Identifier Inference through Neural Networks

Zeyuan Hu
Computer Science Department
University of Texas at Austin
Austin, Texas
`iamzeyuanhu@utexas.edu`

Abstract

Source code can be treated similar as corpus constructed by natural language (Hindle et al., 2012). In this paper, we use the neural network model to study *identifer naming convention* problem. We find that neural network model can predict 16.5% identifiers correctly on a randomly-selected source file by training on the unrelated projects. In addition, we compare the performance of model on character level and word level and explore the impact of different input sentences construction methods on the model performance.

1 Introduction

The idea of applying natural language processing (NLP) towards code corpus is first proposed by Hindle et al. (2012). The authors think software code is similar to the natural language in the sense that they are both "*natural product of human effort*". Like word "bank" usually follows "Federal Reserve" in natural language, software engineers often know what is coming up next after seeing the code fragment `for(i=0;i<10`. Thus, we can use NLP techniques to perform infernece on the code in a similar way that we do with the natural language. The inference task on the code includes identifier naming convention, code formatting preference, design patterns, which are referred as *coding convention inference problem* in general(Allamanis et al., 2014). Formally speaking, *coding convention inference problem* is defined as "the problem of automatically learning the coding conventions consistently used in a body of source code" (Allamanis et al., 2014).

In this paper, we explore the *identifier naming convention* subproblem, which is to predict the identiifer given the context of the source code.

```
identifier ::=
              (letter|"_") (letter | digit | "_")*

letter ::=
              lowercase | uppercase

lowercase ::=
              "a"..."z"

uppercase ::=
              "A"..."Z"

digit ::=
              "0"..."9"
```

Figure 1: Python identifiers lexical definition

Specifically, we carry out our study on Python[1]. Identifiers (also referred to as *names*) in Python are defined in Figure 1 (Rossum, 1995). They can include class names, function names, argument names, and variable names. As a demonstration of the *identifier naming convention* prolem, suppose we have seen `if len(sys.___) < 2:` in the code, we may want to fill in the blank with `argv` instead of other identifiers.

We use the long short-term memory (LSTM) model to approach this problem. Previous work has shown that LSTM model is an effective framework on language modeling task (Sundermeyer et al., 2015). We find that by learning from training corpus, we can predict 16.5% identifiers correctly on a randomly-selected test source file.

2 Model and Evaluation Metric

We use a two layer architecture shown in Figure 2, which is similar to the neural network LM architecture proposed by Sundermeyer et al. (2015). The first layer is an embedding layer that maps each identifier represented by the index appeared in the vocabulary from input sentences into dense vectors of fixed size. Then we feed the output of the embedding layer into the LSTM layer and

[1]`https://www.python.org/`

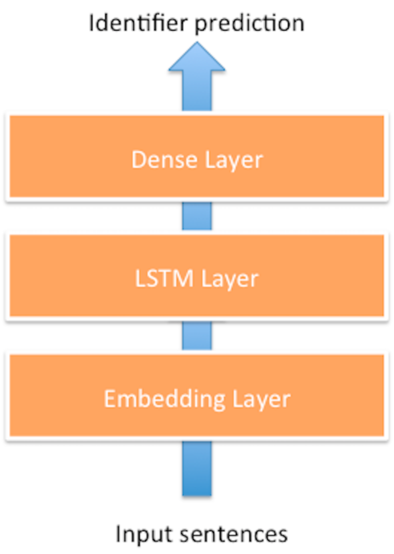

Figure 2: Neural network architecture

then we apply a softmax activation function at the dense layer to produce normalized probability values for each word in the vocabulary. As suggested by Sundermeyer et al. (2015), we use the cross entropy error as our loss function, which is the same as maximum likelihood. The input sentences are constructed by the identifers, which will be discussed in details in the "Experimental Results" section.

