Online Forum (Stack Overflow) Data Processing

CZ 4045 Natural Language Processing Assignment, AY 17/18 Semester 1

Lakhotia Suyash
U1423096J

Venkatesh Nikhil
U1423078G

Jaiswal Shantanu
U1423003C

Jacob Sunny
U1421414L

Kamath Shantanu Arun
U1422577F

ABSTRACT

This report documents our approach and the results of basic natural language processing (NLP) operations such as tokenization, Part-of-Speech (POS) tagging and stemming on content posted on the online programming Q&A forum, Stack Overflow. The dataset we used was obtained from a data dump made publicly available by Stack Exchange. Preliminary analysis such as stemming and POS tagging was applied using an off-the-shelf NLP library for Python. Furthermore, a custom tokenizer was built using a rule-based approach with regular expressions and the performance of this tokenizer was analyzed with respect to manually annotated tokens. Lastly, an application was developed that finds questions of high similarity, which may be used to avoid duplicate threads on an online forum like Stack Overflow.

CCS CONCEPTS

• **Artificial intelligence → Natural language processing**

KEYWORDS

Natural Language Processing, POS Tagging, Stemming, Tokenization, Similarity Detection, Stack Overflow

1 INTRODUCTION

The objective of this project was to explore the field of natural language processing (NLP) using content posted on the online programming Q&A forum, Stack Overflow.

The first component of the end-to-end system extracted a subset of the posts on Stack Overflow that were related to the Java programming language, sanitized these posts and created the dataset. This dataset collection process is described in detail along with statistics of the extracted dataset in Section 2.

Natural language processing techniques like tokenization, stemming & Part-of-Speech (POS) tagging was performed on this extracted dataset using an off-the-shelf open-source NLP library for Python, NLTK [1], to explore the dataset further and understand certain drawbacks of using off-the-shelf tools without fine-tuning for the type of corpus and task at hand. In this case, because the dataset contains a significant amount of programming terminology & Internet-speak, a generic tool like NLTK tends to incorrectly process the dataset. This is further elaborated on in Section 3.

Using the results of processing the dataset using NLTK, we built a custom tokenizer that was fine-tuned to our dataset. This tokenizer uses several regular expressions to correctly tokenize the dataset according to our token definition in Section 3.3. The new tokens are then fed to a POS tagger to verify if a fine-tuned tokenizer works better for irregular tokens that may not be present in a generic corpus. This is explored in detail in Sections 4 & 5.

Lastly, we built an application that detects the similarity between two questions in order to avoid duplicating a question that has already been asked. This is further explained and demonstrated in Section 6.

2 DATASET COLLECTION

The dataset for the task at hand was collected from a data dump of Stack Overflow posts made publicly available by Stack Exchange, Stack Overflow's parent organization (`stackoverflow.com-Posts.7z` from [2]). For the purposes of this report, a post is either a question or an answer on the online forum and a thread is made up of one question and one or more answers.

A total of 500 threads tagged with the programming language 'Java' were retrieved from the data dump, with a total of 2,483 posts (i.e. 500 questions + 1,983 answers). As can be seen from Figure 1 (next page), each extracted thread has at least two posts i.e. at least one answer and most threads (58.4%) have between 2 - 4 posts each. For the rest of this report (except Section 6), question posts and answer posts will be treated the same as the context of the post no longer holds any significance.

After extracting the dataset, the actual text of the posts was extracted and cleaned for further processing. This was done by (1) selecting the post body, (2) removing any multi-line code snippets, (3) deleting hyperlinks (but retaining the text), (4) clearing any HTML tags (except "<code>" & "</code>" to separate inline code blocks within paragraphs) and (5) converting HTML entities to their Unicode form (e.g. "&" to "&"). Furthermore, any redundantly repeated newlines were removed.

After sanitizing the post bodies, the inline code blocks were temporarily removed and the bodies were tokenized using an off-the-shelf word tokenizer included in NLTK. As can be seen in Figure 2, most of the posts in our dataset are lesser than 200 words long.

