Learning Comment Topics from Code

Jinman Zhao and Ainur Ainabekova
Department of Computer Science,
University of Wisconsin-Madison
{jzhao237, ainabekova}@wisc.edu

ABSTRACT

In this paper, we describe the method of learning comment topics against corresponding code fragments in order to generate topics for the source code that does not contain any documentation or comment. Topics are particular distributions over vocabularies that describe or reveal the meaning or intention of the code. The built system might be helpful for searching code techniques as well as for code classification. The major components of our system are: 1. topic models learned from comments that convert comments into topics; 2. code analyzer and encoder that extract information from codes; 3. recurrent neural networks that learn to predict topics from low-level instruction sequences. The results we obtained are somewhat promising, but cannot be stated as very successful. We spare some of our most interesting unexplored ideas in the future work part, which we believe could work well for the aimed task.

General Terms

Languages.

Keywords

Code understanding, code meaning, comment generation, natural language processing for software engineering, topic modeling, recurrent neural networks, low-level code.

1. INTRODUCTION

1.1 Motivation

It has been increasingly important to write comments for the source code while programming [1]. It is claimed that documentation and commenting improves software quality and speeds implementation, which makes it extremely critical to maintain comments consistent and comprehensive for large software projects. Nowadays, large software contain millions of lines of code, for example Linux operating system in 2012 contained approximately 15 million lines of code [2]. The 1.5 million lines of which were written just during couple of years. Looking at this statistics, it is seen how quickly the size of current large codebases increases. Therefore, it is highly important to have well-documented code in order to facilitate developers in building on top of the existing codebase and to decrease the chance of introducing bugs due to misunderstanding.

However, there are still common cases that good documentations or comments are not accessible or even do not exist. This can be a common scenario in small personal or research projects when the developers do not pay attention to or are not skilled at extensively documenting their code. A more serious scenario is when, given the fact that open source code is becoming very popular and developers realized how useful it might be to share the code and knowledge with each other or to work collaboratively on the same code, but it is still not necessarily helpful for the community of programmers to have shared code because of the issues related to commenting it. Firstly, some of the current accessible online code might not have any comments or description at all which is still not very useful if other people cannot easily understand it and use it. Secondly, for example while working on large software collaboratively programmers might have inconsistent documentation and for that case our system might help to proof check the existing comments or work on its refinement to have consistency in comments throughout the codebase. Thirdly, even consistent documentation might lack usefulness because of the fact that some comments might describe the meaning of the smaller piece of code without mentioning enough about how this code fits into the general picture of the rest of the code. Note that the second and the third problem could also trouble relatively stable developer groups such as big software companies.

1.2 Project Goal

Therefore, the initial goal for us was to build a system that can generate comments for code pieces that helps identify the underlying meaning. This can serve as automatic documentation for code without comments or auxiliaries for code with comments. For now, there is no such single work that deals with automatic comment generation for the code. In fact, this problem is not only hard to solve in terms of automatic code understanding but also from the point of natural language processing where it would be necessary to generate meaningful grammatically correct sentences. We realized by doing research that some related work has been done in this field but most of them are not trying to directly address the exact task. Still some more-or-less relevant pieces of work have helped us to approach our goal in a slightly different way.

We then switch our goal to a less ambitious one as to build a system that will generate some keywords or topics that still will give information about the underlying meaning of source code. Solving this problem might be considered as one of the steps in reaching our initial goal of automatic comment generation. We see the possible applications of this kind of system in code searching techniques as well as in code classification both of which can be in turn used for building smart codebases. Thus, this paper will focus on describing the model of our system and its implementation for this smaller problem.

2. PREVIOUS WORKS

We need to consider several areas that might be relevant to solving our problem. One of the parts of our problem was understanding code, in other words doing code analysis with identification of the specific features that give the information about its meaning.

There are various works that try to find similar source code. Previous works emphasize the role of low-level instructions in comparing two pieces of source code. In [3] authors identify programmer style from binary code features and find stylistic similarities in the code of different programmers. Their experiments prove that binary source code can preserve information about programmer style. In order to capture the property of the binary code, authors generate idioms, graphlets, supergraphlets and call graphlets, where idioms are short sequences of instructions and graphlets are sub-graphs of the control flow graph capturing the program structure. After generating all the features for the binary code authors use Support Vector Machine (SVM) as a classifier that, firstly, learns positively correlated features with the given programmer and then is able to predict programmer for the unlabeled data. The second their task is approached by using unsupervised learning method, clustering. Finally, authors obtained accuracy of 81% for their classifier for ten distinct programmers.