Perplexity is often used to quantify the performance of language models. However, as suggested by Jurafsky and H.Martin (2017), an (intinsic) improvement in perplexity does not necessarily indicate a (extrinsic) performance increase in NLP tasks. Thus, in addition to perplexity, we use the *Single Point Suggestion* (SPS) evaluation metric proposed by Allamanis et al. (2014) as a way to perform "an end-to-end evaluation" (Jurafsky and H.Martin, 2017). The steps for calculating the SPS score is shown in Algorithm 1. The SPS score calculates the fraction of identifiers that can be predicted correctly given all the identifiers appeared in the test files. Ideally, the SPS score should be 100%. However, there is a real world implication for a less-than-perfect SPS score. Suppose software engineers use the same variable name for the places that have the same semantic meaning (i.e., think about using `idx` for loop index in the source file). However, he does not follow his naming conventions consistently throughout the whole file (i.e., he use `i` for some loop indices) and model may suggest the most appeared variable names (i.e., `idx` rather than `i`), which lead to the less-than-perfect SPS score.

Algorithm 1 Single Point Suggestion

```
for each test file do
    Total count = number of unique identifiers
    Counter = 0
    for each unique identifier do
        Collect all of the locations where the identifier occurs and names the same entity
        Ask model to suggest a new name and rename all occurrences at once
        if new name = original name then
            Counter += 1
        end if
    end for
    Accuracy = Counter / Total count
end for
```

3 The GitHub Python Corpus

Construction of the corpus is inspired by Allamanis and Sutton (2013). In their work, they collect 14,807 Java projects from GitHub[2] based on the number of forks. They use the number of forks as a way to measure the quality of projects. In this paper, we present a different corpus that is based on Python and uses GitHub's star system to measure the quality of the projects.

In a report published by GitHub (2017), Python is ranked the second most popular langauges on the platform with about 1,000,000 opened pull requests[3], which surpasses the third most popular language Java's opened pull requests by around 14,000. We directly query GitHub through its REST API v3[4] and filter individual projects that have at least one star [5] and the programming language is Python. We use GitHub's star system as a way to create a quality corpus. Study has shown that there is a positive correlation between the number of contributors and the number of stars (Jarczyk et al., 2014). Since on GitHub, developers can only make contributions to the projects by forking first. However, not all forks will be turned into pull requests with code changes that contribute back to the original projects. Thus, we

[2]`https://github.com/`

[3]In GitHub's terminology developers need to make pull request in order to have their code change merged back to the original codebase

[4]`https://developer.github.com/v3/`

[5]the number of stars of a project indicates how many times the project has been endorsed by the developers on GitHub

think the number of contributors can be interpreted as the effective forks. Given the prior success of the GitHub Java corpus (Allamanis and Sutton, 2013) and the positive correlation between the number of stars and the number of effective forks, we think the star system can be a good measure of both the quality and the popularity of the projects.

We then download (`Clone` in Git's terms) the top 1,000 most popular python projects across a wide variety of domains amounting to 25,088,709 lines of code in 135,209 files. Tables 1 and 2 present some projects in our corpus. Only files with the `.py` extension were considered. We split the repository 80%, 10%, 10% based on the lines of code assigning projects into the training, dev set, and test set respectively.

Table 1: Top Projects by Number of Stars in Corpus

Name	# stars	Description
awesome-python	40,963	A curated list of Python resources
httpie	32,534	Modern command line HTTP client
thefuck	32,011	App that corrects previous cmd
flask	30,931	A web application framework
youtube-dl	30,911	console app to download videos

Table 2: Top Projects by LOC in Corpus

Name	kLOC	Description
spaCy	3059	NLP with Python and Cython
hue	880	Data analytics workbench
appscale	796	Implementation of Google App Engine
main	795	Python implementation for .NET framework
kbengine	502	A MMOG engine of server

A potential limitation of the resulting corpus is that we exclude some variations of the python files (i.e., `.pyx` and `.pxd` for Cython). Due to the limit of computing resources, we use 670,536 LOC for training, 302,434 LOC as a dev set, and a randomly-chosen source file from test set to report model performance.

4 Experimental Results

We explore the impact of the charcacter level vs. word level generation, input sequence handling, and we compare the performance between language model and neural network model. Without special note, the experiments are based on the hyperparameter setting in Table 3.

