Attn: Dr. Sun Aixin

NANYANG TECHNOLOGICAL UNIVERSITY

CE/CZ4045 Natural Language Processing

We hereby declare that the attached group assignment has been researched, undertaken, completed and submitted as a collective effort by the group members listed below. We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work. We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work.

Important note: Name must **EXACTLY MATCH** the one printed on your Matriculation Card. Any mismatch leads to **THREE (3)** marks deduction.

Name	Signature / Date
Tan Ching Fhen	[illegible] /25102021
Wei Yao	[illegible] /25102021
Cui Chenling	[illegible] /25102021
Ta Quynh Nga	[illegible] /25102021
Luo Wenyu	Wenyu /25102021

1 Introduction

Natural Language Processing has always been a hot topic in the domain of deep learning. With the development of open-source packages, the process has been made much easier for people to create interesting ideas and reliable results, as well as to achieve better understanding of multiple language tasks.

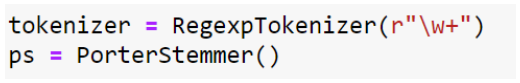
In this assignment, we aim to gain a better understanding of the main components in an end-to-end NLP application. The dataset of our interest is a subset of the famous Yelp review dataset. We first performed dataset analysis, which includes tokenization and stemming, POS (Parts-of-speech) Tagging, investigation of the writing style of contents from different platforms, and discussion on the most frequent <Noun- Adjective> pairs for each rating. The second step is to find out and analyze the indicative adjective phrases of the reviews from a particular business. The last step of the project is to develop a negation detection application to correctly identifies various negation expressions.

2 Dataset Analysis

2.1 Tokenization and Stemming

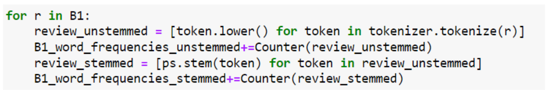
The objective of this part is to explore the word distributions in 2 businesses after removing the stop words.

2.1.1 Processing. We randomly selected 2 businesses, *b1* and *b2*, and extracted their reviews to form datasets, *B1* and *B2*. For each dataset, we performed tokenization and stemming (figure 2) using *Regexptokenizer* and *PorterStemmer* respectively (figure 1) from the *NLTK* toolkit.



```
tokenizer = RegexpTokenizer(r"\w+")
ps = PorterStemmer()
```

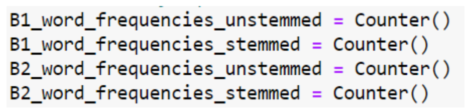
Figure 1: NLTK tokenizer and stemmer



```
for r in B1:
    review_unstemmed = [token.lower() for token in tokenizer.tokenize(r)]
    B1_word_frequencies_unstemmed+=Counter(review_unstemmed)
    review_stemmed = [ps.stem(token) for token in review_unstemmed]
    B1_word_frequencies_stemmed+=Counter(review_stemmed)
```

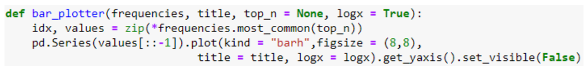
Figure 2: tokenization and stemming on B1

All word frequencies are stored in *Counter* objects from the *collections* library (figure 3). Next, we defined *bar_plotter()* function which plots the word distribution in log-scale (figure 4).



```
B1_word_frequencies_unstemmed = Counter()
B1_word_frequencies_stemmed = Counter()
B2_word_frequencies_unstemmed = Counter()
B2_word_frequencies_stemmed = Counter()
```

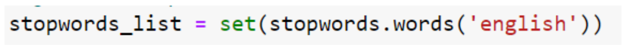
Figure 3: Counter objects storing word frequencies



```
def bar_plotter(frequencies, title, top_n = None, logx = True):
    idx, values = zip(*frequencies.most_common(top_n))
    pd.Series(values[::-1]).plot(kind = "barh",figsize = (8,8),
                            title = title, logx = logx).get_yaxis().set_visible(False)
```

Figure 4: Function that plots the word distributions in log-scale

For the removal of stop words, in addition to the previous codes, we check every word, if it is a stop word listed in *NLTK* (figure 5).



```
stopwords_list = set(stopwords.words('english'))
```

