A · P · U
ASIA PACIFIC UNIVERSITY
OF TECHNOLOGY & INNOVATION

INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT107-3-3-TXSA

TEXT ANALYTICS AND SENTIMENT ANALYSIS

UC3F2011CS(DA)_IS

HAND OUT DATE: 15 DECEMBER 2020

HAND IN DATE: 12 MARCH 2021

WEIGHTAGE: 25%

INSTRUCTIONS TO CANDIDATES:

1 Submit your assignment at the administrative counter.

2 Students are advised to underpin their answers with the use of references (cited using the Harvard Name System of Referencing).

3 Late submission will be awarded zero (0) unless Extenuating Circumstances (EC) are upheld.

4 Cases of plagiarism will be penalized.

5 The assignment should be bound in an appropriate style (comb bound or stapled).

6 Where the assignment should be submitted in both hardcopy and softcopy, the softcopy of the written assignment and source code (where appropriate) should be on a CD in an envelope / CD cover and attached to the hardcopy.

7 You must obtain 50% overall to pass this module.

INDIVIDUAL ASSIGNMENT

CT107-3-3-TXSA

Text Analytics and Sentiment Analysis UC3F2011CS(DA)

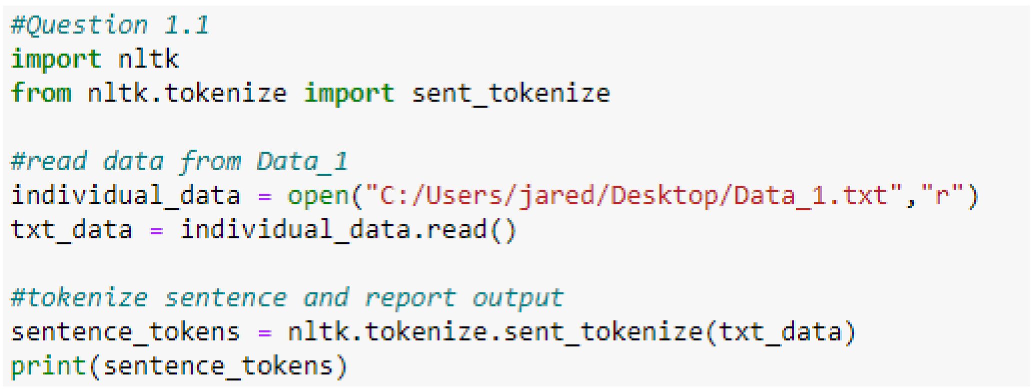
Name	Chan Jia Le	(TP049952)
Intake Code	UC3F2011CS(DA)	
Hand Out Date	15th December 2020	
Hand in Date	12th March 2020	

Table of Contents

1.0 Form Tokenization

1.1 Demonstrate sentence tokenization and report the output

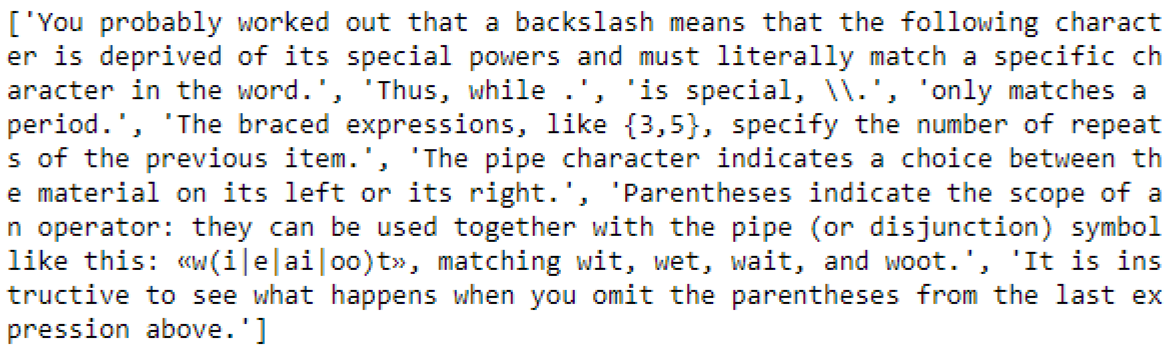
Python Source Code



```python
#Question 1.1
import nltk
from nltk.tokenize import sent_tokenize

#read data from Data_1
individual_data = open("C:/Users/jared/Desktop/Data_1.txt","r")
txt_data = individual_data.read()

#tokenize sentence and report output
sentence_tokens = nltk.tokenize.sent_tokenize(txt_data)
print(sentence_tokens)
```

Figure 1.1.1 Python Source Code for sentence tokenization

Output



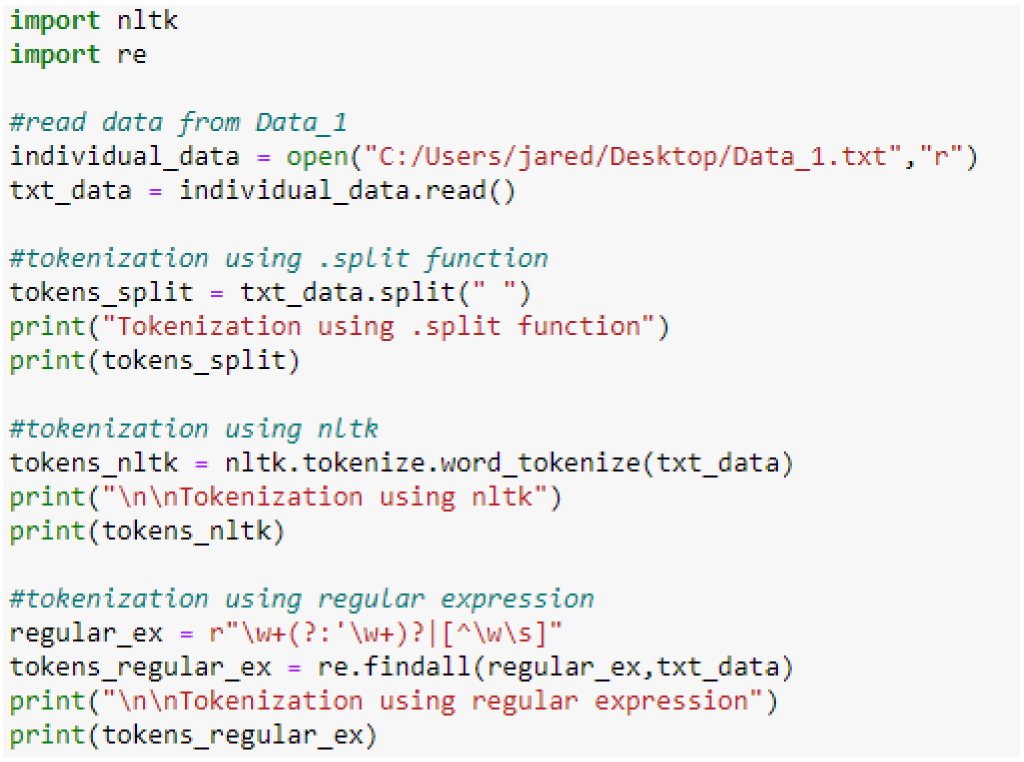
```
['You probably worked out that a backslash means that the following charact
er is deprived of its special powers and must literally match a specific ch
aracter in the word.', 'Thus, while .', 'is special, \\.', 'only matches a 
period.', 'The braced expressions, like {3,5}, specify the number of repeat
s of the previous item.', 'The pipe character indicates a choice between th
e material on its left or its right.', 'Parentheses indicate the scope of a
n operator: they can be used together with the pipe (or disjunction) symbol
like this: «w(i|e|ai|oo)t», matching wit, wet, wait, and woot.', 'It is ins
tructive to see what happens when you omit the parentheses from the last ex
pression above.']
```

Figure 1.1.2 Output for sentence tokenization

Figure 1.1.1 shows the source code for sentence tokenization. The data from Data_1 is read and pass to the program, upon completion of loading the data, NLTK package is utilized for sentence tokenization. Each sentence is tokenized based on the sentence ending character "." and the output is as shown in Figure 1.1.2.

1.2 Demonstrate word tokenisation using the split function and Regular Expression and NLTK packages separately and report the output.