Similar work that uses binary code as a resource for comparing source codes and identifying same code compiled by different compilers is described in [4]. Authors created a search engine called Rendezvous that can search for code in a binary form. Again, in this paper we see that as one of the means of program abstraction authors use n-grams of the instructions mnemonics that are the textual descriptions of instructions compared to opcode. Other abstractions of the program are represented by control-flow subgraphs and data constants where data constants are 32-bit integers and strings that are not changed during the compilation of the program. They achieve F2 values of 0.867 and 0.830 on two different data sets.

Also, we need to mention that some of the works that we found were interesting to us in terms of how they view the problem of program analysis through the lens of natural language processing (NLP). These works inspired us to use similar approaches in our project. For example, in [5] authors suggest to reduce the problem of code completion to the problem of sentence completion in NLP, i.e. consider code as a natural language. They focus on the programs that use mainly APIs and they fill out the holes in the programs by finding the most relevant sequences of code, in other words by predicting probabilities of sentences. They use recurrent neural networks (RNN) and N-grams for these purposes. According to their experiments, in 90% the correct completion of the program appears in top 3 results that were suggested by their system.

Another work that shows how NLP techniques can be used in program analysis is described in [6]. In this paper authors reduce the problem of semantic relatedness between codes to the problem of semantic relatedness between their textual descriptions. For example, they implemented their system for automatic association of Java and Python code fragments. Again they consider the program as a natural language text, although they do apply some lightweight type analysis too. NLP techniques used in their method for measuring the similarity between descriptions are Latent Semantic Analysis (LSA) and term frequency and inverse document frequency (tf/idf) measures. Their method gives 80% precision and 75% recall while identifying similar code fragments.

Another part of our problem is comment analysis of the data that we have and would like to use for this project. There is one work that is very relevant to our goal in these terms. It is the only paper that deals directly with the comments of the source code. The tool iComment that identifies mismatches between code and comments is described in more detail in [7]. This tool is able to analyze comments that are divided into some topic categories like "lock-related" or "call-related" comments. The main idea behind it is to extract rules from comments that are making assumption and extract rules from the source code. Later, they construct decision tree in order to look for inconsistencies between extracted rules. They have particular set of possible rule schemes. For machine learning technique they use Decision Trees. Authors conducted the experiments on some large projects like Linux, Mozilla or Apache and detected 60 comment code inconsistencies, 33 new bugs and 27 bad comments. Some of them were reportedly already analyzed and confirmed by developers.

In [8] authors built a system that translates from C# to Java by applying statistical methods from NLP incorporated with the knowledge of target language's grammar structure. Similar to latter, [9] aims to give a line-to-line translation from Python code to its explanation. [10] uses convolutional neural networks coupled with attention mechanisms to automatically generate suggestions for Java method names.

After considering all of the mentioned previous works and some other works not mentioned here we came to the conclusion that we can use existing approaches to program analysis but to also extend some of the previous works in this field. So, in our system in order to process source code and get features of it we can start with low-level instructions analysis since as we saw they can give the information about meaning of code and for the machine learning approach we might use recurrent neural networks given their powerfulness over other learning techniques like SVM.

3. APPROACH

3.1 Overview

We aim to generate some representations of comments or documentations for each code fragment. The representation should be related to the intension or the meaning of the accompanying code fragment, which refers to relatively self-contained piece of code, for example, a Python function or a Java method. We model this representation of comments as a distribution over a certain set of "topics". Each topic stands for a group of related natural language words or phrases. Thus, the problem is, given a code fragment, or an appropriate presentation of a code fragment, to predict its distribution over topics.

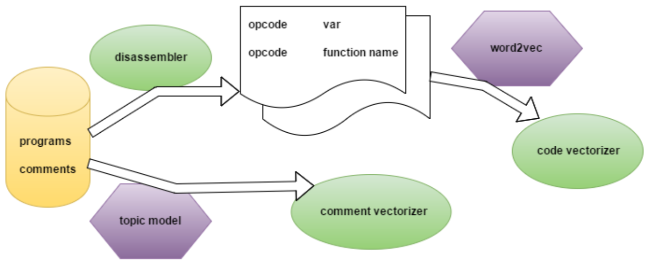
The core part of our approach is a neural network that receives the representation of a code fragment and predicts the representation of accompanying comments.

In the analysis phase, we first gather all program fragment and comment pairs from the training data. Then, we build a topic model over all comments and convert each piece of comments into a distribution over topics. At the same time, we build a feature extractor or vectorizer for all code fragments and convert each code fragment into a numeric representation.

