A Code Completion Approach Combining Pointer Network and Transformer-XL Network

Xiangping Zhang[1,2], Jianxun Liu[1,2(✉)], Teng Long[1,2], and Haize Hu[1,2]

[1] School of Computer Science and Engineering, Hunan University of Science and Technology, Xiangtan 411100, Hunan, China
952259775@qq.com

[2] Hunan Key Lab. for Services Computing and Novel Software Technology, Hunan University of Science and Technology, Xiangtan 411100, Hunan, China

Abstract. Code completion is an integral component of modern integrated development environments, as it not only facilitates the software development process but also enhances the quality of software products. By leveraging large-scale codes to learn the probability distribution among code token units, deep learning methods have demonstrated significant improvements in the accuracy of token unit recommendations. However, the effectiveness of code completion with deep learning techniques is hindered by information loss. To alleviate the above problem, we proposed a code language model which combines the pointer network and Transformer-XL network to overcome the limitations of existing approaches in code completion. The proposed model takes as input the original code fragment and its corresponding abstract syntax tree and leverages the Transformer-XL model as the basis model for capturing long-term dependencies. Furthermore, we integrate a pointer network as a local component to predict the out-of-vocabulary words. The proposed method is evaluated on real PY150 and JS150 datasets. The comparative experimental results demonstrate the effectiveness of our model in improving the accuracy of the code completion task at the token unit level.

Keywords: Code Completion · Transformer-XL · Pointer Network · Out-of-Vocabulary

1 Introduction

The rapid advancement of information technology has increased the complexity of demands placed on software products, resulting in a rise in the difficulty and workload of software development. To address this challenge, improving the quality and efficiency of software development has become a top priority for software engineering researchers. One of the techniques that researchers have been leveraging to enhance the software development environment is code completion technology. This technique is a crucial component of integrated development environments, which predicts class names, method names, and keywords

H. Gao et al. (Eds.): CollaborateCom 2023, LNICST 561, pp. 303–322, 2024.
https://doi.org/10.1007/978-3-031-54521-4_17

in real time, providing developers with multiple options to choose from as they write code. By minimizing spelling errors and enhancing software development efficiency, code completion has become an essential tool in the contemporary software development process [1].

The current research on code completion can be broadly divided into two categories. The first category uses static type information and heuristic rules to predict the token units, such as variable names and method names. However, this approach often overlooks the contextual semantic information of the code, which can lead to inaccurate predictions [2–5]. The second category relies on existing code samples and the semantics of the preceding text to predict the possible token units at the current moment [6–12]. Those works leverage the semantic information of the code to provide more precise and relevant suggestions to developers. Overall, the current research on code completion is aimed at improving the accuracy and efficiency of the software development process by leveraging various techniques, including static type information, semantic information, and heuristic rules. These techniques play a crucial role in the development of modern software products and are constantly evolving to meet the ever-increasing demands of the software engineering industry.

The recent advancement in deep learning techniques has led to a surge of interest in applying these methods to code completion task. By leveraging large-scale codes to learn the probability distribution among code token units, deep learning methods have demonstrated significant improvements in the accuracy of token unit recommendations. However, the important thing to address when using deep learning techniques for code completion task is how to reduce the impact of information loss on the effectiveness of code modeling. The *first* type of information loss in code completion involves the loss of original source code information, which has two aspects. One of the common practices is truncating the source code that exceeds a certain length to be used as input data in deep learning models. However, truncation can result in the loss of valuable information, particularly in dealing with long text data like source code. Table 1 provides statistical information for the two most common datasets used in code completion research. For instance, the PY150 dataset has an average code length of 710.67, and at least 25% of the code fragments in this dataset contain more than 715 words. The JS150 dataset has an average code length of 1739.57, and at least 25% of the code fragments in this dataset contain more than 734 words. Additionally, when converting code into corresponding abstract syntax trees, the number of token units obtained on average is 70% higher than the number of token units in the original codes [13]. Therefore, while deep learning methods have shown potential in improving code completion accuracy, the input length limitation can pose a challenge in handling long text data like source code. The other is the Out-of-Vocabulary (OoV) problem, which arises due to various reasons. Such as, developers often name variables based on their personal preferences during code writing [14]. These variable names tend to occur less frequently than normal token symbols in the entire code corpus, which often results in them being eliminated during code language modeling. As a result, incomplete or inaccurate code suggestions can arise during code completion. To

address these issues, researchers are exploring various techniques such as attention mechanisms, memory networks, and pointer networks that can improve the handling of OoV words in code completion. The *second* type of information loss is caused by the limitations of the neural network model itself. It is widely acknowledged that existing neural network language models have limitations when it comes to modeling long text data. In particular, deep learning-based code modeling approaches that use recurrent neural networks (RNNs) tend to suffer from the problem of forgetting information obtained early in the sequence as the number of time steps increases. For instance, if a variable is defined at the beginning of the source code but used at the end of it, the connection between the variable and its usage may be lost, resulting in less effective code language models and poorer performance in code completion tasks.

In this study, we proposed a code **l**anguage **m**odel which combines the **p**ointer network and **T**ransformer-XL network (PTLM) to overcome the limitations of existing approaches in code completion. Our model takes as input the original code fragment and its corresponding abstract syntax tree (AST) and leverages the Transformer-XL model as the basis for capturing long-term dependencies. Furthermore, we integrate a pointer network as a local component to predict out-of-vocabulary words, which often pose challenges in code completion. Our proposed method is evaluated on real PY150 and JS150 datasets, and the comparative experimental results demonstrate the effectiveness of our model in improving the accuracy of the code completion task at the token unit level. The main contributions of this paper can be summarized as the following three points.

