Fine-Tuning DistilBERT: A Pre-Trained Transformer Model for Software Defect Prediction

Nasraldeen Alnor Adam Khleel[1]* and Károly Nehéz[2]

[1*]Department of Information Engineering, University of Miskolc, Miskolc, H-3515, Hungary.
nasr.alnor@uni-miskolc.hu
[2]Department of Information Engineering, University of Miskolc, Miskolc, H-3515, Hungary.
aitnehez@uni-miskolc.hu

Abstract. Software Defect Prediction (SDP) is an important software engineering task that aims to identify defective code early in the development lifecycle, thereby improving software quality and reducing maintenance costs. Traditional software defect prediction methods often rely on handcrafted features or shallow representations that may fail to capture the rich semantic and contextual information contained in source code. Recent advances in Natural Language Processing (NLP) have introduced transformer-based models capable of learning complex patterns and long-range dependencies directly from sequential data. However, many existing approaches employ large and computationally expensive models, limiting their practicality in resource-constrained environments. This study proposes a software defect prediction approach based on the supervised fine-tuning of DistilBERT, a lightweight variant of BERT that retains strong representation capabilities while significantly reducing computational requirements. The proposed model was evaluated on the Devign dataset from the CodeXGLUE benchmark using Accuracy, Precision, Recall, and F1-score metrics. Experimental results show that the fine-tuned DistilBERT model achieved an Accuracy of 71.43%, Precision of 69.47%, Recall of 65.44%, and an F1-score of 67.39%. In addition, the proposed approach required substantially less training time than several baseline and existing transformer-based methods evaluated on the same benchmark dataset. The findings demonstrate that lightweight transformer architectures can provide an effective balance between predictive performance and computational efficiency, making DistilBERT a practical solution for software defect prediction in resource-constrained software engineering environments.

Keywords: Software Defect Prediction; DistilBERT; Transformer Models; Transfer Learning; Fine-Tuning.

© The Author(s) 2026
S. Árvai-Homolya et al. (eds.), *Proceedings of the 2nd Innovation in Engineering Education Conference (IEE 2026)*, Atlantis Highlights in Social Sciences, Education and Humanities 58,
https://doi.org/10.2991/978-94-6239-823-8_2

1 Introduction

The increasing complexity of modern software systems has made Software Defect Prediction (SDP) a crucial research area. Software defects can negatively impact reliability, performance, and security, often leading to system failures and increased maintenance costs. To mitigate these risks, researchers have developed SDP techniques to identify defective code early in the software development lifecycle [1]. Traditional machine learning approaches, including Decision Trees, Naive Bayes, Random Forest, and Support Vector Machines, primarily rely on static code analysis and handcrafted features. While these methods can be effective in certain contexts, they often struggle to generalize across diverse codebases and fail to capture deeper semantic and contextual information embedded in source code [2]. Recent advances in Natural Language Processing (NLP) and transfer learning have introduced pre-trained transformer-based models capable of learning rich semantic representations from large-scale unlabeled data [3]. By treating source code as a sequence of tokens, these models can capture structural patterns and long-range dependencies that traditional feature-based approaches frequently overlook. Among these models, DistilBERT is a distilled and computationally efficient variant of BERT that retains strong language understanding capabilities while significantly reducing model size and training requirements [4–6]. Although large-scale transformer models such as CodeBERT and RoBERTa have demonstrated strong performance in software engineering tasks, their computational demands may limit practical adoption in resource-constrained environments. Furthermore, most existing studies focus primarily on maximizing predictive performance, with comparatively less attention given to the trade-off between predictive effectiveness and computational efficiency. Consequently, the potential of lightweight transformer architectures such as DistilBERT for software defect prediction remains insufficiently explored. To address this gap, this study investigates the effectiveness of fine-tuning DistilBERT for SDP using the Devign dataset from the CodeXGLUE benchmark. The proposed approach is evaluated using multiple performance metrics, including Accuracy, Precision, Recall, and F1-score, and is compared with baseline and state-of-the-art methods to assess both predictive performance and computational efficiency. The main contributions of this study are as follows:

(i) We investigate the effectiveness of fine-tuning DistilBERT for software defect prediction, emphasizing its potential as a lightweight and efficient transformer-based model.

