DOI: 10.54254/2755-2721/2026.GU32229

A Comprehensive Study of LLM-Based Code Clone Detection

Zhixuan Wu

Huazhong University of Science and Technology, Wuhan, China
3086368748@qq.com

Abstract. Large language models (LLMs) have strong code-understanding skills, so they are well-suited for code clone detection. This is a key code analysis task that shortens development cycles, improves code quality and reduces security vulnerabilities. Recent studies use prompt engineering, parameter fine-tuning and retrieval-augmented generation (RAG) to raise detection accuracy and stability, but systematic surveys on LLM-based clone detection are still not many. To fill this gap, we do a full survey and empirical analysis of LLMs in clone detection, with a focus on Java programs. First, we review existing methods from multiple dimensions, classifying them by methodological techniques and task requirements. To make the study reliable, we collect datasets of typical clone fragments from different perspectives and conduct unified evaluations on ten models of three types: traditional code analysis, specialized pre-trained deep learning models and general-purpose LLMs. We also use various prompting strategies to ensure fair and comparable assessments. We sum up the strengths and limitations of current methods and point out future research directions. Experiments show that for current clone detection tasks, general-purpose LLMs achieve the best overall performance, specialized deep models perform inconsistently in different settings, and traditional analysis methods do worse. However, the effectiveness of LLMs is still affected by task configuration and prompt design, which means there is much room to improve their robustness and consistency.

Keywords: DeepSeek, code clone, code similarity, Java, software development, large language model

1. Introduction

Code clone detection is a basic and key task in software engineering. It aims to find code fragments in a codebase with the same or very similar functions [1]. In the process of software development and maintenance, code clones are widely used. This not only raises the cost of software maintenance, but may also bring possible safety risks and logical mistakes to the software [2]. So, effective and accurate code clone detection methods have great practical value. They can help improve code quality, encourage the reuse of code and make software architecture more optimized.

In recent years, large language models based on the Transformer architecture have made great progress in natural language processing. These models have also shown strong abilities in understanding, creating and analyzing code. Typical models like GPT, DeepSeek and Gemini have achieved excellent results not only in code completion and generation, but also in complex tasks such as understanding the semantic meaning of code and cross-language analysis. This progress has opened up new technical ways for code clone detection. It makes the LLM-based detection methods go beyond the traditional text matching and structural analysis ways, and identify clone code more accurately at the function and semantic levels [3].

But existing studies have only just started to study the application of large language models in code clone detection, and they have put forward some technical methods including prompt engineering, code embedding representations and model fine-tuning. The field still lacks systematic overviews and comprehensive evaluations. Most existing research focuses on a single method or model. There are not enough comparisons between different technical paradigms, and the performance analysis for different clone types and programming language scenarios is also insufficient. Thus, both academic and industrial fields find it hard to fully know the real abilities and limitations of LLMs in code clone detection. This in turn slows down the collection of technical knowledge and the continuous optimization of this research direction.

To solve these problems, this paper takes the Java programming language as a research case, and carries out a systematic overview and practical experimental evaluation on how to apply large language models to code clone detection. First, we review and compare existing studies from two aspects: methodological techniques and task requirements. Second, we build an experimental dataset that includes many types of code clones, and design an evaluation framework with multiple models and different prompting strategies. Last, by comparing quantitative metrics and analyzing errors, we sum up the advantages and limitations of current detection methods. Our test results show that general-purpose large language models perform very well in semantic clone detection, but they still have stability problems in some prompt engineering settings. Specialized pre-trained models can get stable results in structural clone detection, yet their generalization ability is limited in cross-language scenarios. From these findings, we further point out the future research directions. We aim to offer guidance for subsequent studies and promote the deeper application of large language models in the field of code clone detection.

