Yelp Review Data Analysis and Processing

CZ4045 Natural Language Processing

Group 7

Ang Jun Liang
U1722336E
JANG047@e.ntu.edu.sg

Yee Wei Min
U1720243E
C170035@e.ntu.edu.sg

Nigel Ang Wei Jun
U1721087C
NANG009@e.ntu.edu.sg

Lee Yi Yan, Esther
U1522227F
LEEY0174@e.ntu.edu.sg

Nguyen Ngoc Khanh
U1720451D
C170030@e.ntu.edu.sg

ABSTRACT

Yelp is a crowd-sourced platform for users to review businesses and establishments. Users can leave reviews and ratings which they deem the business deserves, ranging from 1 (horrible) to 5 (perfect). The main chunk of the reviews contain the body of texts accompanying the ratings, some of which follow proper grammar rules, some lack proper tense, sentence structures, or even punctuation, which can throw parsers off.

With these pretexts in mind, we will conduct an analysis of these Yelp reviews to showcase and explore the difficulties faced in processing the reviews.

Tokenizing sentences, POS tagging, and further analysis will be conducted. We also explore applications of NLP, such as development of a Noun-Adjective Pair Summarizer to summarize reviews with Noun-Adj pairs, as well as ways to predict a review's star rating based on the review text.

CCS CONCEPTS

• Artificial Intelligence → Natural Language Processing

• Applied Computing → Document Managing and Text Processing

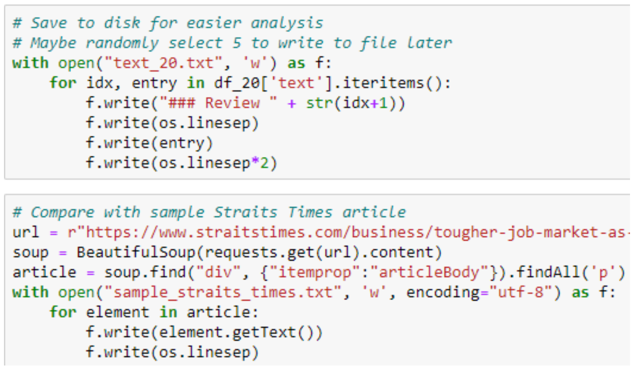
KEYWORDS

NLP, Sentence Segmentation, Tokenization, Stemming, POS Tagging, Noun-Adjective Pair

1 Data Analysis

1.1 Writing Style

To facilitate analysis, we saved the text of 20 reviews in a text file and used web scraping to obtain a random Straits Times article and saved it in another text file.



```python
# Save to disk for easier analysis
# Maybe randomly select 5 to write to file later
with open("text_20.txt", 'w') as f:
    for idx, entry in df_20['text'].iteritems():
        f.write("### Review " + str(idx+1))
        f.write(os.linesep)
        f.write(entry)
        f.write(os.linesep*2)
```

```python
# Compare with sample Straits Times article
url = r"https://www.straitstimes.com/business/tougher-job-market-as-
soup = BeautifulSoup(requests.get(url).content)
article = soup.find("div", {"itemprop":"articleBody"}).findAll('p')
with open("sample_straits_times.txt", 'w', encoding="utf-8") as f:
    for element in article:
        f.write(element.getText())
        f.write(os.linesep)
```

Figure 1.1: Code snippet to save the 20 reviews as a text file and to scrape and save Straits Times article from web.

Characteristics of Straits Times articles:

1. They follow good grammatical structure with no grammatical or spelling errors.
2. The first word in each sentence is always capitalized.
3. Proper Nouns are always capitalized.

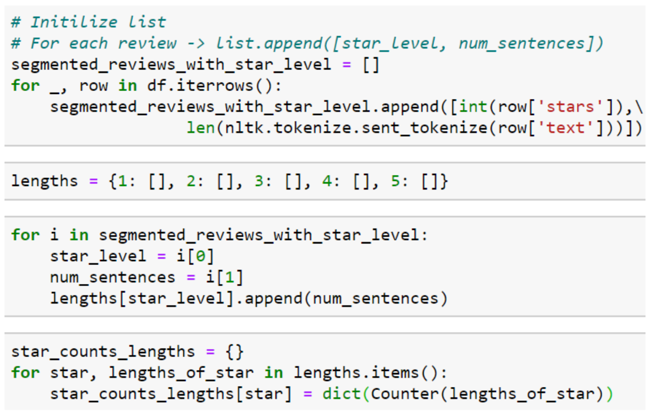
Differences with Yelp reviews:

1. Some reviews use consecutive exclamations.
   E.g. ".. without even asking me!!!!!!"
2. Some reviews use interjection (signify noises people make).
   E.g. "Ugh"
3. Some reviews capitalize words without following the grammatical rules to emphasize the word.
   E.g. ".. and she RIPPED OFF MY TOENAIL!!!!"
4. Some reviews use words not in the dictionary.
   E.g. "**garliciness** of their fries"
5. Overall, colloquial language is used.
   E.g. "Ok onto the food."

In a nutshell, Straits Times articles follow proper grammar while Yelp reviews are more informal and do not follow the rules of English exactly.

1.2 Sentence Segmentation

We first tokenized each review into sentences using the NLTK sentence tokenizer, PunktSentenceTokenizer [1]. Then, we calculated the number of sentences in each review and number of reviews of such length for each rating star.



```
# Initilize list
# For each review -> list.append([star_level, num_sentences])
segmented_reviews_with_star_level = []
for _, row in df.iterrows():
    segmented_reviews_with_star_level.append([int(row['stars']),\
                     len(nltk.tokenize.sent_tokenize(row['text']))])
```

```
lengths = {1: [], 2: [], 3: [], 4: [], 5: []}
```

```
for i in segmented_reviews_with_star_level:
    star_level = i[0]
    num_sentences = i[1]
    lengths[star_level].append(num_sentences)
```

```
star_counts_lengths = {}
for star, lengths_of_star in lengths.items():
    star_counts_lengths[star] = dict(Counter(lengths_of_star))
```

Figure 1.2.1: Code for sentence segmentation and to calculate the number of sentences in each review and the number of reviews of such length for each rating star.

For each rating star, we drew a scatter plot of number of reviews against the length of a review in terms of number of sentences.

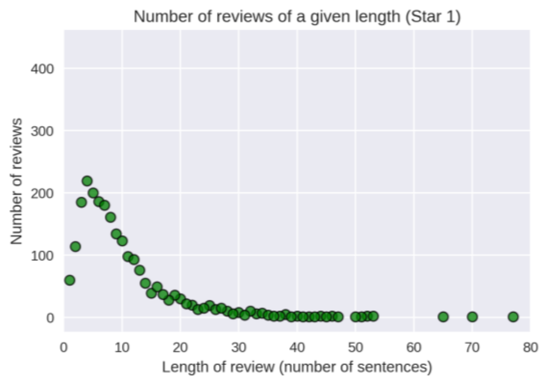


Figure 1.2.2: Distribution of the number of reviews with various lengths in number of sentences before for star 1.