(ii) We develop a complete DistilBERT-based SDP framework and evaluate its performance on a widely used benchmark dataset.

(iii) We demonstrate that DistilBERT achieves competitive predictive performance while significantly reducing training time and computational cost compared with several baseline and existing approaches.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the proposed methodology. Section 4 discusses the experimental results. Section 5 outlines the threats to validity and reproducibility. Finally, Section 6 concludes the paper and presents directions for future research.

2 Related Work

Recent advances in transformer-based architectures have significantly influenced research in software engineering tasks, including software defect prediction (SDP), vulnerability detection, and code summarization. A variety of BERT-based models, such as BERT Base, DistilBERT, RoBERTa, CodeBERT, and CuBERT, have been developed to enhance the efficiency, scalability, and adaptability of the original BERT architecture [5]. These models leverage self-supervised learning to capture semantic and contextual representations from large-scale datasets, making them well-suited for analyzing source code. Among these models, CodeBERT has received considerable attention due to its bimodal pre-training on both natural language and programming languages [8]. Several studies [8–10] have applied CodeBERT to software defect prediction and demonstrated its strong ability to capture semantic relationships in code. In addition to defect prediction, CodeBERT has been successfully applied to tasks such as code summarization, clone detection, and code search [11, 12]. Another line of research focuses on CuBERT, which is trained on a large-scale corpus of Python code from GitHub [9]. CuBERT has demonstrated strong performance across classification and program analysis tasks. Similarly, RoBERTa and other BERT-based approaches have been employed for software defect prediction and related software engineering applications [4, 13]. In addition to domain-specific transformer models, JavaBERT [1] and hybrid approaches combining BERT with BiLSTM networks, attention mechanisms, and data augmentation techniques [2] have shown promising results in software defect prediction. Recent studies have also investigated the limitations and generalization capabilities of machine-learning-based approaches for vulnerability detection. Risse and Böhme [6] demonstrated that state-of-the-art vulnerability detection models may overfit to dataset-specific patterns and struggle to generalize under semantic-preserving code transformations, highlighting the importance of rigorous evaluation methodologies for assessing the true capabilities of machine-learning models in software security tasks. Transformer-based models have also been successfully applied to related software engineering tasks, such as long-lived bug prediction [7] and bug-fixing time estimation [14], highlighting their versatility beyond defect prediction. To provide a concise overview of the most relevant studies discussed in this section, Table 1 summarizes the key references, authors, publication years, research problems, and solution methods related to software defect prediction and transformer-based software engineering applications.

Despite the extensive exploration of large-scale and domain-specific transformer models, most existing approaches prioritize predictive performance, often at the expense of computational efficiency and scalability. In contrast, relatively limited attention has been given to lightweight transformer architectures such as DistilBERT. While DistilBERT offers significant advantages in terms of reduced model size, faster training, and lower resource consumption, its effectiveness for software defect prediction remains underexplored. Furthermore, most prior studies have focused on maximizing predictive accuracy, with comparatively less emphasis on achieving an effective balance between classification performance and computational efficiency. Therefore, this

study aims to address this gap by investigating whether DistilBERT can achieve competitive performance in SDP while providing substantial efficiency benefits, particularly in resource-constrained environments.

Table 1. Summary of Related Studies in Software Defect Prediction.

Reference	Authors	Publication Year	Problem	Solving Method
[1]	Cheng and Takada	2023	Software defect prediction	JavaBERT embeddings combined with a CNN-BiLSTM architecture
[2]	Uddin et al.	2022	Software defect prediction	BERT-based semantic features with BiLSTM, attention mechanism, and data augmentation
[4]	Briciu et al.	2023	Software defect prediction	RoBERTa and CodeBERT semantic feature extraction for defect classification
[6]	Risse and Böhme	2024	Vulnerability detection and model generalization	Evaluation framework for analyzing robustness and overfitting of ML-based vulnerability detection models
[7]	Gomes et al.	2023	Long-lived bug prediction	BERT- and TF-IDF-based feature extraction with machine learning classifiers
[8]	Feng et al.	2020	Code understanding and representation learning	CodeBERT pre-training on natural language and programming language pairs
[9]	Kanade et al.	2020	Source code understanding	CuBERT contextual embeddings for code classification and program analysis tasks
[10]	Pan et al.	2021	Software defect prediction	Fine-tuned CodeBERT models for cross-version and cross-project defect prediction
[11]	Mohammadkhani et al.	2023	Explainability of code models	Analysis of CodeBERT and GraphCodeBERT using attention-based explainability techniques
[12]	Zhou et al.	2021	Generalizability of CodeBERT	Evaluation of CodeBERT across code search and defect prediction tasks
[13]	Yang et al.	2022	API review classification	Fine-tuning transformer-based models including BERT and RoBERTa
[14]	Ardimento and Mele	2020	Bug-fixing time prediction	Transfer learning with BERT for bug report classification and resolution-time prediction

