Enhancing Code Completion with Implicit Feedback

Haonan Jin, Yu Zhou*, and Yasir Hussain
College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, China
jinhaonan@nuaa.edu.cn, zhouyu@nuaa.edu.cn, yaxirhuxxain@nuaa.edu.cn
*corresponding author

Abstract—Code completion has become an important feature of today's integrated development environments (IDEs). This task involves predicting the next code token(s) based on its contextual information within the code. However, most existing code completion approaches do not consider users' feedback during the completion process. In this paper, we propose a framework, EHOPE (Enhance Code Completion with Implicit Feedback), which exploits LSTM(Long Short-Term Memory) and pre-trained model BERT(Bidirectional Encoder Representation from Transformers) to enhance the performance of token-level code completion. By leveraging users' feedback information, we train an LSTM model to supplement the recommendation list. In addition, we re-rank the list of recommendations using the pre-trained model BERT, which is fine-tuned with feedback information. Existing token-level code completion tools can be plugged into EHOPE. We choose two representative code completion approaches from different categories: one based on statistical methods and the other based on deep learning. These approaches serve as baselines to showcase the performance improvements of EHOPE, evaluated using Hit@k (Top-k) and MRR(Mean Reciprocal Rank) metrics. Empirical experiments show that the recommendation performance steadily and substantially improves as the feedback data increases compared with the baselines.

Keywords–Code Completion; Code Suggestion; Deep Neural Networks; Pre-training Model

1. Introduction

Integrated development environments (IDEs) have become essential paradigms for modern software engineers, as IDEs provide a set of helpful services to accelerate software development [1]. One of the most beneficial functions in IDEs is intelligent code completion, which uses the context of existing code to suggest the next probable code tokens. While some code completion approaches rely solely on the given context [2], [3], [4], some methods incorporate more extensive information, such as code token types [5], abstract syntax tree (AST) structures [1], [6], or extended hierarchical context [7]. Nonetheless, the existing approaches are limited in the scope of information they utilize. Only a few of these approaches take users' feedback into account in the recommendation process [8]. Such information is often critical in enhancing the performance of code completion.

Feedback information refers to the data collected and analyzed from users' interactions with a product, service, or system during a recommendation session. In conventional recommendation systems [9], feedback information can significantly improve the accuracy of recommendations [10], [11]. Typically, this feedback is implicit rather than explicit, such as users' ratings. By observing users' behavior, including their selections, duration, repetition, purchases, and more [12], we can gather implicit feedback that indirectly reflects their opinions [13]. During the process of code completion, selecting a recommended token from the recommended list typically signifies its usefulness in helping the user complete their code. Therefore, in this process, the user's choice can be seen as a kind of implicit feedback. In fact, feedback from programmers often leads to the correct answer for code completion, playing an important role in handling similar scenarios and improving the performance of the recommendation system in the future [14], [15]. This underscores the critical role that feedback plays in code completion systems, which may be even more significant than in traditional recommendation systems. Prior research in code completion has largely overlooked the importance of incorporating users' feedback. Traditional code completion techniques ignore the valuable insights that users can provide. These approaches often fall short of accurately predicting the desired code snippets and fail to capture the context-specific preferences and nuances of individual developers.

Without considering users' feedback, code completion systems often struggle to handle complex and evolving programming scenarios. They may produce suboptimal suggestions that do not align with the developer's intentions, leading to frustrating and time-consuming manual edits. This limitation hinders the productivity and efficiency of developers, especially when dealing with unfamiliar or intricate codes.

In this paper, we introduce a framework named EHOPE (**En**hance C**o**de Completion with Im**p**licit F**e**edback) that aims to enhance recommendation performance through the utilization of implicit feedback information. We keep track of the code segment being completed along with the user-selected token during each code completion session and store these pairs in our feedback repository. Tokens are regarded as feedback information for code segments. By integrating feedback information, our framework not only improves the effectiveness of token-level code completion but also enables personalized recommendations. Specifically, based on users' personal interaction history, EHOPE generates distinct recommendation lists for each user when completing the same code segment. Moreover, our framework can incorporate existing recommendation methods as components, providing a flexible

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00030

and customizable solution for various recommendation scenarios.

To efficiently incorporate users' feedback into our code completion system, we utilized an LSTM model, which was trained with feedback data. By leveraging the LSTM model's ability to learn from feedback, we can expand the initial recommendation list produced by the third-party code completion tool to include feedback information. Additionally, we employed a pre-trained BERT [16] model to re-order the list of recommendations. BERT is a deep transformer model that has been pre-trained for language modeling, and it has been widely acknowledged for its remarkable performance in multiple classification and sequence labeling tasks. Notably, Nogueira and Cho [17] were the first to showcase its effectiveness in ranking tasks. We employ feedback information to fine-tune the model. Then we construct a sentence pair, comprising the code segment to be completed and each recommendation item in the list, as input of the next sentence prediction (NSP) task and feed it into BERT. By doing so, we could determine whether the recommendation item was the next sentence to follow the code segment being completed. Based on BERT's predictions, we were able to reorganize the list of recommendations. In general, we use the deep neural network and the pre-trained model that learn from the feedback repository and work together to optimize recommendations.

