
Scalable Computing: Practice and Experience, ISSN 1895-1767, http://www.scpe.org
© 2025 SCPE. Volume 26, Issues 5, pp. 2183–2195, DOI 10.12694/scpe.v26i5.4910

SOFTWARE DEFECT PREDICTION MODEL BASED ON AST AND DEEP LEARNING

ZEZHI YE, CHENGHAI YU*, AND ZHILONG LU

Abstract. Software reliability prediction (SDP) theory is crucial for balancing software value and assessing efficiency. Traditional defect prediction relies on static code metrics for machine learning, but these handcrafted features fail to capture the code's syntactic structure and semantic information. In order to further predict the defects of the software, the Abstract Syntax Tree (AST) of the program was parsed on the basis of the metric data, and extracted as feature vectors, and the data was encoded by dictionary mapping and word embedding as the input of Convolutional Neural Network (CNN). On this basis, the Long Short-Term Memory network (LSTM) and Multi-head attention mechanism were used to further optimize the network, and the particle swarm optimization (PSO) was used to select the hyperparameters of the model, and the defects were predicted by using the model. The results show that the model can learn syntax and semantic features well, and the estimation accuracy is higher and the bias is smaller.

Key words: software defect prediction; Abstract Syntax Tree; Convolutional Neural Network; Long Short-Term Memory; Multi- Head Attention

1. Introduction. As modern software development progresses, software reliability has become a critical concern. This research aims to estimate the failure rate of future systems, hoping to predict software reliability at the early stages of development, thereby enabling developers to improve development and maintenance. However, the reliability of a system will change throughout its lifecycle depending on the conditions it is subjected to, making the selection of appropriate prediction methods a focal point in this research area [24, 2].

Traditional prediction methods typically involve two steps: extracting software metrics and building classification models. The main metrics include:

- *Object-Oriented Software Complexity Metrics* (e.g., C&K metrics)
- *Data Flow-Based Metrics* (e.g., Elshoff method)
- *Control Flow-Based Metrics* (e.g., McCabe method)
- *Program Volume-Based Metrics* (e.g., Halstead's software science method)

The classification models constructed include Naive Bayes [7], cluster analysis, and support vector machines [5]. Current research on static, small-dimensional data largely focuses on machine learning methods, with less emphasis on deep learning. In recent years, some scholars have attempted to introduce deep learning networks into this research area, using artificial neural networks (ANN), deep neural networks (DNN), and CNN [12] to make new attempts on software defect prediction datasets.

However, there are instances where static metrics fall short in accurately determining whether the code contains defects, as code with and without defects may have identical metric attributes. Due to the different syntactic and semantic information they contain, traditional machine learning classifiers find it difficult to differentiate. Abstract Syntax Trees (ASTs), based on the specific syntax structure of a program, embed rich semantic information that aids in accurately analyzing code. Their tree structure can describe the context of the code, allowing deep learning methods to mine the information contained in ASTs to achieve more accurate defect predictions at the source code level.

Therefore, by combining AST with deep learning, semantic feature information is extracted from source code using AST. The LSTM and multi-head attention mechanisms further differentiate key features within the classifier, enabling software defect prediction based on the semantic network.

*School of Computer Science and Technology, Zhejiang Sci-Tech University, Hangzhou, China (Corresponding author, ych@zstu.edu.cn)

2. Related Work.

2.1. Traditional Model. SDP remains a vital research area within the broader field of software engineering, with considerable attention devoted to it in the literature. Most of the literature focuses on designing new discriminative features, filtering defect data, and constructing effective classifiers. For instance, Nagappan and Ball [20] introduced code churn metrics and integrated them with software dependencies to predict defects. Moser [18] thoroughly analyzed the impact of change metrics and static code attributes on predicting defects. Additionally, Arar and Ayan [33] applied the Naive Bayes approach to identify and eliminate redundant features, improving the efficiency of defect prediction models. Mousavi [19] addressed the issue of class imbalance in SDP by utilizing ensemble learning techniques, which help balance the distribution of data across classes. Furthermore, Jing [9] proposed a dictionary learning method that focuses on computing misclassification costs to better predict software defects. In another study, Yu [30] employed correlation feature selection to identify and retain only the features that are strongly correlated with the target class, thereby enhancing the model's predictive capabilities. Ma [16] introduced Transfer Naive Bayes, it uses the data gravity method [21] to adjust the training set to build the classifier. Recent research has also highlighted the benefits of utilizing a small portion of labeled data from the target project to significantly improve prediction performance. For example, Qiu [22] created an innovative multi-component weight learning model by employing the kernel mean matching algorithm. This approach segments the source project's data into various components, applies KMM in each to modify the weights of the source instances. It then uses the weighted source instances from each component and a portion of the labeled data from the target project to construct a predictive model, finally, initialize the weights to build the final classification model. The final step involves initializing and optimizing the weights of source components to build a more accurate ensemble classifier.

We employ deep learning techniques to autonomously derive features, capturing the code information. These features replace static code attributes to improve defect prediction performance.

2.2. Applications of Deep Learning in Software Defect Prediction. Commonly used software data sets rely on manual statistics, which may greatly affect model performance and accuracy. Furthermore, these manual metrics often fail to fully exploit the contextual information of the code, which is crucial for understanding the syntax and semantics of programs. Program syntax and semantics can be represented through AST and Control Flow Graphs (CFG) [3]. Wang utilized Deep Belief Networks (DBN) to create latent features encapsulating program syntax and semantics, which were then used in classifiers to identify erroneous code [27]. Lin et al. [14] employed LSTM networks [8] to learn representations of program ASTs that could be transferred across projects to detect vulnerable functions. Dam [4] constructed a deep tree model based on AST for software defect prediction. Additionally, Li [11] built a hybrid model using features learned from convolutional neural networks [10] combined with static features. CFG, on the other hand, represents the control flow graph of a program, showing all possible paths that can be traversed during program execution. Zhou et al.[32] used CodeBERT to explore real-time prediction and proposed the JIT-SDP model, which extracts code submission and change information and uses it for prediction, thus enhancing the generalization ability of the code prediction model. Parvez [17] used transform to propose Bugsplorer, which hierarchically marks code elements and predicts file-level defects.

