Proceedings of the 2nd International Conference on Software Engineering and Machine Learning
DOI: 10.54254/2755-2721/73/20240357

BERT-based cross-project and cross-version software defect prediction

Binwen Sun

The Hong Kong Polytechnic University, 11 Yuk Choi Road, Hung Hom, Kowloon, Hong Kong

binwen.sun@connect.polyu.hk

Abstract. In recent years, deep learning-based software defect prediction has gained significant attention in software engineering research. This study aims to explore the application of the BERT model in the field of software defect detection. Traditional methods are constrained by manually designed rules and expert knowledge, which leads to limited accuracy and generalization ability. The strengths of deep learning methods lie in their capacity to capture complex semantic and contextual information in code. However, the effectiveness of deep learning models is hindered by the small scale of software defect datasets. To address this issue, we introduce BERT as a pre-trained model and construct a downstream task neural network, comprising a single-layer fully connected layer and a softmax classifier. Additionally, we evaluate four variants of BERT to enhance predictive performance. Through empirical studies on software defect prediction across different versions and projects, we find that utilizing the BERT pre-trained model significantly enhances predictive performance. The experimental results demonstrate that our model outperforms TextCNN by 8.99% in terms of AUC score and LSTM by 5.66%. In terms of the F1 score, our model surpasses TextCNN by 4.51% and LSTM by 15.57%. The primary contribution of this paper is the proposal of a cross-version and cross-project software defect prediction method, leveraging a lightweight BERT-based neural network. We also discuss the reasons for the observed variations in the performance of the four BERT variants during testing.

Keywords: software defect prediction, BERT model, deep learning.

1. Introduction

Software defect detection involves the systematic analysis and examination of software systems or program code using methods and tools to uncover vulnerabilities, with the goal of enhancing software quality, reliability, and security [1]. Automated software defect detection, utilizing machine learning or deep learning techniques, efficiently identifies and addresses defects, thus improving software quality and economizing time and costs. This renders it a pivotal and integral step in contemporary software development [2-3].

In the domain of software defect detection, conventional methods predominantly rely on manually devised rules [4-6], heuristic algorithms [7], and machine learning algorithms. The classical processing workflow entails establishing a software defect dataset and designing numerous code metrics, followed by the utilization of logistic regression, K-Nearest Neighbours Classifier [8], support vector machines

DOI: 10.54254/2755-2721/73/20240357

[9], or tree-based ensemble learning methods to construct and train software defect detection models. These methods typically necessitate substantial human effort and time, and their accuracy and generalization ability are confined by the expertise of the practitioners. Traditional machine learning algorithms struggle to encapsulate the intricate semantics and contextual information within the code, resulting in limited accuracy and generalization ability when confronted with extensive software projects. Furthermore, they mandate continuous adjustments and optimizations when confronted with domain changes and emerging software technologies.

The introduction of deep learning into software defect detection tasks has ushered in notable progress and breakthroughs in this domain. With the advancement of deep learning, neural networks have emerged as a critical component, especially in Natural Language Processing (NLP) tasks, where deep learning methods have showcased exceptional performance. In comparison to traditional methods, deep learning confers several advantages, with the most pivotal being its capability to grasp intricate semantics and contextual information within the code, thereby bolstering accuracy and generalization capability. During the evolution of deep learning, several noteworthy NLP models have surfaced, including TextCNN [10], RNN, and LSTM [11]. These models have achieved significant milestones in text processing. For instance, TextCNN can capture local features through convolutional operations, RNN is adept at processing sequence data, and LSTM addresses the issue of vanishing gradients in conventional RNNs, enabling the model to comprehend contextual relationships more effectively.

