International Journal of Emerging Trends in Computer Science and Information Technology
ISSN: 3050-9246 | https://doi.org/10.63282/3050-9246.IJETCSIT-V7I2P137
Eureka Vision Publication | Volume 7, Issue 2, 304-311, 2026

Original Article

A Large Language Model Framework for Early Software Bug Prediction in Software Engineering

Akhil Reddy Duggasani
Independent Researcher,USA.

Received On: 05/04/2026 ***Revised On: 04/05/2026*** ***Accepted On: 12/05/2026*** ***Published On: 18/05/2026***

Abstract - Test managers may anticipate modules that are prone to defects by using software defect prediction models, which helps them produce high-quality products. Improving software quality and reducing expenses throughout the development process depend on early problem discovery. This study proposes an effective software bug detection framework using Random Forest (RF) and DistilBERT models. Advanced preprocessing and feature engineering are used to enhance the prediction performance. To handle the class imbalance, SMOTE is used, and RobustScaler is used for feature normalization. The performance of the proposed models is assessed using accuracy (acc), precision (prec), recall (rec), F1 score (F1), ROC AUC and training time. Experimental results showed that the RF model achieved superior performance, with 99.98% accuracy, predictive capability of 99.39% accuracy, and an F1-score of 99.69%. Near-perfect AUC values are obtained with ROC analysis, ensuring the robustness of both models. The proposed models are also compared against baseline models, including Artificial Neural Network (ANN), Convolutional Neural Network (CNN), Decision Tree (DT), AdaBoost (ADA), and Support Vector Machine (SVM), and are shown to outperform them. Moreover, SHAP-based interpretability analysis is conducted to identify most important factors influencing software defect forecasts, thereby making proposed models transparent and interpretable. Overall, the study provides a very accurate, interpretable and efficient solution for early predicting software flaws and enhancing software quality.

Keywords - Software Quality, Bug Detection, Software Defect Prediction, Machine Learning, SHAP Explainability, Random Forest, Distilbert.

1. Introduction

Software systems are essential to contemporary technological progress, driving vital applications in sectors such as healthcare, finance, and transportation. Nonetheless, software bugs defects or imperfections in code can undermine system reliability, resulting in performance deterioration, security weaknesses, or catastrophic failures[1]. The growing intricacy of software systems has heightened the difficulty of maintaining code quality, rendering human bug identification ineffective and susceptible to errors[2]. Consequently, automated methodologies for predicting software defects have garnered considerable interest in software engineering research. The importance of software bug prediction lies in its ability to lower development expenses, improve software quality and alleviate dangers linked to defective code. Early detection of modules that are prone to defects allows developers to allocate resources effectively, priorities testing efforts, and produce resilient software solutions[3]. In order to assess the possibility of issues like coupling, cyclomatic complexity, or lines of code, conventional bug prediction algorithms often relied on static code metrics[4].

Machine learning (ML) techniques, including NN, DT, RF, and SVM, have exhibited enhanced efficacy in modeling intricate patterns in software metrics, resulting in more precise predictions[5]. When it comes to streamlining software development processes, ML, LLM based methods have demonstrated significant potential[6][7]. These methods are excellent at handling the intricate problems of dynamic software development, such as controlling system complexity and delivering excellent outcomes. ML models may provide solid insights and solutions for software engineering challenges by integrating data from several sources, including repository metadata, historical change logs, documentation, and issue reports[8]. The growing use of ML in software engineering offers creative solutions for complex problems like security flaws and software defects[9]. The motivation for this work is to improve early software defect detection through an automated, accurate approach. The proposed study uses advanced ML and DL techniques to handle complex software data and enhance prediction performance, thereby improving software quality and reducing maintenance effort. This study's primary contributions are as:

- Developed an effective software defect prediction framework using Random Forest and DistilBERT models.
- Improved defect prediction through engineered software quality and risk-related features.
- Enhanced prediction reliability by addressing class imbalance using SMOTE.
- Increased model transparency and interpretability using SHAP analysis.
- Achieved better performance compared to existing baseline ML and DL models.
- Encouraged the early discovery of software bugs to enhance software quality and maintenance effectiveness.

The novelty of this research lies in the combination of advanced feature engineering, data balancing, explainable AI, and hybrid ML-DL approaches for software defect prediction. In contrast to traditional studies based primarily on traditional software metrics and single-model techniques, this study introduces engineered risk-related features to better represent a hidden pattern of defects. Class imbalance issues are overcome by the use of SMOTE, and SHAP-based interpretability increases transparency by revealing the most influential factors in the defect prediction. In addition, an ensemble of Random Forest and DistilBERT models showed excellent predictive power and performance, making the proposed framework a suitable approach for accurate and reliable software bug detection.

1.1. Paper Layout

The remainder of this article is organized as follows: Section II briefly discusses some background research. Section III presents the methodology and the proposed software fault prediction architecture, and Section IV presents the evaluation results, Section V concludes the paper.