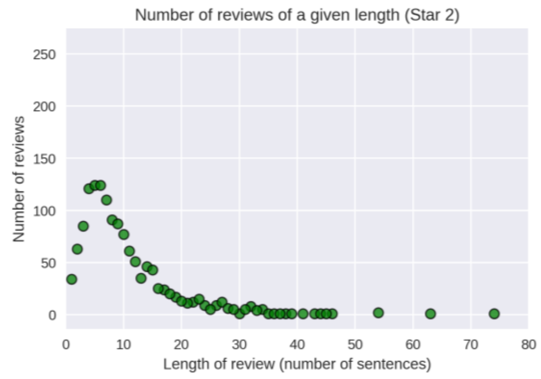


Figure 1.2.3: Distribution of the number of reviews with various lengths in number of sentences before for star 2.

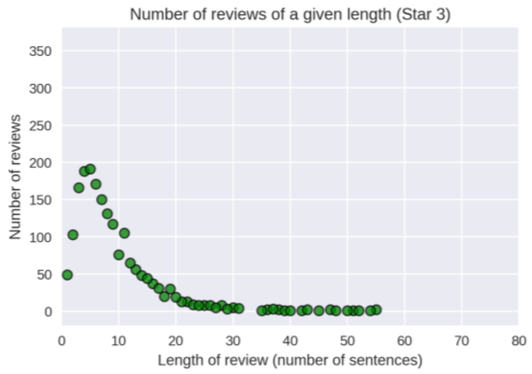


Figure 1.2.4: Distribution of the number of reviews with various lengths in number of sentences before for star 3.

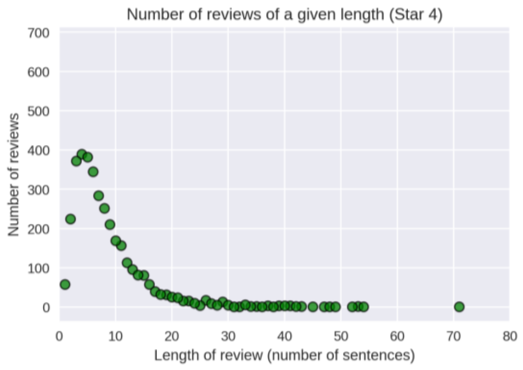


Figure 1.2.5: Distribution of the number of reviews with various lengths in number of sentences before for star 4.

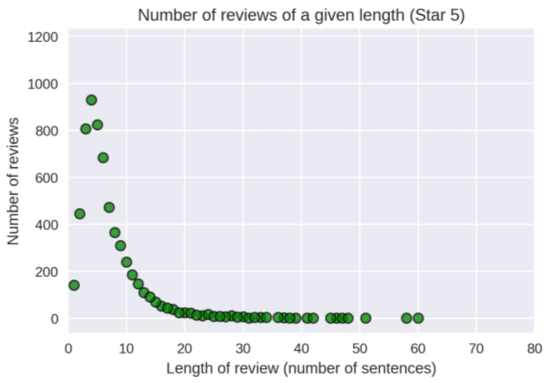
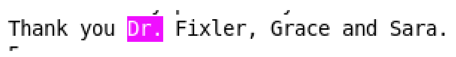


Figure 1.2.6: Distribution of the number of reviews with various lengths in number of sentences before for star 5.

Throughout 1-star to 4-star, the graphs reflected nearly the same distribution of review length. However, the 5-star review length distribution is different from the rest because there is a significant portion of reviews where the length of reviews less than 10 sentences. We hypothesize that unsatisfied customers gave longer feedback to clarify or explain their rating.

Lastly, we sorted the reviews in length-order and chose reviews at the 10th, 20th, 30th, 70th and 90th percentile as our sample reviews. In these 5 sample reviews, the sentence boundary is detected correctly. The words and abbreviations with internal punctuation are successfully not detected as sentence boundary.



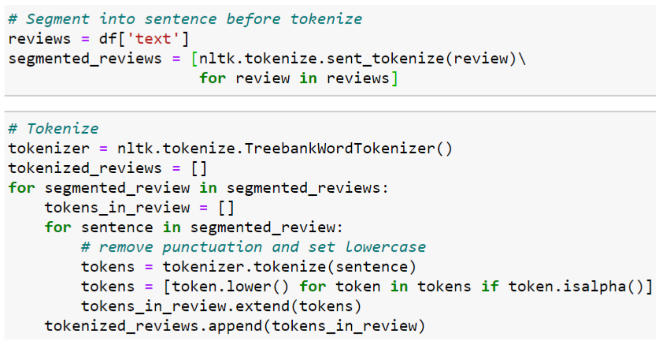
```
Thank you Dr. Fixler, Grace and Sara.
```

Figure 1.2.7: The abbreviation Dr. is not detected as sentence boundary.

1.3 Tokenization and Stemming

1.3.1 Tokenization and stemming of reviews

Each review is first segmented into sentences using the NLTK's sentence tokenizer, PunktSentenceTokenizer. Each sentence is then tokenized into words using TreebankWordTokenizer. Punctuation tokens are removed from the list of tokens.

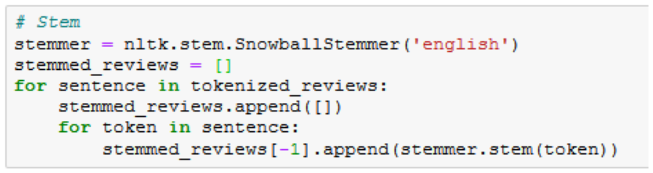


```
# Segment into sentence before tokenize
reviews = df['text']
segmented_reviews = [nltk.tokenize.sent_tokenize(review)\
                     for review in reviews]
```

```
# Tokenize
tokenizer = nltk.tokenize.TreebankWordTokenizer()
tokenized_reviews = []
for segmented_review in segmented_reviews:
    tokens_in_review = []
    for sentence in segmented_review:
        # remove punctuation and set lowercase
        tokens = tokenizer.tokenize(sentence)
        tokens = [token.lower() for token in tokens if token.isalpha()]
        tokens_in_review.extend(tokens)
    tokenized_reviews.append(tokens_in_review)
```

Figure 1.3.1.1: Code snippets for sentence tokenization and word tokenization.

For stemming, we use the Snowball stemming algorithm in NLTK to obtain the root of the word.



```
# Stem
stemmer = nltk.stem.SnowballStemmer('english')
stemmed_reviews = []
for sentence in tokenized_reviews:
    stemmed_reviews.append([])
    for token in sentence:
        stemmed_reviews[-1].append(stemmer.stem(token))
```

Figure 1.3.1.2: Code snippet for stemming

Then, we calculated the number of tokens in each review and number of reviews of such length before and after stemming. The results are used to plot the scatter plots of number of reviews against the length of a review in number of tokens before and after stemming.

