Generalized Tokenization and its Application on Stack Overflow Posts*

Chen Hailin
U1520636J
chen1039@e.ntu.edu.sg

Deng Yue
U1520600L
deng0068@e.ntu.edu.sg

Liu Hualin
U1420733B
liuhualin333@gmail.com

Shi Ziji
U1520644B
ZSHI005@e.ntu.edu.sg

ABSTRACT

As one of the most popular programming forums, Stack Overflow contains both text and code snippets. However, when searching and recommending programming questions to users, code data is usually not used. In this project, we developed a generalized tokenizer which can tokenize both text and code in Stack Overflow posts. We first define the code token and text token. Then, we implemented two tokenizers using regular expression and conditional random field(CRF) respectively. We found regular expression tokenizer has outperformed CRF tokenizer in our dataset in f1 score. Using the best tokenizer, we conducted analysis on irregular tokens and POS tagging. Finally, we developed an application to extract top 4 key words in a post/answer.

1. INTRODUCTION

Stack Overflow is a forum dedicated to programming related topics. Currently, users can search questions on the forum by inputing keywords and can also get recommendation on similar questions. As the biggest dataset on Computer Science related Q&A, it has attracted many researches on topics including good programming style analysis[12], auto-tagging for posts[18], developer interaction in Stack Overflow[21]. However, as a textual dataset, Stack Overflow differ from widely used dataset like Penn Treebank or tweets as there are not only normal language texts but also a large portion of unstructured codes in Stack Overflow. Over the years, researchers have widely studied processing techniques for unstructured natural texts but those techniques cannot be applied directly to unstructured codes[19]. The first step for any further processing is tokenization, in which we group characters to atomic meaningful token and discard not meaningful characters. To get rich information in user posts, we define tokens for both code and normal texts. In this paper, we formulated both tokens definition and developed two tokenizers, a regular expression based and a CRF based. Based on the best tokenizer, We conducted further analysis on irregular tokens and applied POS tagging on randomly chosen sentences. With tokens retrieved in both text and code section, we developed an application which extracts top 4 keywords from any post/answer.

*This paper serves as partial fulfillment of course CZ4045: Natural Language Processing in Fall 2017 semester at Nanyang Technological University, Singapore. Supervised by Assoc Prof. Sun Aixin.

2. DATASET COLLECTION & ANALYSIS

2.1 Dataset Collection

In this section, we will introduce the methods used to obtain the dataset. We shall begin with information on the data source we use, then justify that the dataset satisfy the requirements of this assignment. We will close this section with statistics about our dataset.
The periodic anonymized data dump file from Stack Overflow can be found at the "Stack Exchange Data Dump" website.[1]

2.1.1 Description of Data Dump

The data dump contains all user-generated data on Stack Overflow. It is formatted in XML and zipped via 7-zip and it uses bzip2 compression. It contains posts, answers, usernames, votes, comments, post history, and post links. [1] This data dump contains all posts and answers in the website as of 31th August, 2017.

2.1.2 Requirements of Dataset

The collected dataset should follow three conditions:

(a) It must have at least 500 threads of discussion.

(b) The selected threads should be about only one programming language.

(c) Each thread must have at least two posts. A post is considered as either a question or an answer.

2.1.3 Method used in dataset collection

In dataset collection, we use external package BeautifulSoup[17] to read XML and find desired thread of discussion. A desired thread of discussion is defined as follows:

(a) A desired thread must have "python" in its tags (We would like to focus on python questions)

(b) A desired thread must have at least three answers

(c) A desired thread must have at least 200000 views

First we read the data dump line by line. For each line, we use BeautifulSoup to parse XML, it enables us to read XML properties easily. Based on the definition of desired thread above, we constraint the tags to the three above criteria. Since the total count for the threads are above 600 and each thread contains at least four posts, it fulfills the requirements in section 2.1.2 .

2.1.4 *Dataset Statistics*

External package MatPlotLib [5] is used to visualize dataset statistics. Three figures are generated below:

Figure 1 shows how many posts/answers are in the dataset.

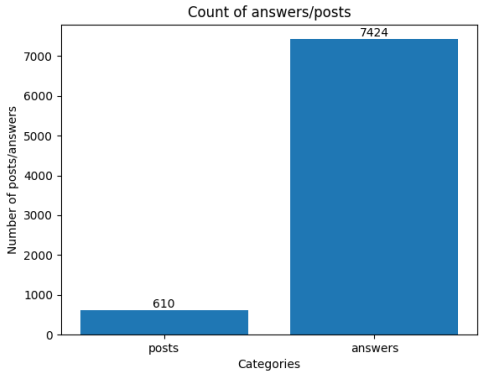

Figure 1: Number of posts and answers in our dataset.

