# CE/CZ4045 Natural Language Processing Assignment 1
# Group 7


Neo Shun Xian Nicholas
U1820539F
nneo003@e.ntu.edu.sg

Fremont Teng Qian He
U1821237C
fteng003@e.ntu.edu.sg

Li Wei-Ting
U1823750J
wli035@e.ntu.edu.sg

Laura Lit Pei Lin
U1821546A
llit001@e.ntu.edu.sg

Maskil Bin Masjuri
U1822825H
maskil001@e.ntu.edu.sg


## 2.2 Contributions of individuals in the assignment

Each part consists of the following components: research, code and writing of the report.

3.1 Domain Specific Data Analysis

- Li Wei-Ting (Tokenization and Stemming)
- Laura Lit Pei Lin (Sentence Segmentation)
- Maskil Bin Masjuri (POS Tagging)

3.2 Development of <Noun – Adjective> Pair Ranker Application

- Fremont Teng Qian He (all components of 3.2)

3.3 Application

- Neo Shun Xian Nicholas (all components of 3.3)

## 3.1 Domain Specific Dataset Analysis

### 3.1.1 Tokenization

For our first dataset, we used 20 medical reports from the field of optometry. Our second dataset uses 20 reports from the domain of petroleum. Our third dataset uses 20 reports from the domain of Natural Language Processing (NLP).

Below is the code used for tokenization. The function used is called tokenize_library().

```
def tokenize_library(txt_files):
  #Create a token library
  token_library = {}
  #Set tokenizer to only regex words
  tokenizer = RegexpTokenizer(r'\w+')
  #Loop for each file
  for i in txt_files:
    #Tokenization into words
    tokenized = tokenizer.tokenize(i.lower())
    #Loop for each token
    for j in tokenized:
      if j.isdigit():
        #If j is a number, ignore
        pass
      else:
        #If token not in library
        if j.lower() not in token_library and j.lower() not in stop_words:
          #Add 1 to dictionary
          token_library[j.lower()] = 1
        elif j.lower() not in stop_words:
          #Add 1 to dictionary
          token_library[j.lower()] += 1
  sorted_dict = dict( sorted(token_library.items(),
                             key=lambda item: item[1],
                             reverse=True))
  return sorted_dict
```

Figure 3.1.1: **Improved tokenize_library() function**

In this function, we store all the tokens into a dictionary with a counter and sort it in descending order.

In order to improve our tokenizer, we remove things such as digits, punctuations and stopwords from the tokens. As a result, we produce the following tokens below with their number of occurrences.

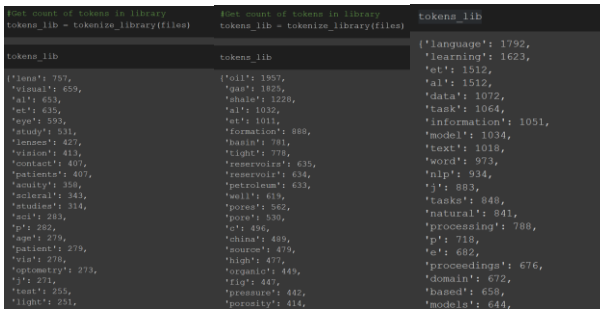


Figure 3.1.2: **Generated cleaned tokens with word count**

As we can see, the tokens generated are all related to the three domains. Tokens that should not be generated are things like p, j, al and et, which are short forms used in documentation. The unwanted tokens can be further removed by adding custom stopwords into the dictionary.

## 3.1.2 Stemming

We use Porter's Stemming for our stemmer. Below is the code to generate the stemmed dictionary from the current set of tokens.

```
#Stemming
#Initialize new dict
stemm_dict = {}
#Loop for each item in dictionary
for w in sorted_dict:
  stemmed_w = porter_stemmer.stem(w)
  #print("Actual: %s  Stem: %s"  % (w,porter_stemmer.stem(w)))
  if stemmed_w not in stemm_dict:
    stemm_dict[stemmed_w] = 1
  else:
    stemm_dict[stemmed_w] += 1

stemm_dict = dict( sorted(stemm_dict.items(),
                          key=lambda item: item[1],
                          reverse=True))
stemm_dict
```

Figure 3.1.3: **Stemmed function**

Below is the list of stemmed words from the stemmed dictionary.