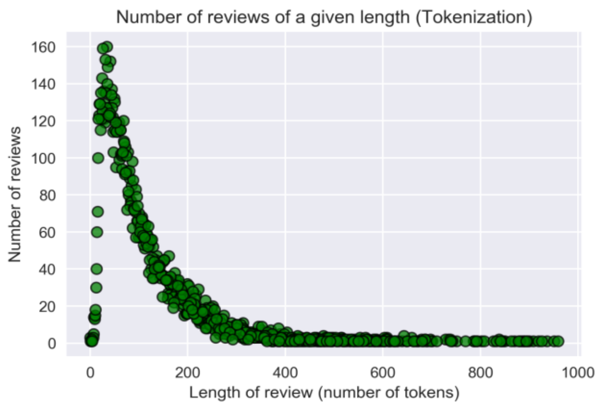


Figure 1.3.1.3: Distribution of the number of reviews with various lengths (in number of tokens) before stemming

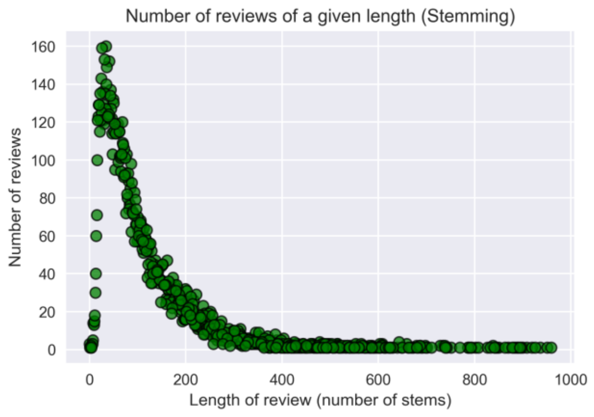
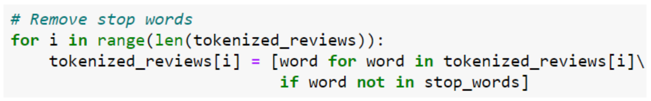


Figure 1.3.1.4: Distribution of the number of reviews with various lengths (in number of tokens) after stemming

As seen from the plots, the number of tokens before and after stemming is the same. This is because what stemming does is to only strip a word or token of its suffixes, hence no words will be removed in the process.

1.3.2 Top 20 Most Frequent Words

Before retrieving the top 20 most frequent words from the review, we must first filter stop words from our data. We make use of the list of English stop words provided by NLTK. These stop words are listed in the appendix. After that, the top 20 most frequent words before and after stemming are obtained.



```
# Remove stop words
for i in range(len(tokenized_reviews)):
    tokenized_reviews[i] = [word for word in tokenized_reviews[i]\
                            if word not in stop_words]
```

Figure 1.3.2.1: Code to remove stop words

```
# Find top 20 tokens for tokenized_reviews
word_dict = {}
for tokenized_review in tokenized_reviews:
    for word in tokenized_review:
        if word not in word_dict:
            word_dict[word] = 1
        else:
            word_dict[word] += 1
top_20 = heapq.nlargest(20, word_dict.items(), key=lambda item: item[1])
# Save to disk
with open ("top_20_tokens.txt", 'w') as f:
    for word in top_20:
        f.write(f"{word[0]}: {word[1]}" + '\n')
# To compare to top 20 stems
with open ("top_20_tokens (stemmed).txt", 'w') as f:
    for word in top_20:
        f.write(f"{stemmer.stem(word[0])}: {word[1]}" + '\n')
```

Figure 1.3.2.2: Code to obtain top 20 most frequent words from tokenized reviews.

```
# Remove stop words
for i in range(len(stemmed_reviews)):
    stemmed_reviews[i] = [word for word in stemmed_reviews[i]\
                           if word not in stop_words]
```

```
# Find top 20 stems for stemmed_reviews
word_dict = {}
for stemmed_review in stemmed_reviews:
    for word in stemmed_review:
        if word not in word_dict:
            word_dict[word] = 1
        else:
            word_dict[word] += 1
top_20 = heapq.nlargest(20, word_dict.items(), key=lambda item: item[1])
print(top_20)
# Save to disk
with open ("top_20_stems.txt", 'w') as f:
    for word in top_20:
        f.write(f"{word[0]}: {word[1]}" + '\n')
```

Figure 1.3.2.3: Code to obtain top 20 most frequent words from stemmed reviews.

	Before stemming		After stemming	
1	food	(8580)	place	(9407)
2	place	(8236)	food	(8731)
3	good	(7919)	good	(8055)
4	great	(6295)	time	(6588)
5	service	(6027)	get	(6390)
6	like	(5534)	servic	(6338)
7	get	(5216)	great	(6326)
8	time	(5172)	like	(6231)
9	would	(5168)	order	(6152)
10	one	(5044)	go	(5993)
11	back	(4717)	one	(5278)
12	go	(4145)	would	(5168)
13	really	(3731)	back	(4744)
14	also	(3333)	friend	(3982)
15	got	(3171)	tri	(3977)
16	us	(2926)	come	(3784)
17	even	(2891)	realli	(3731)
18	order	(2820)	also	(3333)
19	could	(2815)	love	(3297)
20	nice	(2756)	got	(3171)

Table 1.3.2: List of the top 20 most frequent words before and after performing stemming

Some of the top 20 frequent words after stemming like '*servic*' and '*realli*' are not English words and this is because of stemming. Some of the words that are expected to be popular but are not in the list are: *restaurant, delicious, dinner, lunch, taste, fast, slow, long, experience, serve, ambience, atmosphere, cook, warm*. Some of the words in the list that are not expected to be popular and do not carry much meaning: would, back, also, us, even, could. Some words appear in the list because they are quite generic: *get, one, go, got, tri*.

