Project Report *API Mention Recognition* in StackOverflow

CZ4045 - Natural Language Processing

Deka Auliya Akbar
U1323056K
DEKA0001@e.ntu.edu.sg

Peter
U1320143C
PETE0013@e.ntu.edu.sg

Stefan Artaputra Indriawan
U1320079K
SINDRIAW001@e.ntu.edu.sg

Stefan Setyadi Tjeng
U1420118A
STEF0019@e.ntu.edu.sg

Zillion Govin
U1320174E
ZILL0001@e.ntu.edu.sg

ABSTRACT

This project report describes our process of implementing an API Mention recogniser for one of the most well-known online discussion forum for programmers, StackOverflow. Our main steps for completing this project consist of defining our meaning for API Mention, annotating API Mention using Brat Rapid Annotation Tool in our dataset obtained with StackExchange API, and training our model to recognise API Mention in StackOverflow post. The objective of the project is to create a classifier by utilizing the existing Stanford NER tool and train it so that it can recognise API mention based on our definition.

Keywords

Named Entity Recognition, API Mention, CRF Classifier, BIO Encoding

1. INTRODUCTION

StackOverflow, one of the flagship site of StackExchange Network of Q&A sites, has been the most commonly used discussion platform by developers to seek assistance in their programming problems.

In this project, we are tasked with developing an application to recognise API mentioned by StackOverflow users. With around 5000 questions posted on the site daily ranging from various programming languages, we limited the scope of our project to all versions of Java programming language.

In addition, we utilized various tools to assist our Natural Language Processing task, such as StackExchange API, Brat Rapid Annotation Tool, and Stanford Named-Entity Recogniser.

2. METHODOLOGIES

This project can be divided into several steps. First, we need to collect our data set, which is a collection of StackOverflow posts. Second, we need to preprocess the data so that it can be used as the input for the classifier and derive some analysis out of it. Next, we defined what is API Mention and annotate the preprocessed text data to establish the gold standard using Brat. Afterwards, we performed API Mention Recognition by first training a classifier and test it against the gold standard using Stanford NER Classifier. Note that before we train the classifier, the text files are first aggregated into one, shuffled, and split into 4 folds to provide an honest estimate about the classifier performance. Last, we evaluate and perform an error analysis on the classifier model, and derive the possible applications for recognizing API mention.

2.1 Tools

1. Programming Languages: *Python* 3.5 and *Java* 8
2. External python packages:
   (a) *Requests 2.11.1*, an HTTP library for Python. This package is used to collect the StackOverflow posts over the internet.
   (b) *Natural Language Toolkit (NLTK) 3*, an open source library for working with human language data in Python.
   (c) *BeautifulSoup 4.5.1*, a Python library designed for pulling data out of HTML and XML files.
   (d) *Pandas 0.18.1*, an open source library for providing data structures and data analysis tools.
   (e) *matplotlib 1.5.3*, a python 2D plotting library for visualization tools.
3. *StackExchange API*: api.stackexchange.com, provides a way to retrieve StackOverflow posts through REST API in JSON format.
4. *HTML, JavaScript, AJAX* and *Mozilla Firefox*. These tools are used to create HTML viewer for selecting the StackOverflow posts for the project.
5. External Java packages: *Stanford NER* 3.6.0, a Java implementation of a Named Entity Recogniser. This software provides a general implementation of linear chain Conditional Random Field (CRF) sequence models.
6. *Brat* annotation tool brat.nlplab.org, an online environment for collaborative text annotation.
7. *Google Chrome*, web browser for running brat service.

2.2 Data Set Collection

Initially, we downloaded 100 top voted StackOverflow question threads tagged with Java within 6 months period between 2011-2015 (2 data sets / year x 5 years = 10 data sets). This data is retrieved by making a `GET` request to StackExchange API [1]. This call results in obtaining 10 JSON responses which aggregates to 8244 posts in total, consisting of 1000 question posts and 7244 answer posts.

From these JSON files, we then further selected 735 posts from 288 different threads as the data that we used throughout our whole project. [2]

3. DATA PREPROCESSING

Since StackOverflow is a web service, the dataset that we collect still contains HTML tags, therefore data preprocessing is necessary. The preprocessing takes the input of raw JSON data, extract the post body and then write the preprocessed post body that has been cleaned into a new txt file. [3] [4] To help with removing the HTML tags, we use BeautifulSoup 4 from Python library.

