Improving Code Refinement for Code Review Via Input Reconstruction and Ensemble Learning

Jiawei Lu
Zhejiang University
Hang Zhou, China
jiaweilu@zju.edu.cn

Zhijie Tang
Zhejiang University
Hang Zhou, China
tangzhijie@zju.edu.cn

Zhongxin Liu[†]
Zhejiang University
Hang Zhou, China
liu_zx@zju.edu.cn

2023 30th Asia-Pacific Software Engineering Conference (APSEC) | 979-8-3503-4417-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/APSEC60848.2023.00026

***Abstract*—Code review is crucial for ensuring the quality of source code in software development. Automating the code review process is essential to save time and reduce costs, as manually reviewing code can be time-consuming and challenging for developers. Code refinement, an important task for automating code review, aims to automatically modify the code under review to address reviewers' comments. Previous research has fine-tuned pre-trained models like CodeT5 and CodeReviewer for code refinement, showing promising results. However, fine-tuning these models can make them forget the knowledge learned during pre-training and lead to suboptimal performance. To overcome this challenge, we employ an information retrieval method to enable the model to recall its learned knowledge. Furthermore, we propose using prompt templates to reconstruct the input and align the formats of the input data used during fine-tuning and pre-training, thus alleviating knowledge forgetting. Multiple models are created using the retrieval reconstruction and prompt reconstruction methods mentioned above, which are highly complementary. An ensemble learning method is employed to identify the most promising output from the outputs of these models. Our ensemble model achieves an Exact Match (EM) score of 36.32, surpassing the state-of-the-art CodeReviewer model by 19.3% and the popular GPT-3.5-Turbo model by 49.6%.**

***Index Terms*—code refinement, pre-trained models, retrieval augmented, prompt augmented, ensemble learning**

I. Introduction

Code review is a necessary process in software development that can improve the code quality [1]–[4]. One of the main goals of code review is to ensure that the quality of newly developed code aligns with the prescribed code style guidelines before it is integrated into the main software repository [5]. Code style guidelines require that the developed code is error-free, well-engineered, and consistent with the company's coding conventions.

During the code review process, reviewers (developers other than the code writer) must check the submitted source code written by developers and propose modifications, and developers need to continually modify the code until the reviewer agrees that the source code quality is qualified and can be integrated into the main codebase.

Code reviews offer significant benefits but come at a cost. Reviewers invest a substantial amount of time in comprehending the code and providing instructive comments that highlight issues in the submitted code and guide developers in making necessary modifications [6]–[9]. After review, developers must spend time implementing code changes based on the comments received. This process is usually referred to as code refinement and may require multiple cycles. Code refinement is one of the most important tasks during code review. Specifically, it takes as input the original code written by the developer and the review comment, and targets to generate the revised code that addresses the review comment. Automating code refinement can assist developers in understanding how to make the modifications requested by the reviewer, thus significantly reducing development costs.

Many researchers have leveraged deep learning methods for code refinement [10]–[13]. Tufano et al. [10] used a Transformer-based [14] model with two encoders to respectively encode the code under review and the received comment, and generate the revised code using a decoder. Wang et al. [15] pre-trained an encoder-decoder model named CodeT5 for programming languages, which also achieves good performance on code refinement. Li et al. [16] developed a pre-trained model named CodeReviewer. CodeReviewer is specifically designed for code review tasks, which has been applied to code refinement and achieved state-of-the-art (SOTA) performance on this task.

Although existing models show promising results on the code refinement task, there is still some space for improvement. Firstly, prior work has shown that information retrieval techniques can augment generation models [17], [18]. Specifically, given a sample, by retrieving the training sample similar to it and feeding the similar sample into the generation model, we can help the generation model recall the knowledge learned during training and boost the generation performance.

Secondly, existing models directly fine-tune pre-trained models on the code refinement task, while the pre-training tasks of these models are usually heterogeneous from code refinement. For example, although CodeT5 includes pre-training tasks that generate code snippets from natural language descriptions, code refinement requires generating a revised code snippet based on the code snippet before revising and a review comment. The different formats of input data during pre-training and fine-tuning may cause catastrophic forgetting, i.e., making the models forget the knowledge learned during pre-training, and thus may degrade the performance of pre-

† Zhongxin Liu is the corresponding author

trained models. Therefore, it is possible to further improve existing approaches by reconstructing the input to align the input formats of fine-tuning and pre-training.

Thirdly, based on the methods described earlier, we can fine-tune multiple code refinement models. We observe that these models complement each other, i.e., different models can refine different sets of samples. Therefore, it is possible to fine-tune and combine multiple models to further improve performance on the code refinement task.

Based on the first insight, we propose an input reconstruction method for code refinement, which first uses an information retrieval technique named BM25 [19] to retrieve the most similar training sample for each sample and then combines the similar sample with the input of the target sample as new input. Based on the second insight, we propose another input reconstruction method, which uses several prompt templates to reconstruct the input of the code refinement task to align the input data formats of pre-training and fine-tuning. Based on the third insight, we fine-tune three models based on CodeT5 for the code refinement task using the two input reconstruction methods and three prompt templates, respectively. We refer to them as the retrieval-augmented model, Prompt-Augmented Model A, and Prompt-Augmented Model B. Furthermore, we combine the three models with a voting method and construct an ensemble model named ENREFINER to boost code refinement.