1.4 POS Tagging

5 reviews were randomly selected, segmented and then tokenized into sentences, from which 5 sentences were randomly picked. For each sentence, it is then further tokenized into individual words, and POS tagging applied using the Penn Treebank POS Tag sets.

```
# POS Tagging (refer upenn_tagset.txt for tagset)
# Tokenize first, then run POS Tagger
five_POS_tagged_sentences = []
for sentence in five_sentences:
    tokens = tokenizer.tokenize(sentence)
    # remove punctuation and set lower case
    tokens = [token.lower() for token in tokens if token.isalpha()]
    five_POS_tagged_sentences.append(nltk.pos_tag(tokens))
# Save POS Tagging results (each sentence is separated by a line)
with open("POS_tagging_result.txt", 'w') as f:
    for POS_tagged_sentence in five_POS_tagged_sentences:
        for word_and_POS_tag in POS_tagged_sentence:
            f.write(f"{word_and_POS_tag[0]} : {word_and_POS_tag[1]}" + '\n')
        f.write('\n')
```

Figure 1.4: The code snippet for the POS Tagging

1.4.1 Five Tagged Sentences

1. still (RB) a (DT) very (RB) tasty (JJ) sandwich (NN) though (IN)
2. your (PRP$) wallet (NN) will (MD) thank (VB) me (PRP) later (RB)
3. seriously (RB) everything (NN)
4. overall (JJ) i (NN) will (MD) try (VB) again (RB) but (CC) next (JJ) time (NN) with (IN) no (DT) garlic (JJ) bread (NN) and (CC) thin (JJ) crust (NN) pizza (NN)
5. the (DT) store (NN) is (VBZ) chic (JJ) and (CC) clean (JJ)

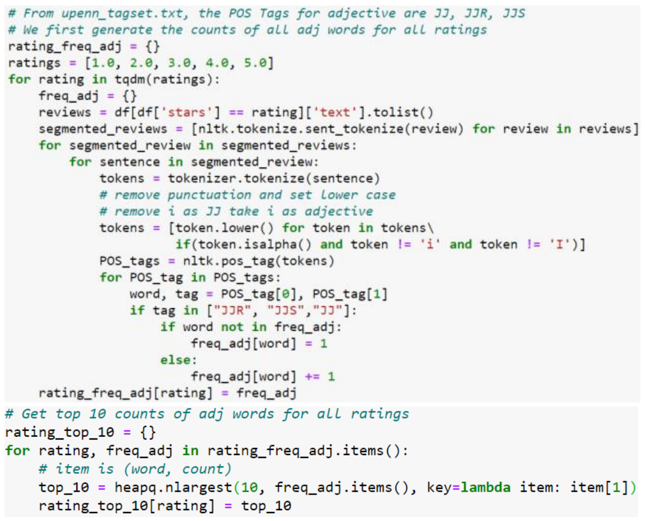
Results are fairly accurate with an overall accuracy of 35/36 = 97.22%. However, the word 'overall' in sentence 4 is incorrectly tagged as an adjective (JJ). In this context, it should be tagged as an adverb (RB) instead.

POS taggers may face challenges in a more informal setting where proper grammar is not enforced; lack of/erroneous punctuation, short-forms, improper form/tenses and spelling can cause POS taggers to tag a word wrongly.

1.5 Most Frequent Adjectives for each Rating

In this section, we will leverage the use of sentence segmentation, tokenization and POS tagging techniques explained in the section above to generate a list of the top-10 most frequently used adjectives for each rating star and a list of the top-10 most indicative adjectives for each rating star.

1.5.1 Top-10 most frequently used adjectives for each rating



```python
# From upenn_tagset.txt, the POS Tags for adjective are JJ, JJR, JJS
# We first generate the counts of all adj words for all ratings
rating_freq_adj = {}
ratings = [1.0, 2.0, 3.0, 4.0, 5.0]
for rating in tqdm(ratings):
    freq_adj = {}
    reviews = df[df['stars'] == rating]['text'].tolist()
    segmented_reviews = [nltk.tokenize.sent_tokenize(review) for review in reviews]
    for segmented_review in segmented_reviews:
        for sentence in segmented_review:
            tokens = tokenizer.tokenize(sentence)
            # remove punctuation and set lower case
            # remove i as JJ take i as adjective
            tokens = [token.lower() for token in tokens\
                      if(token.isalpha() and token != 'i' and token != 'I')]
            POS_tags = nltk.pos_tag(tokens)
            for POS_tag in POS_tags:
                word, tag = POS_tag[0], POS_tag[1]
                if tag in ["JJR", "JJS","JJ"]:
                    if word not in freq_adj:
                        freq_adj[word] = 1
                    else:
                        freq_adj[word] += 1
    rating_freq_adj[rating] = freq_adj
# Get top 10 counts of adj words for all ratings
rating_top_10 = {}
for rating, freq_adj in rating_freq_adj.items():
    # item is (word, count)
    top_10 = heapq.nlargest(10, freq_adj.items(), key=lambda item: item[1])
    rating_top_10[rating] = top_10
```

Figure 1.5.1: Code snippet to generate the top-10 most frequently used adjectives for each rating star.

Firstly, we need to generate the counts of all adjectives for each rating star. To achieve this, we apply sentence segmentation, tokenization and POS tagging to every review of each rating. From the code snippet, we can see that the punctuation tokens are removed, and all the tokens are set to lowercase after tokenization to prepare for POS tagging. Besides, ‘I’ and ‘i’ words are also removed because POS tagging function that we used considers them as adjectives. After POS tagging, we look for the token with the tag in ‘JJ’, ‘JJR’ and ‘JJS’ to count the occurences of all adjectives for each rating. We do this because from the upenn_tagset.txt that is used by the nltk.pos_tag() function, we know that the POS Tags for adjectives are ‘JJ’, ‘JJR’ and ‘JJS’.

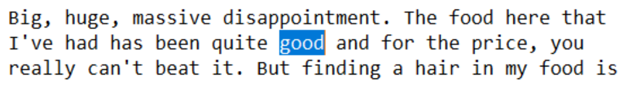
After generating the counts of all adjectives for all ratings, we use heapq.nlargest() function to generate the top 10 most frequent adjective for all ratings.

Rating	**Top-10 most frequent adjectives with counts**			
1	good	(668)	new	(344)
	other	(547)	more	(315)
	bad	(470)	last	(314)
	worst	(374)	horrible	(296)
	first	(345)	great	(262)
2	good	(789)	nice	((217)
	other	(396)	better	(213)
	great	(309)	more	(202)
	bad	(247)	much	(201)
	first	(220)	little	(198)
3	good	(1553)	small	(294)
	other	(570)	more	(291)
	great	(542)	bad	(270)
	nice	(462)	few	(250)
	little	(388)	better	(246)
4	good	(2731)	delicious	(660)
	great	(1873)	best	(503)
	nice	(845)	friendly	(494)
	other	(782)	fresh	(483)
	little	(701)	more	(443)
5	great	(3309)	nice	(886)
	good	(2113)	other	(802)
	best	(1519)	fresh	(678)
	friendly	(1038)	new	(672)
	delicious	(987)	first	(630)

Table 1.5.1: Top-10 most frequent adjectives of each rate

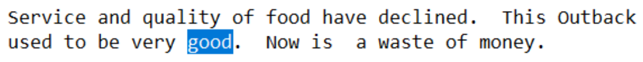
As rating 1 is the lowest rating, we can see some adjectives that show customer’s dissatisfactions like ‘bad’, ‘worst’ and ’horrible’ from Table 1.1. However, we also realized that there is a considerable large amount of ‘good’ appearing in rate 1. Hence, we decided to look into the reviews with rate 1 that contain ‘good’ and 3 examples are selected to be discussed in this report.

