Enhancing Software Quality with Feature-Aware Defect Prediction Models

Niaz Ashraf Khan
Department of Computer Science and Engineering
BRAC University
Dhaka, Bangladesh
niaz.ashraf@bracu.ac.bd

Md. Ferdous Bin Hafiz
Department of Computer Science and Engineering
Southeast University
Dhaka, Bangladesh
ferdous.binhafiz@seu.edu.bd

Md. Mazharul Islam
Department of Electrical and Computer Engineering
North South University
Dhaka, Bangladesh
mazharul.islam1@northsouth.edu

Md. Aktaruzzaman Pramanik
Department of Computer Science and Engineering
University of Liberal Arts Bangladesh
Dhaka, Bangladesh
aktaruzzaman.pramanik@ulab.edu.bd

***Abstract*— Accurately predicting software defects is essential for maintaining high software quality and ensuring efficient allocation of testing efforts. This research conducts a comparative evaluation of various machine learning and deep learning techniques for defect prediction using the NASA JM1 dataset. The study investigates the performance of six conventional machine learning models alongside a feedforward neural network. Among the traditional models, Random Forest yielded the best results with an F1-score of 0.391, while the neural network achieved a closely matching F1-score of 0.381. An analysis of feature importance indicated that metrics related to cyclomatic complexity and lines of code were among the most influential factors in identifying defective modules. These outcomes provide practical guidance for selecting suitable models and prioritizing features, contributing to more effective strategies in software defect prediction and overall quality assurance.**

Keywords—Software defect prediction, machine learning, neural networks, ensemble methods, software quality, feature importance.

I. INTRODUCTION

Software Quality Assurance (SQA) plays a vital role in modern software engineering, particularly as contemporary systems grow in scale, complexity, and societal impact. Today's software applications are deeply embedded in critical domains ranging from finance and healthcare to transportation and communication, making reliability and defect prevention more crucial than ever. Software defects, when left undetected, can result in substantial economic losses, erosion of stakeholder trust, and in extreme cases, compromise human safety. Although traditional quality assurance practices—such as static code analysis, manual code reviews, and testing—have long been part of the development lifecycle, they are increasingly viewed as resource-intensive and sometimes inadequate for today's fast-paced, agile environments. In response to these challenges, the software engineering community has been steadily shifting toward data-driven, automated methods that leverage historical data, machine learning, and predictive analytics to proactively manage software quality and reduce the occurrence of defects more efficiently.

Machine learning (ML) has emerged as a powerful tool for identifying software defects, predicting fault-prone modules, estimating development effort, and optimizing testing processes. ML models can learn from historical software data, such as source code metrics, bug reports, and change logs, to make predictions that support informed decision-making. Studies have shown that supervised learning techniques like SVMs, Decision Trees, and NNs can outperform traditional heuristics in identifying code quality issues [1][2].

Moreover, ML models allow real-time monitoring and feedback for developers, ensuring proactive quality control rather than reactive bug fixing. For instance, recent work has demonstrated how ensemble learning methods can effectively predict defects across multiple projects with minimal domain adaptation. Given the growing availability of software repositories (e.g., GitHub, GitLab) and defect datasets (e.g., PROMISE, NASA), ML offers significant opportunities to enhance software quality systematically.

The integration of ML techniques into software engineering practices represents a paradigm shift in defect prediction. While ML approaches to software defect prediction have been explored previously, several technical gaps remain in the literature:

- Limited comparative analysis: Most studies focus on either traditional machine learning or deep learning approaches, with few comprehensive comparisons across multiple algorithm families.
- Insufficient feature importance transparency: There is inadequate exploration of which software metrics most strongly contribute to defect prediction, limiting the actionable insights for developers.
- Class imbalance handling: Real-world software defect datasets typically exhibit significant class imbalance, yet many studies do not adequately address this challenge.

An extensive comparative analysis was conducted on six traditional machine learning algorithms and a NN architecture using the NASA JM1 dataset, offering valuable insights into model selection for defect prediction. SMOTE was used, which helped improve the detection of the less common defect cases. Additionally, the most predictive software metrics were identified and ranked, providing practical guidance for software development teams.