Figure 5: NLTK stop words list

2.1.2 Word Distribution Plots in log-scale.

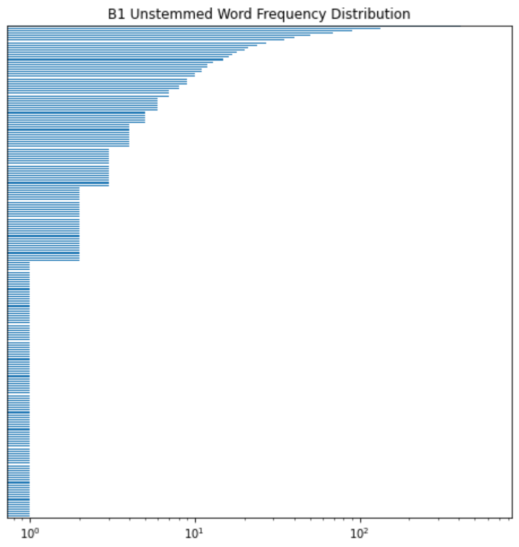


Figure 6: B1 unstemmed word frequency distribution

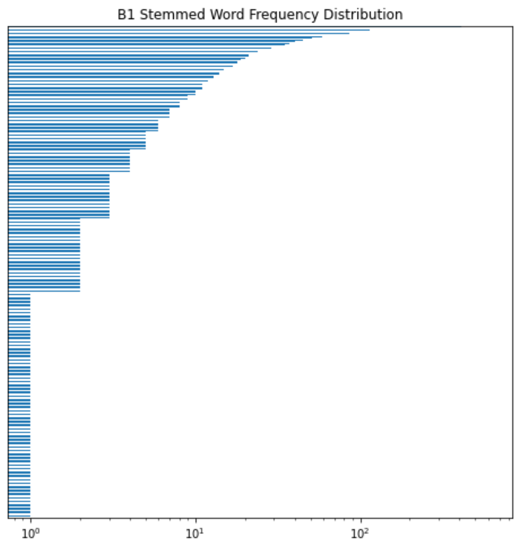


Figure 7: B1 stemmed word frequency distribution

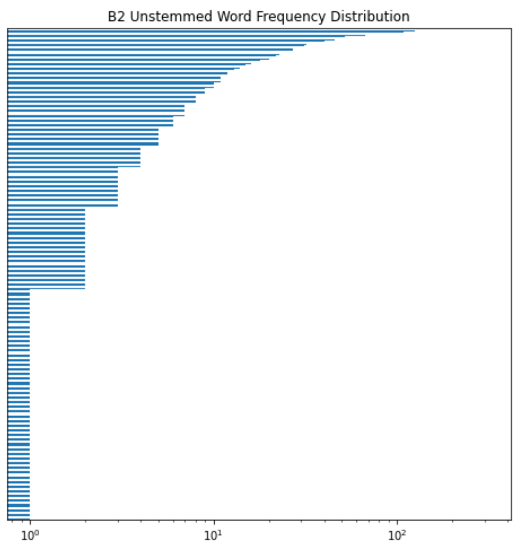


Figure 8: B2 unstemmed word frequency distribution

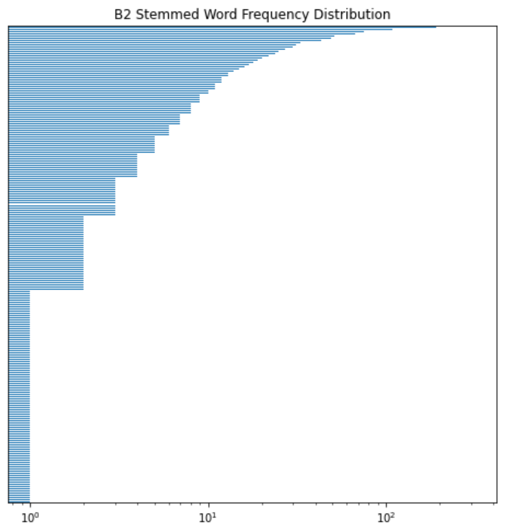


Figure 9: B2 stemmed word frequency distribution

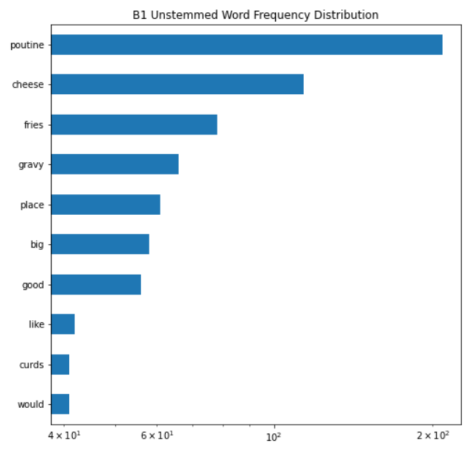


Figure 10: B1 Top 10 frequent non-stop-words unstemmed

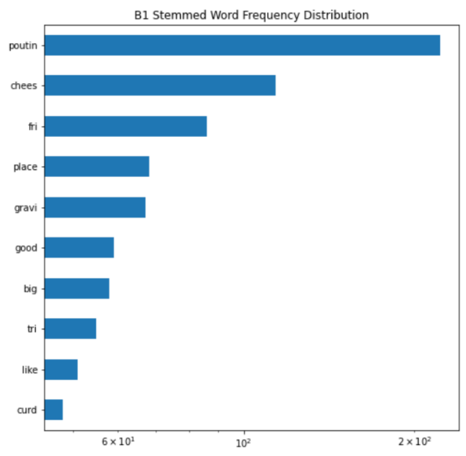


Figure 11: B1 Top 10 frequent non-stop-words stemmed

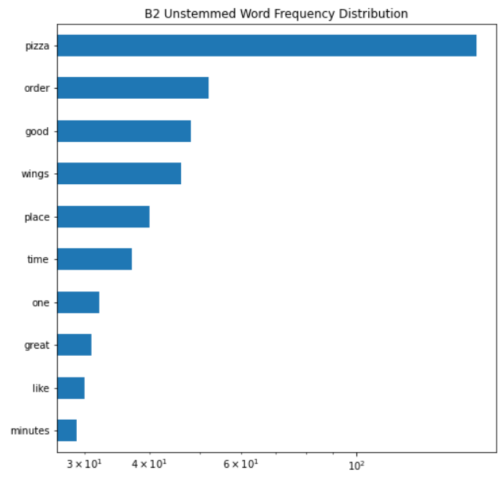


Figure 12: B2 Top 10 frequent non-stop-words unstemmed

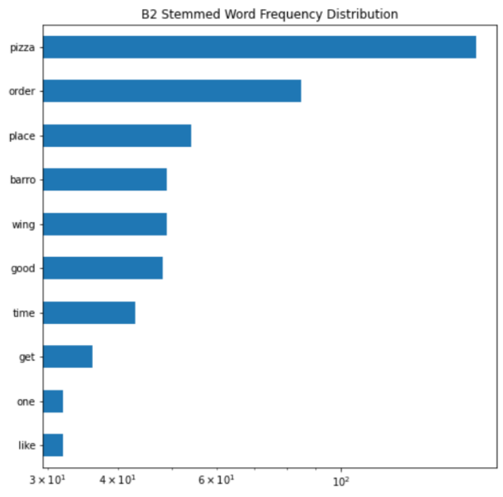


Figure 13: B2 Top 10 frequent non-stop-words stemmed

2.1.3 Discussions. For both B1 and B2, the word distributions are similar for before and after stemming. A large bulk of words are rare and occur only once, as can be seen in the long tails on the plots (Figure 6-9). On the other hand, the most frequently used words are usually stop words like “the” and “to” and so on.

After the removal of stop words, the top 10 most frequently used words are domain related words. We can determine the line of business based on those words. For example, in B1, “poutine”, “cheese” and “fries” are the most frequently used words (Figure 10), so we can imply that b1 sells mostly poutine which is a form of cheese fries. We can also determine the general sentiment towards the restaurant based on the frequent words. For instance, we can imply that the sentiment towards b2 is positive because the words “good” and “great” are frequent (Figure 12).

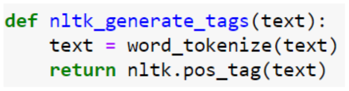
After removal of stop words, the order of top-10 most frequent words can change. For instance, in both B1 and B2, the word “place” increased in ranking (Figure 10-13). Furthermore, many words have lost their meaning due to the stemming procedure. For example, it is unclear what “tri” and “barro” refer to in B1 and B2 (Figure 11 & 13).

Having discussed the findings, we can conclude that it is important to determine the right context to use stemming. If we are performing information retrieval, stemming can be useful to group similar words together to gain more search results. If we are doing sentiment analysis, stemming should not be used because it can remove the semantics and sentiments, which are important.

2.2 POS Tagging

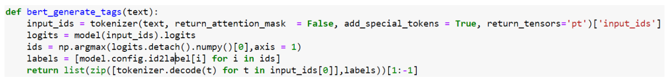
The objective of this part is to explore 2 different POS Tagging methods on 5 selected sentences from the dataset.

2.2.1 NLTK and Transformers. We randomly sampled 5 reviews and applied POS tagging using *NLTK* and a fine-tuned Transformers model, “vblagoje/bert-english-uncased-finetuned-pos", using the *transformers* library.



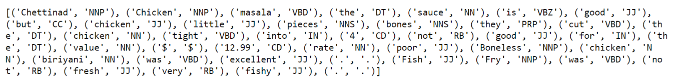
```
def nltk_generate_tags(text):
    text = word_tokenize(text)
    return nltk.pos_tag(text)
```

Figure 14: Function that performs POS tagging with NLTK



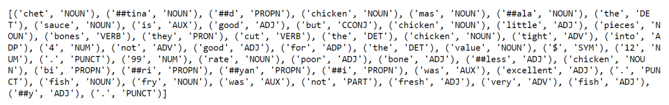
```
def bert_generate_tags(text):
    input_ids = tokenizer(text, return_attention_mask  = False, add_special_tokens = True, return_tensors='pt')['input_ids']
    logits = model(input_ids).logits
    ids = np.argmax(logits.detach().numpy()[0],axis = 1)
    labels = [model.config.id2label[i] for i in ids]
    return list(zip([tokenizer.decode(t) for t in input_ids[0]],labels))[1:-1]
```