Figure 2 shows the distribution of answer counts per question.

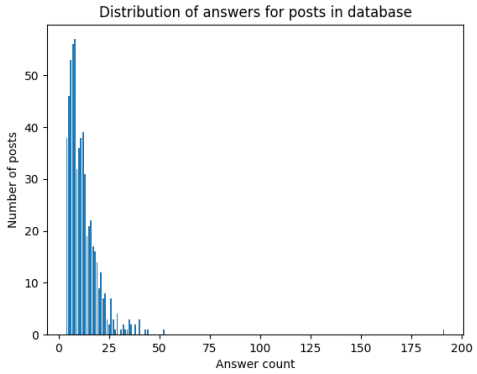

Figure 2: Distribution of answer counts per question in dataset.

Figure 3 shows the distribution of token counts of posts in dataset.
We use nltk's tokenizer for calculating total number of tokens, and it results in 362748 tokens.

2.2 Dataset Analysis

2.2.1 *Stemming*

Stemming is a process to obtain the common form (stems) of words in different morphological forms. It could reduce the number of word variations to avoid over-fitting a search query. We compare the frequency of top 20 most frequent words in our dataset, before and after stemming. We use Snowball stemmer, a variation of Porter Stemmer, from NLTK[8].

We observe that :*After stemming, the number of occurrences of some frequent words increases, which affects their ranking.*

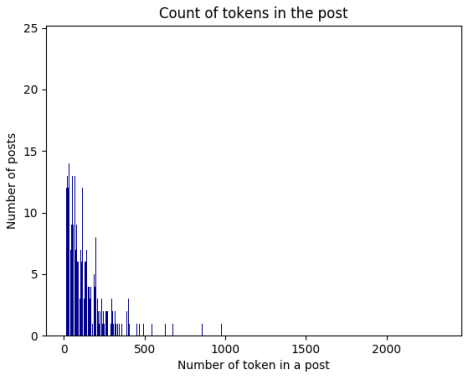

Figure 3: Distribution of token counts of posts in dataset (using NLTK).

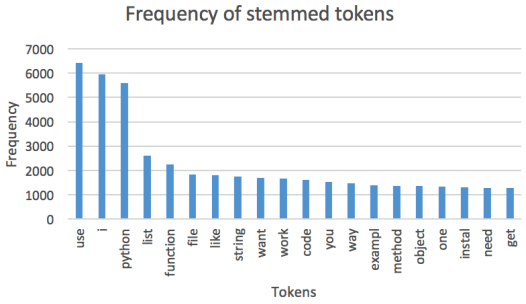

Figure 4: Frequency of top 20 most common unstemmed tokens.

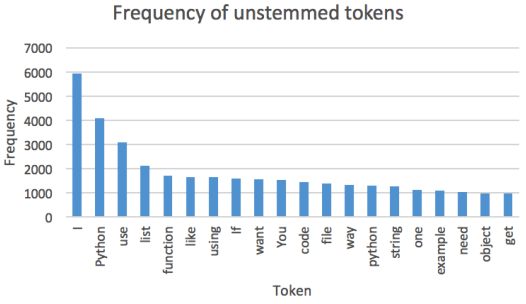

Figure 5: Frequency of top 20 most common stemmed tokens.

For instance, the frequency of token *"use"* has doubled from 3096 to 6417, and the usage of *"function"* has increase from 1713 to 2236 times. However, some words like *"I"* remains unchanged after stemming. We attribute this increment of frequency to the morphological forms of the former type of tokens. The variety of their original form in our dataset is interesting and summarized in appendix A.
Since stemming may effect the frequency ranking, it will therefore affects TF-IDF. We will explain this point in the following section.

2.2.2 *POS-tagging*

We randomly selected 10 sentences from the text section of

dataset, and applied POS-tagging. Three examples are as following:

Example 2.2.2(a) :

IN	DT	NN	PRP	MD	VB	VBG	DT	JJ	NN
In	this	case	you	might	consider	using	a	<	code
NN	VBN	NNP	NNP	NNP		RB	IN	DT	
>	set	<	/code	>	,	especially	if	the	
NN	VBZ	RB	VBN	TO	VB	NNS			
list	is	n't	meant	to	store	duplicates	:		

Example 2.2.2(b) :

WP	VBZ	DT	NN	IN	DT	JJ	
What	is	the	behavior	of	the	pre-increment/decrement	
NNS	(	NNP	:	)	IN	NNP	.
operators	(	++/	--	)	in	Python	?

