2024 3rd International Conference on Artificial Intelligence and Computer Information Technology (AICIT) | 979-8-3503-6544-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/AICIT62434.2024.10730232

Analysis of the Effectiveness of Large Language Model Feature in Source Code Defect Detection

Tao He
Department of Information Security
Naval University of Engineering
Wuhan, Hubei, China
htps2@163.com

Meini Yang*
Department of Basic Courses
Naval University of Engineering
Wuhan, Hubei, China
ymn_411@126.com
* Corresponding author

Wei Hu
Department of Information Security
Naval University of Engineering
Wuhan, Hubei, China
huwei-1212@126.com

Yun Chen
Department of Information Security
Naval University of Engineering
Wuhan, Hubei, China
mathyun@sina.com

***Abstract*—The defects in software source code threaten the usage of software. With the increase in the quantity and complexity of software, traditional methods of source code defect detection are unable to meet the needs of software security testing, leading to the emergence of machine learning-based source code defect detection methods. The effectiveness of machine learning-based method is closely related to the input features. This paper investigates the impact of using outputs from large language model as input features for machine learning models on the task of source code defect detection. Experimental results demonstrate that adding feature from large language model to machine learning models can enhance the effectiveness of source code defect detection. Furthermore, among different strategies for constructing prompt words, few-shot prompting shows relatively better performance.**

***Keywords*—source code, large language model, defect detection**

I. Introduction

With the rapid development of information technology, the widespread use of various software has increased our work efficiency and made our lives more convenient. However, the emergence of code defects in software has caused various issues during software usage. According to Common Vulnerabilities and Exposures (CVE) data, the number of vulnerabilities in software has been increasing year by year, reaching a high of 28,000 vulnerabilities in 2023, leading to a series of serious security incidents. In such a situation, identifying defects in software source code during software development cycle has become an important issue.

In the traditional software security domain, software defect detection methods can be classified into two types: static analysis and dynamic analysis, based on whether the software under inspection is executed [1]. With the rapid growth in software complexity and quantity, traditional software defect detection methods are unable to meet the practical needs of software security testing. Machine learning is an Artificial Intelligence method that learns effective patterns from historical samples and apply them to predict new samples. Machine learning has achieved good results in various tasks such as image recognition, speech processing, and natural language processing. In this context, researchers are exploring the application of machine learning methods to defect detection in source code. The effectiveness of machine learning methods is closely related to the input features of the model. Therefore, it is necessary to investigate the impact of different input features on the models' performance.

This paper mainly discusses adding outputs from large language model as input features into traditional machine learning models for source code defect detection, analyzing the effectiveness of large language model feature. By employing varied prompt word strategies, the output of the large language model is integrated into the support vector machine model as input. The experiment was conducted on the JavaScript dataset. Experimental results demonstrate that adding features from large language model to machine learning models can enhance the effectiveness of source code defect detection. Among different strategies for constructing prompt words, the strategy of constructing prompt words with fewer samples shows relatively better performance.

II. Related Works

Traditional software defect detection methods can be primarily classified into two categories: static analysis methods and dynamic analysis methods. Static software analysis involves inspecting the source code, bytecode, or compiled program without executing the code, to identify defects within the software. This method enables the detection of software issues before execution, reducing costs associated with debugging and fixing in later stages [2,3]. On the other hand, dynamic analysis is conducted during program execution, gathering various types of information to reveal program defects that depend on specific execution contexts or user inputs [4-6].

With the rapid growth in software scale and complexity, many scholars have applied machine learning techniques to code defect detection tasks. Code defect detection can be abstracted as a classification problem in machine learning [7-9]. Utilizing machine learning techniques allows for the identification of patterns related to code defects from annotated code snippets in training data, and then using these learned patterns to determine whether new code snippets have defects. Machine learning-based code defect detection methods can reduce the time cost and manual effort involved in code defect detection.

When implementing code defect detection tasks using machine learning methods, the input features of the machine learning model significantly influence the model's effectiveness. The input features of machine learning-based source code defect detection methods can be categorized into three types: graph-based features, sequence-based features, and text-based features [10]. Graph-based features transform code into abstract syntax trees (AST), control-flow graphs (CFG), and other graph-based formats, which are then used as inputs for machine learning models to construct code defect detection models [11-12]. Sequence-based features utilize sequence information such as code execution paths and function call sequences as inputs for the model [13-14]. Text-based features directly use code text, compiled assembly instructions, and other text content as feature inputs for machine learning models [15-16].