Example 1:



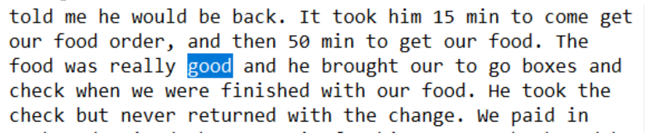
Big, huge, massive disappointment. The food here that I've had has been quite good and for the price, you really can't beat it. But finding a hair in my food is

Example 2:



Service and quality of food have declined. This Outback used to be very good. Now is a waste of money.

Example 3:



told me he would be back. It took him 15 min to come get our food order, and then 50 min to get our food. The food was really good and he brought our to go boxes and check when we were finished with our food. He took the check but never returned with the change. We paid in

From the 3 examples above, we can see that the adjective ‘good’ is used very often in reviews with rating 1 too. This is because the customers often describe more than one perspective and some of these perspectives may not be the reason why they give bad ratings to the business. For example, the reviewer of example 2 described that the food used to be good but is now bad and the reviewer of example 3 thinks that the service is horrible, but the food is good. These 2 examples show that the reviewers may use the word ‘good’ to describe other services of the business or past experiences instead of the reason for giving bad ratings.

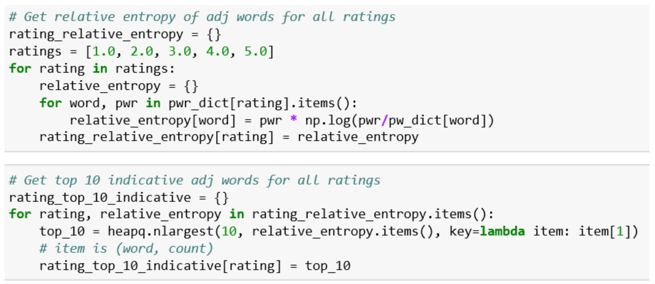
As the rating increases, we can realize that the count of adjectives that describe dissatisfaction like ‘bad’ drops and the count of adjectives that describe satisfaction like ‘great’ increases. In Table 1.1, adjectives like ‘great’, ‘good’ and ‘best’ have high frequency and is reasonable and expected, because giving 5 stars shows that the customer is satisfied.

To conclude, although there are some adjectives that are in line with the rating but some adjectives like *'more'* and *'other'* are less meaningful and has high frequency across different rating. This shows that getting the most frequent adjectives may not be the best way to analyse the reviews of different rating as it will be affected by those commonly used adjectives. Hence, we proceed to find the most indicative adjectives for each rating in the next section.

1.5.2 Top-10 most indicative adjectives for each rating star

In this section, we find the most indicative adjectives for each rating by using the pointwise relative entropy. Let P(w|R1) be the probability of observing word w in all reviews with rating star 1, and let P(w) be the probability of observing word w in all reviews, then relative entropy for word w can be computed as: $P(w|R1)\ log(\frac{P(w|R1)}{P(w)})$. From the formula, we can see that the relative entropy for the word w is larger when P(w|R1) is greater than P(w) and can only be zero if P(w|R1) equals to P(w). Hence, we can conclude that the relative entropy measures how much larger is P(w|R1) compare to P(w). Through relative entropy, we can filter out the adjectives that appear constantly and frequently in all rating star as they have similar P(w|R1) and P(w) and their relative entropy will be low.

To generate the top-10 most indicative adjectives for each rating, P(w|R1), P(w|R2), P(w|R3), P(w|R4), P(w|R5) and P(w) of each adjective are calculated and are placed into the formula to calculate the pointwise relative entropy. Lastly, heapq.nlargest() function is used to get the top-10 most indicative adjective.



```
# Get relative entropy of adj words for all ratings
rating_relative_entropy = {}
ratings = [1.0, 2.0, 3.0, 4.0, 5.0]
for rating in ratings:
    relative_entropy = {}
    for word, pwr in pwr_dict[rating].items():
        relative_entropy[word] = pwr * np.log(pwr/pw_dict[word])
    rating_relative_entropy[rating] = relative_entropy
```

```
# Get top 10 indicative adj words for all ratings
rating_top_10_indicative = {}
for rating, relative_entropy in rating_relative_entropy.items():
    top_10 = heapq.nlargest(10, relative_entropy.items(), key=lambda item: item[1])
    # item is (word, count)
    rating_top_10_indicative[rating] = top_10
```

Figure 1.5.2: Code to generate the top-10 most indicative adjectives for each rating star.

Rating	**Top-10 most indicative adjectives**	
1	1. worst 2. horrible 3. terrible 4. rude 5. bad	6. poor 7. awful 8. worse 9. last 10. disappointed
2	1. bad 2. dry 3. ok 4. better 5. disappointed	6. same 7. okay 8. chinese 9. disappointing 10. second
3	1. good 2. decent 3. ok 4. other 5. small	6. overall 7. little 8. okay 9. average 10. nice
4	1. good 2. great 3. nice 4. delicious 5. little	6. tasty 7. fresh 8. small 9. huge 10. hot
5	1. great 2. best 3. friendly 4. delicious 5. amazing	6. awesome 7. happy 8. excellent 9. super 10. wonderful

Table 1.5.2: Top-10 most indicative adjectives of each rate

From Table 1.5.2, we can observe that these adjectives are in line with the rating. For example, we can see words that describe dissatisfaction in rating star 1 like 'disappointed' and we can see words that describe satisfaction in rating star 5 like 'amazing'. This is because these adjectives only have high probability of occurrence in the specific rating star but do not have a high probability of occurrence in the overall review.

Comparing Table 1.5.1 and Table 1.5.2, we can see that the top-10 most indicative adjectives of each rate are more meaningful compared to the top-10 most frequent adjectives of each rate because adjectives that are frequently used but less meaningful like *'more'* and *'other'* do not appear in the list of top-10 most indicative adjectives.

2 Development of a Noun-Adjective Pair Summarizer

2.1 Extracting Noun-Adjective Pairs

In this section, we discuss how we extracted Noun-Adj pairs from a review and document each improvement we made. Essentially, we would perform sentence segmentation, then we extract the Noun-Adj pairs.