- This paper presents experimental results that demonstrate the effectiveness of PTLM in improving code node value prediction and node type prediction tasks. Specifically, the results show that PTLM outperforms several state-of-the-art models on both the PY150 and JS150 datasets.
- This paper presents a novel approach to OoV token prediction using a pointer network, which allows for direct copying of token units from the original input data. Our method ensures information integrity, which in turn enhances the accuracy of code node value prediction.
- This paper investigates the influence of memory unit length on the performance of Transformer-XL model for code completion task. The experimental results reveal that the ideal length of memory units varies depending on the dataset, and selecting the appropriate length can substantially enhance the accuracy of predictions.

The remainder of this paper is organized as follows: Sect. 2 introduces the related works. Section 3 elaborates the proposed approach. Section 4 introduces the dataset and the experimental settings. Section 5 presents the experimental results. Section 6 concludes the paper.

2 Related Works

2.1 Neural Network-Based Code Completion

Hindle et al. [15] proposed the naturalness of source code that statistical methods could be used to model regular and predictable natural languages.

Tu et al. [16] built on [15] by suggesting that token units in programs have some repetition at a local scale, and that language models often learn only the global laws of the code and ignore the local features in the program. Hou et al. [17] developed an Eclipse plug-in called CACHECA, which combines the Eclipse default completion results with the cache-based N-gram model [18], and found that the performance of the combinatorial approach improved by 1/3 relative to the default plug-in, confirming the effectiveness of local repetition laws for code completion. Vinyals et al. [19] proposed pointer networks, which were initially used to solve combinatorial optimization problems as a variant of neural machine translation models. Pointer networks can be used to efficiently replicate words that need to be reused from input code fragments. Programming languages have local repetition laws, and researchers have considered adding pointer network mechanisms to language models to accurately predict locally recurring words. Bhoopchand et al. [20] proposed a streamlined pointer network that primarily addresses the problem of custom identifier prediction. Li et al. [21] proposed pointer mixture network which combines attention mechanism and pointer network for code completion. The model uses the abstract syntax tree sequence as model input and uses two components and a selector to decide whether to predict the next word in the global word list using the global RNN component or to copy a token from the input code fragment using the pointer network. The model improves the code completion accuracy to some extent, but the input of the model is only the AST sequence, which is an abstract generalization of the source code, and many detailed symbols in the original code fragment are not fully reflected in the AST, e.g., punctuation, operators, etc. This makes the token information in the program incomplete, which in turn affects the accuracy of code completion.

The core idea of code language model to accomplish the code completion task is to estimate the probability of the next occurrence of the token unit symbol in the sequence according to the existing code sequence [22]. For a sequence of code fragment $S = [w_1, ..., w_n]$, the probability of its occurrence is shown as Eq. 1:

$$P_\theta(S) = P_\theta(w_1) \prod_{t=2}^{n} P_\theta(w_t | w_{t-1}, \ldots, w_1) \tag{1}$$

where w_t denotes the t-th token in the sequences of code fragments S and θ denotes all parameters included in this model.

Thus, the purpose of the code completion task at the token level is to find the token w_{t+1} that maximizes Eq. 1 based on the existing sequence of incomplete code fragments S. This process can be described by Eq. 2.

$$\underset{w_{t+1}}{argmax}\, P_\theta(w_1, \ldots, w_t, w_{t+1}) \tag{2}$$

In the traditional approach to code completion, recurrent neural networks are commonly used for feature extraction of tokens in the code. The obtained hidden state vector h_t is then used to calculate the probability of occurrence of the token at the current time step t. As shown in Eq. 3.

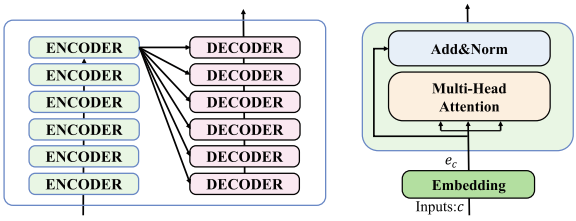
$$P_\theta\left(w_t = \tau | w_{t-1}, \ldots, w_1\right) = \frac{\exp\left(v_\tau^T h_t + b_\tau\right)}{\sum_{\tau'} \exp\left(v_{\tau'}^T h_t + b_{\tau'}\right)} \tag{3}$$

where w_t denotes the t-th token predicted in the code fragment sequence S. h_t denotes the hidden state vector generated by the recurrent neural network at time step t. v_τ denotes the representation vector corresponding to the token τ in the lexicon. b_τ denotes the bias parameter. This approach has limitations in modeling long-term dependencies in the code, as RNNs tend to forget information obtained earlier in the sequence as the number of time steps increases. Additionally, this approach may not effectively handle the Out-of-Vocabulary problem, as it relies on the pre-defined vocabulary set.

Thus most of the existing work can also be considered as using different feature extraction methods to obtain the hidden states that the code sequence has at different time steps for the prediction of the token that appear at the next time step. The hidden states generated in this process need to contain information about the codes that appear in the previous time steps to be able to better predict the possible token units that will appear in the next time step.

2.2 Transformer Model

In recent years, deep learning techniques have been widely used for modeling text sequence data. Recurrent neural networks, particularly Long short-term memory (LSTM) and Gate Recurrent Unit (GRU), were among the most popular model architectures used for this purpose. However, they are limited in their ability to model long dependencies in text sequence data. To address this issue, Vaswani et al. proposed the Transformer neural network model, which uses a self-attentive mechanism to process text sequence data in parallel. Compared to recurrent neural networks, the Transformer model has shown better experimental results and lower resource consumption on a variety of natural language processing tasks [23].



(a) The structure of a Transformer unit (b) The structure of an Encoder

Fig. 1. The structure of Transformer Unit.