In recent years, pre-trained models [12-13] have flourished across diverse domains such as natural language processing and computer vision. The advent of novel models and enhanced training strategies has led to substantial advancements in pre-trained models across various tasks. Exemplary instances include Transformer and BERT (Bidirectional Encoder Representations from Transformers) [14], which leverage extensive unlabeled text data for pre-training to acquire universal semantic representations. In the realm of software defect detection, the software defect datasets are frequently limited in scale, which curtails the detection performance of deep learning models and obstructs the complete exploitation of intricate semantic and contextual information within the code. To surmount this challenge, the introduction of pre-trained models has emerged as a potent solution. Through pre-training on substantial general corpora, pre-trained models can amass rich universal semantic knowledge and achieve an improved understanding of the import and structure of the code. Subsequently, during the fine-tuning phase, the pre-trained model is adapted to the software defect detection task. In comparison to conventional methods founded on manual feature engineering and neural networks relying solely on TextCNN or LSTM, this approach not only heightens model performance but also diminishes the necessity for copious annotated data, augmenting the model's generalization ability, semantic comprehension, and contextual modeling capabilities. Consequently, it becomes better suited for addressing the formidable task of software defect detection.

This study aims to address two principal challenges within individual software defect datasets: the restricted data volume and the feeble generalization ability of software defect prediction models trained for specific projects and versions. To surmount these challenges, we harness a cross-project and cross-version dataset to train the software defect detection model. Traditional data augmentation techniques, such as replacement or deletion, generate synthetic data rather than authentic data. We require a cross-project and cross-version universal software defect prediction model capable of expanding the dataset's dimensions while preserving a substantial quantum of original information. This approach enables us to counteract the dataset’s limitations and obtain more precise and dependable prediction models more effectively. Additionally, we delve into the influence of case sensitivity on the model's classification performance. Code texts often adhere to the camel-case convention for naming variables and functions, resulting in scenarios featuring simultaneous occurrences of uppercase and lowercase letters. Moreover, the programming language for the software defect dataset employed in this article is Java, a case-sensitive language. In this paper, we explore a software defect prediction model grounded in the BERT model. Concretely, BERT functions as the pre-trained model, and we forge a downstream task neural network encompassing a single-layer fully connected layer and a softmax classifier. To assess the model, we execute experiments involving four BERT variants.

DOI: 10.54254/2755-2721/73/20240357

The contributions of this paper encompass:

• The evaluation of the BERT-based software defect detection method's generalization performance across datasets from distinct projects.

• A comprehensive discussion of the factors underpinning the disparities noted in the performance of the four iterations of the lightweight BERT-based software defect detection model during testing.

2. Related Works

The exploration of related works in software defect prediction often centers around widely used public datasets, with NASA [15] and PROMISE [16] datasets emerging as prominent choices. The NASA dataset serves as a repository of publicly available software defect data, encompassing code metrics and defect labels for 14 distinct software projects. These datasets have been extensively employed in software defect prediction research to assess and contrast the effectiveness of diverse machine learning methodologies in categorizing software modules as either defect-prone or non-defect-prone. Nevertheless, Shepperd et al. highlighted certain issues related to data quality within the NASA dataset, including variations between different versions, irrational values, absent values, conflicting values, and duplicated values. These concerns have the potential to impede the credibility and comparability of empirical analyses reliant on the NASA dataset. Thus, the authors recommend that researchers delineate data sources, elucidate preprocessing procedures, and cultivate a comprehensive understanding of the data prior to applying machine learning techniques. Regarding the PROMISE dataset, Watanabe et al. elucidate, in their paper titled "Towards identifying software project clusters with regard to defect prediction," a freshly compiled dataset of software projects, encompassing 92 versions across 38 proprietary, open-source, and academic projects [1]. Recognized as the PROMISE dataset, this publicly accessible software engineering data repository is designed to facilitate the replication and validation of software engineering research. The authors conducted cluster analysis on this dataset to unveil clusters of software projects sharing analogous traits from the perspective of defect prediction [2]. They harnessed various clustering methodologies, including hierarchical clustering, k-means clustering, and Kohonen neural networks, and validated the clustering outcomes through discriminant analysis and statistical tests [3]. They identified two prevailing clusters: proprietary cluster B and proprietary/open-source cluster, subsequently devising defect prediction models for each cluster [4]. This research not only provides a blueprint for reutilizing defect prediction models but also furnishes a valuable dataset for the software engineering community.