The deep learning-based approaches mentioned above generally treat all latent features as equally important, which can result in a failure to pinpoint those features that are most discriminative in terms of crucial syntax and semantics. This oversight can contribute to less accurate defect predictions. To address this issue, our method introduces an attention mechanism that focuses on identifying and prioritizing these key features by assigning them greater weights. Furthermore, we opt to use the AST as the program representation instead of the CFG, as the AST more effectively captures the structural nuances and retains more comprehensive information about the source code.

3. Model Design.

The final model utilizes AST to automatically extract features. These features capture the syntax and semantic information of the program as input. CNN is chosen as the basic classifier, and LSTM and Multi-Head Attention layers are used to learn and analyze the relationships between the data, extracting key features for training a more accurate model. To address the data differences among different datasets, hyperparameter tuning is introduced, with the Particle Swarm Optimization algorithm being used for efficient

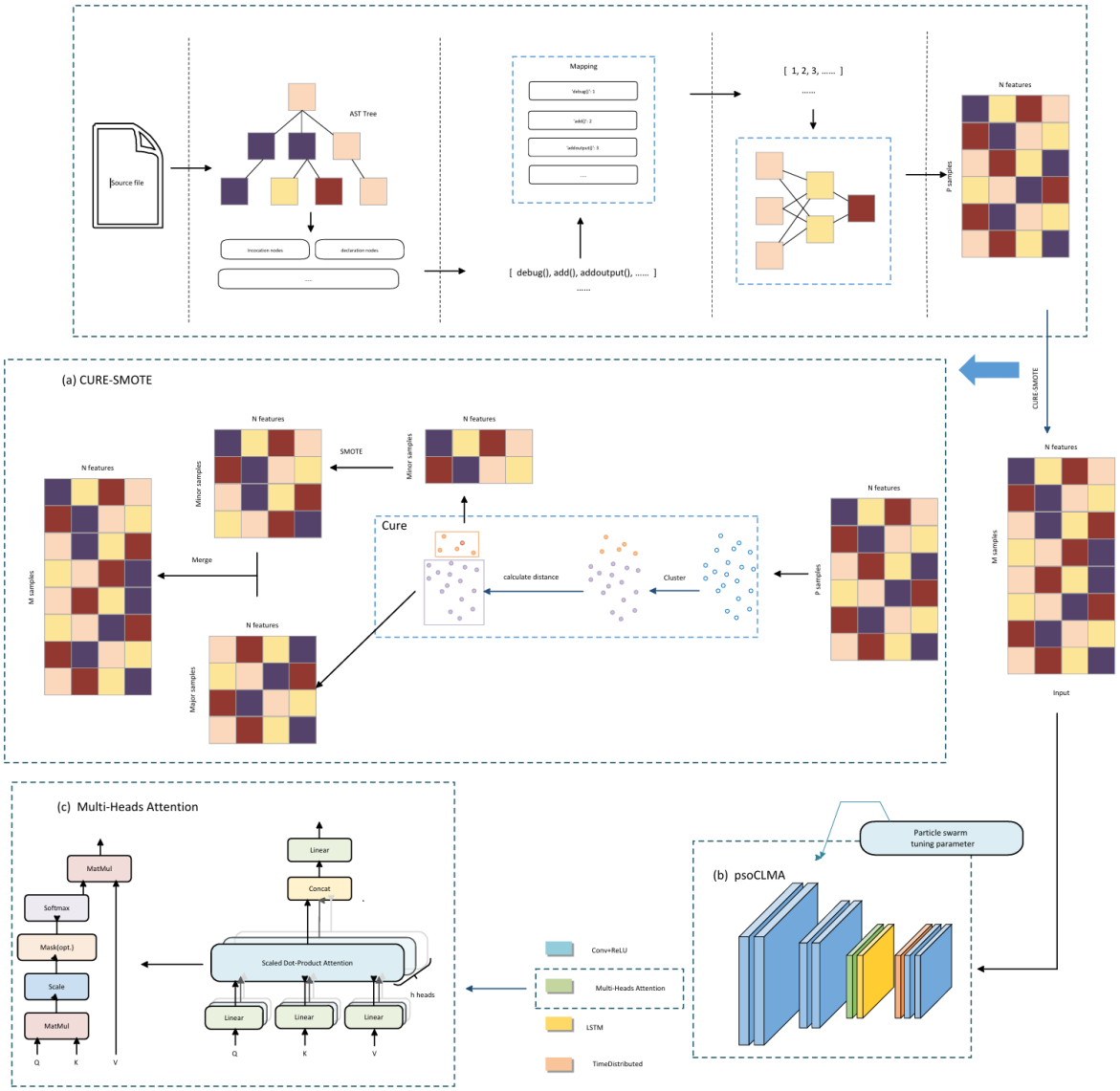

Fig. 3.1: Network framework

parameter adjustment. Final model is show in Figure 3.1.

The main contributions of this paper are as follows: We propose a software defect prediction model (ACNN) that combines AST semantic network and deep learning. Using AST to extract code semantic data avoids the tedious process of manually extracting data and possible deviations. On this basis, according to the characteristics of the dataset, LSTM and Multi-Head Attention are used to make the classifier better focus on the extracted data. Finally, the effectiveness of each improvement is verified through ablation experiments on the same dataset, and the experimental results of different datasets are compared to verify the overall effectiveness of the model proposed in this paper.

3.1. AST and data processing. When constructing AST, three types of nodes are selected to express the code semantic information: (1) method call nodes; (2) declaration nodes; (3) control flow nodes. AST can

```
1  package com;
2  public class A{
3      public static void main(String[]args){
4          int i=0;
5          while(i<10){
6              i++;
7          }
8           System.out.println(i);
9      }
10 }
```

(a)

```
1  package com;
2  public class B{
3      public static void main(String[]args){
4          int i=0;
5          while(i<10){
6
7          }
8           System.out.println(i);
9      }
10 }
```

(b)

Fig. 3.2: Source code of two example files: (a) The clean code A; (b) The defective code B

effectively understand the structure and semantic information of the source code. In Figure 3.2, code snippet A is very similar to code snippet B, which means that manually extracted feature vectors may be identical. However, the syntax trees in Figure 3.3 show that the AST of code A (a) has two more nodes than the AST of code B (b) (i.e., the Statement Expression and Member Reference nodes marked with red boxes).