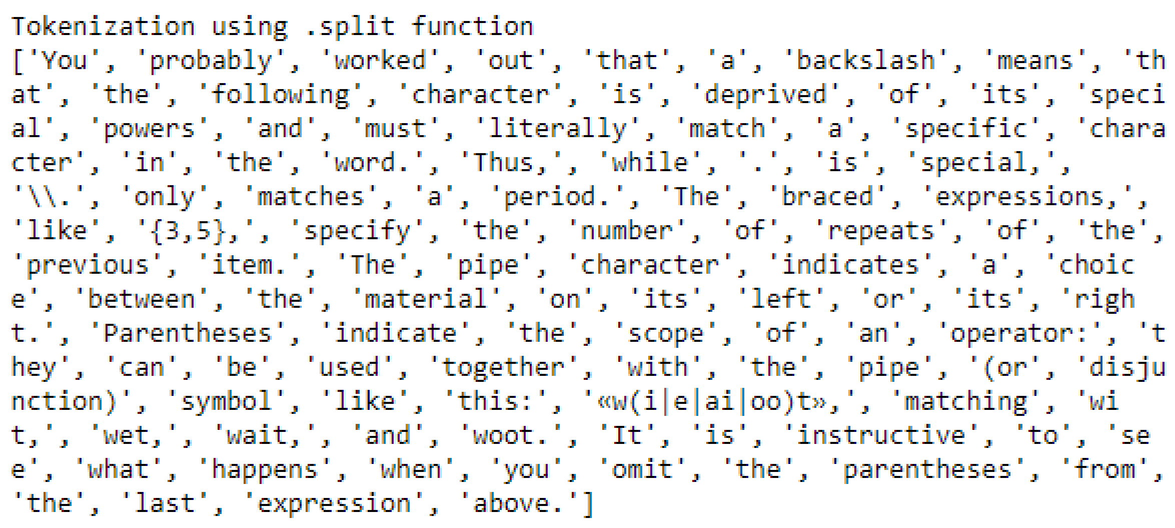
Python Source Code



```
import nltk
import re

#read data from Data_1
individual_data = open("C:/Users/jared/Desktop/Data_1.txt","r")
txt_data = individual_data.read()

#tokenization using .split function
tokens_split = txt_data.split(" ")
print("Tokenization using .split function")
print(tokens_split)

#tokenization using nltk
tokens_nltk = nltk.tokenize.word_tokenize(txt_data)
print("\n\nTokenization using nltk")
print(tokens_nltk)

#tokenization using regular expression
regular_ex = r"\w+(?:'\w+)?|[^\w\s]"
tokens_regular_ex = re.findall(regular_ex,txt_data)
print("\n\nTokenization using regular expression")
print(tokens_regular_ex)
```

Figure 1.2.1 Python Source Code for word tokenization using split, regular expression and NLTK package

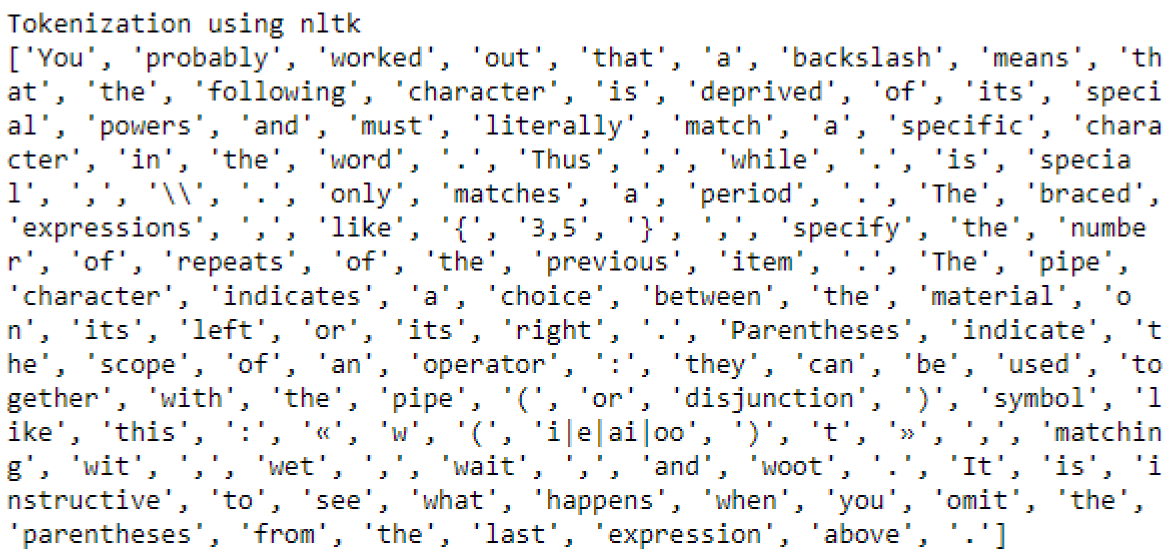
Output for Split function



```
Tokenization using .split function
['You', 'probably', 'worked', 'out', 'that', 'a', 'backslash', 'means', 'th
at', 'the', 'following', 'character', 'is', 'deprived', 'of', 'its', 'speci
al', 'powers', 'and', 'must', 'literally', 'match', 'a', 'specific', 'chara
cter', 'in', 'the', 'word.', 'Thus,', 'while', '.', 'is', 'special,',
'\\.', 'only', 'matches', 'a', 'period.', 'The', 'braced', 'expressions,',
'like', '{3,5},', 'specify', 'the', 'number', 'of', 'repeats', 'of', 'the',
'previous', 'item.', 'The', 'pipe', 'character', 'indicates', 'a', 'choic
e', 'between', 'the', 'material', 'on', 'its', 'left', 'or', 'its', 'righ
t.', 'Parentheses', 'indicate', 'the', 'scope', 'of', 'an', 'operator:', 't
hey', 'can', 'be', 'used', 'together', 'with', 'the', 'pipe', '(or', 'disju
nction)', 'symbol', 'like', 'this:', '«w(i|e|ai|oo)t»,', 'matching', 'wi
t,', 'wet,', 'wait,', 'and', 'woot.', 'It', 'is', 'instructive', 'to', 'se
e', 'what', 'happens', 'when', 'you', 'omit', 'the', 'parentheses', 'from',
'the', 'last', 'expression', 'above.']
```

Figure 1.2.2 Output for split function

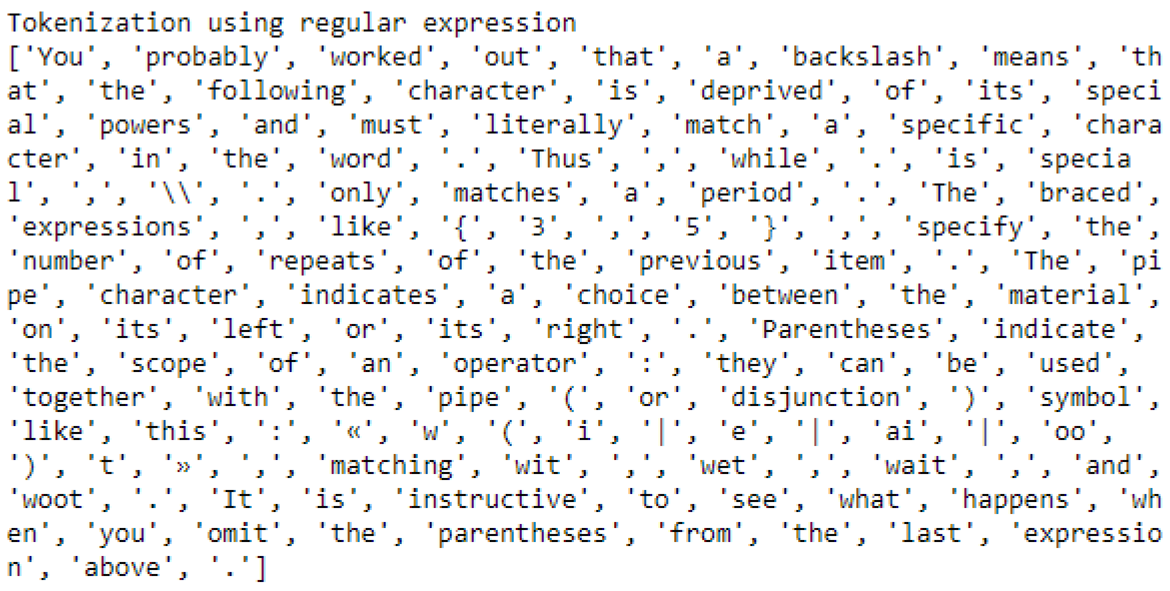
Output for NLTK



```
Tokenization using nltk
['You', 'probably', 'worked', 'out', 'that', 'a', 'backslash', 'means', 'th
at', 'the', 'following', 'character', 'is', 'deprived', 'of', 'its', 'speci
al', 'powers', 'and', 'must', 'literally', 'match', 'a', 'specific', 'chara
cter', 'in', 'the', 'word', '.', 'Thus', ',', 'while', '.', 'is', 'specia
l', ',', '\\', '.', 'only', 'matches', 'a', 'period', '.', 'The', 'braced',
'expressions', ',', 'like', '{', '3,5', '}', ',', 'specify', 'the', 'numbe
r', 'of', 'repeats', 'of', 'the', 'previous', 'item', '.', 'The', 'pipe',
'character', 'indicates', 'a', 'choice', 'between', 'the', 'material', 'o
n', 'its', 'left', 'or', 'its', 'right', '.', 'Parentheses', 'indicate', 't
he', 'scope', 'of', 'an', 'operator', ':', 'they', 'can', 'be', 'used', 'to
gether', 'with', 'the', 'pipe', '(', 'or', 'disjunction', ')', 'symbol', 'l
ike', 'this', ':', '«', 'w', '(', 'i|e|ai|oo', ')', 't', '»', ',', 'matchin
g', 'wit', ',', 'wet', ',', 'wait', ',', 'and', 'woot', '.', 'It', 'is', 'i
nstructive', 'to', 'see', 'what', 'happens', 'when', 'you', 'omit', 'the',
'parentheses', 'from', 'the', 'last', 'expression', 'above', '.']
```