At present, software defect detection is underpinned by two primary technological approaches: traditional machine learning-based methods and the rapidly evolving neural network-based approaches.

Traditional software defect methods, rooted in machine learning, necessitate manual feature engineering, often involving code metrics. Moreover, numerous studies have delineated statistically derived code features for software defect detection tasks. In the domain of traditional machine learning-based software defect prediction, prominent classifiers encompass logistic regression, decision trees, Naive Bayes, Support Vector Machines (SVM), K-Nearest Neighbors (KNN), K-Means, and Random Forest. Notably, the logistic regression classifier serves as a benchmark model. Noteworthy advancements arise from Random Forest and other tree-based classifiers within the traditional machine learning realm. Zhou et al. [17] introduced a software defect prediction model rooted in deep forest, which transforms random forest classifiers into a layered structure to discern more pivotal defect features.

However, as software systems burgeon in complexity and scale, conventional detection methods might encounter limitations. Additionally, for domain-specific and application-specific defect detection, a profound comprehension of the domain's attributes and requisites is indispensable for devising appropriate manual features and detection strategies. Thus, in response to these challenges, a novel approach has materialized in recent years, entailing the harnessing of neural networks (Deep Learning) to automatically glean code features from datasets, thereby enabling automated code defect detection. Pan et al. [18] formulated a software defect detection model built upon CodeBERT [19] and undertook empirical investigations in both cross-project and within-project scenarios. Uddin et al. [20] presented

DOI: 10.54254/2755-2721/73/20240357

a hybrid software defect prediction model hinged on BERT and BiLSTM, amplifying model performance through data augmentation techniques.

3. Method

The methodology employed in our study is depicted in the figure below:

