Software Defect Prediction based on JavaBERT and CNN-BiLSTM

Kun Cheng[1], Shingo Takada[2]

[1]Grad. School of Science and Technology, Keio University Yokohama, Japan
[2]Grad. School of Science and Technology, Keio University Yokohama, Japan

Abstract
Software defects can lead to severe issues in software systems, such as software errors, security vulnerabilities, and decreased software performance. Early prediction of software defects can prevent these problems, reduce development costs, and enhance system reliability. However, existing methods often focus on manually crafted code features and overlook the rich semantic and contextual information in program code. In this paper, we propose a novel approach that integrates JavaBERT-based embeddings with a CNN-BiLSTM model for software defect prediction. Our model considers code context and captures code patterns and dependencies throughout the code, thereby improving prediction performance. We incorporate Optuna to find optimal hyperparameters. We conducted experiments on the PROMISE dataset, which demonstrated that our approach outperforms baseline models, particularly in leveraging code semantics to enhance defect prediction performance.

Keywords
Software defect prediction, JavaBERT, CNN, BiLSTM, Optuna,

1. Introduction

Software defects present significant challenges to the reliability and performance of software systems, often leading to critical issues such as slow software operation, frequent security vulnerabilities, and software crashes. To address these challenges, researchers have turned their attention to software defect prediction (SDP), a key research area aimed at identifying potentially problematic code early in the development process.

Software Defect Prediction (SDP) is a structured process involving data preprocessing, feature extraction, model building, and evaluation[1]. Feature extraction plays a pivotal role in SDP as it determines the model's data representation. SDP methods have traditionally relied on manual feature engineering, a process involving time-consuming and laborious manual design. However, this approach faces challenges in capturing complex semantics and contextual information embedded in software code as systems become more complex. As a result, there's a growing demand for advanced techniques that can effectively exploit the intrinsic semantic and structural meaning of code, along with its statistical properties.

Recent advances in SDP have shifted towards leveraging structural and semantic features directly from source code or through parsing into an abstract syntax tree (AST)[2]. These modern methods employ these features in combination with various classification methods, encompassing both traditional algorithms and deep learning techniques.

SDP encompasses two primary domains: Cross-Project Defect Prediction (CPDP) and Within-Project Defect Prediction (WPDP). CPDP involves training a model on one project and applying it to another, addressing the challenge of generalization across different software environments. In contrast, WPDP focuses on building models within the same project, enhancing defect prediction performance by considering unique project characteristics and evolution patterns. For the purpose of this study, our primary focus lies on WPDP, aiming to improve defect prediction performance within a single project.

In this paper, we introduce an innovative approach to SDP that combines Java Bidirectional Encoder Representations of Transformers (JavaBERT) and Convolutional Neural Networks with Bidirectional Long Short-Term Memory (CNN-BiLSTM). By harnessing JavaBERT's contextual understanding of text data and CNN-BiLSTM's capacity to capture structural features, we improve defect prediction performance. Furthermore, we optimize the model's hyperparameters by introducing Optuna, further refining our predictive model.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 presents the design of our proposed approach. Section 4 covers the implementation details based on the design, and Section 5 offers the evaluation results along with a discussion of potential threats to validity. Finally, Section 6 concludes the paper and discusses future work.

QuASoQ 2023: 11th International Workshop on Quantitative Approaches to Software Quality, December 04, 2023, Seoul, South Korea
chengkun@keio.jp (K. Cheng); michigan@ics.keio.ac.jp (S. Takada)

2. Related Work

Researchers have explored various models for feature extraction in software defect prediction, from traditional machine learning to deep learning. Initially, Support Vector Machines (SVM), as employed by Elish et al.[3], gained prominence for identifying defective modules using static code metrics. However, it struggled to uncover deep semantics within the source code. Deep Belief Networks (DBN), introduced by Wang et al.[4], aimed to extract more complex features from code through unsupervised learning. Yet, its limited depth posed challenges in revealing intricate relationships within the source code. Convolutional Neural Networks (CNNs) were used by Li et al.[5] to predict software defects by analyzing structural correlations between code tokens. While proficient in capturing local patterns, CNNs faced challenges in capturing longer-range connections. Wang et al.[6] introduced an RNN (Recurrent Neural Network)-based model for predicting software reliability. Deng et al.[7] and Liang et al.[8] expanded Long Short-Term Memory (LSTM) models in software defect prediction, capturing temporal patterns in code sequences. However, a single LSTM can only capture one direction temporal pattern in the code sequence. Bidirectional LSTM (BiLSTM) models with attention mechanisms emerged. Wang et al.[9] introduced a gated hierarchical BiLSTM model. Uddin et al.[10] combined BiLSTM with attention and BERT-based embeddings.