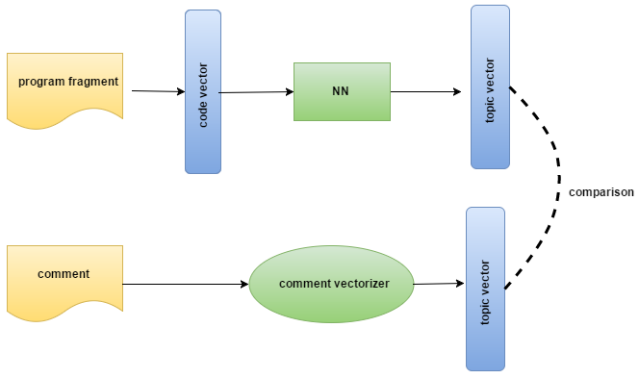
In the training phase, we train our neural networks with those numeric representations.

In the predicting phase, we use the code encoder and the neural networks learnt above, to first convert testing code into numeric representations and then feed into the neural networks in order to obtain the prediction of the topics for the code.

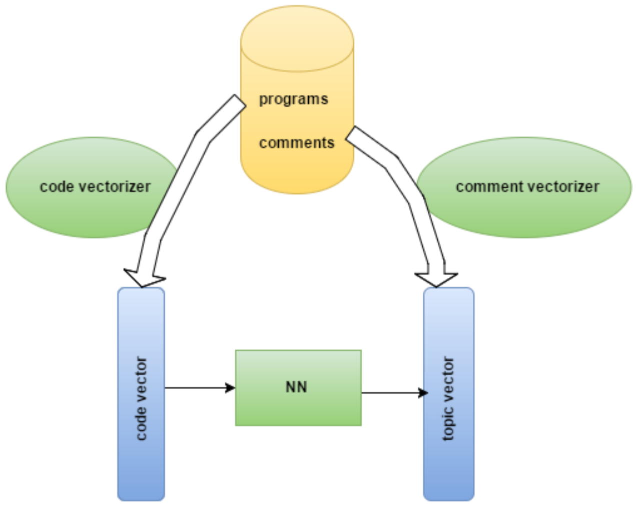
In the Figure 1, the flow graphs are shown for all analysis, training and predicting phases.



(a) Learning phase



(b) Predicting phase with evaluation process



(c) Analysis phase

Figure 1: Overview of our system in different phases.

3.2 Topic Modeling from Comments

Topic modeling algorithms are the algorithms that help people to navigate through a large database of documents, understand large data or search within it by defining underlying themes of each document [11]. Topic modeling showed its power and capability in its applications like searching engines or social networks where it helps to generate the most popular posts, most popular news or the recommendations list of potential user friends, etc. [12].

The intuition behind the idea of generating topics or keywords from the comments in our work is very similar to the idea of generating topics for the Twitter posts, since both of them share common characteristics. Some of those similar characteristics are that both are relatively short in length, they do not contain much of a context, even though some of the comments appear to be pretty long. In addition to that, both of them might contain grammatical errors, sometimes not direct sentence structures, misspellings, abbreviations and words that have different meaning in particular context, especially comments contain words that make sense only in the context of computer science field. Therefore, given this similarity we based our topic modeling on one of the existing methods used in topic modeling for Twitter posts.

This method is one of the basic topic modeling techniques called Latent Dirichlet Allocation (LDA) [13]. The core idea of LDA is that each document represents a distribution over fixed number of topics and each topic is a distribution over some number of also fixed words. All documents together make a collection. Thus, if there are N topics in the whole collection, only some part of it is exhibited in a particular document with different proportions for each topic.

Another algorithm that possibly can be used in the topic modeling is called tf-idf which stands for Term Frequency and Inverse Document Frequency. The purpose of tf-idf is to score the words in a text according to how important they are [14]. The idea behind this algorithm is to give higher score to the words that appear often in a document, but at the same time lower that score if the same word appears often in other documents too, because it means this word is not unique to the context of that particular document.

However, in our project we combine the use of LDA keyword/topic extraction and tf-idf frequency regularization. Using LDA we generate a bag of words for each particular comment block that describes a function. For the first set of experiments we decided to focus only on the functions and their comments, ignoring classes. The result of the topic generation process are a set of automatically generated topics and a distribution over the set of topics for each particular comment, representing how the comment is composed by those topics.

3.3 Program Representation

Program representation is itself a nontrivial task. Widely used representations are code sequence, abstract syntax tree, control flow graph, etc. Different representations affect what our system can learn from it.

3.3.1 Serialization of program

For the pre-processing part of the source code, we took inspiration from some of the previous works that as mentioned earlier use sequences of binary code to generate feature vectors such as [15] and through experiments proved that sequences of binary code contain information about the meaning of code. Working with the sequence of binary code is easier than working with the more complex structures like graphs, trees, etc. This idea is used in our project, so that instead of working with the high level source code we firstly translate source code into binary code and then work with the sequences of it.

