A Novel Self-Attention Based Automatic Code Completion Neural Network

Bohao Wang, Wanyou Lv, Jianqi Shi, and Yanhong Huang*

National Trusted Embedded Software Engineering Technology Research Center,
East China Normal University
{bohao.wang, wanyou.lv}@ntesec.ecnu.edu.cn
{jqshi, yhhuang}@sei.ecnu.edu.cn

***Abstract*—Code completion is one branch of source code modeling tasks. Using a deep learning method to implement it has explored the possibilities of modeling source code with a statistic language model. Recurrent Neural Network (RNN) is a universal feature extractor of Natural Language Processing (NLP), which is used in the code completion field commonly. However, RNN based models are lack of long-range context dependency and have a poor performance in training speed. Besides, some previous models have not handled the issue of out of vocabulary (OOV) well, which hinders further improvements in prediction accuracy. This paper presents a novel automatic code completion neural network, which is based on a self-attention mechanism with open vocabulary to address issues of OOV, slow training speed, and lacking long context-dependency. Experiments in this paper show that our model has a better performance of predicting tokens compared with the traditional N-gram model and RNN based model. In the meantime, we reduced training time significantly. More broadly, the combination of self-attention and open vocabulary has a potential application in the source code modeling field.**

***Index Terms*—Code Completion, Self-Attention, Source Code Modeling, Open Vocabulary**

I. Introduction

As one part of automatic software development, code completion is always a popular research field in software engineering. Code completion, which refers to recommending the next token based on the current context [1], is a technique that allows us to speed up the coding process and to reduce spelling errors during coding. Nowadays, most programmers use Integrated Development Environment (IDE) like Eclipse and IntelliJ IDEA to write code, enjoying the convenient service of code completion which is a basic feature of modern IDE. Traditionally, code completion in IDE relies heavily on compile-time type information to predict the next token [2]. This method only does well in suggesting attributes or methods of classes but fails to predict coding habits of users. Hindle et al. [3] propose that code has a naturalness and is likely to be predictable and repetitive, so they introduce a statistic language model into the field of source code modeling. In the early stage, the N-gram model used to be the statistic language model used in source code modeling [1], [3]–[5]. Later, when the Recurrent Neural Network (RNN) is introduced into Natural Language Processing (NLP) field, source code modeling has tended to RNN based model [6], [7]. Because the vanilla RNN's variants, Long Short-Term Memory (LSTM) and Gated Recurrent Units (GRU) suppress the problem of gradient exploding and gradient vanishing, they are applied to the field of source code modeling [4], [8]. Although some RNN based deep learning static language models have achieved good results in source code modeling, there are still several defects. It is noted that most models can not predict OOV words, which is called neologisms in [9]. A large number of OOV words will affect the performance of automatic code completion distinctly. Although [10] in ICSE 2020 tried to solve this problem, the method they use aggregates the negative effects of lacking long-range dependency on prediction performance. These defects affect the quality and precision of code completion. This motivates us to propose a novel model to address the issues above. In summary, contributions of this paper include:

- We propose a novel self-attention based automatic code completion neural network. The model addresses issues of unable predicting out of vocabulary tokens, lacking long-range dependency ability, and slow training speed, which are defects of current models.
- We evaluate our model in a real word Java code dataset. Compared with previous work, our model has significant improvements in the metric of Mean Reciprocal Rank (MRR) and entropy in three realistic scenarios. In the meantime, our model spends less time in the training process.
- We design and implement a self-attention based model with Open Vocabulary (SABCCOV), a tool to predict the next token based on current tokens. To the best of our knowledge, we are the first to combine the self-attention with the Open Vocabulary mechanisms in code completion.

The rest of this paper is organized as follows. Section II details some defects of existing current models and background knowledge of self-attention mechanism. Section III presents the design of SABCCOV and training setup. Section IV demostrates experiment details and evaluations of our

*Corresponding Author
DOI reference number: 10.18293/SEKE2020-056.

model. We discuss related work in Section V and conclude in Section VI.

