]
# Display result
count = 0
for token in lancasterStemmerStem:
  if count < 10:
    print("'" + token + "'", end = "\t")
    count+=1
  else:
    print("'" + token + "'")
    count = 0
```

```
'it'    'is'    'on'    'thing' 'to'    'autom' 'detect'        'that'  'a'     'particul'      'word'
'occ'   'in'    'a'     'text'  'and'   'to'    'display'       'som'   'word'  'that'  'appear'
'in'    'the'   'sam'   'context'       'howev' 'we'    'can'   'also'  'determin'      'the'   'loc'
'of'    'a'     'word'  'in'    'the'   'text'  'how'   'many'  'word'  'from'  'the'
'begin' 'it'    'appear'        'thi'   'posit' 'inform'        'can'   'be'    'display'       'us'    'a'
'dispers'       'plot'  'each'  'stripe'        'repres'        'an'    'inst'  'of'    'a'     'word'  'and'
'each'  'row'   'repres'        'the'   'entir' 'text'
```

2.3 Differences of The Stemmer

First and foremost, the word stemming using regular expression method does not require any library imported in order to execute but a function is defined for code reuse purpose. It enables the analyser to determine the subfixes that need to be removed from the word that can always depends on the data, so a clean lemma will be produced with a specific subfix defined, but it will be not effective