To illustrate the effectiveness of EHOPE, we have opted for N-gram [18] and CodeGRU [2] as our baseline models. N-gram stands as a representative of statistical-based code completion methods, while CodeGRU exemplifies deep learning-based code completion approaches. The rationale behind selecting these two baselines lies in twofold: firstly, they epitomize distinct methodological paradigms in the realm of code completion, allowing for a comparative assessment of their performance. Secondly, N-gram, a venerable technique, possesses a rich history of application and widespread use in code completion research, while CodeGRU embodies the contemporary shift toward deep learning in the field. We assessed the performance of these baselines using Hit@k/Top-k accuracy and MRR metrics. With the continuous accumulation of feedback information, we were able to improve the Top-1 accuracy of N-gram and CodeGRU by 28.04% and 7.99%, respectively.

The following are the primary contributions of this paper.

- We introduce a novel framework, EHOPE, which enhances token-level code completion accuracy by integrating programmers' feedback information through the use of LSTM and pre-trained model BERT.
- We present the results of a thorough empirical study in which we compare EHOPE to two widely adopted code completion systems. Our findings demonstrate that EHOPE outperforms these systems.

Our aim is not to propose yet another recommendation method, but rather to improve the performance of token-level code completion and make it applicable to a wide range of existing completion systems. To the best of our knowledge, this work represents one of the initial attempts to incorporate LSTM, a pre-trained model, and feedback information in code completion, thereby enhancing its accuracy and efficiency.

Structure of the paper. In section 2, the background of this research is introduced, followed by section 3 which provides the details of our approach. Section 4 presents the experimental settings and comparative results on related code completion systems, while sections 5 and 6 discuss threats to validity and related work respectively. Lastly, section 7 outlines the future research and draws a conclusion.

2. BACKGROUND

2.1 LSTM

Long Short-Term Memory (LSTM) is a type of recurrent neural network (RNN) that is designed to overcome the issue of vanishing gradients in traditional RNN. Unlike traditional RNN, LSTM is capable of selectively retaining or forgetting information over long periods of time.

LSTM consists of a network of memory cells that are connected through a set of gates, including the input gate, forget gate, and output gate. The input gate determines which information from the input should be stored in the memory cell, the forget gate decides what information should be discarded from the memory cell, and the output gate determines what information should be outputted from the memory cell. The structure of the LSTM is shown in Figure 1.

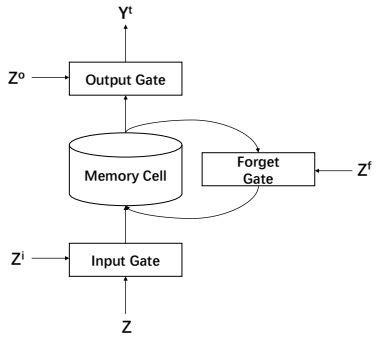


Figure 1: LSTM architecture.

Y^t represents the output at time step t. And Z denotes the preprocessed input, whereas Z^i, Z^f, and Z^o correspond to the gate control signals of the input gate, the forget gate, and the output gate, respectively. And the calculation formula of Z, Z^i, Z^f and Z^o is as follows.

$$Z = \tanh(W \cdot [x^t, h^{t-1}] + b) \quad (1)$$

$$Z^i = \sigma(W^i \cdot [x^t, h^{t-1}] + b^i) \quad (2)$$

$$Z^f = \sigma(W^f \cdot [x^t, h^{t-1}] + b^f) \quad (3)$$

$$Z^o = \sigma(W^o \cdot [x^t, h^{t-1}] + b^o) \quad (4)$$

x^t represents the input at the current time step and h^{t-1} represents the hidden state at the previous time step.

The calculation process of the output Y_t at time step t is as follows.

1) Calculate the cell state C^t at time step t. $\circ$ represents the Hadamard product.

$$C^t = Z^f \circ C^{t-1} + Z^i \circ Z \tag{5}$$

2) Calculate the hidden state h^t at time step t.

$$h^t = Z^o \circ \tanh(C^t) \tag{6}$$

3) Calculate the output Y_t at time step t.

$$Y^t = \sigma(W^y \cdot h^t + b^y) \tag{7}$$

The LSTM architecture allows for the storage and retrieval of information over long periods of time, making it especially useful in applications that require the modeling of complex and structured sequential data. Additionally, the ability to selectively remember or forget information at each time step gives LSTM a unique advantage over other types of RNNs in modeling long-term dependencies in data.

Overall, LSTM has been proven to be a powerful tool in the field of deep learning, and its ability to capture and model long-term dependencies in sequential data has led to its widespread adoption in a variety of applications, including speech recognition [19], [20], natural language processing [21], [22], [23], and time series prediction [24].

2.2 BERT

BERT (Bidirectional Encoder Representations from Transformers) is a powerful pre-trained model developed by Google. Its architecture is shown in Figure 2. It is designed to capture the meaning of natural language more accurately than previous models by using two pre-training tasks: masked language modeling (MLM) and next sentence prediction (NSP). In MLM, BERT is trained to predict a randomly masked word in a sentence, forcing it to consider the entire sentence for context. In NSP, BERT is trained to predict whether two sentences are contiguous or not, encouraging it to understand the relationship between sentences.

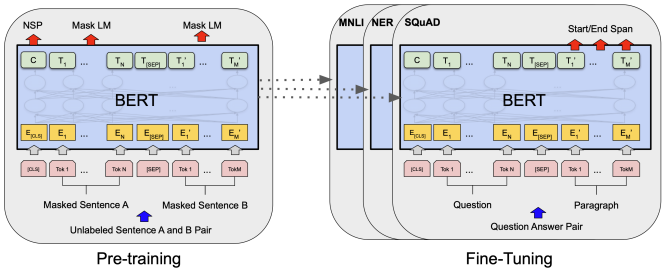


Figure 2: BERT architecture [16].