Figure 2.2.3 Output for NLTK package

Output for Regular Expression



```
Tokenization using regular expression
['You', 'probably', 'worked', 'out', 'that', 'a', 'backslash', 'means', 'th
at', 'the', 'following', 'character', 'is', 'deprived', 'of', 'its', 'speci
al', 'powers', 'and', 'must', 'literally', 'match', 'a', 'specific', 'chara
cter', 'in', 'the', 'word', '.', 'Thus', ',', 'while', '.', 'is', 'specia
l', ',', '\\', '.', 'only', 'matches', 'a', 'period', '.', 'The', 'braced',
'expressions', ',', 'like', '{', '3', ',', '5', '}', ',', 'specify', 'the',
'number', 'of', 'repeats', 'of', 'the', 'previous', 'item', '.', 'The', 'pi
pe', 'character', 'indicates', 'a', 'choice', 'between', 'the', 'material',
'on', 'its', 'left', 'or', 'its', 'right', '.', 'Parentheses', 'indicate',
'the', 'scope', 'of', 'an', 'operator', ':', 'they', 'can', 'be', 'used',
'together', 'with', 'the', 'pipe', '(', 'or', 'disjunction', ')', 'symbol',
'like', 'this', ':', '«', 'w', '(', 'i', '|', 'e', '|', 'ai', '|', 'oo',
')', 't', '»', ',', 'matching', 'wit', ',', 'wet', ',', 'wait', ',', 'and',
'woot', '.', 'It', 'is', 'instructive', 'to', 'see', 'what', 'happens', 'wh
en', 'you', 'omit', 'the', 'parentheses', 'from', 'the', 'last', 'expressio
n', 'above', '.']
```

Figure 3.2.4 Output for NLTK package

Figure 1.2.1 shows the source code for different implementations of word tokenization which includes utilizing split function, NLTK package and regular expression. Figure 1.2.2, figure 1.2.3 and figure 1.2.4 shows the output of each implementation.

1.3 Explain the differences of both the tokenisation operations using the reported output.

Text Data	Output for Split Function	Output for NLTK Package	Output for Regular Expressions
\\.	'\\.'	'\\','.'	'\\','.'
(or disjunction)	'(or', 'disjunction)'	'(', 'or', 'disjunction', ')'	'(', 'or', 'disjunction', ')'
{3,5}	{3,5}	'{', "3,5", '}'	'{', '3', ',', '5', '}'
i|e|ai|oo	i|e|ai|oo	i|e|ai|oo	'i', '|', 'e', '|', 'ai', '|', 'oo'

Table 1.3.1 Comparison between different implementation of tokenization

Table 1.3.1 shows the comparison between the different output provided by different tokenization implementations which includes split function, NLTK package and Regular Expression. Furthermore, upon comparison, it is found that the split function only tokenizes words based on spacings, on the other hand NLTK and Regular Expressions performs tokenization differently. As it is displayed in the table, NLTK package tokenizes {3,5} into '{', "3,5", '}' and regular expressions tokenizes into '{', "3,5", '}'. Similar results can be seen in tokenizing i|e|ai|oo, whereby the Regular Expression tokenizes the word into 'i', '|', 'e', '|', 'ai', '|', 'oo' and NLTK does not perform any tokenization. Regular Expression possess a different approach when tokenizing words, regular expression splits tokens based on punctuation and words while NLTK only breaks down the token by splitting words and punctuation meaning that NLTK considers punctuation as a token.

1.4 Justify the most suitable tokenisation operation for text analytics. Support your answer using the output obtained in task 3.

Based on the comparison conducted in section 1.3, it is shown that Regular Expression is more detailed when it comes to identifying tokens and splitting it. Regular Expression is one of the most powerful tools for substituting, manipulating, and extracting string (McCombe, 2019). Regular Expression or RegEx in short possesses multiple functions like re.search(), match.start(), match.end(), re.match(), etc. In addition to that, RegEx can identify more variety and complex elements like vertical space, special characters, anchors and many more (Kasture, 2019). Hence, it is safe to say Regular Expression is more suitable to be implemented for tokenization operation in text analytics as it possesses more capabilities for different applications.

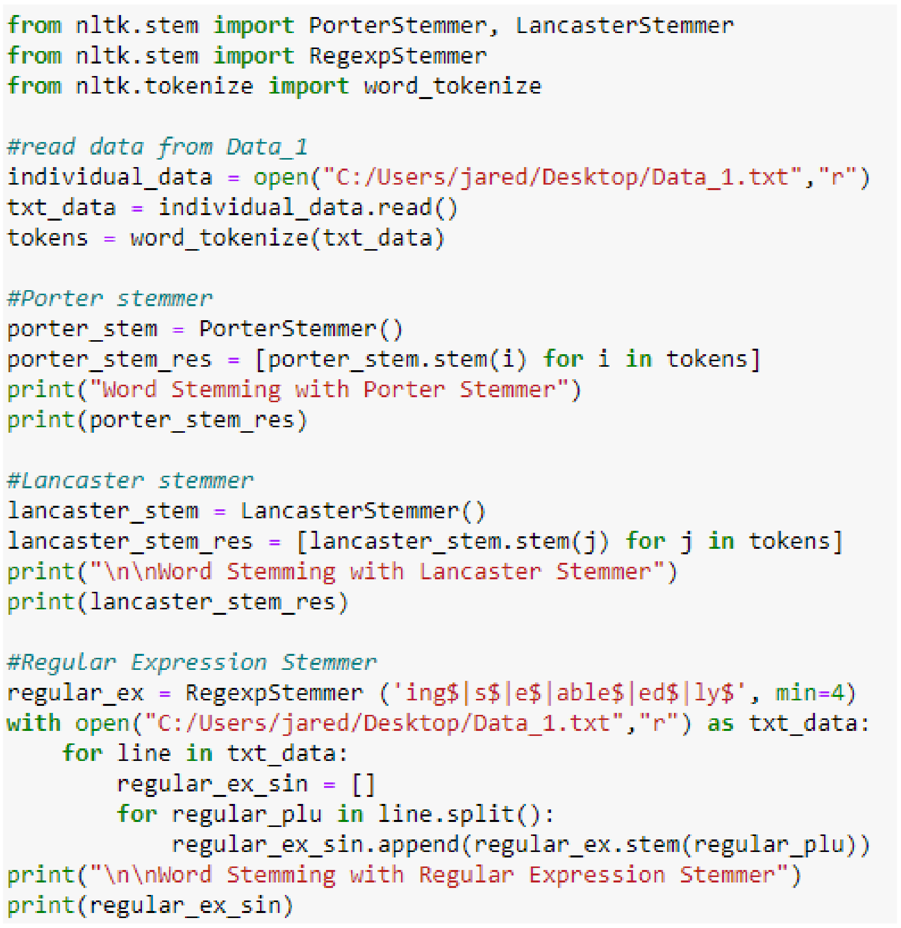
2.0 Form Word Stemming

2.1 Explain the importance of stemming in text analytics

Word stemming is referred as a crude heuristic process that terminates the ends of words. To simplify, word stemming reduces the words by removing its prefixes, suffix, or infix to find its original root form (Jabeen, 2018). For instance, when performing word stemming on the word “Playing”, the “ing” will be removed and the root word “Play” is the output. The implementation of stemming is highly crucial for text analytics as it allows analyst to reduce the dimension of the dataset by identifying words with same meaning, for instance “playing” and “plays” carries the same meaning and the root word is “play”, deleting additional words in the dictionary could greatly improve the performance. Furthermore, stemming possesses significance in information retrieval like word searching and the use of stemming also allows the analyst in retrieving more forms of word which enables additional results (Heidenreich, 2018) (Singal, 2020).

