Improving Accuracy of LLM-based Code Clone Detection Using Functionally Equivalent Methods

Ryutaro Inoue, Yoshiki Higo
Osaka University, Japan, {ry-inoue, higo}@ist.osaka-u.ac.jp

***Abstract*—A code clone is a code snippet identical or similar to another in the source code. The presence of code clones causes the spread of bugs, which means that efficient code clone detection and appropriate refactoring are necessary. Code clone detection using large language models (in short, LLMs) is more accurate than conventional tools that do not use LLMs for code clones with low syntactic similarity. However, even for LLM-based clone detection tools, detecting such code clones is still difficult, and there is room for improvement. In this study, we improved the accuracy of LLM-based code clone detection through fine-tuning using FEMPDataset. The results showed that our fine-tuning improved the accuracy of code clone detection.**

***Index Terms*—code clone, large language model, fine tuning, functionally equivalent methods**

I. Introduction

A code clone is a code snippet identical or are similar to another in the source code [1]. Changing the text of a code clone requires consistent changes occationally; unintentional inconsistent changes can lead to defects [2]. Thus, the presence of code clones can significantly prevent source code changes and impair the maintainability of software systems. Therefore, detecting clones efficiently and refactoring them appropriately is necessary.

Many code clone detection tools have been developed before now. Well-known tools include CCFinder [3], NiCad [4], Oreo [5], and NIL [6], which use lexical analysis and various metrics for detection. Those tools tend to be highly accurate in detecting clones with high syntactic similarity, while they tend to be less accurate in detecting clones with low syntactic similarity.

In contrast, code clone detection with large language models (in short, LLMs) has demonstrated superior accuracy for code clones with low syntactic similarity compared to conventional tools. LLMs have shown promising results in various fields, including natural language processing, and have gained widespread attention [7]. ChatGPT, which can use models such as GPT [8], has reached 100 million users in just two months since its launch. Additionally, Meta's Llama2 [9] and code-llama [10], a fine-tuned model of Llama2, have also been developed.

Dou et al. used several LLMs for code clone detection, comparing their performance with conventional tools [11]. They found that conventional tools like NiCad and Oreo struggled to detect code clones with low syntactic similarity. On the other hand, LLM-based detection techniques, including GPT-3.5-turbo, GPT-4-turbo, and Llama2, achieved higher detection accuracy for code clones with low syntactic similarity than conventional tools. However, the detection accuracy of code clones with low syntactic similarity by GPT-3.5-turbo and GPT-4 is not sufficiently high. Additionally, Llama2 inaccurately identified many pairs of code snippets that are not code clones as code clones.

This study aims to enhance the detection accuracy of code clones with low syntactic similarity through fine-tuning LLMs. The LLMs targeted in this study are GPT-3.5-turbo, Llama2-Chat-7B, and CodeLlama-7B-Instruct. Fine-tuning of GPT-3.5-turbo has been executed using OpenAI's API, adjusting hyperparameters such as epoch numbers and batch sizes as necessary. LLMs require substantial GPU memory and processing time for fine-tuning. For this reason, fine-tuning of Llama2-Chat-7B and CodeLlama-7B-Instruct has been executed using techniques such as Lora [12] and ZeRO [13] to reduce GPU memory consumption. FEMPDataset [14], a dataset of functionally equivalent but structurally diverse Java methods, has been used for fine-tuning and evaluation. This dataset has been divided into training, validation, and testing blocks to assess improvements in detection accuracy.

Section II outlines the definition of clones, previous research on LLM-based code clone detection, and the experimental dataset. Section III describes the methodology of this study. Section IV presents our experimental results. Section V discusses those findings, and Section VI concludes the study, highlighting future challenges and directions.

II. Preparation

This section describes the definition of code clones, previous research on LLM-based code clone detection, and the dataset we use in this study.

A. Code Clone Classification

A code clone is a code snippet identical or similar to another in the source code [1]. Code clones are created in the source code for various reasons such as reusing existing code or reimplementing similar functions [15]. A pair of code snippets that are identical or similar to each other is called a clone pair.

1) Code Clone: Roy et al. classified clones into four types based on their similarity [16].

Type-1(T1): Identical code snippets except for variations in whitespace, layout, and comments.

Type-2(T2): Syntactically identical code snippets except for variations in identifiers, literals, types, whitespace, layout, and comments.

Type-3(T3): Similar code snippets with further differences such as changed, added, or removed statements, and variations in identifiers, literals, types, whitespace, layout, and comments.

Type-4(T4): Two or more code snippets that perform the same computation but are implemented by different syntactic variants.

B. FEMPDataset

FEMPDataset [14] is a dataset of functionally equivalent methods with different structure[1]. Functionally equivalent method pairs in FEMPDataset are classified as T4 in the above categories. FEMPDataset extracts candidates for functionally equivalent method pairs by using mutual execution of test cases and then collects truly functionally equivalent methods by visually determining whether the candidates have the same functionality.