The findings from this study have significant implications for software engineering practices, particularly in resource-constrained environments where optimizing testing efforts can lead to substantial cost savings without compromising software quality.

The structure of this paper is organized as follows: Section II presents a comprehensive review of prior research in software defect prediction, emphasizing both traditional machine learning methods and emerging deep learning-based solutions. Section III elaborates on the research methodology, covering the characteristics of the dataset, preprocessing techniques for handling data quality issues and class imbalance, model development processes, and the evaluation criteria used to assess predictive performance. Section IV details the experimental findings, including a side-by-side comparison of the models' effectiveness and an in-depth analysis of feature relevance to understand their influence on prediction accuracy. Section V offers concluding remarks, summarizing the major insights gained from the study, addressing existing limitations, and outlining potential avenues for future research aimed at improving software defect prediction through intelligent, data-driven approaches. Additionally, the paper reflects on the practical implications of the findings, highlighting how predictive modeling can support proactive quality management in real-world software engineering environments.

II. Related Work

Early approaches relied primarily on statistical methods and simple metrics-based heuristics [3]. The transition to machine learning-based approaches began with traditional algorithms such as logistic regression and decision trees [4].

Menzies et al. [5] conducted one of the pioneering studies using NASA datasets, demonstrating the effectiveness of Naive Bayes classifiers for defect prediction. Subsequently, ensemble methods gained popularity, with Wang et al. [6] showing improved performance using Random Forests and AdaBoost algorithms compared to single classifiers.

Deep learning approaches to defect prediction emerged more recently. Wang et al. [7] implemented a deep belief network that outperformed traditional methods on several public datasets. Li et al. [8] explored convolutional neural networks for defect prediction using code as input, demonstrating promising results but highlighting computational complexity concerns.

The challenge of class imbalance in defect prediction has been addressed through various techniques. Pelayo and Dick [9] compared different sampling methods, including SMOTE, finding significant performance improvements when appropriately addressing class imbalance.

Nam et al. [10] investigated heterogeneous defect prediction, proposing a transfer learning approach that enables effective prediction across different projects with minimal adaptation. Their work emphasized the importance of feature selection and weighting when working with diverse software metrics from different development environments.

Song et al. [11] introduced a novel approach combining genetic algorithms with deep neural networks for software defect prediction. Their hybrid method demonstrated superior performance by optimizing feature selection before feeding data into deep learning models, though at the cost of increased computational complexity. This integration leverages the global search capability of genetic algorithms to enhance the representation power of neural networks. While the approach delivers improved accuracy, it also raises concerns regarding scalability and practical deployment in resource-constrained development environments.

In the domain of explainable AI applications, Castelluccio et al. [12] implemented SHAP (SHapley Additive exPlanations) values to provide transparency into defect prediction models, enabling developers to better understand which code characteristics most significantly contribute to defect proneness. Their work highlighted the need for interpretable models in practical software engineering contexts.

Hafiz et al. [13] demonstrated the value of explainable AI techniques in a related field of malware classification. Their research employed memory analysis with ML and DL algorithms augmented by LIME to enhance malware detection. Their approach with ensemble methods achieved high accuracy, with Random Forest reaching 87.30%, while also providing interpretable insights into model decisions—principles that can be effectively transferred to software defect prediction.

Despite these advances, existing studies present several limitations: (1) most focus on either traditional or deep learning approaches in isolation without comprehensive comparison across algorithm families; (2) many fail to adequately address the persistent class imbalance problem inherent in real-world defect datasets; and (3) there is limited analysis of feature importance to guide software quality improvement efforts. Our work aims to fill these gaps by offering a holistic evaluation framework that considers both predictive performance and practical utility in software engineering contexts.

III. Methodology

Our methodology comprises data preprocessing, feature engineering, model implementation, and evaluation phases. A rigorous experimental framework to ensure reproducible results and fair comparison across different models was applied. Fig. 1 shows the full workflow of the research in great detail.