Vectors of strings cannot be used directly as experiment input. Therefore, a dictionary mapping tokens to integers is constructed based on token frequency. Tokens are sorted by frequency, and an ordered token dictionary is established in descending order of frequency. Properly normalizing the vector lengths can prevent sparse matrices from being used as input. For vectors shorter than the specified length, zero padding is performed to ensure that feature vectors have the same length. Truncates vectors that are too long. Finally, the word embeddings in the network are used as a trainable dictionary while ensuring that the input data will not be a sparse matrix.

Software defect prediction data often exhibits significant class imbalance, where defect instances typically constitute only a small fraction of the total dataset. When such imbalanced data is used directly for model training, the resulting predictions tend to be skewed toward the majority class, leading to biased and inaccurate outcomes. To address this challenge, two primary approaches are commonly employed: oversampling and undersampling. To preserve the integrity and completeness of the data, we opt for the oversampling approach and utilize the CURE-SMOTE algorithm [15, 31] to achieve data balance. This sophisticated algorithm effectively identifies and removes noise and outliers from the original samples. Following this, a modified algorithm is employed to generate new synthetic samples by interpolating between representative points and central points within the clusters. By preprocessing the data in this manner, the model becomes less biased toward the majority class, resulting in more accurate and reliable predictions.

Moreover, models trained on such balanced datasets are better equipped to generalize to unseen data, enhancing their robustness and performance in real-world applications. This approach ensures that the model remains effective across diverse scenarios, making it a valuable tool for software defect prediction. The formula for updating the model parameters is presented below:

$$\mathrm{dist}(\mathrm{P},\mathrm{Q}) = \min_{p\in \mathrm{P}_r, q\in \mathrm{Q}_r}(p,q)$$

$$\mathrm{P}_c \frac{|\mathrm{P}|\cdot \mathrm{P}_c + |\mathrm{Q}|\cdot \mathrm{Q}_c}{|\mathrm{P}|+|\mathrm{Q}|}$$

$$\mathrm{P}_r\,\{p+\alpha\cdot(\mathrm{P}_c-p)|p\in \mathrm{P}_r\}$$

$$\mathrm{X}^n_{new} = \overline{X} + rand(0,1)\times(\mathrm{P}_r - \overline{X})$$

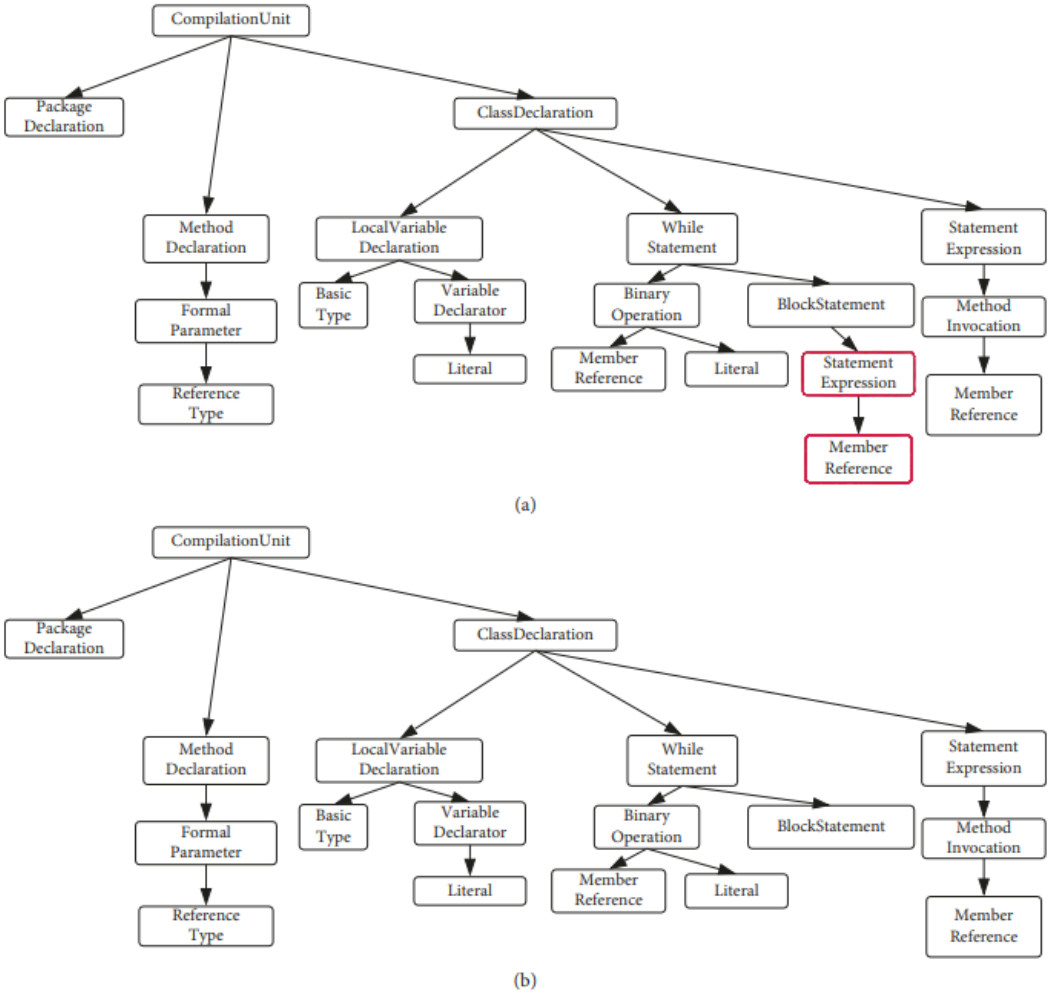

Fig. 3.3: AST of two example code: (a) The clean code A; (b) The defective code B

$$\mathrm{X}_{new} = \overline{X} \cup \mathrm{X}^{\mathrm{n}}_{new}$$

The ratio of processed samples is shown in Figure 3.4.

3.2. Network layers. The core design of the proposed model is based on CNN [6, 1], Multi-Head Attention mechanisms [28, 26, 13], and LSTMs [29]. First, the CNN is used for local feature extraction. Then, the LSTM is used to analyze the associations between data to ensure data integrity, while the Multi-Head Attention is used to enhance the model's data processing by assigning different attention weights to different types of data. The input data, which consists of software-related metrics, is passed through a one-dimensional convolutional layer (Conv1D). This layer can effectively extract local features from the sequence data and adjust the output shape. Next, the output from the convolutional layer goes through a one-dimensional max-pooling layer (MaxPooling1D), which performs pooling operations on the convolutional layer's output, reducing the feature size by half. This helps reduce the number of parameters in the model and extracts the most significant features from the data.