C. LLMs(large language models)

Large-scale Language Models (LLMs) are language models trained on a large corpus. There has been a significant increase in the models' scale and the data volume for training.

D. Fine-tuning techniques of LLMs

LLMs have many parameters, requiring substantial GPU memory for fine-tuning. This subsection discusses several techniques to reduce GPU memory consumption through fine-tuning.

1) Lora (Low-Rank Adaptation): Lora (Low-Rank Adaptation) [12] is a technique that reduces the number of parameters updated through fine-tuning, thereby enabling fine-tuning with fewer resources. Lora approximates the differences between the parameters before and after fine-tuning with a low-rank matrix, reducing the number of parameters and enabling efficient learning. The size of the low-rank matrix is determined by specifying the hyperparameter r, which reduces the number of parameters as r decreases.

2) ZeRO (Zero Redundancy Optimizer): ZeRO (Zero Redundancy Optimizer) [13] is a technique that minimizes the GPU memory required per GPU for training by leveraging multiple GPUs.

During fine-tuning, it is necessary to maintain and calculate information on the dynamic change of learning rate, gradients, and weights on the GPU. ZeRO allows each GPU to make parameter changes for specific layers of the LLMs and then combines the results to change all the parameters of the LLMs. By assigning each GPU to make parameter changes for specific layers, ZeRO reduces the GPU memory required per GPU compared to conventional methods.

E. Indicators of Detection Accuracy

Recall, Precision, and Accuracy are used as performance metrics for code clone detection and LLMs evaluation. The meaning and calculation formulas of those metrics are as follows.

Recall: Percentage of method pairs determined to be clones out of method pairs that are clones.

$$Recall = \frac{TP}{TP + FN}$$

Precision: Percentage of method pairs detected as clones that are clones.

$$Precision = \frac{TP}{TP + FP}$$

Accuracy: Percentage of method pairs that the detection system has correctly identified.

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

TP (True Positive), FP (False Positive), FN (False Negative), and TN (True Negative) are as follows.

TP: Number of method pairs that are detected as clones out of the methods that are clones.

FP: Number of method pairs that are detected as clones out of the methods that are not clones.

FN: Number of method pairs that are not detected as clones out of the methods that are clones.

TN: Number of method pairs that are not detected as clones out of the methods that are not clones.

F. Previous Research on Code Clone Detection using LLMs

We introduce the study by Dou et al. as previous research on code clone detection by LLMs [11]. Their study compared the performance of existing tools without LLMs and LLMs using a single prompt as input based on BigCloneBench [17].

The results show that existing tools have higher recall for clones with high syntactic similarity, while LLMs have higher recall for clones with low syntactic similarity.

Additionally, GPT-3.5-turbo and GPT-4-turbo show higher detection rates than existing tools in detecting Type-4 clones with low syntactic similarity. However, the detection accuracy is not sufficiently high, and there is room for improvement. Llama2-chat-7B shows high recall but low precision, recognizing almost all method pairs as clone pairs. Therefore, this study aims to improve the accuracy of code clone detection by LLMs through fine-tuning.

III. Experiment

This section describes the experimental methodology.

Target LLMs in the experiment are as follows.

- GPT-3.5-turbo
- Llama2-Chat-7B
- CodeLlama-7B-Instruct

The experiment aims to improve the detection accuracy of clones with low syntactic similarity through fine-tuning. We experimented with the following steps.

STEP1: Fine-tuning

Executing fine-tuning of LLMs using FEMPDataset.

STEP2: Clone Detection

Giving method pairs of test data in FEMPDataset to LLMs before and after fine-tuning, and answers are got as either ”Yes” or ”No.”

[1] The dataset is available at https://github.com/YoshikiHigo/FEMPDataset

STEP3: Performance Evaluation

Aggregating the answers of LLMs before and after fine-tuning and comparing their performance.

A. Fine-tuning

FEMPDataset is used for fine-tuning. The dataset is divided into training, validation, and testing data for fine-tuning. Fine-tuning is executed using training and validation data, and performance evaluation is executed using testing data. The number of method pairs for each data is shown in Table I.

Fine-tuning of GPT-3.5-turbo is executed using OpenAI API. Fine-tuning of Llama2-Chat-7B and CodeLlama-7B-Instruct is executed using Lora and ZeRO to reduce GPU memory consumption. The hyperparameter r of Lora, as explained in Section II-D, is set to 2. Fine-tuning is executed by two GPUs, Quadro RTX 8000 and Tesla V100.

B. Clone detection (LLMs execution)

LLMs before and after fine-tuning are executed with the following steps.

