ISSN: 2688-8645

A Combined Approach of Program Analysis and Deep Learning for Code Completion

Yi Liu*

School of Information, Hunan University of Humanities Science and Technology, Loudi, Hunan, 417000, China

*Corresponding author Email: liuyi8401@gmail.com

Abstract

Code completion, a critical feature in integrated development environments, significantly reduces the coding workload for developers. Traditional code completion techniques often focus on the natural language properties of code, overlooking the structural characteristics of programming languages. To address this issue, this paper introduces a novel approach combining program analysis and deep learning to enhance the accuracy and efficiency of code completion. Initially, the research utilizes program analysis techniques to explore the structural and semantic information of code snippets deeply, constructing a program graph. Graph Gated Neural Networks (GGNN) and Transformer technologies are then employed to represent the program graph, capturing the local features and long-range dependencies of the code. The model integrates both the semantic and structural information of code, thereby providing more accurate completion suggestions. Experimental evaluations were conducted on public datasets for Python and JavaScript, two extensively used programming languages. The results demonstrate that our method significantly outperforms existing approaches in terms of code completion accuracy and Mean Reciprocal Rank (MRR).

Keywords

Code Completion; Program Analysis; Graph Gated Neural Networks; Transformer.

1. Introduction

Many studies have validated the natural language properties of code, inspiring efforts to construct language models for code completion tasks. These models treat code snippets as sequences of characters, considering only the natural language properties of the program and neglecting the structural information inherent in code[1]. Compared to natural languages, programming languages possess clearer and more rigid structural information, typically using explicit control structures (such as loops, conditional statements, and function calls) to express logic and organize code. Programming languages are designed for communication with computers, and feature strict syntax and semantic rules. Every programming language has a clear syntactic structure, defining how to organize code and how each element in the code interacts with others[2]. For example, a simple function definition in Python looks like this:

```
def add(a, b):
    return a + b
```

In this example, *def* is a keyword indicating the start of a function definition; add is the function name; *a* and *b* are parameters; and the return keyword denotes the function's return value. This structure is fixed and precise. Any vague or inaccurate instructions can lead to program errors or unpredictable behavior. Therefore, when performing code completion, it is unreasonable to consider only the natural language properties of the code, treating code elements or nodes of the Abstract Syntax Tree (AST) as mere token sequences[3].

ISSN: 2688-8645

Additionally, understanding of semantic information in programs relies both on local code and on long-distance context[4]. In code, variables, functions, classes, and other entities often have dependencies where the value of one part depends on another. Some dependencies are physically close, such as local variables and function parameters, referred to as the locality of the program. However, some dependencies are physically distant, such as imported packages, global variables, function definitions, and function calls. Relying solely on local context to process and represent these relationships can lead to inaccuracies in understanding the true semantic information of code, increasing the likelihood of errors in code completion. The unique local and global characteristics of programming languages impose special requirements on neural networks: relying on local program context may not be reliable and could miss important contextual information[5]. Most of the current state-of-the-art deep learning solutions for software focus primarily on local information and fail to address this issue[6]. In large projects, there may be multiple variables with the same name but different scopes, leading to confusion between global and local variables and causing deviations in code semantic representation, thereby reducing completion performance.

2. Methodology

Initially, code is collected from open-source community websites like GitHub and Stack Overflow. After cleansing, program analysis is performed on the code snippets to accurately extract the data flows and call relationships within the code, constructing a program graph snippets. Subsequently, the program graph undergoes initial characterization using Graph Gated Neural Networks (GGNN) to capture the local features of the code. On a global scale, a transformer is used to identify the code's long-range dependency relationships. This approach enables capturing of both local features and global information of the code. Prediction nodes are then derived using a multilayer perceptron. The specific research methods are as follows:

(1) Deep Mining of Code's Structure and Semantic Information Through Program Analysis

The process begins by serializing all textual symbols from the code snippets, obtaining a sequence of program symbols: $T =< t_1, t_2, \ldots, t_n >$, where the order of the symbols corresponds to their appearance in the source code. This results in a raw text symbol sequence without program analysis. Subsequent application of program analysis strategies involves using inter-procedural backward slicing to obtain program slices. These slices start from a target node and trace backwards along the data flows reaching these variables, retaining all code statements that influence the target node while excluding irrelevant. This method also collects code contexts that affect outside of the local methods. As illustrated in Figure 1, the program analysis encompasses a collection of data flows and updates along these trajectories.

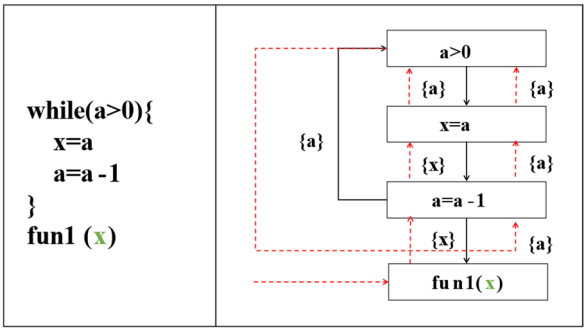

Figure 1. Schematic Diagram of Program Analysis for Code Snippets

ISSN: 2688-8645

This results in the corresponding program graph $G = (V, E)$ for the code snippets, where V represents the set of vertices (nodes) and E is a list of sets of directed edges of different types, i.e., $E = (E_1, \dots, E_k)$, where k is the number of edge types.