BERT has revolutionized the field of natural language processing by achieving state-of-the-art performance on a range of tasks, including text classification, question answering, and language generation. Its ability to handle both context and ambiguity makes it a powerful tool for tasks that require understanding natural language. Furthermore, BERT can be fine-tuned on downstream tasks with relatively small amounts of task-specific data [25], and this availability has made it easier for researchers and developers to apply BERT to their own NLP problems [26], allowing for faster and more accurate results. Overall, BERT has become a widely adopted and influential pre-trained language model in the field of natural language processing, offering a powerful tool for a range of practical applications.

3. Approach

As depicted in Figure 3, the EHOPE framework is primarily comprised of four components.

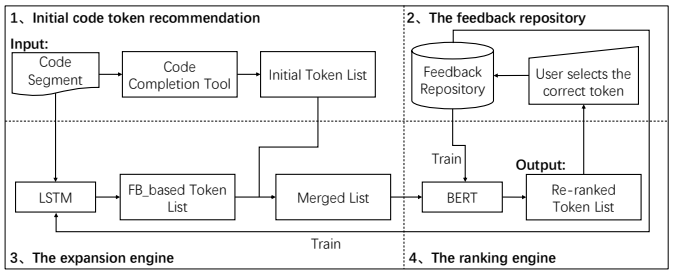


Figure 3: The overview of EHOPE.

a) *Initial code token recommendation*. An initial list of recommended tokens is generated based on the given code segment input. This list is obtained by utilizing the existing algorithms for token-level code completion applied to the input code segment.

b) *The feedback repository* consists of pairs of code segments and their corresponding recommended tokens. Formally, it can be represented as a set of pairs (CS, Tk), where CS represents a code segment and Tk represents the recommended token chosen by users. As users select tokens from the token list, their choices are recorded in the feedback repository as pairs of the code segment and selected token. Initially, the feedback repository is empty, but it grows as users interact with the system.

c) *The expansion engine* which supplements the initial token list when a code segment is given. For each code segment, the LSTM model trained with feedback data generates its own list of recommendations, which can be merged with the initial token list (cf. Section III.A).

d) *The ranking engine* which ranks the recommended tokens for a code segment. To this end, the engine applies the pre-trained model BERT which is fine-tuned with feedback data to predict the score (cf. Section III.B).

Our approach can be summarized in the following workflow.

1) When a user provides a code segment CS to the system, a base code completion method is employed to provide an initial token list L_{CS}.

2) The expansion engine receives the code segment to be completed and generates the token recommendation list L_{FB}, along with the probability value of each token based on the feedback information. Then, L_{FB} is combined with the L_{CS} to form a merged list L_{MG}, with the tokens ordered by their probabilities in decreasing order.
3) The ranking engine takes CS and L_{MG} as inputs and applies the pre-trained model BERT fine-tuned with feedback data to determine the NSP relationship between CS and each token in L_{MG}. Then L_{MG} is reordered based on BERT's prediction, and the system presents new recommendations to the users.

The feedback repository is a critical component of our framework, which is continuously updated throughout the system's lifetime based on user interactions. At the initial stage, the feedback repository is empty, so EHOPE provides an initial token recommendation list as output. When users (e.g., programmers) are presented with recommended tokens, they are implicitly asked to label the most relevant ones as the "ground-truth" recommendations for the given code segment. This feedback, in the form of code segment-token pairs, is stored in the feedback repository. With more user interactions, the feedback repository gradually grows and becomes more comprehensive.

In general, the feedback repository is used in training the LSTM and BERT. However, to improve efficiency and avoid unnecessary computational overhead, we do not retrain these models every time new feedback data is added to the repository. Instead, we schedule retraining to occur after a certain amount of new data has been collected. By doing so, we can balance the need for high-ranking precision with the practical constraints of the system.

3.1 Supplementing recommendation token list

In this section, we outline the operations of the expansion engine. As previously mentioned, we utilize an LSTM neural network to learn from the feedback repository and generate token recommendations for the code completion task. The LSTM takes in a code segment as input and produces a sequence of hidden states based on its internal memory cell and operations of three gates. Specifically, the LSTM's internal memory cell allows it to remember relevant information from previous tokens, while its input gate operation controls the flow of new information into the memory cell. Under the operation of the output gate, the output of the LSTM is the final hidden state, which is then fed into a fully connected layer to generate the token recommendation list L_{FB}, where each recommendation has a corresponding probability value. The initial token recommendation list L_{CS} generated by the basic code completion tool also has corresponding probabilities. The two lists are then merged by sorting the tokens in order of their probability from greatest to smallest, resulting in the merged list L_{MG}. Mathematically, according to equation (1-6), this can be represented as follows:

LSTM output:

$$h^t = f(W^h \cdot h^{t-1} + W^x \cdot x^t) \tag{8}$$

Token recommendation:

$$L_{FB} = \text{softmax}(W^y \cdot h^T + b^y) \tag{9}$$

List merging:

$$L_{MG} = \text{sort}(L_{FB} \cup L_{CS}) \tag{10}$$

Here, x^t represents the input token at time step t, h^t represents the hidden state at time step t, W^h and W^x are the weight matrices for the hidden state and input token, respectively, and W^y is the weight matrix for the output layer. The function f is the LSTM cell operation, which consists of the internal memory cell and gate operations.

