CZ4045 Natural Language Processing Report

Instructor: Dr.Sun Aixin

Zhang Wanlu
U1420638G
zh0012lu@e.ntu.edu.sg

Shao Yiyang
U1420722G
shao0035@e.ntu.edu.sg

Zhao Qingmei
U1422368A
zhao0219@e.ntu.edu.sg

Wang Yiqi
U1520550K
wa0002qi@e.ntu.edu.sg

1. INTRODUCTION

This project is aim to walk through the main components of an end-to-end NLP application and try to solve challenges we encounter. In this project, we use Pandas[1] for data storage and processing. Some functions provided with NLTK[5] help us further tokenize, annotate and tag the dataset. At last, we use Tensorflow[2] to implement a simple Recurrent Neural Network[6] to realize our application for generating new Stackoverflow questions. Four of the team members contribute equally to this project.

2. DATASET COLLECTION

2.1 Raw Data Collection

Since we will use data from Stack Overflow, we get a data dump Posts.mxl from https://archive. org/details/stackexchange. Posts.xml is 55 GB after unzip which contains millions of posts including both question posts and answer posts.

Each post in Posts.xml has an unique ID and the corresponding PostTypeId. When PostTypeId==1, it is a question post. When PostTypeId==2, it is an answer post. The answer post contains a tag ParentId which indicates the ID of question post it is answering. The question post contains a tag AcceptedAnswerId which indicates the ID of accepted answer of the question.

```
<row
    Id="1714731"
    PostTypeId="2"
    ParentId="1714584"
    CreationDate="2009-11-11T11:55:11
    .210"
    Score="11"
    Body="&lt;p&gt;You're using 8c,
    which is the c compiler. 8g will
    compile go,
        and 8l will link.&lt;/p&gt;&#
    xA;"
    OwnerUserId="208625"
    LastEditorDisplayName="user181548"
    LastEditDate="2009-11-11T11:59:08
    .473"
    LastActivityDate="2009-11-11
    T11:59:08.473"
    CommentCount="3" />
```

Listing 1: An example of answer post

2.2 Collect Dataset from Raw Data

The Programming Language we have decided for thread selection is Go Programming Language.

The dataset collection mainly has 3 steps. Firstly, we collected all the posts which has the 'go' tag. Hence, all the selected threads are on Go Programming Language. Secondly, we go through all the posts again to find the code part in the post of the Go Programming Language threads for further process. The code part has been tagged with <code> </code> in body of the post, . Thirdly, randomly select 700 threads which have at least 2 posts (one question and at least one answer). The 3 steps mentioned above are shown as The following pseudo code Listing 2.2.

To save and process the data, we have used the Pandas DataFrame from The Pandas Community [1].

```
# step one (parse_xml_to_csv.py)
question_ids = []
for post in Posts.xml
    if the post is question post:
        if it has 'go' tag:
            if answer count < 1:
                continue
            save the question post
            add the ID to question_ids
    if the post is answer post:
        if its parentID in
    questions_ids:
            save the answer post
save question posts to question.csv
save answer posts to answers.csv

# step two (find_code.py)
code_beginning = '&lt;code&gt;'
code_ending = '&lt;/code&gt;'
for post in Posts.xml:
    if post is in Go Programming
    Language thread:
        if it has code content
            save the code part
save the code part to code.csv

# step three (get_700_threads.py)
random select 700 questions
selected answers which the parentID is
     in 700_question_id_list
save selected questions and answers
```