2. Background of large language models

Since the Transformer architecture first came out in 2017, the development of large language models (LLMs) has experienced a number of important technical changes and paradigm shifts. This development process can be roughly divided into three key stages. The first stage built the basic model of pre-training followed by fine-tuning. Typical models of this stage include early Transformer-based ones such as GPT-1 and BERT. These models were pre-trained on large unlabeled corpora with self-supervised learning, and then fine-tuned for downstream tasks. In this way, they laid a good base for the model's scalability and generalization ability. The second stage is marked by a huge increase in model parameter scale and the appearance of general-purpose capabilities. Classic examples are hundred-billion-parameter models like GPT-3 and PaLM. These models showed unrivaled contextual understanding and few-shot learning abilities, and they also had great potential in professional fields such as code generation. At present, LLMs have entered the third stage, which is characterized by multimodal integration, support for ultra-long context and agent-based collaborative systems. The wide use of Mixture-of-Experts (MoE) architectures has greatly improved model capacity and inference efficiency. Meanwhile, the great expansion of context windows (for example, over one million tokens) and the development of unified multimodal architectures such as GPT-4o and Gemini have together made LLMs better at semantic understanding, reasoning and cross-modal analysis of complex code [4]. These technical advances build a solid technical foundation for the in-depth application of LLMs to software engineering tasks like code clone detection.

From the angle of representative models, the LLM field can be roughly divided into international and domestic ecosystems. On the international side, OpenAI's GPT-4 is well known for its excellent performance in code generation and logical reasoning. Anthropic's Claude series has got much attention for its high precision and safety in code-related tasks. Google's Gemini Ultra takes the lead in multimodal code understanding. Meta's LLaMA series, as a core part of the open-source ecosystem, has led to many derivative models that are specialized for code work. In China, models such as Baidu's ERNIE Bot, Alibaba's Qwen, Zhipu AI's GLM and DeepSeek created by DeepSeek AI all show their own unique strengths in code assistance and reasoning, forming a diverse LLM landscape that is developing at a fast pace.

The application of these models has spread deeply into the field of software engineering, and their core capabilities show in many aspects. They support dozens of programming languages including Python and Java, and can carry out complex code analysis that involves multi-file dependencies and algorithmic optimization. They improve the accuracy of code generation by using reasoning methods such as chain-of-thought. Besides, some advanced models can help developers understand and modify large-scale codebases by making use of ultra-long context windows [5]. All these developments show that large language models are changing from simple text generation tools into core productivity parts that raise the efficiency of software development.

3. Comprehensive study

Given the strong capabilities of large language models in code understanding and recognition, research on LLM-based code clone detection has gained increasing momentum in recent years. This section conducts an in-depth exploration and comprehensive analysis of such work from two dimensions: technical methodologies and task requirements.

3.1. Methodological survey

Current LLM-based approaches to code clone detection mainly rely on techniques such as prompt engineering, code embedding representations, and model fine-tuning or refinement. The following subsections analyze these techniques according to their respective methodological categories.

3.1.1. Prompt engineering

Prompt engineering has emerged as a mainstream approach in this field, as it requires little or no training data and enables LLMs to perform code clone identification through carefully designed instructions [6]. In zero-shot prompting, models are directly asked whether two code snippets constitute a clone. However, existing studies have shown that such simple prompts are limited when dealing with semantically complex clones (e.g., Type-4 clones), as models may misclassify cases due to an imprecise understanding of what constitutes a "clone" [7]. But few-shot prompting and chain-of-thought prompting can greatly boost the performance in detecting semantic clones (Type-3/4) — they work by giving examples or guiding the model to do multi-step reasoning, such as analyzing code functions, code structure and similar lines of code. Tests by Dou et al. [8] show that chain-of-thought prompts can effectively raise the accuracy of GPT-3.5/4 in cross-language and complex clone recognition tasks. For cross-language clone detection, earlier studies point out that prompt design should focus on the "functional equivalence" between code snippets, rather than their superficial syntactic similarity. Moumoula et al. [9] found that telling the model clearly to compare code in terms of overall structure and logic can greatly improve detection performance in cross-language scenarios, instead of directly asking the model whether two code snippets are clones.

3.1.2. Code embedding representations

Code embedding representation is another important research direction. The main idea of this method is to encode code fragments into high-dimensional vector representations. Then we can judge the clone relationships between them by calculating vector similarity or using traditional classifiers, which can reduce the instability that may come from the generative outputs of LLMs [10]. General-purpose text embedding models such as OpenAI's text-embedding-ada-002 have shown strong performance in code clone detection tasks. Some studies state that these models even do better than certain specialized code models like CodeBERT, mainly because they can capture semantic similarity in a more effective way [8]. At the same time, other studies focus on specialized code embedding models including GraphCodeBERT and UniXcoder. These models take in structural information such as abstract syntax trees (ASTs) and data flow during the pre-training process, and then they are fine-tuned specifically for clone detection [11]. Besides, there is a practical and effective strategy for this task: we can feed the generated embedding vectors into traditional classifiers like support vector machines (SVMs) or k-nearest neighbors (k-NN) for training and prediction. Related research shows that this method works particularly well in cross-language clone detection, and it may even perform better than direct inference with the use of LLMs [9].