The development of large language model marks a major advancement in the field of Artificial Intelligence in recent years. These models are trained on massive amounts of text data using deep neural networks, enabling them to learn complex linguistic patterns and generate coherent text outputs tailored to various tasks and inputs. This technology has revolutionized natural language processing and provided significant support for applications such as human-computer interaction, content creation, and knowledge retrieval. Several companies have released their own LLMs, some of which are proprietary while others are open-source. Notable examples of proprietary models include GPT-4 from OpenAI, Gemini from Google, and Yiyan from Baidu. In contrast, examples of open-source models include Meta's Llama, Alibaba's Tongyi Qianwen, Baichuan Intelligence's Baichuan model, and Tsinghua University's GLM model. Additionally, large language model have demonstrated exceptional performance in various software engineering tasks, such as test case generation and code generation [17-18], and they are now being explored for software defect detection by researchers [19].

III. Method

This article plans to train a machine learning model that can determine whether JavaScript code contains defects. In addition to traditional features, the JavaScript code will also be input into a large language model, and the judgment of the large language model on whether the code has defects will be used as a new feature input into the machine learning model.

A. Large Language Model

Language model is a probabilistic model used to compute the likelihood of occurrences of text content. It can compute the probability of a text sequence, reflecting the likelihood of that sequence occurring in the real world. For example, given the sentences ‘I ate an apple.’ and ‘I ate a phone.’, the language model assigns a higher probability to the former, as it aligns better with conventional English usage. Based on this characteristic of language model, given a text sequence, the model can predict the next word that is likely to follow that sequence. For instance, when the input is ‘I ate an’, the language model might predict ‘apple’ as the subsequent word, as this sequence is commonly observed in the corpus.

Large language model is a specific type of language model that exhibit several distinguishing characteristics compared to traditional language models. Firstly, the scale of training data used by large language models far exceeds that of traditional language models, typically reaching tens of billions or even trillions of words. This data is sourced from a variety of domains, including news articles, social media, encyclopedias, and literary works. Such data covers various common forms of natural language, allowing large language models to acquire rich linguistic knowledge. Secondly, the parameter scale of large language models also greatly surpasses that of traditional language models, often comprising tens of billions or even hundreds of billions of parameters. These parameters endow large language models with more powerful expressive capabilities, enabling them to capture more complex and fine-grained linguistic features. Finally, the network structure employed by large language models is typically based on the transformer architecture, which utilizes self-attention mechanisms and consists of multiple encoder and decoder layers. The Transformer architecture efficiently handles long-range dependencies and offers the advantage of high parallelizability. Large language model has had a profound impact on the development of the field of natural language processing. It can be used to implement or enhance various types of natural language processing tasks, such as text classification, named entity recognition, relation extraction, text summarization, machine translation, and question answering systems.

The large language model utilized in this study is Llama 2, an open-source model released by Meta. Llama 2 is an upgraded version of the previously released Llama model, with enhancements in both efficacy and performance. The model features a parameter count ranging from 7 billion to 70 billion.

B. Large Language Model Feature

To generate features for the large language model, specialized prompts need to be constructed. Prompts are fundamental to the use of large language models. The process of designing prompts is known as ‘prompt engineering.’ This involves crafting suitable prompts to supply relevant information, which guides the large language model in performing the required tasks. This article employs three methods: zero-shot prompting, few-shot prompting, and Chain-of-Thought prompting respectively, to generate prompts, and then adds the binary output of the large language model's judgment as a new feature into the machine learning model.

Zero-shot prompting refers to the ability of a large language model to complete a specific task without having seen any data related to that task previously. In other words, directly providing JavaScript code to the large language model and asking it whether the code has defects, and using the binary output of the large language model as a feature input into the machine learning model. The benefit of using this approach is that it does not require any labeled training data and also helps improve the model's generalization ability. To effectively convey requirements to a large language model, it is crucial to provide detailed contextual information and clear instructions within the prompt. These instructions should outline the specific questions to be posed to the model. Additionally, role-playing elements can be incorporated into the prompt. Role-playing involves assigning a specific identity to the model, based on the context of the task at hand, which allows for more targeted interactions with the model. By defining these roles, the user implicitly supplies the model with relevant background information and context, enhancing the model's ability to understand the problem and generate more accurate responses from the appropriate domain of knowledge.