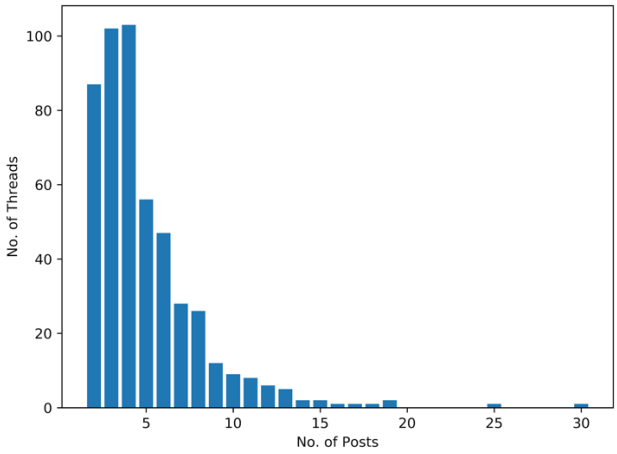

Figure 1: Distribution of No. of Posts in Threads

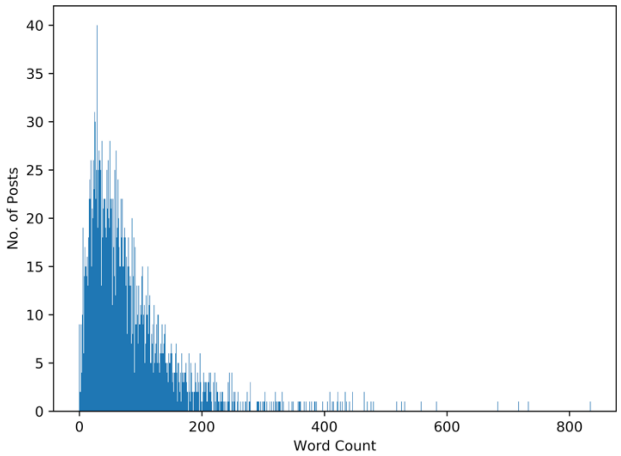

Figure 2: Distribution of Post Word Count

3 DATASET ANALYSIS

Once the dataset was extracted, basic preliminary analysis was performed using NLTK. Before running the stemmer and POS tagger, sentence and word tokenization was performed using the built-in tokenizers provided by NLTK. However, in order to better tokenize the posts due to the presence of non-English terms given the subject matter, we defined a new set of tokens and established a ground truth by manually annotating 100 posts, which is used to evaluate a custom tokenizer (discussed in Section 4).

3.1 Stemming

Stemming is performed on the data using the Porter stemming algorithm, whose implementation is provided as a built-in function in NLTK. The list of stop words is defined in `data_analysis/stop_words.txt` and are excluded from the analysis of the resulting stems. An incomplete code stub for stemming is shown in Source Code 1.

```
porter_stemmer = nltk.stem.porter.PorterStemmer()
for post in posts:
  for paragraph in post.split('\n'):
    paragraph = paragraph.lower()
    tokens = nltk.word_tokenize(paragraph)
    for token in tokens:
      if token == '' or token in stop_words:
        continue
      stem = porter_stemmer.stem(token)
```

Source Code 1: Incomplete Code Stub for Stemming

The top seven most frequent words before stemming were “java”, “use”, “code”, “like”, “using”, “would” & “way”. A list of the 20 most frequent words before stemming are listed in `data_analysis/frequent_words.txt`.

After stemming, the top seven most frequent stems were “use” (from “used”, “using”), “java” (from “java”), “code” (from “code”, “coding”), “like” (from “like”, “liking”, “liked”, “likes”, “likely”), “would” (from “would”), “class” (from “class”, “classes”) & “method” (from “method”, “methods”). As can be seen, the most frequent stems provide a richer insight into the dataset as opposed to the most frequent words. A list of the 20 most frequent stems and their original words can be found in `data_analysis/frequent_stems.txt`.