3.2 Re-ranking recommendation token list

In this section, we provide an overview of the functions performed by the ranking engine. To further improve the recommendation quality, we fine-tune a pre-trained BERT model using the feedback repository. Each token in the L_{MG} is treated as a separate sentence, and we construct two sentences by concatenating the code segment and each token in the L_{MG} recommendation list, respectively. The special [CLS] token is added at the beginning of the first sentence to indicate the classification task, and the [SEP] token is used to separate the two sentences. BERT's input structure is shown in Figure 4.

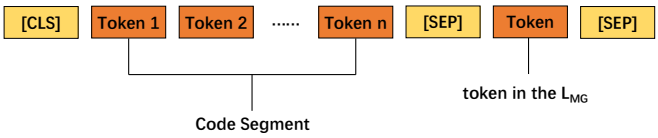


Figure 4: BERT's input structure.

The input to the fine-tuned BERT model is a sequence of tokens, and the output is a sequence of hidden states. The [CLS] token is used as a special token to encode the aggregated information of the entire sequence, which is used for downstream tasks such as classification. The classification output of the [CLS] token can be computed using the following equation.

$$\text{ClsOutput} = \text{softmax}(W^{out} \cdot h^{CLS} + b^{out}) \tag{11}$$

where h^{CLS} is the hidden state of the [CLS] token, and W^{out} and b^{out} are learned parameters of the classification layer.

The concatenated sentences are then fed into the fine-tuned BERT model, and the output of the [CLS] token is used to determine whether the token is the next sentence of the code segment. Specifically, the predicted probabilities are used to

reorder the L_{MG}, with the tokens predicted as "yes" positioned before those predicted as "no". Tokens predicted as "yes" and sharing the same predicted probability are kept in their original relative order. This process results in a reordered list that we return to the users.

4. Evaluation

In this section, we assess the effectiveness of our proposed EHOPE approach. To achieve this, we address the following research questions (RQs).

RQ1 How effective is our method to complete code for given code segments in general?

RQ2 How does the utilization of feedback information enhance the performance of EHOPE in code completion? Specifically, how does the size of the feedback repository affect the performance of code completion?

RQ3 How do LSTM and BERT contribute to EHOPE respectively?

4.1 Baselines

The EHOPE approach can be regarded as a supplementary technique that can be integrated with existing token-level code completion systems. In this work, we choose two representative systems, N-gram and CodeGRU, as baselines for comparison.

N-gram [18] is a widely used statistical language model that predicts the next word in a sequence based on the previous N-1 words. In our experiment, we implement an N-gram model ourselves.

CodeGRU [2] is a context-aware source code modeling model that performs well in code completion. It collects open-source Java projects from GitHub to build training and testing datasets. The training dataset was constructed from a project named "antlr", which had 56,085 lines of code and 407,248 code tokens. The testing dataset was constructed from a project named "batik", which had 195,652 lines of code and 1,246,157 code tokens.

For our experiments, we use the existing datasets provided by CodeGRU. However, due to resource constraints, when comparing with these two baselines, instead of using the entire testing dataset, we randomly selected 20,000 lines of code to test. For each line of test code, the last token is the correct recommendation, which we treat as (pseudo) feedback from the user in the experiment.

4.2 Performance metrics

We adopt two commonly used metrics in the literature (e.g., [2], [27], [28]) to evaluate the effectiveness of our approach.

- Hit@k/Top-k Accuracy, witch measures the percentage of correct predictions among the top-k ranked candidates suggested by the system. In other words, given a set of k predictions, the top-k accuracy measures how many of them are correct. For example, if k is 5 and the correct prediction is among the top 5 candidates, the system is considered to have achieved a top-5 accuracy. Formally,

$$Hit@k = \frac{cnt(k)}{|C|} \tag{12}$$

where $cnt(k)$ represents the number of code segments whose correct next token appears in the top-k, and $|C|$ is the total number of the code segments.

- MRR (Mean Reciprocal Rank) is a metric that measures the performance of a code completion model by taking into account the location index of the correct token in the predicted list. Specifically, it calculates the reciprocal of the index of the correct token and takes the mean value of all the reciprocal values obtained for all the test cases.

$$MRR = \frac{1}{|C|}\sum_{i=1}^{|C|}\frac{1}{rank_i} \tag{13}$$

where $rank_i$ represents the ranking position of the correct token in the $i-th$ code segment.

4.3 Experimental Results

RQ1. How effective is our method to complete code for given code segments in general?

In our experiments, we evaluate the effectiveness of our approach on the testing datasets. After each test, the code segment and the user-selected token (which in the experiment is the correct token) are considered as (pseudo) feedback and added to the feedback repository. Once the number of feedback increases by 4000, the LSTM and BERT are retrained. We compare the results of N-gram and CodeGRU baselines before and after using EHOPE framework, and the results are shown in Table 1. 'Original' indicates the result of N-gram or CodeGRU.