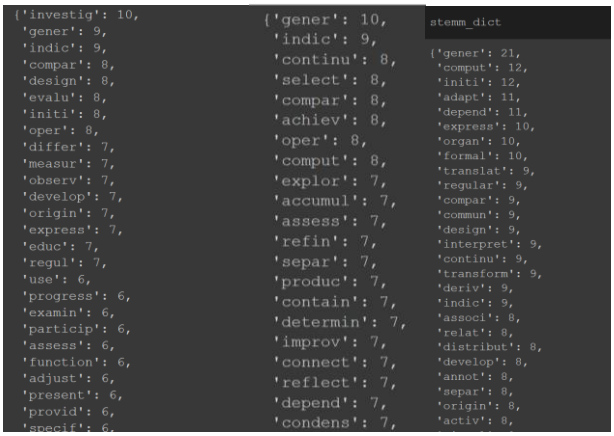


Figure 3.1.4: **Stemmed results**

When comparing before and after stemming, we can see a drastic difference in the number of unique tokens as shown in the figure below. In our case, the number of unique tokens is reduced by nearly 2000.

3.1.5.1: **Optometry data**

```
#Comparision
print("Total unique tokens before stemming: "+str(len(sorted_dict)))
print("Total unique tokens after stemming: "+str(len(stemm_dict)))

Total unique tokens before stemming: 10311
Total unique tokens after stemming: 8037
```

3.1.5.2: **Petroleum**

```
#Comparision
print("Total unique tokens before stemming: "+str(len(tokens_lib)))
print("Total unique tokens after stemming: "+str(len(stemm_dict)))

Total unique tokens before stemming: 11317
Total unique tokens after stemming: 8687
```

3.1.5.3: **NLP**

```
#Comparision
print("Total unique tokens before stemming: "+str(len(tokens_lib)))
print("Total unique tokens after stemming: "+str(len(stemm_dict)))

Total unique tokens before stemming: 15111
Total unique tokens after stemming: 11293
```

Figure 3.1.5: **Comparison of number of unique tokens before and after stemming**

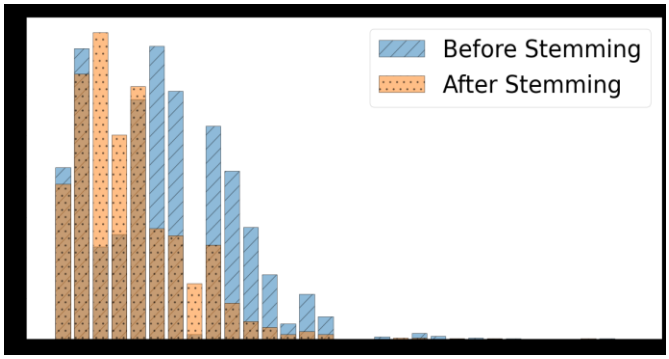
We also created a length distribution function as shown below.

```
#Function to return the length distribution of a dictionary
def getLengthDistri(target_dict):
  #Initialize counter dictionary
  counter = {}
  #Loop for each word in dictionary
  for word in target_dict:
    if len(word) not in counter:
      counter[len(word)] = 1
    else:
      counter[len(word)] += 1
  return counter
```

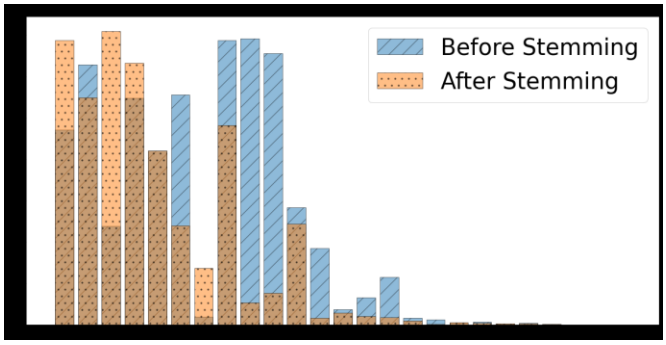
Figure 3.1.6: **getLengthDistri() function**

Below is the plot for the length distribution of tokens before (striped) and after (dotted) stemming.