We evaluate ENREFINER on the dataset provided by CodeReviewer and compare it with the state-of-the-art models including CodeReviewer and GPT-3.5-Turbo. Our model improves CodeReviewer and GPT-3.5-Turbo in terms of Exact Match (EM) by 19.3% and 49.6%, respectively. Our code and experimental results are published at GitHub [1].

To summarize, the contributions of our work are:

- We propose to reconstruct the input of code refinement by combining it with its most similar training sample, which can enable the pre-trained model to recall how modifications were made during fine-tuning.
- We propose to leverage prompt templates to reconstruct the input of code refinement to bridge the gap between the pre-training tasks and the code refinement task.
- We propose a novel approach named ENREFINER, which leverages a voting method to ensemble the models fine-tuned with three input reconstruct methods. ENREFINER outperforms the state-of-the-art model CodeReviewer and the popular GPT-3.5-Turbo model in terms of Exact Match by 19.3% and 49.6%, respectively.
- We conduct comprehensive experiments to demonstrate the effectiveness of our proposed approach and the two input reconstruction methods.

II. RELATED WORKS

A. Transformers-Based Models for Automated Code Review

Transformer [14] is a neural network model based on the attention mechanism with an encoder-decoder architecture. It was originally used for the machine translation task in natural language processing and has been widely applied to various natural language processing tasks, achieving great success. Tufano et al. [10] were the first to use the Transformer architecture for automated code review. This work investigates two tasks, i.e., generating the revised code based only on the submitted code and generating the revised code based on both the submitted code and the review comment, i.e., code refinement. Thongtanunam et al. [20] incorporated the BPE algorithm from natural language processing [21], which effectively reduced the vocabulary size of the Transformer and improved its performance. However, the performance of these models is not satisfactory enough due to the limited number of model parameters.

B. Pre-Trained Models for Automated Code Review

Inspired by the success of large pre-trained models in natural language processing [22]–[25], Feng et al. [26] transferred pre-trained a language model named CodeBERT using large-scale code corpus. Guo et al. [27] pre-trained GraphCodeBERT, which incorporates code structural information into the model and achieves better performance. CodeBERT and GraphCodeBERT have achieved good performance in the field of programming language tasks. Ahmad et al. [28] imitated the model structure of BART [29] and pre-trained their PLBART model, which is also used for bug fixing. Meanwhile, Wang et al. [15] proposed a pre-trained code model named CodeT5 based on the mode structure of T5 [30] and several pre-training tasks customized for code. CodeT5 achieved excellent performance in various downstream tasks, including code generation. These pre-trained models can be used to solve various downstream tasks, including code refinement.

Li et al. [16] pre-trained a model called CodeReviewer, which is dedicated to code review tasks. CodeReviewer initializes its model using the parameters of CodeT5. Its pre-training tasks include “Diff Tag Prediction”, “Denoising Code Diff”, “Denoising Review Comments”, and “Review Comment Generation”. These pre-training tasks are all designed around code review activities. This model achieves state-of-the-art performance on multiple tasks related to code review, i.e., code quality estimation, code refinement, and review comment generation. Different from the models mentioned above, our approach improves code refinement by fine-tuning multiple models with different input reconstruction methods and leverages a voting method to ensemble these fine-tuned models for further improvement.

C. Prompt Techniques for Large Pre-trained Models

During the pre-training phase, large pre-trained models acquire a significant amount of knowledge. However, the input formats of downstream tasks often differ from those of the pre-training tasks. A novel approach known as prompt-tuning focuses on aligning downstream task data with the pre-training task, rather than modifying the parameters of the pre-trained model [31]–[34].

[1]https://github.com/moonmengmeng/EnRefiner

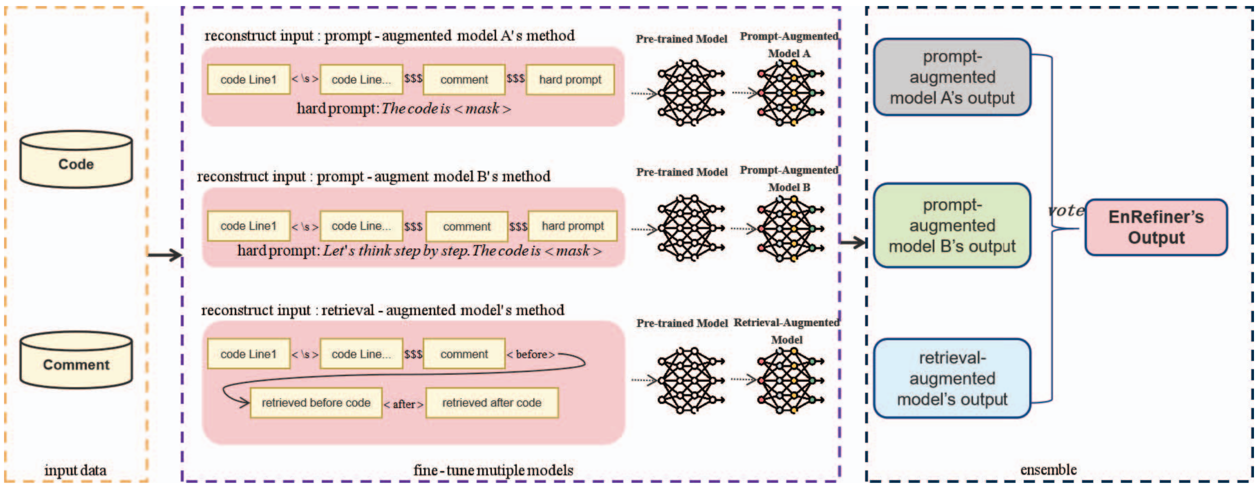
reconstruct input : prompt - augmented model A's method
code Line1
<\s>
code Line...
$$$
comment
$$$
hard prompt
hard prompt: The code is <mask>
reconstruct input : prompt - augment model B's method
hard prompt: Let's think step by step. The code is <mask>
reconstruct input : retrieval - augmented model's method
<before>
retrieved before code
<after>
retrieved after code
Pre-trained Model
Prompt-Augmented Model A
Prompt-Augmented Model B
Retrieval-Augmented Model
Code
Comment
input data
fine - tune mutiple models
prompt-augmented model A's output
prompt-augmented model B's output
retrieval-augmented model's output
vote
EnRefiner's Output
ensemble