2. Literature Review

This study is based on a review and analysis of prior significant research on software dependability and bug prediction.

Fan *et al.* (2026) propose a CAN model that combines the Kolmogorov-Arnold Network's (KAN) nonlinear approximation advantage with TextCNN's ability to extract local features. Experiments using 1880 labeled data samples from the OpenStack project demonstrate that the CAN model outperforms benchmark models such as BERT and CodeBERT, with an accuracy of 0.7492 and an F1-score of 0.8072. Finally, discover that terms like "test" and "api" in bug reports have a significant impact on the LIME, an explainable AI method, prediction of extrinsic problems[10].

Mansour, Said and Kacem (2025) explore a new approach to Software Bug Prediction (SBP) that makes use of a Double-Stacking ensemble model. used data augmentation approaches to strengthen the model's resistance to SBP datasets, such as the NASA JM1 dataset, which frequently exhibits dataset imbalance, with 21 features and 10,885 occurrences. Interestingly, the greatest accuracy of 86.6% was achieved by the highly optimized Double-Stacking model, demonstrating that combining sophisticated optimization with ensemble learning[11].

Xu *et al.* (2025) To solve the aforementioned issues, provide BISP is a hybrid approach that incorporates self-paced ensemble undersampling (SPE), improved subclass discriminant analysis (ISDA), and balanced distribution adaptation (BDA). Experimental results obtained for six classifiers and six cross-project datasets show that contrasted with the cutting-edge transfer learning-based methods TLAP and JDA-ISDA, BISP improves the average balance by 34.8% and 2.3% and improves the average AUC by 26.5% and 8.4%, respectively. Compared with the deep learning approach SRLA, BISP can improve the average balance value by 5.1% [12].

Jasz (2024) presents a solution that attempts to predict method-level bugs while also tracking program dependencies, using an efficient algorithm to achieve greater accuracy. The practical measurements show that the defined approach really outperforms predictions based on traditional metrics in most cases, and with proper filtering, the best-performing RF algorithm according to the F1 can even achieve an improvement of up to 11%[13].

Alsaedi *et al.* (2023) uses ML and natural language processing (NLP) to build an ensemble ML method. Use publicly available datasets from two online software-bug repositories to simulate the suggested paradigm. The simulation results show that proposed model can outperform most existing models, with 90.42% acc without text augmentation and 96.72% acc with text augmentation[14].

Hou *et al.* (2022) generated Call Graph (CG) and the Control Flow Graph (CFG) of each function for each release. To execute the prediction for the specified number of defects, the findings were finally transferred to the Panel Data Model (PDM). The testing results demonstrated that the method beat other prediction techniques by 9.35% to 16.85%, and that the use of CFGM lowered MAE by 5.1% to 27.8% compared to barely using CGM. The forecast of corrected defects might show the quality of the product and help with software engineering quality control[15].

Table I presents a comparative summary of recent ML-based software bug prediction approaches, highlighting datasets, findings, and existing research limitations.

Table 1. Summary of Recent Software Bug Prediction Using Machine Learning Approaches

Author	Key Focus	Dataset	Findings	Limitations / Gaps
Fan et al. (2026)	Developed a CAN model integrating TextCNN and Kolmogorov-Arnold Network (KAN) for bug prediction with explainable AI	1880 labeled samples from the OpenStack project	Achieved 74.92% accuracy and 80.72% F1-score, outperforming BERT and CodeBERT. LIME identified influential keywords such as “test” and “api”.	Limited dataset size and evaluation on a single project reduce generalizability. Computational complexity of hybrid deep models was not discussed.
Mansour, Said and	Proposed a Double-Stacking ensemble model with hyperparameter	NASA JM1 Dataset includes 21	Achieved 86.6% accuracy and demonstrated superiority over single	Focused mainly on one dataset; cross-project validation and

Kacem (2025)	optimization and data augmentation for software bug prediction	characteristics and 10,885 occurrences	classifiers through ensemble learning.	interpretability aspects were limited.
Xu et al. (2025)	Introduced BISP combining BDA, ISDA, and SPE techniques for cross-project defect prediction	Six classifiers and six cross-project datasets	Improved average balance and AUC compared to TLAP, JDA-ISDA, and SRLA methods.	High model complexity and computational overhead; limited discussion on real-time industrial deployment.
Jasz (2024)	Method-level bug prediction using dependency tracking and the Random Forest algorithm	Software project method-level datasets	Improved F-measure by up to 11% compared to traditional metrics and effectively predicted future defects.	Dependency graph construction may increase processing time and scalability issues for large projects.
Alsaedi et al. (2023)	Ensemble ML model using NLP and text augmentation for software bug classification	Mozilla and Eclipse repositories	Accuracy was 90.42% without text augmentation and 96.72% with it.	Performance highly depends on quality of textual data and augmentation techniques; limited focus on code metrics.
Hou et al. (2022)	Bug prediction using Call Graph (CG), Control Flow Graph (CFG), and Panel Data Model (PDM)	Multiple software release versions	Outperformed existing methods by 9.35%–16.85% and reduced MAE significantly with CFG metrics.	Graph construction is computationally expensive and may not scale efficiently for very large software systems.

