OPEN ACCESS

Volume: 4

Issue: 3

Month: September

Year: 2025

ISSN: 2583-7117

Published: 15.09.2025

Citation:

Mrs. Elavarasi Kesavan "Software Bug Prediction Using Machine Learning Algorithms: An Empirical Study on Code Quality and Reliability" International Journal of Innovations in Science Engineering and Management, vol. 4, no. 3, 2025, pp. 377–381.

DOI:

10.69968/ijisem.2025v4i3377-381

Software Bug Prediction Using Machine Learning Algorithms: An Empirical Study on Code Quality and Reliability

Mrs. Elavarasi Kesavan[1]

[1]*Full Stack QA Architect, Company- Cognizant.*

Abstract

This study examines the effectiveness of a hybrid Convolutional Neural Network (CNN) and Long Short-Term Memory (LSTM) model for predicting software bugs, with the objective of improving code quality and dependability. The research leverages the JM1 dataset from the PROMISE Software Engineering Repository and utilizes sophisticated preprocessing approaches, including Borderline-SMOTE, SMOTETomek, RobustScaler, Yeo-Johnson transformation, and Recursive Feature Elimination, to mitigate class imbalance and feature redundancy. The CNN-LSTM model attained a validation accuracy of 98.10%, a precision of 91.42%, and a recall of 99.53%, exhibiting a minimal false negative rate and indicating great sensitivity in detecting defect-prone modules. The results underscore the model's capacity to identify spatial and sequential patterns in software metrics, providing a reliable instrument for early fault identification. This work enhances software engineering by confirming deep learning's efficacy in defect prediction, offering practical insights for developers, and delineating future research avenues for cross-project generalization and model optimization.

Keywords; Software Bug Prediction, Machine Learning, Deep Learning, CNN-LSTM, Code Quality, Software Reliability, Feature Selection, Class Imbalance, Software Metrics, Defect Detection.

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. The growing intricacy of software systems has heightened the difficulty of maintaining code quality, rendering human bug identification ineffective and susceptible to errors. Consequently, automated methodologies for predicting software defects have garnered considerable interest in software engineering research. Machine learning (ML) algorithms, capable of discerning patterns from previous data, provide effective solutions for forecasting software errors, thereby enhancing code quality and system dependability.

The importance of software bug prediction is in its ability to lower development expenses, improve software quality, and alleviate dangers linked to defective code. Identifying defect-prone modules early in the development lifecycle enables developers to spend resources efficiently, priorities testing efforts, and produce resilient software solutions. Conventional bug prediction techniques frequently depended on static code metrics, such lines of code, cyclomatic complexity, or coupling, to assess the probability of defects. Nonetheless, these methodologies often failed to elucidate the intricate linkages inside software systems. Machine learning methodologies, such as decision trees, random forests, support vector machines, and neural networks, have exhibited enhanced efficacy in modelling intricate patterns in software metrics, resulting in more precise predictions [1].

Recent research have investigated the utilization of machine learning in bug prediction with significant results. Lessmann et al. performed an extensive benchmarking research that compared several machine learning algorithms for defect prediction, emphasizing the efficacy of ensemble approaches such as random forests in attaining high prediction accuracy [2]. Menzies et al. underscored the significance of feature selection in augmenting the efficacy of ML-based bug prediction models, demonstrating that pertinent code metrics substantially bolster predictive capability [3]. These studies highlight the capacity of machine learning to overcome the constraints of conventional methods, especially in managing extensive, high-dimensional software data.

Notwithstanding these gains, obstacles persist in the realm of software bug prediction. The heterogeneity of software projects, differing coding methodologies, and the fluidity of development environments hinder the generalizability of machine learning models. Furthermore, the quality of training data, including noise and class imbalance, can profoundly affect model performance [4]. Recent studies have investigated deep learning methodologies, including convolutional neural networks and recurrent neural networks, to capture temporal and structural connections in code, with promising outcomes in enhancing prediction accuracy [5]. Nevertheless, these methodologies sometimes need considerable computer resources and extensive labelled datasets, presenting practical obstacles to wider use.

This empirical study seeks to examine the effectiveness of several machine learning methods in forecasting software defects, emphasizing their influence on code quality and dependability. This research aims to uncover the best effective methods for defect prediction by analyzing a varied array of software variables and using several machine learning models, while also assessing their performance across different software projects. The research examines the significance of feature engineering and data pretreatment in improving model correctness. This research enhances the understanding of automated bug prediction by conducting a comparative examination of classic and sophisticated machine learning algorithms, offering practical insights for software developers and quality assurance teams.