if the data is way too big to define the specific subfix and it may be dangerous where some common subfixes may be missed out to be removed. However, this stemming method produce the cleanest lemma out of all.

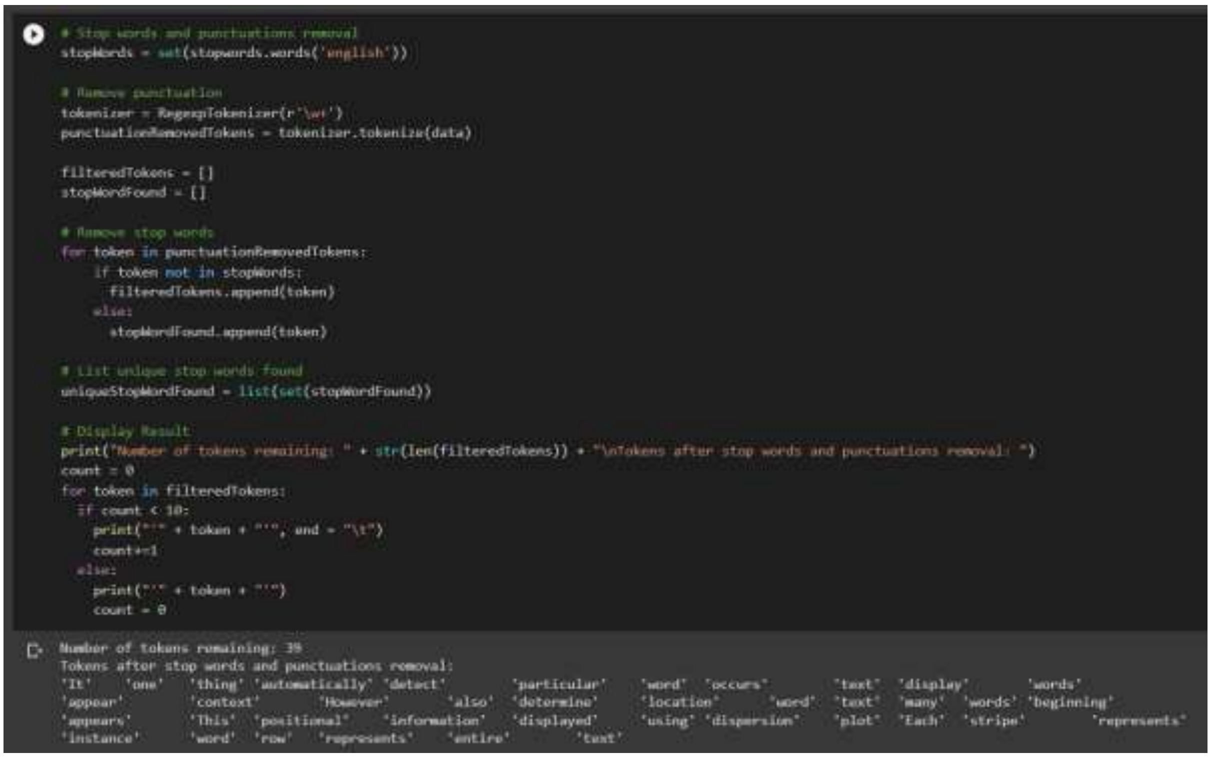
Moving on, the word stemming using Porter Stemmer and Lancaster Stemmer does require import of “PorterStemmer” and “LancasterStemmer” from the library named “nltk.stem”. The advantage of these two stemmers is that they have a defined and reliable subfixes which can be directly used for the purpose of stemming. Both of the stemmer is quite similar but Lancaster will be more aggressive on stemming where it strictly chopping the words and make it confusing while Porter will be least aggressive and make the word somewhat understandable.

2.4 Most Suitable Stemming Operation

From the result of all word stemming methods, the most suitable operation would be stemming using Porter Stemmer because it is reliable on the defined subfixes and simple to fit every situation while it is less aggressive compared to the Lancaster which mean the word after stemming process will still can be understand by the machine.

3.0 Filter Stop Words and Punctuation

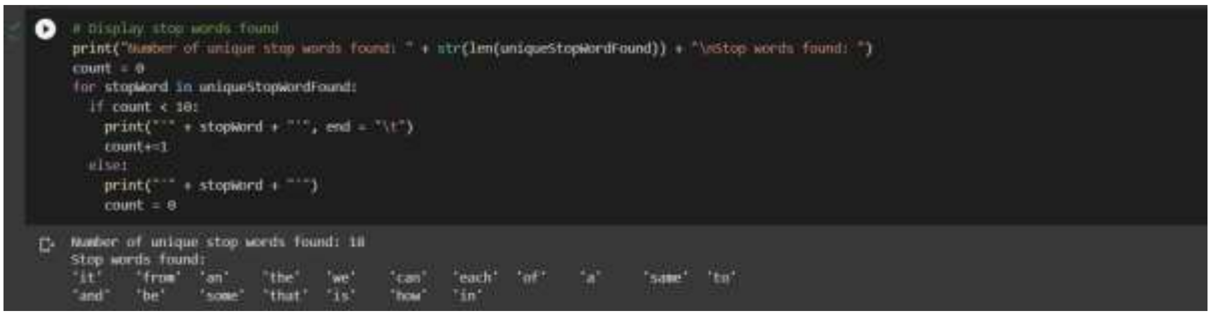
3.1 Stop Words and Punctuations Removal