2.2 Demonstrate word stemming using *Regular Expression, Porter Stemmer* and *Lancaster Stemmer* and report the output.

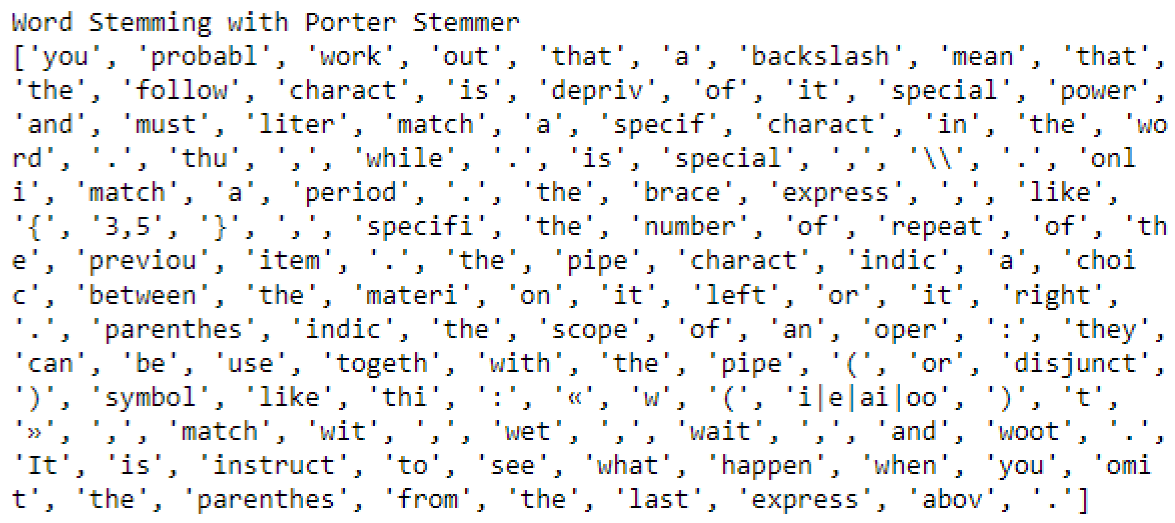
Python Source Code



```python
from nltk.stem import PorterStemmer, LancasterStemmer
from nltk.stem import RegexpStemmer
from nltk.tokenize import word_tokenize

#read data from Data_1
individual_data = open("C:/Users/jared/Desktop/Data_1.txt","r")
txt_data = individual_data.read()
tokens = word_tokenize(txt_data)

#Porter stemmer
porter_stem = PorterStemmer()
porter_stem_res = [porter_stem.stem(i) for i in tokens]
print("Word Stemming with Porter Stemmer")
print(porter_stem_res)

#Lancaster stemmer
lancaster_stem = LancasterStemmer()
lancaster_stem_res = [lancaster_stem.stem(j) for j in tokens]
print("\n\nWord Stemming with Lancaster Stemmer")
print(lancaster_stem_res)

#Regular Expression Stemmer
regular_ex = RegexpStemmer ('ing$|s$|e$|able$|ed$|ly$', min=4)
with open("C:/Users/jared/Desktop/Data_1.txt","r") as txt_data:
    for line in txt_data:
        regular_ex_sin = []
        for regular_plu in line.split():
            regular_ex_sin.append(regular_ex.stem(regular_plu))
print("\n\nWord Stemming with Regular Expression Stemmer")
print(regular_ex_sin)
```

Figure 2.2.1 Python Source Code for Word Stemming

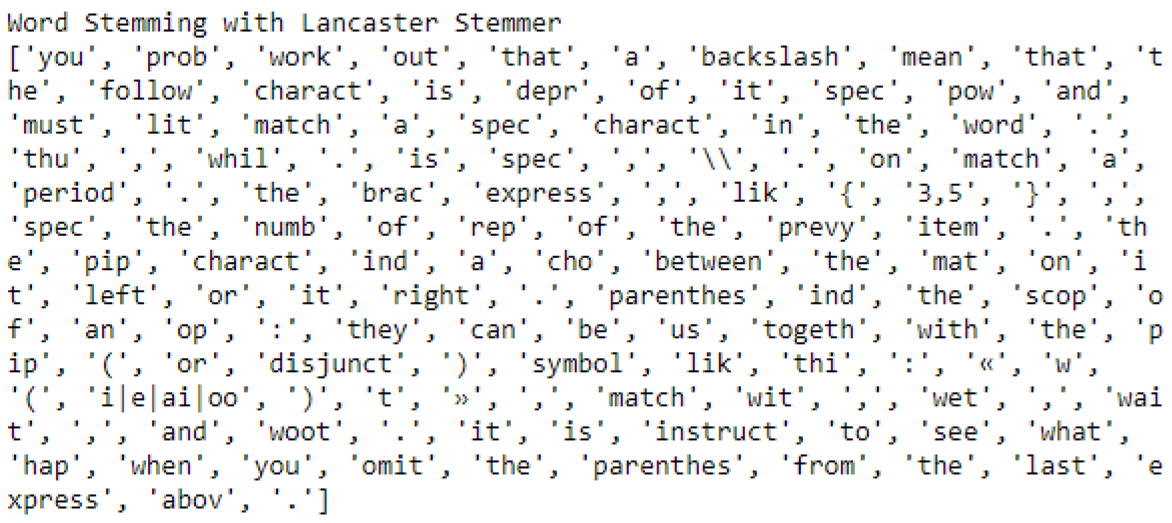
Output for Porter Stemmer



```
Word Stemming with Porter Stemmer
['you', 'probabl', 'work', 'out', 'that', 'a', 'backslash', 'mean', 'that',
'the', 'follow', 'charact', 'is', 'depriv', 'of', 'it', 'special', 'power',
'and', 'must', 'liter', 'match', 'a', 'specif', 'charact', 'in', 'the', 'wo
rd', '.', 'thu', ',', 'while', '.', 'is', 'special', ',', '\\', '.', 'onl
i', 'match', 'a', 'period', '.', 'the', 'brace', 'express', ',', 'like',
'{', '3,5', '}', ',', 'specifi', 'the', 'number', 'of', 'repeat', 'of', 'th
e', 'previou', 'item', '.', 'the', 'pipe', 'charact', 'indic', 'a', 'choi
c', 'between', 'the', 'materi', 'on', 'it', 'left', 'or', 'it', 'right',
'.', 'parenthes', 'indic', 'the', 'scope', 'of', 'an', 'oper', ':', 'they',
'can', 'be', 'use', 'togeth', 'with', 'the', 'pipe', '(', 'or', 'disjunct',
')', 'symbol', 'like', 'thi', ':', '«', 'w', '(', 'i|e|ai|oo', ')', 't',
'»', ',', 'match', 'wit', ',', 'wet', ',', 'wait', ',', 'and', 'woot', '.',
'It', 'is', 'instruct', 'to', 'see', 'what', 'happen', 'when', 'you', 'omi
t', 'the', 'parenthes', 'from', 'the', 'last', 'express', 'abov', '.']
```