2.1.1 Naive Approach using only POS Tags

Our initial simple approach is to only use POS Tags. For each Noun token, we find the closest Adj Token, to form a Noun-Adj pair.

```python
def get_noun_adjective_pairs(sentence):
    nlp = spacy.load("en_core_web_sm")
    doc = nlp(sentence)
    noun_adj_pairs = []
    for i, token in enumerate(doc):
        if token.tag_ in ['NN']:
            j = 1
            direction = 1
            while True:
                if i+j*direction < 0  or i+j*direction >= len(doc):
                    break
                if doc[i+j*direction].tag_ in ['JJ']:
                    noun_adj_pairs.append((token.text.lower(),\
                              doc[i+j*direction].text.lower()))
                    break
                direction *= -1
                if direction == 1:
                    j += 1
    return noun_adj_pairs
```

Figure 2.1.1: Code to extract Noun-Adj pairs using POS Tags

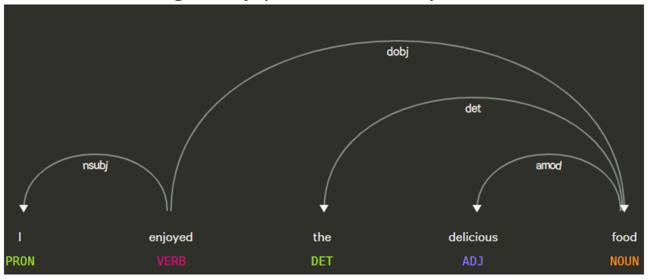
Continuing from here, we will explore 2 additional methods for improvements.

2.1.2 Improved Approach using Dependency Structures

The initial approach is a naive one and fails even for slightly complex sentences. Hence, we came up with an additional method, utilizing the dependency structure of the sentence to find Noun-Adj pairs. The power of dependency structures lies in the fact that we can find the relationship between two words in a sentence:

First part - For each token, if it is a Noun and there exists an Adj token amongst its children tokens, then the token and that child Adj token becomes a Noun-Adj pair.

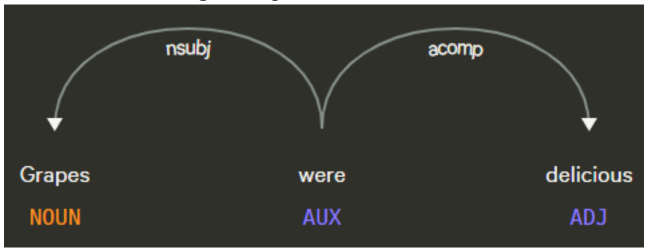
e.g. "I enjoyed the delicious food"



In this example, 'Delicious' (Adj) is child token of 'food' (Noun), so food-delicious is Noun-Adjective pair.

Second part - For each token, if there exists an Adj token and another Noun token amongst its children tokens, then the child Adj token and child Noun token is a Noun-Adj pair.

e.g. "Grapes were delicious"



In this example, 'Grapes' (Noun) and 'delicious' (Adj) are both child tokens of 'were', so 'grapes-delicious' is a Noun-Adj pair).

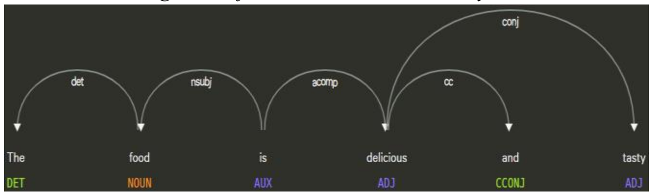
```python
for token in doc:
    '''
    Intuition: Look for noun tokens and adjectives in local subtrees
    '''
    children = list(token.children)
    if (token.tag_ in ['NN', 'NNS', 'NNP', 'NNPS']):
        for child in children:
            if (child.tag_ in ['JJ', 'JJR', 'JJS']):
                noun = [token.i, token.text.lower()]
                adj = [child.i, child.text.lower()]
                noun_adj_pairs.append([noun, adj])
    for i in range(len(children)-1):
        for j in range(i+1, len(children)):
            if children[i].tag_ in ['NN', 'NNS', 'NNP', 'NNPS'] and\
            children[j].tag_ in ['JJ', 'JJR', 'JJS']:
                noun = [children[i].i, children[i].text.lower()]
                adj = [children[j].i, children[j].text.lower()]
                noun_adj_pairs.append([noun, adj])
            if children[i].tag_ in ['JJ', 'JJR', 'JJS'] and\
            children[j].tag_ in ['NN', 'NNS', 'NNP', 'NNPS']:
                noun = [children[j].i, children[j].text.lower()]
                adj = [children[i].i, children[i].text.lower()]
                noun_adj_pairs.append([noun, adj])
```

Figure 2.1.2: Code to extract Noun-Adj pairs using only dependency structure

2.1.3 Dealing with conjunctions

Continuing, we experimented and came up with a heuristic to deal with conjunctions; After running the procedure in *Section 2.1.2*, if there still exists some Adj token that is not part of any Noun-Adj pair, we check if its parent token is an Adj token. If so, the noun token paired to its parent Adj token also forms a Noun-Adj pair with itself.

e.g. "The food is delicious and tasty"



Via the first method depicted in *Section 2.1.2*, we obtain 'food-delicious' as a Noun-Adj pair. However, this procedure will not be able to find the other Noun-Adj pair: 'food-tasty'. Via the heuristic we introduce, the parent token of 'tasty' is 'delicious' which is an Adj token, meaning that the 'food' token paired to 'delicious' also pairs with 'tasty' to form 'food-tasty' as a Noun-Adj pair.

```python
list_of_adj = [noun_adj_pair[1][0] for noun_adj_pair in noun_adj_pairs] # in terms of index
list_of_noun = [noun_adj_pair[0][0] for noun_adj_pair in noun_adj_pairs] # in terms of index
flag = True
while flag:
    flag = False
    for token in doc:
        if token.tag_ in ['JJ', 'JJR', 'JJS']\
        and token.i not in list_of_adj\
        and token.head.i in list_of_adj:
            head_adj_position = list_of_adj.index(token.head.i)
            noun_position = list_of_noun[head_adj_position]
            list_of_adj.append(token.i)
            list_of_noun.append(noun_position)
            noun_adj_pairs.append([[noun_position, doc[noun_position].text.lower()],\
                                   [token.i, token.text.lower()]])
            flag = True
```

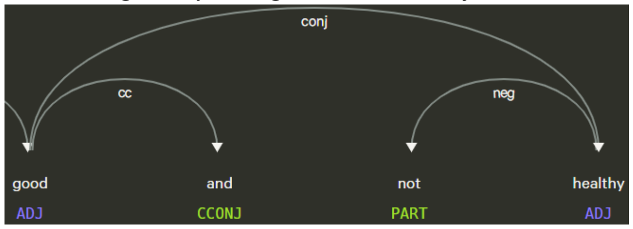
Figure 2.1.3: Code to deal with conjunctions
Note: noun_adj_pairs is computed in Figure 2.1.2

2.1.4 Dealing with 'not'

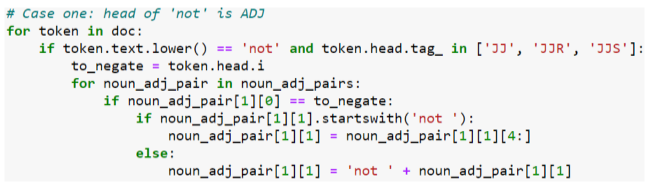
So far, 'food-good' will be extracted from "The food is not good" which is wrong. 'food-not good' should be extracted instead. In this section, we tackle this. The idea is to find exactly which Noun-Adj pair we need to negate; There are 2 main cases:

Case 1 - if the parent token of 'not' token is an Adj token, we go to the list of Noun-Adj pairs we have computed thus far, find the Noun-Adj pair that contains that parent Adj token, and modify the Noun-Adj pair to Noun-not Adj pair.