3.1.7.1: **Optometry data**




3.1.7.2: **Petroleum**




3.1.7.3: **NLP**

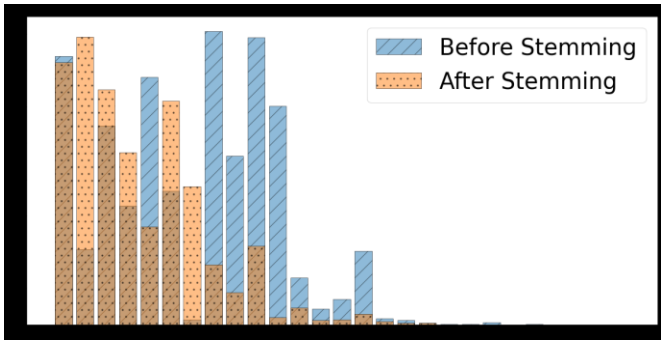



Figure 3.1.7: **Plot of length distributions of tokens before and after stemming**

### 3.1.3 Sentence Segmentation

For sentence segmentation, we define the following function below:

```
#Sentence Segmentation
def sentence_seg(txt_files):
  s_seg = []
  #Loop for each file
  for txt_file in txt_files:
    #Tokenize the file into sentences
    sentences = sent_tokenize(str(txt_file))
    #Append sentence to the full dictionary
    for sentence in sentences:
      s_seg.append(sentence)
  return s_seg
```

Figure 3.1.8: **sentence_seg() function**

The code splits the same documents into sentences. Below is the list of sentences.

```
sentences = sentence_seg(files)
sentences
```

```
['Journal of Optometry (2020) 13, 71---73\n\nwww.journalofoptometry.org\n\nEDITORIAL\n\nJourna
 'Most of them rank the scientific work visibility for individuals, journals or institutions ba
 'They have a significant effect on the public\nand peers perception of scientific merit for aut
 'Not dismissing the limitations of those metrics, which\nhave been object of previous publica
 'Other applications of bibliometrics aim\nto identify the interaction between researchers in
 'So, the IF uses a simple algorithm resulting from a quotient between citations\nreceived (nu
 'Journal of Optometry has evolved significantly in the\nnumber of manuscripts submitted for pu
 'Figure 1 shows a picture of other rankings for Journal\nof Optometry as obtained on March 22
 'Notably, J\nOptom SJR has increased steadily over the last 4 years.',
 'Out\nof the 10 indexed journals in its category, J Optom ranks\n\n3rd along with the other t
 'J Optom\nis also the only one that provides Open Access to readers,\nbeing also free for aut
 'Consequently, metrics calculated by different platforms\nhas increased steadily over the pas
 'All the estimated\nmetrics have increased consistently over the last years.',
 '(Figure 1).',
 'Clarivate IF metric is not available as J Optom is\nnot indexed in that database.',
 'Lower absolute values of SJR\nand SNIP (see figure caption for definitions) should not surpris
 'A summarized view of relevant aspects and calculation\n\n* Corresponding author.',
 'José M. González-Méijome, PhD CEORLab\n--- Center of Physics University of Minho, Portugal.',
 'E-mail address: jgmeijome@fisica.uminho.pt\n\nFigure 1 Evolution of Journal of Optometry metr
 'https://doi.org/10.1016/j.optom.2020.03.005\n1888-4296/© 2020 Spanish General Council of Opt
 'Published by Elsevier España, S.L.U.',
 'This is an open access article under the CC\nBY-NC-ND license (http://creativecommons.org/li
 '72\n\nJ.M.',
 'González-Méijome\n\nmethods of the different metrics introduced in this editorial\nis are su
 'One of those important metrics adopted by several academic institutions around the world has
 'According to Scimago ranking, the Journal of\nOptometry ended the year 2018 in the 1st quart
```