Listing 2: Pseudo Code of Dataset Collection

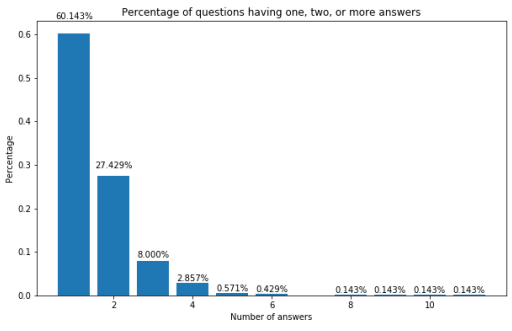

Figure 1: Distribution of Number of Answers

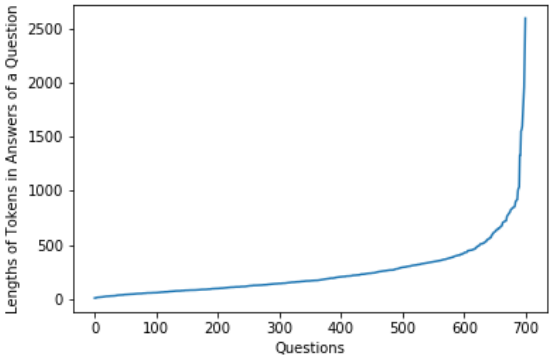

Figure 2: Lengths of Tokens in Answers of a Question

2.3 Dataset Analysis

The following section shows some basic data analysis for our dataset which is obtained from the raw data.

- Number of questions: 700
- Number of answers: 1129
- The distribution of the answers are shown in Figure 1.

For Lengths of Tokens in Answers of a Question (Using TweetTokenizer from nltk): 95 percentile is 714. Min is 8. Max is 2601. As shown in Figure 2.

For Lengths of Lengths of Tokens in an Question (Using TweetTokenizer from nltk): 95 percentile is 643. Min is 20. Max is 2254. As shown in Figure 3.

For Lengths of Lengths of Tokens in a Post (Using TweetTokenizer from nltk): 95 percentile is 1111. Min is 50. Max is 3381. As shown in Figure 4.

3. DATASET ANALYSIS

3.1 Stemming

For this part, we first use a simple tokenizer to tokenize the text, then filter out the stop words by referring to the dictionary provided by NLTK. Here we

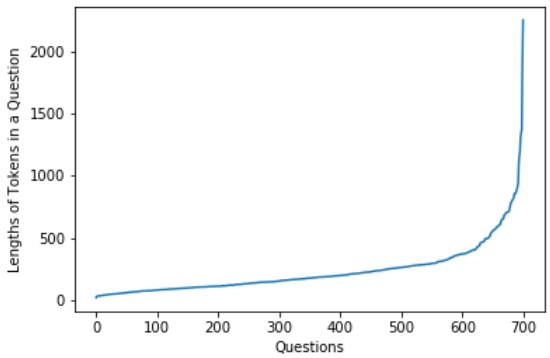

Figure 3: Lengths of Lengths of Tokens in an Question

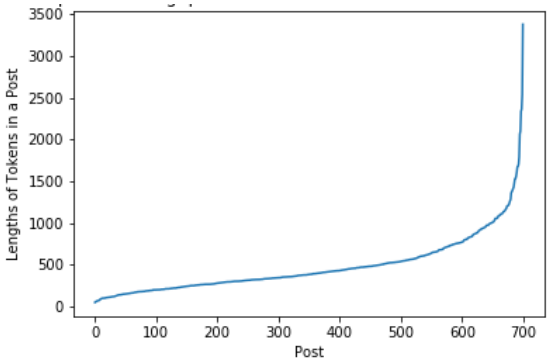

Figure 4: Lengths of Lengths of Tokens in a Post

also do a simple filtering on the digits since we focus more on real words. After tokenization, a simple stemming procedure is conducted by using NTLK built-in stemmer *nltk.SnowballStemmer()*. Create hash maps to keep track of all the occurrence of words or stems, then identify the top-20 most frequent ones. For each stem from the top-20, we list down it's original word from which the stem is reached.

Here are some examples for stem-origin mapping.

```
'go': {'go', 'going'}
'string': {'string', 'strings'}
'func': {'func', 'funcs'}
```

The result shows that golang keywords tend to appear more frequently in the files, which is quite intuitive since all the posts are related to golang. Some words such like "c" is highly possible to be a symbol for the programming language C. The rest, for example "http" and "json", are commonly appear in a Stack Overflow post.