Then, the Multi-Head Attention mechanism is introduced. This layer performs operations of multiple attention heads, helping the model capture important information in the sequence without altering the output

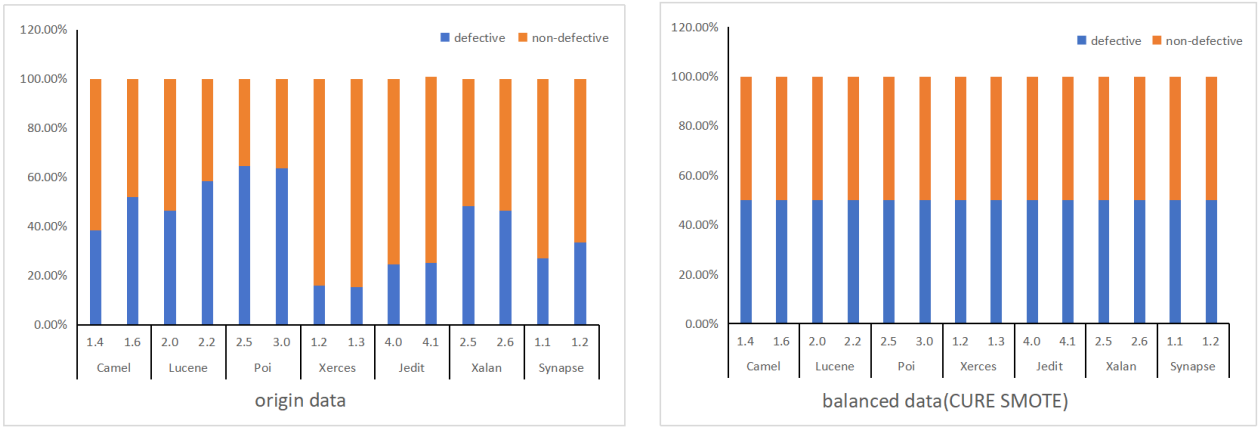

Fig. 3.4: Sample Distribution Before and After

shape. After the attention layer, a masking layer is used to mask the padded parts of sequences of unequal lengths, so that these padded data can be ignored in subsequent processing. Following this is an LSTM layer used to model sequence data and capture long-term dependencies in the sequences. After the LSTM layer, a TimeDistributed layer is used to apply a fully connected layer to each time step in the input sequence. This operation allows the model to generate a single prediction value for each time step, achieving final binary classification prediction. Finally, the outputs of multiple flattened layers are concatenated through a Concatenate layer, followed by two Dense layers for the final classification or prediction.

All parameters within the entire model are designed to be fully trainable, allowing for dynamic adjustment and optimization during the learning process. This network structure has been carefully designed to maximize the strengths of convolutional layers, attention mechanisms, and recurrent structures, each playing a crucial role in processing sequence data. The goal is to enhance the model's ability to perform accurate prediction or classification tasks. The synergy between these components ensures that the model can effectively capture and utilize patterns within the data. The detailed composition and underlying principles of each component are elaborated upon in the following sections.

3.2.1. CNN. CNNs are similar to artificial neural networks and are a type of deep learning model consisting of input layers, convolutional layers, activation layers, pooling layers, fully connected layers, and output layers. The convolutional layers are responsible for feature extraction, the activation layers use activation functions for multilayer mapping learning, the pooling layers extract main features to prevent overfitting, and the fully connected layers integrate extracted features for classification by the classifier.

CNN are extensively used in processing image data, where they have proven to be highly effective. However, there has been relatively limited exploration of building one-dimensional (1D) CNNs to address traditional machine learning problems. Despite the difference in dimensionality, CNNs, whether 1D or 2D, retain the same core features and approaches to problem-solving. The primary challenge lies in managing the dimensionality of the input data and understanding how filters behave as they move across the input.When analyzing experimental data, it becomes clear that there is often a significant correlation among the data points. Recognizing and isolating important information within the sequence is crucial for making more accurate predictions. The ability to focus on key elements in the data sequence enhances the predictive power of the model, demonstrating the importance of correctly handling the dimensionality and filter behavior within CNNs, regardless of their application to 1D or 2D data.

3.2.2. Multi-Head Attention. Traditional neural networks may face multiple information interference when processing input data. To address this issue, Multi-Head Attention is introduced to reprocess the output of the deep convolutional model for better data information extraction. The attention mechanism comprises

