Language Models are Better Bug Detector Through Code-Pair Classification

Anonymous ACL submission

Abstract

Large language models (LLMs) such as GPT-3.5 and CodeLlama are powerful models for code generation and understanding. Fine-tuning these models comes with a high computational cost and requires a large labeled dataset. Alternatively, in-context learning techniques allow models to learn downstream tasks with only a few examples. Recently, researchers have shown how in-context learning performs well in bug detection and repair. In this paper, we propose code-pair classification task in which both the buggy and non-buggy versions are given to the model, and the model identifies the buggy ones. We evaluate our task in real-world dataset of bug detection and two most powerful LLMs. Our experiments indicate that an LLM can often pick the buggy from the non-buggy version of the code, and the code-pair classification task is much easier compared to be given a snippet and deciding if and where a bug exists. Code and data are attached with the submission.

1 Introduction

Large language models (LLMs) like GPT-3.5 (Brown et al., 2020) and CodeLlama (Rozière et al., 2023) have shown impressive capabilities in a variety of source code tasks, including code generation, bug repair, and defect prediction (Alrashedy et al., 2023). These models have billions of parameters, which makes it difficult to fine-tune them for downstream tasks due to limited resources and the requirement for a large labeled dataset. Gathering real-world data is costly and requires human effort. However, in-context learning requires a few examples from labeled dataset where these model learn the new task without update the parameters. Recently, in-context learning has demonstrated strong performance in software engineering tasks, achieving better results in some tasks than traditional fine-tuning techniques.

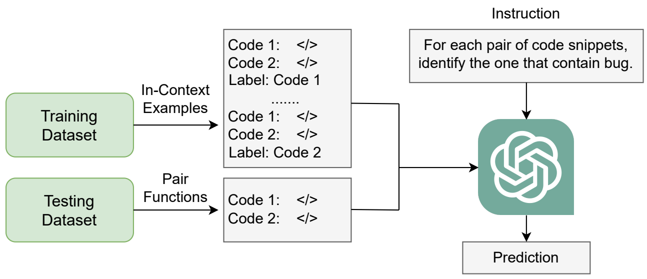

Figure 1: Code-pair classification is an in-context learning approach in which the model receives a pair of functions and identifies the buggy one.

Large language models have demonstrated their capacity to generate code. Additionally, they can debug this generated code without human feedback or the use of external tools. In the real world, developers leverage tools like Co-pilot and GPT to assist in code generation. However, the produced code can occasionally be inaccurate or contain bugs, requiring human intervention for corrections. While these models can generate code, it may still contain bugs. They enhance developer productivity, handling approximately 55% of the tasks. Nonetheless, developers must still verify the accuracy and quality of the generated code. Even though the models can debug, fix, and repair flawed code, they are not yet perfect. Developers allocate about 25–50% of their time to debugging and testing.

The application of LLMs in binary classification tasks for bug detection has been extensively studied. Fine-tuning large language models such as CodeBERT (Feng et al., 2020), CodeT5 (Wang et al., 2021), and PLBART (Ahmad et al., 2021b) on synthetic or weakly labeled data has yielded impressive results on synthetic testing datasets. However, their performance significantly drops when applied to real-world data (Chakraborty et al., 2022). This is because real-world bugs are much more complex. For instance, in Code Snippet 1, the developers makes mistakes in calculating the denominator of the new value. Determining whether this code snippet contains a bug or not is a very challenge, even for human intelligence.

Although numerous prior studies have demonstrated progress in addressing this issue, the performance remains unsatisfactory for real-world applications. In this paper, we introduce a new task: code-pair classification. This involves providing the model with two code snippets—one containing a bug and the other the fixed version. The model's task is to identify the snippet that contains the bug.