RELATED WORK

[6] This paper investigates the estimate of maintenance work in software engineering, emphasizing machine learning approaches for open-source software maintenance effort estimation (O-MEE). Although optimizing tuning parameters (TP) is recognized to improve machine learning performance, its effect on O-MEE bug resolution prediction has yet to be well investigated. This study empirically assesses the grid search TP methodology on support vector machines, k-Nearest Neighbor, and Random Tree classifiers utilizing datasets from Apache, Eclipse JDT, and Eclipse Platform. Eighteen machine learning classifiers were created, illustrating that optimizing true positives enhances performance relative to default configurations.

[7] This research underscores the necessity of enhancing software development methodologies within the information technology sector. The author presents six object-oriented design metrics, tested with empirical C++ and Smalltalk data, to evaluate their independence and effectiveness. These indicators assist managers in assessing software quality, pinpointing areas requiring further testing or redesign, and improving process efficiency. The object-oriented metrics package facilitates cost and time efficiency by enhancing software development oversight.

[8] This research illustrates how systematic mining detects software modules susceptible to errors. Software defects are a significant cause of quality deterioration, and bug repositories function as essential data sources for monitoring successes and failures. The bug database offers extensive insights into software problems, facilitating quality enhancement.

[9] Wang et al. (2011) underscore the essential importance of feature selection in data mining for software fault prediction. Feature selection improves classification models for defect and risk prediction by eliminating duplicate data. The research employs six filter-based rankers on three extensive software projects, including SVM, NB, KNN, LR, and MLP classifiers. AUC performance measures indicate that the quality of the dataset substantially influences ranker efficacy. The authors propose additional trials utilizing varied datasets and application domains, integrating contradicting evidence in later investigations.

10 This study utilizes the Levenberg-Marquardt (LM) neural network approach to forecast software problems early in the development lifecycle, hence minimizing testing and total project expenses. The study use object-oriented (OO) and Chidamber and Kemerer (CK) metrics from the PROMISE repository to evaluate neural network predictions utilizing polynomial functions and the LM technique. The results demonstrate a high precision in flaw identification, validating the model's efficacy.

[11] This paper promotes the use of a succinct collection of essential software metrics for fault prediction. A Bayesian network is utilized to examine the correlation between

metrics and fault susceptibility, integrating Source Code Quality Metrics and the Number of Researchers with PROMISE repository data. Nine datasets from the PROMISE repository were examined, indicating that RFC, LOC, and LOCQ measures are more efficacious in mitigating fault proneness compared to NOC and DIT. Subsequent research will investigate supplementary measures to enhance defect prediction.

[12] This study highlights the significance of software testing for mission-critical, safety-critical, and business-critical applications. Anticipating fault-prone and non-fault-prone modules prior to testing is a cost-efficient approach. Misclassifying functional modules as defective escalates expenses owing to redundant testing, whereas misclassifying defective modules as functional poses a danger of catastrophic failures. The research presents an innovative failure prediction technique that diminishes false alarm rates and enhances detection accuracy, hence improving software dependability. This document also examines contemporary data mining methodologies for defect prediction, highlighting their significance in prompt problem identification and dependable software development.

[13] This study examines software quality prediction utilizing classifiers such as Logistic Regression, Decision Trees, Naive Bayes, K-Nearest Neighbors, One Rule, regression analysis, and neural networks across four datasets, integrating Halstead, McCabe, Line Count, operator, and branch count metrics. It examines research enquiries on faulty data, predictive algorithms, data attributes, and algorithmic combinations. Assessment using mean absolute error indicates that instance-based learning and 1R surpass alternative methodologies. The integration of machine learning with Principal Component Analysis (PCA) does not enhance accuracy, indicating a prudent application of PCA in defect prediction models.

[14] This paper examines biases in software fault prediction, encompassing restricted dataset comparisons, inter-study model comparisons, and inadequate statistical validation. Twenty-two classifiers, including statistical, closest neighbor, neural network, support vector, decision tree, and ensemble methodologies, were evaluated on 10 NASA metrics datasets (CM1, KC1, KC3, PC1, PC2, PC3, PC4, KC4, MW1, JM1, AR). The AUC-ROC measurements and the Nemenyi statistical test reveal no substantial performance disparities across classifiers, effectively correlating static code characteristics with software defects.