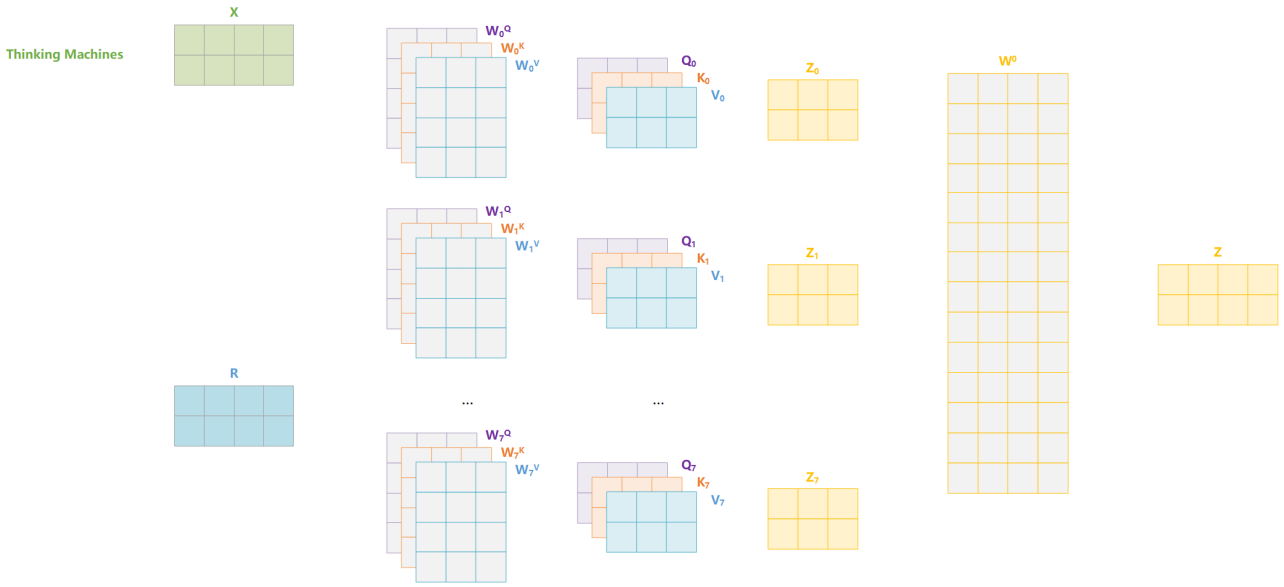

Fig. 3.5: Multi-Head Attention structural diagram

queries, keys, and values, essentially a mapping of relationships. Multi-Head Attention is an optimization of the attention mechanism, further refining the Self-Attention layer. The introduction of multiple attention heads maps inputs to different spaces and computes scores, enhancing the model's ability to focus on features. The structural diagram is shown in Figure 3.5:

3.2.3. LSTM. LSTMs have been improved and popularized by many people for various problems. LSTMs introduce three gate structures for control: forget gate, input gate, and output gate. Using gated units to update the unit state, LSTMs are essentially a special type of RNN capable of better handling data relationships compared to standard RNNs. Its unique memory cell structure allows the model to retain and fully utilize feature information over longer periods, leveraging this capability to learn complex patterns and relationships in software metric data and help build predictive models.

3.3. Hyperparameter Optimization. There are subtle differences between datasets involved in software defect issues, and the selected metrics may not be entirely consistent, which could lead to reduced model generalizability. Therefore, when selecting model parameters, a swarm intelligence algorithm is used to dynamically adjust parameters. The PSO algorithm [23, 25] generates the same number of samples as the problem to be solved, treating each sample as a possible solution. The problem is bio-mimicked as a food source for birds, simulating this foraging for subsequent solving. It can efficiently and conveniently search for optimal parameters within the solution space.

During the search for the optimal solution, particles learn from two values: the global historical best value and the individual historical best value. Particles change their position and flying speed by learning from their own historical best value and the population's historical best value. These learning behaviors are called "self-learning" and "social learning," respectively. The global and individual historical best values are determined by fitness values, derived from the solution in the actual problem. Particles adjust their positions and speeds by referencing individual historical and global historical best values. When the updated position's fitness value surpasses its historical best or global best, the two best values are updated to the current value. During particle movement, it is also necessary to keep away from nearby individuals to avoid collisions with other particles.

The update formulas for particle position and speed are as follows:

$$v_{ij}(\tau+1) = \omega v_{ij} + c_1 r_1 \left|gbest(\tau) - x_{ij}(\tau)\right| + c_2 r_2 \left|pbest(\tau) - x_{ij}(\tau)\right|$$

Table 4.1: Confusion matrix

		True value	
		Positive	Negative
Predicted value	Positive	TP	FP
	Negative	FN	TN

Table 4.2: Dataset details

Name	LIne of files	Defective Rate	Function
camel1.4	872	16.72%	Train set
camel1.6	965	19.57%	Test set
lucene2.0	195	46.67%	Train set
lucene2.2	247	58.47%	Test set
poi2,5	385	64.42%	Train set
poi3.0	442	63.57%	Test set
jedit4.0	306	24.51%	Train set
jedit4.1	312	25.32%	Test set
synapse1.1	222	27.03%	Train set
synapse1.2	256	33.60%	Test set
xalan2.5	803	50.18%	Train set
xalan2.6	885	46.91%	Test set
xerces1.2	440	50.13%	Train set
xerces1.3	453	15.05%	Test set

Table 4.3: Ablation test results

	Accuracy	Recall	F1	Mcc
C+L	0.8070	0.8419	0.8192	0.6292
C+L+M	0.8314	0.8229	0.8295	0.4356
C+L+M+P	0.9123	0.8462	0.8776	0.7658

$$x_{ij}(\tau+1) = x_{ij}(\tau) + v_{ij}(\tau+1)$$

4. Experiments and Analysis. The existence of defects is a binary classification problem. The model's final prediction results in four outcomes as shown in the confusion matrix in Table 4.1. The evaluation metrics used are the standard evaluation metrics for binary classification problems: Accuracy, Recall, F1-score, and Matthews Correlation Coefficient (MCC).

This experiment builds a software defect prediction model based on the Keras framework. Keras is an open source machine learning library based on Python. It is well compatible with various data processing libraries (such as pandas, numpy, etc.) and has built-in multiple classic models, making it convenient for users to create new machine learning models based on classic models. The experimental machine includes GPU: RTX 4090, CPU: Intel(R) Xeon(R) Gold 5418Y * 12 cores.

The dataset used in this paper includes seven open-source Java projects, each project containing two versions: (camel1.4, camel1.6), (lucene2.0, lucene2.2), (poi2.5, poi3.0), (jedit4.0, jedit4.1), (synapse1.1, synapse1.2), (xalan2.5, xalan2.6), and (xerces1.2, xerces1.3). One as the training set and another is used as the test set. The detailed original data information is shown in Table 4.2.

4.1. Ablation experiments. We conducted ablation experiments on the model. As an example, using jedit4.0 and jedit4.1, the outcomes of the ablation experiments are presented in Table 2, where LSTM is denoted as L, Multi-Head Attention as M CNN as C, and PSO as P. Table 4.3 Ablation test results The experimental

Table 4.4: Accuracy of Different Models

	ACNN	SVM+RF	ARNN	CNN
Camel	0.7807	0.6991	0.4935	0.6403
Lucene	0.6009	0.5309	0.5324	0.5744
Poi	0.7159	0.6073	0.7722	0.6613
Xerces	0.8539	0.6719	0.5195	0.8112
Jedit	0.9123	0.7456	0.8843	0.7527
Xalan	0.7429	0.6018	0.5062	0.5858
Synapse	0.7176	0.6784	0.5221	0.6275
average	0.7606	0.6479	0.6043	0.6647