Other higher level representations are probably capable of revealing more syntactic and semantic information of programs. Thus, using them can potentially help us improve the performance of our approach. However, we stay with the sequence representation approach to ease the complexity of our task and it gives us chance to try out our ideas sooner.

In our project, we will be focusing on Python programs and their complied intermediate bytecode sequence. We will be looking at the sequences of binary instructions and also the names of variables and functions associated with the low-level instructions, which contain some higher level information about the meaning of program.

3.3.2 Program embedding
The way we transform the sequence of binary code to the numerical value is by using word2vec [24]. The idea of word2vec is to learn with neural network an autoencoder for each distinctive word which can best predict its environment. The useful property of word2vec is that it clusters or groups words of similar meaning or that have some relation into a vector space. It has shown amazing results in quantizing semantic relations for nature language words.

3.4 Deep learner - Recurrent Neural Networks
For our task when choosing the machine learning technique, we decided to use Recurrent Neural Networks because of its idea of preserving context of the data. As we mentioned earlier there are works that apply NLP techniques to do code analysis and which consider the program as a natural language.

Then, after generating those vectors we can use them easily to feed into the recurrent neural network, where the same unit of neural network recur again and again for all elements in a sequence while keeping track of "contextual" information. There is a special type of RNN called Long Short Term Memory networks (LSTMs) [17]. The largest advantage of this RNN is that it is able to learn long term dependencies. On a figure 2 there is a structure of the simple LSTM. The difference between LSTM and usual RNN is that in LSTM the repeating module has four neural network layers each of which decides different parameters, like which information should be thrown away, or which information should be passed further, or decide what to output [17].

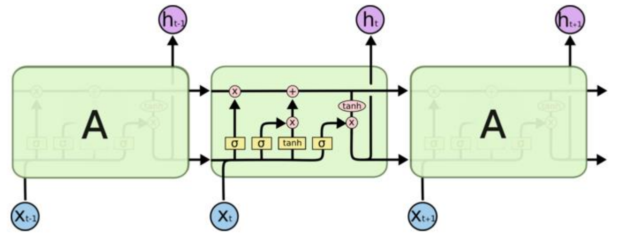


Figure 2: The structure of the repeating module in LSTM.

In our project we use slightly different version of LSTM called Gated Recurrent Unit (GRU) because it is faster and has simpler structure. The difference is that it merges "forget" and "input" gates and makes some other changes [16].

4. EXPERIMENTS
4.1 Data
In our project, we focus on Python programs. Python is now a popular language in virtually all fields of programming. It would be beneficial to a large range of people if we can help gain insight of Python code. Also, Python itself tries to have you put descriptive comments at the first line of a function, class or even module itself, which we call "documentation string". We use this string as the most meaningful and then only comments to help us identify the intention of the Python entity.

We gathered our code data from the source code of major Python libraries: numpy 1.11.0, scipy 0.17.0 and scikit-learn 0.17.1. We choose them because

1. They are all about scientific computing. This can help us narrow down the field to get more specific topics.
2. They have good documentation string for each function, class and submodules. This essentially allows us to extract useful information from them.
3. They are popular among various users. It would be effective if we can help understand programs related to the libraries.

We use Python standard library to parse through libraries to collect all pairs for functions and their documentation strings, filter out functions without documentation string and get 3979 valid pairs for our dataset. We split them randomly into training and testing dataset, where 80% (3183 pairs) goes to training and 20% (796 pairs) goes to testing.

4.2 Implementation
Major tools and libraries we used are listed below.

- Python 3.4.4 with standard library [19]
  - *dis*: disassembler
  - *inspect*: inspect live object
- *gensim* [20]: Python library for topic modeling. Supports word2vec, LDA and tf-idf.
- *Keras* [21]: Python library providing API for neural networks.
- *Theano* [22]: Python library support fast tensor operation. Used as backend of Keras.

4.2.1 Comment processing
For the comment part, or namely documentation strings, we extract only the first paragraph, tokenize and stem the words. We then ignore words that appear only once and convert the passage into bag of words. Next we apply tf-idf to regularize the word frequency and apply LDA using *gensim* in order to obtain a set of topics and a model that can generate distribution over topics for a bag of word tokens.

4.2.2 Code processing
We use Python standard library dis to disassemble functions into the sequence of instructions along with the names of referred variables and functions. Due to the lack of time, we only generate numeric values from sequences of binary instructions using word2vec with the help of *gensim*. Each instruction is learned to be converted into an 8-dimensional vector $v_{instr} \in [0,1]^8$ in order to predict in a best way the surrounding window of 10 successive instruction sequences.