Figure 3.1.9: **List of sentences segmented**

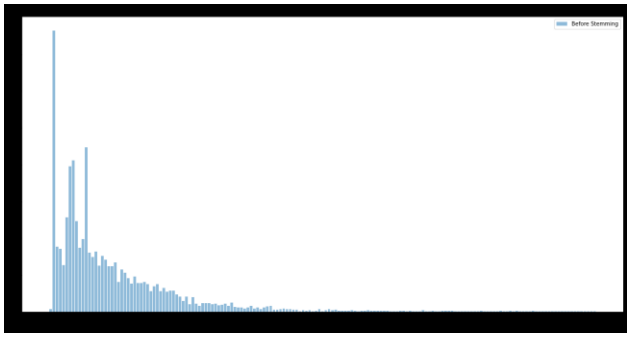
We then calculate the number of tokens in each sentence and set a counter for the number of word distribution in a sentence. This is labelled as the function getSentenceDistribution() as shown below.

```
def getSentenceDistribution(sentences):
  #Initialize counter
  counter = {}
  #Loop for each sentence
  for sentence in sentences:
    #Tokenize the sentence into words
    words = word_tokenize(sentence)
    #Save length of sentence
    if (len(words) not in counter):
      counter[len(words)] = 1
    else:
      counter[len(words)] +=1
  #Return counter as a dictionary
  sorted_dict = sorted(counter.items())
  #Set new dict
  new_dict = {}
  #Loop for each item in dict
  for i in sorted_dict:
    #Store in ascending order
    new_dict[i[0]] = i[1]
  #Return dictionary
  return new_dict
```

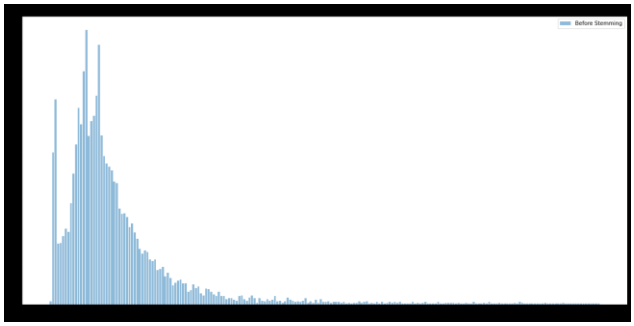
Figure 3.1.10: **getSentenceDistribution() function**

The sentence distribution is then displayed on the graph below. As we can see in the graph, sentences with less than 10 words occur most frequently.

3.1.11.1: **Optometry data**



3.1.11.2: **Petroleum**



3.1.11.3: **NLP**

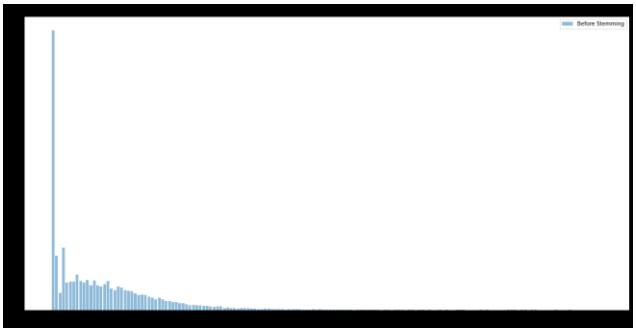


Figure 3.1.11: **Plot of sentence distribution**

## 3.1.4 POS Tagging

We randomly select three sentences as shown below in the following function randomSelectAndTag():

```
#Random select three sentences from dataset
def randomSelectAndTag(sentences):
  selectedSentences = []
  #Save sentence 1
  selectedSentences.append(sentenceTagging(random.choice(sentences)))
  #Save sentence 2
  selectedSentences.append(sentenceTagging(random.choice(sentences)))
  #Save sentence 3
  selectedSentences.append(sentenceTagging(random.choice(sentences)))

  return selectedSentences
```

Figure 3.1.12: **randomSelectAndTag() function**

We then apply POS Tagging to these three sentences as shown below in the function sentenceTagging():

```
#Tag a sentence
def sentenceTagging(sentence):
    #Initialize tagged list
    tagged_list = list()
    #Tokenize words
    wordsList = word_tokenize(sentence)
    # removing stop words from wordList
    wordsList = [w for w in wordsList if not w in stop_words]

    #  Using a Tagger. Which is part-of-speech
    # tagger or POS-tagger.
    tagged = nltk.pos_tag(wordsList)
    for item in tagged:
      tagged_list.append(item)
    return tagged_list
```

Figure 3.1.13: **sentenceTagging() function**

Below is the preview of three random sentences with their respective POS tags.