Example 2.2.2(c) :

RB	VBZ	DT	JJ	NN	PRP	VBD	IN	PRP$
Here	is	some	sample	output	I	ran	on	my
NN	,	VBG	PRP	TO	DT	NN	RB	RB
computer	,	converting	it	to	a	string	as	well
:								

We here assume that code section should not be POS-tagged because of its specialty. We then took the three examples because they represent three types of text sentences in our dataset : (a) a sentence with <code> tags embedded; (b) sentence with special characters; and (c) regular English text. We observes that nltk is achieving high accuracy in cases where the portion of special characters is low and input sequences is well-tokenized. However, nltk will be likely to fail if the input token sequence is illy-formed. For instance, token "pre-increment/decrement" in example 2.2.2(a) should have been further separated into "pre-increment" "/" and "decrement", which will result in noun tags. But it returns adjective tag. The off-the-shelf nltk tokenizer should account for it.

As such, a well-design tokenizer will be required to produce high quality POS-taggs. We will explain our tokenizer design in the next section.

3. TOKENIZATION

In this section, we will first explain our definition of tokens. We will then show how we annotate the original data and discuss the evaluation metric in tokenization. Next we will illustrate the inner mechanisms of the tokenizers implemented in this project. It will feature two kinds of word tokenizers, regular expression tokenizer and conditional random fields tokenizer.

3.1 Token Definition

This section defines text tokens and code tokens. During the tokenization process, we use XML tags like <c>*token*</c> to specify code tokens and use <t>*token*</t> to specify text tokens. We assume that *all code tokens resides in between <code> and </code> tags.*

3.1.1 Code Token Definition

In our experiment, we define text blocks surrounded by *<code>*, *</code>* tag as code blocks. Texts in such code blocks usually contain python source codes, comments and error messages. These texts possess different lexical grammar, vocabulary compared to normal English text. Thus a separate definition of tokens in such codes is necessary and defined as follows:

ID	Raw_Text	Token
1	[1,2]	[<c>1</c>,<c>2</c>]
2	a.b	<c>a</c><c>.</c><c>b</c>
3	user.py	<c>user.py</c>
4	argv[1]	<c>argv</c><c>[1]</c>
5	print()	<c>print(</c>)
6	"text"	"<c>text</c>)"
7	python -m	<c>python</c><c>-m</c>
8	a-b	<c>a</c><c>-</c><c>b</c>
9	1.2.3	<c>1.2.3</c>
10	/usr/lib/python3	<c>/usr/lib/python3</c>
11	:	Not Token
12	>>>	Not Token
13	#	Not Token

Due to limited contexts available, we listed several typical token definition on codes, each assigned an ID. For ID 1, we do not consider '[', ',', ']' as tokens as they are more of syntactic operators instead of concrete variables or function calls. Likewise, ID 11, 12, 13 are not considered tokens. For ID 2, we consider '.' as a code token since '.' is an important operator in codes. On the other side, we don't take '.' as token in text section as it is just a period sign. For ID 5, we consider *print(* as a whole a token to signify it is a function. For ID 4/7/9/10 , we consider tokens as a group of texts carrying atomic semantic meaning such as *-m* and *[index].*

3.1.2 Text Token Definition

The text tokens are defined as follows:

(a) Normal word separated by whitespace is considered as one token. (e.g. "NLP is great!" will be tokenized to <t>NLP</t> <t>is</t> <t>great</t>!)

(b) Special Pronoun is considered as one token. (e.g. "Albert Einstein" will be tokenized to <t>Albert Einstein</t>). We decide to put special pronoun in one text token since a part of the special noun has no meaning.

(c) Formula is considered as one token. (e.g. "E = mc2" will be tokenized to <t>E = mc2</t>). Formula is in one token because part of the formula has no semantic meaning in the sentence.

(d) Noun with two or more dots will be considered as one token (e.g. "Ph.D." will be tokenized to <t>Ph.D.</t>)

3.2 Token Annotation

In our experiment, we randomly selected 50 posts and 50 answers from the dataset collected. Following code token definition and text token definition above, we manually annotated the 100 posts using *<c></c>* annotation for code tokens and *<t></t>* annotation for text tokens

3.3 Evaluation metrics

In this section, we come up with two evaluation metrics, which evaluate performance of our tokenizers. The first one considers tokenization in information retrieval context: from a vast pool of texts, retrieve relevant/correct tokens. As such,

$$precision = \frac{|correct\_tokens \cap retrived\_tokens|}{|retrived\_tokens|}$$

$$recall = \frac{|correct\_tokens \cap retrived\_tokens|}{|correct\_tokens|}$$

$$f1 = 2 * \frac{precision * recall}{precision + recall}$$

The second evaluation considers tokenization as a sequence labelling task: assign a label for each character in original text. The label standard used in this experiment is similar to IOB tagging[15]: assign 'U' to single character token, 'I' to character in non-single character token, 'T' to begging of non-single character token, 'E' to ending character of non-single character token, 'O' to characters not included any token.

