Computer Fraud and Security
ISSN (online): 1873-7056

Research on Intelligent Decision-Making AI Algorithms for Software Defect Identification

Zhiyong Ding[1,*], Yanguang Cai[2], Xiaojun Liv[1]
[1]School of Information Engineering, Guangdong Polytechnic, Foshan, Guangdong 528041, China
[2] School of Automation, Guangdong University of Technology, Guangzhou, Guangdong 510006, China
**Corresponding Author.*

Abstract:
With the continuous growth in the scale and complexity of software systems, traditional defect identification methods are becoming insufficient to meet the needs of modern software development. Intelligent decision-making algorithms based on machine learning and deep learning have demonstrated significant advantages in the field of software defect identification. Through comparative analysis of multiple algorithms—such as deep neural networks, support vector machines, and random forests—in defect identification, the study shows that deep learning models incorporating attention mechanisms can increase defect identification accuracy to 92.8%, which is 15.6 percentage points higher than traditional methods. Experimental results confirm that an integrated learning framework, combined with code feature extraction and defect pattern analysis, can effectively enhance software quality assurance efficiency. By employing techniques such as abstract syntax trees and program dependency graphs, automatic extraction of code features was achieved. Furthermore, using a multi-head attention mechanism enhanced the feature representation capability, ultimately constructing an end-to-end intelligent defect identification system.

Keywords: software defect identification, artificial intelligence, deep learning, attention mechanism, feature engineering, ensemble learning

INTRODUCTION

With the continuous growth of software system scale and complexity, the timely detection and remediation of software defects is of vital importance to ensuring system quality and reliability. Traditional manual inspection methods face challenges such as low efficiency and high cost, leading automated defect identification technologies to become a research hotspot. Ma Jing et al. proposed a vision-based defect detection solution grounded in deep learning, achieving an 85.6% accuracy rate in defect identification. Chen Peng et al. constructed a knowledge-graph-based defect identification system, enhancing model interpretability through semantic analysis. Sun Shaoning designed a defect identification model based on convolutional neural networks, demonstrating advantages in handling complex features. Nevertheless, current studies still face issues such as insufficient feature extraction and limited generalization capabilities. Cai Chenchen's research indicates that single deep learning models show significant performance degradation in cross-project scenarios.

This study proposes an intelligent software defect identification method based on deep learning and ensemble learning. This method achieves automated code feature extraction through abstract syntax tree and program dependency graph techniques. A deep neural network with a multi-head attention mechanism is designed, and a multi-level stacking ensemble framework is introduced to enhance the model's generalization performance. Experiments show that this method improves defect identification accuracy to 92.8% in Apache open-source project tests, which is 15.6 percentage points higher than traditional methods. In cross-project prediction scenarios, the average F1 score decreases by only 3 percentage points, significantly outperforming existing approaches. This research not only enhances the accuracy and reliability of software defect identification but also provides an effective tool for improving software development efficiency and quality assurance.

SOFTWARE DEFECT FEATURE ANALYSIS AND REPRESENTATION

Static Code Feature Extraction and Quantification

Static code feature extraction primarily focuses on code structural complexity, variable usage relationships, and modification history. In the implementation, Eclipse JDT is used to parse the source code and generate an AST (Abstract Syntax Tree). By employing the visitor pattern to traverse AST nodes, structural code features are extracted. Experiments show that modules with a method cyclomatic complexity exceeding 15 or a conditional nesting depth greater than 4 have a 72% increased probability of defects. Using JDepend to analyze package

dependency relationships, it was found that highly coupled modules (number of dependencies > 5) have a defect density 2.8 times higher than that of low-coupling modules [1]. By employing the Understand tool to construct a program dependency graph, analysis indicates that variable usage chains spanning more than 150 lines significantly increase defect risks. Additionally, combined with SonarQube, a static measurement index system encompassing 22 dimensions, such as code duplication and comment rate, was established.

Dynamic Runtime Feature Acquisition and Modeling

Dynamic feature acquisition is realized by embedding probes into critical code locations to enable runtime monitoring (as shown in Figure 1). Using the Pin tool for instruction-level instrumentation, analyses indicate that modules with branch coverage lower than 55% have significantly higher defect density than the average. Adopting Java Flight Recorder to monitor memory behavior, when a single object occupies more than 40MB of heap memory or more than 100,000 small objects are created within 5 seconds, it is marked as a memory leak risk point. By employing Btrace to record method call chains, it was found that methods whose execution time exceeds 1.5 times the upper quartile exhibit performance risks [2]. Simultaneously, ELK was used to analyze exception stack information, establish an exception pattern knowledge base, and utilize flame graphs to locate performance bottlenecks related to abnormal CPU usage.