Most words appear in the both top-20-stems and top-20-words, but just in different representations since one is performed stemming. For example, "package", "value" become "packag", "valu". However, there are also some words that are uniquely existing in just one of the lists. This might because after stemming on the dataset, the total number of types decreases and for each type, the shown-up frequency might increase a

bit, thus, leads to high occurrence of some stems.

For detailed result for this part, please refer to the Appendix A.

3.2 POS Tagging

We randomly select 10 sentences from the dataset and use nltk pos-tagger to do POS tagging on each of them. From the result we can see that nltk pos-tagger is good at dealing with natural language. For example, it is able to separate "'m" from "I'm" and tags the phrase as ['PRP', 'VBP']. However, it cannot handle context-dependent tokens, given one of the selected sentences as an example:

"Use the 'flag' package: http://golang.org/pkg/flag/."

The generated result is
['VB', 'DT', 'NN', 'POS', 'NN', ':', 'NN', ':', 'NN', '.'].

Due to the lack of knowledge about web address, the current tokenizer with split the web address "http://golang.org/pkg/flag/" into three parts: 'http', ':', and '//golang.org/pkg/flag/'. Then nltk pos-tagger will assign some meaningless tags to each token.

Things become even worse when coming to code tagging. Given a short statement:

"if x := 5; x == 5 {fmt.Printf(""Whee!\n"")}"

The pos-tagger will give output like:
['IN', 'VBN', ':', 'JJ', 'CD', ':', 'NNP', 'VBD', 'CD', '(', 'NN', '(', 'JJ', 'NNP', 'NNP', '.', 'NNP', 'NNP', ')', ')']

In the natural language, "if" is usually used as a preposition. However, here with the context of programming language, it is actually a keyword. In addition, it is hard for the pos-tagger to understand the concept of string without any further defined rules. That is why we need to define tokens and design our own tokenizer.

3.3 Token Definition

Based on our selected programming language, Go, we make the definition for special tokens and decide to treat each of the following as one token:

1. KEYWORD There are in total 25 keywords in Go. We pick them out and annotate as "KEYWORD".
2. OPERATOR & PUNCTUATION Operator is frequently used in the code session. We come up with different types of operators. For example, pointer operator'*', arithmetic operator'>', and assignment operator':=', each operator will be annotated based on its own type. Other punctuation such as ';', ',', will be directly annotated as 'PUNCTUATION'.
3. LITERAL As for literals, we define string literal, boolean literal and numerical literal. For numerical literals, we further divide them into decimal literal, hex literal and octal literal and floating literal. Hex literals usually start with 0x or 0X; octal literals start with 0. And there should always be a decimal point in a floating literal.
4. DIRECTORY During the process of annotation, we realize that there are a lot of directories included in both posts and answers. Therefore we define directory as a token, so it will not be split and is able to be referenced as a whole.
5. IDENTIFIER We use identifier to represent variable name, class name and function name appearing in the code. The definition of identifier is based on the naming convention in Go, which is always a combination of alphanumerical characters and underscores.
6. FUNCTION_CALL We define function call as a series of characters which always start with an identifier, appending with a left bracket, then followed by a serious of characters. But mostly importantly, end with a right bracket. Some commonly used functions are make(), append(), etc.
7. RESERVED Most data types in Go are defined by users. However, there are still some predefined data types which cannot be used as anything else. Some example could be 'bool', 'int', and 'string'.
8. COMMENT Last but not least, we also realize it is necessary to annotate comments separately. The reason is that comment is an important component in a program. There are three common forms of comment in Go: "//", "/*", and "*".

4. TOKENIZER

In this part we implement a tokenizer and use it to tokenize all the sentences in our dataset.