3.1.3. Fine-tuning and refinement

Some research studies use fine-tuning and optimization strategies to boost model performance, and they do this by adjusting pre-trained models to fit downstream clone detection tasks. Carrying out full-parameter fine-tuning on models such as CodeBERT and RoBERTa with datasets like CodeNet can bring good results in monolingual clone detection, yet these models still have limited generalization ability when applied to cross-language detection tasks [7]. To make the models perform better in the detection tasks and preserve their general knowledge at the same time, researchers have developed parameter-efficient fine-tuning methods such as adapter-based fine-tuning. A typical example is AdaCCD: it adopts a contrastive learning strategy to achieve cross-language adaptation, which not only enhances the model's robustness in cross-language clone recognition but also keeps the original knowledge of the pre-trained model intact [12].

3.2. Task-oriented survey

From the angle of task requirements, existing research mainly focuses on the performance in several key aspects: the detection ability in a single language and across different languages, the difficulty of detecting different types of clones, the generalization ability across multiple programming languages, and the detection bias towards code generated by LLMs.

3.2.1. Monolingual and cross-language detection capability

When it comes to monolingual and cross-language detection, the current methods — including fine-tuned models and prompt-based LLM approaches — have become quite mature in dealing with Type-1/2 clones, and they can reach F1 scores of more than 0.9. But it is still a tough task to identify

Type-3/4 clones. Advanced models like GPT-4, though, have shown powerful reasoning abilities in this aspect [8]. In cross-language detection tasks, the performance of traditional fine-tuned models drops obviously. Large language models, especially the ones in the GPT family, get advantages from their cross-lingual understanding and implicit code translation abilities, though. As a result, they work well in zero-shot or few-shot settings, and they can even do better than the fully fine-tuned baseline models [7]. Even so, how well they perform is affected by how similar the syntactic features of different language pairs are [9].

3.2.2. Detection difficulty across different clone types

When it comes to the detection difficulty of different clone types, Type-1/2 (syntactic) clones are relatively easy to identify, and most detection methods can deal with them effectively. The difficulty of detecting Type-3 clones rises as the similarity between code fragments decreases, and using chain-of-thought prompting can improve the recall rate of MT3 and WT3 clones to a certain extent [13]. Type-4 (semantic) clones are the most challenging type to detect at present. Related studies show that even for advanced models like GPT-4, the recall rate for Type-4 clone detection is still low, around 0.15 to 0.3. For this reason, this clone type has become the main focus for further development in prompt engineering and embedding-based detection approaches [14].

3.2.3. Generalization across multiple programming languages

When it comes to the generalization ability of large language models in code clone detection across different programming languages, their performance differs with various languages. They show clear advantages in mainstream languages with plenty of training data, such as Java, Python and C++ [8]. This performance variation is greatly affected by the syntactic similarity between language pairs: language pairs with similar syntax like Java and C are easier to handle, while those with big syntactic differences such as Java and OCaml bring much greater challenges [15]. Earlier research suggests that optimized prompt design can reduce the performance decline caused by language differences to some extent and boost robustness in cross-language detection scenarios [15]. Such design focuses on the code's "functional logic" and "overall structure" instead of just the superficial syntactic forms. It is worth noting that LLMs have strong implicit cross-lingual understanding abilities. For example, in cross-language detection tasks involving language pairs with huge syntactic differences and low-resource languages like Java and Ruby, well-designed prompt engineering allows LLMs such as GPT-3.5 to achieve an F1 score of 0.877, which is significantly better than the performance of traditional fully fine-tuned baseline models [7]. These results show that even with the challenges of uneven language resources and syntactic divergence between languages, LLMs still have great potential to perform stably in cross-language clone detection for low-resource languages. This is all because of their powerful semantic understanding and reasoning abilities.