(2) Research on Code Representation Methods Based on Program Graphs

The initial characterization of the program graph is conducted using GGNN. Firstly, each node v is initialized with $x^{(v)} \in R^D$, denoted by $h^{(v)}$. In the GGNN layer, messages are sent from each node $v \in V$ to its neighbors, with the edges being updated as follows:

$$m_k^{(v)} = LinearLayer_k^{(h_{(v)})} \tag{1}$$

After the message passing step, the set of messages at each node is aggregated as follows:

$$m^{(v)} = \sum_{e_k(u,v)\in\varepsilon} m_k^{(u)} \tag{2}$$

After receiving the updated edges, the state vector of node v is updated to:

$$h_{new}^{(v)} = GRU(m^{(v)}, h^{(v)}) \tag{3}$$

Finally, each node tv is processed through a transformer to compute its queries, keys, and values, denoted as q_t, k_t and v_t, to obtain the representation of the code snippet. The attention values for each node are calculated using Equation (4) and then normalized:

$$e_{ij} = {(q_i + b_{ij})k_j^T}\Big/{\sqrt{N}} \tag{4}$$

(3) Using a Feed-Forward Neural Network to Predict Nodes

Firstly, the updated state vector of each node, $h_{new}^{(v)}$, is input into a feed-forward neural network. The network consists of a Multilayer Perceptron (MLP) constructed by stacking multiple linear layers and nonlinear activation layers to capture complex relationships and dependencies between node features. Each layer uses ReLU as the activation function to enhance the model's nonlinear expressive power. The final layer is a linear layer used to generate the final predictions for each node. Each element of the output vector represents the probability of a specific type of code element (such as variable names, function calls, etc.).

During training, the network is optimized using the cross-entropy loss function to accurately predict missing node information. Additionally, regularization techniques, such as L2 regularization, can be introduced to prevent the model from overfitting and maintain its generalization ability on unseen data.

Ultimately, the model scores each node and selects the code element with the highest probability as the completion suggestion. This process not only speeds up the coding but also significantly enhances code quality by providing contextually relevant code snippets.

3. Experimental Evaluation

To comprehensively assess the performance of the proposed code completion method, we selected public datasets for two widely used programming languages: Python and JavaScript. These datasets contain a vast array of code snippets and complete project files, providing with ample experimental material to test and validate our approach.

The experiment primarily includes the following steps:

(1) Data Preparation: Randomly select training, validation, and test sets from the Python and JavaScript datasets, with the training set comprising 70%, the validation set 15%, and the test set 15%. Ensure each collection has sufficient code examples to represent the diversity found in the real world.

(2) Model Training: Utilize the program graph-based code representation method and the feed-forward neural network for node completion described earlier to train the model on the

ISSN: 2688-8645

training set. We employ the Adam optimizer to adjust the model weights, with a learning rate of 0.001 and a batch size of 128.

(3) Parameter Tuning: Optimize the model configuration by evaluating different hyperparameter settings on the validation set.

(4) Performance Evaluation: Run the model on the test set to collect and analyze metrics such as the accuracy of code completion and Mean Reciprocal Rank.

Table 1. Results of our approach

Dataset	JS		PY	
Accuracy	81.37%	82.16%	72.95%	73.94%
MRR	8645%	87.02%	80.33%	81.28%

These results demonstrate that by integrating program analysis techniques and deep learning methods, our code completion system can effectively identify and complete code snippets, thereby enhancing development efficiency and code quality. The table shows improved performance metrics in terms of both accuracy and mean reciprocal rank for Python and JavaScript datasets, indicating the robustness of our proposed approach.

4. Discussion

The analysis of the experimental results reveals that the system performs slightly better with Python code than with JavaScript. This difference may be related to the characteristics of the datasets and the structural complexity of the programming languages. Additionally, the experiments observed a decline in model performance in more complex code structures, such as nested loops and deep function calls, indicating a need for further optimization of the model to handle more complex coding scenarios. Future work will explore more advanced graph neural network models and improved training strategies to enhance the model's generalizability and accuracy across various programming languages. This approach aims to address the nuanced challenges inherent in diverse programming environments and refine the effectiveness of code completion tools.

5. Conclusion

This study proposes a method for code completion that combines program analysis techniques with deep learning, fully leveraging the structural and semantic information of code to enhance the accuracy and efficiency of code completion. By constructing program graphs of code snippets and employing GGNN and Transformer technologies for local and global information representation, our approach enhances the model’s capability to handle complex code structures and better captures long-distance dependencies within the code. Experimental results demonstrate that this method performs well on datasets for Python and JavaScript, two widely used programming languages.

Moreover, we have identified limitations in the current model's performance when dealing with highly nested code and multilevel function calls. Future work will focus on exploring more advanced graph neural network structures and optimizing training strategies to further improve the model's generalizability and applicability across a broader range of programming languages.

Acknowledgments

This work is supported by the Research Project of Hunan Provincial Education Department [Grant No.22C0600].

ISSN: 2688-8645

References

[1] Hindle A, Barr E T, Gabel M, et al. On the naturalness of software[J]. Communications of the ACM, 2016, 59(5): 122-131.

[2] X. Jin and F. Servant. The hidden cost of code completion: Understanding the impact of the recommendation-list length on its efficiency. in Proceedings of the 15th International Conference on Mining Software Repositories, pp. 70-73, 2018.

[3] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems, in Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp. 213-222, 2009.

[4] 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, pp. 269-280, 2014.

[5] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin. A unified multi-task learning model for ast-level and token-level code completion, Empirical Software Engineering, vol. 27, no. 4, p. 91, 2022.

[6] 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.