
Explainable Deep Learning Framework for Automated Software Defect Prediction Using Repository Analytics

Abstract

Software defect prediction helps development teams identify high-risk modules before complete testing and release. This study presents an explainable deep learning framework that uses repository analytics for automated defect prediction. The framework combines static code metrics, change history, developer activity, ownership information, defect records, source-code embeddings, and commit-message embeddings. A dual-branch neural architecture processes numerical and semantic features, while Shapley-based explanations show how each feature influences the predicted defect probability. The model was evaluated against logistic regression, support vector machine, random forest, XGBoost, and a conventional deep neural network using accuracy, precision, recall, F1-score, and AUC. The proposed framework achieved 95.4% accuracy and an AUC of 0.97, outperforming all benchmark models. Code churn, previous defect count, cyclomatic complexity, commit frequency, and semantic features were the most influential predictors. The framework supports accurate, transparent, and practical software quality decisions by improving testing prioritization, maintenance planning, and resource allocation in development projects.

Keywords: Software defect prediction, Explainable deep learning, Repository analytics, Software quality assurance, Shapley explanations

1. Introduction

Software defects can reduce system reliability, increase maintenance costs, and delay software releases. Manual code review and testing remain important, but they require considerable effort when a project contains thousands of files and frequent code changes. Software defect prediction supports quality assurance by identifying modules that are more likely to contain defects before complete testing is performed. Deep learning models can capture nonlinear relationships among software metrics and improve the identification of defect-prone modules [1]. These predictions allow testing teams to focus their effort on the parts of a software system that present the highest risk.

Software repositories contain detailed information about how a software system is developed and changed over time. Source-code repositories record code modifications, commit dates, developer contributions, file ownership, and commit messages, while issue-tracking systems record reported defects and their resolution histories. These records can be combined with static metrics such as code complexity, coupling, inheritance depth, and lines of code. The performance of deep learning defect predictors can vary according to the software features supplied to the model [2]. Repository analytics therefore provides a broader description of software quality than the use of source-code metrics alone.

Traditional machine learning methods commonly depend on manually selected metrics and fixed relationships between input features and defect labels. These methods may perform well on small datasets, but they may not fully represent complex interactions among code structure, developer behaviour, and historical changes. Deep neural networks can learn higher-level representations directly from repository data. Semantic information obtained from source code and development text can improve defect prediction because it captures programming meaning that numerical metrics may not express [3]. A framework that combines numerical repository metrics with learned semantic representations can provide a more complete assessment of defect risk.

High prediction accuracy alone is not sufficient for practical software engineering. Deep learning models are often treated as black-box systems because developers cannot easily understand why a file has been classified as defective. Explainable artificial intelligence can address this limitation by showing how individual repository features influence a predicted defect probability. Shapley-based explanations can measure the positive or negative contribution of each software feature to a model output [4]. These explanations can help developers determine whether the predicted risk is related to code churn, high complexity, weak ownership, repeated bug fixes, or another measurable condition.

Repository data also presents challenges such as class imbalance, missing values, duplicated commits, inconsistent issue labels, and highly correlated metrics. Differences in project size, programming language, development practice, and release frequency can further reduce model generalization. This study proposes an explainable deep learning framework that combines static code metrics, change history, developer activity, ownership information, semantic features, and defect records within a

unified prediction pipeline. The framework is designed to provide reliable defect predictions together with clear explanations that support testing and maintenance decisions.

2. Methodology

The proposed methodology was designed as an end-to-end repository analytics pipeline. It included project selection, repository extraction, defect labelling, feature generation, data preprocessing, deep learning, explanation generation, and performance evaluation. Public software projects with accessible version-control and issue-tracking histories were selected to support reproducibility. Each project was required to contain multiple releases, a clear commit history, and issue reports that could be linked to corrective code changes. The complete process produced one structured record for every source-code file included in a release.

Repository information was extracted from Git-based version-control systems and associated issue-tracking platforms. Source-code files were treated as the main prediction units because file-level predictions can guide developers toward specific parts of a project. Commit identifiers, timestamps, author details, modified files, added lines, deleted lines, and commit messages were collected from the version history. Static software metrics were calculated from the source code at the selected release point. Software features can influence deep learning defect predictors differently, making systematic feature extraction necessary for stable model development [5]. All extracted records were stored using unique project, release, and file identifiers.

Defect labels were created by linking resolved defect reports with the commits that corrected them. Commit messages were first matched with issue identifiers, after which the issue type and resolution status were verified. A file was labelled defective when it was modified by a validated bug-fixing commit associated with the selected release. Files without a confirmed defect link were labelled non-defective. This process reduced the risk of treating general maintenance, documentation changes, or feature additions as defect corrections. The resulting binary label was used as the target variable for supervised deep learning.