A Transformer unit is illustrated in Fig. 1(a). It includes six encoders and six decoders. The encoder contains two layers, a multi-headed attention layer

and a feed-forward neural network layer, respectively. As shown in Fig. 1(b), for a encoder, the input code data c is first converted into the corresponding vector e_c after passing through the embedding layer, after which e_c is fed into a single Transformer. e_c passes through the multi-headed attention layer, and the knowledge contained in the data is extracted from multiple perspectives using a self-attentive mechanism in this layer [24]. The network structure of a single encoder can be represented by the Eq. 4–Eq. 7.

$$e_c = embedding(c) \tag{4}$$

$$z = self\_Attention\_Mechanism\,(e_c) \tag{5}$$

$$n = AddNorm\,(z, e_c) \tag{6}$$

$$o = FFD(n) \tag{7}$$

where c denotes a sequence of code fragments. *embedding* operation denotes converting each token unit in the code sequence c into its respective corresponding word vector e_c. $self\_Attention\_Mechanism$ denotes obtaining the attention score of each element in the code sequence c. $AddNorm$ denotes summing the input data and using the layer normalization operation. FDD denotes a simple feed-forward neural network layer. Equation 8–Eq. 13 represent the specific construction methods for the above three operations.

$$AddNorm\,(z, e_c) = layer\_norm(z) + e_c \tag{8}$$

$$FFD\,(x_l) = W_{l,2} \times GELU\,(W_{l,1} \times x_l + b_{l,1}) + b_{l,2} \tag{9}$$

$$self\_Attention\_Mechanism\,(x_l) = W_O \times softmax\left(Q \times \frac{K^T}{\sqrt{d_k}}\right) \times V \tag{10}$$

$$Q = W_Q \times x_l \tag{11}$$

$$K = W_K \times x_l \tag{12}$$

$$V = W_V \times x_l \tag{13}$$

where $W_{l,1}$, $W_{l,2}$, W_Q, W_K and W_V are all learnable parameters during the training process. $b_{l,1}$ and $b_{l,2}$ denote bias terms. $gelu(-)$ denotes use the *gelu* activation function. Q, K and V denote the query vector, key vector and value vector, respectively, corresponding to the input data after transforming them.

3 The Proposed Method

Figure 2 illustrates the framework of our proposed code completion approach, which contains three parts: code pre-processing, model training, and code completion.

This paper approaches the problem of code completion by dividing the code into two distinct parts. The first part involves processing the source code by utilizing *jieba* library[1] to extract token symbol-level code information. The second

[1] https://pypi.org/project/jieba/.

part involves converting the code into its corresponding abstract syntax tree, and extracting the value and type information of each node by traversing the abstract syntax tree.

During the model training phase, a novel code language model is proposed that combines the Transformer-XL model and the pointer network model to address the issue of modeling long codes. By leveraging the Transformer-XL model's ability to handle long sequence data, high-quality code feature information is obtained. Additionally, the pointer network's capability to select words directly from the context vector as prediction results is utilized to mitigate the impact of the out-of-vocabulary problem in the source code.

Finally, for the code completion part, we adopt the same pre-processing method to handle the code fragments to be completed. After the pre-processing, the code fragments are fed into the trained code language model for prediction, thereby achieving the code completion task.

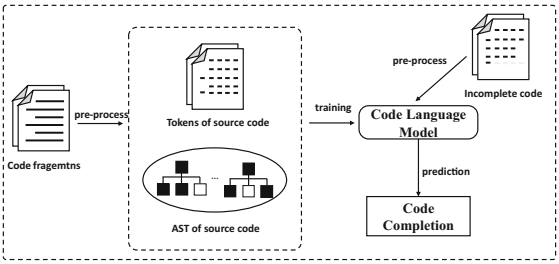


Fig. 2. The framework of code completion

3.1 Code Pre-processing

In previous research on code completion, model effectiveness was validated by predicting the values and types of nodes in the corresponding abstract syntax tree corresponding of the source code. However, the AST is an abstract representation of code information that overlooks some details of symbols in the original code fragment, such as punctuation and operators. This may result in incomplete semantic information in the program, leading to reduced accuracy in code completion. To overcome this limitation, our paper considers the raw text information of the code as input for the code completion model.

This paper employs the word separation function provided by the *jieba* library to pre-process the original text of the code. This function segments the text primarily based on spaces and also distinguishes punctuation from regular words. Figure 3 showcases how a Python code fragment can be divided into 24 token symbols, which denoted as w^d in this paper, following the use of the word separation program in the *jieba* library.

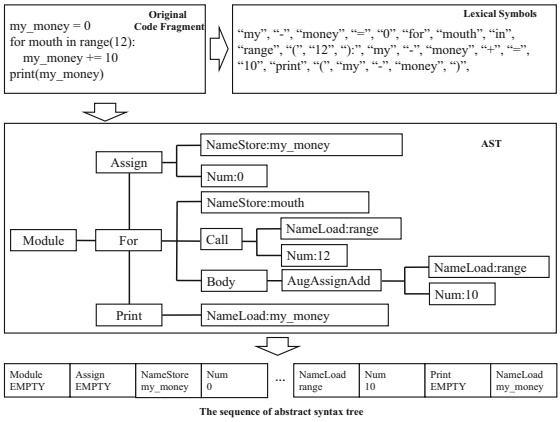


Fig. 3. An example of pre-processing python code fragment

In this paper, we employ the code pre-processing method used in existing code completion work [21] to obtain both the values and types corresponding to the nodes in the abstract syntax tree. First, the code is converted into the corresponding abstract syntax tree. The Python source code are parsed by using the standard library in Python 2.7. The JavaScript source codes are parsed by using the Acorn parser[2]. Then, abstract syntax trees is serialized by performing depth-first traversal on it, and each node data in the sequence is denoted as $w_i^a = [type_i, value_i]$. However, as shown in Fig. 3, non-terminal nodes in the abstract syntax tree do not have corresponding values. Therefore, when the abstract syntax tree is traversed depth-first, a *Empty* symbol is added to these nodes to unify the input data of the subsequent neural network. For example, the Module is transformed into *Module:Empty*.