II. Preliminary

In this section, we mainly talk about existing issues of current models and the basis of self-attention mechanism. Sec.II-A describes a common problem in code completion. The issue of Sec.II-B is a critical factor that affects the performance of the model. And Sec.II-C is an aspect that can be improved continuously. At the end of this section, we will describe the self-attention mechanisms.

A. Out of Vocabulary (OOV)

The vocabulary of natural language processing is commonly formed by top k (assume 50,000) frequency words from a large corpus, which is closed because it can only present limited words. And the indexed vocabulary is used to map a word to the index in the statistic language model. Out of vocabulary words in NLP will be represented by 'UNK'. In natural language, it is feasible that OOV words are replaced by the UNK identifier. The NLP model is rarely affected by it since most words are covered by vocabulary. But the programming language model will be affected seriously on account of many identifiers such as variable names, class names, and method names that are defined by the programmer.

B. Long-Range Dependency

Range dependency means that how many context words does the model need to predict the next word. In [11], LSTM language models use 200 context words on average, which has a longer range commonly than Vanilla RNN and GRU. Empirically, the RNN based model is hard to deal with long-range dependencies that are common in programming language [2]. For example, a class is declared at the top of the file, but it may be used after one hundred lines of declaration. The dependency range of the LSTM model may be enough in NLP, but it is not enough for source code modeling.

C. Slow Training Speed

It is noted that the performance of the deep learning model depends on the scale of data. The more training data, the better the performance of the model. In the source code modeling field, it is an advantage that massive amounts of data are easy to get from some open source communities such as Github. Vanilla RNN calculates hidden state one by one to collect sequence information, that is why Vanilla RNN called Recurrent Neural Network. The architecture of Vanilla RNN demonstrates that it is doomed not to support parallel computing, which will have a strong impact on training speed.

D. Self-Attention Mechanism

The self-attention mechanism is proposed in [12]. An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The matrix of outputs can be computed as:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (1)$$

Intuitively, the self-attention function is to help every token in sequence to calculate how much attention needs to pay to other tokens. In the multi-head version of self-attention, the self-attention function takes n different matrices as input and computed for n times, then output n matrices. Then we concatenate all these matrices and condense them into one matric whose dimension is the same as the original self-attention result.

III. Approach

In this section, we describe the procedure of the data preprocessing, which will get more appropriate data for the model than the raw source code. Then we will present the architecture of the overall network from a block perspective and introduce our training strategy.

A. Dataset and Preprocessing

We use the Gig-token corpus in [13], which has more than 14,000 Java projects from Github. We have a large corpus of source Java code, but we can not feed it into the model directly. In the NLP filed, the common way of input data transformation is mapping some high-frequency token to embedding vectors or one-hot vectors according to a vocabulary. But it can not handle the problem of OOV words described in Sec.II-A. It is hard to solve the issue of OOV completely, but the Open Vocabulary method [10] can solve it partially. Using the Byte Pair Encoding (BPE) algorithm, this method learns a segmentation pattern. BPE is a data compression algorithm that iteratively finds the most frequent pair of bytes in the vocabulary appearing in a given sequence, and then replaces it with a new unused entry [14]. The algorithm is first adapted for word segmentation in [15], it merges pairs of characters or character sequences.

As an illustration, we can segment a Java source code file using a BPE vocabulary file as shown in Fig 1. Some high-frequency tokens are reserved and some low-frequency tokens are split into high-frequency sub-tokens followed by

```
package com.testMain;
public class newClass{
  private String someInformation;
  public newClass(String someinformation){
    this.someInformation = someinformation;
  }
}
```

```
package com . test@@ Main ;
public class new@@ Class {
  private String some@@ Infor@@ mation ;
  public new@@ Class ( String some@@ infor@@
      mation ) {
    this . some@@ Infor@@ mation = some@@
        infor@@ mation ;
  }
}
```

Fig. 1. Java code before/after segmentation with an end-of-subtoken '@@'

'@@'. In such a way, OOV words are decomposed into high-frequency subword, which means they can be represented by word embedding vector or one-hot vectors according to the Open Vocabulary.