3.2.4. Detection bias toward LLM-generated code

Studies have also found that there is detection bias when dealing with code generated by LLMs. To be specific, GPT models can get higher accuracy when identifying clones of code made by the same model, and they show a clear detection preference for such code compared with human-written code clones [14]. This finding has very important practical meanings for real application. As LLMs are used more and more widely in programming assistance, future clone detection tools must be able to identify code from different sources in a fair and accurate way, and avoid the unbalanced performance caused by biases in the training data.

4. Evaluation

To fully verify and assess the performance differences between large language models and pre-trained models in code clone detection tasks, this study designs a set of comparative experiments that cover traditional detection methods, specialized pre-trained models and general-purpose large language models.

4.1. Datasets

In this research work, we use the typical Java dataset BigCloneBench to do experiments on code clone detection with the help of naïve large language models. By running related Python scripts, we pick out 50 pairs of code clones and another 50 pairs of non-clone code samples from this dataset in a random way. The selected clone pairs contain Type-1, Type-2 and Type-3 clones, and this makes sure that the experiments can cover different kinds of code clones in multiple categories. After the above selection work is done, these 100 code pairs in total are used to carry out the relevant code clone detection experiments that are based on large language models (LLMs).

4.2. Metrics

In this study, four quantitative metrics, precision, accuracy, recall, and F1-score, are adopted to evaluate the code clone detection performance of each model. Their definitions and corresponding formulas are as follows:

$$\text{Accuracy} = (TP + TN) / (TP + TN + FP + FN)$$

$$\text{Precision} = TP / (TP + FP)$$

$$\text{Recall} = TP / (TP + FN)$$

$$\text{F1-Score} = 2 \times (\text{Precision} \times \text{Recall}) / (\text{Precision} + \text{Recall})$$

Here, TP (True Positive) denotes the number of clone pairs correctly identified as clones, TN (True Negative) denotes the number of non-clone pairs correctly identified as non-clones, FP (False Positive) denotes the number of non-clone pairs incorrectly classified as clones, and FN (False Negative) denotes the number of actual clone pairs that are incorrectly classified as non-clones.

4.3. Experimental setup

In order to make a full comparison of how well different kinds of code clone detection methods work, the study carries out a great deal of experiments and analysis on three typical types of models. These models include traditional detection tools, pre-trained code models based on Transformer, and the most advanced general large language models. When it comes to traditional detection methods, the study picks out several well-cited and typical tools, and they are CCFinder [16], Deckard [17], NiCad [18] and SourcererCC [19]. For the pre-trained code models based on Transformer, the study uses the models designed specially for program analysis, which are CodeBERT [20], GraphCodeBERT [21] and UniXCoder [22]. As for the advanced general large language models, the study chooses DeepSeek-Chat, Qwen-Max [23] and GLM-4 [24], and all of these models are very popular and widely used at present. Here we make a brief summary of the main features of these models one by one:

CCFinder [16] is a method based on tokens and it relies on suffix-tree matching. This tool is quite fit for doing fast detection work in large-scale codebases.

Deckard [17] is a similarity matching method based on tree structure, and it works especially well when people need to detect syntactic clones.

NiCad [18] is a method based on text normalization and similarity computation, and it can support different types of clones at the same time.

SourcererCC [19] is a light method based on index, and it is made for doing efficient clone detection work in nearly real time.

CodeBERT [20] is a BERT-based model trained in advance with a masked language modeling (MLM) goal. It focuses on the aligned representations of code and natural language, and it can be used for tasks like code search and code summarization. But its performance may not be stable in code clone detection, because its pre-training goals may not match with the clone detection task well.

GraphCodeBERT [21] is an extended version of BERT, and it takes the structural information of code into consideration. This feature lets it catch both semantic and structural features more effectively, so it is very suitable for tasks that need deep code representations, such as code understanding and clone detection.

UniXCoder [22] is a unified cross-modal pre-trained model, and it can support the representations of code, text and graph. It has strong abilities in code representation and generation, and it can perform well in different code understanding tasks in a balanced way. So it is a reliable general model for the semantic representation of code.

DeepSeek-Chat is a general large language model, and it can process long-context content and support multi-turn dialogue. It is an open-source model and can be used for commercial purposes, with the ability to support up to 128K tokens. What's more, it shows strong generalization and stability in all kinds of code-related tasks.