3.2 Transformer-XL Based Code Language Model

The proposed code language model based on Transformer-XL is structured as shown in Fig. 4. Following code pre-processing, the transformed code data is first converted into vector format before being fed into the neural network for feature extraction. As stated earlier, this study utilizes the token symbols w^d and node symbols w_i^a in the abstract syntax tree. To convert these symbols into corresponding word vector representations, we employ embedding techniques from the PyTorch framework, as demonstrated in Eq. 14–16.

$$e_i^s = embedding\,(w_i^s) \tag{14}$$

$$e_i^{type} = embedding\,(type_i) \tag{15}$$

$$e_i^{value} = embedding\,(value_i) \tag{16}$$

[2] https://github.com/acornjs/acorn.

where e_i^s, e_i^{type}, e_i^{value} denote the vector representation corresponding to the i-th token symbol w^d, the i-th node type symbol and the i-th node value symbol in the original text of the code, respectively.

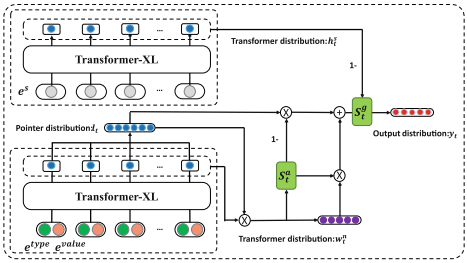


Fig. 4. The structure of the proposed model

This paper utilizes the Transformer-XL neural network model to effectively model the three aforementioned features. The fundamental characteristic of the Transformer-XL network is its segment-level recursive mechanism that establishes connections between various segments via a memory module [24]. The segment-level recursive mechanism in the Transformer-XL neural network enables the modeling of long-range dependencies in the text. Furthermore, this mechanism facilitates information interaction between different fragments and helps to address the fragmentation problem arising from contextual segmentation [24]. As shown in Fig. 5, in the Transformer-XL model, two consecutive fragments $s_\tau = [e_{\tau,1}, ..., e_{\tau,L}]$ and $s_{\tau+1} = [e_{\tau+1,1}, ..., e_{\tau+1,L}]$ of length L are set. The Transformer introduced in the previous section is actually a stack of multiple encoders and decoders, so $h_\tau^n \in R^{L\times d}$ is used to denote the implied state corresponding to the τ-th consecutive fragment s_τ in the n-th layer, and d denotes the length of the implied state. Then, the implied state corresponding to the $\tau+1$-st consecutive fragment $s_{\tau+1}$ in the n-th layer can be obtained from Eq. 17–Eq. 21 as follows.

$$\tilde{h}_{\tau+1}^{n-1} = \left[SG\left(h_\tau^{n-1}\right) : h_{\tau+1}^{n-1}\right] \tag{17}$$

$$q_{\tau+1}^n = h_{\tau+1}^{n-1} W_q^T \tag{18}$$

$$k_{\tau+1}^n = \tilde{h}_{\tau+1}^{n-1} W_k^T \tag{19}$$

$$v_{\tau+1}^n = \tilde{h}_{\tau+1}^{n-1} W_v^T \tag{20}$$

$$h_{\tau+1}^n = Transformer\_Layer\left(q_{\tau+1}^n, k_{\tau+1}^n, v_{\tau+1}^n\right) \tag{21}$$

where the ":" notation indicates that the two vectors are concatenated. $SG(-)$ indicates that the gradient is not back-passed and the implied state from the previous moment is used directly. q, k, v indicate the query vector, key vector and value vector generated from the implied state, respectively.

While the original Transformer model processed long sequential data by dividing the text into segments, it discarded the linkage information between each segment by simply slicing the long text into shorter segments and modeling the sequence. In contrast, the Transformer-XL model records the implicit state information of previous fragments by introducing a memory unit of length d_{mem}. When processing new data, the Transformer-XL model caches and utilizes all the implicit vectors obtained in the previous fragments to participate in the generation of the implicit state of the new fragment. This enables the Transformer-XL to model long-range dependencies in the data. As shown in Fig. 5, the Transformer-XL model appends the implied states generated by the fragments $[e_5^s, e_6^s, e_7^s, e_8^s]$ to the feature generation process of the new code fragments $[e_9^s, e_{10}^s, e_{11}^s, e_{12}^s]$ as well. By setting a memory unit of length $d_{mem} = 4$ to record the implicit state information of previous segments, the Transformer-XL model can solve the long-distance dependence problem. It should be noted that the implicit information of the memory cell is not subject to gradient calculation, i.e., the $SG(-)$ operation in Eq. 17. For more details on the construction of the memory cell, readers may refer to the literature [24].

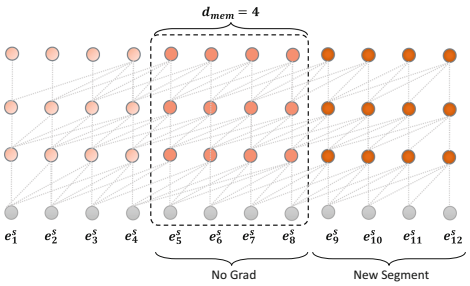


Fig. 5. The diagram of the segment in Transformer-XL

In this study, we first employ the Transformer-XL neural network to model the original text information of the code. As depicted in the upper portion of Fig. 4, the preprocessed raw text information is input into the Transformer-XL neural network. Subsequently, Transformer-XL generates the corresponding implicit state $h_{\tau+1}^s$ for this data, as demonstrated in Eq. 22.

$$h_t^s = Transformer\_XL\left(e^s\right) \tag{22}$$

To obtain the complete node information, we combine the vector information corresponding to the node type and node value data obtained from the abstract syntax tree. This is achieved through stitching the two types of data

together, resulting in the complete node information represented as e^{node}, which is expressed in Eq. 23.

$$e^{node} = \left[e^{type} : e^{value}\right] \tag{23}$$

where the length corresponding to e^{node} is the sum of e^{type} and e^{value}.