Figure 15: Function that performs POS tagging with BERT



```
[('Chettinad', 'NNP'), ('Chicken', 'NNP'), ('masala', 'VBD'), ('the', 'DT'), ('sauce', 'NN'), ('is', 'VBZ'), ('good', 'JJ'),
('but', 'CC'), ('chicken', 'JJ'), ('little', 'JJ'), ('pieces', 'NNS'), ('bones', 'NNS'), ('they', 'PRP'), ('cut', 'VBD'), ('th
e', 'DT'), ('chicken', 'NN'), ('tight', 'VBD'), ('into', 'IN'), ('4', 'CD'), ('not', 'RB'), ('good', 'JJ'), ('for', 'IN'), ('th
e', 'DT'), ('value', 'NN'), ('$', '$'), ('12.99', 'CD'), ('rate', 'NN'), ('poor', 'JJ'), ('Boneless', 'NNP'), ('chicken', 'N
N'), ('biriyani', 'NN'), ('was', 'VBD'), ('excellent', 'JJ'), ('.', '.'), ('Fish', 'JJ'), ('Fry', 'NNP'), ('was', 'VBD'), ('no
t', 'RB'), ('fresh', 'JJ'), ('very', 'RB'), ('fishy', 'JJ'), ('.', '.')]
```

Figure 16: POS tagging with NLTK



```
[('chet', 'NOUN'), ('##tina', 'NOUN'), ('##d', 'PROPN'), ('chicken', 'NOUN'), ('mas', 'NOUN'), ('##ala', 'NOUN'), ('the', 'DE
T'), ('sauce', 'NOUN'), ('is', 'AUX'), ('good', 'ADJ'), ('but', 'CCONJ'), ('chicken', 'NOUN'), ('little', 'ADJ'), ('pieces', 'N
OUN'), ('bones', 'VERB'), ('they', 'PRON'), ('cut', 'VERB'), ('the', 'DET'), ('chicken', 'NOUN'), ('tight', 'ADV'), ('into', 'A
DP'), ('4', 'NUM'), ('not', 'ADV'), ('good', 'ADJ'), ('for', 'ADP'), ('the', 'DET'), ('value', 'NOUN'), ('$', 'SYM'), ('12', 'N
UM'), ('.', 'PUNCT'), ('99', 'NUM'), ('rate', 'NOUN'), ('poor', 'ADJ'), ('bone', 'ADJ'), ('##less', 'ADJ'), ('chicken', 'NOU
N'), ('bi', 'PROPN'), ('##ri', 'PROPN'), ('##yan', 'PROPN'), ('##i', 'PROPN'), ('was', 'AUX'), ('excellent', 'ADJ'), ('.', 'PUN
CT'), ('fish', 'NOUN'), ('fry', 'NOUN'), ('was', 'AUX'), ('not', 'PART'), ('fresh', 'ADJ'), ('very', 'ADV'), ('fish', 'ADJ'),
('##y', 'ADJ'), ('.', 'PUNCT')]
```

Figure 17: POS tagging with BERT

2.2.2 Discussions. The two approaches vary in the nomenclature of the POS tags. For instance, nouns are labeled as “NOUN” in Transformers, but NLTK labels them as “NN.” (Figure 16-17) Also, they differ in terms of tokenization. BERT uses wordpiece tokenization, so words like “Chettinad” can be split into subwords, “chet”, “##tin” and “##d,” where “##d” was given a different POS tag (Figure 17). On the other hand, NLTK tokenization does not split words into sub-words (Figure 16).

2.2.3 Conclusion. In conclusion, though POS tagging with Transformers may have more accurate POS tags due to the use of contextualized word embeddings for token classification, it may not be ideal for POS tagging. The reason being it breaks down some words into sub-words which may have different POS tags - in turn requiring additional processing. Furthermore, it is much slower in runtime. NLTK may be a sufficient toolkit for POS tagging.

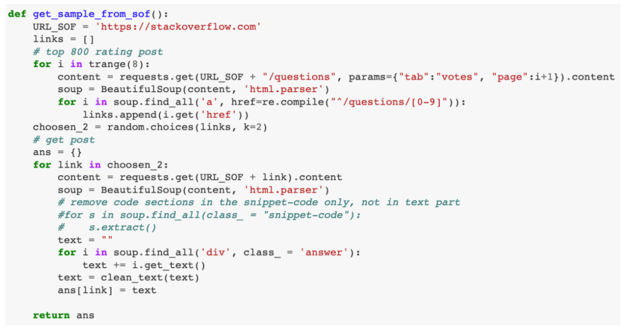
2.3 Writing Style

In this section, in total, 2 randomly generated posts were scraped from each of the 3 different websites: StackOverFlow, Hardwarezone and ChannelNewsAsia. Their writing styles were then investigated in 3 aspects:

1. Are the first words of the sentences capitalized?
2. Are proper nouns capitalized?
3. Do they follow good grammar in general?

Moreover, we also investigated whether we can directly apply tokenizer and POS tagger by NLTK on StackOverflow posts.

2.3.1 Generation of random posts. Web scraping was performed to get posts from the websites. For each website, among all posts scraped, only 2 will be selected via random selection.



```
def get_sample_from_sof():
    URL_SOF = 'https://stackoverflow.com'
    links = []
    # top 800 rating post
    for i in trange(8):
        content = requests.get(URL_SOF + "/questions", params={"tab":"votes", "page":i+1}).content
        soup = BeautifulSoup(content, 'html.parser')
        for i in soup.find_all('a', href=re.compile("^/questions/[0-9]")):
            links.append(i.get('href'))
    choosen_2 = random.choices(links, k=2)
    # get post
    ans = {}
    for link in choosen_2:
        content = requests.get(URL_SOF + link).content
        soup = BeautifulSoup(content, 'html.parser')
        # remove code sections in the snippet-code only, not in text part
        #for s in soup.find_all(class_ = "snippet-code"):
        #    s.extract()
        text = ""
        for i in soup.find_all('div', class_ = 'answer'):
            text += i.get_text()
        text = clean_text(text)
        ans[link] = text

    return ans
```

Figure 18: An example of web scraping from StackOverFlow

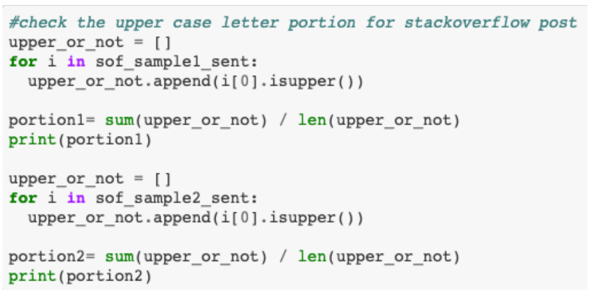
As shown above, we scrape 10 questions in each page and scraped 8 pages in total. Among the 80 questions, 2 were randomly selected and the answers were scraped out using BeautifulSoup. Similar processes were done for posts from Hardwarezone and ChannelNewsAsia as well.

Table 1: URLs of randomly selected posts

Website	URL
StackOverFlow	https://www.stackoverflow.com/ /questions/46155/how-to-validate-an-email-address-in-javascript
StackOverFlow	https://www.stackoverflow.com/ /questions/1200621/how-do-i-declare-and-initialize-an-array-in-java
Hardwarezone	https://forums.hardwarezone.com.sg/threads/anyone-know-what-happens-to-kpc-auntie-and-her-friend.6614765/
Hardwarezone	https://forums.hardwarezone.com.sg/threads/itz-is-time-to-tell-him-hes-not-good-enough-hes-gotta-go.6614774/
ChannelNewsAsia	https://www.channelnewsasia.com/sustainability/global-warming-1-5-degrees-celsius-climate-change-221943
ChannelNewsAsia	https://www.channelnewsasia.com/living/how-make-dalgona-candy-squid-game-2224831

After obtaining the random URLs and its contents, we performed several grammar checks to determine the differences in their writing styles among all posts.