Fig. 1. Overview of the workflow of ENREFINER.

Gao et al. [31] first proposed to use prompt-tuning for large pre-trained models, which changed the traditional pretraining-finetuning mode. Prompt techniques attempt to construct the downstream task data to as closely match the pretraining process as possible, thereby reducing the difficulty of fine-tuning [35], [36]. Prompt techniques can be categorized into two types, i.e., hard prompts and soft prompts. Hard prompts add fixed natural language instructions into the input [37], while soft prompts add tokens which are continuous vectors and can be learned during the tuning stage [38]. Wang et al. [39] migrated prompt technology to software engineering and did a thorough evaluation of defect detection, code summarization, and code translation tasks. In this work, we modified some of the prompt templates they proposed and applied them to code refinement.

III. ENREFINER

A. Overview

ENREFINER is a model that integrates multiple pre-trained models. It reconstructs the input of the pre-trained models through two approaches: retrieval-augmented and prompt-augmented. Due to the need for ensemble voting, the prompt-augmented method in our approach consists of two prompt templates. So we build three data sets (1 retrieval-augmented and 2 prompt-augmented) and fine-tune the pre-trained models, it generates three sub-models. After obtaining the outputs of the sub-models, it performs a voting-based ensemble and selects the output of the overall model.

Figure 1 presents an overview of the workflow of our ENREFINER. Firstly, we developed several reconstruction methods, i.e., retrieval-augmented method and prompt augmented method that transform the input into a specific format and build a data set. Then we used these data sets to fine-tune multiple pre-trained models. Finally, we collect the outputs of the individual sub-models and use an integrated voting-based learning method to produce the final ENREFINER output. The following section will provide detailed descriptions of our approach.

B. Pre-Trained Model

A pre-trained model is a machine learning model that has been trained on a large dataset by experts and is made available for use by others without requiring them to train the model from scratch. It has learned patterns, features, and relationships from extensive data during the training phase, and it can be fine-tuned for specific tasks or domains. Pre-trained models have accelerated the development of AI systems by providing a starting point and saving time and resources for researchers and developers.

T5 [30] is an encoder-decoder pre-trained model that treats every text processing problem as a “text-to-text” problem. CodeT5 [15] is a dual-modal pre-trained model which was trained in both programming languages and natural languages, so it understands both program language and natural language well. Its architecture is the same as T5 [30], using an encoder-decoder framework. Compared to the traditional T5 model, CodeT5’s main innovation is that it has developed various pre-training tasks related to identifiers based on the characteristics of code data, such as predicting whether a code token is an identifier, masking identifier prediction, and so on. This makes CodeT5 have strong code comprehension ability while retaining T5’s understanding of natural language. The state-of-the-art (SOTA) model CodeReviewer utilizes CodeT5’s parameter initialization for their model, so we also adopt the same approach. In this paper, we initialize our single models, i.e., the prompt-augmented models and the retrieval-augmented model, based on the parameter of CodeT5-Base.

C. Train Multiple Models

This section corresponds to the “fine-tune multiple models” part of Figure 1. In this section, we will describe how to create inputs for each pre-trained model and fine-tune them.

TABLE I
AN EXAMPLE OF THE RETRIEVAL METHOD.

Reviewer's Comment: Instead of a 'getattr' call here, can you just set 'task_namespace = None' on the base class? thanks.

Input Code:

```
...
- if getattr(cls, "task_namespace", None) is None:
...
```

Retrieved Buggy Code:

```
- if msg.get('Date',None) is None:
```

Retrieved Fixed Code:

```
+ if msg.Date is None:
```

The Output of Retrieved-Augmented Model:

```
...
+ if cls.task_namespace is None:
...
```

Ground Truth:

```
...
+ if cls.task_namespace is None:
...
```

The retrieval-augmented model retrieval code from the training set helps the model recall its knowledge during the pre-training stage and the prompt-augmented models use the hard prompt technology to effectively compensate for the gap between model pre-training and fine-tuning, improving model performance. Moreover, multiple models fine-tuned with various inputs provide diverse outputs that complement each other. By using subsequent ensemble learning methods, model performance can be improved.