Table 3: Hyperparameter settings

Description	Values
vocabulary size	15000
embedding output dimension	64
dropout rate	0.2
LSTM memory units	64
time steps (word, context)	5
time steps (word, loc)	121
number of epochs	10
batch size	64
learning rate	0.01
optimizer	Adam

4.0.1 Character Level vs. Word Level

Identifier naming convention problem can be treated as a text generation problem. Essentially, we try to generate (i.e., predict) given the context of the surrounding identifers. Thus, we can generate the identifiers character by character or word by word. The advantages of generating identifers on character level are:

- A much smaller vocabulary size. The vocabulary size on character level is 37, which is much smaller than 15,000. Small vocabulary size usually indicates low computation resource is needed. Neural network model is especially computationally expensive when we compute softmax in the output layer (Jozefowicz et al., 2016).

- More diverse representation of identifiers. Word level generation is limited to the identifiers appeared inside the vocabulary. However, character level generation can potentially generate the identifiers that are unseen in the vocabulary.

We run the model on the character level and part of the output is shown in Table 4. As one can see, it is really hard to generate a meaningful identifier on character level given the current architecture of the model. The result is consistent with the previous work on the application of character level neural language model on natural text in the sense that the architecture of the character level model is usually much more complex than the one on word level given the same level of performance (Kim et al., 2016).

We also run the same model on the word level and the result on the identifier prediction is shown in Tabel 5. We can see that predicted identifiers are all meaningful words and we can get

Table 4: Part of the identifer prediction output using model on character level

Identifier	Top four predictions
absolute_import,0	”, ’fer’, ’cet’, ’clannetteeteet’
path	‘alr’, ‘teoetet’, ‘neteeeete’, ‘reee’
version_info	‘ient’, ‘’, ‘seleehere_eekertert_’, ‘feenert__’

Table 5: Part of the identifer prediction output using model on word level

Identifier	Top four predictions
absolute_import	’UNK’, ’self’, ’VGroup’, ’None’
path	’path’, ’listdir’, ’UNK’, ’symlink’
version_info	’version_info’, ’socket’, ’UNK’, ’os’

some identifiers predicition correct (i.e., `path` and `version_info`). Since the model is allowed to predict `UNK`, we are indicating that the rare identifiers can appear in the unusal context.

4.0.2 N-gram vs. Neural Network

Allamanis et al. (2014) and Hindle et al. (2012) use the N-gram as the language model to study the source code. Specifically, Allamanis et al. (2014) implement a cache language model (i.e., “cross-project language models” in the paper) and claim to achieve 94% SPS score on the GitHub Java Corpus. We fail to reproduce their result. Thus, we implement a 5-gram language model with Kneser-Ney smoothing based on KenLM framework (Heafield, 2011) as our own baseline instead. The performance comparison between the 5-gram language model and our neural network model is shown in Table 6.

Both language model and Neural network model have similar performance in terms of perplexity. But it is really surprising to see that 5-gram language model can hardly make any prediction correct. However, our neural network model can at least make 16% of identifier prediction correct. One reason for the low SSP score of 5-gram language model is that there are large gaps among the identifiers in terms of the number of appearances in the training corpus. We sort the identifiers in the vocabulary by their word counts in the training corpus and find that the most frequent identifier `self` appears in the training corpus 2,889,429 times while the 15,000th identifier `literal_block` only appears 96 times. During the prediction, the identifiers with larger word counts may be much more favorable than the identifiers with much smaller word counts.

Table 6: Model performance comparison between 5-gram and neural network model

Models	Perplexity	SSP Score
5-gram with Kneser-Ney smoothing	9.906	9.5%
Neural network (word, context)	**8.729**	**16.5%**
Neural Network (word, loc)	11.38	11.3%

For our neural network model, the model seems to be real good at capturing certain identifier combination with low varition of alternative writing styles. For example, the model makes prediction `argv` correctly for `len(sys.__) < 2` across different test files. The semantics for this code fragment is to test the length of input arguments for the program. Since software engineers always use `len(sys.argv)` to test the length of input arguments, there is not much variation compared with `i`, `idx` in the loop index. Thus, our neural network model can well capture this phenomenon.

For perplexity, the numbers are calculated based on the first 500 identifiers of the test file. If we include all the identifiers of the test file, the perplexity for our neural network models are infinities, which are due to several extremely small probabilities of pattern occurences.