2.3.2 Check first words of the sentences. To check whether the first words of the sentences in each post are capitalized, we used *isupper()* to check and computed the proportion of the sentences with first word being capitalized in all sentences in the particular post.



```
#check the upper case letter portion for stackoverflow post
upper_or_not = []
for i in sof_sample1_sent:
  upper_or_not.append(i[0].isupper())

portion1= sum(upper_or_not) / len(upper_or_not)
print(portion1)

upper_or_not = []
for i in sof_sample2_sent:
  upper_or_not.append(i[0].isupper())

portion2= sum(upper_or_not) / len(upper_or_not)
print(portion2)
```

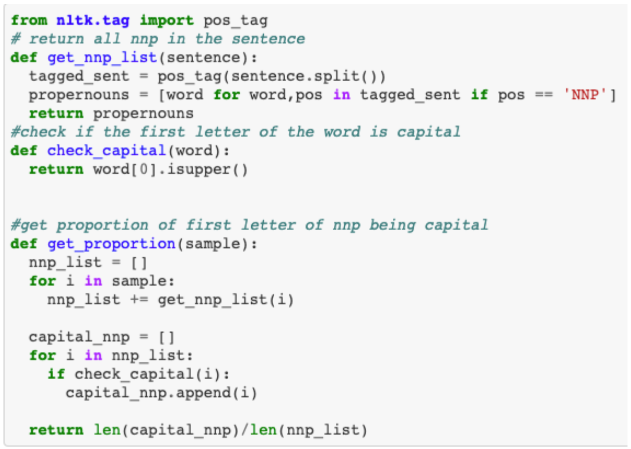
Figure 19: Code to check for capitalized first words

As shown in the table below, the proportion of first letter of the sentence being uppercase letter in StackOverFlow post is 0.5-0.6, the proportion of first letter of the sentence being an uppercase letter in Hardwarezone post is 0.5-0.6, while that in ChannelNewsAsia post is 0.9-1.0. This clearly shows that posts from CNA shows better grammar in terms of capitalization of the first word.

Table 2: Proportion of capitalized first word

Website	Proportion of capitalized first word
StackOverFlow1	0.5
StackOverFlow2	0.607
Hardwarezone1	0.6
Hardwarezone2	0.5
ChannelNewsAsia1	1.0
ChannelNewsAsia2	0.9

2.3.3 Check for proper nouns. To check whether the proper nouns are capitalized., POS tagging was done on the posts to identify the proper nouns. Similar processes as 2.3.2 were done on these nouns.



```
from nltk.tag import pos_tag
# return all nnp in the sentence
def get_nnp_list(sentence):
  tagged_sent = pos_tag(sentence.split())
  propernouns = [word for word,pos in tagged_sent if pos == 'NNP']
  return propernouns
#check if the first letter of the word is capital
def check_capital(word):
  return word[0].isupper()


#get proportion of first letter of nnp being capital
def get_proportion(sample):
  nnp_list = []
  for i in sample:
    nnp_list += get_nnp_list(i)

  capital_nnp = []
  for i in nnp_list:
    if check_capital(i):
      capital_nnp.append(i)

  return len(capital_nnp)/len(nnp_list)
```

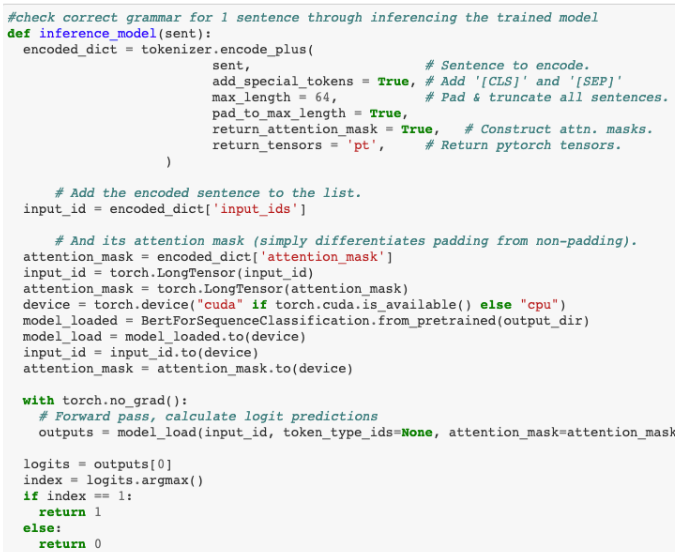
Figure 20: Check for whether first letter of the proper nouns is capitalized

As shown below, the proportion of first letter of proper noun being a capital letter in StackOverFlow post is 0.47-0.62, that in Hardwarezone forum post is 0.07-0.26, while that of ChannelNewsAsia post is 0.83-0.96. This clearly shows that most of the proper nouns in CNA posts are properly capitalized, indicating good grammar in CNA posts as compared to StackOverFlow and Hardwarezone forum posts, in terms of capitalized proper noun.

Table 3: Proportion of capitalized proper nouns

Website	Proportion of capitalized proper nouns
StackOverFlow1	0.62
StackOverFlow2	0.47
Hardwarezone1	0.07
Hardwarezone2	0.26
ChannelNewsAsia1	0.96
ChannelNewsAsia2	0.83

2.3.4 Grammar checker – A deep learning approach. We adopted a deep learning method to check for grammar correctness by finetuning the pre-trained BERT model using the CoLA dataset (Tanwar, 2021). The dataset consists of a set of sentences labelled as grammatically correct or incorrect. This method can cover most of the grammatical errors that may be present in the sentences such as tense errors and incoherent sentence structures. Since the model works for single sentence classification, we fed the model with single sentences one by one.



```
#check correct grammar for 1 sentence through inferencing the trained model
def inference_model(sent):
  encoded_dict = tokenizer.encode_plus(
                        sent,                      # Sentence to encode.
                        add_special_tokens = True, # Add '[CLS]' and '[SEP]'
                        max_length = 64,           # Pad & truncate all sentences.
                        pad_to_max_length = True,
                        return_attention_mask = True,   # Construct attn. masks.
                        return_tensors = 'pt',     # Return pytorch tensors.
                  )

      # Add the encoded sentence to the list.
  input_id = encoded_dict['input_ids']

      # And its attention mask (simply differentiates padding from non-padding).
  attention_mask = encoded_dict['attention_mask']
  input_id = torch.LongTensor(input_id)
  attention_mask = torch.LongTensor(attention_mask)
  device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
  model_loaded = BertForSequenceClassification.from_pretrained(output_dir)
  model_load = model_loaded.to(device)
  input_id = input_id.to(device)
  attention_mask = attention_mask.to(device)

  with torch.no_grad():
    # Forward pass, calculate logit predictions
    outputs = model_load(input_id, token_type_ids=None, attention_mask=attention_mask

  logits = outputs[0]
  index = logits.argmax()
  if index == 1:
    return 1
  else:
    return 0
```

Figure 21: Code to inference DL model for individual sentences

As shown below, the proportion of the grammatically correct sentences in StackOveFlow post samples is 0.855-0.865, the proportion in Hardwarezone post samples is 0.57-0.60, while the proportion of grammatically correct sentences in ChannelNewsAsia is 1.0. This shows that the posts from CNA are grammatically accurate while the posts from Hardwarezone is the most grammtically inaccurate ones, due to the fact that people use a lot of slangs when posting on the forum.

