CE/CZ4045 Assignment Part 1

Lim Jun Hong
Nanyang Technological University
LIMJ0209@e.ntu.edu.sg

Tammy
Nanyang Technological University
tlim045@e.ntu.edu.sg

Delphine
Nanyang Technological University
@e.ntu.edu.sg

Pang Yu Shao
Nanyang Technological University
C170134@e.ntu.edu.sg

ABSTRACT

TODO:

KEYWORDS

NLP, Text Tagging, Information Retreival

1 INTRODUCTION

TODO:

2 DOMAIN SPECIFIC DATASET ANALYSIS

The 3 datasets chosen for analysis are from different domain specific Reddit forums:

- `Investments`: r/wallstreetbets
- `Mechanical Keyboards`: r/mechanicalkeyboards
- `Programming`: r/programming

2.1 Tokenization

The NLTK function *word_tokenize()* was used to tokenize each comment. This function splits tokens based on white space and punctuation. Figure 1 shows the function being implemented in our code and Figure 2 shows the resulting tokens after running the tokenizer function on a single comment.

```
w = nltk.word_tokenize(comment.body)
wsb_words.extend(w)
count+=1
print("Comment:", comment.body)
```

Figure 1: Implementation of tokenizer

```
Comment: Tesla calls!
Token results: ['Tesla', 'calls', '!']
```

Figure 2: Result of tokenizer

2.2 Evaluation of tokenizer results

The tokenizer was successful in tokenizing most data in the chosen dataset. However, upon manual inspection of each tokenized result, we found some incorrect tokens as shown in Table 1. The incorrect tokens were mainly due to domain specific issues. For instance, in r/programming, the tokenizer incorrectly tokenized a line of code and separated the code into three tokens instead of one. Another common error is due to the separation of domain specific terminologies such as investment terms like "options flow" and names of mechanical switches like "Holy Pandas".

Expected Token	Actual token
"Martin Luther King"	"Martin", "Luther", "King"
"Options flow"	"Options","flow"
"Holy Pandas"	"Holy", "Pandas"
":)"	":", ")"
"System.nanoTime()"	"System.nanoTime", "(",")"
"String Buffer:", "1 ms"	"String Buffer:1", "ms"

Table 1: Incorrectly identified tokens

2.3 Methods to improve tokenizer

Based on the results shown in the table above, the tokenizer could be further improved by implementing the following methods.

Context specific tokens The dataset used in this assignment is from an online forum, Reddit. As such, many informal tokens such as emojis are used but the current tokenizer is unable to tokenize it correctly. For example, ":)" will be tokenized into 2 tokens, ":" and ")" which loses its meaning. Tokenizers such as TweetTokenizer are able to recognize such tokens.

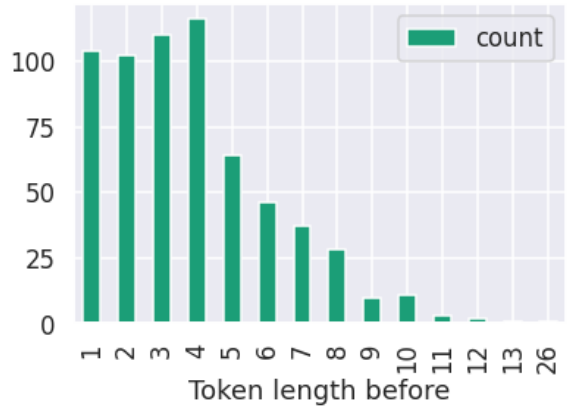
Multi-word expressions: To prevent the tokenizer from splitting multi-word expressions, a function such as the NLTK MWETokenizer could be implemented. This function takes a string that has been divided into tokens and retokenize it by merging into a multi-word expression. For example, the tokens " Martin" ,"Luther", "King" will be retokenized into a single token, "Martin Luther King".

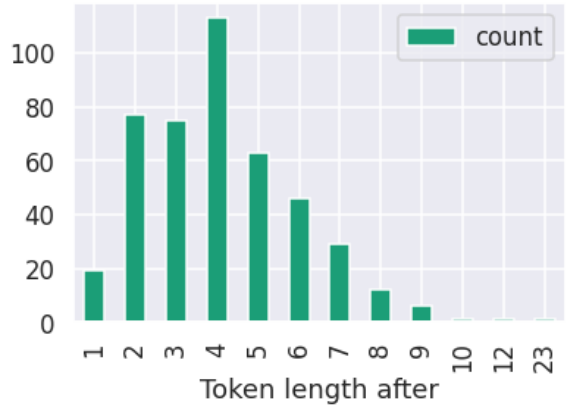
Regular expressions: Regular expressions could also be used to ensure sequences such as money expressions and percentage expressions are tokenized as a singular token. For example, currently word_tokenizer tokenizes the word "60%" to 2 separate tokens, "60" and "%" although it is more meaningful to have it as a single token which could be done by regular expression.