3 Methodology

This study adopts a supervised fine-tuning approach based on DistilBERT to address the software defect prediction (SDP) task. The proposed methodology leverages the contextual representation capabilities of transformer-based models to learn semantic patterns from source code. The overall framework follows a structured pipeline consisting of five key stages: (i) data collection and dataset modeling, (ii) data preprocessing, (iii) feature preparation, (iv) model construction and fine-tuning, and (v) performance evaluation. Each stage is carefully designed to ensure that the input data are appropriately transformed and that the model can effectively learn discriminative features for defect prediction. Unlike traditional SDP approaches that rely on handcrafted code metrics, the proposed method utilizes contextual embeddings generated by a pretrained DistilBERT model, enabling the capture of syntactic and semantic relationships within source code. This allows the model to represent complex code structures and long-range dependencies more effectively. Figure 1 illustrates the overall workflow of the proposed methodology, and each stage is described in detail in the following subsections.

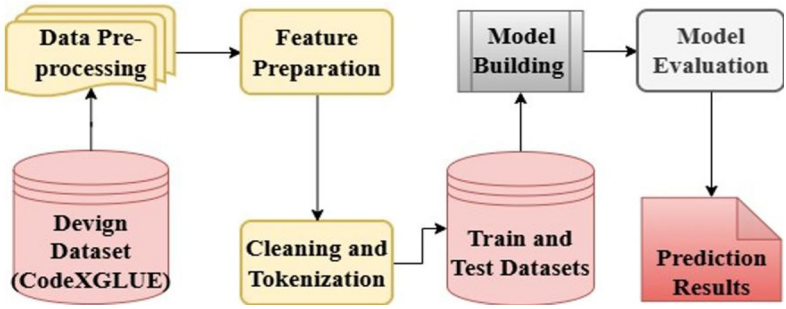


Fig. 1. The proposed methodology.

3.1 Data Collection and Modeling

Benchmark datasets play a critical role in advancing machine learning research for programming language tasks by enabling standardized evaluation, reproducibility, and fair comparison among competing approaches. In this study, the Devign dataset from the CodeXGLUE benchmark suite [15] was utilized for software defect prediction. The dataset consists of real-world C language functions collected from open-source software projects and annotated with binary labels indicating whether a code sample is defective (vulnerable) or non-defective. Originally developed for vulnerability detection research, Devign includes various security-related defects, such as resource leaks, use-after-free vulnerabilities, buffer-related issues, and denial-of-service weaknesses [16]. CodeXGLUE provides a comprehensive benchmark framework for code intelligence applications, including code classification, defect detection, code summarization, and code search. Within this framework, Devign has become one of the most widely

used datasets for software defect prediction and vulnerability detection studies, facilitating direct comparison with previously published approaches. The dataset supports both binary and multi-class classification settings. In this study, the binary classification setting was adopted, where code samples are categorized as either defective or non-defective. The standard CodeXGLUE training, validation, and test partitions were used to ensure consistency with existing benchmark studies and to avoid data leakage between dataset splits. The selection of the Devign dataset was motivated by its use of manually annotated defect labels derived from real-world software projects, its widespread adoption in software engineering research, and its integration within the CodeXGLUE benchmark framework. Table 2 summarizes the key characteristics of the dataset, including the programming language, classification task, and the sizes of the training, validation, and test splits used in this study.