Qwen-Max [23] is the largest version of the Qwen (Tongyi Qianwen) series. This general large language model has a strong ability to understand both Chinese and English code, and it can perform stably in cross-language code tasks. That is why it is fit for the code scenarios with mixed Chinese and English content.

GLM-4 [24] is a large generative model based on the General Language Model (GLM) architecture. It has great abilities in following instructions and adapting to multi-tasks, and it can also show stable performance in the tasks of structured code generation and semantic understanding.

4.4. Results

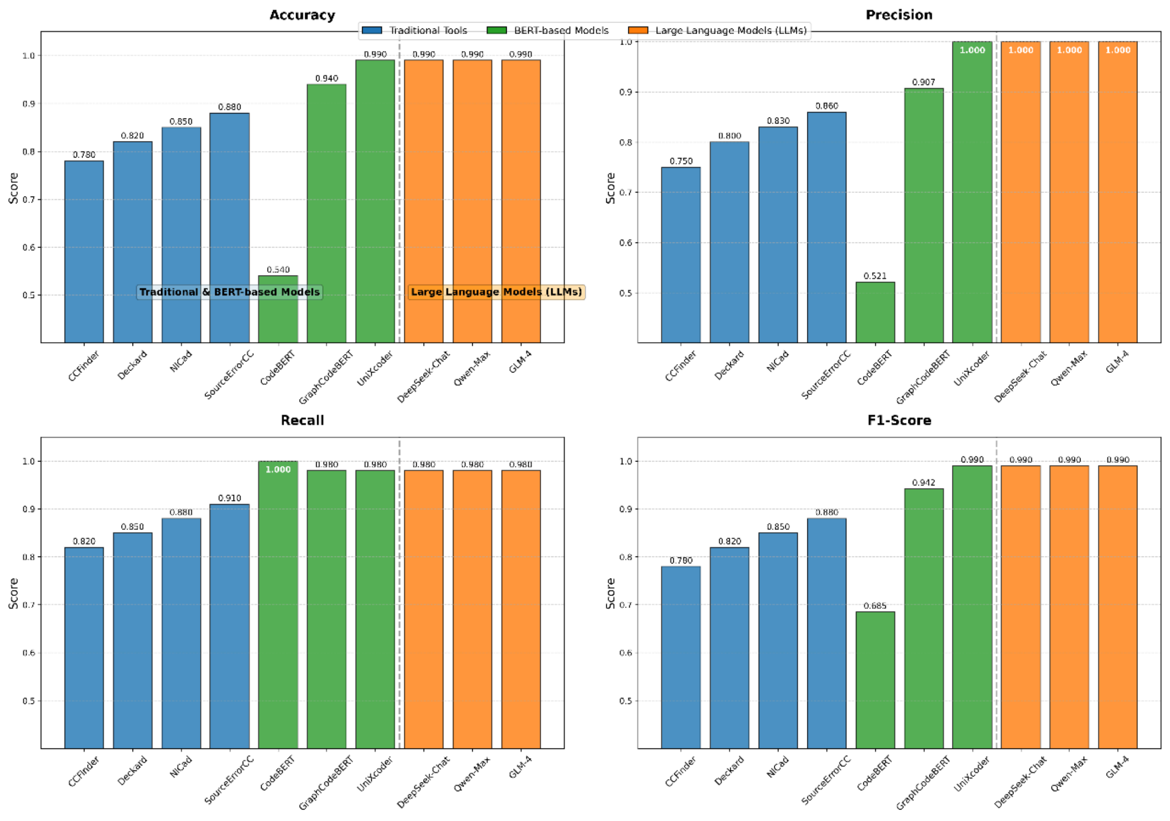

Figure 1. Performance metrics comparison of code clone detection models

The results of how different models work in comparison are shown in Figure 1. As we can see, three large language models do extremely well in code clone detection. They are DeepSeek-Chat, Qwen-Max and GLM-4, and this shows general-purpose LLMs have a clear edge in understanding code meanings. Among the pre-trained models based on BERT, CodeBERT works in a highly unstable way. Its ability to find all target clones is almost perfect, but its ability to find correct clones is really very low. This kind of situation may be caused by a certain fact [25], and the fact is that CodeBERT's pre-training goal fits better with code text modeling than clone discrimination. This makes the model have a trend to over-predict which pairs are clone pairs. But on the other hand, GraphCodeBERT and UniXCoder get quite good results in code clone detection. Especially, UniXCoder works almost as well as large-scale general-purpose LLMs. It has far fewer parameters than those LLMs, and this points out its great ability in understanding code and showing code's semantic meanings. And in comparison, traditional detection methods get bad results in both of the two above-mentioned aspects. The main reason for this is that they lack the ability to do deep semantic understanding of code, and they rely too much on the simple syntactic or structural similarity on the code's surface.