Table 4.5: Recall of Different Models

	ACNN	SVM+RF	ARNN	CNN
Camel	0.7797	0.4521	0.4534	0.6399
Lucene	0.5968	0.5845	1.0000	0.6812
Poi	0.7326	0.7071	0.8718	0.6614
Xerces	0.8604	0.2537	1.0000	0.8111
Jedit	0.8462	0.6618	0.8263	0.7581
Xalan	0.6690	0.6268	0.0000	0.5911
Synapse	0.7137	0.3882	0.9265	0.5962
average	0.7426	0.5249	0.7254	0.6770

data reveals that both LSTM networks and Multi-Head Attention mechanisms have a subtle but generally positive impact on the experimental outcomes. The model combining Convolutional Neural Networks with LSTM demonstrates stable performance, with various evaluation metrics consistently hovering around the 80% mark. The MCC value for this model is close to 0.6, suggesting a moderate level of correlation between predicted and actual results. The introduction of Multi-Head Attention further boosts the model's performance across the board, underscoring the attention mechanism's capability to effectively capture essential data features and enhance classification accuracy. When both LSTM and Multi-Head Attention are integrated and PSO is applied for parameter tuning, the model fully leverages the strengths of both networks. This results in a marked improvement in overall model performance, with the MCC value rising to 0.76—a notable increase of 0.33 compared to using the networks separately. This configuration achieves the highest scores across all evaluation metrics, indicating that the model has reached its optimal performance level.

In the ablation experiments, each incremental addition of a module leads to a noticeable performance gain. Particularly, the integration of the Multi-Head Attention mechanism and the fine-tuning of hyperparameters play a positive effect on significantly enhancing the model's effectiveness. The final model, incorporating all these elements, delivers the best performance across all measured metrics, demonstrating its superior efficiency in classification tasks and validating the importance of each component in the overall model architecture. This comprehensive improvement highlights the synergy between the modules and the effectiveness of the proposed model design.

4.2. Comparison with Other Algorithms. To further validate the model's effectiveness, we selected the following three baseline methods for comparison with our model:

a) **SVM+RF:** A method combining Random Forest (RF) and SVM based on traditional machine learning.

b) **ARNN:** An Attention-based RNN prediction model.

c) **CNN:** A traditional CNN.

When constructing the network architectures for ARNN and CNN.

Box plots were also used to visually present the performance of each model across different datasets, as shown in Figure 4.1, the box plot of experimental results.

Table 4.6: F1 of Different Models

	ACNN	SVM+RF	ARNN	CNN
Camel	0.7690	0.3769	0.6736	0.6402
Lucene	0.5968	0.6409	0.6921	0.6216
Poi	0.7272	0.7374	0.8111	0.6591
Xerces	0.8362	0.1889	0.6797	0.8111
Jedit	0.8776	0.5556	0.8836	0.7567
Xalan	0.7172	0.5963	0.0000	0.5910
Synapse	0.7152	0.4459	0.6530	0.5962
average	0.7485	0.5060	0.6276	0.6680

Table 4.7: MCC of Different Models

	ACNN	SVM+RF	ARNN	CNN
Camel	0.5343	0.1903	NAN	0.2809
Lucene	0.1936	0.2058	NAN	0.1886
Poi	0.4515	0.3230	0.5348	0.3161
Xerces	0.6614	0.0002	NAN	0.6222
Jedit	0.7658	0.3936	0.7963	0.5134
Xalan	0.4798	0.2063	NAN	0.1821
Synapse	0.4315	0.2315	NAN	0.1924
average	0.5026	0.2215	0.1902	0.3280

The experimental results in Tables 4.4-4.7 indicate that the prediction performance of the model in this paper is generally superior to SVM and deep learning network algorithms. Figure 4.1 clearly shows the differences between models on different metrics, with the model exhibiting high accuracy and stability across metrics, indicating high robustness and consistency across different datasets. In contrast, the ARNN model performs exceptionally well on most datasets but slightly underperforms on some, while the SVM model shows more variability in prediction performance and overall does not perform as well as the model in this paper. Taking the jedit project as an example, the accuracy values for ACNN, ARNN, and CNN are 0.9123, 0.8843, and 0.7527, respectively, while SVM+RF only achieves 0.7456. Clearly, ACNN, CNN, and RNN perform better than traditional methods. On average, deep learning methods have higher MCC values than traditional methods. Specifically, ACNN achieves the highest values, and Table 6 shows that the ACNN model also excels in the MCC metric, with an average value of 0.5026, significantly higher than other models. Particularly in the Jedit dataset, the MCC value of ACNN is 0.7658. The MCC values for SVM+RF and CNN models are relatively low, at 0.2215 and 0.3280, respectively. The ARNN model lacks MCC data on some datasets, possibly due to gradient explosions, but its performance is not satisfactory based on available data. This reflects the advantages of the model shown in the article and verifies the stability of the defect prediction model based on deep learning, strong discrimination effect, good sample segmentation effect, and high correlation between prediction results and actual results.

In terms of accuracy, the model in this paper achieves the best results on seven out of eight datasets and also obtains the best average value, indicating good predictive ability. In contrast, while ARNN performs well on a few datasets, showing slight improvements over the model in this paper, its overall accuracy is still inferior. In terms of recall and F1 scores, the model in this paper also achieves the best average values, indicating good sensitivity to positive samples, i.e., defect data, reducing the miss rate, and achieving a good balance between precision and recall. SVM+RF has some shortcomings in defect prediction. Results show that, deep learning methods have improved the ability to distinguish defective code.

Comparing ACNN with other deep learning methods: From the F1 perspective, compared to CNN and ARNN, the proposed ACNN achieves the best F1 metric four times. This indicates that ACNN has better performance than CNN and ARNN in terms of stability in software defect prediction. Regarding MCC, ACNN