Table 4: Proportion of grammatically correct sentences

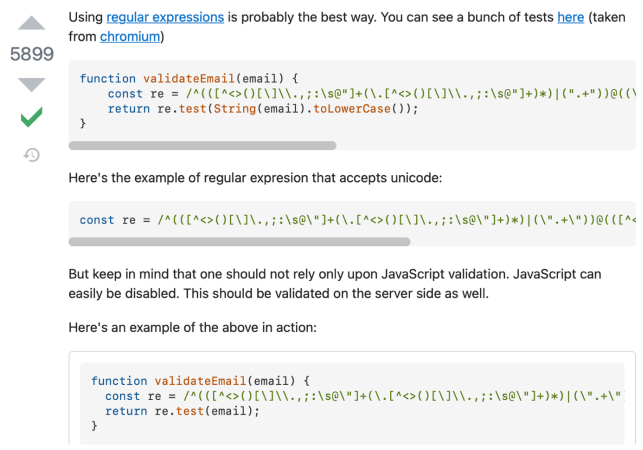
Website	Proportion of grammatically correct sentences
StackOverFlow1	0.86
StackOverFlow2	0.87
Hardwarezone1	0.6
Hardwarezone2	0.57
ChannelNewsAsia1	1.0
ChannelNewsAsia2	1.0

In conclusion, in all 3 aspects, posts from CNA are grammatically correct than other posts. This is largely relevant to the charateristics of these websites and formality of the English used. The posts from StackOverFlow contain code snippets or code blocks, which may make it harder for us to detect the tags correctly, this issue will be discussed later. The posts from Hardwarezone Forum involve casual talks and the use of emojis. Slang words are also present in these posts since it is a free forum where people do not have to limit to formality of English. On the other hand, the posts from ChannelNewsAsia are written in formal English, hence, they have a higher accuracy in terms of grammar usage.

2.3.5 Investigate if NLTK tokenizer and POS tagger can be applied directly on StackOverflow posts. Firstly, we try to apply NLTK tokenizer and POS tagger directly on the 2 StackOverflow posts. The results of the above process of the first 50 tokens in each post are shown below:

No.	Link 1 Tokenization	Link 1 POS tagging	Link 2 Tokenization	Link 2 POS tagging
1	5868	('5868', 'CD')	2947	('2947', 'CD')
2	Using	('Using', 'VBG')	You	('You', 'PRP')

3	regular	('regular', 'JJ')	can	('can', 'MD')
4	expressions	('expressions', 'NNS')	either	('either', 'VB')
5	is	('is', 'VBZ')	use	('use', 'JJ')
6	probably	('probably', 'RB')	array	('array', 'NN')
7	the	('the', 'DT')	declaration	('declaration', 'NN')
8	best	('best', 'JJS')	or	('or', 'CC')
9	way	('way', 'NN')	array	('array', 'JJ')
10	.	('.', '.')	literal	('literal', 'JJ')
11	You	('You', 'PRP')	(	('(', '(')
12	can	('can', 'MD')	but	('but', 'CC')
13	see	('see', 'VB')	only	('only', 'RB')
14	a	('a', 'DT')	when	('when', 'WRB')
15	bunch	('bunch', 'NN')	you	('you', 'PRP')
16	of	('of', 'IN')	declare	('declare', 'VBP')
17	tests	('tests', 'NNS')	and	('and', 'CC')
18	here	('here', 'RB')	affect	('affect', 'VBP')
19	(	('(', '(')	the	('the', 'DT')
20	taken	('taken', 'VBN')	variable	('variable', 'JJ')
21	from	('from', 'IN')	right	('right', 'NN')
22	chromium	('chromium', 'NN')	away	('away', 'RB')
23	)	(')', ')')	,	(',', ',')
24	function	('function', 'NN')	array	('array', 'JJ')
25	validateEmail	('validateEmail', 'NN')	literals	('literals', 'NNS')
26	(	('(', '(')	can	('can', 'MD')
27	email	('email', 'NN')	not	('not', 'RB')
28	)	(')', ')')	be	('be', 'VB')
29	{	('{', '(')	used	('used', 'VBN')
30	const	('const', 'JJ')	for	('for', 'IN')
31	re	('re', 'NN')	re-assigning	('re-assigning', 'VBG')
32	=	('=', 'NNP')	an	('an', 'DT')
33	/^	('/^', 'NNP')	array	('array', 'NN')
34	(	('(', '(')	)	(')', ')')
35	(	('(', '(')	.For	('.For', 'FW')
36	[	('[', 'VB')	primitive	('primitive', 'JJ')
37	^	('^', 'NNP')	types	('types', 'NNS')
38	<	('<', 'NNP')	:	(':', ':')
39	>	('>', 'NNP')	int	('int', 'NN')
40	(	('(', '(')	[	('[', 'NN')
41	)	(')', ')')	]	(']', 'NNP')
42	[	('[', 'VBZ')	myIntArray	('myIntArray', 'NN')
43	]	(']', 'JJ')	=	('=', 'NNP')
44	.	('.', '.')	new	('new', 'JJ')
45	,	(',', ',')	int	('int', 'NN')
46	;	(';', ':')	[	('[', 'VBZ')
47	:	(':', ':')	3	('3', 'CD')
48	s	('s', 'NN')	]	(']', 'NN')
49	@	('@', 'NN')	;	(';', ':')
50	''	("''", "''")	int	('int', 'NN')



5899

Using regular expressions is probably the best way. You can see a bunch of tests here (taken from chromium)

```
function validateEmail(email) {
    const re = /^(([^<>()[\]\\.,;:\s@"]+(\.[^<>()[\]\\.,;:\s@"]+)*)|(".+"))@((\
    return re.test(String(email).toLowerCase());
}
```

Here's the example of regular expresion that accepts unicode:

```
const re = /^(([^<>()[\]\.,;:\s@\"]+(\.[^<>()[\]\.,;:\s@\"]+)*)|(\".+\"))@(([^<
```

But keep in mind that one should not rely only upon JavaScript validation. JavaScript can easily be disabled. This should be validated on the server side as well.

Here's an example of the above in action:

```
function validateEmail(email) {
  const re = /^(([^<>()[\]\\.,;:\s@\"]+(\.[^<>()[\]\\.,;:\s@\"]+)*)|(\".+\"
  return re.test(email);
}
```

Figure 22: The HTML display of the first StackOverflow post.

We can see that the tokenizer and POS tagger perform not well particularly at the code block in the contents.

For example, in the first link, there is a code block `const re = /^(([^<>()[\]\.,;:\s@\"]+(\.[^<>()[\]\.,;:\s@\"]+)*)|(\".+\"))@(([^<>()[\]\.,;:\s@\"]+\.)+[^<>()[\]\.,;:\s@\"]{2,})` (was tokenized from line 30 onwards) in which for the whole right hand side after the `re =`, we think it should be considered as one token so that the POS tagger would tag them as one single part, or if tokenizer tokenizes each punctuation as one token, the POS tagger should tag each of them as PUNCTUATION or something else instead of `NNP` for = in line 32, `VB` for `[` in line 36, `VBZ` for `[`in line 42, `JJ` for `]` in line 43, etc.

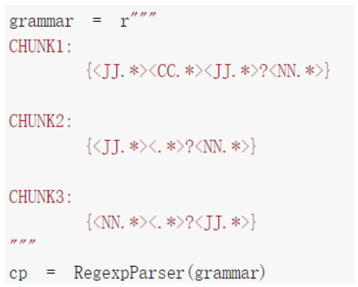
For those code snippets and code blocks like this, it's ambiguous for even human to tokenize and tag POS. Therefore, we think it is not necessary to do tokenization and POS tagging in the code blocks, and we would remove those code blocks as a preprocessing stage before using the NLTK tools to tokenize and tag POS.