2.4 Stemming

After getting the word tokens for each comment, the NLTK function porterStemmer() is used to reduce each word token to its root or base by removing common morphological and inflexional token endings.

To ensure that the data used for plotting of token distribution would be meaningful, we added a list of stop words which are words that do not contribute to a deeper meaning for the analysis. Such words include common punctuations such as commas and fullstops.





The distribution graphs above shows that most tokens have 4 number of characters. Prior to stemming, the upper range for length is from 10 to 26 number of characters. After stemming, this upper range significantly decreased to 7 to 9 number of characters. This is due to the process of stemming reducing common token endings, which thus translates into a lower number of characters.

2.5 Sentence Segmentation

The NLTK function *sent_tokenize()* uses an instance of *PunktSentenceTokenizer* from the *nltk.tokenize.punkt* module which has already been trained and knows very well where to mark the end and beginning of a sentence at what characters and punctuation.

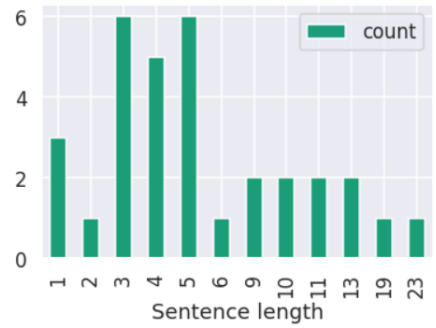


Figure 1: Wallstreetbets

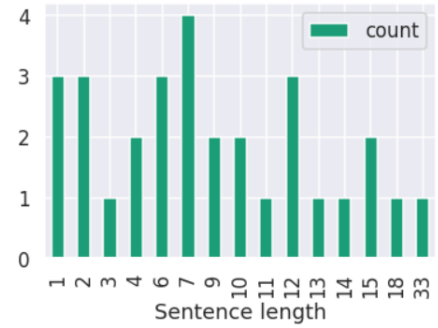


Figure 2: Mechanicalkeyboards

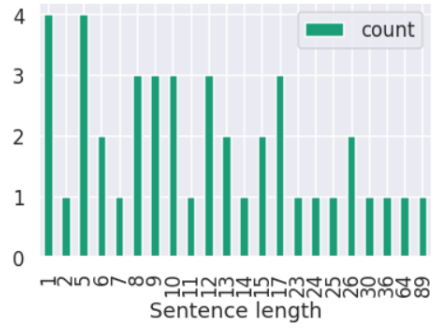


Figure 3: Programming

From the graph of 20 sentences being sampled from each domain, we can see the difference in the sentence length between the different domains. 'Wallstreetbets' is a community mainly focused on the U.S. stock market, 'mechanicalkeyboards' is a community where people would post images about their own mechanical keyboards or seek for recommendations of parts or mechanical keyboards to purchase and lastly, 'programming' is a community where programmers would be there to discuss about programming related question or the uses of different languages.

For 'wallstreetbets', the majority of the sentences are about 3-5 words long which is common in Reddit where people just leave comments about what they think. The comments usually reflect the reaction of a news article posted about the U.S. stock market thus the sentences aren't that long. As compared to 'mechanicalkeyboards', the sentences are much longer, where people would be curious about the parts used in the mechanical keyboard posted or

queries from commenters about the experience with the mechanical keyboard. 'Programming' in reddit has the widest range of sentence length in answer because of the nature of the subreddit. With more questions and elaborated answers, we can tell from Figure 2.3 that there are very long sentences as compared to 'wallstreetbets' and 'mechanicalkeyboards' causing a wide spectrum in length of the sentences.

2.6 Part-Of-Speech Tagging(POS Tagging)

The NLTK function *nltk.pos_tag()* takes a string and tags the words in the string to their respective POS tags. *nltk.pos_tag()* is a machine-learning tagger that has been well-trained by Microsoft and the model is saved for people to use. Since there are no perfect taggers around, there were definitely some errors made by the tagger especially when it comes to domain-specific terms.

For 'wallstreetbets', from one of the randomly selected comments, 'hexo' was tagged as NN instead of NNP, a proper noun when it is a name for a U.S. stock, Hexo Corp. 'gme' was incorrectly tagged as NN instead of a proper noun because 'gme' meant GameStop Corp. in the context of 'wallstreetbets'.