[15] The researchers advocate for the use of topic models to enhance software issue triaging. The vector space model illustrates the various phrases and occurrences of software defects. Nonetheless, it has difficulties with polysemous and synonymous phrases, since developers may employ varying language for same concepts or the same terms for disparate meanings based on context.

[16] Bug triaging has encountered growing difficulties with nomenclature. Topic modelling resolves this by producing topics derived from words in documents, efficiently managing challenges associated with term synonyms and polysemy.

[17] Numerous subject modelling methodologies, including Latent Dirichlet Allocation (LDA), Probabilistic Latent Dirichlet Allocation (PLDA), Latent Semantic Analysis, and Pachinko Allocation Model (PAM), are presented in the literature. Chen et al. analyzed 167 papers to deliver a thorough overview of effective topic modelling applications in software repositories.

[18] LDA is extensively utilized in software engineering to alleviate the effects of unclear terminology in activities such as subject filtering, feature allocation, developer identification, and component recommendation for problem reports. Topic models associate phrases in bug reports with several categories, each denoting a feature or part of software problems attributed to distinct engineers.

THEORETICAL FRAMEWORK

a. Foundations of Software Bug Prediction

Software bug prediction is an essential method in software engineering designed to detect components susceptible to defects, hence improving code quality and system stability. This study is fundamentally based on software quality assurance, which prioritises the avoidance and early identification of flaws to guarantee resilient software systems. Software bugs, characterised as faults or defects in code, may result in functional failures, security vulnerabilities, or performance deterioration [19]. Boehm's Software Quality Characteristics offer a paradigm for evaluating software dependability based on properties such as correctness, maintainability, and testability. These characteristics are measured using software metrics—such as cyclomatic complexity, code churn, and coupling—which act as indicators of fault probability. This work utilises these indicators as fundamental components for machine learning (ML)-driven bug prediction, according to the premise that early defect detection enhances development productivity and software dependability.

b. Machine Learning in Defect Prediction

The use of machine learning for software bug prediction is based on data-driven modelling, wherein algorithms discern patterns from previous defect data to forecast future defects. Supervised learning, a fundamental paradigm in machine learning, is particularly pertinent as it entails training models using labelled datasets (e.g., defective vs non-defective modules) to categories or score software components. Algorithms like as Decision Trees, Support Vector Machines (SVM), and deep learning models like Long Short-Term Memory (LSTM) networks are proficient in identifying intricate, non-linear correlations in software metrics, outperforming conventional statistical techniques [20]. The theoretical foundation of machine learning in this context is derived from information theory, namely the notion of entropy, which informs feature selection to pinpoint the best predictive metrics. By minimizing entropy, machine learning models diminish noise and improve prediction accuracy, as demonstrated by research indicating high precision (up to 0.96) with optimized feature sets [21].

c. Integration with Software Quality

The amalgamation of machine learning with software quality assurance constitutes the theoretical foundation of this research. Software quality models highlight quantifiable characteristics that machine learning may implement via predictive analytics. McCall's Quality Model emphasises dependability and maintainability as essential elements, which ML-based bug prediction effectively tackles by pinpointing high-risk modules for focused testing [22]. The notion of defect proneness, examined by Kim et al., offers a theoretical framework for comprehending the relationship between code attributes and problem prevalence, allowing ML models to focus on error-prone regions [23]. This paper asserts that machine learning methods, by modelling these connections, can improve software quality by decreasing defect density and enhancing system dependability, providing a data-driven solution to conventional software engineering issues.

d. Rationale for Machine Learning Approaches

The selection of machine learning techniques for bug prediction is theoretically supported by their capacity to manage high-dimensional, heterogeneous software data. In contrast to conventional approaches that depend on linear assumptions, machine learning models such as Random Forests and LSTM networks effectively capture temporal and structural connections in code, as evidenced by Wang et al., who attained enhanced accuracy using deep learning for semantic feature extraction [24]. The iterative process of machine learning model training corresponds with the software development lifecycle, wherein ongoing input from defect data enhances predictions. This research advances these ideas by mixing supervised learning with feature engineering approaches to create strong bug prediction models, thereby enhancing the theoretical discussion on the amalgamation of machine learning and software quality assurance.