The Transformer-XL model is utilized for feature extraction of the complete node information e^{node}, which is fed into the network as shown in Eq. 24.

$$h_t^n = Transformer\_XL\left(e^{node}\right) \tag{24}$$

where h_t^n denotes the implied state output by e^{node} after the last layer of Transformer-XL model.

In this study, we adopt the pointer mixture neural network architecture and utilize an extra cache space M_t to compute the context vector c_t, as described below.

$$A_t = V^T \tanh\left(W^m M_t + \left(W^h h^n\right) 1_L^T\right) \tag{25}$$

$$l_t = softmax\left(A_t\right) \tag{26}$$

$$c_t = M_t l_t^T \tag{27}$$

The context vector c_t, the parent vector p_t and h_t^{node} are stitched together to obtain the Transformer-XL token distribution based vector w_t, as shown in Eq. 28–Eq. 29.

$$G_t = \tanh\left(W^g \left[c_t : p_t : h_t^{node}\right]\right) \tag{28}$$

$$w_t^n = softmax\left(W^v G_t + b^v\right) \tag{29}$$

In the original pointer mixture network, a selector S_t^a is first set for choosing whether to select features from the global or local pointer component for the current moment vocabulary prediction, as shown in Eq. 30.

$$S_t^a = \sigma\left(W^s \left[h_t^n : c_t\right] + b^s\right) \tag{30}$$

In this paper, an additional global selector S_t^g is set on top of this method, which serves to enable the model to select the appropriate words from the code raw text information and the code abstract syntax tree information as the prediction result. s_t^g is calculated as follows.

$$S_t^g = \sigma\left(W^g \left[S_t^a w_t : \left(1 - S_t^a\right) l_t : h_t^s\right] + b^g\right) \tag{31}$$

where h_t^s is the implicit state corresponding to the code text. w^g is the matrix of learnable parameters in the global selector.

The final layer of the PTLM model outputs the probability distribution y_t for the token prediction at moment t.

$$y_t = \left[S_t^g S_t^a w_t : S_t^g \left(1 - S_t^a\right) l_t : \left(1 - S_t^g\right) h_t^s\right] \tag{32}$$

4 Experiments and Settings

4.1 Experimental Data

Table 1. The statistics of two datasets used in this work

Dataset	JavaScript	PY150
Code Number	150000	150000
Word Number	229476577	106601060
Average Words	1739.57	710.67
Lower Quartile	89	99
Median Number	251	317
Upper Quartile	734	715
Node Types	95	330
Node Number	2.6×10^6	3.4×10^6

In this study, two benchmark datasets are utilized, namely the JavaScript dataset (JS150)[3] and the Python dataset (PY150)[4]. The two datasets are publicly available and used in previous work [21]. Each dataset consists of 150,000 original code files, accompanied by their corresponding abstract syntax tree data. Both datasets contain 150,000 program files which are stored in their corresponding AST formats, with the first 100,000 used for training and the remaining 50,000 used for testing [21]. The JavaScript dataset contains 44 original node types, while the Python dataset has 181 node types. We following work [21] and consider 95 and 330 node types for JavaScript and Python, respectively. Table 1 presents the essential statistical details of the datasets used in this study.

In the training process of our proposed method, we use specific parameter settings. The initial learning rate is set to 0.00001 and the learning rate decay value is set to 0.6. We conduct 10 training rounds, and each group of experiments is repeated three times, and the average of the three results is presented as the final outcome. All experiments are performed on a server running Ubuntu 16.04, equipped with two Nvidia RTX2080Ti graphics cards, each with 11 GB of video memory.

4.2 Evaluation

To assess the effectiveness of the proposed method, we adopt the code completion metric accuracy used in prior work [21]. Accuracy refers to the ratio of correctly

[3] https://www.sri.inf.ethz.ch/js150.
[4] https://www.sri.inf.ethz.ch/py150.

predicted completion results to the total number of completion operations. It is calculated using the following formula.

$$Accuracy = \frac{P_{True}}{P_{Total}} \tag{33}$$

where P_{True} indicates the number of correctly predicted token units and P_{Total} indicates the number of all predicted token units.

4.3 Baseline Methods

To analyze the effectiveness of the proposed model in this paper for the code completion task, we selected the most effective existing methods for code completion and conducted experimental comparisons. A brief description of these methods is provided below.

- LSTM [25]. The input of each LSTM neural unit comprises two parts, namely the implicit state of the previous neural unit output and the combined information of node types and values present in the nodes of the abstract syntax tree. Once the feature extraction is completed for the entire sequence, the final implied state output is fed into the feed-forward neural network and the softmax layer for predicting the token unit.
- ParentLSTM [21]. This approach generates a fixed-size context vector within a window by leveraging attention scores from previous and current implicit state computations. The model then uses a concatenated vector consisting of the current node's implicit state, the context vector, and the parent node's implicit state as input to the prediction layer.
- Pointer Mixture Network (PMN) [21]. This method extends the ParentLSTM approach by incorporating two components, namely a global RNN and a local pointer, selected by a selector to predict the next token unit from either the global vocabulary or the local context.
- Transformer [26]. To improve computational efficiency and handle longer sequences of text, this method replaces the LSTM neural network with a Transformer neural network. The Transformer network is better suited to model long sequences of text and has superior computational efficiency.
- Transformer-XL [27]. This method utilizes the Transformer-XL neural network for modeling on sequences of abstract grammar trees, as well as for predicting node-to-root paths, which leverages the structural information inherent in the abstract grammar tree. In the implementation, all task weights are set to 0.5.
- CCMC [22]. This method leverages the Transformer-XL model to perform code completion on the PY150 dataset, utilizing a combination of memory and replication mechanisms to enable accurate predictions of out-of-vocabulary words.
- PTLM. The proposed model. This model divides the code into two components, namely the code text information and the code abstract syntax tree information. To tackle the issue of long dependencies in the code, the

Transformer-XL model is integrated into the model to capture the sequential features of both code components. Moreover, to handle the OoV text, which is commonly observed in code analysis tasks, the model is augmented with a pointer neural network that enables it to predict OoV token units.