Table 2. A summary of the Devign dataset.

Task	Dataset Name	Language	Train Size	Validation Size	Test Size
Defect Detection	Devign	C	21,854	2,732	2,732

3.2 Data Preprocessing and Feature Preparation

As DistilBERT operates on tokenized textual input, each source code snippet undergoes preprocessing before model training. Initially, irrelevant elements such as excessive whitespace and inconsistent formatting are removed to standardize the input representation. Subsequently, the cleaned code snippets are tokenized using the DistilBERT WordPiece tokenizer, which converts the source code into sequences of subword tokens suitable for transformer-based processing. To ensure consistent input dimensions, tokenized sequences are padded or truncated to a fixed maximum length. In addition, attention masks are generated to distinguish between valid tokens and padded positions, enabling the model to focus on meaningful parts of the input during training. For feature preparation, the proposed approach leverages contextual embeddings produced by the pre-trained DistilBERT encoder. These embeddings capture both syntactic and semantic information from source code, enabling automatic feature extraction without relying on handcrafted software metrics. The resulting contextual representations are subsequently passed to a classification layer for downstream defect prediction, forming an end-to-end learning framework [7].

3.3 Model Building and Performance Evaluation

The proposed predictive model is built on the DistilBERT architecture, a distilled variant of the BERT encoder that balances computational efficiency and representational capability. A task-specific classification head, implemented as a fully connected linear layer, is added on top of the DistilBERT encoder to perform binary classification by predicting whether a given source code fragment is defective or non-defective. The entire model is fine-tuned end-to-end using labeled training data, allowing both the pre-trained encoder and classification layer to adapt to the software defect prediction task.

The final model is evaluated on an unseen test dataset to assess its generalization capability. To evaluate the effectiveness of the proposed approach, a confusion matrix serves as the basis for computing standard classification metrics. The confusion matrix, shown in Table 3, categorizes predictions into four groups: True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN). Based on these values, four widely used evaluation metrics are computed: Accuracy, Precision, Recall, and F1-score. These metrics collectively provide a comprehensive assessment of classification performance by measuring overall correctness and the trade-off between false positives and false negatives [17,18]. Finally, the performance of the proposed DistilBERT model is compared with baseline and state-of-the-art approaches to assess its effectiveness and computational efficiency in software defect prediction.

Table 3. The Confusion matrix.

Predicted	Actual	
	Class X	Class Y
Class X	TP	FN
Class Y	FP	TN

$$Accuracy = (TP + TN) / (TP + TN + FP + FN) \quad (1)$$

$$Precision = TP / (TP + FP) \quad (2)$$

$$Recall = TP / (TP + FN) \quad (3)$$

$$F1\text{-}Score = (2 * Recall * Precision) / (Recall + Precision) \quad (4)$$

3.4 Implementation Details and Experimental Environment

The proposed software defect prediction framework was implemented in Python using the PyTorch deep learning framework and the Hugging Face Transformers library. The pre-trained DistilBERT model (`distilbert-base-uncased`) was fine-tuned for binary classification on the Devign dataset from the CodeXGLUE benchmark. The dataset was loaded from preprocessed Parquet files and prepared for supervised learning using a custom dataset class. Model training was performed using the Hugging Face Trainer API. The training configuration consisted of 5 epochs, a per-device training batch size of 4, and a gradient accumulation step of 4, resulting in an effective batch size of 16. For evaluation, a batch size of 64 was employed. Optimization was performed using the Adafactor optimizer with a weight decay coefficient of 0.01. A linear learning-rate warm-up strategy with 500 warm-up steps was applied to improve training stability. In addition, mixed-precision (FP16) training was enabled to reduce memory consumption and improve computational efficiency. The DistilBERT model was fine-tuned end-to-end without freezing any encoder layers, allowing all model parameters to be updated during training. Model performance was evaluated at the end of each epoch to monitor the learning process and identify the best-performing model configuration. All experiments were conducted in Google Colab using an NVIDIA Tesla P100 GPU accelerator.

The final model was evaluated on the test dataset using Accuracy, Precision, Recall, and F1-score metrics. A confusion matrix was also generated to provide additional insight into classification performance and error distribution.