```python
# Stop words and punctuations removal
stopWords = set(stopwords.words('english'))

# Remove punctuation
tokenizer = RegexpTokenizer(r'\w+')
punctuationRemovedTokens = tokenizer.tokenize(data)

filteredTokens = []
stopWordFound = []

# Remove stop words
for token in punctuationRemovedTokens:
    if token not in stopWords:
      filteredTokens.append(token)
    else:
      stopWordFound.append(token)

# List unique stop words found
uniqueStopWordFound = list(set(stopWordFound))

# Display Result
print("Number of tokens remaining: " + str(len(filteredTokens)) + "\nTokens after stop words and punctuations removal: ")
count = 0
for token in filteredTokens:
  if count < 10:
    print("'" + token + "'", end = "\t")
    count+=1
  else:
    print("'" + token + "'")
    count = 0
```

```
Number of tokens remaining: 39
Tokens after stop words and punctuations removal:
'It'    'one'   'thing' 'automatically' 'detect'        'particular'    'word'  'occurs'        'text'  'display'       'words'
'appear'        'context'       'However'       'also'  'determine'     'location'      'word'  'text'  'many'  'words' 'beginning'
'appears'       'This'  'positional'    'information'   'displayed'     'using' 'dispersion'    'plot'  'Each'  'stripe'        'represents'
'instance'      'word'  'row'   'represents'    'entire'        'text'
```

3.2 Stop Words Found