In algorithm 1 pseudo code, the function first calls BeautifulSoup with a *html parser* to parse the raw HTML string of a StackOverflow post body into HTML tree. Next, the function flattens the HTML nodes inside the HTML tree into a string by invoking the method *flatten(root_node)*.

Algorithm 1

```
1: function CLEAN(post)
2:     parsed ←BEAUTIFULSOUP(post)
3:     return FLATTEN(parsed)
```

Next, the algorithm 2 shows the pseudo-code for *flatten(root_node)* which extracts the text from the dirty stackoverflow post body by removing some HTML tag nodes and preserving the text of certain tag nodes. After the extraction, only the text content will be taken and the leading and trailing tags for these HTML nodes are removed.

As shown in the pseudocode 2, first we remove the `<blockquote>` and `<img>` nodes. Next, we perform further processing on these tag nodes:

- `<code>`: text content inside the *code* tags is always taken.
- `<a>`: text content inside the *anchor* tags is taken if they do not start with `http`.
- `<pre>`: text content inside *preformatted* tags is taken if they are a single-statement code (ended with one `;`), or a single-line string if it does not contain a semi-colon `;` (multi-line string is excluded).

The text extraction process is done by the `simplify` function with the aid of regular expressions as explained later in subsection 3.1. As for all the other HTML elements, their text contents are extracted but the html tags are removed.

[1] as implemented in `SourceCode/filter.py`

[2] The JSON files of the selected posts are available under `SourceCode/api_mention` directory.

[3] The preprocessing implementation can be seen in `SourceCode/preprocessing.py`

[4] The preprocessed texts are available under `SourceCode/api_preprocessed` directory.

Algorithm 2

```
1: function FLATTEN(node)
2:     if node is Text then
3:         return text content of node
4:     result ← empty string
5:     for each child in children of node do
6:         Ignore image and blockquote tag
7:         if node is pre tag then
8:             content ← text content of node
9:             if content is single-statement code or one-line
   code then
10:                 result ← result + SIMPLIFY(content)
11:             else
12:                 Ignore
13:         else if node is anchor tag then
14:             content ← text content of node
15:             if content is not link then
16:                 result ← result + SIMPLIFY(content)
17:             else
18:                 Ignore
19:         else if node is anchor tag then
20:             content ← text content of node
21:             result ← result + SIMPLIFY(content)
22:         else
23:             result ← result + FLATTEN(child)
24:     return result
```

One example of how the preprocessing works is as follows. Suppose we have a raw post body text:

```
Here is an example:  <pre>String</pre>.
```

Since the text inside <pre> tag is a one-liner, the pseudocode would extract the content of pre tag, hence the final preprocessed text becomes:

```
Here is an example:  String.
```

All in all, the preprocess script essentially modifies the i) input, which is in raw json file of StackOverflow posts with a lot of html tag elements, ii) extract certain text from various HTML nodes, iii) combines them into a string without html tags and iv) write them into a text file for further processing.

3.1 Regular Expressions

Post in StackOverflow usually contains inline codes found inside its paragraphs. Most of the time, these inline codes are contained within *code* tags. Before the tokenization process is carried out, these inline codes are preprocessed by the method `simplify()`, mention in algorithm 2, with the help of regular expressions below:

1. Character literal in *code* tag is replaced by `CharLiteral`. The regular expression to match character literal is:

   ```
   /'(?:\\u.+|\\.|.)?'/
   ```

   Basically, the regex above tries to match unicode literal `\uxxxx`; escaped character such as `\\`, `\"`, `\'`, etc; or any other character.

2. String literal in *code* tag is replaced by `StringLiteral`. The regular expression to match string literal is:

```
/"(?:\\"|[^"])*"/
```

The regex above also handles escaped double quote character.

3. Generic class syntax is simplified to `ClassName<T>`, for example: `Pair<String, Integer>` is transformed into `Pair<T>` for simplicity. The regex to match generic class is:

```
(\w+)\s*<[^<>]*(?:<[^<>]*(?:<[^<>]*(?:<[
^<>]*(?:<[^<>]*(?:<[^<>]*(?:<[^<>]*>[^<>
]*)*>[^<>]*)*>[^<>]*)*>[^<>]*)*>[^<>]*)*
>[^<>]*)*[^<>]*>
```

We understand that nested angle brackets are problems as regular expression has finite number of state. However, the regex above is able to match up to 7^{th} level nested brackets which should be sufficient to cover most cases.