4.0.3 Difference in Input Sentences

One thing to note is how we construct our input sentences. We try two ways: one is to construct the input sentence based on lines of code. In other words, each line is considered as a single sentence (i.e., “Neural Network (word, loc))” in Table 6). The other way is to treat a whole source file as one passage and use a context window to split the passage into sentences (i.e., “Neural network (word, context)” in Table 6). We find that the latter one gives better performance in neural network model. One possible reason behind the performance difference is in the padding of input sentneces. We examine the input sentences based on the first method and find out that the maximum length of a line of code can be 121 identifiers, which is a `__init__` method that contains over 14 arguments with several arguments have complex data structure default input (i.e., `kwds = dict(axis=axis, ...)`. However, there also exists lines of code with simply two or three identifiers (i.e., `import os`). If we pad the input sequence based on the maximum length of line of code, then lines with two or three identifiers will be padded with many (i.e., 119) placeholder numbers. Then, the model will put more focus on learning the pattern of placeholder numbers than the actual ones. However, if we set a cutoff and truncate the lines of code, then certain pattern will be lost. For example, given a line of code `mark = js.JSONInstance`, the identifiers are `mark, js, JSONInstance`. If we set the cutoff to 2, then `JSONInstance` will be discarded and have a chance of not to be learned.

Lastly, the running time are different between two models. Context window construction method has edge over the lines of code construction method. This is because the number of words included in the context window is much smaller than 121 and we do not need to padd every input sentence to the length of 121.

5 Conclusion and Future Work

In this project, we explore the application of neural network in solving the *identifer naming convention* problem. We experiment with both N-gram language model and word-level neural networks. We find out that neural network has better performance both in terms of perplexity and the *Single Point Suggestion* evaluation metric. However, there are still several points waiting for the further exploration:

- **Explore cache language model** (Allamanis et al., 2014) use cache language model and achieve astounding 94% SSP score. As one can see, simple N-gram model may perform poorly because we may implictly assume that the word count distribution for a local project that the test file belongs is the same as the set of projects in the training set. Thus, we may want to build a local language model for the current project. This idea leads us to think whether we can achieve performance gain by building two neural network models in a similar way.

- **Neural network architecture exploration** We explore model both on character level and word level. With all the advantages offered by the character level model, we want to see if given a more sophisticated architecture, we can have better predicting power using character level model.

6 Acknowledgements

The author greatly appreciate Junyi Jessy Li, Milos Gligoric, and Greg Durrett for their invaluable feedback, precious time, and insightful comments.

References

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, FSE 2014, pages 281–293. https://doi.org/10.1145/2635868.2635883.

Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale using Language Modeling. In *The 10th Working Conference on Mining Software Repositories*. IEEE, pages 207–216.

GitHub. 2017. The state of the octoverse 2017. `https://octoverse.github.com/`.

Kenneth Heafield. 2011. KenLM: faster and smaller language model queries. In *Proceedings of the EMNLP 2011 Sixth Workshop on Statistical Machine Translation*. Edinburgh, Scotland, United Kingdom, pages 187–197. https://kheafield.com/papers/avenue/kenlm.pdf.

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, ICSE '12, pages 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322.

Oskar Jarczyk, Błażej Gruszka, Szymon Jaroszewicz, Leszek Bukowski, and Adam Wierzbicki. 2014. *GitHub Projects. Quality Analysis of Open-Source Software*, Springer International Publishing, Cham, pages 80–94. https://doi.org/10.1007/978-3-319-13734-6_6.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. 2016. Exploring the limits of language modeling. https://arxiv.org/pdf/1602.02410.pdf.

Dan Jurafsky and James H.Martin. 2017. Speech and language processing. Preprint on webpage at `https://web.stanford.edu/~jurafsky/slp3/`.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. 2016. Character-aware neural language models. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. AAAI Press, AAAI'16, pages 2741–2749. http://dl.acm.org/citation.cfm?id=3016100.3016285.

Guido Rossum. 1995. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands.

Martin Sundermeyer, Hermann Ney, and Ralf Schlüter. 2015. From feedforward to recurrent lstm neural networks for language modeling. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.* 23(3):517–529. https://doi.org/10.1109/TASLP.2015.2400218.