Figure 2.2.2 Output of Word Stemming using Porter Stemmer

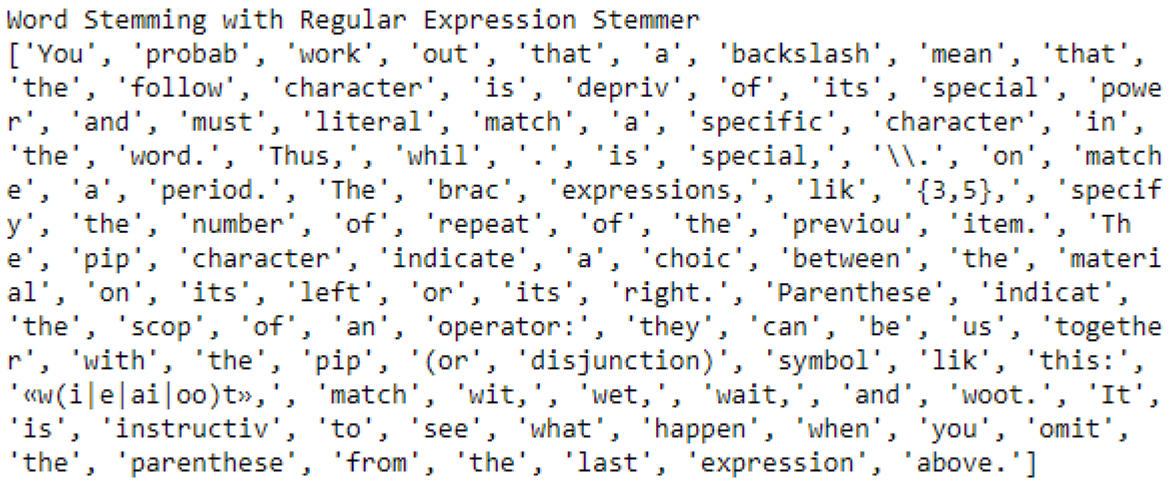
Output for Lancaster Stemmer



```
Word Stemming with Lancaster Stemmer
['you', 'prob', 'work', 'out', 'that', 'a', 'backslash', 'mean', 'that', 't
he', 'follow', 'charact', 'is', 'depr', 'of', 'it', 'spec', 'pow', 'and',
'must', 'lit', 'match', 'a', 'spec', 'charact', 'in', 'the', 'word', '.',
'thu', ',', 'whil', '.', 'is', 'spec', ',', '\\', '.', 'on', 'match', 'a',
'period', '.', 'the', 'brac', 'express', ',', 'lik', '{', '3,5', '}', ',',
'spec', 'the', 'numb', 'of', 'rep', 'of', 'the', 'prevy', 'item', '.', 'th
e', 'pip', 'charact', 'ind', 'a', 'cho', 'between', 'the', 'mat', 'on', 'i
t', 'left', 'or', 'it', 'right', '.', 'parenthes', 'ind', 'the', 'scop', 'o
f', 'an', 'op', ':', 'they', 'can', 'be', 'us', 'togeth', 'with', 'the', 'p
ip', '(', 'or', 'disjunct', ')', 'symbol', 'lik', 'thi', ':', '«', 'w',
'(', 'i|e|ai|oo', ')', 't', '»', ',', 'match', 'wit', ',', 'wet', ',', 'wai
t', ',', 'and', 'woot', '.', 'it', 'is', 'instruct', 'to', 'see', 'what',
'hap', 'when', 'you', 'omit', 'the', 'parenthes', 'from', 'the', 'last', 'e
xpress', 'abov', '.']
```

Figure 2.2.3 Output of Word Stemming using Lancaster Stemmer

Output for Regular Expression Stemmer



```
Word Stemming with Regular Expression Stemmer
['You', 'probab', 'work', 'out', 'that', 'a', 'backslash', 'mean', 'that',
'the', 'follow', 'character', 'is', 'depriv', 'of', 'its', 'special', 'powe
r', 'and', 'must', 'literal', 'match', 'a', 'specific', 'character', 'in',
'the', 'word.', 'Thus,', 'whil', '.', 'is', 'special,', '\\.', 'on', 'match
e', 'a', 'period.', 'The', 'brac', 'expressions,', 'lik', '{3,5},', 'specif
y', 'the', 'number', 'of', 'repeat', 'of', 'the', 'previou', 'item.', 'Th
e', 'pip', 'character', 'indicate', 'a', 'choic', 'between', 'the', 'materi
al', 'on', 'its', 'left', 'or', 'its', 'right.', 'Parenthese', 'indicat',
'the', 'scop', 'of', 'an', 'operator:', 'they', 'can', 'be', 'us', 'togethe
r', 'with', 'the', 'pip', '(or', 'disjunction)', 'symbol', 'lik', 'this:',
'«w(i|e|ai|oo)t»,', 'match', 'wit,', 'wet,', 'wait,', 'and', 'woot.', 'It',
'is', 'instructiv', 'to', 'see', 'what', 'happen', 'when', 'you', 'omit',
'the', 'parenthese', 'from', 'the', 'last', 'expression', 'above.']
```

Figure 2.2.4 Output of Word Stemming using Regular Expression Stemmer

Figure 2.2.1 shows the source code for different implementations of word stemming which includes Porter stemmer, Lancaster stemmer and Regular Expression Stemmer. Figure 2.2.2, figure 2.2.3 and figure 2.2.4 shows the output of each implementation. Though all are performing word stemming, the output has some slight difference which will be further explained in the section 2.3.

2.3 Explain the differences among *Regular Expression stemmer, Porter Stemmer* and *Lancaster Stemmer* using the obtained output.

Text Data	Porter Stemmer	Lancaster Stemmer	Regular Expression Stemmer
probably	probabl	prob	probab
deprive	depriv	depr	depriv
Its	it	it	its
power	power	pow	power
literally	liter	lit	literal
specific	specif	spec	specific
special	special	spec	special
indicate	indic	ind	indicate

Table 2.3.1 Comparison of Output between different stemming methods

Upon completion of a series of comparisons as shown in Table 2.3.1, it is identified that stemming using regular expressions possesses the highest accuracy compared to porter and Lancaster stemmer. In comparison of ‘literally’, ‘specific’, ‘power’, ‘special’ and ‘indicate’, it is shown that regular expression stemmer produces little to no error in root words as compared to Porter Stemmer and Lancaster Stemmer which converted the words into something that doesn’t have any meaning. Though there are still some errors using regular expression when converting root words like ‘deprive’, ‘probably’ but compared to Porter Stemmer and Lancaster Stemmer, it is the one with the highest accuracy. Thus, Lancaster Stemmer possesses the worst accuracy resulting in many erroneous root words.

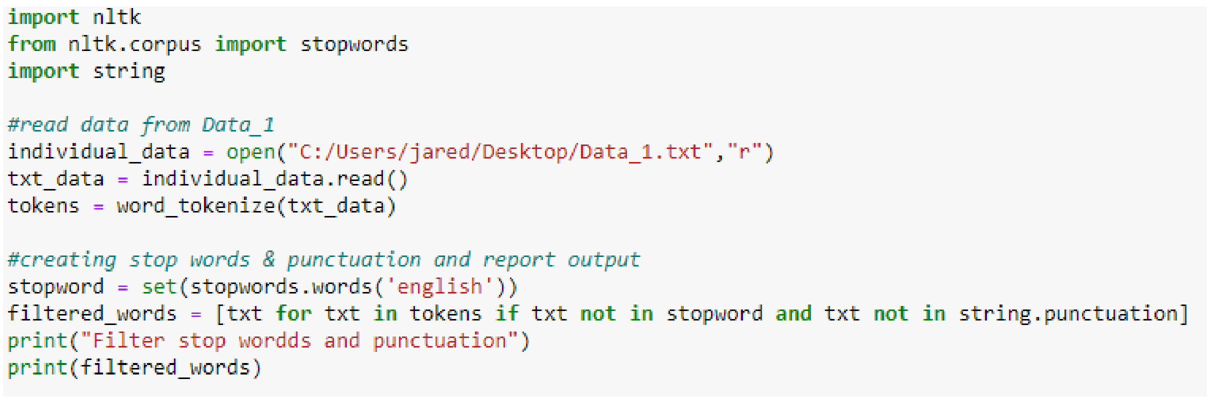
2.4 Justify the most suitable stemming operation for text analytics. Support your answer using the output obtained in task 2.

Based on the comparison conducted in section 1.3 and the output obtained in section 1.2, it indicates that Regular Expression for words stemming contains the highest accuracy and is the most suitable stemming approach in the bunch, as the output root words produced, though contains some errors but it still is the closest to the root words as compared to the other approaches. Furthermore, regular expression is more flexible and possesses more flexibility which enables the analyst to tailor the model based on different cases resulting in a higher accuracy stemming.

3.0 Filter Stop Words and Punctuation

3.1 Demonstrate how to filter the *stop words and punctuations* from the given text corpus and report the output after filtering.