3.5 Proposed Framework

The overall workflow of the proposed DistilBERT-based software defect prediction framework is summarized in Algorithm 1.

Algorithm 1. DistilBERT-Based Software Defect Prediction Framework

Input: Devign dataset D

Output: Fine-tuned DistilBERT model M and prediction results

1. Load training, validation, and test partitions from D.
2. Preprocess source code samples and corresponding labels.
3. Remove samples that exceed the maximum input length.
4. Initialize the DistilBERT tokenizer.
5. Tokenize source code samples and generate attention masks.
6. Construct training, validation, and test datasets.
7. Load the pre-trained DistilBERT model with a binary classification head.
8. Configure model training parameters.
9. Fine-tune the DistilBERT model using the training dataset.
10. Monitor performance on the validation dataset during training.
11. Evaluate the final model on the test dataset.
12. Compute Accuracy, Precision, Recall, and F1-score.
13. Generate a confusion matrix for error analysis.
14. Return the fine-tuned model M and evaluation results.

4 Experimental results and discussions

This section presents the experimental results obtained using the proposed DistilBERT-based approach for software defect prediction. As described in the methodology, the model was fine-tuned on the Devign dataset and evaluated using standard classification metrics, including Accuracy, Precision, Recall, and F1-score. Table 4 summarizes the performance of the fine-tuned DistilBERT model across these metrics. The results indicate that DistilBERT achieves an Accuracy of 71.43%, along with Precision, Recall, and F1-score values of 69.47%, 65.44%, and 67.39%, respectively. These results demonstrate balanced classification performance and suggest that the model can identify defective code with a reasonable trade-off between false positives and false negatives. The relatively close values of Precision and Recall indicate that the model does not exhibit a strong bias toward either over-predicting or under-predicting defective instances, contributing to stable overall performance. The experimental findings further

demonstrate that a lightweight transformer architecture can effectively adapt to software defect prediction through supervised fine-tuning. Despite being substantially smaller than many code-specific transformer models, DistilBERT is capable of capturing meaningful contextual information from source code and achieving competitive predictive performance on the Devign benchmark dataset. A qualitative examination of the prediction outcomes suggests that misclassified samples are often associated with complex code structures, long functions, or subtle defect patterns that require deeper semantic understanding of program behavior. Such cases may involve long-range dependencies, intricate control flows, or context-sensitive defects that are difficult to capture using token-based representations alone. These observations indicate that integrating additional program representations, such as abstract syntax trees, control-flow graphs, or data-flow information, may further improve defect prediction performance. When comparing the proposed approach with previously reported studies, it should be noted that direct comparisons remain challenging because many software defect prediction studies report only Accuracy while omitting additional evaluation metrics such as Precision, Recall, and F1-score. Since software defect datasets are frequently imbalanced, Accuracy alone may not provide a comprehensive assessment of model effectiveness. Therefore, this study reports multiple evaluation metrics to provide a more reliable and informative evaluation of classification performance. Overall, the findings suggest that DistilBERT provides a practical balance between predictive performance and computational efficiency. While larger code-specific models may achieve higher predictive performance due to specialized pretraining on source code repositories, the proposed approach offers advantages in terms of model size, computational requirements, and training efficiency. These characteristics make DistilBERT a viable alternative for software defect prediction in resource-constrained environments and rapid development scenarios.

Table 4. Performance analysis for the proposed DistilBERT model.

Model	Performance Measures			
DistilBERT	Accuracy	Precision	Recall	F1-Score
	71.43	69.47	65.44	67.39

To further evaluate the effectiveness of the proposed DistilBERT model, its performance was compared with several baseline and state-of-the-art methods reported in previous studies. However, direct comparisons remain challenging because many software defect prediction studies primarily report Accuracy while omitting additional evaluation metrics such as Precision, Recall, F1-score, and AUC. Since software defect prediction datasets are often imbalanced, relying solely on Accuracy may provide an incomplete assessment of model performance. Therefore, this study reports multiple evaluation metrics to provide a more comprehensive and reliable evaluation of the proposed approach. Table 5 compares the proposed method with existing approaches. While some prior models report higher Accuracy under specific experimental settings, many studies do not provide a complete set of evaluation metrics, making direct com-