Example of IOB tagging :

I		I	o	v	e		P	y	t	h	o	n
U	O	T	I	I	E	O	T	I	I	I	I	E

As such, this method will measure tagging precision and recall for each label and also the weighted average score for overall performance.

After close comparison, We choose the first metric as our main evaluation metric because

We compared two metrics here as found that 1)The first metric is more intuitive to humans and in accord with ultimate goal of tokenization, which is to recognize correct tokens. 2) In second metric, 'I' will appear much more frequently than other tags as most tokens are longer than 3 characters. This creates an imbalance in data and encourage our model to predict everything as 'I'. If we remove 'I' in metrics, there is still imbalance between 'T,E' tag and 'U' tag.

Due to those reasons We choose the first metric as our main evaluation metric to give overall scores.

3.4 Regular Expression Tokenizer

3.4.1 Principle

Tokenization can be considered as a preprocessing process for Natural Language Processing. It is a process that divides a text into different tokens for further processing. To do the tokenization, we must first get token definition. Then, every word following token definition can be considered as a token. As majority of token definitions can be formulated as rules[4], using regular expression to match tokens is a viable approach.

3.4.2 Text Tokenizer

The text tokenizer is used to obtain text tokens in the file. It first reads the file line by line. Then it uses regular expression matching to divide text in the line into word tokens. After that It will use <t> tag to highlight found text tokens in the original file.

For normal word tokens, tokenizer simply split on whitespace and strip the punctuation following or prior to the word. For special pronoun, it stores a variable lastword. If the lastword is capitalized and has no following punctuation and current word is also capitalized, it will be considered as a special pronoun and grouped together. The special pronoun ends when the current word is not capitalized. For formulas, it will group the left and right token to the "=" sign together. For special nouns like Ph.D., e.g., it will not split on "." when there are at least two tokens of this type in one word.

The text tokenizer works fine with most of the cases. However, in some situations, it may tokenize words into a special noun wrongly. For example, in the sentence: "Show Mary the code." The text tokenizer will group "Show Mary" into a special pronoun, which is obviously wrong. However, since solving this problem requires a lot of rules and information such as pos tagging, we leave this problem as it is now.

Besides, the text tokenizer will encounter difficulties when dealing with decimal points and complicate math formula. It may divide the number/formula into multiple text tokens. Since there are a lot of edge cases in the formula, we leave this problem unsolved.

Though we leave these two problems unsolved, the accuracy of the regular expression text tokenizer is still pretty high.

3.4.3 Code Tokenizer

Our code tokenizer utilizes python module *tokenize*, which provides a lexical scanner and outputs preliminary tokens. Prior to feeding codes into python's tokenize module, we will firstly detect several tokens with pattern including file paths, '1.3.4', 'python3-tk', '<' liked. The regular expression involved are:

Pattern	Regular Expression
File Path	`(\w\:)?((\\[^,;\'\" ]+)+ |(/[^,;\'\" ]+)+)(\.\w+)?`
'1.3.4' liked	`\d+(\.\d+)+`
'python3-tk' liked	`\d\w]+(-[\d\w]+)+")`
'<' liked	`\d\w]+(-[\d\w]+)+")`

Thereafter, we will substitute each token with a special identification number in original text, to prevent python tokenize module from editing those tokens. In the end, those ids will be translated back to tokens we firstly detected.

For tokens detected by tokenize python module, tokens will be assigned a type. we will only accept tokens with meaningful types which include variable name, number, error token, operation. Tokens only consisting of special characters will be discarded. For comments and strings blocks, as we found that people like to write codes in comments and code tokenizer has considerably good performance on normal text as well, We will escape the comment token '#' and tokenize the content of comments with the code tokenizer.

3.4.4 Result Analysis

We evaluated the regular expression tokenizer results based on the 100 annotated posts. The precision, recall and f1

score of the model are:

Precision	Recall	f1 score
0.9578	0.9729	0.9653

The tokenizer is not 100% correct. Here are some false positive examples:

Raw Text	False Positive
Windows Vista	<t>Windows</t> <t>Vista</t>
C:\Program Files\Intel\iCLS	<c>C:\Prgram</c> <c>Files</c> <c>$\Intel\iCLS</c>

There are also some false negative examples:

Raw Text	False Negative
dir.py	<c>dir.py</c>
C:\Program Files\Intel\iCLS	<c>C:\Program Files\Intel\iCLS</c>

From these examples, we observed errors mainly come from three parts. The first kind of error occurs when tokenizing strings with first letter capitalized. Although we have applied some regular expression to decide whether to split strings or not, it did not cover all case and cannot determine whether these strings are related. The second kind of error comes from handling path. The tokenizer split strings by backslash but this is wrong when applying on path. The third kind of error comes from python file name. The tokenizer did not know whether the dot is a period or a symbol in file name.