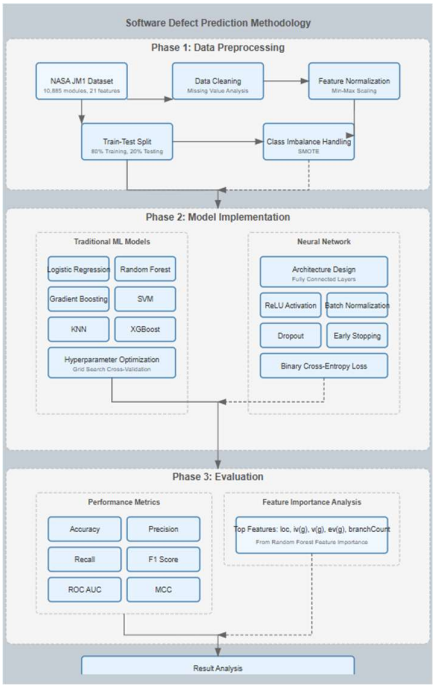

Fig. 1. Software defect prediction methodology

A. Dataset

In this study, we made use of the NASA JM1 dataset, which consists of software metrics and corresponding defect labels for a total of 10,885 modules extracted from a real-world software system. This dataset provides a rich set of twenty-one features that capture various aspects of software complexity, module size, and internal structure. Each module is annotated with a binary label indicating whether it contains one or more defects. A notable characteristic of the dataset is its pronounced class imbalance: out of the total modules, 8,779 (approximately 80.65%) are labeled as non-defective, while only 2,106 (around 19.35%) are defective. This imbalance poses challenges for model training and necessitates appropriate handling strategies. A summary of the dataset's key statistics is presented in Table I for quick reference. Additionally, the interrelationships among the input features are visualized through the correlation matrix shown in Fig. 2, providing insight into potential multicollinearity and feature relevance. This understanding is essential for selecting and engineering features that contribute meaningfully to predictive accuracy, especially in imbalanced learning scenarios.

TABLE I. JM1 DATASET CHARACTERISTICS

Characteristic	Value
Total instances	10,885
Features	21
Defective modules	2,106 (19.35%)
Non-defective modules	8,779 (80.65%)

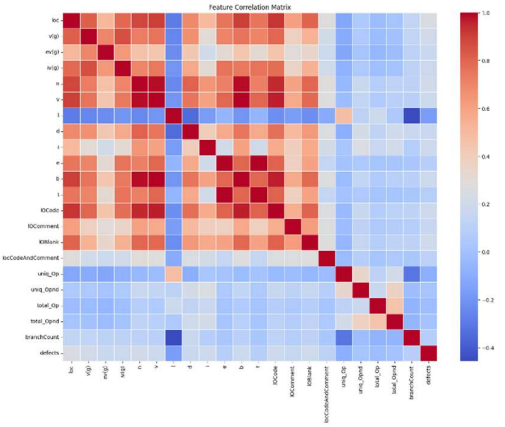

Fig. 2. Feature correlation matrix of the dataset

B. Preprocessing and Feature Engineering

Our preprocessing pipeline consisted of several key steps designed to ensure data quality and model performance:

1) Data Cleaning: Missing values in the dataset were handled using statistical means. The dataset completeness was verified using (1):

$$M_{missing} = \sum_{i=1}^{n}\sum_{j=1}^{m} 1(xij = NA) \quad (1)$$

2) Feature Normalization: To address the varying scales of features, the min-max normalization was applied as shown in (2):

$$x'ij = \frac{xij - \min(x_j)}{\max(x_j) - \min(x_j)} \quad (2)$$

Here, x_{ij} represents the original value of feature j for instance i, and x'_{ij} is the normalized value.

3) Train-Test Split: The dataset was split into training and testing sets in a 4:1 ratio, using stratified sampling to retain the original class distribution across both subsets.