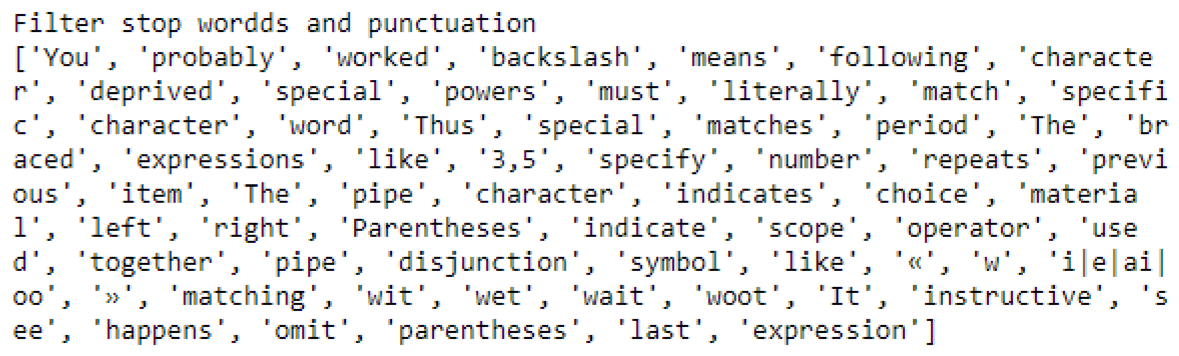
Python Source Code



```
import nltk
from nltk.corpus import stopwords
import string

#read data from Data_1
individual_data = open("C:/Users/jared/Desktop/Data_1.txt","r")
txt_data = individual_data.read()
tokens = word_tokenize(txt_data)

#creating stop words & punctuation and report output
stopword = set(stopwords.words('english'))
filtered_words = [txt for txt in tokens if txt not in stopword and txt not in string.punctuation]
print("Filter stop wordds and punctuation")
print(filtered_words)
```

Figure 3.1.1 Python Source Code for Filtering Stop words and Punctuations

Output for Filtering Stop words and Punctuation



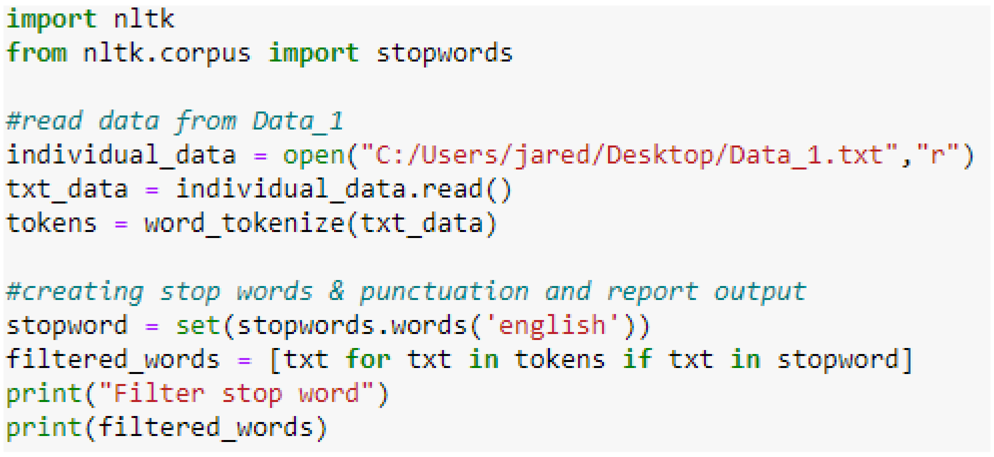
```
Filter stop wordds and punctuation
['You', 'probably', 'worked', 'backslash', 'means', 'following', 'characte
r', 'deprived', 'special', 'powers', 'must', 'literally', 'match', 'specifi
c', 'character', 'word', 'Thus', 'special', 'matches', 'period', 'The', 'br
aced', 'expressions', 'like', '3,5', 'specify', 'number', 'repeats', 'previ
ous', 'item', 'The', 'pipe', 'character', 'indicates', 'choice', 'materia
l', 'left', 'right', 'Parentheses', 'indicate', 'scope', 'operator', 'use
d', 'together', 'pipe', 'disjunction', 'symbol', 'like', '«', 'w', 'i|e|ai|
oo', '»', 'matching', 'wit', 'wet', 'wait', 'woot', 'It', 'instructive', 's
ee', 'happens', 'omit', 'parentheses', 'last', 'expression']
```

Figure 3.1.2 Output for Filtering Stop words and punctuation

Figure 3.1.1 shows the source code for the implementation of filtering stop words and punctuation, the data will first be tokenized using NLTK and stopwords package is imported to filter the stop words and punctuation. Upon completion of the filtering, the results are as shown in figure 3.1.2 whereby each word are filtered based on not in the stopword and punctuation of the English language.

3.2 Report the output of the stop words found in the given text corpus.

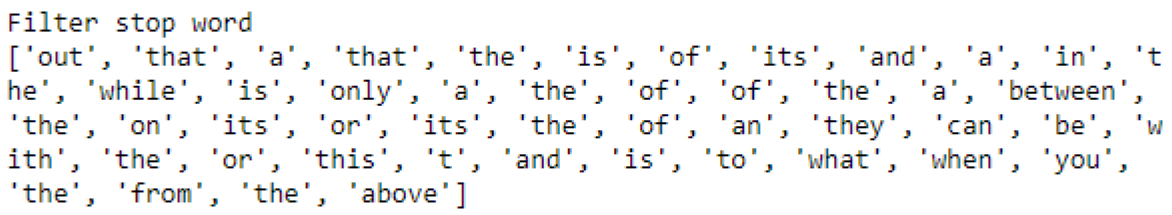
Python Source Code



```python
import nltk
from nltk.corpus import stopwords

#read data from Data_1
individual_data = open("C:/Users/jared/Desktop/Data_1.txt","r")
txt_data = individual_data.read()
tokens = word_tokenize(txt_data)

#creating stop words & punctuation and report output
stopword = set(stopwords.words('english'))
filtered_words = [txt for txt in tokens if txt in stopword]
print("Filter stop word")
print(filtered_words)
```

Figure 3.2.1 Python Source Code for finding stop words

Output for searching stop words



```
Filter stop word
['out', 'that', 'a', 'that', 'the', 'is', 'of', 'its', 'and', 'a', 'in', 't
he', 'while', 'is', 'only', 'a', 'the', 'of', 'of', 'the', 'a', 'between',
'the', 'on', 'its', 'or', 'its', 'the', 'of', 'an', 'they', 'can', 'be', 'w
ith', 'the', 'or', 'this', 't', 'and', 'is', 'to', 'what', 'when', 'you',
'the', 'from', 'the', 'above']
```

Figure 3.2.2 Output for Searching Stop Words

Figure 3.1.1 shows the source code for the implementation for searching stopwords, the data will first be tokenized using NLTK and stop word package is included for importing stop words for English, after that, the filtered words will search whether the tokens are stop words with an if statement. Upon completion of the search, the results are as shown in figure 3.2.2 which is the stopwords contained in the text data.

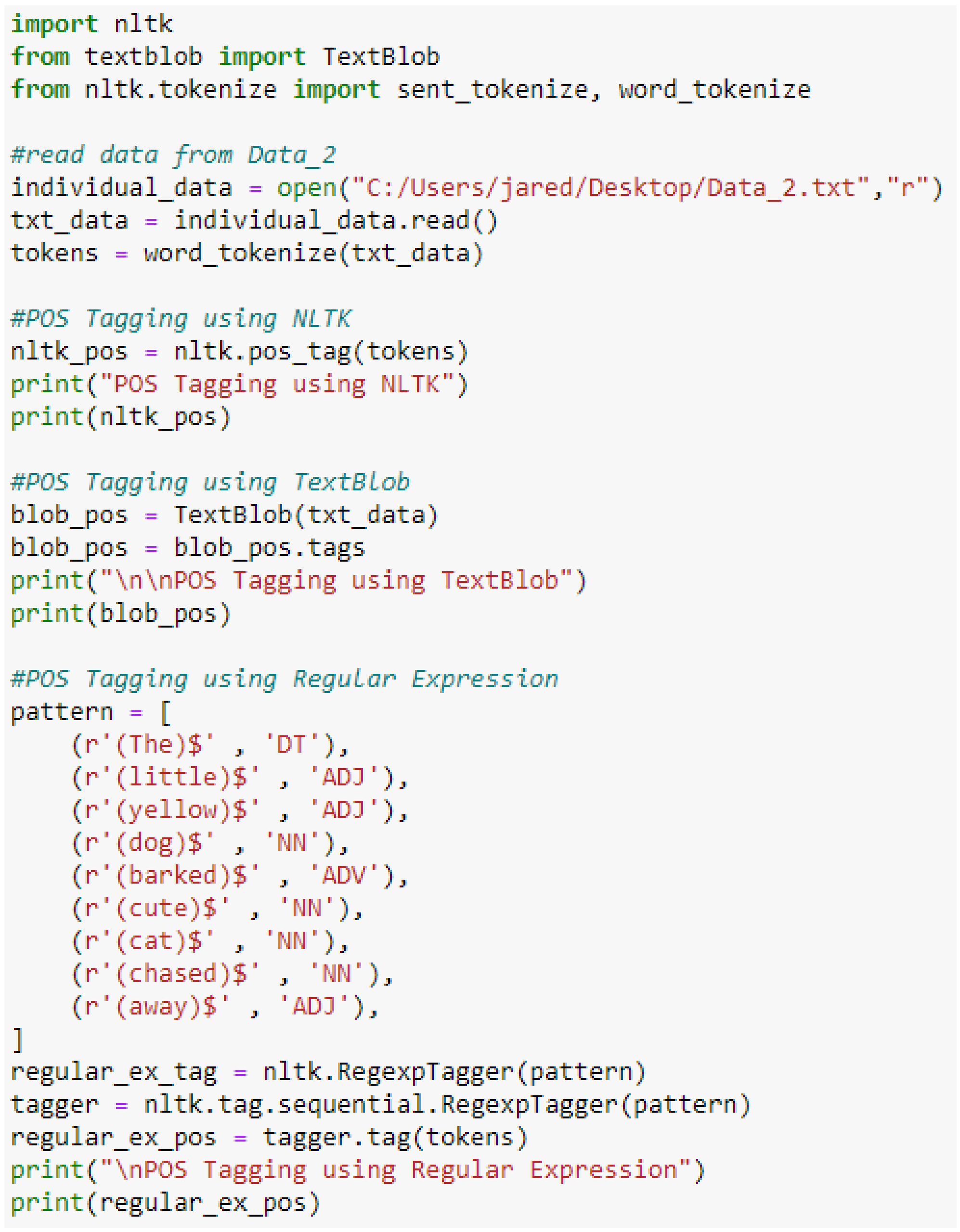
3.3 Explain the importance of filtering the stop words and punctuations in text analytics.