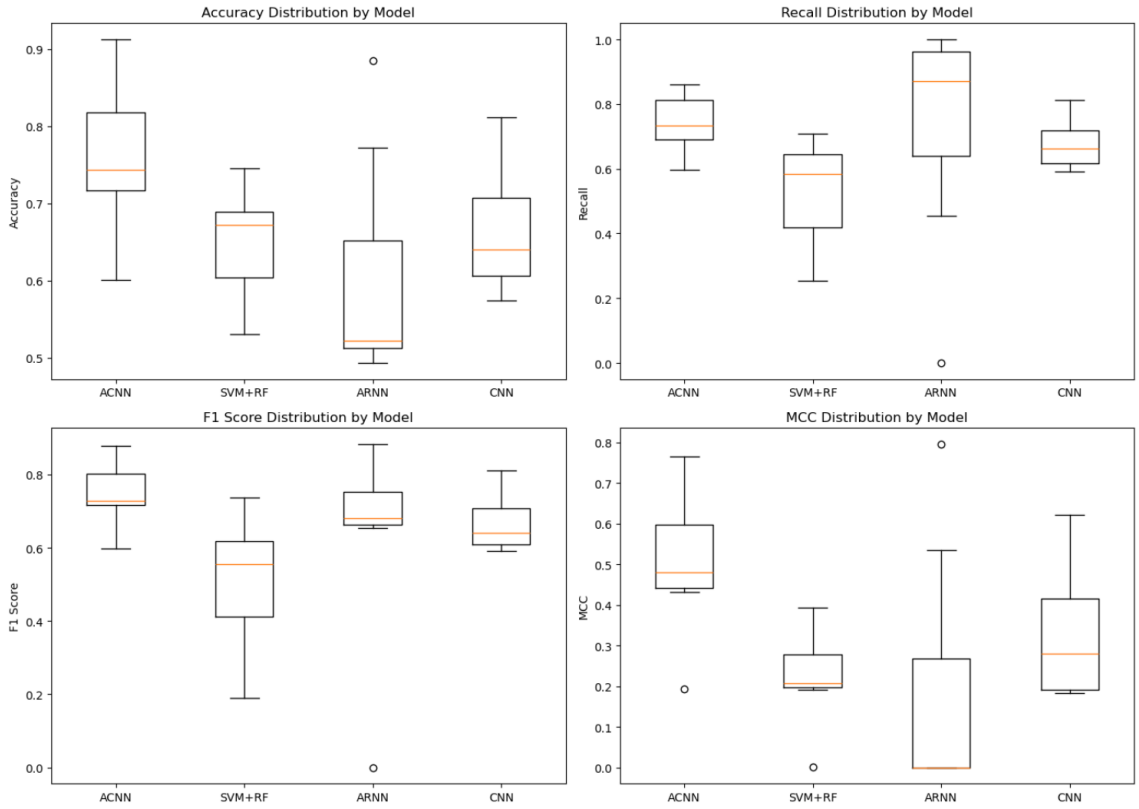

Fig. 4.1: The box plot of experimental results

improves CNN's MCC value by an average of 0.1746 and ARNN's by 0.3124. This indicates that ACNN improves the ability to distinguish software defects. According to the average F1 metric and ACC of the seven projects in the table, the proposed ACNN is 12.05% higher than CNN in the F1 metric and 53.35% higher in MCC. Particularly, in the jedit dataset, ACNN's F1 metric is 15.98% higher than CNN's, and MCC is 49.17% higher, indicating that the attention mechanism positively impacts further generating key features, thereby improving defect prediction performance.

A comprehensive analysis of the experimental results shows that the ACNN model performs particularly well when dealing with unbalanced datasets, especially when the dataset contains more defective samples (such as Synapse, Poi), and its ability to identify minority classes is better than other models. This is particularly evident in the comparison of F1 value and MCC value. ACNN has good multi-level feature extraction capabilities: the model combines CNN, LSTM and multi-head attention mechanism, which can not only capture the local features of the code, but also learn the long-term dependencies between codes through LSTM, and further extract key features using multi-head attention mechanism. This enables the model to better mine grammatical and semantic features on complex datasets, especially in projects such as Jedit and Xerces, and achieve higher prediction accuracy. The particle swarm optimization (PSO) algorithm is used to dynamically adjust the model parameters, which improves the generalization ability of the model, enabling it to maintain good performance on different datasets, especially in terms of recall rate and F1 value, which can balance the correctness and error rate of prediction and reduce bias.

The ACNN model performs well on most datasets, but there are still some limitations on some specific datasets. Especially on the Lucene dataset, ACNN scores relatively low. This may be due to the small size of the Lucene dataset, where grammatical and semantic information have little impact on the prediction results,

making it impossible for the multi-layer deep learning model to fully extract effective features.

Overall, the model in this paper performs excellently, showing outstanding results in precision, defect sample recognition ability, and overall performance. Deep neural networks perform well on some datasets and may have good predictive capabilities in certain situations, but overall are inferior to the model in this paper. In contrast, traditional support vector machine models perform well on individual datasets, particularly in terms of accuracy, but overall performance is weaker, especially in recall and comprehensiveness, significantly lower than the model in this paper.

The experimental data demonstrates that incorporating CNN, LSTM, attention mechanisms and particle swarm algorithm optimization parameters effectively improves the model's classification capability. High robustness and consistency indicate an advantage in practical applications, providing a foundation for further research and application.

5. Conclusion. Software systems today are intricately intertwined with computer technology, making the effective evaluation of overall quality before the development process begins a critical factor in enhancing development efficiency. As a result, accurately predicting software defects has become a central focus of ongoing development efforts and a key area of research interest among scholars. Traditional methods for defect prediction have largely relied on mathematical calculations, which, while useful, can be limited in their ability to fully capture the complexities of software systems. In contrast, employing deep learning techniques for prediction allows for a more nuanced judgment and analysis of data, enabling more precise and reliable outcomes.The ACNN (Attention-based Convolutional Neural Network) model introduced in this study represents a significant advancement in this area, as it can automatically extract syntactic and semantic features directly from the source code. This capability leads to more accurate predictions and a deeper understanding of the underlying code structures. Extensive experiments conducted on seven open-source projects have demonstrated that the ACNN model outperforms traditional static prediction models in terms of overall performance and exhibits greater universality and adaptability. Additionally, when compared to other deep learning-based models, ACNN consistently delivers better results. On average, the ACNN model improves accuracy by 14.43%, enhances the F1 measure by 12.05%, and increases the Matthews Correlation Coefficient (MCC) by 53.35% compared to baseline methods. These substantial improvements suggest that the ACNN model is better suited to the specific challenges of software defect prediction datasets and achieves higher accuracy in predicting potential issues.

Skipping manual data statistics and using AST to implement automatic code information extraction makes the model more realistic. The optimized model has higher prediction accuracy, providing a basis for intelligent defect prediction in actual software development and testing, to further validate the effectiveness and robustness of the ACNN model in the domain of defect prediction, we plan to extend our experiments to include a broader range of projects, including personal and smaller-scale projects. This expanded testing will provide deeper insights into the model's adaptability and performance across different contexts, ultimately contributing to its refinement and the advancement of software defect prediction techniques.

