LLM Based Automatic Code Review and Code Fix Generation

Md. Asif Haider[1], Sk. Sabit Bin Mosaddek[1], Ayesha Binte Mostofa[1], Toufique Ahmed[2], Anindya Iqbal[1]
[1]Department of CSE, Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh
[2]University of California, Davis (UC Davis), California, USA
Email: elhanasif8@gmail.com, 1805106@ugrad.cse.buet.ac.bd, 1805062@ugrad.cse.buet.ac.bd, tfahmed@ucdavis.edu, anindya_iqbal@yahoo.com

Background

Code review, the manual process of inspecting source code by fellow teammates, is recognized as a crucial part of the software development lifecycle. Despite **helping detect errors and encouraging further code improvement**, code review activities take a toll on the **developers' valuable time and efforts.** Yang et al [1] reported millions of code reviews take place in a typical software project involving thousands of reviewers annually, occupying nearly half the project cost and duration. Hence, it is of significant demand to automate both code review and fix generation tasks.

With the rapid advances in deep learning techniques, researchers proposed many pretrained models [2] focusing on source code. Although novel fine-tuning attempts on large-scale datasets showed promising results, accurate code review and fix generation remain **challenging due to their inherent diverse and non-unique nature**. Training on huge datasets requires costly computing resources imposing a negative impact on carbon footprint globally.

Fortunately, large language models can **reduce the need for repetitive training while offering amazing few-shot learning capabilities** [3], which refers to prompt engineering of the model with a few **similar query-response pairs**. Designing efficient prompts for the mentioned tasks yet remains less explored, motivating us toward this research direction.

Motivation and Problem Formulation

The objectives for the research are **fourfold**. First, we aim to **investigate the effectiveness of LLM-based few-shot learning strategy** in software engineering task automation. We want to work on two of the most popular but less explored software engineering tasks: **generating review comments**, and then **fixing the inefficient, low-quality code** based on those reviews automatically.

Secondly, we wish to explore a few **high-performing large language model APIs** offered by OpenAI, the market-leading international research organization producing generative models with billions of learning parameters. Our objective here is to **efficiently utilize the API-provided limited prompt space**, eventually designing a cost-effective query strategy.

Most importantly, we plan to experiment with two of our devised prompt engineering techniques as the starter: incorporating **human language summary** and programming language-specific **semantic insights** like **function call graphs** for a given code. The motivation is to strengthen the giant-corpus human-language conversational agent by combining relevant program-level linguistic context with it. BLEU score is widely used to assess the quality of machine-generated text, hence **improving the state-of-the-art BLEU scores** remains our key objective.

Finally, we want to conduct additional **ablation studies** to determine the isolated, stand-alone influence of each of our applied augmentation strategies on the output result

Proposed Methodology

1. Reproducing Baseline

First, we download the publicly available pretrained model checkpoints and labeled train, validation, and test datasets. The challenge in reproducing the baseline results lies in fine-tuning the giant deep learning model '**CodeReviewer' with 223 millions of parameters** by Microsoft Research [4] .

2. Natural Language Summary Data Collection

'CodeT5' [5], an identifier-aware encoder-decoder transformer model achieves the current best performance on **code summarization** tasks. Hence, we employ this model to take **programming language code snippet as input** and get a **natural language summary as output**. We tune the model to work with 512 token inputs and 50 token outputs per batch (4096 tokens roughly represent 3000 English words). CodeT5 model offers similar challenges to that of the CodeReviewer model mentioned above.

3. Function Call Graph Data Collection

Function call graph is a graph theoretical representation of **function flows in a code** and their internal relationships. Call graphs have been widely used to facilitate understanding the **structure, evolution, and execution flow** of software systems. We experiment with Tree-sitter, a popular syntax tree generator tool to generate function call graphs for 9 of the most used programming languages: **C, C++, Go, Python, C#, Java, PHP, JavaScript, and Ruby.**

Experimentation

1. Experimental Models

At this point, we are ready to invoke the OpenAI API with our augmented prompt. We select 5000 entries per task for the initial experiment via random sampling. We plan to play with two of the most promising OpenAI models: **Codex** and **GPT 3.5-Turbo** respectively. While Codex specializes in **source code generation**, GPT 3.5-Turbo is known for its supreme performance in **general-purpose language understanding.**

2. Input-Output Representation

The paid APIs have a limitation on the input-output allowance (4096 tokens at a time), hence **effectively utilizing the prompt space** turns out to be a major challenge. In the case of 3-shot prompting, we have 1024 tokens for each exemplar. We initially plan to allocate 512 tokens for the original code snippet (diff hunk), and 3/5th of the remaining half tokens for the function call graphs, leaving the rest for summaries.

3. Result Analysis

We then evaluate the BLEU scores for our experiment samples, with and without incorporating the **BM-25 information retrieval algorithm** to fetch relevant exemplar shots, keeping the former one as the baseline. Finally, we compare our output results with the already reproduced numbers earlier and analyze the performance improvement. Additionally, we proceed to validate our findings via the **Wilcoxon signed-rank** statistical hypothesis test **for different metrics and paired values. We also wish to conduct an ablation study** to determine the stand-alone influences of the proposed augmentations.

Findings

So far, we have performed experiments with the 5000 sample test set for the code review generation task. The table below presents the BLEU scores computed and the confidence interval at 95%.

The 1st row of the table shows the baseline of the comparison, without any call graph and summary integrated. The table clearly shows that incorporating function call graph improves the BLEU score, albeit by a small margin. Interestingly, code summary does not seem to improve performance, when considered both solo and combined with call graph.

Variant	BLEU Score (%)	Confidence Interval @ 95%
Davinci_BM25_base	4.52	0.128
Davinci_BM25_call	**4.6**	0.126
Davinci_BM25_sum	4.42	0.124
Davinci_BM25_both	4.42	0.127

Table 1: BLEU Scores on Code Review Generation Task

Conclusion and Future Work

In this project, we expect our proposed prompt augmentation techniques to improve the standard BLEU metric on top of the state-of-the-art results for the chosen tasks and datasets. We hope to achieve our anticipated answers to the research questions: **few-shot prompt engineering with relevant natural and programming language-specific knowledge augmentation indeed helps LLMs identify improved context to achieve better cost-effective performance** compared to many pretrained and finetuned models with a huge carbon footprint and compute requirements.

However, the code summary produced from the oldfile portions tend to hamper the performance. One possible reason behind this can be the poorly generated summary from the entirety of the old files. If only the responsible functions can be located and then summarized, the BLEU score might improve. We will also perform our experiment with the code refinement task as the next step. **We expect software industries will hugely benefit from automated code review and refinement activities** and focus on more important goals**.**

References

[1] X. Yang, R. G. Kula, N. Yoshida and H. Iida, "Mining the Modern Code Review Repositories: A Dataset of People, Process and Product," 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX, USA, 2016, pp. 460-463.

[2] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using pre-trained models to boost code review automation. In Proceedings of the 44th International Conference on Software Engineering (ICSE '22). Association for Computing Machinery, New York, NY, USA, 2291–2302. https://doi.org/10.1145/3510003.3510621

[3] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., . . . Amodei, D. (2020). Language Models are Few-Shot Learners. *ArXiv*. /abs/2005.14165

[4] Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., & Sundaresan, N. (2022). Automating Code Review Activities by Large-Scale Pre-training. *ArXiv*. /abs/2203.09095

[5] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021). CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *ArXiv*. /abs/2109.00859

10th NSysS 2023, December 21-23, 2023, Khulna, Bangladesh