parisons difficult. Furthermore, differences in datasets, preprocessing strategies, training configurations, and evaluation protocols may influence the reported results. Consequently, the comparison should be interpreted with caution. Despite these limitations, the proposed DistilBERT-based model demonstrates competitive performance on the Devign dataset, achieving balanced Precision, Recall, and F1-score values. This indicates that the model maintains a reasonable trade-off between false positives and false negatives, which is essential in practical software defect prediction scenarios. A qualitative examination of the prediction outcomes suggests that some misclassified samples are associated with complex code structures, long functions, or subtle defect patterns that require deeper semantic understanding of program behavior. These observations highlight opportunities for future improvements through the incorporation of additional structural code information.

Table 5. Comparison of the proposed model with baselines and existing approaches.

Model		Performance Measures				
		Dataset	Accuracy	Precision	Recall	F1-Score
Baseline Models	RoBERTa [15]	CodeXGLUE- Devign	61.05	-	-	-
	CodeBERT [15]	CodeXGLUE- Devign	62.08	-	-	-
	CodeBERT [3]	CodeXGLUE- Devign	61.82	-	-	-
	GraphCodeBERT [3]	CodeXGLUE- Devign	61.38	-	-	-
	RoBERTa [19]	CodeXGLUE- Devign	56.62	-	-	-
	CodeBERT [19]	CodeXGLUE- Devign	63.91	-	-	-
Other Models	UniXcoder [6]	CodeXGLUE- Devign	69.30	71.70	78.70	68.00
	CoTexT [6]	CodeXGLUE- Devign	67.30	68.40	85.10	63.50
	GraphCB [6]	CodeXGLUE- Devign	65.50	73.40	66.10	62.90
	CodeBERT [6]	CodeXGLUE- Devign	65.10	84.70	57.20	59.60
	VulBERTa [6]	CodeXGLUE- Devign	63.90	64.30	75.90	65.20
	PLBart [6]	CodeXGLUE- Devign	63.30	67.50	65.80	61.80
	RoBERTa [20]	QT	87.00	27.00	48.00	35.00
	CodeBERT [20]	QT	87.00	26.00	42.00	32.00
	RoBERTa [20]	Openstack	80.00	32.00	50.00	39.00
	CodeBERT [20]	Openstack	80.00	33.00	56.00	42.00
Proposed model	DistilBERT	CodeXGLUE- Devign	71.43	69.47	65.44	67.39

The results further demonstrate that a lightweight transformer architecture can achieve performance comparable to more complex and resource-intensive approaches. While code-specific models may achieve higher predictive performance due to specialized pretraining on source code repositories, DistilBERT offers advantages in terms of

computational efficiency, memory requirements, and training time. This trade-off between predictive performance and computational cost makes DistilBERT particularly attractive for resource-constrained environments and rapid prototyping scenarios. Overall, the findings suggest that DistilBERT represents a viable alternative for software defect prediction when computational efficiency is an important consideration. The results also emphasize the importance of evaluating models using multiple performance metrics to obtain a more reliable and comprehensive assessment of performance than can be achieved through Accuracy alone.

Furthermore, the training time and computational requirements of the proposed DistilBERT-based approach were compared with those reported for baseline and state-of-the-art methods evaluated on the same dataset. As shown in Table 6, all approaches were trained using GPU-based accelerators, including either a single NVIDIA P100 or a dual P100 (P100 ×2) configuration. For consistency, training time is defined as the duration required to train a model from initialization to completion, excluding data preprocessing and evaluation. The results indicate that the proposed DistilBERT model requires substantially less training time than the baseline and other transformer-based approaches. This improvement can be attributed to DistilBERT's compact architecture, which contains fewer parameters than larger transformer models while retaining strong representation capabilities. Although differences in implementation details, optimization strategies, and hardware configurations may influence training-time comparisons, the observed results consistently demonstrate the efficiency advantages of the proposed approach. The reduced computational cost enables faster experimentation and model iteration while maintaining competitive predictive performance. These characteristics make DistilBERT particularly suitable for practical software engineering applications, especially in environments where computational resources, development time, or deployment constraints are important considerations.

Table 6. Training time and computational setup comparison.