4) Class Imbalance Handling: SMOTE [14] was applied to address class imbalance. For each non-majority class instance l_i, SMOTE generates synthetic instances as shown in (3):

$$l_{new} = l_i + \lambda \cdot (l_{nn} - l_i) \quad (3)$$

Here, l_{nn} is one of the k −nearest neighbors of l_i in the minority class. The number of synthetic instances generated was calculated as shown in (4):

$$N_{synthetic} = |D^{-}_{train}| - |D^{+}_{train}| \quad (4)$$

Here, D^{-}_{train} and D^{+}_{train} represent non-defective and defective instances in the training set, respectively.

C. Model Implementation:

Several ML models were implemented and evaluated, each with its specific mathematical formulation:

1) Logistic Regression (LR): The probability of defect presence is modeled as shown in (5):

$$P(y = 1|x) = \frac{1}{1 + e^{-z}} \quad (5)$$

The model is optimized by minimizing the log-loss as shown in (6):

$$L(\beta) = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)] \quad (6)$$

Here, $p_i = P(y = 1|x_i)$ *and* $y_i \in 0,1$ is the true label.

2) RF: RF constructs multiple decision trees and aggregates their predictions as shown in (7):

$$\hat{y} = \text{mode}(\widehat{y_1}, \widehat{y_2}, \dots, \widehat{y_T}) \quad (7)$$

Here, $\widehat{y_t}$ is the prediction of the t-th tree, and T is the total number of trees. Each decision tree recursively splits the data to maximize information gain as shown in (8):

$$IG(L, a) = H(L) - \sum_{v \in \text{Values}(a)} \frac{|L_v|}{|L|} H(L_v) - \quad (8)$$

Here, $H(L)$ is the entropy of dataset as shown in (9):

$$H(D) = -\sum_{c \in C} p_c \log_2(p_c) \quad (9)$$

Here, p_c being the proportion of instances belonging to class c.

3) Gradient Boosting: Gradient Boosting builds an additive model as shown in (10):

$$F_M(x) = \sum_{m=1}^{M} \gamma_m h_m(x) \tag{10}$$

Here, h_m are weak learners and γ_m are weights. The model is trained iteratively, with each new weak learner fitting the negative gradient of the loss function as shown in (11):

$$h_m = \arg\min_h \sum_{i=1}^{n} \left[r_{i,m-1} - h(x_i)\right]^2 \tag{11}$$

Here, $r_{i,m-1} = -\left[\frac{\partial L(y_i, F_{m-1}(x_i))}{\partial F_{m-1}(x_i)}\right]$ represents the residual error.

4) Support Vector Machine (SVM): SVM finds the hyperplane that maximizes the margin between classes. For non-linearly separable cases, the kernel trick shown in (12) was used:

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j) \tag{12}$$

Here, ϕ is a feature mapping to a higher-dimensional space.

5) KNN: KNN classifies instances based on the highest voting among neighbors which are closest as shown in (13):

$$\hat{y} = \arg\max_{c \in C} \sum_{i \in N_k(x)} 1_{(y_i = c)} \tag{13}$$

Here, $N_k(x)$ represents the k nearest neighbors to instance x, determined by distance metric as shown in (14):

$$d(x_i, x_j) = \sqrt{\sum_{l=1}^{m} (x_{il} - x_{jl})^2} \tag{14}$$

6) XGBoost: XGBoost optimizes a regularized objective function as shown in (15):

$$\mathcal{L} = \sum_{i=1}^{n} l(y_i, \hat{y}i) + \sum t = 1^T \Omega(f_t) \tag{15}$$

Here, l is the loss function, $\hat{y}_i$ is the prediction, and $\Omega(f_t) = \gamma T + \frac{1}{2}\lambda||w||^2$ is the regularization term.