1) **Getting Method Pairs**
   Getting functionally equivalent method pairs and functionally inequivalent ones from FEMPDataset. Those pairs are targets of clone detection to check whether the target LLMs can recognize clone pairs correctly.
2) **Creating Prompts**
   Converting each method pair into prompts asking whether it is a clone pair .
3) **Getting Answers**
   Inputting the prompts to LLMs before and after fine-tuning to get either ”Yes” or ”No” answers.

1) Dataset for Evaluation: LLMs performance is evaluated using the testing data of the divided FEMPDataset.

2) Prompts: The prompts consist of two roles: ”system” and ”user.” In the system role, the answers are instructed as ”Yes” or ”No.” In the user role, the prompts are instructed to determine whether the two methods are a clone pair or not.

C. Performance Evaluation

We aggregate the answers from LLMs and calculate the three values of Recall, Precision, and Accuracy to evaluate the performance. We used 219 method pairs in FEMPDataset as test data for evaluating performance.

TABLE I
NUMBER OF METHOD PAIRS FOR EACH DATA

	Train data[2]	**Validation data**	**Test data**
Clone method pairs	1,081	132	129
Non-clone method pairs	674	88	90
Total	1,755	220	219

[2] We excluded one non-clone method pair from the train data that exceeded 4,096 tokens at the prompt to reduce the GPU resources required.

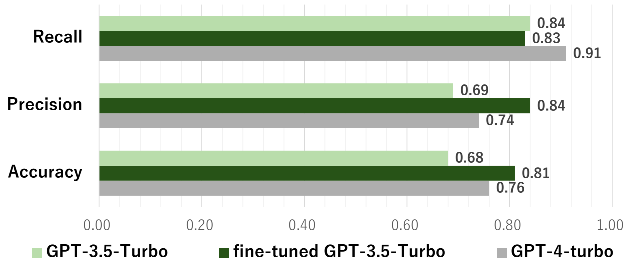

Fig. 1. Comparison of GPT-3.5-turbo and GPT-4-turbo performance

IV. Experimental Results

This section describes the results of the experiment. The comparisons before and after fine-tuning of GPT-3.5-turbo, Llama2-Chat-7B, and CodeLlama-7B-Instruct were executed using the testing data of FEMPDataset.

A. Evaluation of GPT-3.5-turbo

Recall, Precision, and Accuracy of GPT-3.5-turbo before and after fine-tuning and GPT-4-turbo are shown in Table II.

After fine-tuning, GPT-3.5-turbo shows significant improvement in Precision. Recall remains almost the same. This result indicates that fine-tuning has enabled the correct classification of non-clone pairs.

Comparing GPT-4-turbo and fine-tuned GPT-3.5-turbo, the fine-tuned GPT-3.5-turbo has higher Accuracy.

B. Evaluation of Llama2-Chat-7B

Recall, Precision, and Accuracy of Llama2-Chat-7B before and after fine-tuning are shown in Table III. A graphical summary of the above results is shown in Figure2.

Before fine-tuning, Llama2-Chat-7B recognized all functionally equivalent method pairs and functionally inequivalent ones as clone pairs. However, after fine-tuning, Llama2-Chat-7B can correctly classify some functionally inequivalent method pairs. Although Recall has declined, Precision and Accuracy have also improved. This result indicates that the performance has improved compared to Llama2-Chat-7B before fine-tuning.

TABLE II
EVALUATION OF GPT-3.5-TURBO, FINE-TUNED GPT-3.5-TURBO, AND GPT-4-TURBO

model	Recall	Precision	Accuracy
GPT-3.5-turbo	0.84	0.69	0.68
fine-tuned GPT-3.5-turbo	0.83	0.84	0.81
GPT-4-turbo	0.91	0.74	0.76

TABLE III
EVALUATION OF FINE-TUNED LLAMA2-CHAT-7B AND LLAMA2-CHAT-7B

model	Recall	Precision	Accuracy
Llama2-Chat-7B	1.00	0.60	0.60
fine-tuned Llama2-Chat-7B	0.78	0.66	0.63

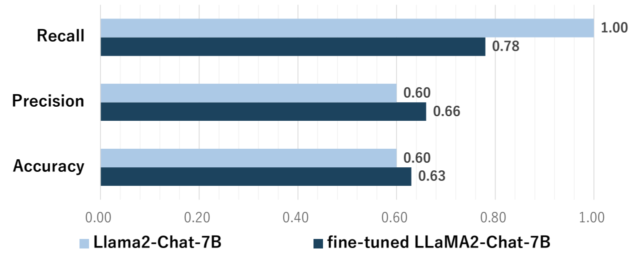

Fig. 2. Comparison of Llama2-Chat-7B performance

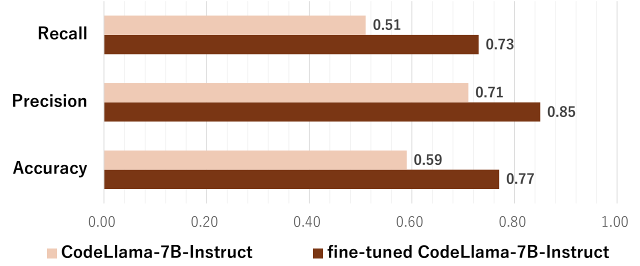

Fig. 3. Comparison of CodeLlama-7B-Instruct performance

C. Evaluation of CodeLlama-7B-Instruct

Recall, Precision, and Accuracy of CodeLlame-Instruct-7B before and after fine-tuning are shown in Table IV. A graphical summary of the above results is shown in Figure 3.