Research Gap: There are still a number of challenges in software bug prediction research. There are numerous models available, but they are trained and tested on small datasets, limiting their potential to apply to various software projects. Defect datasets are commonly imbalanced, which can bias and lead to inaccurate predictions. The advanced deep learning and graph-based techniques also demand a high computational load and have scalability challenges. Moreover, many methods are not explainable and rely solely on text or code metrics, not effectively combining multiple metric types. Most models are developed for offline analysis and not for real-time CI/CD environment. Hence, lightweight, scalable, interpretable and robust software bug prediction models are needed.

3. Methodology

The methodology comprised loading and pre-processing of a software defect prediction data set. Once the data is determined to be complete and free of duplicate data, EDA is performed to gain insight into the data structure. After features are designed, the data is splitting into training, validation, and test sets (new features are created), and the RobustScaler is used to normalize the data. SMOTE is used to train data in order to correct class imbalance. Lastly, the RF and DistilBERT models are trained for defect prediction and the SHAP analysis is used to identify the key characteristics that influence prediction. Figure 1 present the system pipeline.

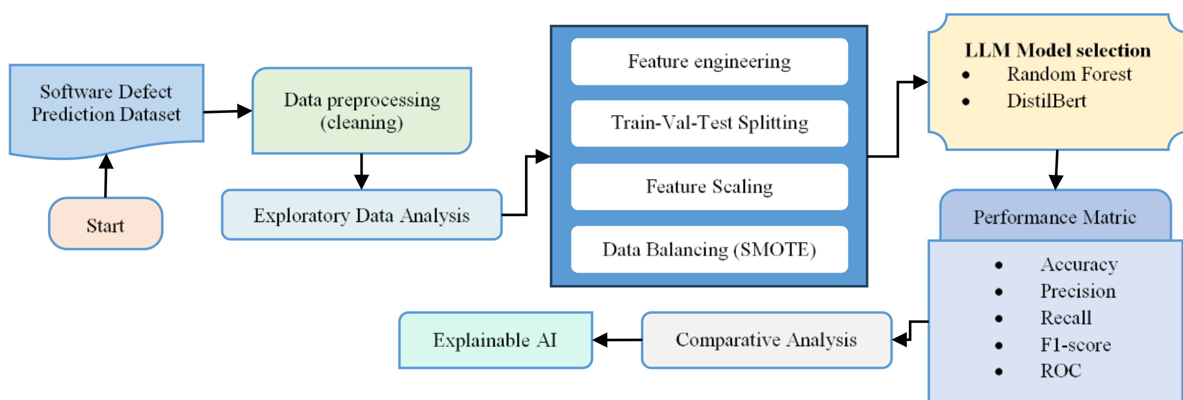


Figure 1. Proposed Flowchart For Software Bug Detection

The overall steps of the proposed methodology are discussed in the next section.

3.1. Data Gathering and pre-processing

The software defect prediction dataset[1] is sourced from Kaggle and consists of 60,000 samples, 23 features associated with software quality and defects. The dataset contains

software quality metrics such as cyclomatic complexity, test coverage, static analysis warnings, previous defects, and code churn that can be used to predict software defects. Data integrity is verified through initial inspection and validation, and the data is loaded before further processing and analysis. The data set is checked for duplicates and missing information, but neither is discovered.

3.2. Exploratory Data Analysis (EDA)

EDA is used to learn more about data properties, feature distributions, correlations, and data imbalance. Multiple statistical summary and visualization techniques are employed to find patterns and facilitate feature engineering and model building.

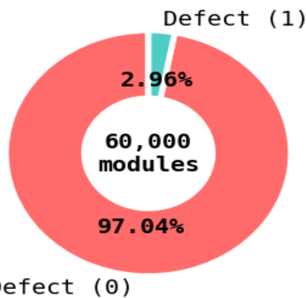


Figure 2. Donut Chart For Target Distribution

The distribution of target classes in dataset is shown in a donut pie chart in Figure 2. The data set is very imbalanced with only 2.96% defective modules and 97.04% non-defective modules. This imbalance could lead to models being skewed towards the majority class, hindering the effectiveness of defect detection. Hence, it is essential to use class balancing methods to enhance model's dependability and forecast accuracy.