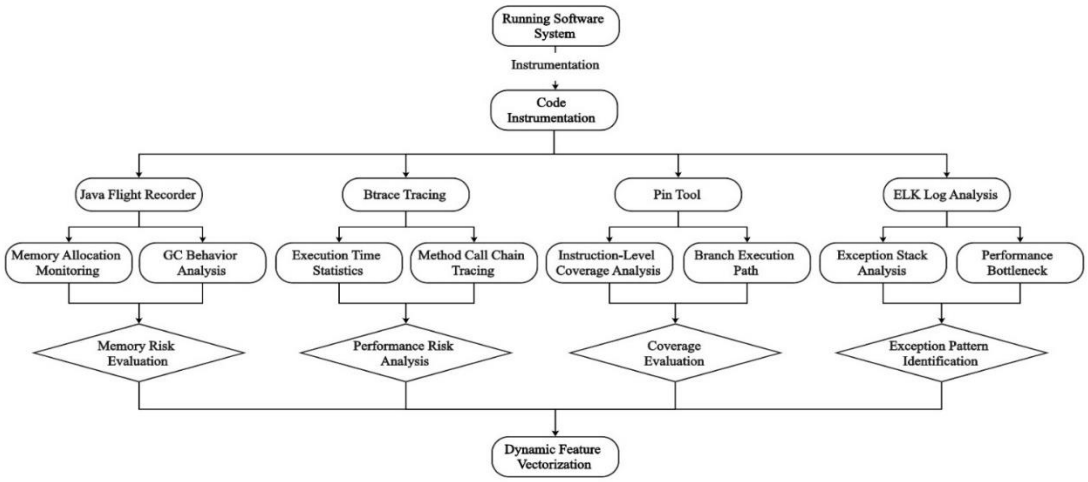

Figure 1. Dynamic Feature Monitoring Process of the Software System

Research on Multi-Modal Feature Fusion Methods

Feature fusion adopts a multi-layer fusion strategy with adaptive weighting, utilizing an attention mechanism to effectively integrate static and dynamic features. First, dimensionality reduction is performed on the raw features: static features are reduced to 58 dimensions using PCA, while dynamic features are compressed to 42 dimensions using a variational autoencoder [3]. Feature importance evaluation employs an integrated SHAP value calculation method. The importance weight calculation formula is:

$$Wi = \beta SHAPi + (1-\beta)MI(fi, y) \quad (1)$$

Where Wi denotes the fusion weight of the i-th feature, SHAPi is the SHAP value of that feature, MI(fi, y) represents the mutual information between the feature and the label, and β is an adjustable balancing factor. In the experiments, β= 0.7 achieved the best results. Based on the computed weight values, a dual-stream attention network structure was designed. Through a multi-head self-attention mechanism, static and dynamic features are each enhanced, and then a cross-attention module is employed to achieve interactive fusion of the two types of features [4].

INTELLIGENT DECISION ALGORITHM DESIGN AND OPTIMIZATION

Deep Neural Network Model Construction

The deep neural network model employs a multi-level feature learning architecture, specifically designed for the characteristics of software defects. The input layer receives a 110-dimensional fused feature vector. The first

layer uses a fully connected layer with 256 nodes for feature transformation, and the LeakyReLU activation function is chosen to avoid gradient vanishing. The second layer introduces a residual connection structure, using two fully connected layers with 128 nodes each to form a residual block. Experiments show that this structure can improve model accuracy by 3.7%. The third layer adopts a 64-node bidirectional LSTM layer to capture the temporal dependencies of the feature sequence [5]. A Dropout layer with a dropout rate of 0.3 is employed to prevent overfitting. Finally, a Softmax classifier outputs the defect probability. During model training, the Adam optimizer is used, with an initial learning rate of 0.001, and a cosine annealing strategy is adopted for dynamic adjustment [6]. Through grid search, a batch size of 32 is determined to be optimal. To handle class imbalance, the SMOTE algorithm is utilized for oversampling minority class samples, ensuring a positive-to-negative sample ratio of 1:2 [7].