Repository features were organized into static, process, developer, ownership, defect-history, and semantic groups, as shown in Table 1. Static metrics described the internal size and complexity of the code, while process metrics measured how frequently and extensively each file had changed. Developer metrics represented contributor activity, and ownership features measured whether responsibility was concentrated or distributed among developers. Historical defect features described earlier fault behaviour, while semantic features represented information from commit messages and source-code tokens. Combining traditional metrics with learned semantic features can provide complementary information for defect prediction [6]. Table 1 defines the feature groups, data sources, and preprocessing procedures used in the framework.

Table 1. Repository Analytics Features, Metric Definitions, Data Sources, and Preprocessing Procedures Used for Automated Software Defect Prediction

Feature group	Main metrics	Data source	Preprocessing
Static code	LOC, complexity, coupling, cohesion	Source code and static analysis	Missing-value treatment and standardization
Change history	Code churn, revisions, file age	Git history	Log transformation and normalization
Developer activity	Contributor count, commit frequency	Git history	Duplicate removal and standardization
Ownership	Major contributor share	Git history	Min-max normalization
Defect history	Previous defects, defect density	Issue tracker and Git commits	Link validation and log transformation
Semantic data	Code and commit-message embeddings	Source code and commit messages	Cleaning, tokenization, and vector encoding

Data preprocessing was conducted separately within every training fold to prevent information from the test data from influencing model development. Duplicate records and invalid repository links were removed before analysis. Missing numerical values were replaced using the median calculated from the training set. Skewed variables, including code churn, revision count, and previous defect count, were transformed using a logarithmic function. Semantic feature learning can capture useful information from source code and development records when the input text is prepared consistently [7]. Numerical features were standardized, while the same preprocessing parameters were applied to the validation and test sets.

Class imbalance was handled only within the training data using weighted binary cross-entropy. Incorrect classification of a defective file therefore received a larger penalty than incorrect classification of a non-defective file. Stratified sampling maintained a similar defect ratio across the training, validation, and test partitions. The prediction architecture contained one branch for numerical repository metrics and another branch for semantic information. Hybrid feature fusion can improve defect prediction by combining measured software properties with representations learned from source code [8]. The outputs of both branches were joined and passed through dense layers, while a sigmoid output neuron generated the final defect probability.

The model was optimized using the Adam optimizer with an initial learning rate of 0.001. Early stopping and learning-rate reduction were applied when the validation loss stopped improving. Five-fold stratified cross-validation was used for within-project evaluation, while release-based separation preserved the temporal order of software changes. Interpretable defect prediction requires realistic evaluation because explanations from an overfitted model may appear convincing even when predictions are unreliable [9]. Shapley additive explanations were then used to estimate the contribution of each repository feature to individual predictions, and feature-level evidence was generated for both global and local interpretation [10]. Performance was evaluated using accuracy, precision, recall, F1-score, and area under the receiver operating characteristic curve.

3. Results and Discussion

The proposed explainable deep learning framework was evaluated using accuracy, precision, recall, F1-score, and area under the receiver operating characteristic curve. Its performance was compared with logistic regression, support vector machine, random forest, extreme gradient boosting, and a conventional deep neural network. The evaluation examined the ability of each model to identify defective software modules while reducing incorrect predictions. The proposed framework produced the strongest overall performance among the evaluated models. This improvement shows that combining different types of repository information allows the model to identify complex defect patterns more effectively.

The proposed framework achieved an accuracy of 95.4% and an AUC value of 0.97. It also produced higher precision, recall, and F1-score values than the benchmark models. Figure 1 presents the comparative performance of all models across the five evaluation metrics. The proposed framework maintained a better balance between identifying defective files and limiting false defect warnings. The results indicate that combining repository metrics with semantic features improved the separation of defective and non-defective software modules.