Data Sharing. The datasets used and/or analyzed during the current study are available from the corresponding author on reasonable request.

REFERENCES

[1] K. Cao, Y. Wang, and H. Tao, *A sleep staging model based on selt-attention mechanism and bi-directional lstm*, Software Guide, 23 (2024), pp. 24–32.
[2] X. Chen, L. Wang, Q. Gu, and et al, *A survey on cross-project software defect prediction methods*, Journal of Computers, 41 (2018), pp. 254–274.
[3] K. D. Cooper, T. J. Harvey, and T. Waterman, *Building a control-flow graph from scheduled assembly code*, 2002.
[4] H. K. Dam, T. Pham, S. W. Ng, and et al, *A deep tree-based model for software defect prediction*, 2018.
[5] K. O. Elish and M. O. Elish, *Predicting defect-prone software modules using support vector machines*, Journal of Systems and Software, 81 (2008), pp. 649–660. Software Process and Product Measurement.
[6] Z. Ge, N. Liu, and S. Wang, *Bearing fault diagnosis method based on mckd and improved cnn-lstm*, Journal of Tianjin University of Technology, pp. 1–11.
[7] S. Guo, R. Huang, and Q. Li, *A software defect prediction algorithm using improved weighted naive bayesian*, Control Engineering of China, 28 (2021), pp. 600–605.

[8] S. Hochreiter and J. Schmidhuber, *Long short-term memory*, Neural Computation, 9 (1997), pp. 1735–1780.
[9] X. Jing, S. Ying, Z. Zhang, and et al, *Dictionary learning based software defect prediction*, in Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, New York, NY, USA, 2014, Association for Computing Machinery, p. 414–423.
[10] Y. Kim, *Convolutional neural networks for sentence classification*, 2014.
[11] J. Li, P. He, J. Zhu, and et al, *Software defect prediction via convolutional neural network*, in 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), 2017, pp. 318–328.
[12] J. Li, P. He, J. Zhu, and M. R. Lyu, *Software defect prediction via convolutional neural network*, 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), (2017), pp. 318–328.
[13] Z. Li, L. Li, J. Chen, and D. Wang, *A multi-head attention mechanism aided hybrid network for identifying batteries' state of charge*, Energy, 286 (2024), p. 129504.
[14] G. Lin, J. Zhang, W. Luo, and et al, *Cross-project transfer representation learning for vulnerable function discovery*, IEEE Transactions on Industrial Informatics, 14 (2018), pp. 3289–3297.
[15] L. Ma and S. Fan, *Cure-smote algorithm and hybrid algorithm for feature selection and parameter optimization based on random forests*, BMC bioinformatics, 18 (2017), p. 169.
[16] Y. Ma, G. Luo, X. Zeng, and et al, *Transfer learning for cross-company software defect prediction*, Inf. Softw. Technol., 54 (2012), p. 248–256.
[17] P. Mahbub and M. M. Rahman, *Predicting line-level defects by capturing code contexts with hierarchical transformers*, 2023.
[18] R. Moser, W. Pedrycz, and G. Succi, *A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction*, 2008 ACM/IEEE 30th International Conference on Software Engineering, (2008), pp. 181–190.
[19] R. Mousavi, M. Eftekhari, and F. Rahdari, *Omni-ensemble learning (oel): Utilizing over-bagging, static and dynamic ensemble selection approaches for software defect prediction*, International Journal on Artificial Intelligence Tools, 27 (2018), p. 1850024.
[20] N. Nagappan and T. Ball, *Using software dependencies and churn metrics to predict field failures: An empirical case study*, in First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), 2007, pp. 364–373.
[21] L. Peng, B. Yang, C. Y.H., and et al, *Data gravitation based classification*, Information Sciences, 179 (2009), pp. 809–819.
[22] S. Qiu, L. L., and S. Jiang, *Multiple-components weights model for cross-project software defect prediction*, IET Softw., 12 (2018), pp. 345–355.
[23] F. Shao and Q. Duan, *Network reconfiguration of ship power system based on improved particle swarm optimization algorithm*, Manufacturing Automation, 44 (2022), pp. 100–103.
[24] H. Tao, X. Niu, L. Xu, and et al, *A comparative study of software defect binomial classification prediction models based on machine learning*, Software Quality Journal, 32 (2024), p. 1203–1237.
[25] J. Wang, C. Si, K. Wang, and et al, *Intrusion detection method based on ensemble learning and improved pso-ga feature selection*, Journal of Jilin University (Engineering and Technology Edition), pp. 1–9.
[26] L. Wang and H. Jin, *Multi-fault recognition of air data system based on multi-head selfattention mechanism*, Journal of Jiangsu University, 44 (2023), pp. 444–451.
[27] S. Wang, T. Liu, and L. Tan, *Automatically learning semantic features for defect prediction*, in Proceedings of the 38th International Conference on Software Engineering, ICSE '16, New York, NY, USA, 2016, Association for Computing Machinery, p. 297–308.
[28] Y. Wang, *Study on cardinality estimation method based on multi-head self-attention mechanism*, master's thesis, Nanjing University of Posts and Telecommunications, 2023.
[29] Y. Xia and C. Xiong, *Vehicle trajectory prediction combing bidirectional lstm and attention mechanism*, Journal of Chongqing University of Posts and Telecommunications (Natural Science Edition), 36 (2024), pp. 299–306.
[30] Q. Yu, S. Jiang, and J. Qian, *Which is more important for cross-project defect prediction: Instance or feature?*, in 2016 International Conference on Software Analysis, Testing and Evolution (SATE), 2016, pp. 90–95.
[31] L. Zhang, S. Y.T., and Y. Zhu, *Software defect prediction based on improved smote*, Computer Engineering and Design, 44 (2023), pp. 2965–2972.
[32] X. Zhou, D. Han, and D. Lo, *Assessing generalizability of codebert*, in 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2021, pp. 425–436.
[33] Ömer Faruk Arar and K. Ayan, *A feature dependent naive bayes approach and its application to the software defect prediction problem*, Applied Soft Computing, 59 (2017), pp. 197–209.

Edited by: Manish Gupta
Special issue on: Recent Advancements in Machine Intelligence and Smart Systems
Received: Sep 13, 2024
Accepted: Oct 11, 2024