7) Neural Network (NN): Our neural network model consists of fully connected layers with various regularization techniques as shown in (16):

For each layer l with input $a^{[l-1]}$ and output $a^{[l]}$:

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]} a^{[l]} = g^{[l]}(z^{[l]}) \tag{16}$$

Here, $W^{[l]}$ are weights, $b^{[l]}$ are biases, and $g^{[l]}$ is the activation function, specifically ReLU for hidden layers.The model is optimized using binary cross-entropy loss as shown in (17):

$$L = -\frac{1}{n}\sum_{i=1}^{n} [z_i \log(\hat{z}_i) + (1 - z_i)\log(1 - \hat{z}_i)] \tag{17}$$

D. Evaluation Metrics

To comprehensively evaluate model performance, the following metrics were employed:

1) Accuracy: Accuracy is calculated as shown in (18):

$$Accuracy = \frac{True\ Positives\ and\ Negatives}{Number\ of\ Total\ Samples} \tag{18}$$

2) Precision: Precision is calculated as shown in (19):

$$Precision = \frac{TP}{TP + FP} \tag{19}$$

3) Recall: Recall is calculated as shown in (20):

$$Recall = \frac{TP}{TP + FN} \tag{20}$$

4) F1 Score: F1 score is calculated as shown in (21):

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{21}$$

5) ROC AUC : Auc is calculated as shown in (22):

$$AUC = \int_0^1 TPR(FPR^{-1}(t))dt \tag{22}$$

E. Feature Importance Analysis

Multiple techniques to assess feature importance were employed:

1) Random Forest Importance: Based on mean decrease in impurity (Gini importance) as shown in (23):

$$I_j = \sum_{t=1}^{T} \sum_{n \in S_t} \frac{N_n}{N} \Delta i(s_t, j) \tag{23}$$

Here, $\Delta i(s_t, j)$ is the decrease in impurity for feature j at split s_t.

2) Permutation Importance: Calculated as shown in (24):

$$I_j = \mathcal{L}\left(y, \widehat{y_{\pi_j}}\right) - \mathcal{L}(y, \hat{y}) \tag{24}$$

Here, $\widehat{y_{\pi_j}}$ is the prediction after permuting feature j, and $\mathcal{L}$ is the loss function.

F. Model Training and Hyper-parameter Optimization

For each model, grid search cross-validation was performed to identify optimal hyperparameters as shown in (25):

$$\theta^* = \arg\min_{\theta \in \Theta} \frac{1}{k} \sum_{i=1}^{k} \mathcal{L}\left(D_i^{val}, f_\theta\left(D_i^{train}\right)\right) \quad (25)$$

Here, θ represents model hyperparameters, Θ is the hyperparameter space, k is the number of cross-validation folds, and $\mathcal{L}$ is the loss function evaluated on validation fold D_i^{val} after training on D_i^{train}.

IV. Results and Discussion

A. Model Performance Comparison

Table II presents the performance metrics for all implemented models on the test set.

TABLE II. Model Performance Metrics

Model	Accuracy	Precision	Recall	F1	ROC AUC
LR	0.6266	0.2827	0.6057	0.3855	0.6759
RF	0.7754	0.4110	0.3729	0.3910	0.7234
GBoosting	0.7671	0.3914	0.3682	0.3794	0.7165
SVM	0.7689	0.3669	0.2684	0.3100	0.6179
KNN	0.6325	0.2658	0.5107	0.3496	0.6288
XGBoost	0.7859	0.4290	0.3230	0.3686	0.7089
NN	0.7446	0.3585	0.4062	0.3808	0.6843

The performance comparison reveals several interesting patterns:

RF achieved the highest F1 score (0.3910) among all models, suggesting it provides the best balance between precision and recall for defect prediction. XGBoost exhibited the highest accuracy (0.7859) and precision (0.4290), but with lower recall compared to some other models. Logistic Regression demonstrated the highest recall (0.6057), catching more defects overall, but at the cost of many false positives (low precision of 0.2827). The Neural Network delivered competitive performance (F1 score of 0.3808), close to the best traditional models while offering different trade-offs between precision and recall. SVM showed the lowest F1 score (0.3100), suggesting it may not be well-suited for this defect prediction task. A McNemar's test was used to compare the RF and NN model. The result (p = 0.728) shows no significant difference between them, meaning their performance is statistically similar [15]. This indicates that, despite architectural differences, both models make comparable prediction errors on the same data instances. Such statistical validation reinforces the reliability of the neural network as a viable alternative to traditional ensemble methods like RF.