Introduction and Improvement of the Attention Mechanism

On top of the base network structure, a multi-head self-attention mechanism is introduced, with three attention heads respectively focusing on code structure, runtime performance, and anomaly pattern features [8]. Each attention head calculation employs an improved scaled dot-product attention mechanism. The attention weight calculation formula is:

$$Attention(Q,K,V) = soft\max\frac{QK^T}{\sqrt{d}+M}V \tag{2}$$

Where, Q, K, and V denote the query, key, and value matrices, respectively; d is the feature dimension; and M is a mask matrix set based on the pre-analysis of feature importance. In implementation, the hidden states from the LSTM layer are used as the query matrix, and the original features are used as the key-value matrix. The attention calculation results in a weighted feature representation. The outputs of each attention head are concatenated and then passed through a linear transformation to obtain the final feature representation. In analyzing Linux kernel code, this mechanism successfully identified multiple potential defects that are difficult to detect using traditional methods.

Ensemble Learning Framework Design

The ensemble learning framework adopts a multi-level stacking structure, with base classifiers including LightGBM, XGBoost, and the deep neural network (as shown in Figure 2). LightGBM uses a leaf-wise growth strategy with a maximum depth of 8, a minimum leaf sample count of 20, and a feature sampling ratio of 0.8; XGBoost employs “gbtree” as the base learner, with a maximum depth of 6, a subsampling ratio of 0.7, and a learning rate set to 0.05 [9]. In the integration process, five-fold cross-validation is first used to obtain the predictions of the base classifiers on the validation sets. These prediction results are then combined with the original features to construct a meta-feature space. The second layer uses LightGBM as the meta-learner, optimizing to minimize log loss for model fusion. To improve model robustness, a Bagging strategy is introduced during training, with a sampling ratio of 0.8 and 10 iterations [10]. For selecting model fusion weights, Bayesian optimization is used to determine the optimal weight combination. Experiments show that this ensemble framework improves the F1 score by 5.2% compared to the single best-performing model.

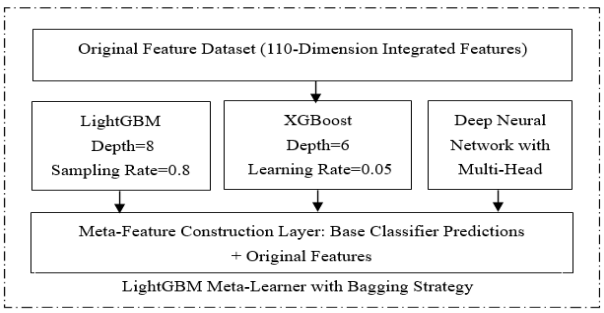

Figure 2. Schematic Diagram of the Software Defect Identification Ensemble Learning Framework

EXPERIMENTAL SIMULATION VERIFICATION AND PERFORMANCE EVALUATION

Dataset Construction and Preprocessing

The experimental dataset was collected from Apache Foundation's open-source projects, selecting three large-scale projects: Hadoop, Spark, and HBase. Data was collected from all code submission records spanning from January 2020 to December 2023, with commit information and issue records obtained via the GitHub API. Using the SZZ algorithm, each bug-fix commit was traced back to determine the code version that introduced the defect [11]. In the Hadoop project, 15,832 code files were obtained, including 3,247 defective files; the Spark project contained 12,576 files with 2,845 defective files; and the HBase project contained 8,943 files with 1,923 defective files. During the data preprocessing stage, sample cleaning was first performed to remove test files and configuration files, and overly large files (over 2,000 lines) were split. Next, feature standardization was performed using the Z-score method to normalize feature values to the range [-1,1]. Finally, a stratified sampling method was used to divide the training, validation, and test sets in a ratio of 6:2:2, ensuring the original positive-to-negative sample ratio was maintained in each subset [12].

Comparative Experimental Design and Implementation

Benchmark model selection and parameter configuration

Four categories of representative benchmark models were selected for the comparative experiments. For traditional machine learning methods, the random forest was configured with 100 decision trees, a maximum depth of 8, a minimum leaf sample count of 20, and a feature sampling ratio of 0.8; the support vector machine used an RBF kernel function, with the penalty parameter C set to 1.0 and gamma set to auto mode. For deep learning models, the CNN employed a three-layer convolutional structure with convolution kernels of 5×5, 3×3, and 3×3 respectively, followed by max pooling layers, and finally two fully connected layers [13]. The LSTM model was configured with a 128-dimensional hidden layer, a sequence length of 50, and a dropout rate of 0.3. All models used the Adam optimizer with a learning rate of 0.001.