4.4 Research Questions

We conducted a series of experiments to assess the efficacy of the approach proposed in this paper, with a focus on addressing the following research questions.

Research Question 1: What is the effectiveness of the proposed methods in this paper for the code completion task?

To answer this problem, we selected several existing code completion methods that have shown strong performance and utilized the same dataset employed by these methods for comparative evaluation of experimental results.

Research Question 2: The effect of different code structures on the experimental effect of code completion.

In the proposed method, we divided the code into two distinct parts: the code's original text and its corresponding abstract syntax tree. Here we want to examine the impact of these three distinct code structures on the efficacy of the code completion task.

Research question 3: The effect of the length of memory cells in Transformer-XL on the code completion task.

To examine the impact of memory cell length on code completion task, we will experiment with varying memory cell lengths in the Transformer-XL model. The length of the memory cell is essential for the model to retain implicit states in sequence data. This research question aims to explore how different memory cell lengths influence the model's performance on the code completion task.

5 Experimental Results and Analysis

5.1 Research Question 1

In this paper we selected six baseline methods to perform experiments on the PY150 and JS150 datasets for prediction of node values and node types in the code completion task, respectively. In the experiments we set the optimizer of all the comparison methods to Adam optimizer and the number of implicit units of the LSTM model to 128 and the learning rate to 0.001. For the PTLM model proposed in this paper, we set its extra cache space length to 100 and the number of layers of the Transformer-XL neural network to two layers.

The first is a comparison experiment on the PY150 dataset. The global vocabulary size of the abstract grammar tree nodes in the dataset is 1000, 10,000 and 50,000, where PY_1K, PY_10K and PY_50K represent the global vocabulary size corresponding to the values of the nodes in the dataset, respectively. The main reason for this phenomenon is that it is difficult for the model to construct information about token units that do not appear in the vocabulary, and a larger

vocabulary will contain more token units related to node values, and the model can learn the features of these token units and make predictions, thus improving the accuracy of the code completion task. Also according to Table 2, we have the following observations and analyses.

1. Methods based on LSTM models, such as LSTM, Attentional LSTM, and Pointer Mixture Network, achieve lower accuracy on node-type prediction tasks than those based on Transformer models. The reason for this phenomenon is that the length of the sequences obtained after converting the code into an abstract syntax tree increases significantly, and the LSTM-like models are less capable of modeling such long sequences than the Transformer-like models, so they are less effective in the type prediction task.
2. Attentional LSTM model is compared with LSTM model, because Attentional LSTM introduces the attention to the code structure information, i.e., the new vector formed by splicing the implicit state of the current node, the context vector and the implicit state of the node's parent node as the input of the prediction layer. Therefore, Attentional LSTM can achieve better prediction results than the native LSTM model.
3. The Pointer Mixture Network (PMN) model can achieve better results than the LSTM and Attentional LSTM models because it introduces a hybrid pointer mechanism to predict the next token unit from the global word list or local context information by setting a selector, which makes it possible to achieve the prediction of OoV token units. When the vocabulary size is 1000, the Pointer Mixture Network model has a significant improvement over the LSTM and Attentional LSTM models in the node value prediction task. This also shows the effectiveness of the Pointer Network in this model.

The PTLM model achieves better results than other methods for the node type prediction and node value prediction tasks. The PTLM achieves significantly better results than the Pointer Mixture Network, Transformer and Transformer-XL models for the node value prediction task because the PTLM model setting up a global selector, a local selector, and obtaining feature information from different data sources.

Table 2. The comparison with different methods on Python dataset

Model	PY_1K		PY_10K		PY_50K	
	type	value	type	value	type	value
LSTM	71.6	63.6	73.2	66.3	74.7	67.3
Attentional LSTM	73.2	64.9	75.5	68.4	77.8	69.8
PMN	75.2	66.6	77.8	68.9	78.4	70.1
Transformer	76.6	64.4	78.4	66.5	78.9	68.0
Transformer-XL	77.4	65.1	79.1	67.2	80.4	69.1
CCMC	78.0	65.0	79.3	69.6	81.4	70.2
PTLM	78.5	68.9	79.9	71.8	82.0	73.0

Also in this paper, experiments are conducted on the JavaScript dataset to compare the effectiveness of different methods on the code completion task. Where JS_1K, JS_10K, and JS_50K denote the global vocabulary size corresponding to the values of the abstract syntax tree nodes in the JS150 dataset of 1000, 10000, and 50000, respectively, the experimental results are shown in Table 3. It can be seen that on this dataset, the better the results achieved by the model on the node value prediction task as the size of the vocabulary table increases because it allows the model to capture more semantic information of the source code.