TABLE I
DATASET STATISTICS

	Full train	Small train	Test	Valid
Projects	13255	107	38	36
Files	1.9M	12K	8.2K	7.1K
Tokens(original)	1.6B	20M	5.9M	4.6M
Toekns(2K)	2.5B	31.9M	9.4M	7.4M
Tokens(5K)	2.2B	27.8M	8.3M	6.5M
Tokens(10K)	2.1B	25.8M	7.7M	6.1M

It is noted that the BPE algorithm needs code to learn segment patterns and produce a vocabulary file. And the number of BPE merging operation affect the performance of the model, so we set three different value, 2k, 5k, and 10k. As Table I shows, data was divided into four parts: full train, small train, test, and valid, which is the same as [4] except moving a small part randomly from the full train projects as a BPE training corpus. Followed [10], we set the size of BPE training corpus to 1000 projects. We provide the procedure of data preprocessing as follows:

1) Remove comments.
2) Replace non-ascii characters with a special identifier (We use '-UNK-' in experiments).
3) Tokenize the file[1], which means every line is one token (Punctuations such as ';' is also considered as one token).
4) As for BPE data, use the BPE algorithm[2] to get a vocabulary file and a segment pattern file.
5) As for train, test, and valid data, use the BPE algorithm with the segment pattern file on these corpora and get segmented files.
6) Add start and end identifiers (We use '<s>' and '</s>') at the top and bottom of every Java file then integrated all files in train/test/valid projects into one file as train/test/-valid file.

After preprocessing, words were split into subwords and subwords can be represented by word embedding, which means most words can be dealt with by the model. Due to the segmentation of the BPE algorithm, we can suppress the ***Out of Vocabulary*** issue effectively. However, it is noted that the number of tokens has increased at least one-third of the original after the procession of the BPE algorithm. It will magnify the negative effects brought by the problem of ***Long-Range Dependency***.

B. Model Architecture

We proposed a self-attention based model with Open Vocabulary as shown in Figure 2.The input and output of the model is a fixed-length sequence. The model has three parts including the **input block**, the **transformer block** and the **output block**. In the input block, the input sequence will be represented by the sum of word embedding and positional encoding, which will be the input tensor of the transformer block. The transformer block has a masked multi-head self-attention layer, a position-wise fully connected feed-forward layer, and some residual connections. The masked multi-head self-attention layer and feed-forward layer are followed by a layer normalization. It looks like the encoder part in [12]. Instead, we replace the multi-head self-attention layer with a masked self-attention layer. The input and output of the transformer block have the same dimension, so it can repeat any time we want. Due to the self-attention mechanism, the transformer block in our model can solve the problems of ***Long-Range Dependency*** and ***Slow Training Speed***. The output block consists of a linear layer and a softmax layer. The output sequence is obtained by the output tensor processed by the argmax function.

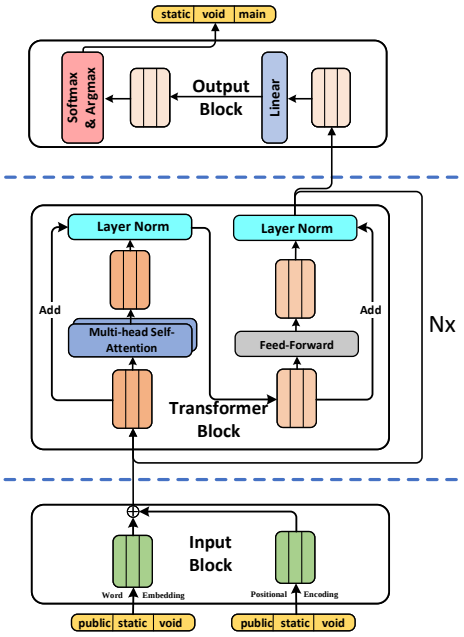

Fig. 2. Overall Architecture: Based on the input of 'public static void main', the model predicts 'static void main'. Rectangles composed of 3 small rectangles represents the intermediate vector.