The first solution we can think of is instead of retrieving the plain text of the StackOverflow content, we retrieve the HTML content of the webpage, then do find the code snippet/ block (which in the StackOverflow HTML script is tagged as `<code></code>`) and remove the code snippet/block. After the removal, we extract the text inside the HTML tags and do the tokenization and POS Tagging as usual. The second solution is that, instead of removing the code snippets and code blocks, we replace those with UNK stands for UNKNOWN POS tag, then apply the tokenizer and POS tagger as usual.

2.4 Most frequent <Noun – Adjective> pairs for each rating

The main approach is to use chunking in parsing the sentences. As the first step, we randomly selected 1 review from 50 distinct business id with rating 1 and do POS tagging after padding all the punctuations. The reason for padding the punctuations is to ensure that 'tasty, food' and 'tasty food' will not be recognized as different pairs. A list of tagged sentences will be returned for grammar parsing.

It is difficult to cover all the possible patterns of which a noun - adjective pair could occur given our insufficient domain knowledge. However, we noticed that some of the common patterns could be easily extracted:



```
grammar  =  r"""
CHUNK1:
        {<JJ.*><CC.*><JJ.*>?<NN.*>}

CHUNK2:
        {<JJ.*><.*>?<NN.*>}

CHUNK3:
        {<NN.*><.*>?<JJ.*>}
"""
cp  =  RegexpParser(grammar)
```

Figure 23: Code for generating parse tree using chunking grammar

Chunk 1 represents the pattern <adjective> <conjunction> <adjective> <noun>.

Some examples could be 'sweet and sour chicken' and 'delicious but expensive food'. In the example 'sweet and sour chicken', there exist two noun-adjective pairs: {sweet chicken} and {sour chicken}. While in the example 'delicious but expensive food', there exist two pairs as well: {delicious food} and {expensive food}.

Chunk 2 represents the pattern <adjective> <noun>.

Some examples could be 'good service', 'terrible taste', and ' wonderful experience', where the pattern itself represents the noun-adjective pair to be extracted .

Chunk 3 represents the pattern <noun> <one or two words> <adjective>.

Some examples could be 'the food is good', 'service extremely slow', 'cream is not tasty'. The adjective might not be directly next to the noun but is still describing it. In the example 'the food is good', the noun-adjective pair would be {good food}. While in the example 'service extremely slow', the pair would be {slow service}. However, in the phrase 'cream is not tasty', the negation word 'not' is not an adjective but is describing the noun 'cream'. Therefore, the noun-adjective pair {tasty cream} extracted will not be the most accurate pair.

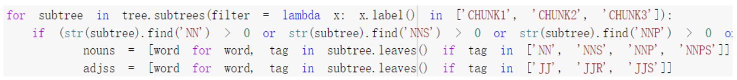
Below shows an example of the parse tree generated from the chunking grammar defined for the review "Always an awesome place to meet and hang out w/ friends for a good time. The food is fantastic for what is really primarily a bar.... go for the fish tacos!"

(S Always/NNS an/DT (CHUNK2 awesome/JJ place/NN) to/TO meet/VB and/CC hang/VB out/RP (CHUNK2 w//JJ friends/NNS) for/IN a/DT (CHUNK2 good/JJ time/NN) ./. The/DT (CHUNK3 food/NN is/VBZ fantastic/JJ) for/IN what/WP is/VBZ really/RB primarily/RB a/DT bar/NN ./. ./. ./. ./. go/VB for/IN the/DT (CHUNK2 fish/JJ tacos/NN) !/. !/.)

From the parse tree, we identify subtrees with the root being CHUNK1, 2 or 3 and extract the pair of noun and adjective words in the subtrees. In the above examples, the pairs identified and their frequencies are :

('awesome place', 1), ('w/ friends', 1), ('good time', 1), ('fantastic food', 1), ('fish tacos', 1)

When identifying noun adjective pair from subtrees, any word with POS tag NN, NNS, NNP or NNPS – singular or plural nouns and proper nouns, are defined as noun. While any word with POS tag JJ, JJR or JJS – Adjective, competitive or superlative adjectives, are defined as adjectives.



```
for  subtree  in  tree.subtrees(filter  =  lambda  x:  x.label()  in  ['CHUNK1',  'CHUNK2',  'CHUNK3']):
    if  (str(subtree).find('NN')  >  0  or  str(subtree).find('NNS')  >  0  or  str(subtree).find('NNP')  >  0  o
            nouns  =  [word  for  word,  tag  in  subtree.leaves()  if  tag  in  ['NN',  'NNS',  'NNP',  'NNPS']]
            adjss  =  [word  for  word,  tag  in  subtree.leaves()  if  tag  in  ['JJ',  'JJR',  'JJS']]
```

Figure 24: Code for finding noun and adjective from subtrees

It is observed that the order of chunking grammar affects the result largely, as each word can only belong to one subtree. We have therefore investigated on the frequency of the three chunking grammar that we have defined. The function get_pattern_count takes only one grammar at a time, parses the randomly selected 500 samples using the grammar and returns the count of chunks identified. We have done 20 times of randomly sampling to get a more accurate distribution of chunk frequency.

As a result, the grammar {<JJ.*><.*>?<NN.*>} (CHUNK2) has the highest frequency, followed by {<NN.*><.*>?<JJ.*>}

(CHUNK3) and lastly {<JJ.*><CC.*><JJ.*>?<NN.*>} (CHUNK1).

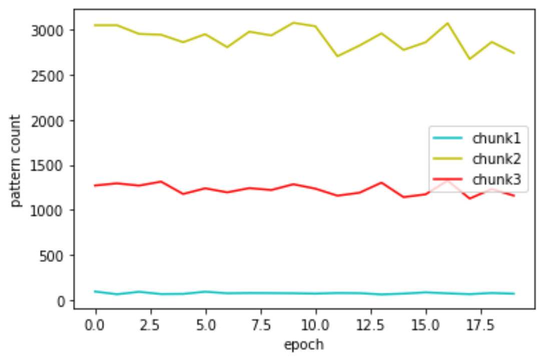


Figure 25: Plot for frequency of chunks in 20 times of random sampling

However, considering that CHUNK1 is a superset of CHUNK2, CHUNK1 needs to be passed in earlier than CHUNK2 despite having a much lower frequency. For the example 'sweet and sour chicken', passing in CHUNK2 grammar first will result in the below subtree structure:

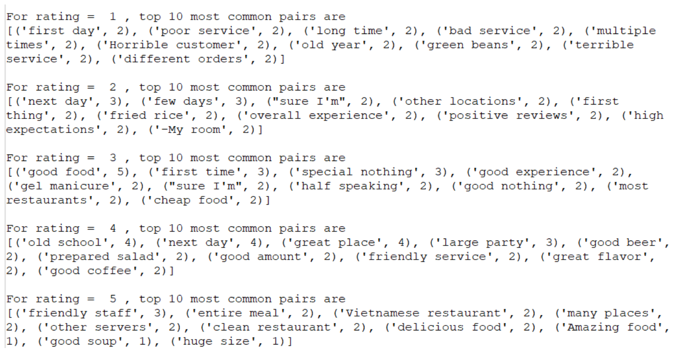
sweet/JJ and/CC (CHUNK2 sour/JJ chicken/NN)

While the expected subtree is:

(CHUNK1 sweet/JJ and/CC sour/JJ chicken/NN)

Therefore, CHUNK1 is passed in first, followed by CHUNK2 and then CHUNK3 which has a lower frequency.