In short, SVM has difficulty discovering the deep semantics of the source code, DBN has limited depth so it is difficult to understand the complex relationships in the source code, CNN has difficulty capturing long-distance correlations, and RNN and LSTM can only capture a single temporal pattern. BiLSTM may have challenges in capturing local patterns.

To solve these problems, we combine the advantages of CNN in detecting local patterns with the advantages of BiLSTM in processing sequences, allowing for comprehensive code inspection. We further incorporate JavaBERT to dynamically adjust token embeddings based on the entire input sequence, thereby deepening the representation and capturing interdependencies among code tokens.

3. Proposed Methodology

Our software defect prediction method consists of several key steps, all aimed at improving prediction performance. As shown in Figure 1, we first use JavaBERT to convert the code into vector representations. Next, we employ the CNN-BiLSTM model for feature extraction, focusing on local patterns and context. We also incorporate statistical features to fully utilize all available information. Optuna automatically executes the above combination of JavaBERT and CNN-BiLSTM multiple times, and outputs the best hyperparameter values through these executions. Then we retrain the model in another version of the code based on the obtained hyperparameters and test the model performance.

3.1. Embedding with JavaBERT

BERT (Bidirectional Encoder Representations from Transformers)[11] is a language model widely employed in natural language processing (NLP) tasks. Unlike conventional embeddings, BERT excels at capturing intricate contextual associations. Traditional methods like Word2Vec[12] and GloVe[13] generate static contextual representations, whereas BERT, utilizing multi-layer bidirectional transformers, enables tokens to gather information from both preceding and succeeding tokens.

In our approach, we leverage a pretrained BERT model, JavaBERT[14], fine-tuned for Java code. JavaBERT has been trained on a dataset of 2,998,345 Java files from GitHub open source projects. JavaBERT's transformer architecture dynamically adapts token embeddings based on the entire input sequence, enhancing representation depth and capturing code token interdependencies. The JavaBERT embeddings, denoted as E_{JavaBERT}, are computed by applying the model's encoder to tokenized Java code. For a sequence of code tokens $C = \{c_1, c_2, \ldots, c_n\}$, JavaBERT embeddings are computed as:

$$E_{\text{JavaBERT}} = \text{Encoder}_{\text{JavaBERT}}(c_1, c_2, \ldots, c_n)$$

Models typically cannot process code text sequences directly. Through JavaBERT, we embed code text into a continuous vector space, using these vectors as inputs to the model, making it easier for the model to compute and understand the code.

3.2. Feature Extraction using CNN-BiLSTM

We combine Convolutional Neural Networks (CNN) and Bidirectional Long Short-Term Memory networks (BiLSTM) to extract features. This is the key part of our approach, where after extracting features with CNN, it is refined with the sequential capabilities of BiLSTM.

3.2.1. Feature Extraction with CNN

Utilizing Convolutional Neural Networks (CNN)[15] for feature extraction involves sliding a small window, known as a filter, over various parts of the code. This filter examines a small segment of the code at a time, calculating a value at each sliding position to create a "feature map."

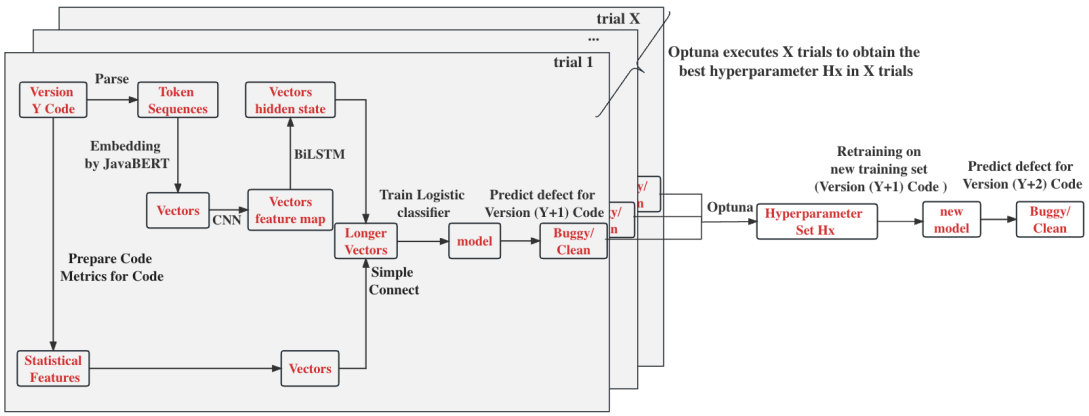

Figure 1: Overview of Methodology

The positions in the code correspond to positions in the feature map. The observed code segment within the filter's scope is termed the "input sequence slice." As the filter traverses the entire code, it analyzes these input sequence slices, effectively capturing distinct features that characterize the code's structural and syntactical elements.