Stop words in the English language is words that doesn’t contain any meaning, but it is used for connecting words to construct a sentence like “the”, “is”, “a” etc. (Brownlee, 2017) whereas punctuation is tools utilized for separating or ending a sentence in English like comma (,) or full stop (.) (Liam, 2020). The removal of punctuation and stop words enables easier identification of key words and features of the text data and it also reduces the size of the text data allowing for better efficiency when conducting text analytics. Furthermore, by filtering stopwords and punctuation, it allows the analyst to construct better model with a better focus on words that contain meaning which could result in a more accurate text analytics model (Singal, 2020).

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

4.1 Demonstrate POS tagging using *NLTK POS tagger, textblob POS tagger* and the R*egular Expression tagger* and report the output.

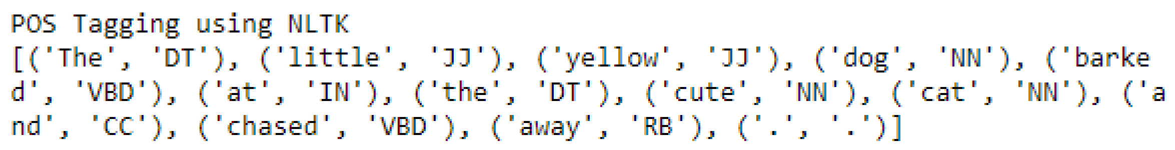
Python Source Code



```python
import nltk
from textblob import TextBlob
from nltk.tokenize import sent_tokenize, word_tokenize

#read data from Data_2
individual_data = open("C:/Users/jared/Desktop/Data_2.txt","r")
txt_data = individual_data.read()
tokens = word_tokenize(txt_data)

#POS Tagging using NLTK
nltk_pos = nltk.pos_tag(tokens)
print("POS Tagging using NLTK")
print(nltk_pos)

#POS Tagging using TextBlob
blob_pos = TextBlob(txt_data)
blob_pos = blob_pos.tags
print("\n\nPOS Tagging using TextBlob")
print(blob_pos)

#POS Tagging using Regular Expression
pattern = [
    (r'(The)$' , 'DT'),
    (r'(little)$' , 'ADJ'),
    (r'(yellow)$' , 'ADJ'),
    (r'(dog)$' , 'NN'),
    (r'(barked)$' , 'ADV'),
    (r'(cute)$' , 'NN'),
    (r'(cat)$' , 'NN'),
    (r'(chased)$' , 'NN'),
    (r'(away)$' , 'ADJ'),
]
regular_ex_tag = nltk.RegexpTagger(pattern)
tagger = nltk.tag.sequential.RegexpTagger(pattern)
regular_ex_pos = tagger.tag(tokens)
print("\nPOS Tagging using Regular Expression")
print(regular_ex_pos)
```

Figure 4.1.1 Python Source Code for different POS Tagger approaches

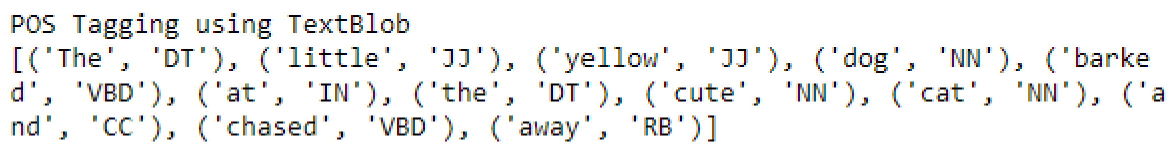
Output for NLTK POS tagging



```
POS Tagging using NLTK
[('The', 'DT'), ('little', 'JJ'), ('yellow', 'JJ'), ('dog', 'NN'), ('barke
d', 'VBD'), ('at', 'IN'), ('the', 'DT'), ('cute', 'NN'), ('cat', 'NN'), ('a
nd', 'CC'), ('chased', 'VBD'), ('away', 'RB'), ('.', '.')]
```

Figure 4.1.2 Output for POS Tagging using NLTK

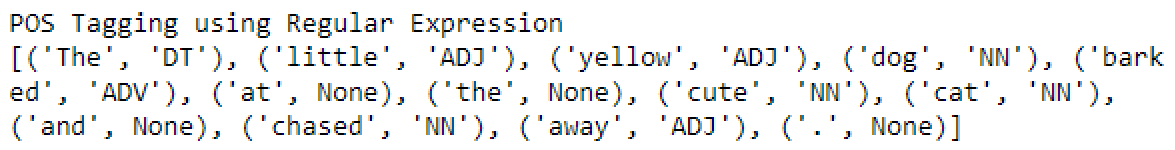
Output for TextBlob POS tagging



```
POS Tagging using TextBlob
[('The', 'DT'), ('little', 'JJ'), ('yellow', 'JJ'), ('dog', 'NN'), ('barke
d', 'VBD'), ('at', 'IN'), ('the', 'DT'), ('cute', 'NN'), ('cat', 'NN'), ('a
nd', 'CC'), ('chased', 'VBD'), ('away', 'RB')]
```

Figure 4.1.3 Output for POS Tagging using TextBlob

Output for Regular Expression POS tagging



```
POS Tagging using Regular Expression
[('The', 'DT'), ('little', 'ADJ'), ('yellow', 'ADJ'), ('dog', 'NN'), ('bark
ed', 'ADV'), ('at', None), ('the', None), ('cute', 'NN'), ('cat', 'NN'),
('and', None), ('chased', 'NN'), ('away', 'ADJ'), ('.', None)]
```

Figure 4.1.4 Output for POS Tagging using Regular Expression

Figure 3.1.1 shows the source code for the implementation of POS Tagging using different operations which includes NLTK package, TextBlob and Regular Expression from NLTK, the data will first be tokenized using NLTK and different approaches will use the token to perform POS Tagging as shown in figure 3.1.1. Upon completion of the POS Tagging, the results are as shown in Figure 4.1.2, 4.1.3 and 4.1.4.

4.2 Explain the differences of the POS taggers using the output obtained in the above question.

POS Tagging using NLTK	POS Tagging using TextBlob	POS Tagging using Regular Expression
('the', 'DT')	('the', 'DT')	('the', None)
('.', '.')	-	('.', None)