4. Method and constructor call are simplified to:

`methodName(args)` and `ClassName<T>(args)`

respectively. The regex to match method and constructor call is:

```
/(\w+)\s*((?:<T>)?)\s*\([^()]*(?:\([^()]*
(?:\([^()]*(?:\([^()]*(?:\([^()]*(?:\([^
()]*(?:\([^()]*\)[^()]*)*\)[^()]*)*\)[^(
)]*)*\)[^()]*)*\)[^()]*)*\)[^()]*)*[^()]
*\)/
```

This regex also shares the same problem with the previous regex. Brackets in method or constructor call may be nested indefinitely. However, this regex is able to match up to 7^{th} level nested brackets which should be sufficient to cover most cases. Also any brackets inside character or string literals in method or constructor call arguments have been substituted to *CharLiteral* and *StringLiteral* as shown in the first two step. Thus, it will not break this regex.

4. SUMMARY STATISTICS

Various data for statistics is gathered from the dataset that is used for the assignment, including length of the posts, number of threads, as well as number of questions and answers, in total, and the distribution for the answers on all of our questions dataset. For the input, input method is used from fileinput, a python library, to read list of files that contain preprocessed dataset.

In total, there are 228 threads for the entire dataset, with the longest thread has 10 posts for the particular question post. From 228 threads, there are 139 question posts, and 596 answer posts.[5] All of the posts are vary in lengths. The shortest post is 1 word only, while the longest post contains 1730 words. Overall, the total number of words on all of the posts in the dataset are 94.816 words[6].

As for the distribution of the answers, since the number of the answers above 5 are pretty scarce, we decided to make an upper limit of 6 (other answers more than 6 will increase count of 6 answers instead) so that the data can be more condensed. From the data gathered, question with 1 answer only has a really high count, soaring around 42% of the total data, or around 110 in total. While the second highest count is around 26%,or around 70, for question with 2 answers. The rest of the details can be seen from figure 1.

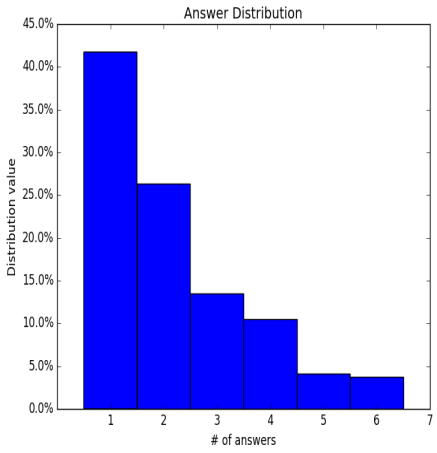


Figure 1: Histogram for distribution of answers

5. DATA SET ANALYSIS

5.1 Stemming

In linguistic morphology and information retrieval, stemming is the process of reducing inflected (or sometimes derived) words to their word stem, base or root form, generally a written word form.

Snowball stemmer is used from NLTK library to stem the English word, and also stopwords from the same library to exclude any stopword from entering the top-20 most frequent words in our entire dataset. some new words are also added to the stopword dictionary such as n't or would as we believe that these are should be included in stopwords as well.

Based on table 1, the word 'use' is the most frequent one after stemming the words. However, without stemming, the word 'method' take number 1 position, although both of them does not change much in terms of rank. The word 'use' is the most frequent word after stemming most likely because 'use' appears with several suffixes in various posts, such as 'uses', 'using', or 'used', and actually several of them are in the most frequent words as well. If the word are not stemmed, they will be counted as separate words, and thus diluted the total number of word 'use' in the dataset.[7]

Table 1 indicate the rank of words before and after stemming.

[5]The detailed result are available under `SourceCode/Stat_API/` in `num_of_thread.txt`.

[6]The detailed results are available under `SourceCode/Stat_API/` in `post_length.txt`.