The process of feature extraction using CNN is mathematically expressed as:

$$y[i,j] = \sigma\left(\sum_m \sum_n x[i+m, j+n] \cdot w[m,n] + b\right)$$

where $x[i,j]$ is the input at position (i,j), $w[m,n]$ represents the kernel at position (m,n), b is the bias, and σ signifies the activation function.

3.2.2. Refinement of Features with BiLSTM

The Bidirectional Long Short-Term Memory (BiLSTM)[16] layer enhances the features extracted by the Convolutional Neural Networks (CNN). What sets BiLSTM apart is its capability to capture both short-term and long-term dependencies within the code, which perfectly complements the local feature extraction carried out by CNN.

The forward and backward computations in BiLSTM can be unified into a single mathematical representation:

$$h_t = \text{BiLSTM}(x_t, h_{t-1}, h_{t+1})$$

In this equation, h_t represents the hidden state at time step t in the Bidirectional Long Short-Term Memory (BiLSTM) model. It is computed based on the input x_t at the current time step, the previous hidden state h_{t-1}, and the next time step $t+1$'s hidden state h_{t+1}. The BiLSTM model effectively captures sequential patterns and dependencies in data by considering information from both directions. It analyzes the sequence of tokens, capturing dependencies extending both backward and forward within the code. This dynamic construction of code features considers token order, revealing evolving patterns and connections over time, amplifying the feature representation. In summary, we refine the feature maps obtained from CNN using BiLSTM to achieve a comprehensive code representation. This fusion of capturing local patterns and accounting for temporal dependencies improves software defect prediction performance.

3.3. Integration with Statistical Features

Our methodology integrates the refined BiLSTM outputs with statistical features (such as shown in Table 2) extracted from dataset. This step concatenates the vectors obtained from the BiLSTM and the vectors of statistical features obtained from the dataset into longer vectors, making full use of the description information of the code.

3.4. Hyperparameter Optimization by Optuna

Optuna, a powerful hyperparameter optimization framework developed by Akiba et al.[17], plays a vital role in our approach by automating hyperparameter tuning for the CNN-BiLSTM model. There are similar frameworks such as Ray Tune, etc., but Optuna is more lightweight and easier to use. It employs the Tree-structured Parzen Estimator (TPE) algorithm to efficiently explore and exploit the hyperparameter space, enhancing the performance of our Software Defect Prediction task.

In this section, we will discuss a crucial step in our methodology: determining optimal hyperparameters by leveraging shared features among different versions of the same project. Usually, code with similar version numbers exhibits a high degree of similarity. By harnessing these inherent similarities, we attempt to find hyperparameters that can generalize across various versions, ultimately enhancing model performance.

Using the Ant project as an example, our aim is to demonstrate the transferability of hyperparameters obtained from training on one version (e.g., 1.5) to another (e.g., 1.6). This transferability is valid as both versions originate from the same project, sharing similar code structures and functionalities. This enables the hyperparameters obtained from one version to serve as a foundation for other versions within the same project, thereby solidifying our model configuration.

We start by selecting version pairs, using the Ant project as an illustration. Here, we designate version 1.5 for training and version 1.6 for testing. Next, we define the performance metric to optimize, such as the F1 score. Subsequently, Optuna conducts multiple experiments, traversing various hyperparameter combinations and evaluating their performance on the designated testing dataset. Through these iterative experimentation and evaluation stages, Optuna determines the hyperparameter set that maximizes the chosen performance metric.

This process can be represented as:

$$H_x = \text{Optuna } f(\text{Ant } 1.5, \text{Ant } 1.6)$$

Here, $f(\text{Ant } 1.5, \text{Ant } 1.6)$ embodies the objective function maximized during the hyperparameter optimization process, with Ant 1.5 as the training dataset and Ant 1.6 as the testing dataset. After obtaining optimal hyperparameters H_x through the Optuna process, we seamlessly transfer them across different project versions. H_x is applied to reconfigure the training and testing sets. For instance, in the Ant project, H_x is then used on different version pairs, such as training on Ant 1.6 with H_x and testing on Ant 1.7.