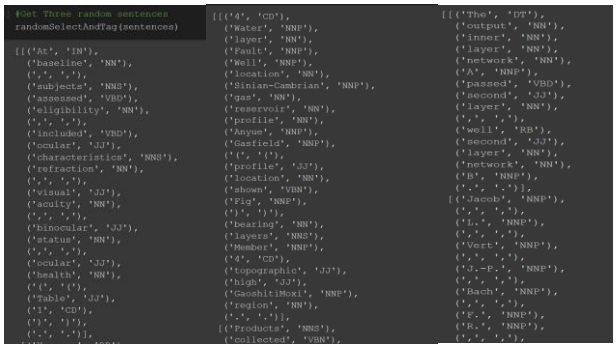


Figure 3.1.14: **POS Tagging of three randomly selected sentences**

The results are accepted and can handle well with domain specific terms. This works as well for all 3 domains as well.

## 3.2 Development of <Noun - Adjective> Pair Ranker Application

For this particular part of the assignment, we are required to develop a noun-adjective pair ranker application.

The dataset we used would be the food review of Clinton Street Baking Company & Restaurant scraped from the following yelp website: https://www.yelp.com/biz/clinton-street-baking-company-and-restaurant-singapore?osq=Waffles&start=20&sort_by=rating_desc.

This is the output of our corpus

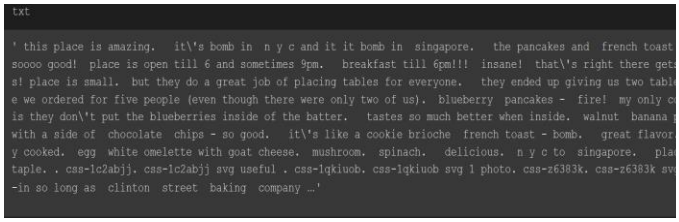


Figure 3.2.1: **Output of our corpus**

Among the dataset of 78 reviews, we handpicked 30 reviews of mixed ratings with the most meaningful noun-adjective pairs for the experiment.

Manually, we have determined our own top 5 noun-adjective pairs:

1. Best (adjective) pancakes (noun)
2. Warm (adjective) maple (noun)
3. American (adjective) style (noun)
4. French (adjective) toast (noun)
5. Fluffy (adjective) pancake (noun)

We then developed our application in Assignment 1 3-2.ipynb in python. After importing the scraped data using panda dataframe, we then utilize the regex library to clean the sentences. Using the nltk library, we then tokenize and tag each word with a POS tagging. As shown below.

Figure 3.2.2: **Tagged list**

After our data is tagged with the respective grammatical classes, we then find the occurrence of all adjective-noun pairs in the 30 reviews. We then sort their occurrences in descending order as shown in image below. The frequency of occurrences will determine the ranking of each adjective-noun pair.

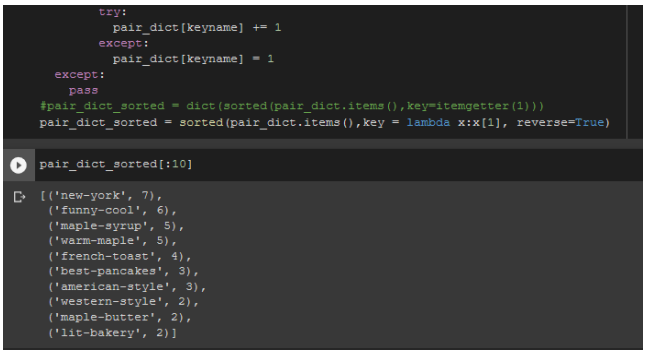


Figure 3.2.3: **Top 10 adjective-noun pair determined by the ranker**

As we can see from Figure 3.2.1, our top five predicted adjective-noun pairs can be mostly seen in the dictionary found by the application developed. The top five predicted pairs and the top five generated are not exactly the same, due to exceptional cases with the wrong POS tagging. For example ‘cool’ is tagged as a noun despite being an adjective, as determined by the nltk library.