[7]The detailed results for stemming are available under `SourceCode/Stat_API/` directory

Rank	Before		After	
	Total	Word	Total	Word
1	590	method	1089	use
2	477	use	777	method
3	381	code	450	class
4	357	Java	448	string
5	329	class	417	java
6	272	like	400	code
7	263	using	345	object
8	253	one	343	call
9	245	new	335	stream
10	230	String	331	return
11	205	type	329	valu
12	202	way	300	one
13	202	also	292	like
14	201	value	275	type
15	201	object	266	implement
16	198	array	258	array
17	184	get	248	new
18	181	want	248	also
19	181	need	246	list
20	178	used	238	need

Table 1: 20-most frequent words before and after stemming

5.2 POS Tagging

Part of Speech (POS) tagging or POST has been one of the important task in text processing. By performing POST, we tokenize sentences and assign each token with a label.[[1]]

We used Natural Language Toolkit (NLTK) for Python to perform POST to 10 randomly (manually) selected sentences from our dataset.[8]

Using the default pre-trained Maxent treebank POS tagging model in NLTK, we applied POST to each of the 10 sentences using `nltk.pos_tag()` method after tokenizing the sentences with `nltk.word_tokenize()` method and obtained the following results:

1. $\underbrace{It}_{PRP}\ \underbrace{has}_{VBZ}\ \underbrace{generally}_{RB}\ \underbrace{been}_{VBN}\ \underbrace{the}_{DT}\ \underbrace{case}_{NN}\ \underbrace{the}_{DT}\ \underbrace{Java}_{NNP}\ \underbrace{source}_{NN}$ $\underbrace{code}_{NN}\ \underbrace{has}_{VBZ}\ \underbrace{been}_{VBN}\ \underbrace{foward}_{RB}\ \underbrace{compatible}_{JJ}\ \underbrace{.}_{.}$

2. $\underbrace{I}_{PRP}\ \underbrace{have}_{VBZ}\ \underbrace{a}_{DT}\ \underbrace{server}_{NN}\ \underbrace{side}_{NN}\ \underbrace{implemented}_{VBD}\ \underbrace{in}_{IN}\ \underbrace{Scala}_{NNP}$ $\underbrace{and}_{CC}\ \underbrace{React/Flux}_{NNP}\ \underbrace{based}_{VBD}\ \underbrace{front}_{NN}\ \underbrace{end}_{NN}\ \underbrace{.}_{.}$

3. $\underbrace{No}_{DT}\ \underbrace{,}_{,}\ \underbrace{you}_{PRP}\ \underbrace{appear}_{VBP}\ \underbrace{to}_{TO}\ \underbrace{have}_{VB}\ \underbrace{a}_{DT}\ \underbrace{legit}_{NN}\ \underbrace{bug}_{NN}\ \underbrace{.}_{.}$

4. $\underbrace{There}_{EX}\ \underbrace{is}_{VBZ}\ \underbrace{no}_{DT}\ \underbrace{actual}_{JJ}\ \underbrace{difference}_{NN}\ \underbrace{in}_{IN}\ \underbrace{the}_{DT}\ \underbrace{functionality}_{NN}$ $\underbrace{between}_{IN}\ \underbrace{the}_{DT}\ \underbrace{2}_{CD}\ \underbrace{version}_{NN}\ \underbrace{'s}_{POS}\ \underbrace{loop}_{NN}\ \underbrace{.}_{.}$

5. $\underbrace{For}_{IN}\ \underbrace{me}_{PRP}\ \underbrace{,}_{,}\ \underbrace{the}_{DT}\ \underbrace{reason}_{NN}\ \underbrace{is}_{VBZ}\ \underbrace{a}_{DT}\ \underbrace{likely}_{JJ}\ \underbrace{performance}_{NN}$ $\underbrace{inprovement}_{NN}\ \underbrace{at}_{IN}\ \underbrace{a}_{DT}\ \underbrace{negligible}_{JJ}\ \underbrace{cost}_{NN}\ \underbrace{in}_{IN}\ \underbrace{terms}_{NNS}\ \underbrace{of}_{IN}$ $\underbrace{code}_{NN}\ \underbrace{clarity}_{NN}\ \underbrace{.}_{.}$

6. $\underbrace{I}_{PRP}\ \underbrace{'ve}_{VBP}\ \underbrace{got}_{VBN}\ \underbrace{of}_{IN}\ \underbrace{the}_{DT}\ \underbrace{following}_{JJ}\ \underbrace{five}_{CD}\ \underbrace{6}_{CD}\ \underbrace{methods}_{NNS}$ $\underbrace{to}_{TO}\ \underbrace{do}_{VB}\ \underbrace{it}_{PRP}\ \underbrace{.}_{.}$