3.5 Conditional Random Fields Tokenizer

3.5.1 HMMs

Conditional random fields (CRF) is a probabilistic model for predicting sequence label data[7]. Before CRF was introduced, Hidden Markov Chains (HMMs) had been widely applied to linguistic tasks including topic segmentation, part-of-speech (POS) tagging, etc[9]. However, HMMs relies on Markov assumption and output independence assumption which restrict it from modeling dependence on long range observations or multiple interacting features.

3.5.2 MEMMs

To compensate such limitation, Maximum entropy Markov models(MEMMs) was introduced[10]. MEMMs assign each source state an exponential model which takes observation features as inputs and outputs probability distribution of next state. MEMMS are usually trained by iterative scaling method, a general solver for maximum entropy methods. However, MEMMs possesses a weakness called the label bias problem, which means the transitions leaving a given state compete only against each other, rather than against all other transitions in the model. To illustrate why this is a problem, consider figure 6 listed below.

Assume we use this finite state model to match word 'rib'

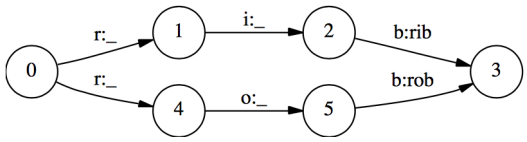

Figure 6: label bias problem[3]

and 'rob' and the left node denotes the starting state. Each edge denotes *observation : label*. If we are given a word 'rib', the first observation will be 'r' and paths 0-4 and 0-1 are matched. Thus State 0 will pass its probability mass to state 1 and state 4, supposing each is 50%. The next observation is 'i' and the upper path can be matched. Thus state 1 will pass all its probability mass to state 2. However, as state 4 only has one path, model will have no other choice but pass all probability mass of state 4 to state 5. Now state 5 has the probability of 50% as well. Such argument holds for third observation. As a result, this model will output 'rib', 'rob' with equal probability score, from observation 'rib'. This example reveals the bias of MEMMs to favor state with less outgoing states, regardless of observations.

3.5.3 CRF Principle

CRF is an improved model over HMMs and MEMMS. Compared to HMMs, CRF, as a non-generative model, can have overlapping non-independent features and do not have strong assumptions on sequence data. Compared to MEMMs, CRF does not have per state normalization and introduced global competition among all states by having a single exponential model for the joint probability of the entire sequence of labels given the observation sequence. Thus CRF eliminate the label bias problem.

Formally, consider X random variable over data sequences to be labeled, and consider Y a random variable over corresponding label sequences(in our case IOB tags).

CRF definition:
Let G = (V,E) be a graph such that $Y = (Y_v)_{v \in V}$ *, so that Y is indexed by the vertices of G. Then (X,Y) is a conditional random field in case, when conditioned on X, the random variables* Y_v *obey the Markov property with respect to the graph:* $p(Y_v|X, Y_{w,w \neq v}) = p(Y_v|X, Y_{w,w \sim v})$*,where w* $\sim$ *v means that w and v are neighbors in G.*

As CRF is a maximum entropy model, it has convex optimization space and thus is guaranteed to converge to global optimum.

In practice, people have applied CRF for NLP tasks such as sentence and tokens spiting[20], named entity recognition(NER)[11], POS taggin[6], etc.

3.5.4 Formulation and Features

As introduced in section 3.3, we formulate the task of tokenization to classification each character with an IOB tag. In CRF context, X is the sequence of characters of the given text, Y is the output IOB label. We define several feature functions including the lower case character, boolean indica-

tor of whether this character is uppercase, digit, character. Each Y is associated with 11 X(5 before and 5 after), if available.