2 Related Work

LLMs for bug detection: Applying LLM-based defect detection is an active research area in the artificial intelligence and software engineering communities (Hellendoorn et al., 2020; Chen et al., 2022). (Chen et al., 2023b) proposed self-debugging techniques where the model generates code and then debugs the generated code by itself without human feedback. The model's ability to identify and fix bugs without human intervention enhances the concept of rubber duck debugging. PLBART is a bidirectional and auto-regressive model that was pre-trained on both natural language and source code (Ahmad et al., 2021a). This model follows the same architecture as BART, which is a sequence-to-sequence Transformer (Vaswani et al., 2017). The model was evaluated on vulnerability detection clone detection. (Fu et al., 2022) proposed VulRepair to automatically detect and repair vulnerabilities using the T5 architecture (Raffel et al., 2020).

In-context learning: The (Brown et al., 2020) Introduced the concept of in-context learning, where large language models learn new tasks without updating the model's parameters. This approach has been successfully applied in many applications, such as code generation (Gao et al., 2023) code optimization (Madaan et al., 2023b) and comment generation (Wang et al., 2024). Using the concept of self-consistency in defect repair demonstrates a better improvement than the Chain of Thought (COT) approach, where the author in (Ahmed and Devanbu, 2023) included commit-log messages in a few-shot setting. In (Zhou et al., 2023), the authors introduced DocPrompting, a novel approach that prompts the Language model using relevant documentation, enhancing to improve the accuracy of code generation. The LLM of code shows improvement in code edits and refactoring. In (Madaan et al., 2023a), the authors introduce the Performance-Improving Edits (PIE) dataset tailored for code optimization. Demonstrating a few examples of slower and faster versions of code using in-context learning, the results indicate that the LLM successfully speeds up the program. (Chen et al., 2023a) proposes a "Program of Thoughts" (PoT) prompt where the model generates text and code to solve complex numerical reasoning tasks.

3 Experimental Setup

In this section, we describe the dataset used to evaluate our approach and the chosen pretrained language models.

3.1 Real-world dataset

The PyPIBugs, proposed by (Allamanis et al., 2022), is the largest real-world dataset for bug detection. It contains both the buggy code and its fixed version of functions from real-world applications. The authors did not release their dataset due to licensing limitations, but they provided supplementary materials that help us to reconstruct the dataset. The dataset contain a total of 2,289 buggy functions and each buggy one have its verison of fixed function, so the total is 4578. It has a variety of buggy code types, which include variable misuse, swapped arguments, and incorrect binary operator detection. We randomly split the dataset into training, validation, and testing sets with ratios of 80%, 10%, and 10% respectively.

3.2 Models

Fine-tuning approach: We chose two well-known pretrained models for code, which are CodeBERT and CodeT5. We fine-tune the models through several experiments, using various permutations of hyper-parameters including: batch size {16, 32, 64} and learning rate {3-e6, 1-e5, 2-e5, 3-e5}. We fine-tune the models using the training set, save checkpoints with the lowest validation loss, and then test the models on the testing set.

- **CodeBERT:** A pretrained model based on a Transformer encoder and follows the same architecture as BERT. This model was pre-trained on both source code and natural language. due to the limited resource, we fine-tune codebert-base [1] with 125 millions of parameters.

[1]https://huggingface.co/microsoft/codebert-base

- **CodeT5:** This model, proposed by (Wang et al., 2021), builds on the T5 (Text-to-Text Transfer Transformer) architecture. CodeT5 was pretrained on the CodeSearchNet data and includes a large dataset of C/C# programs that were collected from real-world repositories on GitHub.

In-context learning: We consider two language models, GPT-3.5 and CodeLlam, in evaluating our approach. In the in-context Learning approach, selecting demonstration examples is significantly important, so we followed (Liu et al., 2023) as he demonstrated an excellent technique for choosing the demonstration examples. We embed all examples from both the training and testing sets using the OpenAI “text-embedding-ada-002” model, which is an exceptionally powerful tool for embedding text and code. Subsequently, we train FAISS using the training set and use the testing set to query and select the nearest examples from the training set based on Euclidean distance.