```
# Display stop words found
print("Number of unique stop words found: " + str(len(uniqueStopWordFound)) + "\nStop words found: ")
count = 0
for stopWord in uniqueStopWordFound:
  if count < 10:
    print("'" + stopWord + "'", end = "\t")
    count+=1
  else:
    print("'" + stopWord + "'")
    count = 0
```

```
Number of unique stop words found: 18
Stop words found:
'it'    'from'  'an'    'the'   'we'    'can'   'each'  'of'    'a'     'same'  'to'
'and'   'be'    'some'  'that'  'is'    'how'   'in'
```

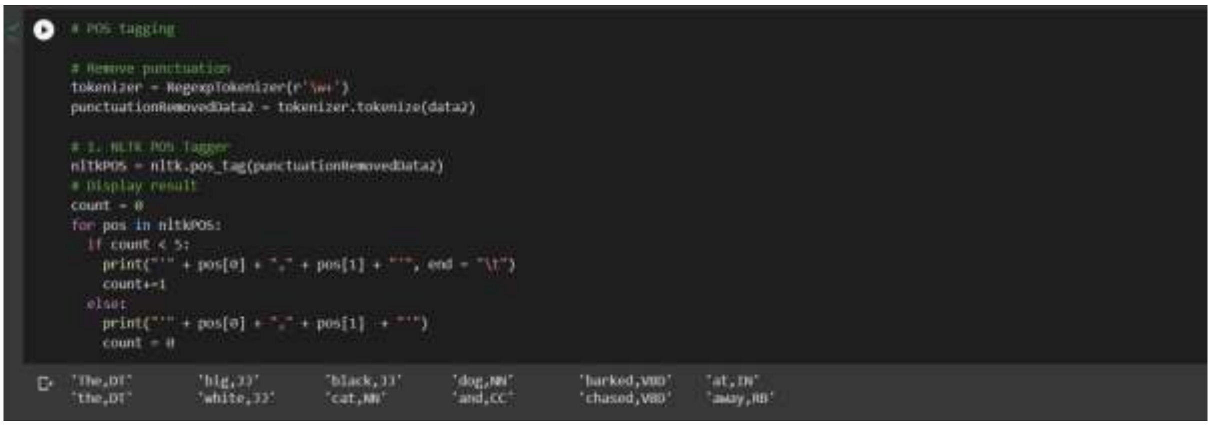
3.3 Importance of Filtering the Stop Words and Punctuations

For filtering stop words, the data is filtered out the low-level information which is abundance in any human language which will remain the important message for the machine or human to focus on while it also significantly reduces the data size which will lead to less storage equipment needed and faster data processing speed.

For filtering punctuations, it will help the data to treat every text equally since every process will not be different treated on every sentence because every punctuation is removed and making sentences cannot be determined anymore. Besides, it also produces a cleaner and smaller in size data that make the machine easier to process.

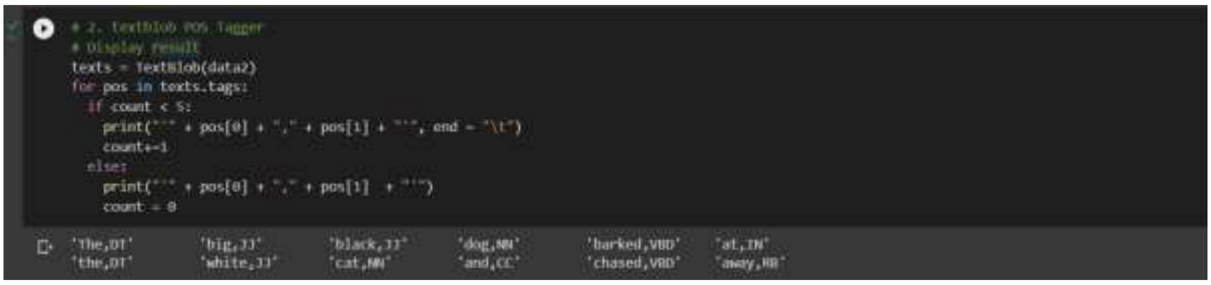
4.0 Form Parts of Speech (POS) Taggers & Syntactic Analysers

4.1 POS Tagging



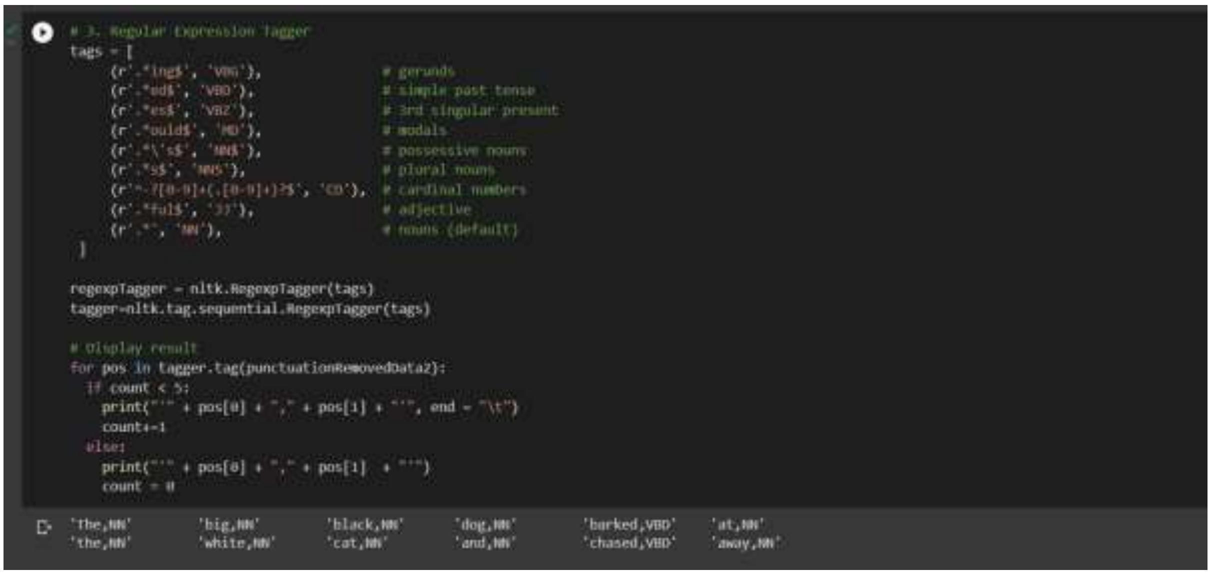
```python
# POS tagging