3.5.5 *Results Analysis*

We use python sklearn wrapper of CRFuite[13] in our experiment. It implements Linear-chain (first-order Markov) CRF and Elastic Net (L1 + L2) regularization. We have two settings of training: 1) train one CRF across for both text tokens and code tokens 2) train two CRF with respectively text data and code data and predict text and code using corresponding CRF. We perform hyper parameter tuning to find the L1, L2 normalization parameters for our task. The

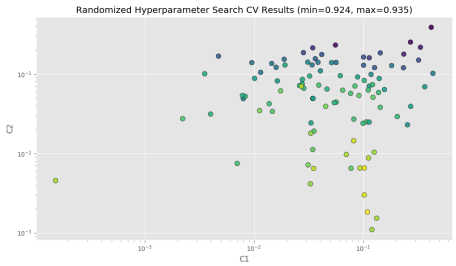

Figure 7: hyper parameter tuning

figure above showed the hyper parameter space searching on single CRF set-up. We perform same process for 2 CRF set-up as well.

We using the annotated 50 posts and 50 answers as training data. We perform 5-fold cross validation for each experiment.

Setting	precision	recall	f1
One CRF	0.9404	0.9225	0.9313
One CRF(hyper tuned)	0.9401	0.9285	0.9342
Two CRF	0.9424	0.946	0.9441
Two CRF(hyper tuned)	0.9479	0.9481	**0.948**

As shown above, using dedicated CRF model for code and text and hyper parameter tuning both slightly increase performance. Finally CRF model achieves F1 score around 0.94 in 5-fold cross validation.

To study the model performance, we select several interesting predicated results. Due to the limitation of space, we use {} to denote token border here.

Raw Text	Ground Truth	Prediction
e.g.	{e.g.}	{e.g}.
/usr/.../al.py	{/usr/.../al.py}	{/usr/.../al}{.}{.py}
1.5.6	{1.5.6}	{1.5}{.}{6}
#holds	#{holds}	{#holds}

The ground truths are false negatives and predictions above are false positives. These errors above indicate CRF was not fully able to capture all rules in token definition like {1.5.6} and {filePath}. Besides, it is hard to equip CRF model with knowledge on common tokens like {e.g.}.

However, with the consideration that our training data is small compared to other dataset like [2] or [14], we argue that our CRF still obtained remarkable performance on tokenization. Listed below are correct results predicted by CRF. As we can see, it learned to escape special character like `{` / `,` / `>>>` and `[`. It also learned to concatenate function call with the left parentheses as a single token.

[{10}, {2}]
>>> {a}
`['{__name__}']`
{__init__(){self})

3.6 Performance & Analysis

We evaluated two tokenizers by the first evaluation metrics mentioned in 3.3. The performance are:

	Regular Expression	CRF
precision	0.966	0.9457
recall	0.981	0.945
f1	0.974	0.9453

It shows regular expression tokenizer performs better than CRF tokenizer. We think this is because we can define specific rules in regular expression tokenizer for specific case, however, CRF tokenizer is a statistic method and cannot add specific rules. Therefore, we decide to apply regular expression in our tokenizer.

4. FURTHER ANALYSIS

4.1 Non-Standard Tokens

Irregular token is a vogue definition. Given the context of Stack Overflow and our token definition, we think irregular tokens should satisfy : *(a) it must be a token; (b) it is not commonly used in daily life, thus it will not include numbers; and (c) there is not entry for it in Oxford Dictionary.* Based on that, we observe the following non-standard English tokens using our tokenizer (in the form of token followed by its frequency):

('.', 22768) ('=', 14723) ('-', 8969) ('def', 3230) ('%', 2584) ('+', 2124) ('>', 1601) ('print(', 1500) ('==', 1457) ('*', 1290) ('<', 1266) ('foo', 1033) ('sys', 1027) ('py', 757) ('!', 736) ('$', 723) ('|', 716) ('len(', 679) ('**', 624) ('datetime', 614)

These 20 tokens are obtained from top 150 most frequent tokens (including both text and code). Most of the irregular tokens are special characters (13), followed by keywords (4), module name (2), and "foo". Its distribution exhibits long-tail effect, where "." tops the overall most frequent tokens with 22768 times. We think it might be due to its frequent usage in both text and code section. While the second top irregular is "=", which is neither surprising for its usage in assignment or evaluation expressions.

Another interesting fact we have observed is that *def*,*print*, and *len* are frequently used keywords.

4.2 Part-Of-Speech Tagging on Non-Standard Tokens

We tagged 10 sentences that contain irregular tokens using Hidden Markov Model (nltk.tag.hmm[8]), and the following are some examples:

Example 4.2 (a):

PRP	VBP	VBG	PRP	VBZ	JJ	TO	VB	IN
I	am	hoping	it	is	possible	to	do	without

VBG IN NN
tinkering with sys.path :

Example 4.2 (b):

NN (UH JJ (NNP))
plt.plot (x ; np.sin (x**2))

We observe that, because our tokenizer can distinguish a file path or filename, the POS-tagger can then correctly identify the tag for them. For example, "sys.path" in Example a is a noun. Also, we can identify the math expressions like "x**2", "plt.plot", "np.sin", we produces more meaningful result when compared to using original nltk tokens. However, the POS-tagger still cannot handle tagging for irregular word well.