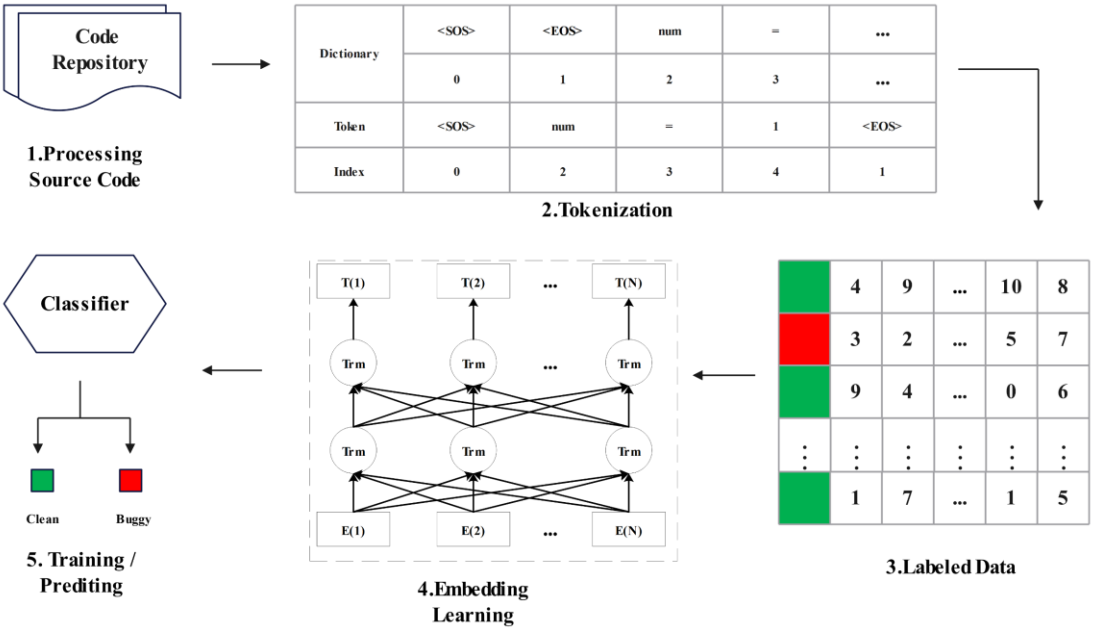

Figure 1. The workflow of our model

In our investigation, we initially delve into version control repositories to extract essential files for the dataset. These files are subsequently categorized as either containing errors or being error-free. Subsequently, we extract vocabulary files from the software defect dataset and preprocess the data by encoding and tokenizing it, rendering it compatible with the BERT model. Next, the preprocessed data is fed into the BERT model provided by the HuggingFace community. Lastly, we construct a downstream task neural network based on BERT and proceed with training the network for predictive purposes.

Traditional software defect detection methods have not fully harnessed the semantic information present in the source code, thereby curbing the classification performance of software defect detection models. Moreover, the constrained scale of existing defect datasets has somewhat impeded the advancement of classification performance for extant models, exemplified by the limited size of the PROMISE dataset. To address this challenge, methods like code metric-based statistical learning or abstract syntax tree-based approaches frequently overlook specific function and variable names. To surmount this issue, we have embraced the standard BERT (Bidirectional Encoder Representations from Transformers) model. BERT constitutes a pre-trained natural language processing model widely deployed across various text processing tasks.

The advent of the "pre-training" technique stems from the practical scenario of sparse annotated resources juxtaposed with abundant unlabeled resources. In certain specific tasks, a meager quantity of relevant training data poses a challenge for the model to glean meaningful patterns. Consequently, we endeavor to leverage extensive pre-trained models on datasets replete with ample data to amplify model

DOI: 10.54254/2755-2721/73/20240357

performance on smaller datasets. This approach can be conceptualized as bifurcating the training process into two stages: foundational learning and task-specific learning, corresponding to the pre-training and fine-tuning phases. BERT's pre-training phase encompasses two tasks: Masked Language Model (MLM) and Next Sentence Prediction (NSP). The MLM task simulates a "fill in the blanks" exercise, wherein a word is concealed, and its context is employed to predict the concealed term. The NSP task endeavors to ascertain whether two given sentences are contiguous within the original text. Post pre-training, BERT can be fine-tuned for sundry downstream tasks such as text classification, sequence labeling, and reading comprehension, culminating in superior experimental outcomes.

In our research, in accordance with the BERT model's specifications, we adopted a subword tokenization strategy based on WordPiece to tokenize each statement in the source code into discrete words or symbols. This process is acknowledged as BERT's tokenization process. Initially, we compiled a dataset vocabulary file encompassing all potential tokens. Subsequently, we preprocessed the source code text by eliminating comments and newline characters, adding the "<SOS>" tag at the commencement and the "<EOS>" tag at the culmination of the text. Thereafter, predicated on the pre-established vocabulary file, we embraced a greedy longest-match-first algorithm to fragment less frequent words into word pieces comprised of more prevalent subwords. This operation is imperative due to code variables frequently adhering to camel case naming conventions, leading to the segmentation of a term like "studentId" into "student" and "id."

If a term within the code text eludes identification within the vocabulary, it is substituted with "<UNK>" (indicating an unknown token). Through this process of word piece tokenization, the source code text undergoes transformation into an integer list, with each integer signifying the ID of a token. This list of integers can be employed as input for the BERT model.

The downstream task models encompass further computations on the extracted features from BERT to derive the computational outcomes requisite for the binary software defect detection task expounded in this paper. Within this study, two distinct downstream task models have been delineated to accommodate diverse iterations of the BERT model. Our model adopts a fully connected neural network structure with weight matrices of dimensions 768x2 and 1024x2. The classifier employs the softmax regression technique. Within softmax regression, the model gauges the likelihood of the code text containing defects. If the probability surpasses 0.5, the text is classified as "buggy"; contrarily, it is deemed "clean."