7. $\underbrace{Usually}_{RB}\ \underbrace{you}_{PRP}\ \underbrace{'ll}_{MD}\ \underbrace{be}_{VB}\ \underbrace{looking}_{VBG}\ \underbrace{for}_{IN}\ \underbrace{the}_{DT}\ \underbrace{hostname}_{NN}$ $\underbrace{your}_{PRP\$}\ \underbrace{host}_{NN}\ \underbrace{has}_{VBZ}\ \underbrace{in}_{IN}\ \underbrace{a}_{DT}\ \underbrace{specific}_{JJ}\ \underbrace{context}_{NN}\ \underbrace{.}_{.}$

8. $\underbrace{This}_{DT}\ \underbrace{is}_{VBZ}\ \underbrace{just}_{RB}\ \underbrace{micro\text{-}optimisation}_{JJ}\ \underbrace{that}_{IN}\ \underbrace{you}_{PRP}\ \underbrace{should}_{MD}$ $\underbrace{n't}_{RB}\ \underbrace{worry}_{VB}\ \underbrace{about}_{IN}\ \underbrace{.}_{.}$

9. $\underbrace{It}_{PRP}\ \underbrace{'s}_{VBZ}\ \underbrace{a}_{DT}\ \underbrace{feature}_{NN}\ \underbrace{of}_{IN}\ \underbrace{the}_{DT}\ \underbrace{Java}_{NNP}\ \underbrace{language}_{NN}\ \underbrace{.}_{.}$

10. $\underbrace{After}_{IN}\ \underbrace{digging}_{VBG}\ \underbrace{around}_{IN}\ \underbrace{for}_{IN}\ \underbrace{a}_{DT}\ \underbrace{while}_{NN}\ \underbrace{,}_{,}\ \underbrace{I}_{PRP}\ \underbrace{ca}_{MD}$ $\underbrace{n't}_{RB}\ \underbrace{say}_{VB}\ \underbrace{that}_{IN}\ \underbrace{I}_{PRP}\ \underbrace{find}_{VBP}\ \underbrace{the}_{DT}\ \underbrace{answer}_{NN}\ \underbrace{,}_{,}\ \underbrace{but}_{CC}\ \underbrace{I}_{PRP}$ $\underbrace{think}_{VBP}\ \underbrace{it}_{PRP}\ \underbrace{'s}_{VBZ}\ \underbrace{quite}_{RB}\ \underbrace{close}_{JJ}\ \underbrace{now}_{RB}\ \underbrace{.}_{.}$

Observing the results, there are a few misclassifications such as classifying *legit* as *Noun* (3^{rd} sentence), and *micro-optimisation* as *Adjective* (8^{th} sentence). This might be due to *legit* being an informal form for *legitimate*, and *micro-optimisation* being a not commonly used word (mainly used in the topic of optimising code in programming).

Further discussion speculates that the misclassification occurs due to the fact that POST relies on the previous word for the tagging process. In the case of *legit*, the word *legit* appears after the word *a*, which is a *Determiner* and is usually used to mention a *Noun* afterwards.

The same explanation goes for *micro-optimisation*. The previous word, *just*, is tagged as an *Adverb*, and is commonly used to modify the meaning of an *Adjective*. This fact may possibly contribute to the misclassification of *micro-optimisation* as *Adjective*.

6. API MENTION ANNOTATION

We collected 735 StackOverflow posts before proceeding to annotate the API Mention in dataset using Brat rapid annotation tool, a web-based tool for text annotation. Brat is chosen as it supports text span annotation which is useful for NER task.

6.1 Definition of API Mention

As mentioned in the introduction, we limited our scope of focus to all existing versions of Java programming language. Our definition of API mention will be based solely on classes definition, including Abstract classes and Interfaces, and their methods defined in JDK documentation.

[8]All the codes, selected texts and the POS tag results are available under `SourceCode/pos_tagging` directory.

To put it simply, we will annotate the following features as API mention:

- Class Name
  - Class Name Mention
    - e.g. `String`, `Integer`, `Stream`, etc.
  - Package Mention that ends with class name
    - e.g. `java.lang.String`
- Method Call
  - Static Method
    - e.g. `String.trim()`
  - Instance Method
    - Starting from "."
    - e.g. `.equals()` from `variable.equals()`
  - Method only (with or without parenthesis)
    - e.g. `toString()`, `trim()`, etc.