4.2.3 Neural networks
For the learning part in order to use recurrent neural networks we used the Python library Keras which provides handy APIs to help build various neural network structures.

The neural network structure we use is shown in the table 1. We have first a GRU layer with 64 output neurons, then a fully connected hidden layer with 128 neurons and *tanh* activation, finally a full connected output layer with 20 neurons corresponding to each topic and *softmax* activation to help normalize the output into a distribution. The notion of activations we used follow the standard. Namely,

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

layer0	layer1	layer2	layer3
I(200)	GRU(64, W=0.25, u=0.25)	F(128, tanh, d=0.5)	F(20, softmax)

Table 2: The detailed structure of our neural network. I(n) stands for an input layer with input length n. GRU(n, W, u) stands for a GRU layer with output dimension n, drop rate W for input gates and drop rate u for recurrent connections. F(n, f, d) stands for fully connected layer with n neurons and with output activation function f and drop rate d. Note dropping is only effective in training phase and is transparent during prediction.

Topic ID	10 top words of the topics and their weights									
1	0.088* seri	0.047* legendr	0.045* hermit	0.029* histogram	0.024* evalu	0.022* frequenc	0.021* polynomi	0.020* divid	0.019* point	0.018* hermite
2	0.040* degre	0.040* vandermond	0.039* matrix	0.034* pseudo	0.032* sequenc	0.032* fill	0.025* discret	0.021* transform	0.021* document	0.017* eigenvalu
3	0.045* space	0.035* charact	0.024* divid	0.022* append	0.022* true	0.020* integr	0.018* fals	0.018* error	0.016* pad_amt	0.015* string
4	0.048* represent	0.046* string	0.043* helper	0.042* varianc	0.039* wishart	0.035* distribut	0.030* case	0.027* spline	0.020* function	0.020* comput
5	0.082* predict	0.071* class	0.059* probabl	0.040* target	0.029* label	0.021* dictionari	0.021* companion	0.019* regress	0.019* data	0.017* posterior
6	0.047* evalu	0.031* repeat	0.029* execut	0.027* behavior	0.024* call	0.021* multivari	0.019* normal	0.014* result	0.012* equival	0.012* axi
7	0.036* chebyshev	0.028* decis	0.026* final	0.025* implement	0.025* estim	0.023* integr	0.023* method	0.022* singl	0.022* seri	0.021* transform
8	0.049* test	0.035* normal	0.027* perform	0.024* system	0.023* vector	0.023* classif	0.020* represent	0.019* function	0.018* probabl	0.018* transfer
9	0.029* map	0.028* appli	0.025* model	0.025* reduct	0.022* project	0.022* data	0.021* gener	0.021* learn	0.020* random	0.019* fit
10	0.034* standard	0.034* dimens	0.027* input	0.026* maximum	0.026* arrai	0.026* axi	0.025* statist	0.021* updat	0.020* comput	0.020* deviat

Table 1: 10 topics out of 40 generated from training data.

and

$$softmax(\vec{x})_i = \frac{x_i}{\sum_{x_i \in \vec{x}} e^{x_i}}$$

We train the neural network using categorical cross entropy [18] as objective loss function and Adam [23] as optimizer. In addition, we also utilized drop-out to help prevent overfitting and train more robust weights in the GRU and the penultimate fully-connected layer.

4.3 Evaluation method

We will use cross entropy, which is a popular method for measuring the difference between distributions, to compare the predicted topic distribution and the truth distribution generated from comment in the test dataset. Note this is also used as the objective of our NN.

Another possible idea of evaluating results is to use comparison of top 3 topics generated for the given source code with top 3 topics from the comments of test set. This gives another sense of the accuracy of our predictions.

One of the challenges we are facing right now is coming up with the evaluation procedure for our topic generator: answering a question how reasonable topics are generated from given comments. Currently, it seems like it might be evaluated only by human. e.g. see if each or most of the topics actually appear to capture some of closely related group of keywords, and if two pieces of comments which are similar in meanings are actually assigned with similar distribution over topics.

Also, it is necessary to evaluate our code vectorizer: answering a question of how reasonable and meaningful is a vector representation for the given piece of code. We probably might look closer into how word2vec tool evaluates its results.

4.4 Results

Due to the lack of time, unfortunately, we did not fulfill all the experiments that we planned to conduct. In this part we would like to present some of our results for code and comment processing.

4.4.1 Topic model for comments

We generate 40 topics for all the comments in our dataset. In Table 2 you can see the first 10 out of 40 topics generated by topic model.

In Table 3 you can see the similarity score between the first function with id 3301 and other top similar functions according to their comments.