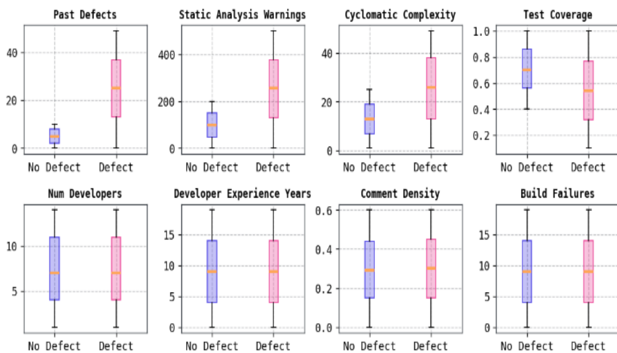


Figure 3. Top Features Distribution By Defect Class

The distribution of significant software metrics for both faulty and non-defective classes is displayed in Figure 3. Past defects, static analysis warnings and cyclomatic complexity tend to be higher for defective modules than for non-defective ones. Defective classes also tend to have higher percentages of features, such as build failures and more developers involved. Defective modules, on the other hand, tend to have lower test coverage, which is a sign of inadequate testing. Overall, it shows distinct differences in behaviour of features in defect and non-defect software modules.

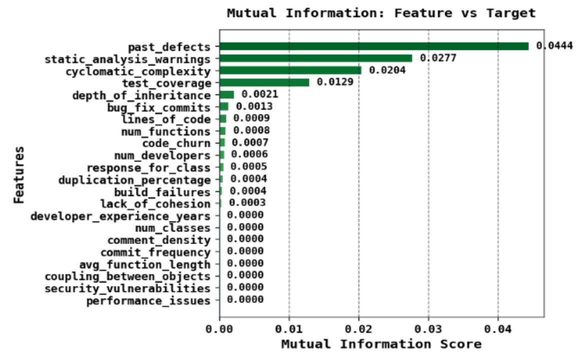


Figure 4. Features and Target Variable Score

Figure 4 shows the mutual information values of software metrics with respect to the target variable for the bug detection model. Past_defects is the most contributing feature, with a score 0.0444, followed by static_analysis_warnings, cyclomatic_complexity and test_coverage. The remaining features have only a minor impact, suggesting that historical defect information and code complexity metrics are important factors in software bug prediction.

3.3. Feature Engineering

Feature engineering is used to create new, significant features from the available data in order to improve the models' capacity for prediction. Other attributes that are added include complexity_score, churn_per_commit, bug_fix_ratio, dev_productivity, risk_index, and comment_bucket, which help to capture the software quality, developer activity, and bug-risk attributes of the software. These engineered features captured hidden patterns in the data and added 6 more features, bringing the total to 29, which helped make the defect prediction models more effective.

3.4. Splitting, scaling and Class Balancing

In order to provide effective training and evaluation of the model, data set is partitioned into three subsets. The training set had 42,000 rows, while the testing and validation sets each had 9,000 rows. This splitting approach helped train the models, tune hyperparameters, and evaluate performance on unseen data. The feature values are then normalized using Robertsdale after splitting to minimize effect of outliers. The median is used by Robertsdale to scale the data, as well as the interquartile range (IQR), which is calculated as Equation. 1:

$$X_{scaled} = \frac{X-\text{Median}(X)}{IQR} \qquad (1)$$

where$IQR = Q_3 - Q_1$, $Q_1$is the first quartile, and $Q_3$is the third quartile. This scaling method improves stability and performance of ML models. To address the problem of class imbalance, the few instances in the minority class are identified relative to the majority class. To overcome this, the training set is oversampled using SMOTE [16]. The purpose of SMOTE is to create synthetic samples for minority class in order to improve model's prediction accuracy for software flaws and assist balance the class distribution.

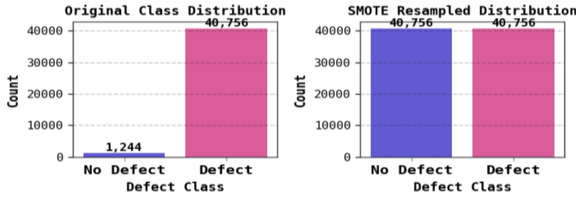


Figure 5. Class Distribution Before and After SMOTE

The class distribution before and after applying the SMOTE technique is shown in Figure 5. The non-defect class has only 1244 instances while the defect class has 40756 instances with a huge class imbalance. With SMOTE, the minority class is synthetically oversampled, resulting in an equal distribution of both classes: 40,756 minority and 40,756 majority samples. This is an equilibrated distribution that can enhance the learning ability and prediction performance of ML models.

3.5. Model Selection

Model selection is used to determine the best method for predicting software defects. In this study, RF and DistilBERT are selected for their strong classification performance, robustness, and ability to handle complex data patterns. RF is chosen for its high accuracy, interpretability, and efficient training performance, while DistilBERT is selected for its DL capability and contextual feature learning. The selected models are trained and evaluated to compare their effectiveness in detecting software defects.