Model		Training Time (min)	GPU Configuration
Baseline Models	RoBERTa [15]	60	GPUs P100 x2
	CodeBERT [15]	60	GPUs P100 x2
Other Models	CodeBERT [3]	49	GPUs P100
	GraphCodeBERT [3]	73	GPUs P100
	RoBERTa [19]	100	GPUs P100
	CodeBERT [19]	100	GPUs P100
Proposed model	DistilBERT	17	GPUs P100

5 Threats to Validity and Reproducibility

Although the proposed DistilBERT model achieved promising results on the Devign benchmark dataset, several factors may influence the validity and generalizability of

the findings. The experiments were conducted using the standard CodeXGLUE training, validation, and test partitions, and all preprocessing and tokenization procedures were applied independently to prevent information leakage between data splits. Nevertheless, the reported results are based on a single training run. Consequently, the variability associated with random initialization, optimization dynamics, and hardware-specific factors was not explicitly quantified. As a result, the reported performance metrics should be interpreted as indicative rather than definitive estimates of model effectiveness. Future studies should perform multiple independent runs and report statistical measures such as standard deviations, confidence intervals, and significance tests to provide a more rigorous assessment of model robustness. A further limitation is that the evaluation was conducted on a single benchmark dataset. Although the Devign dataset is widely adopted and provides a standardized basis for comparison, the results may not fully generalize to other software projects, programming languages, or defect prediction scenarios. Additional experiments on diverse datasets would strengthen the external validity of the findings. To support reproducibility, the implementation details, training configuration, dataset source, and evaluation procedures are fully described in Section 3. Future work will further assess the proposed approach across multiple software engineering benchmarks to provide additional evidence of its effectiveness and generalizability.

6 Conclusion

This study presented a software defect prediction approach based on supervised fine-tuning of DistilBERT, a lightweight transformer model. Experimental evaluation on the Devign benchmark dataset demonstrated that the proposed approach can effectively distinguish between defective and non-defective source code instances. The model achieved an Accuracy of 71.43%, Precision of 69.47%, Recall of 65.44%, and an F1-score of 67.39%, indicating balanced classification performance and a reasonable trade-off between false positives and false negatives. In addition to achieving competitive predictive performance, the proposed model required substantially less training time and computational resources than larger transformer-based architectures. These results demonstrate that lightweight transformer models can provide an effective balance between predictive performance and computational efficiency, making them attractive for practical software engineering applications. Despite these promising results, several limitations remain. The evaluation was conducted on a single benchmark dataset, which may limit the generalizability of the findings. Furthermore, DistilBERT was originally designed for natural language processing and therefore lacks the code-specific knowledge available in models pre-trained on large-scale source code repositories. In addition, handling highly complex code structures and long code sequences remains challenging for transformer-based approaches that rely primarily on token-level representations. Future work will focus on evaluating the proposed approach across multiple datasets and programming languages, exploring parameter-efficient fine-tuning (PEFT) techniques, and incorporating additional program representations such as abstract syntax trees, control-flow graphs, and data-flow information. Further investigation of code-

aware pretraining strategies may also help improve predictive performance and generalization capability.

Acknowledgments. The authors gratefully acknowledge the financial assistance from the Institute of Information Science, Faculty of Mechanical Engineering and Informatics, University of Miskolc.

Disclosure of Interests. The authors declared that they have no competing interests in this work. We declare that we do not have any commercial or associative interest that represents a conflict of interest in connection with the work submitted.