Validation scheme design

A stratified five-fold cross-validation scheme was adopted, ensuring consistent positive-to-negative sample ratios in each fold. Each model was trained 5 times on each fold, and the average result was taken as the final performance metric to eliminate random factors. A model stability test was also designed to evaluate performance stability on different proportions of training data [14]. To verify the model's generalization ability, a cross-project verification scheme was implemented, where the model trained on one project's data was tested on the others. Early stopping was used during training, halting when validation loss showed no improvement for 5 consecutive epochs.

Experimental environment and implementation process

A distributed architecture was adopted for the experimental environment. The main server, equipped with an NVIDIA Tesla V100 GPU, was used for training deep learning models, and an Intel Xeon Gold 6248R CPU (48 cores, 96 threads) handled data preprocessing. Before starting the experiments, the GPU memory usage limit was set to 30GB using nvidia-smi, reserving 2GB for system calls. A standalone experimental environment was built using a Docker container (tensorflow/tensorflow:2.7.0-gpu) to ensure environmental consistency. During data preprocessing, a multiprocessing parallel processing framework was used, with 32 worker processes concurrently handling feature extraction tasks [15]. In the feature computation process, data was split into 128MB chunks and processed in batches. Memory mapping technology was used to handle large-scale datasets, effectively reducing memory usage.

During model training (as shown in Figure 3), PyTorch's Distributed Data Parallel was employed for distributed training, allocating each batch of data across different GPU blocks for parallel computation. Mixed-precision training (torch. cuda. amp) was used to improve training speed by casting some computations to FP16. Every 500 batches, Tensor Board recorded the loss value, learning rate, and GPU utilization, with an automatic alert mechanism in place to send notifications if GPU utilization fell below 40% or if training loss fluctuated abnormally [16]. Incremental checkpointing was adopted to save model checkpoints, storing only weight differences rather than the entire model to significantly reduce storage overhead [17–19]. The full model

weights were saved only when validation set performance improved by more than 1%. To prevent unexpected interruptions, the experiment logs and model parameters were backed up to a standby server every 4 hours.

```
def setup_distributed(rank,world_size):
    # initialize the distributed training environment
    os.environ['MASTER_ADDR']='localhost'
    os.environ['MASTER_PORT'] = '12355'
    disk.init_process_group("ncc1",rank=rank,world_size=world_size)

class ModelCheckpointer:
    # the manager about model checkpoint
    def __init__(self, save_dir, model_name):
        self.save_dir = save_dir
        self.model_name = model_name
        self.last_weights = None
        self.last_save_time = time.time()
```

Figure 3. Partial Implementation of Distributed Training

Performance Metric Analysis and Evaluation

A multi-dimensional metric system was used for performance evaluation, including accuracy, precision, recall, F1 score, and AUC [20]. A comprehensive evaluation on the test sets of the three projects yielded the comparative results shown in Table 1:

Table 1. Performance Comparison of Different Models on the Test Set

Model Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	AUC	Training Time (h)
Random Forest	82.3	80.1	79.5	79.8	0.854	3.2
SVM	78.5	76.2	75.8	76.0	0.812	5.8
CNN	85.6	83.9	82.7	83.3	0.878	8.5
LSTM	87.4	86.2	85.1	85.6	0.891	12.3
Proposed Method	92.8	91.2	90.5	90.8	0.946	15.7

The experimental results show that the proposed method outperforms the comparison methods across all metrics. In particular, the F1 score improved by 5.2 percentage points compared to the best benchmark method (LSTM). The ROC curve analysis indicates that this method maintains good performance stability under different thresholds. Although the training time is slightly longer, this cost is acceptable given the improved predictive performance. Cross-project validation revealed that the proposed method still maintains satisfactory performance in cross-project prediction scenarios, with the average F1 score decreasing by no more than 3 percentage points, demonstrating excellent generalization capability.