Although large language model shows excellent zero-shot capabilities, they may not perform well with complex tasks using zero-shot prompting. Few-shot prompting can improve performance by providing demonstration examples within the prompts for context learning. In the task of code defect detection, first, a small amount of JavaScript source code along with labels indicating whether the code has defects is provided to the large language model. Then, the large language model is asked to determine whether new JavaScript code contains defects. Similar to zero-shot prompting, few-shot prompting reduces the need for a large amount of labeled data.

The task of detecting code defects is complex. To improve the performance of large language model in this task, besides zero-shot prompting and few-shot prompting, this paper also uses Chain-of-Thought (CoT) prompting helps model reason step by step, enhancing their ability to detect defects. CoT is a method that encourages large language models to explain their reasoning processes. It aids in understanding and evaluating the model's capabilities and limitations. Compared to traditional cloze-style problem-solving methods, large language models with Chain-of-Thought capabilities can perform reasoning in a manner similar to human learning. In the code defect detection task, on one hand, a small number of instances of code defect judgments along with specific reasons for judgment are provided to the large language model. On the other hand, phrases such as 'Let's think step by step' are added to the prompts, guiding the large language model in determining whether code contains defects.

C. Other Features

Besides the feature from large language model, inspired by the work of MICHAEL PRADEL [20], useful information for code defect detection can also be found in the names of variables and functions. Therefore, this paper extracts information including function names, parameter types, parameter order, binary operators, and so on from JavaScript code as input features. These features are then represented as vectors using embedding techniques, along with the feature from large language model, are input into the machine learning model.

D. Support Vector Machine

This paper utilizes support vector machine (SVM) model as the machine learning model. Support Vector Machine is a machine learning model that utilizes a kernel function to project samples into a high-dimensional space, where it calculates the maximum margin hyperplane to effectively classify positive and negative samples. This method has strong mathematical theoretical support and interpretability. Additionally, its computational complexity depends on the number of support vectors rather than the dimensionality of the sample space, thus mitigating the problem of 'curse of dimensionality' in machine learning.

IV. Experiment

A. Experimental Data and Setup

This paper conducts experiments using the JavaScript dataset employed in the work of MICHAEL PRADEL [20]. The dataset comprises 150,000 JavaScript files collected from various open-source projects. Specifically, 100,000 files from this dataset are used as training data to train the machine learning model, while the remaining 50,000 files serve as test data to evaluate the performance of the trained model. Following their paper's methods for feature extraction, negative sample generation and code defect detection categories, this study focuses on detecting three types of code defects: function parameter order errors, binary operator parameter errors, and incorrect use of binary operators.

B. Experimental Results and Analysis

Due to the significant workload of manually checking whether JavaScript code has defects, in the experiment, only the top 100 JavaScript codes ranked by the confidence level computed by the support vector machine were manually confirmed. When the code does not match the expected results, it is considered to have defects.

TABLE I. Recognition Accuracy of Different Types of Errors

	Function parameter order error	**Binary operator parameter errors**	**Binary operators errors**
traditional features	31%	35%	43%
traditional features+LLM feature(zero-shot)	35%	46%	50%
traditional features+LLM feature(few-shot)	39%	51%	56%
traditional features+LLM feature(CoT)	38%	51%	54%

The experimental results are shown in Table 1. As seen from Table 1, the addition of large language model feature to the support vector machine model has led to a certain degree of improvement in detecting code defects related to function parameter order errors, binary operator parameter errors, and binary operator errors. This is mainly because the large language model is built on a vast amount of text from the internet, including code content. There are significant differences between the JavaScript code of these three types of errors and the common JavaScript code on the Internet, so they can be accurately recognized by the large language model, and consequently enhance the overall recognition performance of the machine learning method.

For zero-shot prompting, few-shot prompting, and Chain-of-Thought prompting, these three different promptings have a positive effect on improving the effects of the model. Among them, few-shot prompting has the greatest effect on code defect recognition. This is mainly because the use of few-shot prompting allows large language model to have a clear understanding of the tasks which it should complete. Based on the information learned before, it has achieved better model effects through transfer learning.