*1) **Train Retrieval-Augmented Model**:* Inspired by Wei et al. [17] and Zhang et al. [18], retrieving the sample similar to the input code in the training set and appending it to the input can help the generation model recall the knowledge learned during training and boost the generation performance. We use the method of BM25 [19], a ranking function used by search engines to estimate the relevance of documents to a given search query. In the BM25 algorithm, we have a query and a batch of documents Docs, and we need to calculate the relevance score between the query and each document Doc. BM25 can find a code in the training set that is most similar to the target code by setting the target code as a query and the training set as Docs. We calculate the BM25 scores for the current buggy code versus each buggy code in the training set and select the one with the highest BM25 score. We append this buggy code and its refined version to the input. We also use the same method to retrieve similar code from the training set at the time of validating and testing. We use the constructed data set and fine-tune the pre-trained model. We name it as the retrieval-augmented model. This model with the retrieval method also helps us to find more correct results. Table I shows our retrieval-augmented method helps the pre-trained model for code refinement. A more detailed code of this sample is shown in table III.

We have created a retrieval-augmented model and its input reconstruction method as follows:

- Retrieval-Augmented Model: *[code lines linked by <\s>] $$$ [review comment] $$$ <before> [retrieved buggy code] <after> [retrieved fixed code]*

*2) **Train Prompt-Augmented Models**:* The different formats of input data used in the pre-training and fine-tuning phases can lead to the forgetting of the knowledge learned during the pre-training stage, resulting in a decline in the performance of pre-trained models. So, we can streamline the pre-training and fine-tuning process by aligning the input formats, thereby reducing the difficulty of model fine-tuning.

Specifically, we adjust the input data formats using prompt templates and use such formats during fine-tuning, validation, and testing. But how to design the prompt templates. Although no prior work has specifically investigated the best prompt templates for code review tasks, Wang et al. [39] empirically studied the effectiveness of different prompt templates in code intelligence. Based on their work, we choose the best-performing prompt template "[x] The code is [z]" to reconstruct the input. [x] and [z] refer to the input and output of the target task.

In addition to this prompt template, recent studies [37], [40] in the field of natural language processing have shown that the Chain-of-Thought (CoT) prompting [22] technique can substantially improve the reasoning ability of large language models and lead to better performance. Code refinement aims to refine a given code snippet based on the review comment, which usually involves reasoning. Therefore, it may also benefit from the CoT technique. Based on this observation, besides the simple prompt template studied by Wang et al. [39], we also use a CoT prompt template "[x] Let's think step by step. The code is [z]" to elicit reasoning in the pre-trained model.

We reconstruct the input using each prompt template and feed the reconstructed input into pre-trained models for fine-tuning, validation and, testing. The fine-tuned models are referred to as prompt-augmented models. Using the prompt templates studied by Wang et al. [39] and the CoT prompt template, we obtain two prompt-augmented models and refer to them as Prompt-Augmented Model A and Prompt-Augmented Model B, respectively. Their input data formats are as follows:

- Prompt-Augmented Model A: *[code lines concatenated by <\s>] $$$ [review comment] $$$ The code is <mask>*
- Prompt-Augmented Model B: *[code lines concatenated by <\s>] $$$ [review comment] $$$ Let's think step by step. The code is <mask>*

D. Ensemble Learning for Collecting Models' outputs and Voting

This section corresponds to the "ensemble" in Figure 1. By employing the two input reconstruction methods mentioned above, we have fine-tuned three models. We observe that they complement each other. i.e., the cases where the three models produce correct outputs are different. This inspires us that we can improve the performance of the fine-tuned models if we can select and combine their correct outputs. Specifically, each input, each fine-tuned model will produce an output. The outputs of the two models may be the same or different. Each output is considered as one vote, with the number of votes for a new output being 1, and for an output that has appeared before, adding 1 to the original number of votes. We will select the output that gets the most votes (i.e., the most frequent) among the models. If there is a tie, we will use the output of the Prompt-Augmented Model A because it performs best on the validation set. This method is similar to students checking their answers after an exam, where an answer that most students have chosen is more likely to be correct. A common concern is that, if the task is too difficult, most of the models may not be able to produce correct results. If most models point to the same incorrect answer, this indicates that the performance of the pre-trained model has approached its limits and a more powerful model is needed.

An example of voting is shown in Table II. Although the output of Prompt-Augmented Model A is Answer2, the other two models' output is the same as Answer1, so Answer1 is adopted. In this particular example, both Prompt-Augmented Model B and retrieval-augmented model successfully generated the correct results. Prompt-Augmented Model A ends up with incorrect results but other models help correct it. It is worth noting that although the retrieval-augmented model produces the correct answer in this case, it might also retrieve unsuitable training methods. However, in such cases, the other models step in and compensate for their errors, creating a complementary dynamic. To summarize, the ensemble method mitigates the occasional mistakes of one model with other models, resulting in an overall balanced and effective approach.

TABLE II
AN EXAMPLE OF THE VOTING METHOD.

Reviewer's Comment: It might be clearer to keep it false in constructor and set the flag at 'MegaClient::readnodes' call to new Node.

Input Code:
 	
`plink = NULL;`
`memset(&changed, 0, sizeof changed);`
`- changed.newnode = true;`
`Node* p;`

Output Answer 1 (The output of the retrieval-augmented model and the Prompt-Augmented Model B):
 	
`plink = NULL;`
`memset(&changed, 0, sizeof changed);`
`Node* p;`

Output Answer 2 (The output of the prompt-augmented A model):
 	
`plink = NULL;`
`memset(&changed, 0, sizeof changed);`
`+ changed.newnode = false;`
`Node* p;`

The Output of ENREFINER (2 models vote for this):
 	
`plink = NULL;`
`memset(&changed, 0, sizeof changed);`
`Node* p;`

Ground Truth:
 	
`plink = NULL;`
`memset(&changed, 0, sizeof changed);`
`Node* p;`

IV. EXPERIMENTAL DESIGN

To investigate the performance of our ENREFINER model on the code refinement task, we aim to answer the following research questions (RQs):

RQ1: How effective is our approach on the code refinement task? In RQ1, we aim to verify the effectiveness of ENREFINER on the code refinement task. To answer this RQ, we compared ENREFINER with state-of-the-art baselines.