Since the posts from Stack Overflow normally will contain code blocks, which are wrapped by html tag **<code></code>**, we filter out all the code for the posts in our dataset for special treatment. When implementing this part, we assume everything in a code block is written in golang. Although this may not always hold, we realize that it does not affect the final result that much. The code tokenizer is designed specifically for processing the code block, and the natural language tokenizer is for dealing with other parts of a post. Both tokenizers are implemented using regular expressions.

```
for index, value in enumerate(parts):
    if code_or_not[index]:
        dic, post = tokenize_code(
    value, dic, post)
    else:
        token =
    tokenize_natrual_language(value)
```

4.1 Code Tokenizer

When pass a code block to the tokenizer, it will process the text line by line. The advantage for doing so is that the code is executed in such an order and each line is carrying some meaning. But the shortcomes is that it cannot handle the situation properly where one command is written in several lines, such as a comment block. But in most of the cases, our method functions well. Based on the syntax of Go language, we write several regular expression patterns for splitting the code in a manner of what we defined previously. For example,

```
function_call = r'(?:\w+\.\w+\s*\(.*\)
    )|(?:\w+\s*\(.*\))'
string_literal = r'(?:\"\s*.*?\n?\")
    |(?:\'\s*.*?\n?\')'
```

Thus, when a function call or a string literal is detected, it will be counted as one token. Becasue the overall pattern consists of many regular expressions, we After discussion, we decide to treat comment as natural language such that when encounter a comment symbol //, the rest of the line will be passed to Natural Language Tokenizer for further tokenization.

4.2 Natural Language Tokenizer

The tokenizer for analyzing natural language takes in several steps which are listed as follow.

- Split the text into tokens by spaces. Since we wish to preserve the most information of a single token, we use space to do splitting first, such that a meaningful token won't be divided into parts. For example, if we use the build-in word tokenizer provided by NLTK, then *https://stackoverflow.com/* would be tokenized as ['https', ':', '//stackoverflow.com/'], however, our tokenizer will treat this as one token. The disadvantage of approaching in such a way is that we could not preserve meaningful name entities properly. A detailed discussion on Name Entity Recognition is in the next section.

```
space_tokenizer = RegexpTokenizer(
    r'\S+')
tokens = space_tokenizer.tokenize(
    txt)
```

- Analyze each token and split it if there is any punctuation mark. For instance, "match," would be separated to "match" as well as ",". This step is to pick out all the punctuation marks and put it as a individual token. The **split_punc(token)** function is written for this step. In this function, the pairwise punctuation is checked such that a consecutive parenthesis "()" will stay as it is. Other detailed features for a better punctuation splitting is also implemented in the mentioned function.

- Check token abbreviation and split out. Based on our annotation, all the abbreviations are parted into its base word as well as prefix.

4.3 Extra Try-Out

We tried to fulfill a name-entity-recognition feature in our tokenizer but we failed to do so. It might be due to our regular expression is too simple to handle all the case, but we also think it could be due to the limitation of regular expression. Probably use a CRF to detect name eneities will provide a better result. Here is a general idea on how we implement this feature, however, we didn't include it in our final submission.

Based on some analysis, we realize that in most of the cases, the name entity consists of several consecutive words starting with capital letters. So we try to use regular expression to find consecutive words based on this observation. Then analyze the POS tag of each word. Try to filter out those that are impossible to be in a name entity. However, this simple implementation cannot handle the case where lower-case conjunction exists in the entity name, and it's really dangerous to filter out based on human assumption.

We also tried to use the nltk.ne_chunck and nlkt.Tree to detect all name entities in a sentence. This generally provides a much better result than the first one. But when we test the ne_chunk with name entity "Natrual Language Processing", it separate into "Natrual Language" and "Processing" although each single word is detected as a NNP.