Additionally, those API mentions must adhere to the following formatting requirements:

- Case Sensitive
- Exact Spelling
- Not surrounded by `<blockquote></blockquote>` tag
- Not enclosed by multi-statement or multi-line codes in `<pre></pre>` tag

For the method call, we do not define any regulations for the number of parameters or non-existing parentheses as long as it is a valid method name defined in Java documentation.

6.2 BIO Encoding

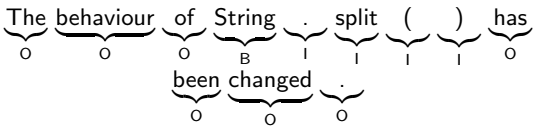
The next crucial step following annotation was to map every single word with a set of tokens.

We implemented BIO (Beginning, Inside, Outside) tagging format to tokenize our annotated dataset by assigning label *B (Beginning)* to every start of an API mention, *I (Inside)* to the next chunk of the mention, and *O (Outside)* to other words. For example:

- Original Text

  The behaviour of String.split() has been changed.

- Tokenized Text



The resulting brat annotated dataset from the previous part (*.ann* format) played a great role in this step as it enabled our code to recognise the beginning and end of an API mention easily. The resulting BIO-formatted dataset was then used as our training and testing data.

7. API MENTION RECOGNITION

In order to perform API Mention Recognition, we use a classifier. Classifier is a supervised learning task where the classifier is trained using training data with some gold standard. Then, the trained classifier will be tested against an unseen set of data (test data) and should able to predict the correct labels for the test data.

7.1 CRF Classifier

The training data will be fed to Stanford NER which implemented a linear chain CRF classifier. As an introduction, a Conditional Random Field (CRF) classifier is a discriminative sequence classifier. It relies on training a set of weighted feature function.

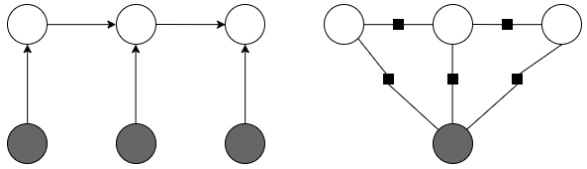


Figure 2: Hidden Markov Model (left) and Conditional Random Field (right)

A Linear Chain CRF is a special type of CRF that relies only on the current label and the previous label. The feature function in CRF also allows it to learn the relation between tags whereas HMM models assume a strong independence between tags. As seen in figure 2, the HMM relies on observation likelihood whereas CRF feature function can be used on the whole sequence which makes it context sensitive. HMM can be thought of as a strict linear chain CRF where it only have the conditional distribution of state transition and observation likelihood as its feature function.

Stanford NER uses local dependant feature function to understand context and Viterbi algorithm to infer the most likely sequence of tags.

7.2 Training and Testing

We used tab separated value files which consist of token-BIO label pairs as the input to the NER classifier. These tsv files are required to establish the gold standard for the classifier. In order to generate the tsv files, the text data is first annotated by hand using brat, then a brat converter is run to tokenize the annotated text and generate the respective BIO labels accordingly. The way the tokenizer works is that it first split the text by whitespace, and then by punctuations, hence each token will consist of either consecutive alphanumerical characters or a punctuation. Since brat stores the annotation positions of the annotated text in *.ann*, it is able to compute the BIO labels for each token easily. [9]

After the gold standard is established, we used K-Fold Cross Validation to evaluate our classifier model with K = 4. First, all the tsv files are aggregated and shuffled randomly using seed = 42. Next, the aggregated tsv files are split into 4 folds, therefore, 4 training files and 4 testing files will be generated. [10] Each training file will have 3 folds, and the

[9]The tsv files of the gold standard can be seen in the `.conll` files under `SourceCode/brat/data/stackoverflow/` directory.

[10]The splitting implementation is done by `SourceCode/split.py`.

last remaining fold will be isolated and used for testing. [11]

Next, we trained the Stanford NER classifier on the 3 folds and tested the classifier model against the 1 remaining fold by feeding the corresponding training and the testing files. This process is repeated 4 times to cover for all the available folds.