RQ2: How effective are the retrieval-augmented model and prompt-augmented models? In RQ2, we aim to analyze whether the proposed data reconstruction methods can directly improve performance. To answer this RQ, we conducted ablation experiments on the three sub-models in our ensemble model.

RQ3: Does our ensemble model show a performance improvement compared to individual sub-models on the code refinement task? In RQ3, we aim to analyze whether our proposed ensemble model can help find more potentially correct results compared to a single model. To answer this RQ, we analyze the outputs of different single models to investigate whether they are complementary to each other.

A. Baseline Models

To demonstrate the effectiveness of ENREFINER, we compared it to the state-of-the-art models CodeT5 and CodeReviewer as well as the currently most popular GPT-3.5-Turbo model. ENREFINER fine-tunes its single models based on CodeT5. By comparing ENREFINER and CodeT5, we can see whether our approach can improve code refinement. GPT-3.5-turbo is the most popular large language model today, and CodeReviewer is the SOTA model on the code refinement task now. CodeReviewer also uses CodeT5 to initialize parameters, so we can visualize the advantages of our model when we compare ENREFINER with CodeReviewer.

CodeT5 [15] is a pre-trained model based on the Encoder-Decoder architecture that has been introduced in Section III-B.

TABLE III
AN EXAMPLE OF A PERFECT PREDICTION GENERATED BY ENREFINER.

Reviewer's Comment: Instead of a 'getattr' call here, can you just set 'task_namespace = None' on the base class? thanks.

Input Code:

```
cls = super(Register, metacls).__new__(metacls, classname, bases, classdict)
- if getattr(cls, "task_namespace", None) is None:
cls.task_namespace = metacls._default_namespace
metacls._reg.append(cls)
```

CodeReviewer:

```
cls = super(Register, metacls).__new__(metacls, classname, bases, classdict)
cls.task_namespace = None
metacls._reg.append(cls)
```

ENREFINER:

```
cls = super(Register, metacls).__new__(metacls, classname, bases, classdict)
+ if cls.task_namespace is None:
    cls.task_namespace = metacls._default_namespace
    metacls._reg.append(cls)
```

Ground Truth:

```
cls = super(Register, metacls).__new__(metacls, classname, bases, classdict)
+ if cls.task_namespace is None:
    cls.task_namespace = metacls._default_namespace
    metacls._reg.append(cls)
```

We excerpt the performance of CodeT5 from the CodeReviewer paper.

CodeReviewer [16] is a pre-trained model specifically designed for code review tasks. Like CodeT5, it uses the Encoder-Decoder architecture and initializes model parameters with CodeT5's parameters. CodeReviewer includes three new pre-training tasks, i.e., "Diff Tag Prediction", "Denoising Objective", and "Review Comment Generation". CodeReviewer has achieved state-of-the-art performance on multiple downstream tasks related to code review, including the code refinement task. We excerpt the performance of CodeReviewer from the CodeReviewer paper.

GPT-3.5-Turbo is a large pre-trained language model developed by OpenAI, based on the GPT architecture. Since the API for GPT-4 [41] has not been made available to the public, we are unable to use GPT-4 to perform experiments for code refinement. Therefore, we are using GPT-3.5-Turbo instead. As a language model, it can generate text and code based on the input prompt received from users. We use the OpenAI API "gpt-3.5-turbo-0301". Our input prompt format is as follows:

"Please fix the following code according to the attached comment, output only the code, and try to keep the format of the code before the change. code: [the code to be refined] comment: [the review comment attached to the code]"

Specifically, since GPT-3.5-Turbo did not go through fine-tuning, it directly outputs the modified code after receiving relevant prompt inputs. GPT-3.5-Turbo did not receive code style and formatting information about the CodeReviewer's dataset. To ensure fairness, when evaluating the output of GPT-3.5-Turbo, we tokenized and recombined the output of GPT-3.5-Turbo output, removed comments, blank lines, and useless spaces, and then compared Exact Match.

B. Dataset

We use the code refinement dataset provided by CodeReviewer. In their paper, Li et al. utilize the GitHub REST API to gather pull request data from various projects. This method of data collection is widely used by many previous researchers [42], [43]. The GitHub API provides an HTTP request-based interface for accessing information related to repositories. By sending HTTP requests to GitHub, they can easily retrieve data in JSON format for various aspects such as branches, commits, pull requests, code diff, review comments, and more. This approach enables a convenient and efficient way of collecting and analyzing data from GitHub repositories. The dataset exists in the form of code diffs, where a code diff that needs to be modified receives a comment and is modified according to the review comment. The dataset includes 150,406 training set samples, 13,103 validation set samples, and 13,104 test set samples. To ensure fairness, we followed the original partitioning of the dataset used by CodeReviewer. The dataset is stored in the form of "JSONLINES" and contains multiple "JSON" dictionaries, the corresponding keys for the data that the model needs to use in the dictionaries are 'old', 'comment', and 'new' which denote the buggy code before the fix, the comment provided by the reviewer, and the code after the fix, respectively.