For 'mechanicalkeyboards', there are words that are incorrectly tagged too. There are many different keyboard switches that are named by their brands. One of the comments had 'alpaca/', 'tangerine' and 'switches' tagged as adjective, noun, and plural nouns respectively when these 3 words are supposed to be tagged as a proper noun. Domain-specific words are not identified correctly due to the limitation of the NLTK library.

For 'programming', there are still technical roles in the infocomm technology that are tagged incorrectly. 'System admins' are tagged separately by NLTK when it is a role in the office. Some commonly used programming terms such as 'print' and 'i-o' also known as input output are technical terms used in the programming syntax but are tagged as adjectives by NLTK.

From the results of 3 different domains, we can see that NLTK POS tagging system is not perfect, however there is no perfect tagging system for all the domains. The results are as expected even though NLTK's machine learning model is often being updated by the open-source community. If the POS Tagging system is not trained with the correct corpus or edited by the programmers, domain specific terms would highly likely be identified incorrectly.

3 APPLICATION: SEARCH ENGINE

Based on the reviews collected in the previous section, A simple search engine is developed which allows the user to search for related reviews based on the user's input.

With the user's input (i.e., the search query), the list of relevant reviews is retrieved and the reviews are ranked according to their similarity, which is presented in descending order to the user.

3.1 Implementation

The application is implemented using `Python 3.7.9` with the following external libraries:

- `pandas`: For general DataFrame processing
- `scikit-learn`: For calculating tf-idf / cosine similarity

3.2 TF-IDF Vectorization of Reviews

For implementing the search engine, the reviews must first be represented by `tf-idf` vectors instead of plain-text. This will allow the computation of the similarity between the document/review vector and the query vector.

To generate the `tf-idf` vectors, every review is first preprocessed by case-folding, removing stopwords and tokenized to generate a **Bag of Words** representation. The term counts are used to generate the `tf-idf` values.

- **Term Frequency** (`tf`), is a measure of how often a term occurs in the document.
- **Inverse Document Frequency** (`idf`), is a measure of how much information the term provides (i.e., how rare the term is) across **all documents**.

The `scikit-learn` library provides a module,

`sklearn.feature_extraction.text.TfidfVectorizer`

which automatically performs basic preprocessing of the text mentioned above and transforms the array of documents into a tf-idf vector while also performing some normalizing and smoothing to the data.

3.3 Querying and Ranking Reviews

A Search Engine is required to take in a user's query and display *relevant* documents to the user. Since tf-idf vectors have already been built for the existing documents (reviews), the user's query can be transformed to a tf-idf vector as well and the similarity can be computed based on some similarity measure.

`TfidfVectorizer` also provides a method `transform`, which builds a tf-idf vector from a document based on the Language Model that it has learned. With both the query and documents represented in `tf-idf` vectors, the similarity between the query vector and all document vectors can be computed. A few distance measures that can be used include:

- Manhattan Distance
- Euclidean Distance
- Cosine Similarity

For our search application, the **Cosine Similarity** measure is used. Search queries have a short length in nature, therefore using Manhattan / Euclidean distance could cause a potential document of interest have a large distance (i.e., appear dissimilar) when compared to the query. For instance, consider the example vocabulary in Table 2 and the following documents and query:

- Document 1: apple apple apple apple apple apple
- Document 2: banana orange
- Query: apple

These would yield the following term count vectors:

$$D1 = \begin{pmatrix} 6 & 0 & 0 \end{pmatrix}$$

$$D2 = \begin{pmatrix} 0 & 1 & 1 \end{pmatrix}$$

$$Q = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$$

By calculating the euclidean distance between the query and the documents, the following Euclidean distances are obtained:

$$Euclidean(D1, Q) = 5$$

$$Euclidean(D2, Q) = 1.732$$

Table 2: Example Vocabulary

Word Id	Word
0	apple
1	banana
2	orange

Here, the distance between D2 and Q is smaller than that of the distance between D1 and Q, however it can be observed by D1 should be the more similar document. Therefore, it can be concluded that measures such as Manhattan and Euclidean distances can be affected by the magnitude of the vector, even when both the query and the document vectors share similar components.

The Cosine Similarity overcomes this as it is a measure of the *angle* between two vectors. As search queries are short in nature, it is able to retrieve documents which share similar components to the query vector.

3.4 Examples

```
Reviews 1 to 5 (out of 167) for search query
"Fountain light show"
=============================================

1: The Jewel is the gigantic shopping mall with varieties
of shops and restaurants. The famous one is the fountain
that shall make you stop to see it.
At night time there are the light show at Fountain.

Recommend for visiting

2: This has to be seen to be believed. Gardens, waterfalls,
restaurants, shops, adventure park and monorail. Allow at
least a few hours here and stay for light show. Light show
not great but colored waterfall is impressive. A must see.

. . .
```