As Figure 2 shows, the input is 'public static void' and the model will output the sequence of 'static void main'. Predictions for position i can only depend on the known input at positions less than i. The most critical part of the whole model is the self-attention mechanism. Benefits of the self-attention including two aspects. First, the training process can be shortened observably since the self-attention mechanism supports parallelized-training. The RNN based model computes the next hidden units depending on previously hidden units. Unlike the RNN based model, the self-attention based model does not need previously results, so it can be parallelized by vectorization. The second aspect is the self-attention can learn longer-range dependency than LSTM based model. In order to capture the information between two tokens(noted token i and token j), the RNN based model needs at least $|j\text{-}i|$ steps because of the serial computation. And it may lose some information if the distance is too long. But the

[1]The library we use is https://github.com/SLP-team/SLP-Core
[2]We use https://github.com/rsennrich/subword-nmt

vectorization of the self-attention based model can ignore the distance between two tokens in the same segment and correlate them within one step.

The multi-head self-attention layer considers the attention weight of every position of the input, which means the third token in the input sequence may be taken into account when predicting the second position in the output. For example, the model will take 'void' (third token in input) into accounts when it is predicting 'void' (second token in output). But as a prediction task, it is a cheating trick to consider the context that has not shown yet. In [12], the masked mechanism guarantees that the model will not make the prediction based on future information. And this is the reason we replace it with the masked multi-head self-attention layer.

The model we proposed is used to predict sub-tokens, which means it may output part of the correct token (some sub-token is a complete correct token). Further, we use the beam-search-like algorithm [10] to predict complete tokens directly. As Figure3 shows, if the model predicts the token ends with '@@', the model will continue to predict the token until the next predited token without '@@'. Finally, the model will concatenate all the predited subtokens, delete the '@@' and the result is the prediction of the model.

C. Training

As Section III-A described, we have two training sets on a different scale, including small training and full training. Most setup of two training sets is the same. We fix the length of the input sequence feed to the network to 512. If the length of the input is longer than 512, then split it into a few segments and supplement segment whose length is less than 512 with special tokens. The batch size is 32 for the small train and 64 for the full train, respectively. The number of the transformer block is 3, the number of multi-head is 2, and the dropout rate is 0.2. Other hyperparameter of the transfomer block is the same as the encoder in the [12]. The loss function is entropy, which will be discussed in detail in Section IV-B. The optimizer is adam optimizer with an initial learning rate 0.0003 and a learning rate decay strategy. The strategy is that the learning rate will be half if the current epoch's valid loss is bigger than the last epoch. Max training epoch is 50 for the small train and 5 for the full train. If the valid loss of the current epoch has not been decreased for last 4 epoch, the training process will be stopped early.

IV. Experiment

In this section, we will give a description of our experiment scenarios and evaluation metrics. In these experiments, we used the popular deep learning framework Tensorflow. We implemented and evaluated the proposed model on a Linux PC with an Intel i7-5960X processor @3.0 GHz and an NVIDIA GTX 1080Ti GPU.