4.4 Result and Analysis

After running our tokenizer, the result for 100 post will be stored in 100 different files. The tester.py file is used to calculate the accuracy of the proposed tokenizer. Each time the tester will read in a pair of csv file from the generated result and the labeled one, then compare between these two and print out the wrong tokens as well as the number of mistakes which are counted based on the ground truth annotation. Based on the intuitive way to calculate accuracy, as shown below, our tokenizer provides impressive result and reaches an accuracy of around **97.6%**.

$$\frac{TotalNumOfTokens - WrongNumOfTokens}{TotalNumOfTokens}$$

With the analysis on the test report, here point out some scenarios where our tokenizer could make a mistake:

- When we do annotation previously, we use nltk.word_tokenize() to help split some normal sentence. But it seems that this word tokenizer will convert a pair of double quote into something strange. Here is an example,

```
import nltk
a='''"Use the 'flag' package"'''
nltk.word\_tokenize(a)

>>> ['``', 'Use', 'the', "'flag",
    "'", 'package', "''"]
```

Note that the two double quotes are parsed to really different symbols. However, our tokenizer

will provide a more normal result. Due to this problem, there are many 'wrong' items in our generated tokens since we use the annotation as ground truth. So if count this case as correct, the accuracy could be even higher.

```
a='''"Use the 'flag' package"'''
tokenize_natural_language(a)

>>> ['"', 'Use', 'the', "'flag'",
    'package', '"']
```

- Since the tokenizer is implemented in a way where space is really important, if there is no space between two sentence, the tokenizer would fail to identify where to break the word. For instance, ['normally.What'], obviously the dot in between stands for a period, but it's treated as one token in the result. This might be handled in a more perfect way but under the surroundings of Stack Overflow, it's really hard to detect whether a dot sign is a period or stands for something else.

- If the punctuation mark is a kind of pairwide mark, such as (),{},[], and both left and right part appear in a single token, the tokenizer will not break them. Here are some examples,

```
Generated: ['"work"']
Annotated: ['"', 'work', '"']
Generated: ['(IPv6)']
Annotated: ['(', 'IPv6', ')']
```

- Due to imperfect processing of the punctuation mark, sometimes the tokenizer could not take out the mark from the token which leads to a mistake. For example for the generated one, ['#2:'] is treated as one whole, while the annotated one separates it into three ['#', '2', ':']

Apart from the previously mentioned problems, the tokenizer itself still has many defects that could be improved by adding more complex rules or using external package.

5. FURTHER ANALYSIS

5.1 Irregular Tokens

After tokenizing the dataset with our own tokenizer, we try to find the top-20 most frequent tokens which are not standard English. A detailed result is shown in Appendix Section B. We have decided to ignore decimal literals, floating literals and punctuation. They do appear on the top of the list, however, we think those tokens cannot well reflect the characteristic of our dataset.

The remaining top most frequent tokens are Go keywords, for example, "func", "struct", "'fmt"', following by the reserved data types, "bool", "float64". File format such like "xml", "json", "bson" also appear in the token list. We have to point out that, the 'func' token has been appeared 1720 times in our dataset which means that almost all the threads has this irregular token, as there are only 700 threads in our dataset.

In addition, there are some Go-related professional technical terms, for example, 'goroutine', 'ResponseWriter', 'AnswerJSON', 'Golang', 'float64', 'GOPATH'. Some commonly used function like 'len' also show up in the final result.

Note that abbreviation tokens also appear quite frequently in the dataset, for example, "'ve", "'s", "". However, considering that their original form is in English and some even have proper pos tagging, for example, 'POS' for "'s", we decide to exclude them from our final result.

From the result of top-20 irregular tokens, we can conclude that the language presentation used by Stack Overflow User contains not only standard English, but also code syntax keywords and special professional technical terms. Especially for the the code syntax keywords such as 'func' which appears 1720 times in our dataset.

5.2 POS Tagging

Given one example from the selected sentences: **"func generateToken() (string, error) {"** with 'func' irregular token.

If we use word_tokenize from NLTK to tokenize the above sentence, the result will be

['func', 'generateToken', '(', ')', '(', 'string', ',', 'error', ')', ''].

Since word_tokenize has no knowledge about "function call" and other coding syntax.

However, by using our tokenizer defined with the regular expression, we are able capture the tokens of code elements. The above sentence would be tokenized by our tokenizer into

['func', 'generateToken()', '(string,', 'error', ')', '', '('].