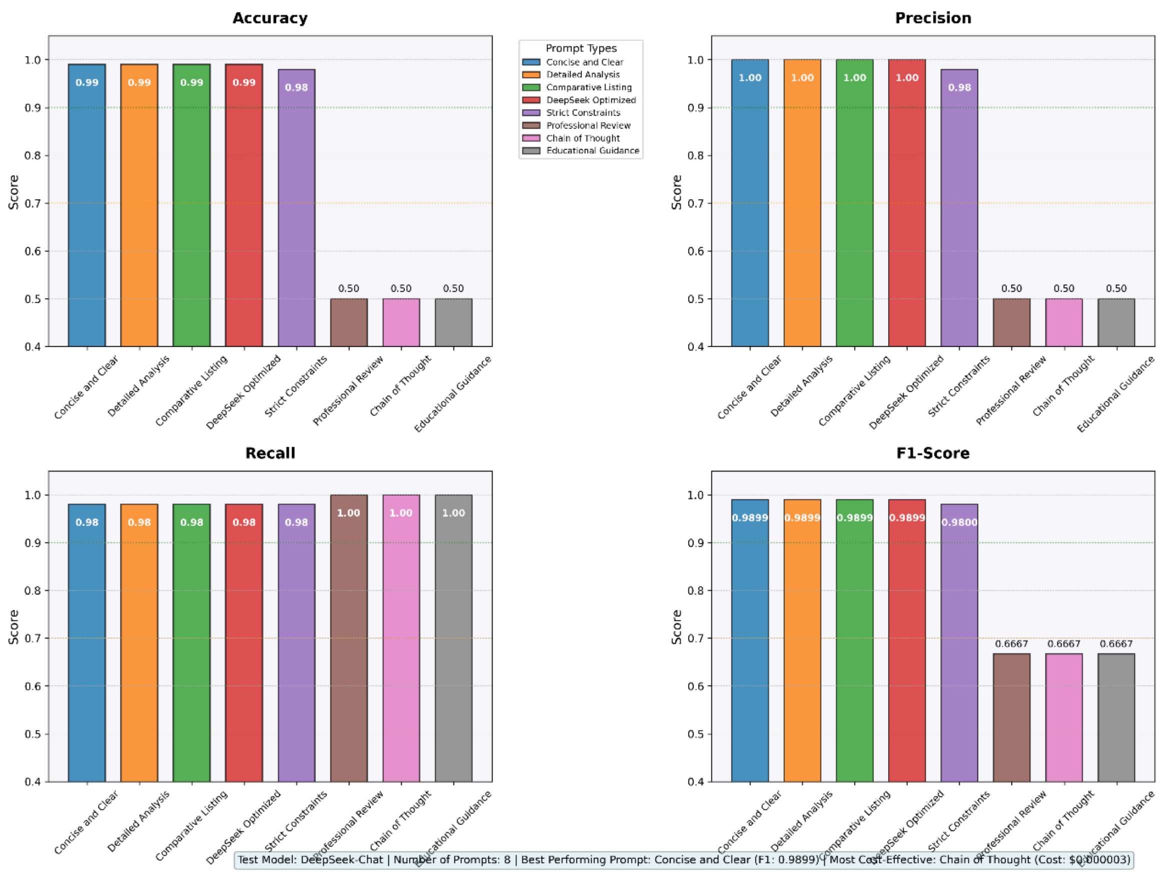

Figure 2. DeepSeek-chat multi-prompt performance metrics comparison

The comparison of detection results under different prompt design methods is shown in Figure 2. This figure explains how different prompt designs influence the clone detection results of DeepSeek-Chat. Specifically, simple and clear prompts can bring about better overall results in the whole clone detection process. But on the other hand, prompts based on step-by-step thinking analysis or teaching-related guidance lead to lower precision in clone detection that is based on LLMs. This may be because the more complex meaning requirements in such prompts disturb the model's focus on the semantic meanings of code. And this condition will effectively widen the range of clone identification in practice. So, even though the accuracy and precision go down in a clear way, the recall of the model remains at a fairly high level all the same.