4.4.2 Vectorization for bytecode instructions

There are 84 unique Python bytecode instructions appeared in a total number of ~400k instructions we get from all functions in our dataset. The mean length of bytecode instruction sequence is 83.1 for our whole dataset, with a minimum 2 and a maximum 1761. Table 4 gives information about some of the binary instructions and their vector representation as well as similarity measures for the instruction BINARY_ADD and other top similar in the meaning instructions.

4.4.3 Neural network

We trained our NN on training dataset for 200 epochs. The training and testing loss curves over the number of epochs are shown in Figure 3. The training time is approximately 140 seconds per epoch.

3301	scipy.cluster.hierarchy	linkage	1	Performs hierarchical/agglomerative clustering on the condensed distance matrix y.
2618	sklearn.decomposition.online_lda	_update_doc_distribution	1	E-step: update document-topic distribution.
1359	sklearn.cluster.k_means_	_labels_inertia_minibatch	0.999	Compute labels and inertia using mini batches.
2677	sklearn.neural_network.rbm	gibbs	0.983	Perform one Gibbs sampling step.
3	scipy.cluster.hierarchy	cophenet	0.982	Calculates the cophenetic distances between each observation in the hierarchical clustering defined by the linkage ``Z``.
3030	scipy.io.matlab.mio5_params	_convert_codecs	0.907	Convert codec template mapping to byte order
2822	numpy._import_tools	get_pkgdocs	0.723	Return documentation summary of subpackages.
1300	sklearn.cluster.k_means_	fit	0.722	Compute k-means clustering.
445	scipy.io.netcdf	itemsize	0.722	Return the itemsize of the variable.
393	scipy.stats.stats	f_oneway	0.722	Performs a 1-way ANOVA.

Table 4: The similarity score between first function and the rest according to comments.

Instruction	Similarity	Vector
BINARY_ADD		(-0.58, 1.44, 2.59, 1.45, -3.26, 3.50, -1.04, 6.47)
BINARY_SUBTRACT	0.9796327353	(0.23, 2.31, 3.36, 1.12, -2.90, 2.85, -0.58, 7.54)
BINARY_MULTIPLY	0.9578515887	(0.44, 2.09, 2.36, 0.39, -2.34, 2.97, -1.14, 8.94)
BINARY_POWER	0.9076402187	(-1.16, 2.90, 2.60, -0.90, -1.08, 3.64, -2.92, 8.05)
BINARY_TRUE_DIVIDE	0.9009826183	(0.05, 3.70, 2.34, -0.68, -1.19, 2.61, -1.53, 7.86)
UNARY_NEGATIVE	0.835010767	(1.45, -0.95, 0.55, 0.21, -1.79, 1.49, 0.25, 6.41)
LOAD_FAST	0.7789117694	(0.93, 0.20, 0.50, 0.52, -1.17, 0.69, 1.01, 1.89)
STORE_FAST	0.685264051	(2.01, 0.48, -0.46, -0.20, -2.15, 1.86, -1.01, 1.92)
INPLACE_MULTIPLY	0.6432930231	(1.52, 0.85, 4.73, 0.14, -0.04, -1.18, 1.14, 5.49)
LOAD_CONST	0.6002776027	(-0.29, -0.20, -1.39, 2.25, -1.15, 1.36, -0.47, 1.63)
INPLACE_TRUE_DIVIDE	0.5996887088	(0.12, 1.09, 4.18, 1.13, 1.23, -1.69, 0.56, 5.49)

Table 3: Vector representation of some of the binary instructions and similarity scores between first instruction and the rest.

4.5 Discussion

For the learned topic model, we can see some of the topics clearly try to capture discriminative signatures in distribution over all possible words. One example is Topic 2, where the presence of “vandermond” and “eigenvalue” indicates that the topic is probably related to matrix operations over polynomial equation systems. Although it may still remain a tricky task to come up with a proper short name or description for each of the topics it generated.

For the vector representation of bytecode instructions, we can see our learned representation is rather reasonable in the sense that it assigned similar vectors to instructions with similar functionality. It can be seen that the instruction BINARY_ADD is projected closely to other binary arithmetic operations, among which BINARY_SUBSTRACT is identified as the most similar one.

Our NN seems to converge with training loss above 2.3 and testing loss above 2.5 according to the loss curve in Figure 3. We regard this not a very good result. The reason could simply be that we lose too much useful information from variables and the function/method called from the object, as well as the structure and control flow of the object. Besides we also cut off instruction sequences at certain length and ignore the rest due to the limitation in the implementation of our RNN. Thus, a good-enough prediction may not be probable based on insufficient inputs. Another possible reason could be that the true output of comment topic distributions are themselves inaccurate or erroneous. If our learner can get bad learning example it could then learn bad.