3.5.1. Random Forest

The RF is a classification model that utilizes concept of ensemble learning, which combines many classifiers to improve the outcome[17]. The RF model consists of multiple DTs that are applied to subsets of the dataset, and their predictions are averaged to compute performance metrics. On accuracy and other metrics, the number of trees utilized has a considerable influence. However, after a given number of trees, the model improvement becomes constant. For training purposes, knowing the appropriate number of trees is essential. In this instance, the random state is 42 and 1000 trees are used. Random Forest is employed as a classifier.

3.6. DistilBERT

DistilBERT, a lightweight version of Bidirectional Encoder Representations from Transformers (BERT), is used for software defect prediction due to its faster training speed and lower computational requirements[18]. To determine whether software modules are broken, the model is trained on balanced, preprocessed data. The following parameters are used for training: a learning rate of 2×10^{-5}, a batch size of 16, a max sequence length of 128, and 3 training epochs. Optimization is performed using the Adam optimizer, and the loss function is cross-entropy. To assess the model's predictive performance, the training data set is used, followed by validation on the validation data set and testing on the test data set.

3.7. Evaluation Criteria

he classifiers' ability is assessed using evaluation metrics including acc, rec, prec, and AUROC. A confusion matrix, a matrix-like representation of expected versus actual classes, is used to assess this metric.

- True Positive (TP) indicates number of accurately categorized positive reports.
- False Positive (FP) indicates quantity of negative samples that were mistakenly classified as positive records.
- True Negative (TN) indicates number of negative records that were accurately classified.
- False Negative (FN) indicates quantity of positive samples that were incorrectly categorized as negative records.

Then, several assessment metrics are adjusted, as shown by Equations (2) to (6)

$$Accuracy = \frac{TP+TN}{TP+TN+FP+FN} \quad (2)$$

$$Precision = \frac{TP}{TP+FP} \quad (3)$$

$$Recall = \frac{TP}{TP+FN} \quad (4)$$

$$F1\text{-}Score = \frac{2\times Precision \times Recall}{Precision+Recall} \quad (5)$$

$$AUROC = \int_0^1 TPR(FPR)\, d(FPR) \quad (6)$$

A classifier's performance is assessed using accuracy, which compares accurately projected instances relative to overall number of instances. The accuracy rate is proportion of all expected positive samples that are properly detected. The probability that a sample correctly identified as a positive among the actual positive samples is known as the recall rate. The acc and rec harmonic mean is F1, which ranges from 0 to 1. A binary classifier's ability to discriminate between positive and negative data is gauged by its AUC.

3.8. Model Interpretation

The SHapley Additive Explanations (SHAP) analysis, which helps assess importance of each feature in model's predictions, has been used to interpret model's results. SHAP values are used to determine most important characteristics for software defect prediction.

4. Results and Discussion

An Intel Core i7 CPU (3.4 GHz, 8 cores) and 16 GB of RAM were used in experimental arrangement. Python 3.9 is used for all calculations in the Jupyter Notebook on a Windows 10 computer. The performance of the proposed RF and DistilBERT models for bug detection is compared in Table II. The RF model outperformed DistilBERT in terms of accuracy (99.98%). DistilBERT performed marginally better in terms of Precision (99.66%). In summary, the results presented that RF is superior in performance and training speed to DistilBERT, which is also a strong DL alternative for bug detection.

Table 2. Performance of Propose Models for Software Bug Detection

Metric	RF	DistilBERT
Accuracy	99.98	99.39
Precision	99.76	99.66
Recall	99.99	99.71
F1 Score	99.88	99.69

Train Time (s)	110.87	133.8

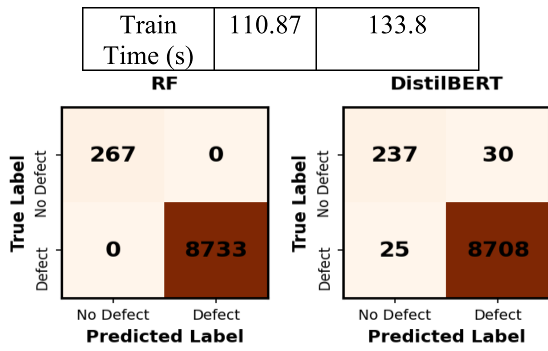


Figure 6. Confusion Matrix of RF and Distilbert Model

The confusion matrices for the RF model and DistilBERT model are presented in Figure 6. The RF model correctly classified all 267 non-defect and 8,733 defect samples without any misclassification, resulting in a perfect classification accuracy. On the other hand, the DistilBERT model has successfully classified 237 non-defect and 8,708 defect samples. But DistilBERT gave 30 FP and 25 FN. In overall, both models performed well, and RF model achieved good accuracy and consistency than DistilBERT.