4. Experiment

4.1. A. Experiment Dataset

For our experiments, we employed 6 projects from the PROMISE dataset. The table below presents dataset information, encompassing Project Name, Description, File Numbers, Bug File Numbers, and Bug Rate.

Table 1. Dataset description

Project Name	File Numbers	Bug File Numbers	Bug Rate	Description
camel-1.6	965	188	19.5%	A framework of enterprise integration
lucene-2.0	195	91	46.7%	An engine library for searching text
lucene-2.2	247	144	58.3%	
synapse-1.0	157	16	10.2%	Adapters for transporting data
synapse-1.1	222	60	27.0%	
synapse-1.2	256	86	33.6%	

DOI: 10.54254/2755-2721/73/20240357

The table below shows the number of training instances and testing instances we have:

Table 2. Statistics of training set and test set

	Training Instances	Test Instances
Total	1705	300

4.2. B. BERT

We evaluated four slightly distinct BERT models, all provided by Hugging Face: bert-base-uncased, bert-base-cased, bert-large-uncased, and bert-large-cased. The designations "base" and "large" correspond to the model's parameter size. "Cased" denotes that the model distinguishes between uppercase and lowercase letters in English, while "uncased" implies that case distinctions are disregarded. The bert-base model features 12 stacked transformer blocks, 768 hidden units, 12 attention heads, and a total of 110 million parameters. Conversely, the bert-large model encompasses 24 stacked transformer blocks, 1024 hidden units, 16 attention heads, and a total of 340 million parameters.

4.3. C. Evaluation Metrics

We employed four evaluation metrics to gauge the model's performance: AUC, F1 score, Precision, and Recall. These metrics are standard in software defect detection.

The F1 score, a merger of precision and recall, serves as a pivotal metric for evaluating classification models. The F1 score, often represented as the harmonic mean of precision and recall, is calculated using the formula:

$$F1 = \frac{2^* \text{ precision } * \text{ recall}}{\text{precision} + \text{recall}}$$

The AUC metric (Area Under the Curve) comprehensively assesses the predictive accuracy of classification models. It quantifies performance by calculating the area beneath the Receiver Operating Characteristic (ROC) curve. The AUC value ranges from 0 to 1, with a higher value indicating superior performance. An AUC of 1 signifies perfect prediction, while an AUC of 0.5 suggests performance equivalent to random guessing.

Precision gauges the proportion of true positive samples among instances predicted as positive by the model. It's calculated as:

$$\text{Precision} = \frac{\text{TruePositives}}{\text{TruePositives+FalsePositives}}$$

Recall, also known as True Positive Rate or Sensitivity, quantifies the model's capacity to predict positive instances. It's calculated using:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives+False Negatives}}$$

5. Baseline Model

To benchmark our method's effectiveness against other defect prediction approaches, we compared our classifier with two baseline classifiers: TextCNN and LSTM.

5.1. TextCNN

TextCNN employs convolutional filters of varying window sizes on input text sequences to capture local features and generate fixed-size feature maps. These maps are then subjected to max-pooling layers to extract important features, followed by predictions using fully connected layers and a softmax classifier, determining the class label for the input text.

5.2. LSTM

LSTM (Long Short-Term Memory) is a recurrent neural network model well-suited for handling sequential data with long-term dependencies. By utilizing gating mechanisms like input gates, forget

DOI: 10.54254/2755-2721/73/20240357

gates, and output gates, LSTM effectively captures long-range dependencies in sequences while mitigating vanishing and exploding gradient problems. LSTM has exhibited remarkable achievements in natural language processing tasks, including text classification, language modeling, and machine translation.