4.5. Discussion

From the analysis of all the related experiments, the current ways to detect code clones with LLMs are both practical and really effective in use. When we look at different kinds of models, general large language models do well in detecting semantic code clones, but the special pre-trained models are more stable in detecting syntactic code clones. In detail, CodeBERT is a common model for showing code features, and it does not work well in clone detection tasks. This is because it mainly learns code just like normal text, and it is not naturally fit for telling different code clones apart. Also, even the best models shown in the figures can't reach completely perfect accuracy in clone detection. A deeper analysis finds out that some code pairs judged in the wrong way have wrong labels in the dataset itself, and this leads to errors that people can't avoid at all. Besides, large language models may make big wrong judgments when people use certain prompt designs. This fact shows that the model's sensitivity to different prompts is still a shortcoming of the current detection ways. The research work in the future can develop in several different directions. First, we need to make further improvement on prompt strategies, and this can make the models work better in detecting those complex code clones. Second, combining the structural information of code and the semantic features of code can help build more reliable frameworks for code clone detection. Third, we should push forward the building of high-quality benchmark datasets in an active way, and these datasets need to include code of many languages and all types of code clones. Finally, we need to pay more attention to finding out the possible value of model adjusting and fitting skills, especially in the cross-language and low-resource situations of code clone detection.

5. Conclusion

This paper makes a full survey and check on LLMs' present state and work results in code clone detection tasks. We review research methods fully and test with a lot of experiments. We find LLM-based methods work well in most cases for code clone detection. They have clear advantages that traditional methods can hardly match, especially in understanding code meanings and cross-language detection. We focus on three technical parts, which are prompt design, code embedding features and model adjustment. We sum up the latest research progress and analyze how different methods apply and their limits in different task situations. In the experiment tests, we take Java code as a typical example. We compare and analyze the work results of many different models. The results show prompt-based methods with general-purpose LLMs do better than specialized pre-trained models in most test standards. They work very stably when finding complex types of code clones. Our study also finds some unsolved problems about text information, clone-pair marks and prompt design. It shows that people can still make more progress in understanding instructions and making tasks fit different situations.

References

[1] Tanizawa, Y., Kawai, M., Takahashi, K., & Takizawa, H. (2026). Semantic equivalence verification of HPC codes using LLMs. Proceedings of the Supercomputing Asia and International Conference on High Performance Computing in Asia Pacific Region Workshops (SCA/HPCAsiaWS '26), 370–377. ACM. https: //doi.org/10.1145/3784828.3785337

[2] Yue, Q., Liu, J., Sun, X., & Zhang, X. (2021). A review of research progress in code clone detection. Computer Science, 48(S02), 14.

[3] Alhijawi, B., Jarrar, R., AbuAlRub, A., & Bader, A. (2025). Deep learning detection method for large language models-generated scientific content. Neural Computing and Applications, 37(1), 91–104. https: //doi.org/10.1007/s00521-024-10538-y.

[4] Sajjadi Mohammadabadi, S. M., Kara, B. C., Eyupoglu, C., Uzay, C., Tosun, M. S., & Karakuş, O. (2025). A Survey of Large Language Models: Evolution, Architectures, Adaptation, Benchmarking, Applications, Challenges, and Societal Implications. Electronics, 14(18), 3580. https: //doi.org/10.3390/electronics14183580

[5] Mao, Y., Xu, Y., Li, J., Meng, F., Yang, H., Zheng, Z., Wang, X., & Zhang, M. (2025). LIFT: Improving Long Context Understanding of Large Language Models through Long Input Fine-Tuning. ArXiv, abs/2502.14644.