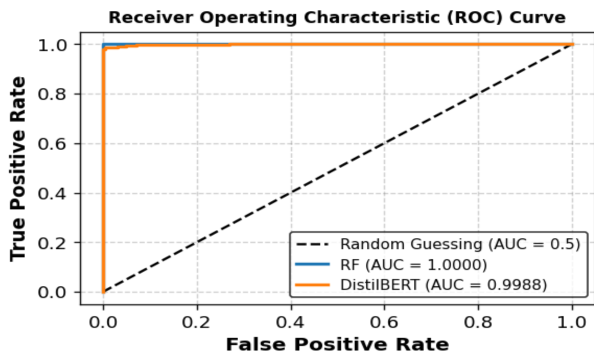
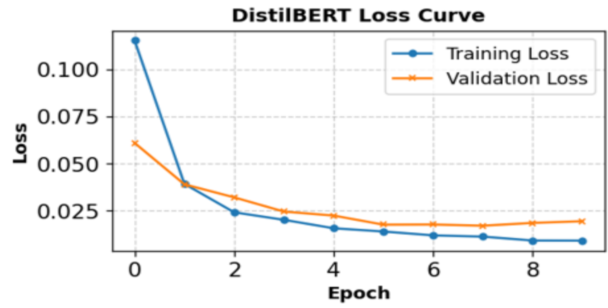


Figure 7. Plot ROC Curve of Propose Models

The suggested models' classification performance is displayed by ROC curve in Figure 7. The Random Forest (RF) shows an AUC of 1.0000, indicating it can perfectly distinguish between the classes, and the DistilBERT curve is very similar, with an AUC of 0.9988, indicating very good predictive ability. The dashed diagonal line is the baseline and represents random guessing (AUC = 0.5). Overall, the figure shows that both models perform significantly better than random classification, underscoring their reliability and robustness.



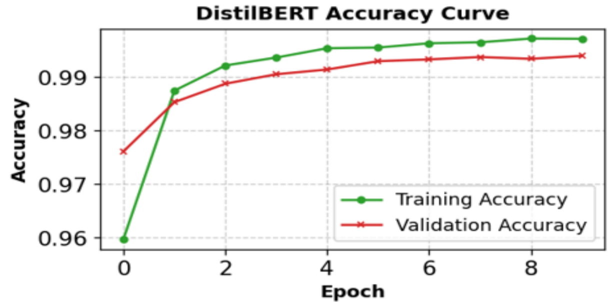


Figure 8. Training and Validation Accuracy/Loss of DistilBert Model

The training and validation performance of DistilBERT model is shown for every epoch in Figure. 8. The loss curve (top) indicates that the network is learning well and is not overfitting because the validation loss is always greater than the training loss. The accuracy curve at the bottom shows a sharp improvement, the training accuracy has exceeded 0.99, and the validation accuracy is nearly as high, indicating good generalization and convergence. Overall, demonstrates that the model has excellent predictive performance and is optimized in a stable way.

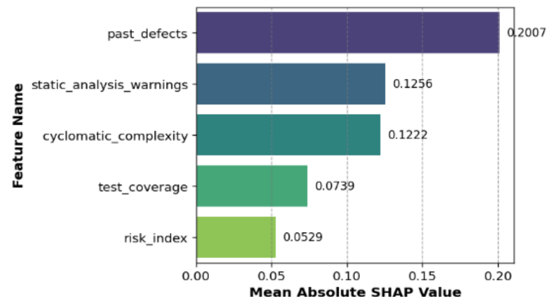


Figure 9. Top 5 SHAP Important Features for Defect Prediction

Figure 9 shows the top five important features influencing the defect prediction model using SHAP analysis. past_defects had the highest impact, followed by static_analysis_warnings and cyclomatic_complexity. test_coverage and risk_index also contributed to prediction performance. The findings reveal the main factors influencing the classification of software defects.

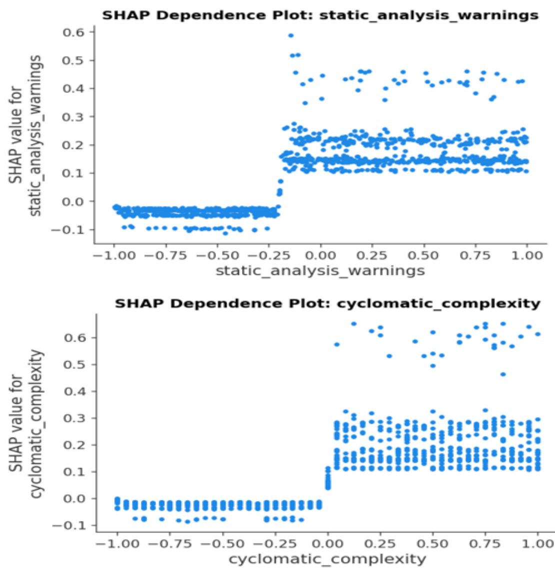


Figure 10. SHAP Dependence Plots for Static Analysis Warnings and Cyclomatic Complexity

The SHAP dependency charts for cyclomatic complexity and static analysis warnings are shown in Figure. 10, illustrating their impact on the model predictions. SHAP values in both plots are around zero for lower values of the features (less than approximately −0.25), and then increase rapidly as the features increase in value; for static analysis warnings SHAP values are clustered between 0.1 and 0.3, while for cyclomatic complexity they increase up to 0.6. The results indicate that increased number of code warnings and code complexity have more of an impact on the LLM model output, indicating their critical role in software defect likely prediction.