TABLE I: Evaluation results for our framework comparing with baselines ('Abs. imp.' stands for 'absolute improvement'; 'Rel. Imp.' stands for 'relative improvement')

Approach	Technique	Hit@1	Hit@3	Hit@5	Hit@10	MRR
EHOPE + N-gram	Original	0.2194	0.2578	0.2653	0.2702	0.2394
	ours	**0.4998**	**0.6319**	**0.6759**	**0.7191**	**0.5764**
	Abs. Imp.	28.04%	37.41%	41.06%	44.89%	33.70%
	Rel. Imp.	127.80%	145.11%	154.77%	166.14%	140.77%
EHOPE + CodeGRU	Original	0.5187	0.6896	0.7397	0.7847	0.6139
	ours	**0.5986**	**0.7576**	**0.8056**	**0.8425**	**0.6867**
	Abs. Imp.	7.99%	6.8%	6.59%	6.78%	7.28%
	Rel. Imp.	15.40%	9.86%	8.91%	8.64%	11.86%

Table I indicates a significant improvement in almost all metrics compared to the baselines. Specifically, our EHOPE approach outperforms the baselines by 127.80%, 145.11%, 154.77%, 166.14%, 140.77% for N-gram and 15.40%, 9.86%, 8.91%, 8.64%, 11.86% for CodeGRU. This result demonstrates the effectiveness of our feedback repository in enhancing the performance of code completion.

RQ2. How does the size of the feedback repository affect the performance of EHOPE?

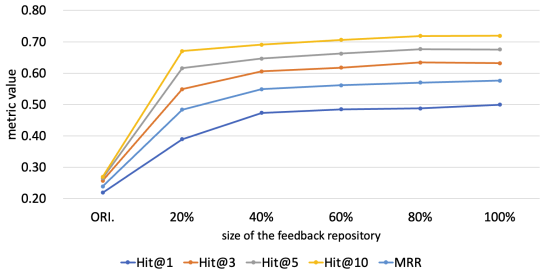
In Experiment 1, we add feedback for each test item to the feedback repository, where the maximum number of feedback is equal to the size of the testing dataset. In Experiment 2, we aim to investigate the impact of the number of feedbacks on code completion performance. Therefore, we limit the maximum number of feedback in the repository and increase

it by 20% for each subsequent experiment. The baseline corresponds to the case where the maximum number of feedback in the repository is 0%, indicating that the feedback repository does not participate in the code completion process. The experimental results are presented in Table II.

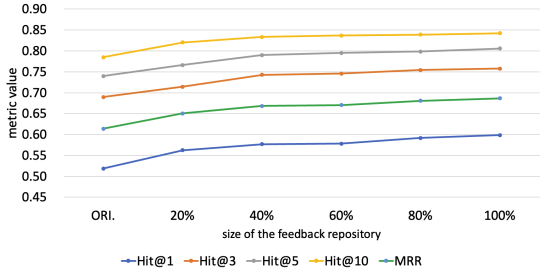
TABLE II: Performance comparison with varying feedback repository sizes

Approach	Metric	Original	20%	40%	60%	80%	100%
EHOPE + N-gram	Hit@1	0.2194	0.3894	0.4738	0.4848	0.4875	**0.4998**
	Hit@3	0.2578	0.5491	0.6059	0.6174	**0.6340**	0.6319
	Hit@5	0.2653	0.6159	0.6468	0.6623	**0.6766**	0.6756
	Hit@10	0.2702	0.6704	0.6907	0.7059	0.7187	**0.7191**
	MRR	0.2394	0.4837	0.5489	0.5617	0.5698	**0.5764**
EHOPE + CodeGRU	Hit@1	0.5187	0.5623	0.5770	0.5783	0.5918	**0.5986**
	Hit@3	0.6896	0.7143	0.7428	0.7457	0.7543	**0.7576**
	Hit@5	0.7397	0.7662	0.7900	0.7952	0.7985	**0.8056**
	Hit@10	0.7847	0.8204	0.8335	0.8369	0.8387	**0.8425**
	MRR	0.6139	0.6504	0.6683	0.6707	0.6808	**0.6867**

To provide a clearer illustration of the trend, we have plotted the results in Figure. 5. It can be observed that there is a consistent improvement in performance as the size of the feedback repository accumulates. This trend is observed in both of the baselines, which suggests the robustness and versatility of our approach for token-level code completion. Notably, all the metrics show significant enhancement, with N-gram achieving a MRR increase of 33%, and CodeGRU achieving a MRR increase of over 7%.



(a) The performance of N-gram.



(b) The performance of CodeGRU.

Figure 5: Learning curves of EHOPE with feedback information for baselines.

Arguably, the most significant improvement in our approach is observed in the Hit@1 metric, which indicates that our method can accurately rank the correct next token to the top-1 position by leveraging the feedback information. As illustrated in Figure. 6, the Hit@1 metric exhibits a substantial

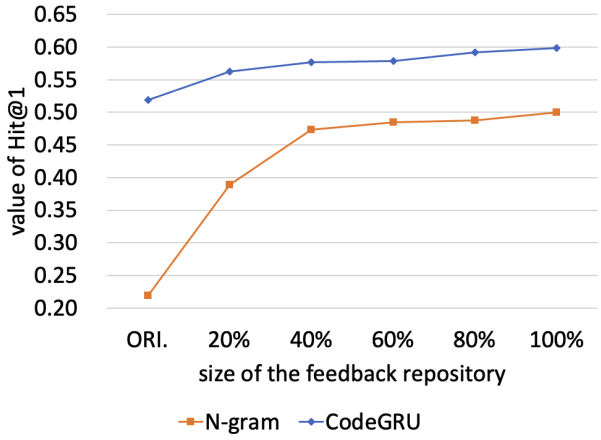


Figure 6: The performance metrics of Baselines Hit@1.

increase in both baselines, with an improvement of 28.04% for N-gram and 7.99% for CodeGRU.