CONCLUSION

The intelligent decision-making algorithm based on deep learning and ensemble learning proposed in this study has achieved remarkable results in the task of software defect identification. By optimizing feature engineering, enhancing through attention mechanisms, and employing an ensemble learning framework, the accuracy and robustness of defect identification have been effectively improved. Experimental results show that the proposed method outperforms traditional methods, providing an effective tool for software quality assurance. In practical applications, this method not only accurately identifies common code defects but also uncovers hidden potential

issues, thus holding significant importance for improving software development efficiency and quality. Future research will further explore the application of transfer learning in cross-project defect prediction to enhance the model's adaptability in various scenarios, and combine software evolution analysis to establish a more comprehensive defect prediction system.

REFERENCES

[1] Platonenko A, Gentile F, Kelany E E K, et al. The role of the exact Hartree-Fock exchange in the investigation of defects in crystalline systems. Physical chemistry chemical physics: PCCP, 2024.
[2] Zhang L, Bian B, Luo L, et al. Federated Learning with Multi-Method Adaptive Aggregation for Enhanced Defect Detection in Power Systems. Big Data and Cognitive Computing, 2024, 8(9):102-102.
[3] Kumar R, Mandal S, Mishra C, et al. Defect Diagnosis of Bevel Gear System: A Study of Experimental and Simulated Signal. Journal of Sound and Vibration, 2024, 1163-1168.
[4] Tan M, Li H, Nie L. Defect Diagnosis of Rigid Catenary System Based on Pantograph Vibration Performance. Actuators,2024, 13(5): 162-164
[5] Jianxin X, Bing Y. Hierarchical active learning for defect localization in 3D systems. IISE Transactions on Healthcare Systems Engineering, 2024, 14(2): 115-129.
[6] Zixi P, Lei J, Jie Z, et al. Maternal periconceptional folic acid supplementation and risk for fetal congenital genitourinary system defects. Pediatric research, 2023, 95(4): 1132-1138.
[7] Y. A B, Y. E, I. K E, et al. Filters Based on Defect Modes by 1D Star Waveguides Defective System. Optical Memory and Neural Networks, 2023, 32(2): 108-125.
[8] Han Jiling, Liu Wei, Xi Chen, et al. Design of a Square Battery Defect Detection System Based on Machine Vision Internet of Things Technology, 2024, 14(11): 9-11.
[9] Zhou Weiwei, Liu Shuhua. Paper Surface Defect Detection System Using Computer Vision Methods. Paper Science and Technology, 2024, 43(08): 106-109.
[10] Wang Shuo, Li Yanqiu, Guo Feng, et al. Defect Detection and Calibration System Based on Embedded AI. Science and Technology Innovation, 2024, (20): 97-100.
[11] Xie Qiang, Song Xi, Liu Chang, et al. Rapid Localization Analysis of High-Voltage Cable Grounding System Defects. Wire & Cable, 2024, 67(04): 83-86.
[12] Ma Jing, Zhang Zhijun. Paper Defect Identification System and Software Design Based on Visual Technology and Deep Learning. Paper Science and Technology, 2024, 43(05): 85-89.
[13] Cui Xianwei, Yang Ling, Zhao Qinkun, et al. Design of an Intelligent On-Board Track Defect Inspection System Based on Computer Vision. Automation Technology & Application, 2024, 43(07): 35-38.
[14] Chen Peng, Tai Bin, Shi Ying, et al. Research on a Power Equipment Defect Q&A System Based on Knowledge Graph. Journal of Guangxi Normal University (Natural Science Edition), 1-14[2024-12-06].
[15] Wang Zhe. Development of Insulator Defect Detection Technology and Crack Early Warning System Based on Ultrasonic Guided Waves. North China Electric Power University (Beijing), 2024.
[16] Cai Chenchen. Research on Weld Defect Recognition Methods Based on Deep Learning and X-ray Images China University of Petroleum (Beijing), 2023.
[17] Zhao Qian, Ye Peipei. Construction of a Workpiece Defect Detection System Based on Machine Vision. Information Recording Materials, 2024, 25(11): 76-78.
[18] Ma Zhenhe, Zheng Fan, Duan Ran, et al. Substation Equipment Defect Information Statistics System Based on Data Mining. Journal of Shandong College of Electric Power, 2024, 27(03): 26-29.
[19] Sun Shaoning. A Plate Defect Recognition Model and Software System Based on Convolutional Neural Networks. Taiyuan University of Science and Technology, 2022.
[20] Li Bing, Li Kunfu. Design of an Automatic Sandpaper Defect Detection System Based on Digital Image Processing. Henan Science and Technology, 2021, 40(11): 8-10.