This operation optimizes hyperparameters across version pairs, contributing to enhanced model adaptability and performance in varying project iterations.

4. Experimental Setup

4.1. Research Questions

Our experiment addresses the following research questions (RQ) :

RQ1: How does the performance of our CNN-BiLSTM model compare against baseline models?

Table 1
Selected Projects in the PROMISE Java Dataset

Project	Versions (Buggy Rate)
Ant	1.5, 1.6, 1.7 (0.109, 0.263, 0.224)
Camel	1.2, 1.4, 1.6 (0.36, 0.171, 0.201)
JEdit	3.2, 4.0, 4.1 (0.346, 0.256, 0.263)
Lucene	2.0, 2.2, 2.4 (0.489, 0.611, 0.615)
Poi	2.0, 2.5, 3.0 (0.120, 0.654, 0.641)
Synapse	1.0, 1.1, 1.2 (0.102, 0.270, 0.336)
Xalan	2.4, 2.5, 2.6 (0.163, 0.509, 0.468)

RQ2: How does the performance of the proposed model vary across different software projects and within the different versions of each project in the PROMISE dataset?

RQ3: How do different hyperparameter settings impact the performance of the combined CNN-BiLSTM model in code defect prediction?

4.2. Dataset and Data Preprocessing

Our study uses the PROMISE[18] dataset, exclusively comprised of Java projects. This dataset spans various domains and project scales, providing project details like name, description, version, and bug rate. Table 1 shows an overview of the projects we use that are in the PROMISE Java Dataset. Since Optuna's process of finding hyperparameters takes a lot of time, we only selected a part of the projects in the PROMISE data set. Statistical features also play a vital role in code analysis, offering insights into code composition and behavior. To enhance our study, we carefully selected a subset of these features, as shown in Table 2.

To prepare the data for analysis, we conducted thorough data preprocessing. Using the "javalang"[19] Python library, we removed redundant code elements such as comments, white spaces, and unnecessary details. This process allowed us to extract essential token sequences, capturing the code's semantics. To address class imbalance in software defect prediction, we implemented random oversampling exclusively on the "Bug" class files. This deliberate strategy generated synthetic data instances, improving class distribution and mitigating potential bias towards the majority class.

4.3. Experimental Settings

For each project listed in Table 1, we selected the smallest two version numbers to serve as versions Y and Y+1 for Optuna's hyperparameter optimization process. The search space for the hyperparameters was specified as shown in Table 3. The number of trials for each project was set to 30. After completing these experiments, each project will produce a different set of hyperparameters

Table 2
Selected Statistical Features

Measure of Functional Abstraction (MFA)
Coupling Between Methods (CBM)
Data Access Metric (DAM)
Coupling Between Object classCA (CBO)
Lines Of Code (LOC)
Afferent Couplings (CA)
Number Of Children (NOC)
Lack of COhesion in Methods (LCOM)
Average Method Complexity (AMC)
Inheritance Coupling (IC)
Response For a Class (RFC)
Efferent Couplings (CE)
Measure Of Aggregation (MOA)
Weighted Methods per Class (WMC)
Depth of Inheritance Tree (DIT)
Lack of COhesion in Methods (LCOM3)
Cohesion Among Methods of class (CAM)
Number of Public Methods (NPM)

that allow the model to output the highest F1 score, and a model trained on these parameters using version Y. These hyperparameters were then applied to train new models on version Y+1 for each project. Then the model trained on version Y and the model trained on version Y+1 were evaluated against the code of version Y+2. We conducted each evaluation test three times and calculated the mean to obtain the experimental result.

Table 3
Search Space for Hyperparameters

Hyperparameter	Search Range
Number of Epochs	3 to 10
Batch Size	[16, 32, 64, 128]
Learning Rate	1×10^{-5} to 1×10^{-2} (Log-uniform)
Filter Sizes	[3, 5, 7, 9, 11]
Number of Filters	32 to 512
Hidden Units	[16, 32, 64, 128, 256, 512, 1024]

4.4. Baseline Models

We compare our proposed approach against the following baseline models:

- Support Vector Machine (SVM): SVM, a classic and widely adopted machine learning algorithm, excels in both linear and non-linear classification tasks and is known for its effectiveness in handling high-dimensional data.
- Convolutional Neural Network (CNN): CNNs excel at extracting hierarchical features from structured data, making them suitable for capturing local patterns in software defect prediction.
- Bidirectional Long Short-Term Memory (BiLSTM): BiLSTM enhances LSTM by considering bidirectional information flow, enabling it to capture both past and future contexts.