Table 3. The comparison with different methods on JavaScript dataset

Model	JS_1K		JS_10K		JS_50K	
	type	value	type	value	type	value
LSTM	77.2	69.9	82.6	75.8	84.9	78.6
Attentional LSTM	79.7	71.7	85.0	78.1	86.1	80.6
PMN	81.8	74.2	85.8	78.9	86.6	81.0
Transformer	83.1	74.4	84.7	79.9	85.8	80.2
Transformer-XL	82.2	75.2	85.8	80.8	86.2	81.5
PTLM	84.6	75.9	87.9	81.7	88.9	83.8

5.2 Research Question 2

As mentioned in the previous section, in this work we divide the code into two parts: the original code text and the abstract syntax tree of the code. For the original text of the code, we use a simple word division technique to divide it, and feed the divided token into the proposed code language modeling model for feature extraction. For the abstract syntax tree of the source code, we divide the node information into two parts, which are node type and node value. This section will introduce the influence of these three parts of the code on the experimental effect. The experiments are conducted on the PY150 and JS150 datasets, and the effects of the code completion task on the different datasets are obtained by removing these three structures and their combinations in the PTLM model.

In Table 4 and Table 5, "-source" denotes the model obtained by removing the original code text from the PTLM model. "-source, -type" indicates the removal of the original code text data and the node type data in the abstract syntax tree from the PTLM model, and the rest of the model structure can be followed in the same way. For the case of node type removal, we do not perform the task of node type prediction, so it is indicated by "-" in the results presentation. For the part of original code and node value removal, we do not perform the prediction of node values, and similarly, it is indicated by " -" to indicate the results. From Table 4 and Table 5, we observe the following phenomena.

Table 4. The comparison with different structure of model on Python dataset

Model	PY_1K		PY_10K		PY_50K	
	type	value	type	value	type	value
PTLM	78.5	68.9	78.9	71.8	80.8	73.0
-source	76.4	65.1	77.1	67.2	79.4	69.1
-type	–	64.9	–	65.7	–	66.2
-value	73.2	62.6	75.5	68.4	77.8	69.8
-source, -type	–	60.9	–	63.6	–	64.9
-source, -value	71.5	–	72.4	–	74.4	–
-type, -value	–	55.9	–	56.6	–	58.9

Table 5. The comparison with different structure of model on JS150 dataset

Model	JS_1K		JS_10K		JS_50K	
	type	value	type	value	type	value
PTLM	84.6	75.9	87.9	81.7	88.9	83.8
-source	79.3	68.1	80.2	69.8	82.7	72.0
-type	–	66.5	–	66.9	–	68.1
-value	73.8	64.9	75.0	65.1	76.0	66.8
-source, -type	–	60.9	–	63.6	–	64.9
-source, -value	69.3	–	72.4	–	74.1	–
-type, -value	–	60.9	–	63.6	–	64.9

1. when the PTLM model removes the raw text information of source code (i.e., the "-source" structure result), the effect of the model decreases, especially the prediction accuracy for node values decreases more than the prediction accuracy for node types. This phenomenon suggests that code-source text information can provide information that is not contained in the node values in the abstract syntax tree, and that using code-source text information as model input can improve the effectiveness of the code completion task.
2. When the node type (-type) is removed from the input data, the model is unable to predict the node values. Moreover, comparing the model structures of "-source, -type" and "-source", we can see that the model's effectiveness in the node-value prediction task decreases significantly when the node type is removed. The reason for this phenomenon is that the pairwise correlation between node type and node value splicing can assist the model in predicting node values, e.g., when the node type is known, the prediction range of the corresponding node value is also reduced, thus improving the prediction accuracy.
3. As can be seen in Table 4 and Table 5, the combined structure of the three data types in the code works better for the node type prediction and node

value prediction tasks than the combined case of the other structures. This inspires the need for a comprehensive evaluation of the different features of the code in order to improve the effectiveness of the code analysis task.

5.3 Research Question 3

The parameter examined in this section is the effect of the memory cell length d_{mem} in the Transformer-XL model on the experimental effect. The memory cell lengths of all Transformer-XL modules in the PTLM model are set to 50, 100, 150, 200, 250 and 300, and the rest of the parameters are kept consistent for the experiments. The experimental results are shown in Fig. 6.

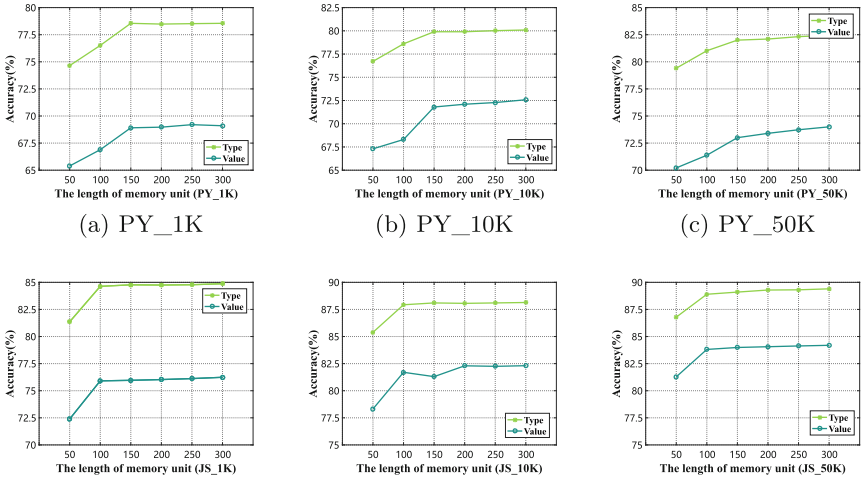


Fig. 6. The effect of different length of memory unit

From Fig. 6(a)–Figure 6(f), it can be seen that the performance of the model starts to stabilize at the memory cell length of 150 when the dataset is the PY150 dataset. In contrast, for the JS150 dataset, the performance of the model starts to stabilize at a memory cell length of 100. A reasonable explanation for this phenomenon is that the different distribution of code lengths in the two datasets makes the length of information to be cached by Transformer-XL different. As shown in Table 1, the upper quartile in the PY150 dataset (715) is larger than the upper quartile in the JS150 dataset (734). This indicates that there are some longer length data in the PY150 dataset, and therefore a larger memory cell space needs to be set for the whole PY150 dataset. This experiment shows that setting the memory cell parameters in Transformer-XL needs to be set according to the statistical information of the specific dataset.