B. Confusion Matrix:

Fig. 3 and Fig. 4 illustrate the confusion matrices of the better performing machine learning models used for defect detection. Logistic Regression (Fig. 3) shows a fair balance but suffers from a relatively high number of false negatives. RF (Fig. 3) improves on this with more correct predictions for defect cases. XGBoost (Fig. 4) maintains a solid balance between detecting defects and avoiding false alarms. Lastly, the Neural Network (Fig. 4) model achieves high true negatives and decent true positives, showing promise despite some false negatives. Random Forest and XGBoost models demonstrated better balanced performance, with an improved ability to identify true positive samples while maintaining reasonable false positive rates. The Logistic Regression model, while catching more defects overall, produced many false positives, which would lead to inefficient allocation of testing resources in practical scenarios.

Fig. 3. Confusion matrix of Logistic Regression (left) and RF (right)

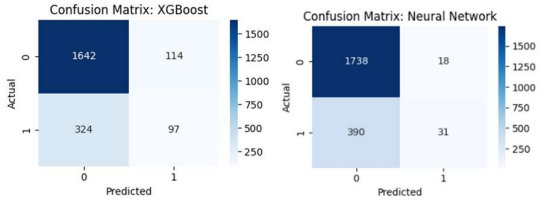

Fig. 4. Confusion matrix of XGBoost and NN

Random Forest and XGBoost models demonstrated better balanced performance, with an improved ability to identify true defects while maintaining reasonable false positive rates. The Logistic Regression model, while catching more defects overall, produced many false positives, which would lead to inefficient allocation of testing resources in practical scenarios.

C. Feature Importance

Feature importance analysis from both Random Forest and XGBoost models highlighted several critical predictors of software defects, as shown in Table III.

TABLE III. Feature Importance

Rank	Feature	Description	Importance
1	loc	Lines of Code	0.1348
2	iv(g)	Cyclomatic Complexity	0.1194
3	v(g)	Cyclomatic Complexity Var.	0.0762
4	ev(g)	Essential Complexity	0.0702
5	branchCount	Number of Branches	0.0557

The XGBoost model showed similar feature importance rankings, with iv(g), ev(g), and loc as the top three features. This consistency across different models reinforces the significance of these metrics for defect prediction.

The dominance of complexity metrics (iv(g), v(g), ev(g)) and size metrics (loc) aligns with software engineering theory, which suggests that more complex and larger code modules are more prone to defects. The high importance of branchCount further supports this, as more conditional branches increase code paths and potential failure points.

D. Neural Network Performance

The neural network architecture demonstrated competitive performance, achieving an F1 score of 0.3808, which is comparable to the best-performing traditional model—Random Forest, which attained an F1 score of 0.3910. Throughout training, the model exhibited steady improvements in both accuracy and AUC, indicating that it was learning effectively from the data. Regularization techniques such as dropout were applied, helping to mitigate overfitting and enhance the model's generalizability. Notably, the model maintained a better trade-off between precision (0.3585) and recall (0.4062) compared to the majority of traditional classifiers. By achieving relatively strong recall without sacrificing too much precision, the neural network offers a practical advantage for identifying defective modules early in the development process. These results suggest that deep learning can complement traditional methods and serve as a viable alternative in software quality assurance pipelines.

E. ROC and Precision-Recall Curves

The ROC curve analysis in Fig. 5 showed that Random Forest achieved the highest AUC (0.7234), followed by Gradient Boosting (0.7165) and XGBoost (0.7089). The neural network model obtained an AUC of 0.6843, indicating good discriminative ability.