# Remove punctuation
tokenizer = RegexpTokenizer(r'\w+')
punctuationRemovedData2 = tokenizer.tokenize(data2)

# 1. NLTK POS Tagger
nltkPOS = nltk.pos_tag(punctuationRemovedData2)
# Display result
count = 0
for pos in nltkPOS:
  if count < 5:
    print("'" + pos[0] + "," + pos[1] + "'", end = "\t")
    count+=1
  else:
    print("'" + pos[0] + "," + pos[1]  + "'")
    count = 0
```

```
'The,DT'        'big,JJ'        'black,JJ'      'dog,NN'        'barked,VBD'    'at,IN'
'the,DT'        'white,JJ'      'cat,NN'        'and,CC'        'chased,VBD'    'away,RB'
```



```python
# 2. textblob POS Tagger
# Display result
texts = TextBlob(data2)
for pos in texts.tags:
  if count < 5:
    print("'" + pos[0] + "," + pos[1] + "'", end = "\t")
    count+=1
  else:
    print("'" + pos[0] + "," + pos[1]  + "'")
    count = 0
```

```
'The,DT'        'big,JJ'        'black,JJ'      'dog,NN'        'barked,VBD'    'at,IN'
'the,DT'        'white,JJ'      'cat,NN'        'and,CC'        'chased,VBD'    'away,RB'
```



```python
# 3. Regular Expression Tagger
tags = [
      (r'.*ing$', 'VBG'),               # gerunds
      (r'.*ed$', 'VBD'),                # simple past tense
      (r'.*es$', 'VBZ'),                # 3rd singular present
      (r'.*ould$', 'MD'),               # modals
      (r'.*\'s$', 'NN$'),               # possessive nouns
      (r'.*s$', 'NNS'),                 # plural nouns
      (r'^-?[0-9]+(.[0-9]+)?$', 'CD'),  # cardinal numbers
      (r'.*ful$', 'JJ'),                # adjective
      (r'.*', 'NN'),                    # nouns (default)
 ]

regexpTagger = nltk.RegexpTagger(tags)
tagger=nltk.tag.sequential.RegexpTagger(tags)

# Display result
for pos in tagger.tag(punctuationRemovedData2):
  if count < 5:
    print("'" + pos[0] + "," + pos[1] + "'", end = "\t")
    count+=1
  else:
    print("'" + pos[0] + "," + pos[1]  + "'")
    count = 0
```