The top 10 frequency noun-adjective pairs of 50 random reviews from random distinct businesses are shown below:



```
For rating =  1 , top 10 most common pairs are
[('first day', 2), ('poor service', 2), ('long time', 2), ('bad service', 2), ('multiple
times', 2), ('Horrible customer', 2), ('old year', 2), ('green beans', 2), ('terrible
service', 2), ('different orders', 2)]

For rating =  2 , top 10 most common pairs are
[('next day', 3), ('few days', 3), ("sure I'm", 2), ('other locations', 2), ('first
thing', 2), ('fried rice', 2), ('overall experience', 2), ('positive reviews', 2), ('high
expectations', 2), ('-My room', 2)]

For rating =  3 , top 10 most common pairs are
[('good food', 5), ('first time', 3), ('special nothing', 3), ('good experience', 2),
('gel manicure', 2), ("sure I'm", 2), ('half speaking', 2), ('good nothing', 2), ('most
restaurants', 2), ('cheap food', 2)]

For rating =  4 , top 10 most common pairs are
[('old school', 4), ('next day', 4), ('great place', 4), ('large party', 3), ('good beer',
2), ('prepared salad', 2), ('good amount', 2), ('friendly service', 2), ('great flavor',
2), ('good coffee', 2)]

For rating =  5 , top 10 most common pairs are
[('friendly staff', 3), ('entire meal', 2), ('Vietnamese restaurant', 2), ('many places',
2), ('other servers', 2), ('clean restaurant', 2), ('delicious food', 2), ('Amazing food',
1), ('good soup', 1), ('huge size', 1)]
```

Figure 26: Top 10 frequent noun adjective pairs generated

It is seen that though not absolutely accurate, some common pairs for lower rating include 'poor service' and 'horrible customer', while those for higher rating include 'friendly staff', 'clean restaurant' and 'good amount'. Overall, the sentiment is rather positive except for reviews with rating 1.

It is also found that the pairs identified are not all valid, such as 'sure I'm' and '-My room'. However, the proportion of such invalid pairs is not very significant in the results obtained.

3 Extraction of Indicative Adjective Phrases

In order to find the most indicative adjective phrases from the reviews, we have reused the approach adopted in part 2.4, chunking to obtain the adjective phrases. Firstly, we randomly select a business b1 by its *business_id* and obtain all the reviews for this specific business and manually go through all the reviews to look for patterns for adjective phrases. After going through the sampled reviews, we have analyzed some common adjective phrase patterns:

<JJ|JJS|JJR><CC>*<JJ|JJS|JJR|VBG>+

<RB|RBR|RBS*>+<JJ|JJR|JJS|VBN|VBG>+<CC>*<JJ|VBN|VBG>*

<JJR|VBD|VBN><IN><JJ|NN>

<DT><NN><JJ|JJR|JJS>

<JJ|JJR|JJS><JJ|JJR|JJS>

Chunk 1 represents the pattern **<adjective><conjunction><adjective>**. Some examples found are "fast and efficient", "friendly and accommodating" where adjectives are connected by a conjunction.

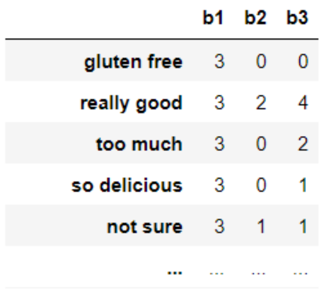
Chunk 2 represents the pattern **<adverb><adjective><conjunction><adjective>**. Some examples found are "not helpful and unclean" and "really great". Adverbs can have multiple occurrences in this case.

Chunk 3 represents the pattern **<adjective><preposition><adjective/noun>**. Some examples found are "cleaner than other", "brushed with butter" and "topped with cream". In this case, phrases that started with past tense verbs which usually describe the state of an object, or adjective phrases that show comparison are selected.

Chunk 4 represents the pattern <article/determiner><noun><adjective>. Examples include "a little bit", "a bit disappointing". This is not a very common pattern and have a relatively high chance to select non-adjective phrases like "a meal special". However, as we want to try to include as many patterns as possible.

Chunk 5 is a pattern that finds phrases with <adjective><adjective>. This is a pattern that makes sure those phrases with two consecutive adjective words are included. Examples such as "good enough" and "pretty decent" can be found.

After applying chunking on the selected business reviews with the above-mentioned rules, we obtain a list of extracted adjective phrases. To determine the most indicative adjective phrases, we randomly selected two other businesses, b2 and b3, and performed chunking with same set of grammar rules to obtain all adjective phrases found in those reviews. Our objective is to find phrases that appear often in b1 but relatively less often in b2 and b3. In order to find such phrases, we decided to use Bag of words and TF-IDF. We use bag of words to first create a set of vectors containing the count of occurrences of adjective phrases in these three businesses.



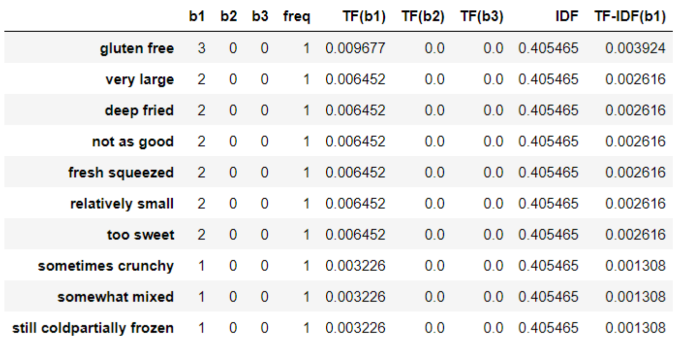
	b1	b2	b3
gluten free	3	0	0
really good	3	2	4
too much	3	0	2
so delicious	3	0	1
not sure	3	1	1
...	...	...	...

Figure 27: Bag of words vector table

We then calculated its TF and IDF score. Term Frequency (TF) measures how frequently a term t appears in a document d. Inverse Document Frequency (IDF) measures how important a term is. These two scores can be calculated by using the following formulas:

$$TF = \frac{No.\ of\ t\ in\ d}{No.\ of\ terms\ in\ d} \qquad IDF = log\frac{No.\ of\ documents}{No.\ of\ documents\ with\ term\ t+1}$$

Finally, we determine the most indicative adjective phrases by sorting the phrases based on TF-IDF score and obtain the top 10 phrases with highest TF-IDF score. A high TF-IDF score indicates that the phrase is not only appearing frequently in b1, but also occurring rarely in other businesses' reviews. Phrases that appear often not only in b1 but also in the other two businesses will be eliminated after sorting, such as "really good" that occurs 3 times in b1, 2 times in b2 and 4 times in b3. While phrases that are more indicative to b1 are being kept as shown in the table below.



	b1	b2	b3	freq	TF(b1)	TF(b2)	TF(b3)	IDF	TF-IDF(b1)
gluten free	3	0	0	1	0.009677	0.0	0.0	0.405465	0.003924
very large	2	0	0	1	0.006452	0.0	0.0	0.405465	0.002616
deep fried	2	0	0	1	0.006452	0.0	0.0	0.405465	0.002616
not as good	2	0	0	1	0.006452	0.0	0.0	0.405465	0.002616
fresh squeezed	2	0	0	1	0.006452	0.0	0.0	0.405465	0.002616
relatively small	2	0	0	1	0.006452	0.0	0.0	0.405465	0.002616
too sweet	2	0	0	1	0.006452	0.0	0.0	0.405465	0.002616
sometimes crunchy	1	0	0	1	0.003226	0.0	0.0	0.405465	0.001308
somewhat mixed	1	0	0	1	0.003226	0.0	0.0	0.405465	0.001308
still coldpartially frozen	1	0	0	1	0.003226	0.0	0.0	0.405465	0.001308

Figure 28: Result table containing top 10 most indicative adjective phrases

4 Application: Negation Detection

We built an NLP (Natural Language Processing) application that identifies negation expression spans. For example, given a sentence "It didn't work," our application will extract the negation expression span "didn't." If there are no negation spans, our application returns no answers. We define negation expressions as any expression that states that something is not true or is not the case, through negative words, phrases, or clauses.