From the result produced by our tokenizer, token 'generateToken()' could be tagged as FUNCTION CALL and 'func' can be recognized as KEYWORD. However,by apply POS tagging based on our tokenized result, pos tagging ('func', 'NN', ('generateToken()', 'NN'), ('(string,', 'NNP'), ('error', 'NN'), (')', ')'), ('', '(') is produced. For the result produced by POS tagging, it is obvious that the POS tagging result does not catch the meaning of these code sentence at all.

6. APPLICATION

6.1 Function Description

Our application is to generate stack-overflow-style question title which is related to Go Programming Language.

Our goal is to generate the question tile which also can have the same characteristic of Stack Overflow question titles. Since the text content in Stack Overflow website is including standard English language, code, and also informal cyber speak, the task is more challenging and different than the conventional nature language text generation applications. Hence, we did not choose the traditional NLP text generation algorithm such as Shannon's Method (The one

we have learned during the lecture). Instead, we turn to the state-of-the-art algorithm recurrent neural network (RNN).

6.2 Implementation

We have studied several popular RNN models for generating text and compared them to select the best one for our application. We will discuss two main methods, Char-LSTM[6] and Word-LSTM[4], which are both based on the famous Char-RNN[3] model, in detail.

6.2.1 RNN

A recurrent neural network (RNN) is a class of artificial neural network where connections between units form a directed cycle. RNNs can use their internal memory to which captures information about what has been calculated so far and make use of sequential information. RNNs are called recurrent because they perform the same task for every element of a sequence, with the output being depended on the previous computations. Hence RNN is suitable for our task, to predict the next word in a sentence, it is better to use all the information of the words came before it.

6.2.2 LSTM

Long Short Term Memory networks (LSTM) is a very special kind of recurrent neural network. Sometimes, we only need to look at recent information to perform the present task. On the other hand, the information long time ago is needed for current information. Memorizing and forgetting date have been added into LSTM to solve these problems.

6.2.3 Training Dataset

We have been used all the question titles from our dataset as the training data. Total training data length is 1224220 characters. The vocabulary size is 113 characters (There are 113 unique characters).

6.2.4 Char-LSTM

The model simply constructed by 1 input layer, 2 hidden LSTM layer with 100 hidden neurons and finally, 1 output layer with softmax activation layer. Categorical_crossentropy is our loss function as the outputs are actually fitted in the 113 character categories. Optimizer for the model is RMSPropR. The Char-LSTM application is inspired by a blog from Trung Tran [6].

The shape of the input training data (X) for each batch to train the model is (batch_size=50, seq_size=50, vocab_size=113). The output (y) has the same shape as X, but each character in y is the next character corresponding to the character in X. For example, the training data is "Today I go ...". Hence, X[0] will be "Today I ..." and then the y[0] will be "oday I ..."

The network only consists of 2 hidden layers as more layers would increase the training time. Due to the time limit, parameters have not been tuned. As the nature of RNN which is sensitive to hyper-parameter, the performance can be greatly improved after parameter tuning. Hence we believe that if we have enough time to tune the network, it could have better performances.

Since this method generates the text character by character, the model may generate the word which is not even a English words nor readable. To avoid this problem, we have decided to implement the LSTM which is based on word.

6.2.5 Word-LSTM

The idea of this attemp is inspired by an open source project on Github[4]. It suggests a word-level input for the neural network. This will not only improve the correctness of generated words, but also contains some property information of each English word, such as forms, part of speeches, etc. It will also help the model gain a better idea of the overall sentence structure of the text. After some tuning and training, we get some satisfying results. The model could generate some readable questions. The basic sentence structure is mostly correct. And some phrases and expressions are actually related to Golang and programming specifically. However, some details such as different forms of a verb, single or plural form of a noun, may cause some errors. In addition, the overall meaning of the question may not always make sense. In our opinion, the result could be improved with more dataset and careful tuning of the network.

Some examples of generated questions is listed be.