3.2 POS Tagging

Using NLTK’s `pos_tag()` function, POS tagging was performed on 10 sentences from the dataset. An incomplete code stub for POS tagging is shown in Source Code 2.

```
for post in posts:
  for paragraph in post.split('\n'):
    sentences += nltk.tokenize.
                 sent_tokenize(paragraph)

for sentence in sentences:
  words = nltk.word_tokenize(sentence)
  pos_tags[sentence] = nltk.pos_tag(words)
```

Source Code 2: Incomplete Code Stub for POS Tagging

The detailed results of the tagging process can be viewed in `data_analysis/sentences_pos_tags.txt`. An example result is presented on the next page.

```
"Converting a CSV file to XML doesn't add any value." =>
[('Converting', 'VBG'), ('a', 'DT'), ('CSV', 'NNP'), ('file', 'NN'), ('to', 'TO'), ('XML', 'NNP'), ('does', 'VBZ'), ("n't", 'RB'), ('add', 'VB'), ('any', 'DT'), ('value', 'NN'), ('.', '.')]
```

While the POS tagger manages to tag the tokens in the sentences correctly most of the time, there are a few exceptions where the POS tagger is unable to identify the correct tags. For example, in "...and then serializing to XML does make sense…", the POS tagger tags "XML" as a 'VB' or Verb, when it should be a 'NNP' or Proper Noun. This is probably because the NLTK POS tagger has not been trained on a corpus that features programming terminology.

3.3 Token Definition & Annotation

Upon further inspection of the tokenization provided by the off-the-shelf tool, we notice a few repeated errors. For example, the programming language "C++" is always divided into three separate tokens, one for each character. This highlights the need for a custom tokenizer that can take into account some aspects of the subject matter at hand.

After analyzing the contents of the Stack Overflow posts, we defined the following token types to build our custom tokenizer:

- Each non-alphanumeric character is a separate token, except in the cases listed below where they are considered part of a larger token:
  - Ellipsis ("..."), Arrow ("->")
  - Clitics ("'ll", "'ve", etc.)
  - Text Emojis (";)", ":P", etc.)
  - Programming Languages ("C#", "C++", ".NET", etc.)
  - Abbreviations ("e.g.", "i.e.", "etc.", etc.)
  - Decimal Numbers
  - URLs ("java.net", "http://asm.objectweb.org/", etc.)
  - Package Names ("com.foo.bar.class", etc.)
  - File Paths ("com/foo/Bar.java", etc.)
- An inline code block is a single separate token (from "<code>" to "</code>").
- A numerical value is a separate token.
- Clitics are separate tokens from the adjoining word.
- A 'word' (defined by a chain of strictly alphabetical characters) is a separate token.

In order to establish a ground truth to use as a benchmark for our custom tokenizer, we annotated the first 100 posts in our dataset in a semi-manual approach.

First, we wrote a rudimentary regular expression based tokenizer that split the 100 posts into semi-correct 'tokens'. This tokenization was then manually corrected to match our token definition precisely (barring overlooked human errors). The ground truth token annotations can be found in `tokenization/annotation/ground_truth.txt`.

4 TOKENIZATION

We developed a rule-based tokenizer to tokenize the dataset based on our token definitions described in Section 3.3. The tokenizer goes through the following rules (implemented using regular expressions) in order:

1. Inline code blocks are extracted as single tokens by identifying blocks surrounded by "<code>" & "</code>" HTML tags.

```
r'(<code>.*?<\/code>)'
```

2. Space tokenization is performed by splitting the sentences based on newlines, spaces and other space characters.

```
r'\s+'
```

3. A dictionary of commonly used text emoticons and programming languages / frameworks is used to filter out the respective tokens.

```
mark_tokens(space_tokenized, lambda token: token.lower()
            in emoticons or token.lower() in langs or
            code_tokenizer.match(token) is not None)
```

4. Period tokenization is performed by splitting words directly followed by ".", whilst handling edge cases of abbreviations and decimal numbers.

```
r'(.*\.[\s]*$)+'  # period tokenization
r'([a-zA-z]\.([a-zA-z]\.)+)'  # abbreviations
r'(([+|-]?\d+)?\.((\d+)|[\w]))'  # decimal numbers
```