C. Evaluation Metrics

When the model outputs exactly the same code as the ground truth, it is considered to satisfy the exact match, and the Exact Match (EM) score is the number of outputs satisfying the exact match divided by the total number of samples. We use Exact Match as the evaluation metric, and it has been widely used previously in code refinement tasks [10], [16], [20], [44]. This is because the traditional metric BLEU [45] used in natural language processing generation tasks only evaluates the similarity between the predicted and target outputs, while a small change in the source code can lead to compilation and execution errors. Moreover, excessive error messages can reduce developers' confidence in automatic code review models and may make them unwilling to use these models. The Exact Match score is a more suitable metric for this task. Only when a prediction matches the target exactly it will be considered correct. Therefore, we use the same evaluation metric, i.e., Exact Match as the CodeReviewer's paper [16] used in the code refinement task when we compared the results of ENREFINER with other models.

D. Implementation Details

We implement our model with PyTorch [46]. In this paper, we build our single models, i.e., the prompt-augmented models and the retrieval-augmented model, based on CodeT5-Base. In terms of hyperparameter settings, we used the grid to search on the validation set to determine the most suitable hyperparameters for the dataset and finally set the length of CodeT5's input and output to 320 and 256 for prompt-augmented models, respectively. Due to the inclusion of retrieved code, the length of the input has significantly increased. Therefore, we increased the input length of the retrieval-augmented model to 512 so that the model could get our constructed information. We set the batch size to 8 and the learning rate to 5e-5. We use the AdamW optimizer [47] and train for enough epochs until the Exact Match score does not exceed that of the previously saved best model for five consecutive epochs. At that point, we early stopped the training and used the saved best model as the trained model. Additionally, we conducted all experiments on a computer equipped with 8 GeForce RTX 3090 GPUs with 24 GB memory.

V. RESULT ANALYSIS

In this section, we answer each research question based on our experiment results. Specifically, we first compare our proposed approach ENREFINER with baselines on the code refinement task in RQ1. Then, we conduct ablation studies to illustrate the effectiveness of every single model in ENREFINER. In RQ3, we analyze the superiority of the ensemble model over each sub-model.

A. Result Analysis for RQ1

We compared ENREFINER with all baseline models. Table IV shows ENREFINER outperforms all baseline models on the code refinement task. Out of 13,014 test samples, ENREFINER automatically fixed 4,760 buggy codes, while the old state-of-the-art model CodeReviewer only fixed 3,973 buggy codes and GPT-3.5-Turbo only fixed 3,182 buggy codes. In RQ2 and RQ3, we will explain the reasons for the performance improvement of individual models and the performance improvement achieved through model ensembling, respectively.

TABLE IV
COMPARISON WITH BASELINE MODELS ON THE CODE REFINEMENT TASK.

Model	EM
GPT-3.5-Turbo	24.28
CodeT5†	24.41
CodeReviewer†	30.32
ENREFINER	**36.32**

† CodeT5 and CodeReviewer's performance is from CodeReviewer's paper.

Table III shows an example of a refined code generated by our model. CodeReviewer did not understand the reviewer's comment "set 'task_namespace' = None on the base class", while ENREFINER correctly completed the task. The reviewer proposed to set the attribute "task_namespace" to None directly **on the base class** instead of using "getattr". By doing so, the reviewer suggests that the desired behavior can be implemented without the need for an additional function call. Because "task_namespace" is set to None on the base class, in the "__init__" method of the derived class, the value of "task_namespace" can be inherited directly from the class attribute "cls.task_namespace". So in this case, in the derived class, it is only necessary to check whether "task_namespace" is None. Codereviewer did not understand the meaning of the reviewer's comment and simply copied "task_namespace = None" from the comment, while ENREFINER completed the task perfectly.

In conclusion, our model ENREFINER achieves an EM score of 36.32, and it has shown performance improvements over all baseline models on the code refinement task, including the state-of-the-art model CodeReviewer.

TABLE V
ABLATION STUDY OF DIFFERENT SUB-MODELS.

Model	EM
CodeT5†	24.41
CodeReviewer†	30.32
Prompt-Augmented Model A	**35.27**
Prompt-Augmented Model B	35.01
Retrieval-Augmented Model	34.58

† CodeT5 and CodeReviewer's performance is from CodeReviewer's paper.

B. Result Analysis for RQ2

Table V shows the Retrieval-Augmented Model attained an EM score of 34.58. The retrieved code changes indeed offer additional information in certain cases, aiding the model in making accurate refinement. As shown in Table I, the retrieved Buggy Code is very similar to the input Code, and the modification method of its corresponding Fixed Code is also very similar to the actual modification method made by the model. In this example, the retrieval-augmented method greatly assists the model in recalling the knowledge learned during training and improves the generation performance. However, they can also be misleading in certain situations, resulting in overall improvement that is not particularly high compared to prompt-augmented models. Nevertheless, the multiple answers generated by this approach enable a boost in the overall model performance through voting.

Furthermore, according to the experiments conducted by Wang et al. [39] and others, the prompt used by the Prompt-Augmented Model A is highly beneficial for code generation. This prompt template and the reconstruction method we designed can streamline the pre-training and fine-tuning process by aligning the input formats, thereby reducing the difficulty of model fine-tuning. Prompt-Augmented Model A is the most effective of all the sub-models and achieved an EM score of 35.27 in Table V. Prompt-Augmented Model B achieved an EM score of 35.01. Although the prompt template of Prompt-Augmented Model B is not as effective as the code-specific prompt template proposed by Wang et al. [39], it complements the Prompt-Augmented Model A by providing an alternative perspective. This allows it to rectify errors made by the Prompt-Augmented Model A through voting.