- Can I I convert reflect.New's to stdout rabbitmq in golang?
- What GAE golang graceful block specific key As FlatBuffers slice?
- How to use malformed ways to upload protobuf to Linux?
- Is any no more ObjectId than multiple Html request than shared-memory HTTP connection?
- Is io.WriterTo's cross platform to improve Single type instance's values?

7. CONCLUSION

In this assignment we explore NLP by using data from Stack Overflow and various available Python packages for NLP tasks.

The first step is data preparation. During the process, we learn that data collection could actually be challenging but also crucial for us to proceed to any analysis. Next we analyze and annotate our dataset, where we realize the necessity to come up with detailed token definition to help the tokenizer better "understands" a sentence in the context of Stack Overflow. Then we design our own tokenizer and apply it to the dataset. Due to the limitation of regular expression, it is hard for the tokenizer to handle all possible situations in real life. However, the final result is still quite considerable.

At last, we create a nlp-related application by utilizing Recurrent Neural Network. We experimented

several models and tried to train the best result. The final result is quite exciting that the model is able to generate questions with correct English sentence structures, although the details of the content generated still need to be improved.

In conclusion, this assignment is fruitful and enjoyable for each member. And we gain not only hands on experiences but also deeper understanding about NLP.

References

[1] Pandas Community. *Python Data Analysis Library¶*. URL: http://pandas.pydata.org/.

[2] Tensorflow Community. *Tensorflow*. URL: https://www.tensorflow.org/.

[3] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. May 2015. URL: http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[4] Luser. *Multi-layer Recurrent Neural Networks (LSTM, RNN) for word-level language models in Python using TensorFlow*. June 2017. URL: https://github.com/hunkim/word-rnn-tensorflow.

[5] NLTK Team. *NLTK 3.2.5 Documentation*. URL: http://www.nltk.org/api/nltk.html#module-nltk.util.

[6] Trung Tran. *Creating A Text Generator Using Recurrent Neural Network*. Nov. 2016. URL: https://chunml.github.io/ChunML.github.io/project/Creating-Text-Generator-Using-Recurrent-Neural-Network/.

APPENDIX

A. STEMMING RESULT

```
top 20 stems
['go', 'string', 'type', 'use', 'func', 'err', 'fmt', 'return', 'main', 'valu',
    'http', 'packag', 'int', 'nil', 'println', 'struct', 'time', 'file', '
    function', 'error']

top 20 words
['go', 'string', 'func', 'err', 'type', 'fmt', 'main', 'http', 'return', 'nil',
    'println', 'struct', 'int', 'package', 'error', 'use', 'time', 'code', 'c',
    'json']

top 20 stem-origins
{'err': {'err', 'errs'},
 'error': {'error', 'errors'},
 'file': {'file', 'fileds', 'files', 'filing'},
 'fmt': {'fmt'},
 'func': {'func', 'funcs'},
 'function': {'function',
  'functional',
  'functionality',
  'functionally',
  'functions'},
 'go': {'go', 'going'},
 'http': {'http'},
 'int': {'int', 'ints'},
 'main': {'main', 'mainly', 'mains'},
 'nil': {'nil'},
 'packag': {'package', 'packages'},
 'println': {'println'},
 'return': {'return', 'returned', 'returning', 'returns'},
 'string': {'string', 'strings'},
 'struct': {'struct', 'structs'},
 'time': {'time', 'timed', 'times', 'timing'},
 'type': {'type', 'typed', 'types', 'typing'},
 'use': {'use', 'used', 'useful', 'uses', 'using'},
 'valu': {'value', 'valued', 'values'}}
```

B. IRREGULAR TOKENS

```
token:          frequency
func:           1720
struct:         864
json:           341
http:           340
"fmt":          313
chan:           202
bool:           127
ok:             117
app:            117
JSON:           109
goroutine:      105
golang:         103
float64:        101
goroutines:     97
ResponseWriter: 91
AnswerJSON:     89
GOPATH:         88
panic(err):     83
bson:           83
len:            82
```