5. Clitics are separated from their adjoining words.

```
r'(\S+\'\S+)'
```

6. Ellipses and parentheses are matched and tokenized.

```
r'(\.\.\.|->|[(){}\[\],])'
```

7. Decimal numbers and version numbers (e.g. "1.x") are matched and tokenized.

```
r'(([+|-]?\d+)?\.((\d+)|[\w]))'
```

8. Internet URLs are matched and tokenized.

```
r'(https?:\/\/(?:www\.|(?!www))[a-zA-Z0-9][a-zA-Z0-9-
]+[a-zA-Z0-9]\.[^\s]{2,}|www\.[a-zA-Z0-9][a-zA-Z0-9-
```

```
]+[a-zA-Z0-9]\.[^\s]{2,}|https?:\/\/(?:www\.|(?!www))
[a-zA-Z0-9]\.[^\s]{2,}|www\.[a-zA-Z0-9]\.[^\s]{2,})'
```

9. Package names are matched and tokenized.

```
r'([A-Za-z][A-Za-z0-9_]*(\.[a-z0-9_]+)+[0-9a-z_A-Z])'
```

10. Filepaths are matched and tokenized.

```
r'((.*)?(\/[^/\n ]*)+\/?\n?)'
```

11. An additional rule is used to recognize words separated by "/" to imply an "or" as opposed to a filepath. For example, "you can use a getter/setter to solve this".

```
r'(^[^\/]+[\/][^\/]+$)'
```

4.1 Tokenizer Performance

The custom tokenizer was run on the same 100 posts used to establish the ground truth values and its performance was calculated by running a diff tool on the output against the ground truth. The observed metrics are tabulated in Figure 3 and further calculations are performed on the next column.

Token Count	9892
True Positives	8711
True Negatives	1176
False Positives	3
False Negatives	2

Figure 3: Performance Metrics of Custom Tokenizer

We define **True Positives** to be cases where characters were expected to be split into tokens and actually did. For example:

1. [should, n't]
2. [I, 'll]

We define **True Negatives** to be cases where characters were not expected to be split into tokens and actually didn't. For example:

1. [<code>XML</code>]
2. [C#], [C++]
3. [3.0]
4. [http://ebay.custhelp.com/cgi-bin/ebay.cfg/php/enduser/std_adp.php?p_faqid=1230]

To differentiate between true positives and true negatives, we kept track of both counts during the tokenization process by using a marker string ("__FT__"). The implementation of the tokenizer `custom_tokenizer()` can be found in the Python script `tokenization/custom_tokenizer/tokenizer.py`.

From the observations in Figure 3, we can derive the F1 score of our tokenizer:

$$\textbf{Precision} = \frac{TP}{TP + FP} = \frac{8711}{8714} = 0.9997 = \mathbf{99.97\%}$$

$$\textbf{Recall} = \frac{TP}{TP + FN} = \frac{8711}{8713} = 0.9998 = \mathbf{99.98\%}$$

$$\textbf{F1} = \frac{2 * (\text{Recall} * \text{Precision})}{\text{Recall} + \text{Precision}} = 0.9998 = \mathbf{99.98\%}$$

As can be seen from our observations as well as the derived metrics above, our custom tokenizer works extremely well and is very robust with a negligible error rate. However, in order to better understand the 6 errors, we took a closer look at what they were.

We define **False Positives** to be cases where characters were not expected to be split into tokens and actually split. All of the false positives dealt with text emoticons being directly next to other characters. For example, ";))" was tokenized to [";", ")", ")"] instead of [";)", ")"].

We define **False Negatives** to be cases where characters were expected to be split into tokens and actually didn't split. Both cases of false negatives dealt with more than two words concatenated with "/" to imply "or" being matched as filepaths instead. For example, "caching/db/etc" is tokenized to ["caching/db/etc"] instead of ["caching", "/", "db", "/", "etc"].

All of the above error cases are unavoidable if the tokenizer is to be defined by hard-coded deterministic rules. Perhaps, a statistical approach trained on a large corpus in a future version will reduce the number of edge cases that result in errors.

5 FURTHER ANALYSIS

5.1 Irregular Tokens

During the creation of our tokenizer, we observed that there are several non-English words (i.e. irregular tokens) that exist in our dataset due to the subject matter. In order to recognize these irregular tokens, we used the English vocabulary available in the PyEnchant library for Python [3].