In assessing the predictive performance, this paper utilizes three widely accepted metrics: precision, recall, and the F1-score.

5. Results and Discussion

In this section, we present the results of our study and discuss their implications, addressing the research questions (RQ) that guide our investigation.

5.1. Impact of JavaBERT-based Embeddings with CNN-BiLSTM Model

To address RQ1, we assessed the performance of our model in comparison to baseline models. Table 4 presents a detailed performance comparison between our CNN-BiLSTM model and the baseline models concerning precision, recall, and F1-score. For instance, "ant_1.5_1.6" represents the experimental results obtained by using version 1.5 of Ant as the training dataset and version 1.6 as the test dataset. The results demonstrate a consistent outperformance of our model across all metrics. Figure 2 complements the table by providing a visual representation of the F1 scores, where the x-axis represents pairs of software versions used for training and testing (e.g., ant_1.5_1.6), and the y-axis represents the corresponding F1 values obtained during testing. This figure shows that the F1 of our model is higher than the base model most of the time.

5.2. Model Performance Variability Across PROMISE Projects and Versions

To address RQ2, Figure 3 presents the F1 scores of our model across different projects and their respective versions in the PROMISE dataset. In this figure, the x-axis represents pairs of software versions used for training and testing (e.g., ant_1.5_1.6), while the y-axis represents the corresponding F1 values obtained during testing. When we examined the model's performance across different projects and its various versions, we observed certain noteworthy patterns. Specifically, within the same project,

Table 4
Comparison of Experimental Results with Baseline Models

project	SVM			CNN			BiLSTM			CNN-BiLSTM		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
ant_1.5_1.6	0.5133	0.6304	0.5659	0.5526	0.4565	0.5000	0.5120	0.6957	0.5899	0.6364	0.6848	0.6597
ant_1.6_1.7	0.5849	0.5602	0.5723	0.5230	0.5482	0.5353	0.2486	0.5241	0.3372	0.5868	0.5904	0.5886
ant_1.5_1.7	0.4297	0.6807	0.5268	0.3653	0.7349	0.4880	0.4194	0.7048	0.5258	0.5531	0.5964	0.5739
camel_1.2_1.4	0.3645	0.5103	0.4253	0.5625	0.4966	0.5275	0.3186	0.4966	0.3881	0.4647	0.7724	0.5803
camel_1.4_1.6	0.2687	0.0957	0.1412	0.5392	0.2926	0.3793	0.2908	0.3883	0.3326	0.4957	0.3032	0.3762
camel_1.2_1.6	0.5000	0.1915	0.2769	0.3571	0.1862	0.2448	0.2908	0.3883	0.3326	0.3976	0.5266	0.4531
jedit_3.2_4.0	0.4333	0.1733	0.2476	0.4715	0.7733	0.5859	0.5208	0.6667	0.5848	0.4741	0.8533	0.6095
jedit_4.0_4.1	0.3835	0.7727	0.5126	0.5889	0.6709	0.6272	0.5517	0.7273	0.6275	0.7838	0.3671	0.5000
jedit_3.2_4.1	0.4783	0.1667	0.2472	0.5039	0.8101	0.6214	0.5455	0.7273	0.6234	0.4803	0.7722	0.5922
lucene_2.0_2.2	0.7681	0.4454	0.5638	0.6918	0.7692	0.7285	0.7571	0.4454	0.5608	0.6371	0.9875	0.7745
lucene_2.2_2.4	0.6923	0.7310	0.7111	0.6329	0.6650	0.6485	0.7739	0.4518	0.5705	0.6204	0.9806	0.7600
lucene_2.0_2.4	0.6120	0.9848	0.7549	0.6339	0.7208	0.6746	0.7768	0.4416	0.5631	0.6204	0.9806	0.7600
poi_2.0_2.5	0.6996	0.6573	0.6778	0.6781	0.3992	0.5025	0.8053	0.7339	0.7679	0.7240	0.9839	0.8342
poi_2.5_3.0	0.8560	0.7429	0.7954	0.7038	0.7214	0.7125	0.8547	0.7143	0.7782	0.6943	0.9571	0.8048
poi_2.0_3.0	0.7436	0.7250	0.7342	0.7333	0.5893	0.6535	0.8559	0.7214	0.7829	0.7034	0.9571	0.8109
synapse_1.0_1.1	0.4815	0.2281	0.3095	0.5946	0.3860	0.4681	0.5000	0.3158	0.3871	0.5077	0.5789	0.5410
synapse_1.1_1.2	0.5152	0.3953	0.4474	0.5417	0.4535	0.4937	0.5439	0.3605	0.4336	0.5190	0.4767	0.4970
synapse_1.0_1.2	0.5455	0.2791	0.3692	0.5634	0.4651	0.5096	0.5273	0.3372	0.4113	0.4483	0.4535	0.4509
xalan_2.4_2.5	0.6609	0.4258	0.5179	0.5922	0.4059	0.4817	0.5721	0.3221	0.4122	0.5957	0.7176	0.6510
xalan_2.5_2.6	0.6494	0.5221	0.5788	0.6274	0.6397	0.6335	0.5344	0.3431	0.4179	0.5804	0.8848	0.7010
xalan_2.4_2.6	0.6333	0.5123	0.5664	0.6506	0.2647	0.3763	0.5344	0.3431	0.4179	0.5983	0.8431	0.6999
Average	0.5626	0.4967	0.5020	0.5766	0.5452	0.5425	0.5588	0.5166	0.5164	0.5772	0.7270	0.6295