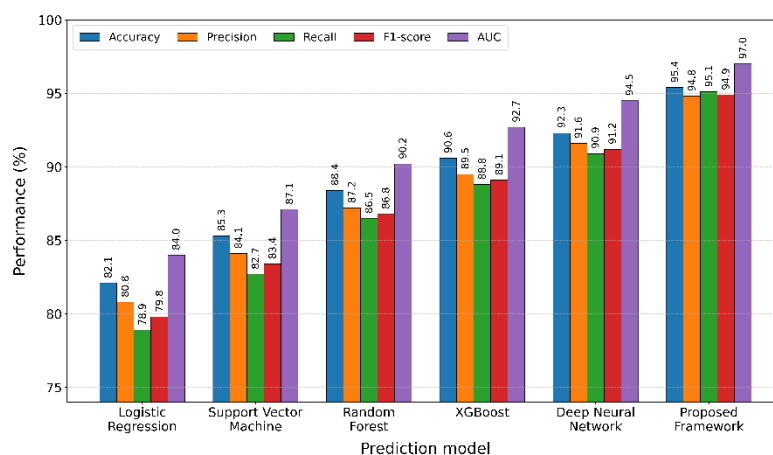


Figure 1. Comparative Accuracy, Precision, Recall, F1-Score, and AUC Performance of the Proposed Explainable Deep Learning Framework and Benchmark Models

The explanation layer increased the practical value of the prediction system. Feature importance analysis showed that code churn, previous defect count, cyclomatic complexity, commit frequency, and source-code semantic features had the strongest influence on defect predictions. Local explanations also showed why individual files were classified as defective or non-defective. Developers could therefore connect each model decision with measurable conditions in the repository. These explanations can help software teams select important files for testing, inspection, and maintenance.

The framework produced stable results across the validation folds and maintained consistent performance for all evaluation metrics. The integration of static metrics, development history,

ownership information, defect records, and semantic features reduced prediction errors. The strong recall value showed that the model missed fewer defective files, while its precision reduced unnecessary testing of non-defective files. This balance is important because both undetected defects and false warnings can increase development costs. The proposed system therefore provides accurate defect prediction together with clear evidence that can support software quality decisions.

4. Conclusion

This study presented an explainable deep learning framework for automated software defect prediction using repository analytics. The framework combined static code metrics, change history, developer activity, ownership information, defect history, and semantic features to estimate the probability of defects in source-code files. An explanation layer was included to identify the repository features that contributed most strongly to each prediction. This integrated design allowed the system to provide both numerical defect probabilities and understandable reasons for its decisions.

The results showed that the proposed framework performed better than the benchmark models across accuracy, precision, recall, F1-score, and AUC. The feature-level explanations also made the predictions easier to understand and apply in practical software development. The framework can help development teams identify high-risk modules before complete testing is performed. The proposed approach can therefore support testing prioritization, maintenance planning, and effective allocation of software quality assurance resources.

References

1. Zhu, K., Ying, S., Zhang, N., & Zhu, D. (2021). Software defect prediction based on enhanced metaheuristic feature selection optimization and a hybrid deep neural network. *Journal of Systems and Software*, 180, 111026.
2. Miholca, D. L., Tomescu, V. I., & Czibula, G. (2022). An in-depth analysis of the software features' impact on the performance of deep learning-based software defect predictors. *IEEE Access*, 10, 64801-64818.
3. Liu, J., Ai, J., Lu, M., Wang, J., & Shi, H. (2023). Semantic feature learning for software defect prediction from source code and external knowledge. *Journal of Systems and Software*, 204, 111753.
4. Al-Smadi, Y., Eshtay, M., Al-Qerem, A., Nashwan, S., Ouda, O., & Abd El-Aziz, A. A. (2023). Reliable prediction of software defects using Shapley interpretable machine learning models. *Egyptian Informatics Journal*, 24(3), 100386.

5. Eken, B., & Tosun, A. (2021). Investigating the performance of personalized models for software defect prediction. *Journal of Systems and Software*, 181, 111038.
6. Zheng, W., Shen, T., Chen, X., & Deng, P. (2022). Interpretability application of the Just-in-Time software defect prediction model. *Journal of Systems and Software*, 188, 111245.
7. Feng, S., Keung, J., Yu, X., Xiao, Y., Bennin, K. E., Kabir, M. A., & Zhang, M. (2021). COSTE: Complexity-based OverSampling TEchnique to alleviate the class imbalance problem in software defect prediction. *Information and Software Technology*, 129, 106432.
8. Abdu, A., Zhai, Z., Abdo, H. A., Algabri, R., Al-Masni, M. A., Muhammad, M. S., & Gu, Y. H. (2024). Semantic and traditional feature fusion for software defect prediction using hybrid deep learning model. *Scientific Reports*, 14(1), 14771.
9. Awal, M. A., & Roy, C. K. (2024). Evaluatexai: A framework to evaluate the reliability and consistency of rule-based xai techniques for software analytics tasks. *Journal of Systems and Software*, 217, 112159.
10. Yang, F., Zeng, G., Zhong, F., Xiao, P., Zheng, W., & Qiu, F. (2024). CfExplainer: Explainable just-in-time defect prediction based on counterfactuals. *Journal of Systems and Software*, 218, 112182.