RQ3. How do LSTM and BERT contribute to EHOPE respectively?

Recall that our approach makes use of two models, i.e., LSTM and BERT. To clarify the extent to which each model contributes to the performance enhancement of EHOPE, we conducted an ablation analysis on our approach. This experiment involved disabling either the LSTM or BERT model and recording the corresponding performance metrics. We then compared the results of the baselines with those of LSTM and BERT separately. The detailed experimental findings are presented in Table III.

TABLE III: Evaluation results for our framework comparing with baselines ('Rel. Imp.' stands for 'relative improvement')

Approach	Technique	Hit@1	Hit@3	Hit@5	Hit@10	MRR
EHOPE + N-gram	Original	0.2194	0.2578	0.2653	0.2702	0.2394
	LSTM	0.4928	0.6240	0.6681	0.7145	0.5702
	BERT	0.2227	0.2584	0.2653	0.2702	0.2412
	EHOPE	**0.4998**	**0.6319**	**0.6756**	**0.7191**	**0.5764**
	Rel. Imp. LSTM	124.61%	142.05%	151.83%	164.43%	138.18%
	Rel. Imp. BERT	1.50%	0.23%	0%	0%	0.75%
	Rel. Imp. EHOPE	127.80%	145.11%	154.77%	166.14%	140.77%
EHOPE + CodeGRU	Original	0.5187	0.6896	0.7397	0.7847	0.6139
	LSTM	0.5858	0.7512	0.7985	0.8360	0.6765
	BERT	0.5645	0.7090	0.7507	0.7870	0.6448
	EHOPE	**0.5986**	**0.7576**	**0.8056**	**0.8425**	**0.6867**
	Rel. Imp. LSTM	12.94%	8.93%	7.95%	6.54%	10.20%
	Rel. Imp. BERT	8.83%	2.81%	1.49%	0.30%	5.03%
	Rel. Imp. EHOPE	15.40%	9.86%	8.91%	8.64%	11.86%

The experimental results presented in the table reveal the impact of LSTM and BERT models on enhancing code completion performance. The two models make distinct contributions to both baselines, with a consistent improvement trend observed for each. When comparing the performance of BERT alone to the N-gram baseline, we observe a relatively modest improvement. This can be attributed to the inherent limitations of the initial recommendation list generated by the N-gram model. Since we do not incorporate LSTM to augment the recommendations, the enhancement achieved by solely using

BERT is constrained. Besides, we also find it interesting that the models focus more on improving metrics such as Hit@1, Hit@3, and MRR, rather than Hit@5 and Hit@10. Among these, Hit@1 has the most significant impact. Although LSTM and BERT optimize the performance in different ways, our findings indicate that neither of them outperforms the combined approach, which validates the methodology used by EHOPE.

5. Limitations and Threats to Validity

5.1 Limitations

One of the limitations of this work is the reliability of users' feedback. The choices and feedback provided by users play a crucial role in model training and improvement. However, user feedback can be influenced by individual preferences, user errors, or other factors, introducing a certain level of noise. A solution to this problem is to employ appropriate techniques for data filtering and noise reduction to identify and eliminate potential erroneous or outlier feedback. Another limitation of this work is the lack of data in the feedback repository during the initial stages, which hinders its effective participation in the training of LSTM and BERT models. In our future work, we consider using techniques such as active learning to overcome this limitation and to provide a more robust approach.

5.2 Construct Validity

While our experiments provide comprehensive details and demonstrate the efficacy of our approach, it should be noted that altering the neural network settings for training or evaluating on different test sets may produce varying outcomes. Additionally, the choice of evaluation metrics poses a potential challenge to ensure construct validity. Although the widely adopted Top-k metric [29], [30], [31] is commonly employed for assessing deep learning-based source code models, we further evaluate our proposed approach using the MRR metric [31], [32] to reinforce its effectiveness and mitigate this potential concern.

5.3 Internal Validity

Internal validity threats refer to potential experimental errors and biases [33]. In this study, the main threats arise from possible biases in the data. To address this concern, we use the same data published in the replication packages of the original work to ensure a fair comparison with the baselines. Furthermore, to minimize the possibility of errors during re-implementation, we directly employ the tools provided by the original authors.

5.4 External Validity

Threats to external validity refer to the ability to generalize the results of the experiments to other scenarios beyond the scope of the study [33]. As with any empirical investigation, it is difficult to ensure that our framework will perform effectively when applied to third-party completion approaches. However, we are confident that the two widely adopted tools chosen to showcase the benefits of our approach are indicative, and our extensive experiments effectively illustrate the performance improvements achieved.

5.5 Conclusion Validity

In addressing conclusion validity, our study maintains reliability through consistent experimental conditions, the use of multiple evaluation metrics, and transparent methodology reporting. These practices ensure that our findings concerning the effectiveness of the EHOPE framework in code completion are well-supported and generalizable, enhancing the confidence in our conclusions.