A. Scenarios

In the code completion task, the next token is predicted based on current tokens. Whether tokens in the current file

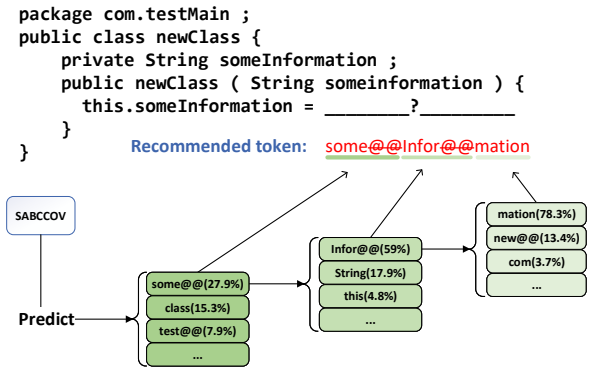

Fig. 3. An example of our model's prediction.

or other files that belong to the same project can be trained for the model is a question. Followed [10], there are also 3 test scenarios including Static, Dynamic, Maintenance in our experiments to check the performance of our model.

Static: This is the basic mode. The model is trained by the large corpus from the internet only. It provides code completion service directly without changed or improved in further usage. In this scenario, local code will not be used for the further training of the model. It may not guarantee to provide a high-quality service of the code completion because everyone has a different code style. The model learned from the large corpus is only a generalization ability.

Dynamic: In this mode, the model has two stage training prcess. The second stage is that, based on the Static mode, the model will be trained by reading the code from the file that the user is editing currently. We denoted it as a local model. With the Dynamic strategy, the model will learn the habits of the coder gradually and prefer to predict the token which reflects the style of the coder finally. The Dynamic strategy here is that the model will read every 512 sub-tokens and executes one step gradient descent to update the model. This mode will have a better performance than the Static mode. However, it will read data that may not be allowed by the user because of security and privacy.

Maintenance: This mode has three stage training process. Based on the Static mode, the model here will use codes in the current project files firstly, which is the second training stage. Lastly, just like Dynamic mode, the model will be trained by reading the code user is editing currently. It is obvious that the model trained in this mode has the best performance than the others since it obtains the largest data set. The disadvantage of this mode is that uploading data in projects may be strictly forbidden in most enterprises.

B. Evaluation Metrics

In our experiments, we evaluated both the intrinsic and extrinsic performance of our model. Intrinsic performance means that evaluating the predicting ability of a language model without other task's inference which may bring some other information. We use entropy as the intrinsic metric, which is a standard measure employed in the previous work. Given a sequence $S = \{t_1, t_2, \ldots, t_{\{|m|\}}\}$, the probability of

TABLE II
ENTROPY RELFECTS THAT THE LOWER ENTROPY, THE MORE REASONABLE THE TOKEN PREDICTED BY THE MODEL.

Entropy		Nested Cache N-gram	Open Vocabulary NLM [10](2k/5k/10k)	SABCCOV(2k/5k/10k)
Small Train	Static	-	4.90 / 4.78 / 4.77	2.62 / **2.57** / 2.63
	Dynamic	2.57	2.33 / 2.27 / 2.54	1.59 / **1.58** / 1.61
	Maintenance	2.23	1.46 / 1.51 / 1.60	1.32 / **1.29** / 1.29
Full Train	Static	-	3.59 / 3.35 / 3.15	1.84 / 1.79 / **1.74**
	Dynamic	2.49	1.84 / 1.72 / 1.70	1.22 / 1.19 / **1.17**
	Maintenance	2.17	1.03 / 1.06 / 1.04	1.08 / 1.02 / **0.99**

TABLE III
MRR REFLECTS THE INVERSE OF THE AVERAGE EXPECTED POSITION IN THE RANK LIST.