That is an example of a challenge which we faced, where the POS is tagged wrongly to specific words. Another challenge we faced was the cleaning of data, which required thorough regex expressions to replace each sentence. Last but not least, we noticed through our manually picking of reviews in this case, bad ratings generally have lesser occurrences of adjective-nouns pairs as compared to positive ones. This can be seen from our yelp_Clinton_Street_Baking Company_&_Restaurant_scrapped.csv dataset.

## 3.3 Application on Sentimental Application

**Here are the general approach we took for section 3.3:**

1. Load corpus.txt for analysis
2. Do further data cleaning by eliminating duplicate fullstops, question marks and exclamation marks using regular expression
3. Split corpus into separate sentences and store it into a list using sent_tokenize from the nltk.tokenize library
4. Tokenize the the words in each of the sentences
5. To evaluate the sentiment of each sentences by using Textblob(x).sentiment.polarity from the textblob library

**1. Load corpus.txt for analysis**

From section 3.2, we have cleaned the dataset that we've scraped and saved it as corpus.txt. In section 3.3, we will use corpus.txt to do our analysis.

**2. Do further data cleaning by eliminating duplicate fullstops, question marks and exclamation marks using regular expression**

We will be tokenizing the sentences into individual words. Either the fullstops, question marks or exclamation marks will be used to separate the sentences. As such, we will want to eliminate duplicates of these punctuations so that we will not get a sentence which only contains a punctuation mark. To do so, we use regular expression to eliminate the duplicates which can be seen in the code snippet shown in Figure 3.3.1:

```
regex = r"([!?.])\1{0,}(?=\1)"
data_cleaned = re.sub(regex,'',data)
data_cleaned
```

Figure 3.3.1: **Use of regex**

**3. Split corpus into separate sentences and store it into a list using sent_tokenize from the nltk.tokenize library**

Using the nltk.tokenize library, we split the corpus into separate sentences using sent_tokenize. The sent_tokenize will then split the sentences by detecting either a full stop, an exclamation mark or a question mark. The individual sentence will then be stored into a list of separate sentences. A sample output is shown in Figure 3.3.2:

```
'but that is to be expected of most of the restaurants in the downtown area.',
'i went for breakfast and the food was amazing.',
'with a variety of  american-style options to choose from.',
'i shared the chocolate chip pancakes and a scrambled eggs dish.',
'and both were fantastic.',
'the pancakes were fluffy and the chocolate chips were a perfect complement.',
'it was such a sweet.',
'rich dish and  i wish  i had gotten a larger share  ( i split between 4 people though.',
```

Figure 3.3.2: **Snippets of sentence outputs using sent_tokenize**

**4. Tokenize the the words in each of the sentences**

To eliminate a short sentence with one word or less, we use the module word_tokenize from the nltk library to remove these sentences. We also noticed that some sentences of the data we scraped have the 'css' tag accompanied with it at the beginning of the sentence, hence we will also eliminate that. Figure 3.3.3 and Figure 3.3.4 shows how we eliminate both the short sentences and the 'css' tag, and the output snippets of some of the tokenized words respectively.

```
#List for sentences
sentence_library = []
for i in sentences:
  if len(word_tokenize(i)) > 3 and i[:3] != 'css':
    sentence_library.append(i)
    print(word_tokenize(i))
#sentence_library
```

Figure 3.3.3: **Removal of short sentences & 'css' tag**

```
['good', 'coffee', 'and', 'tea', '.']
['pancakes', 'are', 'huge', 'and', 'tasty', '.']
['bacon', 'is', 'crunchy', 'and', 'very', 'nice', '.']
```

Figure 3.3.4: **Snippets of the output of some tokenized words**

**5. To evaluate the sentiment of each sentences by using Textblob(x).sentiment.polarity from the textblob library**

To get the sentiment of each of the sentences, we first retrieve the sentences from the variable sentence_library as shown in Figure 3.3.3. We make use of Textblob(x).sentiment.polarity from the textblob library to determine the sentiment of the sentences. To do so, we defined a function named "senti(x)", where we passed x as the individual sentences from the list of sentences in the sentence_library variable. Then for different polarities, we classify the sentences into 3 classes: positive, neutral and negative. Figure 3.3.5 shows the user-defined function that we use to do our sentiment analysis.

```
def senti(x):
  if TextBlob(x).sentiment.polarity > 0:
    return "positive", TextBlob(x).sentiment.polarity
  elif TextBlob(x).sentiment.polarity == 0:
    return "neutral", TextBlob(x).sentiment.polarity
  else:
    return "negative", TextBlob(x).sentiment.polarity
```