```
'The,NN'        'big,NN'        'black,NN'      'dog,NN'        'barked,VBD'    'at,NN'
'the,NN'        'white,NN'      'cat,NN'        'and,NN'        'chased,VBD'    'away,NN'
```

4.2 Differences of The POS Taggers

First and foremost, the NLTK POS tagger does require the library named “nltk” with “averaged_perceptron_tagger” package downloaded, then import “RegexpTokenizer” from “nltk.tokenize” for preprocessing in order to execute. It tags all of the tokens without filtering the data which mean a punctuation removal is needed to be performed first before POS tagging. NLTK POS tagger is simple to fit every situation while and reliable with defined tags on common subfixes.

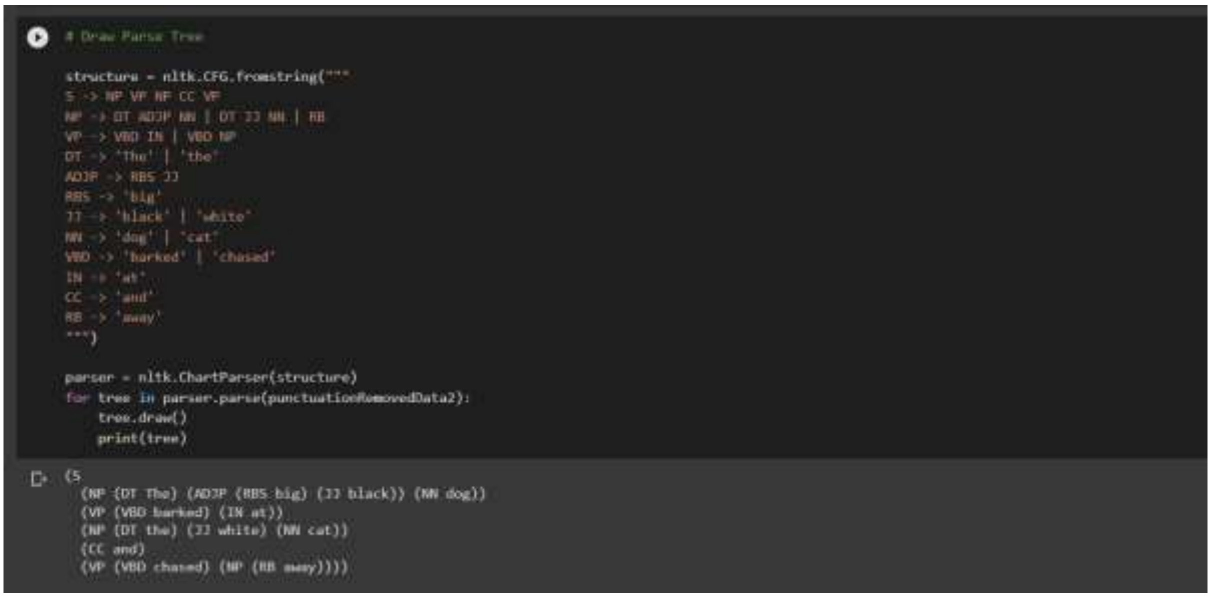
Moving on, the textblob POS tagger does require the library named “textblob” in order to execute. This tagger is similar to NLTK in term of simple to fit every situation and reliable but this tagger does remove punctuation automatically without any need to pre-processing the data.

Last but not least, the regular expression tagger is tagger that used NLTK Tag but with self-defined tags. It enables the analyser to determine the tags for specific subfixes but it will be not effective if the data is way too big to define the specific subfix and it may be dangerous where some tags may be missed out to be removed.

4.3 Most Suitable POS Tagger

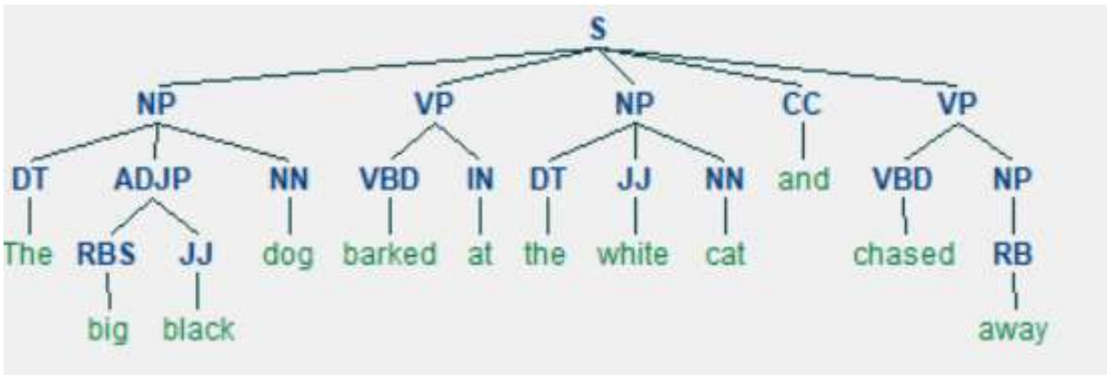
From the result of all POS taggers, the most suitable POS taggers would be textblob taggers because it is more reliable as it is built on top of NLTK, it provides easier interface to utilize the NLTK library and it is good for Natural Language Processing. Besides that, it helps on data pre-processing such as where it eases the effort to remove punctuation from the data before tagging the tokens.

4.4 Possible Parse Trees



```
# Draw Parse Tree

structure = nltk.CFG.fromstring("""
S -> NP VP NP CC VP
NP -> DT ADJP NN | DT JJ NN | RB
VP -> VBD IN | VBD NP
DT -> 'The' | 'the'
ADJP -> RBS JJ
RBS -> 'big'
JJ -> 'black' | 'white'
NN -> 'dog' | 'cat'
VBD -> 'barked' | 'chased'
IN -> 'at'
CC -> 'and'
RB -> 'away'
""")

parser = nltk.ChartParser(structure)
for tree in parser.parse(punctuationRemovedData2):
    tree.draw()
    print(tree)
```

```
(S
  (NP (DT The) (ADJP (RBS big) (JJ black)) (NN dog))
  (VP (VBD barked) (IN at))
  (NP (DT the) (JJ white) (NN cat))
  (CC and)
  (VP (VBD chased) (NP (RB away))))
```


S
NP
VP
NP
CC
VP
DT
ADJP
NN
VBD
IN
DT
JJ
NN
and
VBD
NP
The
RBS
JJ
dog
barked
at
the
white
cat
chased
RB
big
black
away