7.3 Results

Stanford NER classifier generates 4 tsv results corresponding to the K-th fold which consist of 3 tab-separated values: the token, gold standard annotation label, and the classifier annotation label.[12] Other than the label predictions, the classifier also computes the scores for the following items:

1. *True Positives*: number of tokens which are correctly labeled by the classifier.

2. *False Positives*: number of tokens which are not labeled as API Mention in the Gold Standard, but incorrectly labeled as API Mention by the classifier.

3. *False Negatives*: number of tokens which are labeled as API Mention in the Gold Standard, but not labeled as API Mention by the classifier.

4. *Precision*: measure the number of predicted entity labels that matches exactly with the ones in the gold standard evaluation data.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

5. *Recall*: measure the number of labels in the gold standard that appear at exactly the same location in the predictions.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

6. *F1*: harmonic mean of Precision and Recall.

$$F1 = \frac{2 \times TP}{2 \times TP + FP + FN}$$

8. EVALUATION AND ERROR ANALYSIS

We use K-fold to evaluate the performance of Stanford NER. Table 2, 3, 4, 5 show the performance result in each fold respectively. From these results, we can observe that all 4 models have high precision and decent recall. [13] [14]

These trained models have false positive and false negative predictions due to a number of reasons. One reason is that the classifier mistakenly classify some user's defined method or class as an API mention, which are supposed to be excluded by our definition.

[11]The training and testing files (`train-[idx].tsv` and `test-[idx].tsv`) are available under `SourceCode/train/` directory.

[12]These predicted labels are available in `result-[idx].tsv` files under `SourceCode/train` directory.

[13]Note that in order to reproduce the result as shown in these tables, use seed number 42.

[14]The training results can be found under `SourceCode/` directory in `train/result-0.txt, train/result-1.txt, train/result-2.txt`, and `train/result-3.txt` for fold-1, fold-2, fold-3, and fold-4 result respectively

Table 2: Fold-1 result

Fold-1		Predicted	
		True	False
Actual	True	892	116
	False	68	27,626
	Precision	Recall	F1
	0.9292	0.8849	0.9065

Table 3: Fold-2 result

Fold-2		Predicted	
		True	False
Actual	True	813	154
	False	135	33,020
	Precision	Recall	F1
	0.8576	0.8407	0.8491

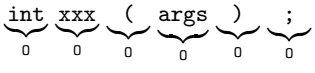
8.1 False Positive Case

Here are some cases of false positive prediction from Stanford NER:

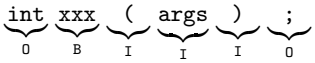
- In stackoverflow.com/q/22561853, at the sentence:

  > ... then it's essentially looking for this signature:
  >
  > `int xxx(args);`

  the model classifies `int xxx(args);` as an API mention which is not supposed to be according to our definition. Actual prediction:

  Classifier's prediction:

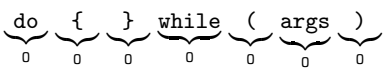
  

  Although Stanford NER never learns about this method, it may classifiy `xxx(args)` as an API mention due to the similarity of token sequence as all other method signature.

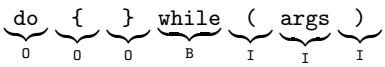
- In stackoverflow.com/q/24609841, at the sentence:

  > Here, do { } while(!...) is a blocker/interrupt

  Stanford NER misclassify `while(args)` as an API mention due to the preprocessing result done by using regular expression. Actual prediction for this sequence of token is:

  Classifier's prediction:

  This happens due to the fact that the string `'while(args)'` has same signature as all other JDK method and constructor (`'methodName(args)'` and `'constructor(args)'`).

Table 4: Fold-3 result

Fold-3		Predicted	
		True	False
Actual	True	941	139
	False	94	34,995
	Precision	Recall	F1
	0.9092	0.8713	0.8898

Table 5: Fold-4 result

Fold-4		Predicted	
		True	False
Actual	True	963	162
	False	77	33,546
	Precision	Recall	F1
	0.9260	0.8560	0.8896

8.2 False Negative Case

Here are some cases of false negative prediction from Stanford NER:

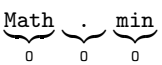
- In stackoverflow.com/q/22501695, at the sentence:

  > The approach used by `Math.min` is similar to what Jesper proposes but a little ...