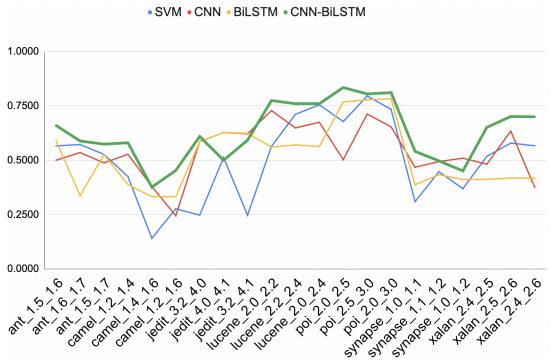

Figure 2: F1 Score Comparison Visualization

such as Lucene, POI, and Xalan, our models show a high degree of performance consistency across different versions. This shows that our model is able to predict results consistently when dealing with different versions of certain projects. This consistency can be partially attributed to the higher code similarity found between versions within the same project, making it easier for models to capture shared features and patterns.

There are some differences between versions of Ant and Synapse, these differences are relatively minor. In contrast, projects such as Camel and JEdit show more performance fluctuations, even within the same project. This suggests that the predictive performance of our model tends to vary when applied to certain projects. Although we cannot pinpoint the exact reasons behind these changes at this time, we speculate that they may have been influenced by a variety of factors, including project-specific characteristics, code complexity, and domain-related differences.

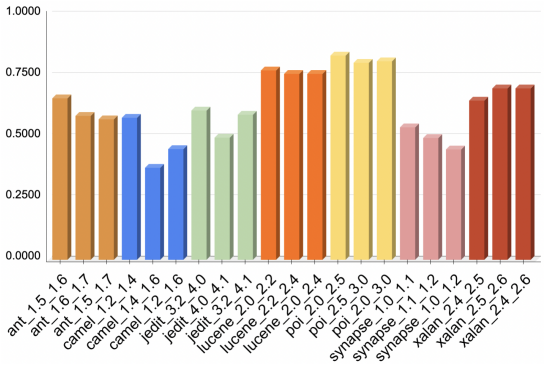

Figure 3: F1 Score Across PROMISE Projects

Table 5
Hyperparameter combinations obtained through Optuna

proj	Optuna Time	num_epochs	batch_size	learning_rate	filter_size	num_filters	rnn_hidden
ant_1.5_1.6	8.18 h	8	64	0.000185	3	186	256
camel_1.2_1.4	30.92 h	7	32	0.000148	7	120	1024
jedit_3.2_4.0	4.32 h	6	32	0.000251	11	157	64
lucene_2.0_2.2	1.21 h	3	128	0.008864	9	178	128
poi_2.0_2.5	3.93 h	6	128	0.000015	3	240	256
synapse_1.0_1.1	3.09 h	7	64	0.000458	5	346	64
xalan_2.4_2.5	35.23 h	5	128	0.000232	5	194	16

5.3. The impact of hyperparameters on the performance of CNN-BiLSTM model

To address RQ3, in this section, we study the impact of hyperparameters on the performance of the CNN-BiLSTM model for code defect prediction. Initially, we set the hyperparameters to the following values: the number of epochs is 10, the batch size is 64, the learning rate is 1e-4, the number of CNN filters is 128, the number of BiLSTM hidden units is 256, and the CNN filter size is 5 . After that, we fixed other hyperparameters, and then gradually manually adjust one of the other parameters, the CNN filter or the number of BiLSTM hidden units, to observe changes in model performance.