In summary, all three sub-models improve the EM score to some extent compared to CodeReviewer according to table V. Using reconstruction methods to construct the three single sub-models has helped EnRefiner achieve better performance. For every single model, the reconstruction methods are simple, effective, and do not require additional costs. Also, creating other sub-models can be used for voting purposes and further improve the EM score.

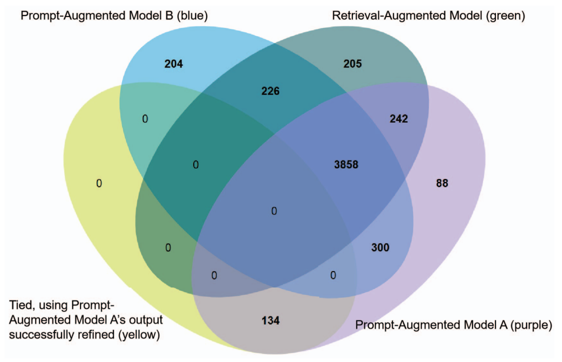

Fig. 2. The numbers of correct outputs generated by different models. Overlapping numbers denote the numbers of samples where all overlapping models produced the correct output.

TABLE VI
Ablation study of ensemble method.

Model	EM
Prompt-Augmented Model A	35.27
Prompt-Augmented Model B	35.01
Retrieval-Augmented Model	34.58
EnRefiner	**36.32**

C. *Result Analysis for RQ3*

As depicted in Figure 2 and Table VI, out of the 13,014 test samples, Prompt-Augmented Model A successfully refined 4,622 test samples, while Prompt-Augmented Model B and Retrieval-Augmented Model fixed 4,588 and 4,531 test samples respectively. Each model can produce correct outputs in various scenarios, and by combining these accurate outputs, we can obtain a more precise answer. According to our voting rules, when two or more models correctly refine a test sample, the overall EnRefiner model produces the correct answer. In cases where the three models generate different outputs, resulting in a tie, the answer from the Prompt-Augmented Model A is selected. Thus, through the voting process, EnRefiner managed to refine a total of 4,760 test samples among them 3,858 from consensus, 226 from Prompt-Augmented Model B and Retrieval-Augmented Model, 242 from Prompt-Augmented Model A and Prompt-Augmented Model B, 300 from Prompt-Augmented Model A and Retrieval-Augmented Model, and 134 from tied result in adopting Prompt-Augmented Model A.

Compared to relying solely on the Prompt-Augmented Model A, the voting approach rectified 226 samples where the model's output was erroneous. Additionally, in 88 samples where only the Prompt-Augmented Model A produced the correct answer while the other two models provided the same incorrect answer, EnRefiner outputted incorrect results.

The success of this ensemble method is attributed to leveraging the strengths of three different models. The Retrieval-Augmented Model retrieves exemplars to provide hints and references for the model, easing the difficulty of generation. Prompt-Augmented Model A utilizes prompts to activate the knowledge acquired during the pre-training phase. Meanwhile, Prompt-Augmented Model B inspires the model to gradually arrive at the correct answers through step-by-step reasoning.

Overall, the diverse and complementary nature of different sub-models enabled EnRefiner to refine an additional 138 test samples compared to the best sub-model, showcasing the benefits of integrating multiple models.

VI. Discussion

A. *Limitations*

One of our limitations is that automatic code refinement cannot achieve complete correctness, so full automation is not possible. However, our goal is not to replace developers during code review activities but to work with them by providing refinement suggestions and assisting them in completing modifications. Fully automated reviews may introduce bugs into the main program, and achieving a 100% correct automation transformation may not be realistic. Furthermore, bypassing developers could negatively impact code maintainability. So our automated code review is designed to assist developers and work together with them.

Another limitation of EnRefiner is the high computational overhead since it uses multiple sub-models. However, the training time of each pre-trained model on a GeForce RTX 3090 GPU is 36 hours, which is acceptable since we just need to train the three models once. After fine-tuning, EnRefiner takes less than 0.3 seconds on average with 3 GeForce RTX 3090 GPUs to produce the refined version of a buggy code. Although the time cost of EnRefiner is three times that of CodeReviewer, we argue that it is worthwhile because the inference time cost is still acceptable for practical use and EnRefiner outperforms CodeReviewer by substantial margins.

B. Threats to validity

In this section, we will analyze the potential threats to the validity of our empirical study.

Threats to internal validity. We only used one form of prompt for GPT-3.5-Turbo, and other prompt formats may yield different outputs. To alleviate this threat, we experimented with multiple prompt templates on a small amount of data and used the best one in our paper.

Threats to external validity. External validity mainly focuses on the selection of the dataset. The dataset is built only from Java projects, which may not represent all languages. However, Java is one of the most popular programming languages. Moreover, the pre-trained model has been trained on datasets from multiple programming languages, demonstrating its ability to generalize across various programming languages.

Threats to construct validity. Construct validity mainly concerns the rationality of the evaluation metric. Like other research in code review tasks, we believe that the widely used BLEU score [45] in text generation tasks is not suitable for evaluating code review generation tasks. As small code errors can lead to program crashes, even programs with high BLEU scores can be completely wrong. EM can reflect the model's ability to completely fix the buggy code. Therefore, we used EM as the evaluation metric.