References

1. Cheng, K. and Takada, S., 2023. Software Defect Prediction based on JavaBERT and CNN-BiLSTM.
2. Uddin, M.N., Li, B., Ali, Z. et al. Software defect prediction employing BiLSTM and BERT-based semantic feature. Soft Comput 26, 7877–7891 (2022). https://doi.org/10.1007/s00500-022-06830-5
3. Jieke Shi, Zhou Yang, Bowen Xu, Hong Jin Kang, and David Lo. 2023. Compressing Pre-trained Models of Code into 3 MB. In Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22). Association for Computing Machinery, New York, NY, USA, Article 24, 1–12. https://doi.org/10.1145/3551349.3556964
4. Briciu, A., Czibula, G. and Lupea, M., 2023. A study on the relevance of semantic features extracted using BERT-based language models for enhancing the performance of software defect classifiers. Procedia Computer Science, 225, pp.1601-1610. https://doi.org/10.1016/j.procs.2023.10.149
5. Min, B., Ross, H., Sulem, E., Veyseh, A.P.B., Nguyen, T.H., Sainz, O., Agirre, E., Heintz, I. and Roth, D., 2023. Recent advances in natural language processing via large pre-trained language models: A survey. ACM Computing Surveys, 56(2), pp.1-40. https://doi.org/10.48550/arXiv.2111.01243
6. Risse, N., & Böhme, M. (2024). Uncovering the limits of machine learning for automatic vulnerability detection. In 33rd USENIX security symposium (USENIX Security 24) (pp. 4247-4264). https://doi.org/10.48550/arXiv.2306.17193
7. Gomes, L., da Silva Torres, R. and Côrtes, M.L., 2023. BERT-and TF-IDF-based feature extraction for long-lived bug prediction in FLOSS: A comparative study. Information and Software Technology, 160, p.107217. https://doi.org/10.1016/j.infsof.2023.107217
8. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. and Zhou, M., 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155. https://doi.org/10.48550/arXiv.2002.08155
9. Kanade, A., Maniatis, P., Balakrishnan, G. and Shi, K., 2020, November. Learning and evaluating contextual embedding of source code. In International conference on machine learning (pp. 5110-5121). PMLR. https://doi.org/10.48550/arXiv.2001.00059
10. Pan C, Lu M, Xu B. An Empirical Study on Software Defect Prediction Using CodeBERT Model. Applied Sciences. 2021; 11(11):4793. https://doi.org/10.3390/app11114793
11. A. H. Mohammadkhani, C. Tantithamthavorn and H. Hemmatif, "Explaining Transformer-based Code Models: What Do They Learn? When They Do Not Work?," 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM), Bogotá, Colombia, 2023, pp. 96-106, doi: 10.1109/SCAM59687.2023.00020.

12. X. Zhou, D. Han and D. Lo, "Assessing Generalizability of CodeBERT," 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 2021, pp. 425-436, doi: 10.1109/ICSME52107.2021.00044.
13. Yang, C., Xu, B., Khan, J.Y., Uddin, G., Han, D., Yang, Z. and Lo, D., 2022, March. Aspect-based api review classification: How far can pre-trained transformer model go?. In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 385-395). IEEE. https://doi.org/10.48550/arXiv.2201.11327
14. P. Ardimento and C. Mele, "Using BERT to Predict Bug-Fixing Time," 2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS), Bari, Italy, 2020, pp. 1-7, doi: 10.1109/EAIS48028.2020.9122781.
15. Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., and Li, G., 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664. https://doi.org/10.48550/arXiv.2102.04664
16. Zhou, Y., Liu, S., Siow, J., Du, X. and Liu, Y., 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in neural information processing systems, 32. https://doi.org/10.48550/arXiv.1909.03496
17. N. A. A. Khleel and K. Nehéz, "A novel approach for software defect prediction using CNN and GRU based on SMOTE Tomek method," Journal of Intelligent Information Systems, vol. 60, no. 3, pp. 673–707, Jun. 2023, doi: 10.1007/s10844-023-00793-1.
18. N.A.A.Khleel, K.Nehéz, "Software defect prediction using a bidirectional LSTM network combined with oversampling techniques", Cluster Comput (2023). https://doi.org/10.1007/s10586-023-04170-z.
19. Zou, W., Li, Q., Ge, J., Li, C., Shen, X., Huang, L. and Luo, B., 2023. A Comprehensive Evaluation of Parameter-Efficient Fine-Tuning on Software Engineering Tasks. arXiv preprint arXiv:2312.15614. https://doi.org/10.48550/arXiv.2312.15614
20. Guo, Y., Gao, X., Zhang, Z., Chan, W.K. and Jiang, B., 2023, October. A study on the impact of pre-trained model on Just-In-Time defect prediction. In 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS) (pp. 105-116). IEEE. https://doi.org/10.48550/arXiv.2309.02317

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