Precision-recall curves, which are especially useful for evaluating models on imbalanced datasets, revealed consistent patterns, with Random Forest and XGBoost outperforming others across various threshold values. These models maintained higher precision at increased recall levels, indicating better capability in identifying true positives without a significant rise in false alarms. This highlights their effectiveness in scenarios where correctly detecting the minority class is critical for downstream decision-making.

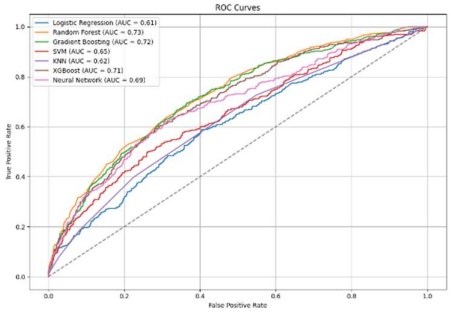

Fig. 5. ROC curve of the models

V. Conclusion

This study compares traditional ML and DL methods for predicting software defects using the NASA JM1 dataset. It was found that ensemble methods (especially RF) performed the best overall, while neural networks also gave promising results.

One of the main takeaways is that ensemble models generally outperform other machine learning techniques for this task, with Random Forest achieving the highest F1 score of 0.3910. It was also found that certain complexity and size metrics, like iv(g), v(g), ev(g), and loc, are key predictors of defects, offering useful insights for software engineers. Additionally, by analyzing feature importance, we made the models more transparent and easier to interpret—an important step toward real-world adoption. We also observed that combining multiple metrics yielded better performance than relying on individual features. These findings suggest that a multifaceted approach can enhance defect prediction and support informed decision-making in software quality assurance.

Nevertheless, the study is not without limitations. We only used one dataset, which may make it harder to apply these findings to other projects. Also, while the results are competitive, there is still room to improve, especially in terms of precision. For future work, we suggest exploring more advanced deep learning models tailored to defect prediction and including a broader range of metrics, such as process and developer-related features.

In summary, predicting software defects is still a tough but important challenge. Our research helps by offering practical guidance on model choice, highlighting key features, and addressing common issues like class imbalance. As software becomes more complex, having reliable tools for defect prediction will be even more crucial for ensuring quality and reliability.

References

[1] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Transactions on Software Engineering*.

[2] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Appl Soft Comput*.

[3] T. J. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*.

[4] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*.

[5] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener, "Defect prediction from static code features: current results, limitations, new approaches," *Automated Software Engineering*.

[6] S. Wang and X. Yao, "Using class imbalance learning for software defect prediction," *IEEE Trans Reliab*.

[7] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th international conference on software engineering*, 2016.

[8] J. Li, P. He, J. Zhu, and M. R. Lyu, "Software defect prediction via convolutional neural network," in *2017 IEEE int conf on software quality, reliability and security (QRS)*, IEEE, 2017, pp. 318–328.

[9] L. Pelayo and S. Dick, "Applying novel resampling strategies to software defect prediction," in *NAFIPS 2007-2007 Annual meeting of the North American fuzzy information processing society*.

[10] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *2013 35th int Conf on Sof Engr (ICSE)*, IEEE, 2013, pp. 382–391.

[11] Q. Song, Y. Guo, and M. Shepperd, "A comprehensive investigation of the role of imbalanced learning for software defect prediction," *IEEE Transactions on Software Engineering*.

[12] M. Castelluccio, G. Poggi, C. Sansone, and L. Verdoliva, "Land use classification in remote sensing images by convolutional neural networks," *arXiv preprint arXiv:1508.00092*, 2015.

[13] M. F. Bin Hafiz, N. A. Khan, Z. Kamal, S. Hossain, and S. Barman, "A Robust Malware Classification Approach Leveraging Explainable AI," in *2024 Int Conf on Int Sys for Cyb (ISCS)*, IEEE, 2024, pp. 1–6.

[14] V. C. Nitesh, "SMOTE: synthetic minority over-sampling technique," *J Artif Intell Res*, vol. 16, no. 1, p. 321, 2002.

[15] T. G. Dietterich, "Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms," *Neural Computation*