V. Conclusions and FUTURE WORK

This paper constructs a machine learning model to identify defects in JavaScript code. In addition to traditional features, large language model feature is also added to the input features. Experimental results show that large language model feature can improve the accuracy of the machine learning model in judging whether the code has defects. Among zero-shot prompting, few-shot prompting, and Chain-of-Thought prompting, few-shot prompting performs the best in improving the model's ability to identify defect codes.

This study focuses solely on the identification performance of large language model for three types of errors in JavaScript code: function parameter order errors, binary operator parameter errors, and binary operator errors. These errors require only a small amount of code context for identification. However, in real-world code defect detection, there are many defects that require a good understanding of longer code contexts for accurate identification. Identifying such defects will be the research content in the future. Additionally, apart from general large language model, there are also specialized large language model designed specifically for program code. Discussing the defect detection effectiveness of these specialized large language model is also meaningful research.

ACKNOWLEDGMENT

This study was sponsored by the Foundation of Naval University of Engineering (No. 2022508030, No. 2022501040).

REFERENCES

[1] Harzevili, N.S., A.B. Belle, J. Wang, et al., *A Survey on Automated Software Vulnerability Detection Using Machine Learning and Deep Learning.* arXiv preprint arXiv:2306.11673, 2023.

[2] Jovanovic, N., C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. in Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06). 2006. IEEE: 263-268.

[3] Shastry, B., F. Yamaguchi, K. Rieck, et al. Towards vulnerability discovery using staged program analysis. in Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment. 2016. Springer: 78-97.

[4] Ganesh, V., T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. in Proceedings of the 2009 IEEE 31st International Conference on Software Engineering. 2009. IEEE: 474-484.

[5] Rawat, S., V. Jain, A. Kumar, et al. VUzzer: Application-aware Evolutionary Fuzzing. in Proceedings of the 2017 Network and Distributed System Security Symposium. 2017. 1-14.

[6] Pham, V.-T., M. Böhme, and A. Roychoudhury. Model-based whitebox fuzzing for program binaries. in Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. 2016. 543-553.

[7] Jeon, S. and H.K. Kim, AutoVAS: An automated vulnerability analysis system with a deep learning approach. Computers & Security, 2021. 106: 298-308.

[8] Shippey, T., D. Bowes, and T. Hall, Automatically identifying code features for software defect prediction: Using AST N-grams. Information and Software Technology, 2019. 106: 142-160.

[9] Dinella, E., H. Dai, Z. Li, et al. Hoppity: Learning graph transformations to detect and fix bugs in programs. in Proceedings of the International Conference on Learning Representations. 2020. 233-239.

[10] Lin, G., S. Wen, Q.-L. Han, et al. Software vulnerability detection using deep neural networks: A survey. in Proceedings of the IEEE. 2020. 1825-1848.

[11] Wang, S., T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. in Proceedings of the 38th International Conference on Software Engineering. 2016. 297-308.

[12] Nam, J., S.J. Pan, and S. Kim. Transfer defect learning. in Proceedings of the 35th International Conference on Software Engineering. 2013. IEEE: 382-391.

[13] Grieco, G., G.L. Grinblat, L. Uzal, et al. Toward large-scale vulnerability discovery using machine learning. in Proceedings of the sixth ACM Conference on Data and Application Security and Privacy. 2016. 85-96.

[14] Li, Z., D. Zou, S. Xu, et al., Vuldeepecker: A deep learning-based system for vulnerability detection. arXiv preprint arXiv:1801.01681, 2018.

[15] Peng, H., L. Mou, G. Li, et al. Building program vector representations for deep learning. in Proceedings of the 8th International Conferenc on Knowledge Science, Engineering and Management. 2015. Springer: 547-553.

[16] Lee, Y.J., S.-H. Choi, C. Kim, et al. Learning binary code with deep learning to detect software weakness. in Proceedings of the 9th International Conference on Internet (ICONI) Symposium. 2017. 211-219.

[17] Wang, J., Y. Huang, C. Chen, et al., Software testing with large language model: Survey, landscape, and vision. IEEE Transactions on Software Engineering, 2024. 50: 911-936.

[18] Xu, F.F., U. Alon, G. Neubig, et al. A systematic evaluation of large language model of code. in Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming. 2022. 1-10.

[19] Yang, A.Z., C. Le Goues, R. Martins, et al. Large language model for test-free fault localization. in Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024. 1-12.

[20] Pradel, M. and K. Sen. Deepbugs: A learning approach to name-based bug detection. in Proceedings of the ACM on Programming Languages. 2018. 1-25.