After running our script, we see that the top few irregular tokens are "java", "API", "JVM", "C#" & "app". The first three are obvious due to the fact that all the posts in our dataset are about the Java programming language. However, "C#" being mentioned so many times is interesting because it may imply that these two

languages are either constantly being compared to each other or code is constantly being ported from one language to the other. The complete list of most frequent irregular tokens can be found in `tokenization/custom_tokenizer/irregular_tokens_stats.txt`.

5.2 POS Tagging (w/ Irregular Tokens)

Upon running the NLTK POS tagger on 10 sentences with irregular tokens tokenized by our custom tokenizer, we notice that most tokens are tagged correctly, similar to Section 3.2. For example, in the sentence below, the two irregular tokens, "CSV" & "XML", are correctly tagged as 'NNP' or Proper Nouns.

"Converting a CSV file to XML doesn't add any value." =>
```
[('Converting', 'VBG'), ('a', 'DT'), ('CSV', 'NNP'), ('file', 'NN'), ('to', 'TO'), ('XML', 'NNP'), ('does', 'VBZ'), ("n't", 'RB'), ('add', 'VB'), ('any', 'DT'), ('value', 'NN'), ('.', '.')]
```

When compared to the results obtained in Section 3.2, similar irregularities exist in the results obtained since we use the same off-the-shelf tool to actually perform the POS tagging. For example, in the case below, the token "XML" is still marked as a verb.

"…serializing to XML does make sense…" =>
```
[...('serializing', 'VBG'), ('to', 'TO'), ('XML', 'VB'), ('does', 'VBZ'), ('make', 'VB'), ('sense', 'NN')...]
```

This time around, however, the POS tagger performs poorly with hyphenated words. Since our custom tokenizer is defined to split hyphenated words, we observe poor results from the POS tagger in such cases (example below).

"ready-made" =>
`[('ready-made', 'JJ')]` vs.
```
[('ready', 'JJ'), ('-', ':'), ('made', 'VBN')]
```

By going through different iterations of POS tagging with different sets of sentences, we came to the conclusion that the tags for the irregular tokens would have to be learned by a custom POS tagger in order to achieve the best results. For example, ";)" will need to be tagged as 'EMO' for emoticon. It is not enough to simply provide the right tokens using our custom tokenizer if the POS tagger does not have enough information about what these tokens mean.

6 APPLICATION: Detecting Question Similarity & Finding Duplicates

Discussion forums which serve a large user base, such as Stack Overflow, are often susceptible to users submitting new questions that may have already been previously answered. Given the multitude of questions submitted every minute on Stack Overflow, it would be of great convenience if a user, prior to submitting a new question, is first routed to previous threads that resolve potentially similar questions.

Hence, we developed an application that allows the user to input his/her question, and receive existing potential question duplicates in response.

To detect duplicate questions for a given question, we use a weighted ensemble for both syntactic and semantic language features of the provided questions. More specifically, we use a weighted combination of the following:

1. **WordNet Synonym Sets Distance**: Using NLTK, we access Princeton's WordNet corpus and compute synonym sets for each of the terms in the given two questions. We then compute the average maximum distance for each synonym set of Question 1 against Question 2.
2. **Word Vectors Cosine Similarity:** We calculate the average of word vectors for each question and output the cosine similarity between resultant averaged vectors for given questions. We obtain Stack Exchange specific word vectors trained using Word2Vec – a neural network language modelling technique.
3. **Word Mover's Distance:** Using the Gensim library for Python, we calculate the word mover's distance between individual word vectors of two sentences. In essence, we try to calculate the distance between two questions, by attempting to recreate Question 1 through Question 2 using word vectors.

6.1 Demonstration

A sample run of the application is reproduced below:

For example, given:

Q: "Why is it so difficult in Java when any kind of module system should be able to load jars dynamically. I'm aware that one can write one's own <code>ClassLoader</code> module. However, that is a lot of work for something as easy as calling a method with a jar file as its argument. Are there any alternative suggestions?"