6. Results Analysis

Table 3. Performance comparisons with base models

	AUC	Precision	Recall	F1
TextCNN	0.6267	0.6000	0.7600	0.6706
LSTM	0.6600	0.7926	0.4333	0.5603
BERT-L-U	**0.7166**	0.6946	**0.7733**	**0.7157**
BERT-L-C	0.6733	0.6494	0.7533	0.6712
BERT-B-U	0.6800	**0.7934**	0.4866	0.6675
BERT-B-C	0.6566	0.7764	0.4400	0.6397

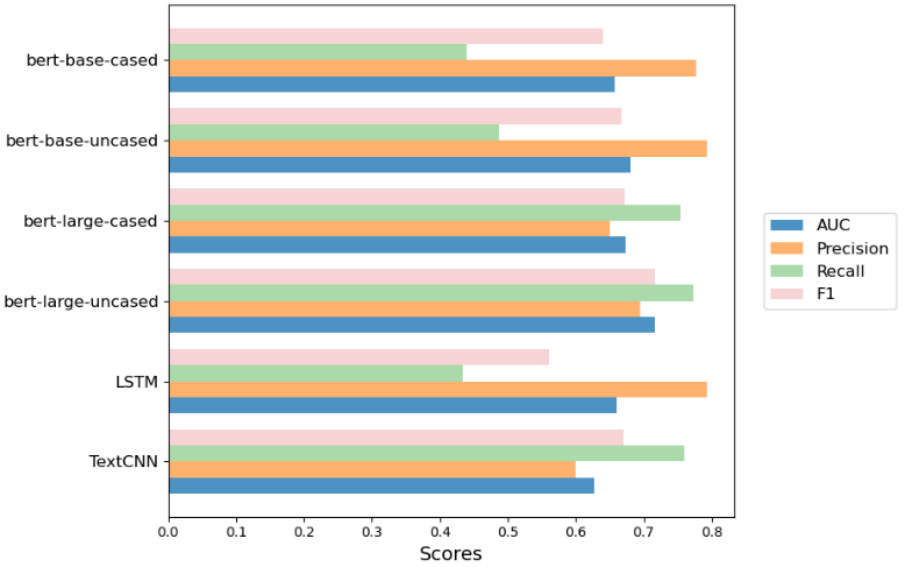

Figure 2. Comparisons with base models

To enhance clarity in the table, the following abbreviations are employed to represent the model names:

blu-sdp: Software Defect Prediction Model based on bert-large-uncased

blc-sdp: Software Defect Prediction Model based on bert-large-cased

bbu-sdp: Software Defect Prediction Model based on bert-base-uncased

bbc-sdp: Software Defect Prediction Model based on bert-base-cased

(1) The BERT-based approach outperforms TextCNN and LSTM across all evaluation metrics. The results demonstrate that our model surpasses TextCNN by 8.99% in AUC and LSTM by 5.66%, and in terms of F1 score, it outperforms TextCNN by 4.51% and LSTM by 15.57%. This underscores the efficacy of introducing pre-trained models for software defect detection datasets.

(2) Among the BERT variants, the software defect classifier based on bert-large-uncased exhibits exceptional performance, surpassing other BERT model variations. It achieves the highest values in AUC, Recall, and F1 metrics. This finding suggests that, in software defect detection tasks, pre-trained models with more parameters and case insensitivity deliver outstanding results. This is attributed to the predominant lowercase usage in code text, despite Java's case sensitivity.

DOI: 10.54254/2755-2721/73/20240357

7. Conclusion and Future Work

This paper introduces a BERT-based approach for software defect detection, employing a pre-trained BERT model alongside fully connected layers and a softmax classifier to harness code semantics and bolster classification performance. Through a series of experiments, we compared different iterations of BERT models, with the method built upon bert-large-uncased demonstrating optimal results. Furthermore, our discussion delves into the implications of model parameters and case sensitivity on classification efficacy.

To advance predictive capabilities, our future research endeavors encompass broadening the software defect prediction dataset and utilizing this data to train a dedicated pre-trained model specifically tailored for software defect detection. We intend to explore whether customized pre-trained models, in tandem with more intricate downstream task neural networks, can further amplify model performance. Concurrently, we will closely monitor the latest advancements in real-time software defect prediction, seeking opportunities to integrate cutting-edge developments into our methodology.