5. APPLICATION

Tagging is an important feature in many websites such as news websites and online forum. It can help users to find certain topic efficiently. Tags are usually manually attached to articles by editors. However, as millions of articles are published online everyday, tagging consumes a lot of efforts. There are also a lot of articles have never been assigned any tag. In this case, we have built a auto tagging application on the question posts in our dataset based on our regular expression tokenizer. The basic idea is to find the most N important keywords in a post.

5.1 Methodology

Our application mainly contains two steps, which are tokenization and term frequency inverse document frequency calculation.

5.1.1 Tokenization

The dataset is first split into two parts, which are text part and code part. Then we tokenized these two parts by our text tokenizer and code tokenizer separately. After several experiments on code tokenization, we think only function name can provide useful information for users. Therefore, function name tokens are remained in code tokens while others are removed. Stop words are also been removed in both parts.

5.1.2 Term Frequency Inverse Document Frequency

After tokenization, we applied term frequency inverse document frequency(tf-idf) method on these tokens to find the most N important keywords. It is a statistic method that is intended to reflect the importance of a word to a document in a collection or corpus.[16] Tf-idf is the product of two statistics, term frequency and inverse document frequency. Term frequency tf(t,d), is the number of term t occurs in document d divided by the number of all terms occurs in document d, which is

$$tf(t,d) = \frac{n(t,d)}{\sum_{t' \in_d} n(t',d)}$$

Inverse document frequency idf(t,D), is obtained by dividing the total number of documents by the number of documents containing the term, and then taking the logarithm of that quotient. The formula is:

$$idf(t,D) = log\frac{|D|}{|\{d \in_D : t \in_d\}|}$$

Then tf-idf is calculated as

$$tfidf(t,d,D) = tf(t,d) * idf(t,D)$$

Tf-idf score can be increased with a high term frequency(in the given document) and a low document frequency of the term in the whole collection of documents.

5.1.3 Selection

To let users know the basic idea of a question post, we select top three high tf-idf score tokens from text part and one function name token from code part. If there is no function name in code part, we replace it with a text token.

5.2 Performance

Here are the first 6 question posts' tags get by our application:

Table 1: Top 4 tags of first 6 question posts

post id	tag 1	tag 2	tag 3	tag 4
3061	function's	go	figured	bar(
19151	Build	Basic Python	iterative	iterator
22617	pm	numbers	Hours	str(
22676	download	utility	MP3	file
25665	PDF	text	converting	modules
35988	C-like	structures	tired	MyStruct(

From these tags, we may know the the first question is asking about function, the second question is asking about Python Iterator, and the third question is asking about dates or numbers and so on and so forth. To examine it, we checked first three original question posts from our database:

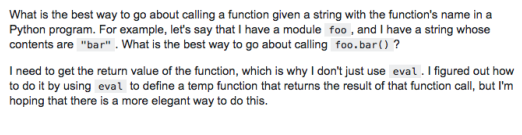

Figure 8: question 3061

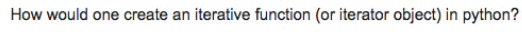

Figure 9: question 19151

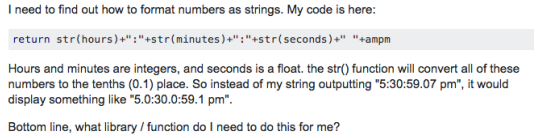

Figure 10: question 22617

As the above figures showed, our assumptions based on generated tags are quite meaningful.

6. ACKNOWLEDGMENTS

7. APPENDIX A

Table for Stems of Top 20 Most Frequent Word

Section	Subsection	Contributors
Dataset Collection	Dataset Collection	Liu Hualin
	Dataset Statistics & Visualization	Liu Hualin
Data Analysis	Stemming	Shi Ziji
	POS Tagging	Shi Ziji
Tokenization	Token Definition	Chen Hailin, Deng Yue, Liu Hualin, Shi Ziji
	Token Annotation	Chen Hailin, Deng Yue, Liu Hualin, Shi Ziji
	Evaluation Metrics	Chen Hailin, Deng Yue
	Regular Expression Tokenizer	Chen Hailin, Liu Hualin
	CRF Tokenizer	Chen Hailin, Deng Yue
Further Analysis	Non-standard Tokens	Shi Ziji
	POS Tagging on Non-standard Tokens	Shi Ziji
Application	Application	Deng Yue
Report	Report	Chen Hailin, Deng Yue, Liu Hualin, Shi Ziji