5. FUTURE WORK

There is definitely a lot of experiments that can still be done related to the idea of our project. Here we list some of the promising ideas that unfortunately were not explored by us more due to lack of time. But we believe they have high potential to generate even more meaningful results than we currently obtained.

5.1 Data

We should say that our data is not sufficient (around 4000 python functions). Major methods we use during this project, including topic modeling, word2vec and deep neural networks, fulfill their powerfulness only upon abundance of accessible data. We can see our bytecode instruction vectorizer generates

5.2 Comment processing

We should be more careful on data cleaning, which is critical to text learning task.

5.3 Code processing

We didn't implement our idea of taking variable and function names alone with the instruction sequence. This undoubtedly can harm the final performance of our system.

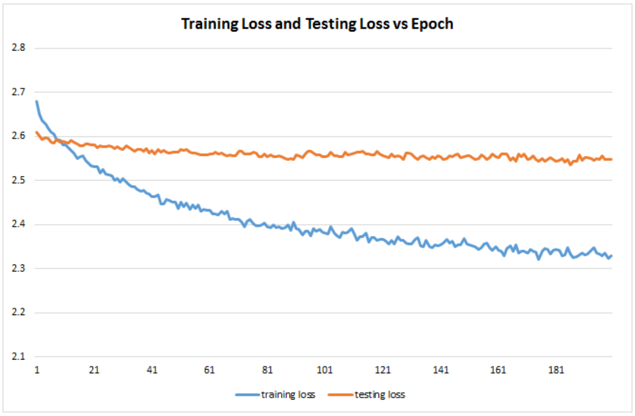


Figure 3: The change in training loss and testing loss over 200 epochs.

5.4 Neural network structures

We can try various combinations of structures and parameters, and experiment how they affect the behavior of the system in different aspects such as prediction accuracy, training and predicting time.

One thing worth trying is to add an additional convolutional layer, or layers, upon the input sequence before it gets fed into the recurrent layer. The rationale is that the verbose length of the input low-level instruction sequence weakens the contextual connection between codes, also harming the training efficiency of the recurrent neural net. Convolutional neural networks are shown to be capable of summarizing over low-level representations (such as pixel images) and can generally be trained faster than recurrent neural nets. With the help of convolutional layer(s), our structure could be better at capturing critical effects of successive instructions and how they are meaningfully connected to the whole program fragment, as well as reducing the training effort by feeding less lengthy sequences to recurrent layer.

5.5 Recursive analysis

Recursive nature of program structures. Programs are in natural recursive structures. Python as an example, package contains modules, module contains definition of classes and functions, function contains definition of other functions or, more commonly, calls to other functions. We can lose valuable information if we ignore this fact and just flatten them into series.

Two kinds of very helpful information underlying the structure could be:

1. **Context of Higher-level entity.** Higher-level entities are entities in the upper level of the static syntax tree, such as a class to its member functions or a module to the classes and functions reside in it. They are sometimes critical in understanding the different meanings of the lower entities which could otherwise look very similar or even identical. One example could be that one function which computes the Euclidean distance of two input arguments could mean an equation for a sphere in a class dealing with graphics, or otherwise mean the mean least square error in a machining learning module.
2. **Meaning of lower-level entity.** In almost all the cases, we need to know the meaning of subsequent entities (E.g. function calls) to help know better about the entity itself we are analyzing. However, knowing the meaning of a lower-level entity can involve understanding lower-level entities to it recursively. Note that in this scenario, we are to deal with dynamic behavior of the program, namely function calls, which is almost always trickier than the static property. Difficulties arise with the introduction of polymorphisms, overloading and etc. In future, we could start with static analysis of function call hierarchies to see to what extent this idea can help with the task.

Recursive neural networks (RvNN). Make use of recursive neural networks which fits perfectly for this propose. RvNN has shown its proficiency in learning and predicting for recursive structures, for example parsing natural language sentence and understanding natural scene images [25].

Different from recurrent neural networks (RNN) where the same NN unit recurs again and again along the coming sequence, RvNNs use the same unit again and again at all nodes in an input that is inherently tree structure. This nature makes it able to be easier adapted and faster trained for recursive structures like trees.

In future, we could analyze the code statically and build recursive tree structure capturing encapsulation and calling hierarchies for given libraries. Then we could use refined ways to vectorize each entity in the tree structure and feed it to a RvNN system. The system will first propagate from bottom to top over the whole tree in order to capture context of higher-level entities. Then our system could apply a same RNN structure from bottom to top again to predict for each entity with the information of its both higher-level entity and all lower-level entity. The system can be trained using similar scheme for other RvNNs and RNNs.