- **GPT-3.5:** This is one of the most powerful models from OpenAI. We conducted our experiments using "GPT-3.5-turbo," which is one of OpenAI’s models boasting a total of 154 billion parameters. It can handle an exceptionally long context of up to 16,385 tokens.

- **CodeLlama:** A large language model for code based on Llama 2. There are two foundational models: CodeLlama-Python, which specializes only in Python, and CodeLlama-Instruct, which is an instruction-following model. All the models are trained on sequences of 16k tokens with 7B, 13B, and 34B parameters each. We use CodeLlama-Instruct with 34B parameters to evaluate our approach.

4 Experimental Results

4.1 Main Results

Fine-tuning results: We adjust CodeBERT and CodeT5 using the training set by varying hyperparameters such as batch size, learning rate, and number of epochs. Subsequently, The model that had the lowest validation loss was evaluated on the test set. Table 1 presents the results of the binary classification task for both CodeBERT and CodeT5. The accuracy is comparable to random guessing at approximately 50%, and both models exhibit significantly poor performance on the F1-score. The models are fine-tuned on a small dataset, which makes it difficult for the model to learn the downstream task.

```python
# Buggy code
def __rel_change(self, new: float) ->
    float:
  if self._likelihoods:
    old = self._likelihoods[-1]
    return abs((new - old) / old)
  return inf

# Fixed code
def __rel_change(self, new: float) ->
    float:
  if self._likelihoods:
    old = self._likelihoods[-1]
    return abs((new - old) / new)
  return inf
```

Code Snippet 1: Example of a variable misuse bug found in real-world code.

Secondly, there is the multi-stage fine-tuning. First, the models are fine-tuned on a large synthetic dataset for bug detection to learn the domain-specific task. Then, they are further fine-tuned on the PyPIBugs dataset. Overall, this approach shows a 10% improvement in accuracy performance. It also significantly improves the F1-score, raising it from 36.41 to 60.26 for codeBERT and from 49.67 to 59.68 for CodeT5.

In-context (binary classification) results: To select demonstration examples, we retrieve relevant samples from the training set using FAISS. For each function, we obtain the nearest functions along with their pairs and labels for context. We then input the test function into the model to predict whether the function contains a bug. This task is a binary classification, similar to the previous one, but now we use GPT-3.5 and CodeLlama. GPT-3.5 achieves slightly better performance than a random guess, with an accuracy of 54%, and the F1-score is around 60%, comparable to multi-stage fine-tuning. On the other hand, CodeLlama demonstrates poor performance in both accuracy and F1-score.

In-context (code-pair classification) results: Since in-context learning is very sensitive to the demonstration examples, we also retrieve relevant pair examples and prompt the model with two paired functions: the buggy version and the fixed version. We then instruct the model to select the buggy one. The results show a significant improvement in accuracy compared to binary classification. For GPT-3.5, the accuracy increased from 54.15%

Table 1: We evaluated the code-pair classification task for bug detection using a real-world dataset. Our results were compared with those of two baseline methods: the fine-tuning approach and in-context binary classification.

Approach	Tasks	Models	Accuracy	F1 Score
Supervised learning (Directly fine-tune)	Binary classification	CodeBERT CodeT5	51.96 50.00	36.41 49.67
Supervised learning (Alrashedy et al., 2023)	Binary classification	CodeBERT CodeT5	61.13 60.48	60.26 59.68
In-context learning	Binary classification	GPT-3.5 CodeLlama	54.15 50.44	60.67 32.24
In-context learning (Ours)	Code-pair classification	GPT-3.5 CodeLlama	72.93 69.87	84.34 82.26

to 72.93%. The accuracy of CodeLlama made an impressive jump from random guessing at 50% to 69.87%. The F1 scores for both models are very impressive, standing at 84.34% and 82.26% respectively.