VII. Conclusion and Future Work

In this paper, we focus on the code refinement task, which automatically modifies the code under review to address reviewers' comments. We propose a novel approach EnRefiner which uses two reconstruction methods and three different input formats to fine-tune the pre-trained model and builds three sub-models, i.e., 'retrieval-augmented model', 'Prompt-Augmented Model A' and 'Prompt-Augmented Model B'. Among them, the data reconstruction method used in the prompt-augmented models helps to bridge the gap between the pre-training task and downstream tasks. The data reconstruction method used in the retrieval-augmented model helps recall pre-trained knowledge. Finally, we integrate submodels through a voting-based ensemble learning approach. Experimental results show that our method outperforms state-of-the-art models.

In the future, it is hoped that our model can be extended, which is that we can employ a classification model to learn how to select the output of sub-models. Although we have made some attempts and found that using CodeBERT for classification is not ideal, there may be better classifiers that can perform well and achieve the intended goal.

References

[1] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 192–201.

[2] R. Morales, S. McIntosh, and F. Khomh, "Do code review practices impact design quality? a case study of the qt, vtk, and itk projects," in *2015 IEEE 22nd international conference on software analysis, evolution, and reengineering (SANER)*. IEEE, 2015, pp. 171–180.

[3] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: a case study at google," in *Proceedings of the 40th international conference on software engineering: Software engineering in practice*, 2018, pp. 181–190.

[4] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 202–211.

[5] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 81–90.

[6] A. Bosu and J. C. Carver, "Impact of peer code review on peer impression formation: A survey," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 133–142.

[7] X. Yang, R. G. Kula, N. Yoshida, and H. Iida, "Mining the modern code review repositories: A dataset of people, process and product," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 460–463.

[8] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, 2013, pp. 202–212.

[9] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.

[10] R. Tufano, L. Pascarella, M. Tufano, D. Poshyvanyk, and G. Bavota, "Towards automating code review activities," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 163–174.

[11] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.

[12] V. J. Hellendoorn, J. Tsay, M. Mukherjee, and M. Hirzel, "Towards automating code review at scale," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1479–1482.

[13] B. Wu, B. Liang, and X. Zhang, "Turn tree into graph: Automatic code review via simplified ast driven graph convolutional network," *Knowledge-Based Systems*, vol. 252, p. 109450, 2022.

[14] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[15] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.

[16] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu *et al.*, "Automating code review activities by large-scale pre-training," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1035–1047.

[17] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: exemplar-based neural comment generation," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 349–360.

[18] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1385–1397.

[19] S. E. Robertson and S. Walker, "Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval," in *SIGIR'94: Proceedings of the Seventeenth Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval, organised by Dublin City University*. Springer, 1994, pp. 232–241.

[20] P. Thongtanunam, C. Pornprasit, and C. Tantithamthavorn, "Autotransform: automated code transformation to support modern code review process," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 237–248.

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

[22] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[23] 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.

[24] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[25] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

[26] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[27] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[28] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[29] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," *arXiv preprint arXiv:1910.13461*, 2019.

[30] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[31] T. Gao, A. Fisch, and D. Chen, "Making pre-trained language models better few-shot learners," *arXiv preprint arXiv:2012.15723*, 2020.

[32] R. Suvorov, E. Logacheva, A. Mashikhin, A. Remizova, A. Ashukha, A. Silvestrov, N. Kong, H. Goka, K. Park, and V. Lempitsky, "Resolution-robust large mask inpainting with fourier convolutions," in *Proceedings of the IEEE/CVF winter conference on applications of computer vision*, 2022, pp. 2149–2159.

[33] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," *arXiv preprint arXiv:2109.01652*, 2021.

[34] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, "Learning transferable visual models from natural language supervision," in *International conference on machine learning*. PMLR, 2021, pp. 8748–8763.

[35] M. E. Peters, S. Ruder, and N. A. Smith, "To tune or not to tune? adapting pretrained representations to diverse tasks," *arXiv preprint arXiv:1903.05987*, 2019.

[36] R. Puri and B. Catanzaro, "Zero-shot text classification with generative language models," *arXiv preprint arXiv:1912.10165*, 2019.

[37] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," *arXiv preprint arXiv:2205.11916*, 2022.

[38] B. Lester, R. Al-Rfou, and N. Constant, "The power of scale for parameter-efficient prompt tuning," *arXiv preprint arXiv:2104.08691*, 2021.

[39] C. Wang, Y. Yang, C. Gao, Y. Peng, H. Zhang, and M. R. Lyu, "No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 382–394.

[40] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[41] Y. Anand, Z. Nussbaum, B. Duderstadt, B. Schmidt, and A. Mulyar, "Gpt4all: Training an assistant-style chatbot with large scale data distillation from gpt-3.5-turbo."

[42] R. Heumüller, S. Nielebock, and F. Ortmeier, "Exploit those code reviews! bigger data for deeper learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1505–1509.

[43] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "Core: Automating review recommendation for code changes," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 284–295.

[44] R. Tufano, S. Masiero, A. Mastropaolo, L. Pascarella, D. Poshyvanyk, and G. Bavota, "Using pre-trained models to boost code review automation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2291–2302.

[45] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[46] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[47] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," 2017.