During the experiment, we found that the memory consumption of the graphics card increased with the increase of memory cells, so we set the memory cell size to 150 for the PY150 dataset and 100 for the JS150 dataset in the subsequent experiments.

6 Conclusions

In this work, we have proposed a code language model that combines the pointer network and the Transformer-XL network to overcome the information loss problem of existing code completion approaches. The proposed model takes as input the original code fragment and its corresponding abstract syntax tree, and uses the Transformer-XL model as the base model for capturing long-term dependencies. The proposed method is evaluated on real PY150 and JS150 datasets. The comparative experimental results demonstrate the effectiveness of our model in improving the accuracy of the code completion task at the token unit level. In the next work, we will explore the complementary effects of the Transformer-XL model at the code API level.

Acknowledgements. This work was supported by the National Natural Science Foundation of China (61872139).

References

1. Izadi, M., Gismondi, R., Gousios, G.: CodeFill: multi-token code completion by jointly learning from structure and naming sequences. In: Proceedings of the 44th International Conference on Software Engineering, pp. 401–412 (2022)
2. Yang, Y., Xiang, C.: Improve language modelling for code completion by tree language model with tree encoding of context (S), pp. 675–777 (2019)
3. Fang, L., Huang, Z., Zhou, Y., Chen., T.: Adaptive code completion with meta-learning. In: Proceedings of the 12th Asia-Pacific Symposium on Internetware, pp. 116–125 (2020)
4. Popov, A., Orekhov, D., Litvinov, D.: Time-efficient code completion model for the R programming language. In: Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021), pp. 34–39 (2021)
5. Kyaw, H.H.S., Funabiki, N., Kuribayashi, M.: An implementation of offline answering function for code completion problem in PLAS. In: 2021 IEEE 3rd Global Conference on Life Sciences and Technologies (LifeTech), pp. 162–165 (2021)
6. Raychev, V., Vechev, M., Yahav, E.: Code completion with statistical language models. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 419–428 (2014)
7. Robbes, R., Lanza, M.: How program history can improve code completion. In: 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 317–326 (2008)
8. Proksch, S., Lerch, J., Mezini, M.: Intelligent code completion with bayesian networks. ACM Trans. Softw. Eng. Methodol. **25**(1), 1–31 (2015)
9. Lee, Y.Y., Harwell, S., Khurshid, S.: Temporal code completion and navigation. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 1181–1184 (2013)

10. Nguyen, A.T., Nguyen, H.A., Nguyen, T.T.: GraPacc: a graph-based pattern-oriented, context-sensitive code completion tool. In: 2012 34th International Conference on Software Engineering, pp. 1407–1410 (2012)
11. Omori, T., Kuwabara, H., Maruyama, K.: A study on repetitiveness of code completion operations. In: 2012 28th IEEE International Conference on Software Maintenance (ICSM), pp. 584–587 (2012)
12. Zhang, X., Liu, J., Shi, M.: A parallel deep learning-based code clone detection model. J. Parallel Distrib. Comput. **181**, 104747 (2023)
13. Guo, D., Lu, S., Duan, N.: UnixCoder: unified cross-modal pre-training for code representation. arXiv preprint arXiv:2203.03850 (2022)
14. Shi, J., Yang, Z., He, J., Xu, B., Lo, D.: Can identifier splitting improve open-vocabulary language model of code? In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 1134–1138 (2022)
15. Hindle, A., Barr, E.T., Gabel, M.: On the naturalness of software. Commun. ACM **59**(5), 122–131 (2016)
16. Tu, Z., Su, Z., Devanbu, P.: On the localness of software. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 269–280 (2014)
17. Franks, C., Tu, Z., Devanbu, P.: CACHECA: a cache language model based code suggestion tool. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2, pp. 705–708 (2015)
18. Henkel, J., Lahiri, S.K., Liblit, B.: Code vectors: understanding programs through embedded abstracted symbolic traces. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 163–174 (2018)
19. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Advances in Neural Information Processing Systems, vol. 28 (2015)
20. Bhoopchand, A., Rocktäschel, T., Barr, E.: Learning python code suggestion with a sparse pointer network. arXiv preprint arXiv:1611.08307 (2016)
21. Li, J., Wang, Y., Lyu, M.R.: Code completion with neural attention and pointer networks. arXiv preprint arXiv:1711.09573 (2017)
22. Yang, H., Kuang, L.: CCMC: code completion with a memory mechanism and a copy mechanism. In: Evaluation and Assessment in Software Engineering, pp. 129–138 (2021)
23. Tay, Y., Dehghani, M., Bahri, D., Metzler, D.: Efficient transformers: a survey. ACM Comput. Surv. **55**(6), 1–28 (2022)
24. Dowdell, T., Zhang, H.: Language modelling for source code with transformer-XL. arXiv preprint arXiv:2007.15813 (2020)
25. Liu, C., Wang, X., Shin, R., Gonzalez, J.E., Song, D.: Neural code completion (2016)
26. Kim, S., Zhao, J., Tian, Y., Chandra, S.: Code prediction by feeding trees to transformers. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 150–162 (2021)
27. Liu, F., Li, G., Wei, B., Xia, X., Fu, Z., Jin, Z.: A self-attentional neural architecture for code completion with multi-task learning. In: Proceedings of the 28th International Conference on Program Comprehension, pp. 37–47 (2020)