Our application outputs the following similar questions from the existing corpus:

Q_ID_409: "*Why is it so hard to do this in Java? If you want to have any kind of module system you need to be able to load jars dynamically. I'm told there's a way of doing it by writing your own <code>ClassLoader</code>, but that's a lot of work for something that should (in my mind at least) be as easy as calling a method with a jar file as its argument.*"

Similarity Score: 0.952; Potential Duplicate

Q_ID_185: "*For various reasons calling <code>System.exit </code> is frowned upon when writing Java Applications, so how can I notify the calling process that not everything is going according to plan?"*

Similarity Score: 0.75

As can be seen in the example above, our application is able to capture the following:

1. Semantic similarity between phrases such as "I'm aware that one can do it" and "I'm told there is a way of doing it".
2. Usage of keywords such as "ClassLoader", "jar file", etc.
3. Similar contexts of words such as "hard", "frowned upon", "difficult" etc.
4. Syntactic structure of the questions.

6.2 Application Dependencies

In developing the application, we used our custom tokenizer and NLTK's POS tagger to tokenize and tag questions. We also used the NLTK library to obtain WordNet synonym sets (synsets) [4] by converting POS tags into synset tags and computed path similarity between these word synsets.

To obtain Stack Exchange relevant word vectors (obtained from neural network language models), we used the AskUbuntu question dataset [5].

To calculate the word mover's distance, we used the Gensim library for Python [6] with preloaded word vectors obtained from the AskUbuntu dataset mentioned above.

7 CONCLUSIONS

Through the course of building this rudimentary end-to-end NLP system, we realized that while off-the-shelf toolkits may be extremely powerful for generic English-language text, it is often much better to build one's own tokenizer (or perhaps, POS tagger too) from scratch or on top of existing tools in order to better suit the corpus that is being processed and the task at hand.

Additionally, using a statistical approach to build the custom tokenizer by training it like a machine learning model may help improve its robustness and coverage.

8 FUTURE WORK

Our custom tokenizer, catered to programming language forums, works well in most of our test cases. However, since the custom tokenizer is rule based, as we add more data to the corpus it may be difficult to detect and satisfy all corner cases using just rules. Moreover, our tokenizer is designed with the assumption that the dataset only contains posts with the tag 'Java'. If we expand the corpus to include other tags, a statistical approach in designing the tokenizer is more practical and will be more generalized and extensible.

Another useful component to add would be a custom POS tagger. As we have remarked multiple times in the report, using an off-the-shelf POS tagger with a custom tokenizer does not work very well. Building a custom POS tagger that is capable of handling different token definitions will prove to be very useful in such use cases to handle irregular token types.

Ultimately, a very useful NLP library can be created to allow users to customize and fine-tune standard NLP tasks such as tokenization, stemming and POS tagging. This would allow for more standardized schemes in the research and development of NLP projects.

Acknowledgements

We wish to express our sincere gratitude towards our supervisor Associate Professor Sun Aixin for his support and guidance provided through the course of this project.

We wish to thank the School of Computer Science and Engineering, Nanyang Technological University for providing us the opportunity to work on this project.

CONTRIBUTIONS

Lakhotia Suyash: Code for Sections 2 - 3, Report for all Sections
Venkatesh Nikhil: Code for Section 3, Report for all Sections
Jaiswal Shantanu: Code & Report for Section 6
Jacob Sunny: Code & Report for Section 4 & 5
Kamath Shantanu Arun: Code & Report for Section 4 & 5

REFERENCES

[1] NLTK Python Library, http://www.nltk.org/
[2] Stack Overflow Data Dump, https://archive.org/details/stackexchange
[3] PyEnchant Python Library, http://pythonhosted.org/pyenchant/
[4] NLTK WordNet, http://www.nltk.org/howto/wordnet.html
[5] AskUbuntu Dataset, https://github.com/taolei87/askubuntu
[6] Gensim Python Library, https://radimrehurek.com/gensim/
[7] Matplotlib Python Library, https://matplotlib.org/
[8] Scipy Python Library, https://www.scipy.org/
[9] PyEMD Python Library, https://github.com/wmayner/pyemd