6. CONCLUSION

In this project, we try to shed some light on the interesting and intricate task of automatic comment generation by starting with predicting topics that are related to the meaning or intension for a code fragment. We build topic models for Python function documentation strings, the place to put principle comments encouraged by the language, and treat the topic composition of the string as the indicator of the function's intension. We then make use of only instruction names in the function's compiled bytecode sequence and vectorize instructions in a way to best reveal their surroundings. Finally, we build a RNN with additional fully connected layers and trained it against our code dataset collected from major Python library. The result is moderate but we believe it can be greatly improved with more data, more careful text processing of comments, consideration of lexical and syntactic information of the code and more informative recursive model combined with recursive neural networks.

7. ACKNOWLEDGMENTS

We thank Prof. Aws Albarghouthi for his instruction and suggestions during this course project. We thank David Brown for the inspiring discussion on program classification and Sidharth Mudgal for his discussion and suggestion on technologies and implementations of neural networks.

8. REFERENCES

[1] Raskin J. Comments are more important than code. ACM Queue. 2005 Mar 18;3(2).

[2] Linux Foundation. http://www.linuxfoundation.org/news-media/infographics

[3] Rosenblum N, Zhu X, Miller BP. Who wrote this code? identifying the authors of program binaries. InComputer Security–

ESORICS 2011 2011 Sep 12 (pp. 172-189). Springer Berlin Heidelberg.

[4] Khoo WM, Mycroft A, Anderson R. Rendezvous: a search engine for binary code. InProceedings of the 10th Working Conference on Mining Software Repositories 2013 May 18 (pp. 329-338). IEEE Press.

[5] Raychev V, Vechev M, Yahav E. Code completion with statistical language models. InACM SIGPLAN Notices 2014 Jun 9 (Vol. 49, No. 6, pp. 419-428). ACM.

[6] Sinai MB, Yahav E. Code similarity via natural language descriptions. POPL Off the Beaten Track, OBT. 2014;15.

[7] Tan L, Yuan D, Krishna G, Zhou Y. /* iComment: Bugs or bad comments?*/. InACM SIGOPS Operating Systems Review 2007 Oct 14 (Vol. 41, No. 6, pp. 145-158). ACM.

[8] Karaivanov S, Raychev V, Vechev M. Phrase-based statistical translation of programming languages. InProceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software 2014 Oct 20 (pp. 173-184). ACM.

[9] Oda Y, Fudaba H, Neubig G, Hata H, Sakti S, Toda T, Nakamura S. Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation (T). InAutomated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on 2015 Nov 9 (pp. 574-584). IEEE.

[10] Allamanis M, Peng H, Sutton C. A Convolutional Attention Network for Extreme Summarization of Source Code. arXiv preprint arXiv:1602.03001. 2016 Feb 9.

[11] Blei DM. Probabilistic topic models. Communications of the ACM. 2012 Apr 1;55(4):77-84.

[12] Hong L, Davison BD. Empirical study of topic modeling in twitter. InProceedings of the first workshop on social media analytics 2010 Jul 25 (pp. 80-88). ACM.

[13] Blei DM, Lafferty JD. Topic models. Text mining: classification, clustering, and applications. 2009 Jun 15;10(71):34.

[14] Lott B. Survey of Keyword Extraction Techniques. UNM Education. 2012 Dec 4.

[15] Morris, A., and D. B. Brown. Identifying Program Subject from Binary Code. University of Wisconsin, Madison. DOI= http://pages.cs.wisc.edu/~jerryzhu/hack/761/CS761_Alexander_Morris.pdf

[16] Jozefowicz R, Zaremba W, Sutskever I. An empirical exploration of recurrent network architectures. InProceedings of the 32nd International Conference on Machine Learning (ICML-15) 2015 (pp. 2342-2350).

[17] Hochreiter S, Schmidhuber J. Long short-term memory. Neural computation. 1997 Nov 15;9(8):1735-80.

[18] Cross entropy https://en.wikipedia.org/wiki/Cross_entropy

[19] Python standard library. https://docs.python.org/3/library/

[20] *Gensim* - topic modeling for humans. https://radimrehurek.com/*gensim*/

[21] Keras - deep learning library. http://keras.io/

[22] Theano - python library to optimize mathematical expressions. http://deeplearning.net/software/theano/

[23] Kingma D, Ba J. Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. 2014 Dec 22.

[24] Mikolov T, Chen K, Corrado G, Dean J. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781. 2013 Jan 16.

[25] Socher R, Lin CC, Manning C, Ng AY. Parsing natural scenes and natural language with recursive neural networks. InProceedings of the 28th international conference on machine learning (ICML-11) 2011 (pp. 129-136).