Figure 4 and Figure 5 show our experimental results, the x-axis is the change in the number of CNN filters and BiLSTM hidden units, and the y-axis shows the F1 score. We can see that the model performance fluctuates greatly when a single parameter changes. For example, the smaller the number of CNN filters, the better the performance of the model. In Figure 5, the F1 score drops after BiLSTM hidden unit is 16, but performs better and tends to be stable after 256. Exploring the impact of each hyperparameter individually would be a time-consuming task, and it is difficult to predict how the model will behave when these hyperparameters are combined. So we used Optuna, which will constantly try to search for hyperparameters that can make the model perform better based on the search algorithm.

Figures 6 and 7 show the F1 score (y-axis) for a certain number of trials (x-axis). Specifically, Figure 6 is a scatter plot, representing the F1 score that was obtained in each trial, e.g., when the trial number is 5, the F1 score is the value for the fifth trial. Figure 7 represents the best model performance that can be achieved based on the search until the current trial model is executed. So, in figure 7, when the trial number is 5, the F1 score is the best F1 score from the first to the fifth trial. We can observe that through continuous repetition and search, Optuna can gradually search for better results. The entire process is automated, which greatly simplifies our hyperparameter tuning process.

Table 5 provides a summary of hyperparameter combinations obtained through Optuna. These combinations have been identified to bring better performance for our code defect prediction model.

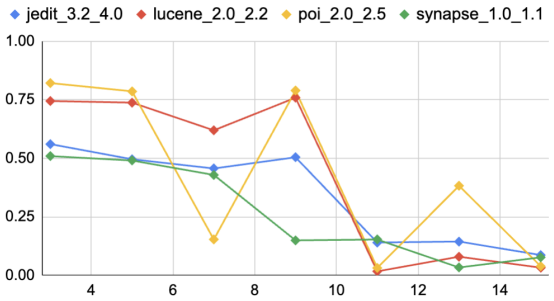

Figure 4: Effect of CNN Filter Length on F1 Score

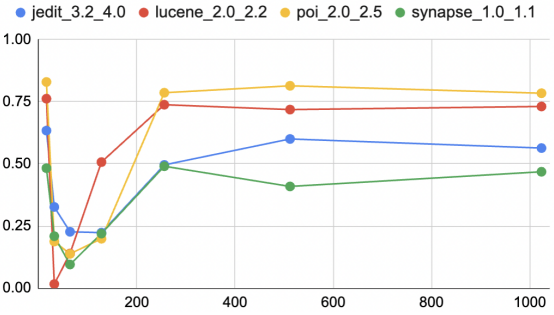

Figure 5: Effect of BiLSTM Hidden Units on F1 Score

5.4. Threats to Validity

In our research, we have identified and addressed several potential threats to the validity of our findings.

The implementation of our Python experimental code for processing source code text and building models poses a potential threat due to the possibility of bugs. To mitigate this, we took measures by leveraging mature third-party

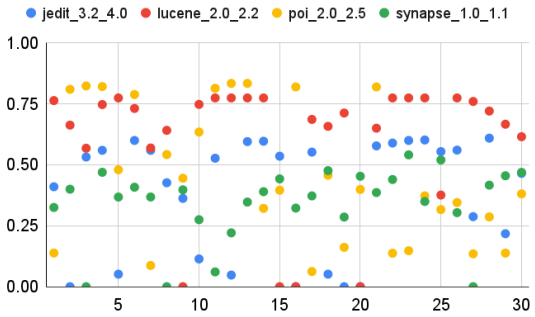

Figure 6: Scatter Plot of F1 Scores Across Optuna Trials

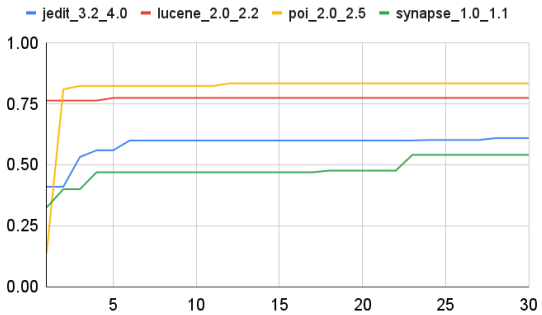

Figure 7: Progressive Improvement of Best Model Performance

libraries (such as javalang and PyTorch) and conducting thorough code inspections. Additionally, we applied random oversampling during data preprocessing, which could introduce bias. Future work will explore alternative methods to handle class imbalance and assess their impact on results. Moreover, the use of Optuna for hyperparameter optimization introduces potential variability in results due to different search spaces and numbers of trials. To reduce these threats, we plan to conduct more extensive searches and explore larger search spaces.