[6] Marvin, G., Hellen, N., Jjingo, D., Nakatumba-Nabende, J. (2024). Prompt Engineering in Large Language Models. In: Jacob, I.J., Piramuthu, S., Falkowski-Gilski, P. (eds) Data Intelligence and Cognitive Informatics. ICDICI 2023. Algorithms for Intelligent Systems. Springer, Singapore. https: //doi.org/10.1007/978-981-99-7962-2_30

[7] Khajezade, M., Wu, J.J., Fard, F.H., Rodríguez-Pérez, G., & Shehata, M.S. (2024). Investigating the Efficacy of Large Language Models for Code Clone Detection. 2024 IEEE/ACM 32nd International Conference on Program Comprehension (ICPC), 161-165.

[8] Dou, S., Shan, J., Jia, H., Deng, W., Xi, Z., He, W., Wu, Y., Gui, T., Liu, Y., & Huang, X. (2023). Towards Understanding the Capability of Large Language Models on Code Clone Detection: A Survey. ArXiv,

abs/2308.01191.

[9] Moumoula, M. B., Kabore, K., Klein, J., & Bissyandé, T. (2024). Large language models for cross-language code clone detection. arXiv. https: //doi.org/10.48550/arXiv.2408.04430

[10] Kotsiantis, S., Verykios, V., & Tzagarakis, M. (2024). AI-Assisted Programming Tasks Using Code Embeddings and Transformers. Electronics, 13(4), 767. https: //doi.org/10.3390/electronics13040767

[11] Nashaat, M., Amin, R., Eid, A. H., & Abdel-Kader, R. F. (2025). An enhanced transformer-based framework for interpretable code clone detection. Journal of Systems and Software, 222(C), 112123. https: //doi.org/10.1016/j.jss.2025.112123

[12] Du, Yangkai et al. "AdaCCD: Adaptive Semantic Contrasts Discovery based Cross Lingual Adaptation for Code Clone Detection." ArXiv abs/2311.07277 (2023): n. pag.

[13] Zhang, Z., & Saber, T. (2025). Exploring the Boundaries Between LLM Code Clone Detection and Code Similarity Assessment on Human and AI-Generated Code. Big Data and Cognitive Computing, 9(2), 41. https: //doi.org/10.3390/bdcc9020041

[14] Zhang, Z., & Saber, T. (2024). Assessing the code clone detection capability of large language models. 2024 IEEE International Conference on Code Quality (ICCQ), 75–83.

[15] Moumoula, M.B., Kaboré, A.K., Klein, J., & Bissyandé, T.F. (2024). The Struggles of LLMs in Cross-Lingual Code Clone Detection. Proceedings of the ACM on Software Engineering, 2, 1023 - 1045.

[16] Kamiya, T., Kusumoto, S., & Inoue, K. (2002). CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. IEEE Trans. Software Eng., 28, 654-670.

[17] Jiang, L., Misherghi, G., Su, Z., & Glondu, S. (2007). DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. 29th International Conference on Software Engineering (ICSE'07), 96-105.

[18] Cordy, J. R., & Roy, C. K. (2011). The NiCad clone detector. IEEE International Conference on Program Comprehension. 219-220. 10.1109/ICPC.2011.26.

[19] Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., & Lopes, C. V. (2016). SourcererCC: Scaling code clone detection to big-code. Proceedings of the 38th International Conference on Software Engineering(pp. 1157-1168). ACM. https: //doi.org/10.1145/2884781.2884877

[20] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. ArXiv, abs/2002.08155.

[21] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Yin, J., Jiang, D., & Zhou, M. (2020). GraphCodeBERT: Pre-training Code Representations with Data Flow. ArXiv, abs/2009.08366.

[22] Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). UniXcoder: Unified Cross-Modal Pre-training for Code Representation. Annual Meeting of the Association for Computational Linguistics.

[23] Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., & others. (2024). Qwen2 technical report. arXiv preprintarXiv: 2407.10671.

[24] Team, G. L. M., Zeng, A., Xu, B., Wang, B., Zhang, C., Yin, D., et al. (2024). ChatGLM: A family of large language models from GLM-130B to GLM-4 all tools (Version 1). arXiv. https: //arxiv.org/abs/2406.12793

[25] Arshad, S., Abid, S., & Shamail, S. (2022). CodeBERT for code clone detection: A replication study. 2022 IEEE 16th International Workshop on Software Clones (IWSC)(pp. 39-45). IEEE. https: //doi.org/10.1109/IWSC55060.2022.00015.