4.1. Comparative Analysis and Discussion

Table III shows the baseline models and proposed models for bug detection. Traditional ML and DL algorithms such as ANN, CNN, DT, ADA, and SVM achieve accuracies ranging from 76.07% to 83%, with SVM achieving the best baseline F1-score of 89.96%. On the other hand, the proposed RF and DistilBERT models achieved results that are much better than all the existing models. These findings show how reliable and successful the suggested methods are for precise bug discovery.

Table 3. Comparison for Bug Detection between Base and Proposed Models

Model	Accuracy	Precision	Recall	F1
ANN [19]	81.95	83.70	96.39	89.60
CNN [20]	83	67	33	44
DT [21]	76.07	73.76	76.02	76.0
ADA[11]	80.66	85.0	91.45	89.30
ANN[22]	79	43	21	29
SVM[23]	82.15	91.29	88.22	89.96
Random Forest	99.98	99.76	99.99	99.88
DistilBert	99.39	99.66	99.71	99.69

This research provides an effective approach for accurate software bug detection using Random Forest and DistilBERT models. The accuracy and F1-scores of the suggested models are substantially higher than those of the baseline models, and they also improve the predictability of flaws. Moreover, the study highlights the effectiveness of feature engineering, SMOTE balancing, and SHAP interpretability in improving model performance and interpretation. These insights can support software developers and organizations in identifying defects earlier, minimizing maintenance expenses, improving software quality and making software development methods more dependable.

4.2. Limitation and Future Work

The study has limitations, as the proposed Random Forest and DistilBERT models perform well, however. The models have been trained and tested on a sample dataset, and may not be generalizable for other software projects or real-world environments. Moreover, DistilBERT needed more computational resources and training time than Random Forest. Larger and more varied datasets could be used in future to enhance generalising properties of the models. Additionally, the feasibility and utility of advanced DL and ensemble methods can be explored and tested in the real-time software development environment to enhance their defect prediction's precision and resilience.

5. Conclusion and Future Work

Protecting software security requires anticipating software flaws. An efficient way to determine whether a program has software bugs is to use appropriate metrics. ML techniques have been applied extensively in recent years to test software fault prediction tools, with positive outcomes. The findings of this study demonstrated that both models RF and DistilBERT performed exceptionally well in predicting software issues. The proposed model, the RF model, yielded an accuracy of 99.98%, and DistilBERT gave 99.39% accuracy among the proposed models. Moreover, RF required less training time (110.87s) than DistilBERT (133.8s), indicating that RF is more computationally efficient. Based on SHAP interpretability analysis, the most influential factors affecting software defect prediction have been identified. Through comparative analysis with baseline models like ANN, CNN, Decision Tree, AdaBoost and SVM, it is observed that the proposed models yielded higher accuracy and reliability as compared to any of the existing models. The proposed framework can help software developers to identify bugs as early as possible in the software development process, facilitate software debugging, lower software maintenance costs, and enhance software quality and reliability.

References

[1] K. al-Sulbi and A. Attaallah, "Symmetric bug prediction in software requirement by machine learning algorithms," *Sci. Rep.*, vol. 15, no. 1, p. 38276, Oct. 2025, doi: 10.1038/s41598-025-22193-x.

[2] M. Kumari, R. Singh, and V. B. Singh, "Prioritization of Software Bugs Using Entropy-Based Measures," *J. Softw. Evol. Process*, vol. 37, no. 2, 2025, doi: 10.1002/smr.2742.

[3] X. Du *et al.*, "CoreBug: Improving Effort-Aware Bug Prediction in Software Systems Using Generalized k-Core Decomposition in Class Dependency Networks," *Axioms*, vol. 11, no. 5, 2022, doi: 10.3390/axioms11050205.

[4] E. Kesavan, "Software Bug Prediction Using Machine Learning Algorithms: An Empirical Study on Code Quality and Reliability," *Int. J. Innov. Sci. Eng. Manag.*, pp. 377–381, Sep. 2025, doi: 10.69968/ijisem.2025v4i3377-381.

[5] C. Z. Yang, C. C. Hou, W. C. Kao, and I. X. Chen, "An empirical study on improving severity prediction of defect reports using feature selection," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2012. doi: 10.1109/APSEC.2012.144.