Our choice of a subset of projects from the PROMISE dataset due to time constraints may impact the generalizability of our findings, as the results may not generalize well to other projects. To address this, we intend to include a broader range of projects in future research.

We evaluated our models using a limited set of performance metrics, specifically precision, recall, and F1 measure. To reduce these threats, we will consider incorporating additional metrics such as AUC-ROC and MCC, among others, to provide a more comprehensive assessment of model performance.

6. Conclusion and Future Work

In this research, we have introduced a novel approach that leverages JavaBERT-based embeddings with a CNN-BiLSTM model for software defect prediction. Our approach harnesses semantic and contextual information in program code to enhance prediction accuracy. Through comprehensive experiments on the PROMISE dataset, we have demonstrated the superiority of our model over baseline models based on precision, recall, and F1-score metrics.

Although our study improves the performance of software defect prediction compared to baseline models, we still have many future works to do. In addition to what we discussed in the "threats to validity" session, we can also train the BERT model in different languages to adapt our methods to different programming languages.

References

[1] S. Omri, C. Sinz, Deep learning for software defect prediction: A survey, in: Proceedings of the IEEE/ACM 42nd international conference on software engineering workshops, 2020, pp. 209–214.

[2] F. Meng, R. Huang, J. Wang, A survey of software defects research based on deep learning, in: 2023 6th International Conference on Information Systems and Computer Networks (ISCON), IEEE, 2023, pp. 1–5.

[3] K. O. Elish, M. O. Elish, Predicting defect-prone software modules using support vector machines, Journal of Systems and Software 81 (2008) 649–660.

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

[5] J. Li, P. He, J. Zhu, M. R. Lyu, Software defect prediction via convolutional neural network, in: 2017 IEEE international conference on software quality, reliability and security (QRS), IEEE, 2017, pp. 318–328.

[6] J. Wang, C. Zhang, Software reliability prediction using a deep learning model based on the RNN encoder–decoder, Reliability Engineering & System Safety 170 (2018) 73–82.

[7] J. Deng, L. Lu, S. Qiu, Software defect prediction via LSTM, IET software 14 (2020) 443–450.

[8] H. Liang, Y. Yu, L. Jiang, Z. Xie, Seml: A semantic LSTM model for software defect prediction, IEEE Access 7 (2019) 83812–83824.

[9] H. Wang, W. Zhuang, X. Zhang, Software defect prediction based on gated hierarchical LSTMs, IEEE Transactions on Reliability 70 (2021) 711–727.

[10] M. N. Uddin, B. Li, Z. Ali, P. Kefalas, I. Khan, I. Zada, Software defect prediction employing BiLSTM and BERT-based semantic feature, Soft Computing 26 (2022) 7877–7891.
[11] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, arXiv preprint arXiv:1810.04805 (2018).
[12] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, J. Dean, Distributed representations of words and phrases and their compositionality, Advances in neural information processing systems 26 (2013).
[13] J. Pennington, R. Socher, C. D. Manning, Glove: Global vectors for word representation, in: Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), 2014, pp. 1532–1543.
[14] N. T. De Sousa, W. Hasselbring, JavaBERT: Training a transformer-based model for the Java programming language, in: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), IEEE, 2021, pp. 90–95.
[15] K. Fukushima, Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position, Biological cybernetics 36 (1980) 193–202.
[16] M. Schuster, K. K. Paliwal, Bidirectional recurrent neural networks, IEEE transactions on Signal Processing 45 (1997) 2673–2681.
[17] T. Akiba, S. Sano, T. Yanase, T. Ohta, M. Koyama, Optuna: A next-generation hyperparameter optimization framework, in: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, 2019, pp. 2623–2631.
[18] J. Sayyad Shirabad, T. Menzies, The PROMISE Repository of Software Engineering Databases., School of Information Technology and Engineering, University of Ottawa, Canada, 2005. URL: http://promise.site.uottawa.ca/SERepository.
[19] C. Thunes, javalang: pure Python Java parser and tools, 2020.