Table 4.2.1 Comparison of different operation for POS Tagging

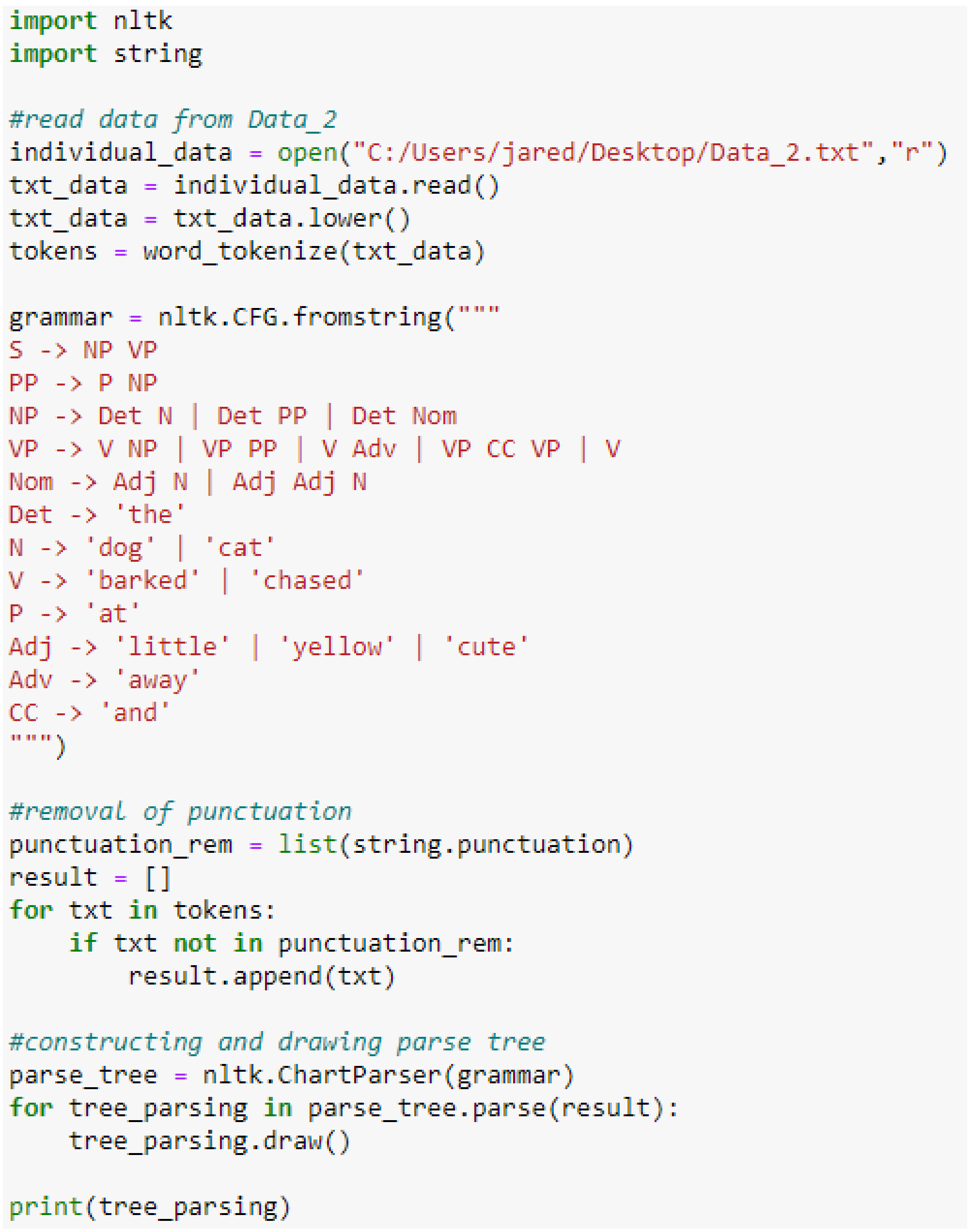
Upon completion of a series of comparisons as shown in Table 4.2.1 based on the output, it is identified that there are differences when handling POS tagging, as shown in the table above, NLTK and TextBlob performs similar when tagging normal words unless it involves in punctuation, on the other hand, POS Tagging using Regular Expression possesses the worst results as it requires the analyst to manually tag each token which results in additional efforts needed and at the same time loses accuracy. Furthermore, NLTK tagger not only provides accurate results when tagging normal words but also includes tagging punctuation while TextBlob doesn't register any punctuation and Regular Expressions requires manual tagging.

4.3 Justify the most suitable POS tagger for text analytics. Support your answer using the output obtained in task 1.

Upon completion of the comparison conducted in section 4.2 based on output on 4.1, NLTK POS Tagger tags not only normal tokens but also the punctuations, while as TextBlob ignores tagging punctuations and Regular Expression requires manual tagging for all tokens. Upon analysis of each results on output 4.1, it is identified that TextBlob is the most suitable POS Tagger for Text Analytics because the need for tagging punctuations in text analytics is unnecessary for getting insight or information. Hence, NLTK is not the most suitable POS Tagger and Regular Expression requires too much effort and is not efficient compared to the others. Overall, TextBlob serves as the best approach when it comes to POS tagging in text analytics.

4.4 Draw possible parse trees for the given sentences using suitable python codes and report the Parse Trees along with the Python code.

Python Source Code



```python
import nltk
import string

#read data from Data_2
individual_data = open("C:/Users/jared/Desktop/Data_2.txt","r")
txt_data = individual_data.read()
txt_data = txt_data.lower()
tokens = word_tokenize(txt_data)

grammar = nltk.CFG.fromstring("""
S -> NP VP
PP -> P NP
NP -> Det N | Det PP | Det Nom
VP -> V NP | VP PP | V Adv | VP CC VP | V
Nom -> Adj N | Adj Adj N
Det -> 'the'
N -> 'dog' | 'cat'
V -> 'barked' | 'chased'
P -> 'at'
Adj -> 'little' | 'yellow' | 'cute'
Adv -> 'away'
CC -> 'and'
""")

#removal of punctuation
punctuation_rem = list(string.punctuation)
result = []
for txt in tokens:
    if txt not in punctuation_rem:
        result.append(txt)

#constructing and drawing parse tree
parse_tree = nltk.ChartParser(grammar)
for tree_parsing in parse_tree.parse(result):
    tree_parsing.draw()

print(tree_parsing)
```

Figure 4.4.1 Python Source Code for Parse Tree

Output for Possible Parse Tree

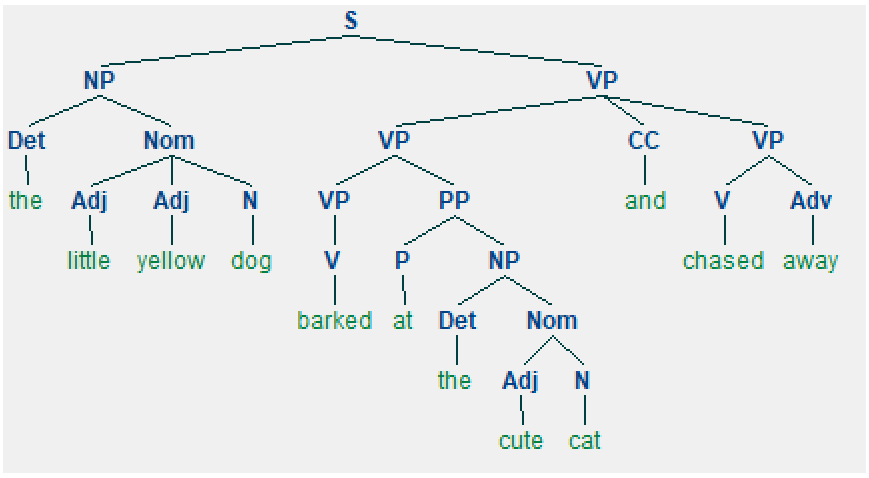


Figure 4.4.2 Output for the Possible Parse Tree

Figure 4.4.1 shows the source code for the construction of Parse Tree using the NLTK package and String package. Upon completion of reading the data from text file, the text will then be turned into lower case before tokenization. Furthermore, grammar is constructed and the NLTK Chart Parser is used for drawing the possible parse tree, the output tree is as shown in the Figure 4.4.2.

5.0 References

1. Brownlee, J., 2017. *How to Clean Text for Machine Learning with Python.* [Online]
   Available at: https://machinelearningmastery.com/clean-text-machine-learning-python/
   [Accessed 7 March 2021].

2. Heidenreich, H., 2018. *Stemming? Lemmatization? What?.* [Online]
   Available at: https://towardsdatascience.com/stemming-lemmatization-what-ba782b7c0bd8
   [Accessed 1 March 2021].

3. Jabeen, H., 2018. *Stemming and Lemmatiztion in Python.* [Online]
   Available at: https://www.datacamp.com/community/tutorials/stemming-lemmatization-python
   [Accessed 9 March 2021].

4. Kasture, N., 2019. *Regular Expressions — An excellent tool for text analysis or NLP.* [Online]
   Available at: https://medium.com/analytics-vidhya/regular-expressions-an-excellent-tool-for-text-analysis-or-nlp-d1fa7d666cb9#:~:text=Regular%20Expressions%20—%20An%20excellent%20tool%20for%20text%20analysis%20or%20NLP,-Niwratti%20Kasture&text=A%20fascinating%20program
   [Accessed 8 March 2021].

5. Liam, 2020. *What is Punctuation? Useful Punctuation Rules & Punctuation Marks in English.* [Online]
   Available at: https://7esl.com/punctuation/
   [Accessed 7 March 2021].

6. McCombe, M., 2019. *Text Processing Is Coming.* [Online]
   Available at: https://towardsdatascience.com/text-processing-is-coming-c13a0e2ee15c
   [Accessed 8 March 2021].

7. Singal, G., 2020. *Importance of Text Pre-processing.* [Online]
   Available at: https://www.pluralsight.com/guides/importance-of-text-pre-processing
   [Accessed 7 March 2021].