References

[1] X. Sun, W. Zhou, B. Li, Z. Ni, and J. Lu, "Bug Localization for Version Issues With Defect Patterns," IEEE Access, vol. 7, pp. 18811–18820, 2019, doi: 10.1109/ACCESS.2019.2894976.

[2] Association for Computing Machinery, PROMISE : 7th International Conference on Predictive Models in Software Engineering : Banff, Canada, Sept 20-21, 2011 : co-located with ESEM 2011.

[3] J. (Jean) Bézivin, J.-Marie. Favre, Bernhard. Rumpe, Association for Computing Machinery., and ACM Sigsoft., GaMMa '06: : proceedings of the 2006 International Workshop on Global Integrated Model Management. Association for Computing Machinery, 2006.

[4] T. J. Mccabe, "A Complexi-ty Measure," 1976.

[5] R. Harrison, S. J. Counsell, and R. V Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," 1998.

[6] M. Jureczko and D. D. Spinellis, "Using Object-Oriented Design Metrics to Predict Software Defects 1*." [Online]. Available: http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm

[7] M. A. Elsabagh, M. S. Farhan, and M. G. Gafar, "Cross-projects software defect prediction using spotted hyena optimizer algorithm," SN Appl Sci, vol. 2, no. 4, Apr. 2020, doi: 10.1007/s42452-020-2320-4.

[8] A. B. Nasser et al., "A Robust Tuned K-Nearest Neighbours Classifier for Software Defect Prediction Lower Limb Exoskeleton View project Artificial Image processing based decision making for grading and sorting of rotatiinally symmetric products View project A Robust Tuned K-Nearest Neighbours Classifier for Software Defect Prediction." [Online]. Available: https://www.researchgate.net/publication/362850366

[9] H. A. Alhija, M. Azzeh, and F. Almasalha, "Software Defect Prediction Using Support Vector Machine."

[10] Y. Kim, "Convolutional Neural Networks for Sentence Classification," Aug. 2014, [Online]. Available: http://arxiv.org/abs/1408.5882

[11] "LSTM".

[12] T. B. Brown et al., "Language Models are Few-Shot Learners." [Online]. Available: https://commoncrawl.org/the-data/

[13] A. Vaswani et al., "Attention Is All You Need."

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," Oct. 2018, [Online]. Available: http://arxiv.org/abs/1810.04805

[15] M. Shepperd, Q. Song, Z. Sun, and C. Mair, "Data quality: Some comments on the NASA software defect datasets," IEEE Transactions on Software Engineering, vol. 39, no. 9, pp. 1208–1215, 2013, doi: 10.1109/TSE.2013.11.

DOI: 10.54254/2755-2721/73/20240357

[16] Gunes. Koru and Association for Computing Machinery., PROMISE 2010 : 6th International Conference on Predictive Models in Software Engineering : Timișoara, Romania, September 12-13, 2010. ACM, 2010.
[17] T. Zhou, X. Sun, X. Xia, B. Li, and X. Chen, "Improving defect prediction with deep forest," Inf Softw Technol, vol. 114, pp. 204–216, Oct. 2019, doi: 10.1016/j.infsof.2019.07.003.
[18] C. Pan, M. Lu, and B. Xu, "An empirical study on software defect prediction using codebert model," Applied Sciences (Switzerland), vol. 11, no. 11, Jun. 2021, doi: 10.3390/app11114793.
[19] Z. Feng et al., "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Feb. 2020, [Online]. Available: http://arxiv.org/abs/2002.08155
[20] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan, and I. Zada, "Software defect prediction employing BiLSTM and BERT-based semantic feature," Soft comput, vol. 26, no. 16, pp. 7877–7891, Aug. 2022, doi: 10.1007/s00500-022-06830-5.