E.g. "The food is good and not healthy"



In this example, the parent token of 'not' token is 'healthy' which is an Adj token. The Noun-Adj pair that contains 'healthy' is 'food-healthy' (can be obtained through the heuristic explained in Section 2.1.3). Hence, the Noun-Adj pair 'food-healthy' will be changed to 'food-not healthy'. In addition, we also deal with multiple 'not's by checking if the Adj begins with 'not'. If so, remove it; Else, add 'not' at the front.

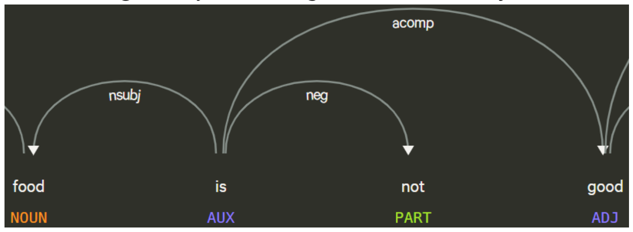


```
# Case one: head of 'not' is ADJ
for token in doc:
    if token.text.lower() == 'not' and token.head.tag_ in ['JJ', 'JJR', 'JJS']:
        to_negate = token.head.i
        for noun_adj_pair in noun_adj_pairs:
            if noun_adj_pair[1][0] == to_negate:
                if noun_adj_pair[1][1].startswith('not '):
                    noun_adj_pair[1][1] = noun_adj_pair[1][1][4:]
                else:
                    noun_adj_pair[1][1] = 'not ' + noun_adj_pair[1][1]
```

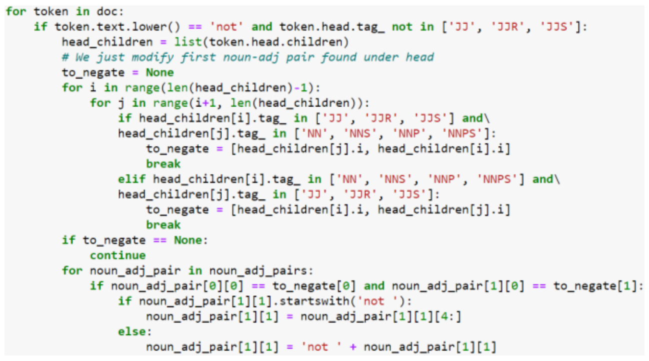
Figure 2.1.4: Code to deal with 'not' if the parent token of 'not' token is an Adj token.

Case 2 - the parent token of 'not' token is not an Adj token. In this case, we check if there exists some Noun-Adj pair formed between the children of its parent token (Second way in Section 2.1.2). If so, we negate that Noun-Adj pair.

E.g. "The food is not good and not healthy"



In this example, the parent token of 'not' token is 'is' which is not an Adj token. However, there exists a Noun-Adj pair 'food-good' between its children. Hence, 'food-good' will be modified to 'food-not good'.

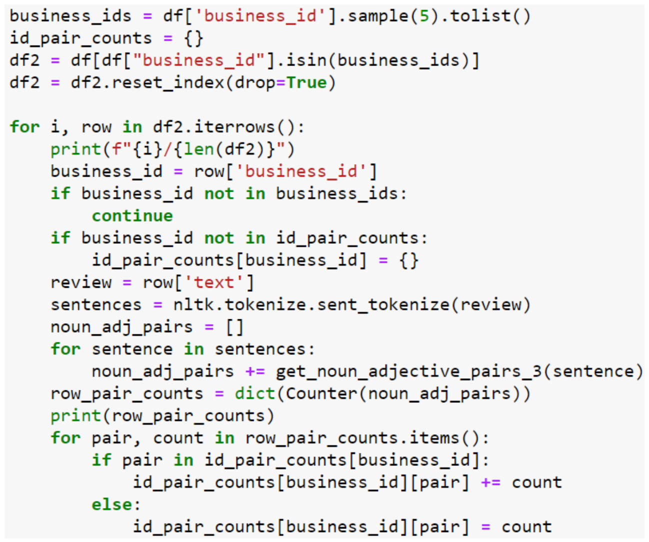


```
for token in doc:
    if token.text.lower() == 'not' and token.head.tag_ not in ['JJ', 'JJR', 'JJS']:
        head_children = list(token.head.children)
        # We just modify first noun-adj pair found under head
        to_negate = None
        for i in range(len(head_children)-1):
            for j in range(i+1, len(head_children)):
                if head_children[i].tag_ in ['JJ', 'JJR', 'JJS'] and\
                head_children[j].tag_ in ['NN', 'NNS', 'NNP', 'NNPS']:
                    to_negate = [head_children[j].i, head_children[i].i]
                    break
                elif head_children[i].tag_ in ['NN', 'NNS', 'NNP', 'NNPS'] and\
                head_children[j].tag_ in ['JJ', 'JJR', 'JJS']:
                    to_negate = [head_children[i].i, head_children[j].i]
                    break
        if to_negate == None:
            continue
        for noun_adj_pair in noun_adj_pairs:
            if noun_adj_pair[0][0] == to_negate[0] and noun_adj_pair[1][0] == to_negate[1]:
                if noun_adj_pair[1][1].startswith('not '):
                    noun_adj_pair[1][1] = noun_adj_pair[1][1][4:]
                else:
                    noun_adj_pair[1][1] = 'not ' + noun_adj_pair[1][1]
```

Figure 2.1.5: Code to deal with 'not' if the parent token of 'not' token is not an Adj token.