4.2 Error Analysis

We conducted an experiment on error analysis and found that the model achieves an accuracy of up to 80% on small functions with fewer than 250 tokens. The model learns and performs better with smaller demonstration examples and inputs. We randomly selected 50 misclassified examples and observed that they contained bugs, specifically of the wrong operator type. We noted that the models struggle to distinguish between buggy functions and their fixed versions when the functions exceed 2000 tokens in length.

In the binary classification task for in-context learning, we ran the experiment three times. The accuracy for GPT-3.5 consistently ranged from 53% to 56%, suggesting that the prediction is akin to random guessing. For CodeLlama, the accuracy was 50%, accompanied by a significant drop in the F1-score.

5 Conclusion and Future Work

We introduced the concept of code-pair classification, a novel approach to bug detection in which Large Language Models (LLMs) are given two versions of a function: one with a bug and the other fixed version. The task for the LLMs is to identify the version containing the bug. This approach was evaluated using two advanced LLMs, GPT-3.5 and CodeLlama. The findings suggest that an LLM is often capable of distinguishing the buggy version from the bug-free one. Furthermore, the task of code-pair classification is much easier compared to being given a snippet and deciding if and where a bug exists.

6 Limitations

Our approach assumes that the input to the model consists of a pair of functions: the buggy function and its corrected version. This makes it a much easier task for the model to distinguish the buggy function from the fixed one. For future work, it would be powerful to train the model on pairs of functions rather than on single functions to boost performance. Examples of this include contrastive learning (Li et al., 2023) and consider other loss functions such as triplet and hinge losses. Secondly, our results in bug detection are still not stellar. This is because the performance of LLM on real-world data tends to be low, as cited in (Allamanis et al., 2022; Hellendoorn et al., 2020; Chen et al., 2022). However, our approach demonstrates an improvement in such situations.

Acknowledgements

References

Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021a. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online. Association for Computational Linguistics.

Wasi Uddin Ahmad, S. Chakraborty, B. Ray, and K. Chang. 2021b. Unified pre-training for program understanding and generation.

Toufique Ahmed and Premkumar Devanbu. 2023. Better patching using llm prompting, via self-consistency.

M. Allamanis, H.Jackson-Flux, and M. Brockschmidt. 2022. Self-supervised bug detection and repair.

Kamel Alrashedy, Vincent J. Hellendoorn, and Alessandro Orso. 2023. Learning defect prediction from unrealistic data. *arXiv preprint arXiv:2311.00931*.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam Girish Sastry, Amanda Askell, Sandhini Agarwa, l Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Clemens Winter Jeffrey Wu, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners.

Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep learning based vulnerability detection: Are we there yet?

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023a. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou1. 2023b. Teaching large language models to self-debug.

Zimin Chen, Vincent J Hellendoorn, Pascal Lamblin, Petros Maniatis, Pierre-Antoine Manzagol, Daniel Tarlow, and Subhodeep Moitra. 2022. Plur: A unifying, graph-based view of program learning, understanding, and repair.

Z. Feng, D.Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, and D. Jiang. 2020. Codebert: A pretrained model for programming and natural languages.

Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. Vulrepair: A t5-based automated software vulnerability repair. In *Proceedings of joint meeting on european software engineering conference and symposium on the foundations of software engineering*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models.

Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2020. Global relational models of source code.

Haochen Li, Zhou, Xin, Tuan, Luu Anh, Miao, and Chunyan. 2023. Rethinking negative pairs in code search. *arXiv preprint arXiv:2310.08069*.

Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. 2023. What makes good in-context examples for gpt-3? arXiv:2101.06804. Version 1.

Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023a. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023b.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*.

Chaozheng Zongjie Li Wang, Yun Peng, Shuzheng Gao, Sirong Chen, Shuai Wang, Cuiyun Gao, and Michael R. Lyu. 2024. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning.

Yue Wang, W. Wang, S. Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *International Conference on Learning Representations (ICLR)*, Kigali, Rwanda.