6. Related Work

6.1 Code Completion Based on Statistic

Code completion is a topic of great interest in software engineering research. Early approaches to code completion relied on heuristic rules and static-type information to provide suggestions [34]. And researchers [30] have discovered that source code exhibits certain statistical properties that can be effectively modeled using statistical language models [35], [36], [37], [38]. The N-gram model, in particular, has been widely used for this purpose. However, the traditional N-gram model fails to capture the unique property of localness in the source code [39]. To overcome this limitation, researchers have proposed various modifications to the N-gram model, such as adding a cache mechanism [39] or considering unlimited vocabulary, nested scope, locality, and dynamism [35]. These improved models have been shown to achieve better performance than traditional N-gram based approaches.

6.2 Code Completion Based on Deep Learning

In recent years, deep learning techniques started to be employed in code completion research [40]. White et al. [29] conduct experiments and find that a relatively straightforward RNN model can achieve better results than n-gram models in specific software engineering tasks, including code suggestion. Svyatkovskiy et al. [41] develop an LSTM-based code completion system for recommending Python method calls, which is integrated into the Intellicode extension for Visual Studio Code. Hussain et al. [42] propose a deep semantic net (DeepSN) to leverage the semantic information in the source code for improving code suggestion. DeepSN employs an enhanced hierarchical convolutional neural network with code-embedding to extract top-notch features and learn useful semantic information, and utilizes long short-term memory to capture long and short-term context dependencies in the source code. Li et al. [1] introduce a pointer mixture network to tackle the problem of out-of-vocabulary (OOV) tokens in code completion. Karampatsis et al. [4] present a novel open-vocabulary neural language model for source code that uses the BPE algorithm, beam search, and cache mechanism to predict OOV tokens and reduce vocabulary size. The results of their experiments show that this model outperforms both N-gram models and closed vocabulary neural language models, achieving state-of-the-art performance in token-level

code completion. Liu et al. [3] propose a code completion model based on a vanilla LSTM network, while Kim et al. [6] present a transformer model that leverages syntactic structure to enhance performance. Hussain et al. [43] present a novel transformer-based self-supervised learning technique, Transformer Gated Highway, which surpasses comparable recurrent and transformer models and boosts the modeling performance for the source code suggestion task such as code completion.

6.3 Ranking Recommendation Results

In addition to various code completion approaches, several initiatives have addressed the ranking of recommendation candidates. Many of these initiatives utilize machine learning techniques. For instance, Niu et al. [44] employ the LTR (Learning to Rank) technique to provide code example recommendations based on user queries. They utilize a pairwise LTR algorithm to train a ranking schema that can be effectively utilized for future queries. This approach enables the generation of accurate and relevant code examples for users' needs. In contrast to our approach, they do not incorporate users' feedback in their recommendation process. Zhou et al. [14] also applied LTR techniques. Additionally, they employed active learning techniques to establish a new API recommendation model. They utilized users' selection history as feedback information to enhance the performance of query-based API recommendation systems. Though the work leverages the feedback information as ours, it addresses the API recommendation problem. Wang et al. [45] introduce an active code search approach that integrates feedback into the code search process. Their approach extends and incorporates the refinement technique from the Portfolio [46] tool. The initial step involves retrieving search results from the Portfolio tool for a given query. Subsequently, feedback from users is collected for each fragment in the result list, leading to the expansion of the query representation. The list of results is then re-ranked based on the similarity score between the original query and the expanded queries. While the research utilizes feedback information similar to ours, it focuses on solving the code fragment search problem and notably does not employ the LTR technique. In the field of API recommendation, Thung et al. [47] present WebAPIRec, an automated approach that addresses web API recommendation as a personalized ranking task. This approach utilizes historical data on API usage to transform the recommendation process. By training a model, WebAPIRec aims to minimize ordering errors in the suggested ranking of Web APIs. However, feedback information is also neglected in their approach. Liu et al. [48] propose RecRank, a discriminative approach that focuses on improving the top-1 recommendation within the APIREC framework [49]. RecRank utilizes usage path-based features to accurately rank the recommendation list generated by APIREC, thereby enhancing the overall recommendation quality. On the contrary, our approach remains agnostic to any specific component recommendation method. Furthermore, RecRank also lacks consideration for feedback information.

7. Conclusion

In this paper, we introduce EHOPE, a novel framework for improving the performance of code completion systems. EHOPE takes a code segment and an existing code token recommendation as input, and builds a new code completion model by leveraging the user's selection history as feedback information and utilizing LSTM and BERT. As more feedback information is incorporated, EHOPE outperforms baseline code completers and achieves increasingly better code completion results. Our experiments demonstrate that EHOPE significantly enhances the effectiveness of state-of-the-art code completers. In future work, we plan to develop a full-fledged tool based on EHOPE as a plugin for mainstream IDEs, with the aim of providing better support for programming. Moreover, the approach presented in this paper has broader applicability and we plan to extend it to other recommendation scenarios in software engineering.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No.61972197, No. 62150410441) and the Natural Science Foundation of Jiangsu Province(No. BK20201292).

References

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

[2] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, "Codegru: Context-aware deep learning with gated recurrent unit for source code modeling," *Information and Software Technology*, vol. 125, p. 106309, 2020.

[3] C. Liu, X. Wang, R. Shin, J. E. Gonzalez, and D. Song, "Neural code completion," 2016.

[4] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1073–1085.

[5] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.

[6] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 150–162.

[7] C. B. Clement, S. Lu, X. Liu, M. Tufano, D. Drain, N. Duan, N. Sundaresan, and A. Svyatkovskiy, "Long-range modeling of source code files with ewash: Extended window access by syntax hierarchy," *arXiv preprint arXiv:2109.08780*, 2021.