MRR		Nested Cache N-gram	Open Vocabulary NLM [10](2k/5k/10k)	SABCCOV(2k/5k/10k)
Small Train	Static	-	62.87% / 63.80% / 63.75%	71.54% / **71.79%** / 71.59%
	Dynamic	74.55%	76.94% / 77.51% / 77.32%	77.54% / **77.72%** / 77.45%
	Maintenance	77.04%	77.48% / 78.49% / 78.69%	78.15% / 78.94% / **78.97%**
Full Train	Static	-	68.69% / 69.87% / 70.84%	77.88% / 78.42% / **78.81%**
	Dynamic	75.0%	78.99% / 79.88% / 80.36%	81.99% / 82.51% / **82.88%**
	Maintenance	77.3%	78.85% / 80.31% / 81.16%	80.71% / 81.78% / **82.30%**

t_i is estimated by $p(t_i|t_1, \ldots, t_{i-1})$. The entropy is defined as:

$$H_p(s) = -\frac{1}{|m|}\sum_{i=1}^{|m|} logp(t_i|t_1, \ldots, t_{i-1}) \quad (2)$$

Entropy corresponds to the average number of bits required in every prediction. But our model predicts a sub-token other than a complete token, we follow [10] and change the subformula in Equation 2 as follows:

$$p(t_i|t_1, \ldots, t_{i-1}) = \prod_{n=1}^{N} p(w_{in}|t_1, \ldots, t_{i-1}, w_{i1}, \ldots, w_{i,n-1}) \quad (3)$$

In Equation 3, we assume that the token t_i is split into sub-tokens $t_i = \{w_{i1}, ..., w_{iN}\}$. The combination of Equation 2 and Equation 3 is the loss function of the model to check whether the model is converging.

The extrinsic performance here we use is Mean Reciprocal Rank (MRR). The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer. MRR is the average of reciprocal ranks or results for a sample of queries Q defined as:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (4)$$

For example, if the correct word ranks first in the option list, then MRR is 1. If the correct word ranks second, the MRR is 0.5 and so on. In our experiments, we provide ten most likely tokens for users to choose, which means the length of the options list is 10, so the MRR of the predicted word is at least 0.1 and the unpredicted word is 0.

C. Results

The results demonstrate that our model is particularly better than the N-gram model [4] and RNN based Open Vocabulary NLM [10]. Our model has a faster training speed as shown in Table IV. In the small training dataset, one training epoch time of our model was almost half of the Open Vocab NLM's training time. As for the full training dataset, our model spent an hour less than the previous work every epoch. Open Vocabulary NLM only has one layer of RNN but our model has three transformer blocks, which means our model has more parameters and is more complicated. But our model was still a lot faster than their model. It is noted that the model with a faster training speed is always the first choice when performance is the same.

Furthermore, our model has not only a fast training speed but also a better performance. Table II and III shows that performance comparison statistics of the Open Vocabulary NLM, the Nested Cache N-gram model and our model. In the metric of entropy, our model with 5k BPE operations had the best results among three models in the small training and our model with 10k BPE operations won in the full training no matter in which scenarios. Due to the large number of samples, even minor improvements (e.g., 0.01 bits) in entropy can be statistically significant in language modeling. It was almost same in the metric of MRR. In Table III, there was a great improvement in the Static scenario between our model and Open Vocabulary NLM and a slight improvement in Dynamic and Maintenance scenarios. It was declared in Section IV-A that Dynamic and Maintenance scenarios need the authorization of reading user code, which may be seen as an invasion of privacy. Under normal circumstances, the Static scenario is the most favored option of users. So it is meaningful to improve the performance of the model in the Static scenario.

TABLE IV
TRAINING TIME(MIN/EPOCH)

	BPE Operations	2k	5k	10k
Small Train	Open Vocab NLM	18	16	16
	Our model	**8**	**7**	**8**
Full Train	Open Vocab NLM	717	669	728
	Our model	**561**	**555**	**628**

V. Related Work

Code completion is a basic feature of IDE for a long time. Traditionally, code completion in IDE relies heavily on compile-time type information to predict the next token [1]. The deep learning method finds a way that learns the probability distribution of tokens from a large source code corpora to improve the accuracy of token prediction. In 2012, Hindle et al. [3] first proposed that programming languages have usefully predictable statical properties that can be captured in statistical language models. Based on Hindle's work, Tu et al. [1] put forward that source code has the property of localness and proposed a cache mechanism to improve prediction accuracy. Hellendoorn et al. [4] enhanced established language modeling approaches to handle the special challenges of modeling source code.