  Stanford NER fails to recognise `Math.min` as an API mention. Actual prediction:

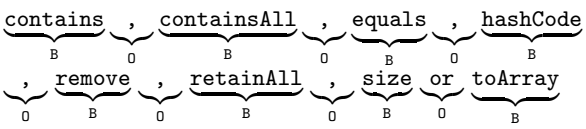
  Classifier's prediction:

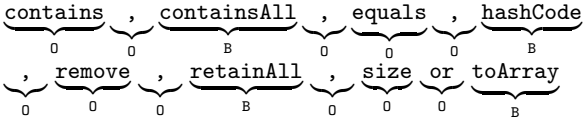
- In stackoverflow.com/q/6416788, at the sentence:

  > As soon as you call `contains`, `containsAll`, `equals`, `hashCode`, `remove`, `retainAll`, `size` or `toArray`, you'd have to traverse the elements anyway.

  In this case, Stanford NER fails to recognise `contains`, `equals`, `remove`, and `size`. These methods are founded in `AbstractCollection` class. Actual perdiction:

  Classifier's prediction:

9. APPLICATIONS

Developers often utilize APIs to get their job done on a daily basis. Learning how to use APIs has been an important skill in developing software applications. We have proposed various ideas of how we can make use of the API Mention Recogniser to assist the work of a software developer.

Table 6: Micro- and macro-averaged results

	Micro-averaged	Macro-averaged
Precision	0.9061	0.9055
Recall	0.8634	0.8632
F1	0.8842	0.8838

These ideas can then be developed to create useful real-world applications. The final product of API annotation tool caters for any programming language, in the form of a plugin for various browsers. When the user enable the plugin, he/she can enable a scanning feature by right-clicking on the web page (similar to Google Translate). The scanning feature allows our tool to scan the HTML page as an input, and output the underlined or annotated API mentions on the same page with a sharp precision.

To enhance it further, we can add event listeners on the annotated API Mentions and connect them with various web services therefore allowing direct access to the API Mention information.

9.1 Automatic API Mention Recognition

Our first idea is providing a service for annotating API Mention from user input text. Firstly, the user sends a full random text as an input to the annotation service. A robust trained classifier model will then annotate the API mention from the given text and highlight the recognised API Mention by underlining it, or giving a list of the API mention that has been annotated, depending on user's preference.

We can improve user interactivity on the annotation by performing further actions based on user's actions. For example, the API recogniser will direct the users to the relevant API Mention documentations when the user clicks on the annotation. Our tool can also give the example of the API usage when the user hovers over the API Mention.

9.2 Automatic API Mention Searching

While using Brat for our annotation process, we found a feature in Brat that can perform direct search on the annotated entities by clicking the annotation and on the search link. For experimental purpose, we set the search option for API Mention query searching to be a simple directed Google Search to Java 8 API Documentation.

> Search: `<api_mention_keyword>`
> `site:https://docs.oracle.com/javase/8/docs/api`

However, we can extend this further. Suppose we have a full-fledge web service for API Mentions that provides API for querying the full information about API Mentions, including their definitions, background theoretical concepts, examples of correct usages, common misconceptions, etc. With this service, the role of API Mention annotator is significant. After the automatic API recognition is done on the user input text, we can connect our API annotation service with the web service API to extract the important information about the annotated API, and then display this information once the user clicks the annotated API Mention within the same page. This process frees users from having to search the API on the internet (browsing through several webpages and performing testing/validation to find correct information about the API Mention is time consuming). Therefore, the API mention recogniser can speed up

the learning process of various APIs and promotes productivity for developers in using APIs when developing software applications.

10. REFERENCES

[1] TextMiner (2014, July 9).*Dive Into NLTK, Part III: Part-Of-Speech Tagging and POS Tagger*. Retrieved from `http://textminingonline.com`

[2] The Stanford Natural Language Processing Group, `nlp.stanford.edu/software/CRF-NER.shtml`

[3] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. 2005. Incorporating Non-local Information into Information Extraction Systems by Gibbs Sampling. *Proceedings of the 43nd Annual Meeting of the Association for Computational Linguistics (ACL 2005),* pp. 363-370. `nlp.stanford.edu/ manning/papers/gibbscrf3.pdf`

[4] Sutton, C. and A. McCallum (2012). An Introduction to Conditional Random Fields. *Found. Trends Mach. Learn.4*(4), 267-373. doi:10.1561/2200000013