[8] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, pp. 181–212, 2010.

[9] P. Resnick and H. R. Varian, "Recommender systems," *Communications of the ACM*, vol. 40, no. 3, pp. 56–58, 1997.

[10] G. Salton and C. Buckley, "Improving retrieval performance by relevance feedback," *Journal of the American society for information science*, vol. 41, no. 4, pp. 288–297, 1990.

[11] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *Acm Computing Surveys (CSUR)*, vol. 44, no. 1, pp. 1–50, 2012.

[12] D. W. Oard, J. Kim *et al.*, "Implicit feedback for recommender systems," in *Proceedings of the AAAI workshop on recommender systems*, vol. 83. Madison, WI, 1998, pp. 81–83.

[13] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *2008 Eighth IEEE international conference on data mining*. Ieee, 2008, pp. 263–272.

[14] Y. Zhou, X. Yang, T. Chen, Z. Huang, X. Ma, and H. Gall, "Boosting api recommendation with implicit feedback," *IEEE Transactions on Software Engineering*, vol. 48, no. 6, pp. 2157–2172, 2021.

[15] Y. Zhou, H. Jin, X. Yang, T. Chen, K. Narasimhan, and H. C. Gall, "Braid: an api recommender supporting implicit user feedback," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1510–1514.

[16] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[17] R. Nogueira and K. Cho, "Passage re-ranking with bert," *arXiv preprint arXiv:1901.04085*, 2019.

[18] C. E. Shannon, "A mathematical theory of communication," *The Bell system technical journal*, vol. 27, no. 3, pp. 379–423, 1948.

[19] A. Graves, N. Jaitly, and A.-r. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," in *2013 IEEE workshop on automatic speech recognition and understanding*. IEEE, 2013, pp. 273–278.

[20] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks," in *International conference on machine learning*. PMLR, 2014, pp. 1764–1772.

[21] Z. Huang, W. Xu, and K. Yu, "Bidirectional lstm-crf models for sequence tagging," *arXiv preprint arXiv:1508.01991*, 2015.

[22] S. Ghosh, O. Vinyals, B. Strope, S. Roy, T. Dean, and L. Heck, "Contextual lstm (clstm) models for large scale nlp tasks," *arXiv preprint arXiv:1602.06291*, 2016.

[23] Y. Zhang and J. Yang, "Chinese ner using lattice lstm," *arXiv preprint arXiv:1805.02023*, 2018.

[24] K. Roy, A. Ishmam, and K. A. Taher, "Demand forecasting in smart grid using long short-term memory," in *2021 International Conference on Automation, Control and Mechatronics for Industry 4.0 (ACMI)*. IEEE, 2021, pp. 1–5.

[25] T. Zhang, F. Wu, A. Katiyar, K. Q. Weinberger, and Y. Artzi, "Revisiting few-sample bert fine-tuning," *arXiv preprint arXiv:2006.05987*, 2020.

[26] C. Sun, X. Qiu, Y. Xu, and X. Huang, "How to fine-tune bert for text classification?" in *Chinese Computational Linguistics: 18th China National Conference, CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18*. Springer, 2019, pp. 194–206.

[27] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 689–699.

[28] R. F. Silva, C. K. Roy, M. M. Rahman, K. A. Schneider, K. Paixao, and M. de Almeida Maia, "Recommending comprehensive solutions for programming tasks by mining crowd knowledge," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 358–368.

[29] 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.

[30] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[31] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 323–334.

[32] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 311–322.

[33] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research-an initial survey." in *Seke*, 2010, pp. 374–379.

[34] D. Hou and D. M. Pletcher, "Towards a better code completion system by api grouping, filtering, and popularity-based ranking," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, 2010, pp. 26–30.

[35] 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.

[36] F. Liu, L. Zhang, and Z. Jin, "Modeling programs hierarchically with stack-augmented lstm," *Journal of Systems and Software*, vol. 164, p. 110547, 2020.

[37] 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.

[38] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," *Advances in neural information processing systems*, vol. 32, 2019.

[39] 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.

[40] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[41] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: Ai-assisted code completion system," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 2727–2735.

[42] Y. Hussain, Z. Huang, and Y. Zhou, "Improving source code suggestion with code embedding and enhanced convolutional long short-term memory," *IET Software*, vol. 15, no. 3, pp. 199–213, 2021.

[43] Y. Hussain, Z. Huang, Y. Zhou, and S. Wang, "Boosting source code suggestion with self-supervised transformer gated highway," *Journal of Systems and Software*, vol. 196, p. 111553, 2023.

[44] H. Niu, I. Keivanloo, and Y. Zou, "Learning to rank code examples for code search engines," *Empirical Software Engineering*, vol. 22, pp. 259–291, 2017.

[45] S. Wang, D. Lo, and L. Jiang, "Active code search: incorporating user feedback to improve code search relevance," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 677–682.

[46] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[47] F. Thung, R. J. Oentaryo, D. Lo, and Y. Tian, "Webapirec: Recommending web apis to software projects via personalized ranking," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 145–156, 2017.

[48] X. Liu, L. Huang, and V. Ng, "Effective api recommendation without historical software repositories," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 282–292.

[49] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "Api code recommendation using statistical learning from fine-grained changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 511–522.