All Recall, Precision, and Accuracy have improved. The performance has improved significantly compared to CodeLlama-7B-Instruct before fine-tuning.

V. Discussion

As a result of fine-tuning GPT-3.5-turbo, Llama2-Chat-7B, and CodeLlama-7B-Instruct, the improvement of code clone detection accuracy was confirmed for all three language models. This result suggests that fine-tuning is effective in code clone detection using LLMs.

The effect of fine-tuning depends on the type of pre-trained data and the performance of the model before fine-tuning. CodeLlama is a model that learned about programs based on Llama2, and Llama2-Chat is a model that learned about dialogue in chat format. CodeLlama-7B-Instruct shows a more significant improvement in performance before and after fine-tuning than Llama2-Chat-7B. This result suggests that the pre-trained data affects the effect of fine-tuning.

VI. Conclusion

In this study, we attempted to improve the accuracy of code clone detection by LLMs through fine-tuning using FEMPDataset. As a result of fine-tuning, all models in the experiment improved in code clone detection accuracy.

TABLE IV
Evaluation of fine-tuned CodeLlama-7B-Instruct and CodeLlama-7B-Instruct

model	Recall	Precision	Accuracy
CodeLlama-7B-Instruct	0.51	0.71	0.59
fine-tuned CodeLlama-7B-Instruct	0.73	0.85	0.77

The following three points are raised as future research directions.

Adding models of the experiment

We believe conducting similar experiments for models not covered in this study (e.g., models focused on source code) will allow for performance comparison between models.

Performance evaluation using other benchmarks

We believe executing similar experiments for other datasets than FEMPDataset would provide a more detailed evaluation of fine-tuning performance.

Improvement of Prompt

In this study, we did not improve the prompts. We believe improving the prompts using technologies such as Chain of thought [18] will improve performance.

Acknowledgements

This research was supported by JSPS KAKENHI Japan (JP21K18302, JP21H04877, JP22H03567, JP22K11985).

References

[1] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. ICSM*, 1998, pp. 368–377.

[2] M. Mondal, C. Roy, and K. Schneider, "A Fine-Grained Analysis on the Inconsistent Changes in Code Clones," in *2020 IEEE ICSME*, 2020, pp. 220–231.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[4] C. Roy and J. Cordy, "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization," in *2008 IEEE ICPC*, 2008, pp. 172–181.

[5] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. Lopes, "Oreo: detection of clones in the twilight zone," in *ESEC/FSE 2018*, 2018, p. 354 – 365.

[6] T. Nakagawa, Y. Higo, and S. Kusumoto, "NIL: large-scale detection of large-variance clones," in *ESEC/FSE 2021*, 2021, p. 830 – 841.

[7] W. X. Zhao et al., "A Survey of Large Language Models," 2023.

[8] OpenAI, "GPT-4 Technical Report," 2023.

[9] H. Touvron et al., "Llama 2: Open Foundation and Fine-Tuned Chat Models," 2023.

[10] B. Rozière et al., "Code Llama: Open Foundation Models for Code," 2023.

[11] S. Dou et al., "Towards Understanding the Capability of Large Language Models on Code Clone Detection: A Survey," 2023.

[12] E. Hu et al., "LoRA: Low-Rank Adaptation of Large Language Models," in *ICLR 2022*, 2022.

[13] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "ZeRO: Memory optimizations Toward Training Trillion Parameter Models," in *SC20*, 2020, pp. 1–16.

[14] Y. Higo, "Dataset of Functionally Equivalent Java Methods and Its Application to Evaluating Clone Detection Tools," *IEICE Trans. Inf. & Syst.*, 02 2024.

[15] C. Roy and J. Cordy, "A Survey on Software Clone Detection Research," *School of Computing TR 2007-541*, pp. 3–7, 01 2007.

[16] C. Roy, J. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[17] J. Svajlenko, J. Islam, I. Keivanloo, C. Roy, and M. Mia, "Towards a Big Data Curated Benchmark of Inter-project Code Clones," *Proc. 30th ICSME*, pp. 476–480, 09 2014.

[18] J. Wei et al., "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," 2022.