Figure 3.3.5: **The senti(x) function**

After passing the individual sentences, we are able to get the sentiment as well as the polarity of each of the sentences. Based on the output, we can observe that the majority of the sentences are classified with the correctly classified

sentiment with a small number of errors. Figure 3.3.6 shows some output snippets of correctly classified sentiments of the sentences and Figure 3.3.7 shows those with incorrectly classified sentiment.

```
definitely worth a try.
('positive', 0.3)

the service was great and the atmosphere was too.
('positive', 0.8)

i ordered the blueberry pancakes and the veggie sandwich.
('neutral', 0.0)

the veggie sandwich was good.
('positive', 0.7)

it was packed with avocados which is a plus but it's a little bit pricy.
('negative', -0.1875)
```

Figure 3.3.6: **Correctly classified sentiment**

```
it was way too much for me.
('positive', 0.2)

overall felt that the food was overrated.
('neutral', 0.0)

maybe  i was expecting too much but it just did not blow me off.
('positive', 0.2)

although the place looked sparkling clean.
('positive', 0.3666666666666667)

i was surprised to see a certain small animal running through the dining area.
('positive', 0.021428571428571425)
```

Figure 3.3.7: **Incorrectly classified sentiment**

## 6. Additional Analysis

From Part 5, we analysed the sentiment for each of the sentences. In this part, we will pick some sentences that have misclassified sentiment and explain the possible reasons for the misclassification.

**Examples**

```
it's bomb in  n y c and it it bomb in  singapore.
('neutral', 0.0)
```

Figure 3.3.8: **An example of a sentence with incorrectly classified sentiment**

**Correction:** “It’s bomb” is slang used by Americans to describe something that is good which should give a positive sentiment.
**Possible reason:** As slang does not follow a proper English semantics, the parser may not know the positive sentiment and thus a neutral sentiment is given to this sentence in Figure 3.3.8.

```
i would definitely recommend getting a few things to split.
('negative', -0.1)
```

Figure 3.3.9: **Another example of a sentence with incorrectly classified sentiment**

**Correction**: This sentence should be of neutral sentiment instead of negative. The meaning of this sentence is that the writer is suggesting readers to order more food so that it can be shared among the group, which seems to have a neutral sentiment
**Possible reason:** There exists words like ‘definitely’ which has the extreme connotation and ‘split’ that may have negative connotation which thus makes this sentence have a negative sentiment.

```
maybe  i was expecting too much but it just did not blow me off.
('positive', 0.2)
```

Figure 3.3.10: **Another example of a sentence with incorrectly classified sentiment**

**Correction**: This sentence should have a negative sentiment as the writer is suggesting that the food did not meet his/her expectation.
**Possible reason:** There exists word/phrases such as “much” and “blow me off” which usually give a positive connotation. Therefore, the positive polarity may outweigh the negative polarity contributed by the “did not” in the sentence.

## REFERENCES


https://www.researchgate.net/publication/340854416_Journal_of_Optometry_bibliometrics/link/5ea0ec13a6fdcc88fc3615c7/download
https://www.researchgate.net/publication/257732670_Research_in_Optometry_A_challenge_and_a_chance
https://journals.lww.com/optvissci/pages/viewallmostpopulararticles.aspx
https://www.yelp.com/biz/clinton-street-baking-company-and-restaurant-singapore?osq=Waffles&start=20&sort_by=rating_desc
https://www.journals.elsevier.com/journal-of-petroleum-science-and-engineering/most-downloaded-articles
https://www.researchgate.net/publication/319164243_Natural_Language_Processing_State_of_The_Art_Current_Trends_and_Challenges
https://www.researchgate.net/publication/235788362_REVIEW_ON_NATURAL_LANGUAGE_PROCESSING