2.2 Summarize Businesses by Frequent Noun-Adjective Pairs

We randomly sample 5 business ids and obtain all the reviews corresponding to each of these 5 business ids. We then segmented these reviews into sentences before extracting Noun-Adj pairs. We noted counts of each Noun-Adj pair for each business id.



```
business_ids = df['business_id'].sample(5).tolist()
id_pair_counts = {}
df2 = df[df["business_id"].isin(business_ids)]
df2 = df2.reset_index(drop=True)

for i, row in df2.iterrows():
    print(f"{i}/{len(df2)}")
    business_id = row['business_id']
    if business_id not in business_ids:
        continue
    if business_id not in id_pair_counts:
        id_pair_counts[business_id] = {}
    review = row['text']
    sentences = nltk.tokenize.sent_tokenize(review)
    noun_adj_pairs = []
    for sentence in sentences:
        noun_adj_pairs += get_noun_adjective_pairs_3(sentence)
    row_pair_counts = dict(Counter(noun_adj_pairs))
    print(row_pair_counts)
    for pair, count in row_pair_counts.items():
        if pair in id_pair_counts[business_id]:
            id_pair_counts[business_id][pair] += count
        else:
            id_pair_counts[business_id][pair] = count
```

Figure 2.2: Code to find the 5 most frequent noun-adj pairs for each business id selected randomly.

Business ID	Noun-Adjective Pair	Frequency
caq9CTtWB-8K0tdFUhTfAQ	food-italian	16
	food-great	13
	food-good	12
	onions-green	9
	sauce-creamy	8
	yogurt-frozen	29

R4R7ttLXfKKWM0VEMoaW4w	staff-friendly	16
	place-clean	12
	yogurt-best	9
	employees-friendly	7
mF2EW3twSrFPmT_RVV1-Qg	food-good	11
	food-great	9
	sushi-good	8
	food-chinese	8
	service-good	6
0kPm1zEpeXFRg8D2phqgCQ	coffee-good	6
	time-first	6
	coffee-hot	5
	staff-friendly	5
	service-worst	4
6RbCJLiwNYwS6ab9vzD_zg	pita-fresh	10
	staff-friendly	8
	food-lebanese	6
	bread-fresh	6
	food-great	5

Table 2.2: The top-5 Frequent Noun-Adjective Pairs for each chosen Business ID

The results are reasonable as a summary for a business. They allow us to learn certain things about a business. For example, for business id 'R4R7ttLXfKKWM0VEMoaW4w', we know that it is a clean yogurt shop, and the people there are friendly. However, they can only give a broad summary of the shop, as specific knowledge will come from Noun-Adjective pairs with a much smaller count and will not be reflected here. Another limitation is that a Noun-Adjective pair may not consider surrounding context and it can be misleading if we simply look at the Noun-Adjective pairs.

3 Application - Predicting Ratings from Reviews

3.1 Data Processing

We first need to convert the data into a suitable format before building a model. For our application, each data sample is simply 2 things - the review and the label (rating). First, we segment the review into sentences, then into tokens. We then convert each token to lowercase. We only keep tokens which are alphanumeric and removed stopwords. To convert tokens into a numeric vector for classification, for Support Vector Machine, K Nearest Neighbors, Random Forest, Logistic Regression, we used TF-IDF [2]. For Feed Forward Neural Network (FFN), we used GloVe word embeddings [3] instead. This was because TF-IDF gave a very large number of features causing our FFN to train slower and overfit very easily. When using the GloVe word embeddings, each review is represented as **the average of its word vectors.** Unknown words not in the GloVe dictionary are ignored. As for the labels, since the rating is from 1 to 5, we changed it to 0 to 4 instead.

Simply put, word embeddings are vector representations of a particular word. Using word embeddings can help to reduce the dimensionality of data. Also, words that share common contexts in the corpus are located close to each other in the vector space, which intuitively, give us better results. "pen" and "pencil" will have more similar vector representations compared to "pen" and "lion". We used the Tweets version of pretrained GloVe embeddings as we believe it is the most similar domain to our dataset.

3.2 Building models

We tried out several different classification models such as Support Vector Machine, K Nearest Neighbours, Random Forest, Logistic Regression and Feed-forward neural network. We used 5-fold stratified cross validation to find the best hyperparameters for each model. The models were all implemented with the Scikit-learn library except the Feed-forward neural network which was implemented with the Tensorflow library. We report the best hyperparameters for each model and their 5-fold validation accuracy.

Model	Hyperparameters	Accuracy

Support Vector Machine	C = 1
	gamma = 0.6
	kernel = 'rbf'
		61.4%

Feed Forward neural network	2 hidden layers (126-16) Used Dropout (rate=0.5) after each fully connected layer and EarlyStopping to reduce overfitting Loss function=categorical cross entropy Optimizer = Adam	59.9%

K Nearest Neighbours	weights = 'uniform'
	n_neighbors = 85
		52.8%

Random Forest	n_estimators = 168	55.3%

Logistic Regression	penalty = 'l2'
	C = 2.6
		61.0%

Table 3.2: The hyperparameters and accuracy for each model.

3.3 Limitations

The first problem is that GloVe embeddings always gives the same vector representation for the same word, regardless of the context. However, the same word can have different meanings in different scenarios. E.g. "River bank", "Save money in a bank". This can be addressed using BERT embedding models instead [4]. Another problem is that by averaging the vectors for each sentence, we lose the sequential information in a sentence. E.g. "Boy eats food" is the same as "Food eats boy". This can be addressed using Bidirectional-LSTM [5].

CONTRIBUTIONS

Ang Jun Liang - Data Analysis, Noun-Adjective Pair Summarizer, Application, Coding
Nigel Ang Wei Jun - POS Tagging, Noun-Adjective Pair Summarizer
Yee Wei Min - Most Frequent Adjectives for each Rating, Report Formatting
Nguyen Ngoc Khanh - Sentence Segmentation
Lee Yi Yan, Esther - Tokenization and Stemming

THIRD PARTY LIBRARIES

All analysis is performed in Python. The libraries used are: numpy [6], pandas [7], TensorFlow [8], nltk [9], spaCy [10].

REFERENCES

[1] Kiss, T., & Strunk, J. (2006). Unsupervised multilingual sentence boundary detection. Computational Linguistics, 32(4), 485-525.
[2] Pennington, J., Socher, R., & Manning, C. (2014, October). Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP) (pp. 1532-1543).
[3] Ramos, J. (2003, December). Using tf-idf to determine word relevance in document queries. In Proceedings of the first instructional conference on machine learning (Vol. 242, pp. 133-142).
[4] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
[5] Schuster, M., & Paliwal, K. K. (1997). Bidirectional recurrent neural networks. IEEE Transactions on Signal Processing, 45(11), 2673-2681.
[6] NumPy developers. (2019). *"NumPy"*. Retrieved from https://numpy.org/
[7] Anonymous. "Pandas - PyData". Retrieved from https://pandas.pydata.org/
[8] Anonymous. "TensorFlow.org." Retrieved from https://www.tensorflow.org/.
[9] Anonymous. (2019, August). "Natural Language Toolkit — NLTK 3.4.5 ...." Retrieved from https://www.nltk.org/.
[10] Anonymous. "spaCy." Retrieved from https://spacy.io/.

APPENDIX

1 List of Stop Words Used

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