[6] B. Krishnan, A. Thaneeru, R. Lingam, and S. K. Kaata, "The Future of Cloud Data Engineering: Multi-Tenant, Multi-Region Pipelines Leveraging LLM-Powered Data Governance," in *2025 1st International Conference on Advancement in Futuristic Technologies (ICAFT)*, Belagavi, India: IEEE, 2025, pp. 1–8, Dec. doi: 10.1109/ICAFT66710.2025.11453308.

[7] B. P. Singh and H. Singh, "Using LLMs for Autonomous Cloud Infrastructure Entitlement Management to Prevent Overprivileged Access," *J. Eng. Comput. Sci.*, vol. 5, no. 4, pp. 1–14, April, 2026, doi: https://doi.org/10.5281/zenodo.19488212.

[8] A. H. Dao and C. Z. Yang, "Severity prediction for bug reports using multi-aspect features: A deep learning approach," *Mathematics*, 2021, doi: 10.3390/math9141644.

[9] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, 2018, doi: 10.1145/3276517.

[10] G. Fan, Y. Liang, L. Zu, H. Yu, Z. Huang, and W. Chen, "Automatic identification of extrinsic bug reports for just-in-time bug prediction," *Sci. Comput. Program.*, vol. 249, p. 103410, Apr. 2026, doi: 10.1016/j.scico.2025.103410.

[11] I. Mansour, M. Ben Said, and Y. H. Kacem, "Enhanced Software Bug Prediction Using Double-Stacked Ensembles and Halving Search Optimization," *Procedia Comput. Sci.*, vol. 270, pp. 3789–3798, 2025, doi: 10.1016/j.procs.2025.09.504.

[12] B. Xu *et al.*, "Cross-Project Aging-Related Bug Prediction Based on Transfer Learning and Class Imbalance Learning," *IEEE Trans. Dependable Secur. Comput.*, 2025, doi: 10.1109/TDSC.2025.3567957.

[13] J. Jasz, "The Effectiveness of Hidden Dependence Metrics in Bug Prediction," *IEEE Access*, 2024, doi: 10.1109/ACCESS.2024.3406929.

[14] S. A. Alsaedi, A. Y. Noaman, A. A. A. Gad-Elrab, and F. E. Eassa, "Nature-Based Prediction Model of Bug Reports Based on Ensemble Machine Learning Model," *IEEE Access*, 2023, doi: 10.1109/ACCESS.2023.3288156.

[15] Z. Hou, L. Gong, M. Yang, Y. Zhang, and S. Yang, "Software Bug Prediction based on Complex Network Considering Control Flow," in *2022 IEEE 22nd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, IEEE, Dec. 2022, pp. 246–254. doi: 10.1109/QRS-C57518.2022.00044.

[16] R. B. Bahaweres, F. Agustian, I. Hermadi, A. I. Suroso, and Y. Arkeman, "Software defect prediction using neural network based smote," in *International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, 2020. doi: 10.23919/EECSI50503.2020.9251874.

[17] K. S. Bharath and P. Jagadeesh, "An Innovative Software Bug Prediction System using Random Forest Algorithm for Enhanced Accuracy in Comparison with Logistic Regression Algorithm," in *2023 Intelligent Computing and Control for Engineering and Business Systems, ICCEBS 2023*, 2023. doi: 10.1109/ICCEBS58601.2023.10449266.

[18] A. Ali, Y. Xia, Q. Umer, and M. Osman, "BERT based severity prediction of bug reports for the maintenance of mobile applications," *J. Syst. Softw.*, 2024, doi: 10.1016/j.jss.2023.111898.

[19] M. Jumare, H. and Darius, and T. Chinyio, "Software Defect Prediction Using Machine Learning and Deep Learning Techniques," *Kasu J. Comput. Sci.*, vol. 1, no. 3, pp. 527–543, 2024, doi: 10.47514/kjcs/2024.1.3.0010.

[20] N. A. A. Khleel and K. Nehéz, "A novel approach for software defect prediction using CNN and GRU based on SMOTE Tomek method," *J. Intell. Inf. Syst.*, 2023, doi: 10.1007/s10844-023-00793-1.

[21] S. M. H. Kabir, M. T. Rahman, and A. H. Mridul, "Software Defect Prediction Using Traditional Machine Learning and Ensemble Learning Algorithms," *Smart Wearable Technol.*, no. May, pp. 1–16, 2025, doi: 10.47852/bonviewswt52025645.

[22] S. Haldar and L. F. Capretz, "Interpretable Software Defect Prediction from Project Effort and Static Code Metrics," *Computers*, vol. 13, no. 2, p. 52, Feb. 2024, doi: 10.3390/computers13020052.

[23] B. Arasteh, S. S. Sefati, E. C. Popovici, I. F. Ince, and F. Kiani, "A Bedbug Optimization-Based Machine Learning Framework for Software Fault Prediction," *Mathematics*, 2025, doi: 10.3390/math13213531.