4.1 Problem formulation

We treat this task as an extractive question answering task – given a question and a context, extract the answer span. In our context, we simply use the word "negation." as the question for all training samples. In an extractive question answering task, the question and context concatenated using special tokens (Figure 29). They are tokenized and fed into the model as a sequence of indices (Figure 30). The model is then trained to predict the start and end positions of the negation span (Figure 30).

```
"<s>negation</s></s>Weren't we going to the moon?</s>"
```

Figure 29: Untokenized input

```
[0, 23156, 1258, 2, 2, 771, 8663, 75, 52, 164, 7, 5, 6950, 116, 2]
```

Figure 30: Tokenized input with start position (yellow) and end position (green) labelled

4.2 Model

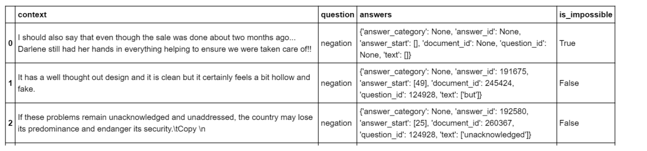
Our application uses the Transformers model architecture, more specifically the RoBERTa model architecture though the transformers library. Due to limited data, we further train a fine-tuned question answering model "deepset/roberta-base-squad2" on our own custom dataset.

4.3 Dataset

We manually annotate the first occurring negation expression span using the Haystack Annotation Tool (Table 5) – this means that our model is trained to return only the first occurring negation span. We then tokenized and labelled the start and end positions of the negation spans (Table 6). For sentences with no negation expression, the start token is annotated as the start and end positions (Table 6).

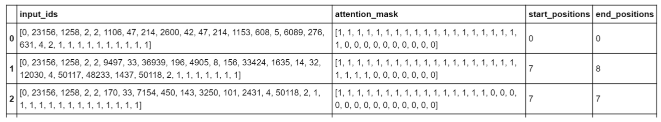
We manually annotated 600 samples, where each sample may or may not contain negation span. We split our dataset into a training set of 480 samples and a validation set of 120 samples.

Table 5: Raw manually annotated samples



	context	question	answers	is_impossible
0	I should also say that even though the sale was done about two months ago... Darlene still had her hands in everything helping to ensure we were taken care of!!	negation	{'answer_category': None, 'answer_id': None, 'answer_start': [], 'document_id': None, 'question_id': None, 'text': []}	True
1	It has a well thought out design and it is clean but it certainly feels a bit hollow and fake.	negation	{'answer_category': None, 'answer_id': 191675, 'answer_start': [49], 'document_id': 245424, 'question_id': 124928, 'text': ['but']}	False
2	If these problems remain unacknowledged and unaddressed, the country may lose its predominance and endanger its security.\tCopy \n	negation	{'answer_category': None, 'answer_id': 192580, 'answer_start': [25], 'document_id': 260367, 'question_id': 124928, 'text': ['unacknowledged']}	False

Table 6: Tokenized and labelled samples for model input

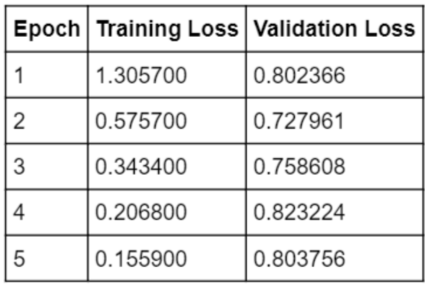


	input_ids	attention_mask	start_positions	end_positions
0	[0, 23156, 1258, 2, 2, 1106, 47, 214, 2600, 42, 47, 214, 1153, 608, 5, 6089, 276, 631, 4, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]	0	0
1	[0, 23156, 1258, 2, 2, 9497, 33, 36939, 196, 4905, 8, 156, 33424, 1635, 14, 32, 12030, 4, 50117, 48233, 1437, 50118, 2, 1, 1, 1, 1, 1, 1, 1]	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]	7	8
2	[0, 23156, 1258, 2, 2, 170, 33, 7154, 450, 143, 3250, 101, 2431, 4, 50118, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]	[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]	7	7

4.4 Training

We further train “deepset/roberta-base-squad2" for 5 epochs on the training set and evaluated the model on the validation set at the end of each epoch (Table 7). The model with the lowest validation loss, epoch 2 model, was chosen as the final model for our application.

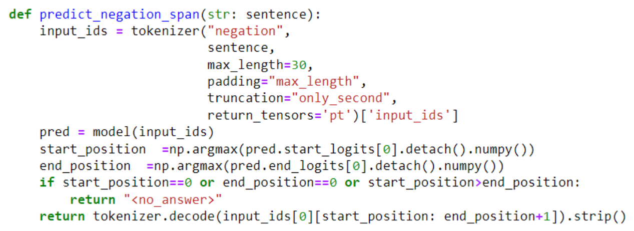
Table 7: Training and validation loss



Epoch	Training Loss	Validation Loss
1	1.305700	0.802366
2	0.575700	0.727961
3	0.343400	0.758608
4	0.206800	0.823224
5	0.155900	0.803756

4.5 Application - Inference time

Our application uses a stream-lit interface. Upon start-up, the model and tokenizer are instantiated once using the *transformers* library. Thereafter, user’s input will call the *predict_negation_span* function (Figure 31), which will return the most probable negation span. If the start and end positions are invalid or zero, “<no_answer>” is returned. Our application is also hosted online at the Hugging-face model repository (refer to Readme.txt).



```
def predict_negation_span(str: sentence):
    input_ids = tokenizer("negation",
                          sentence,
                          max_length=30,
                          padding="max_length",
                          truncation="only_second",
                          return_tensors='pt')['input_ids']
    pred = model(input_ids)
    start_position  =np.argmax(pred.start_logits[0].detach().numpy())
    end_position  =np.argmax(pred.end_logits[0].detach().numpy())
    if start_position==0 or end_position==0 or start_position>end_position:
        return "<no_answer>"
    return tokenizer.decode(input_ids[0][start_position: end_position+1]).strip()
```

Figure 31: Function that makes prediction at inference time

5 Conclusion

This assignment enabled us to apply and practice the basics of NLP taught during lectures, such as tokenization, POS tagging and stemming. We learnt to use many useful NLP toolkits like NLTK library and transformers library. Furthermore, it challenged us to explore and research NLP topics outside the curriculum, such as TF-IDF and question answering. Overall, it was a fulfilling project to undertake.

6 Contributions

Name	Contributions
Tan Ching Fhen	Part 2.1, 2.2 and part 4, report
Wei Yao	Part 2.4, report
Ta Quynh Nga	Part 2.3.5, report
Cui Chenling	Part 2.3.1 – 2.3.4, report
Luo Wenyu	Part 3, report

ACKNOWLEDGMENTS

This project would not be possible without the guidance from our professor, Dr Sun Aixin, and we also thank our group mates for putting in all the required efforts.

REFERENCES

[1] Tanwar, R., 2021. How To Use BERT Transformer For Grammar Checking?. [online] Analytics India Magazine. Available at: <https://analyticsindiamag.com/how-to-use-bert-transformer-for-grammar-checking/> [Accessed 25 October 2021].