Li et al. [2] and Liu et al. [16] proposed that structural information can also be used to improve the performance of the model. They predicted the terminal node and the non-terminal node in source code using the AST tree. It is critical that the AST tree needed to guarantee both semantic information and structural information when modeling the AST tree. In this method, they can predict not only the token itself, namely the terminal node, but also the token's structural information and type which is a non-terminal node. In 2016, Raychev et al. [17] used the decision tree to model the AST sequence of code to predict token. Liu et al. [16] showed that structural information in AST and sequences of the token can learn mutually and get better results with multi-task learning.

Out of vocabulary words used to be called neologisms [9], which means unseen identifier names that have not been used in the training set. In [9], they split OOV words on camel case and underscores and could only handle part of neologisms. And [2] tried to solve the OOV problem by augmenting an RNN with a pointer network [18]. Some researches focused on the techniques for automatic splitting identifiers [19], [20]. One obvious feature of their splitting technique is that human can understand the sub-identifers after splitting. Instead, Karampatsis et al. [10] first proposed Open Vocabulary that split OOV words to incomprehensible sub-words. Based on their technique, we proposed a novel network in this paper to improve the performance and the training speed.

VI. Conclusions and Future Work

In this paper, we proposed a self-attention based code completion model with Open Vocabulary. First, we alleviated the OOV problem by adopting the Open Vocabulary mechanism. Second, We added the self-attention mechanism to address the issue of sequence augment brought by the Open Vocabulary method. In the meantime, we speeded up the training process and improved the performance of the model. To our best knowledge, we are the first to combine the self-attention with the Open Vocabulary mechanisms and get significant results in the code completion field. Besides, we believe our model may inspire other researchers in the source code modeling field. Our embedding layer is trained during the training process now. In the future, we may train the embedding layer using large corpora in advance. Then we insert it directly into the model and fine tune this layer during the training process. And we also plan to improve our model not to predict only one token but a series of tokens that can present user intent.

Acknowledgment

This work is partially supported by STCSM Projects (No. 18QB1402000 and No. 18ZR1411600), SHEITC Project (2018-GYHLW-02012).

References

[1] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 269–280.

[2] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," *arXiv preprint arXiv:1711.09573*, 2017.

[3] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 837–847.

[4] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 763–773.

[5] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 532–542.

[6] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 334–345.

[7] M. Sundermeyer, H. Ney, and R. Schlüter, "From feedforward to recurrent lstm neural networks for language modeling," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 23, no. 3, pp. 517–529, 2015.

[8] H. K. Dam, T. Tran, and T. Pham, "A deep language model for software code," *arXiv preprint arXiv:1608.02715*, 2016.

[9] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 38–49.

[10] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: Open-vocabulary models for source code," in *International Conference on Software Engineering (ICSE)*, 2020.

[11] U. Khandelwal, H. He, P. Qi, and D. Jurafsky, "Sharp nearby, fuzzy far away: How neural language models use context," *arXiv preprint arXiv:1805.04623*, 2018.

[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[13] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 207–216.

[14] P. Gage, "A new algorithm for data compression," *The C Users Journal*, vol. 12, no. 2, pp. 23–38, 1994.

[15] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," *arXiv preprint arXiv:1508.07909*, 2015.

[16] F. Liu, G. Li, B. Wei, X. Xia, M. Li, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," *arXiv preprint arXiv:1909.06983*, 2019.

[17] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016.

[18] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in neural information processing systems*, 2015, pp. 2692–2700.

[19] E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker, "Mining source code to automatically split identifiers for software analysis," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 71–80.

[20] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "An empirical study of identifier splitting techniques," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1754–1780, 2014.