Stem	Morpheme
like	like Like likely liked likes
string	string strings String Strings STRING
file	file files File Files FILE filed FIle
get	get getting gets Getting GET Get Gets
list	list List listing Lists lists listed listings LIST Listing
use	use using Using useful used uses Use Useful Uses usefully USE usefulness Used USe USING
you	you You YOU
function	function functions Function functional functionality functioning functionally Functions Functional FUNCTION FUNCTIONALLY
i	I i
want	want wanted wants wanting Wanting Want
one	one One ones ONE
method	method methods METHOD Methods Method methodically
need	need needed needs Need Needed needing
code	coding codes coded Code code Coding Codes CODE
way	way ways Way WAY
exampl	Example examples Examples EXAMPLE example EXAMPLES
instal	installed install installation installing Installing Install installer installations installers installs Installing Installation Installers INSTALLED install Installed installable Instal instaled instalation Installer
object	objects Object object Object's objected OBJECT objective
python	Python python pythonic Pythonic pythonic pythoners pythons PYTHON Pythons Python's pythonators Pythonically Pythoneers Pythoning
work	working work works WORKED worked Work Works Worked Working WORK

8. REFERENCES

[1] Stack exchange data dump. `https://archive.org/details/stackexchange`, 2017. Accessed 31-August-2017.

[2] J. Bos, V. Basile, K. Evang, N. J. Venhuizen, and J. Bjerva. The groningen meaning bank. In *Handbook of Linguistic Annotation*, pages 463–496. Springer, 2017.

[3] L. BOTTOU. *Une approche theorique de l'apprentissage connexionniste et applications a la reconnaissance de la parole*. PhD thesis, 1991.

[4] R. Dridan and S. Oepen. Tokenization: returning to a long solved problem a survey, contrastive experiment, recommendations, and toolkit. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics: Short Papers-Volume 2*, pages 378–382. Association for Computational Linguistics, 2012.

[5] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[6] T. Kudo, K. Yamamoto, and Y. Matsumoto. Applying conditional random fields to japanese morphological analysis. In *EMNLP*, volume 4, pages 230–237, 2004.

[7] J. Lafferty, A. McCallum, and F. C. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. 2001.

[8] E. Loper and S. Bird. Natural language processing toolkit, 2002. `http://nltk.sourceforge.net/`.

[9] C. D. Manning, H. Schütze, et al. *Foundations of statistical natural language processing*, volume 999. MIT Press, 1999.

[10] A. McCallum, D. Freitag, and F. C. Pereira. Maximum entropy markov models for information extraction and segmentation. In *Icml*, volume 17, pages 591–598, 2000.

[11] A. McCallum and W. Li. Early results for named entity recognition with conditional random fields, feature induction and web-enhanced lexicons. In *Proceedings of the seventh conference on Natural language learning at HLT-NAACL 2003-Volume 4*, pages 188–191. Association for Computational Linguistics, 2003.

[12] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example?: A study of programming q&a in stackoverflow. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 25–34. IEEE, 2012.

[13] N. Okazaki. Crfsuite: a fast implementation of conditional random fields (crfs), 2007.

[14] R. Ordelman, F. Jong, A. Hessen, and H. Hondorp. Twnc: a multifaceted dutch news corpus. *ELRA Newsletter*, 12(3-4), 2007.

[15] D. Pierce and C. Cardie. Limitations of co-training for natural language learning from large datasets. In *Proceedings of the 2001 Conference on Empirical Methods in Natural Language Processing*, pages 1–9, 2001.

[16] J. Ramos. Using tf-idf to determine word relevance in document queries. 01 2003.

[17] L. Richardson. Beautifulsoup. `https://www.crummy.com/software/BeautifulSoup/`, 2004–. [Online].

[18] C. Stanley and M. D. Byrne. Predicting tags for stackoverflow posts. In *Proceedings of ICCM*, volume 2013, 2013.

[19] X. Sun, X. Liu, J. Hu, and J. Zhu. Empirical studies on the nlp techniques for source code data preprocessing. In *Proceedings of the 2014 3rd International Workshop on Evidential Assessment of Software Technologies*, pages 32–39. ACM, 2014.

[20] K. Tomanek, J. Wermter, and U. Hahn. Sentence and token splitting based on conditional random fields. In *Proceedings of the 10th Conference of the Pacific Association for Computational Linguistics*, pages 49–57, 2007.

[21] S. Wang, D. Lo, and L. Jiang. An empirical study on developer interactions in stackoverflow. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1019–1024. ACM, 2013.