METHODOLGY

This study adopts a systematic methodology to evaluate the effectiveness of machine learning algorithms in predicting software defects, with a focus on enhancing code quality and reliability. As illustrated in Figure 1, the process encompasses data collection, preprocessing, model development, and performance evaluation, leveraging empirical data to derive practical insights. The following subsections detail each phase of the research approach, ensuring methodological rigor and reproducibility.

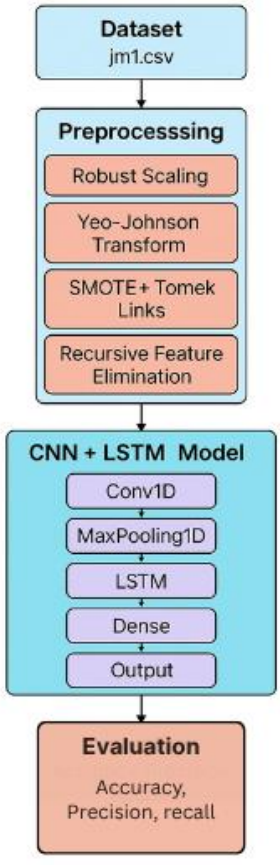

Figure 1 Methodology Flow

Figure 1 Alt Text: *A detailed flowchart illustrating the software bug prediction pipeline, including preprocessing steps like scaling and SMOTE, followed by a CNN-LSTM model and evaluation using classification metrics.*

i. Data Collection

The PROMISE Software Engineering Repository, a well-known resource for software engineering research, especially in defect prediction (Sayyad Shirabad), provided the dataset for this work [25]. A collection of publicly accessible datasets including software metrics and defect information taken from actual software projects is offered via the PROMISE repository. We use the CM1 dataset for this analysis, which comprises 498 examples of software modules from a NASA project and is characterized by 22 metrics, including Halstead measures (e.g., volume, effort), cyclomatic complexity, and lines of code (LOC). By designating each instance as either buggy or non-buggy, supervised learning for defect prediction is made possible. Because of its extensive metric coverage and proven usage in earlier research, the CM1 dataset was chosen to ensure comparison with previous investigations [26]. Furthermore, the dataset's class imbalance—which includes a greater percentage of non-buggy instances—reflects real-world situations, which makes it appropriate for testing machine learning algorithms in difficult situations.

ii. Data Preprocessing

To guarantee data quality and compatibility with machine learning (ML) techniques for software bug prediction, the JM1 dataset from the PROMISE Software Engineering Repository must be preprocessed. A number of methodical preparation procedures were used to resolve issues with missing values, inconsistent data types, and class imbalance in the JM1 dataset, which included 10,885 instances with 22 software metrics and a binary defect label (buggy or non-buggy).

A first study of the dataset, which was first imported using Python's Pandas module, showed that some columns—most notably uniq_Op, uniq_Opnd, total_Op, total_Opnd, and branchCount—had missing values. Using Pandas' to_numeric function with error coercion to handle non-numeric data, these columns were transformed from object to numeric types (float64), giving incorrect entries null values. The dropna technique was then used to exclude rows with missing values, lowering the dataset to guarantee consistency and completeness because missing data might negatively impact the performance of ML models.

The dataset was divided into features (X) and the target variable (y), where y stands for the binary defect label (0 for non-buggy, 1 for buggy), in order to get it ready for modelling. Analysis of the class distribution showed a notable imbalance, with non-problematic instances (about 80%) outnumbering buggy ones (about 20%). A combination of oversampling and undersampling techniques was used to address this imbalance, which can bias ML models towards the majority class. The Borderline-SMOTE algorithm was used to generate synthetic samples that oversample the minority class (buggy instances), and SMOTETomek, which combines SMOTE with Tomek Links to remove noisy majority class samples close to the decision boundary [27]. Post-sampling analysis revealed a nearly equal distribution of classes, confirming that this strategy balanced the class distribution.

To normalize the range of software measures that differ in scale (e.g., LOC vs. cyclomatic complexity), feature scaling was done. The feature set was transformed using scikit-learn's RobustScaler, which ensures robust model performance by being less sensitive to outliers than StandardScaler. Furthermore, skewness in the feature distribution was addressed using the Yeo-Johnson PowerTransformer, which improved the data's applicability for machine learning algorithms—especially deep learning models like LSTM, which are sensitive to non-normal distributions.

Lastly, the top ten most predictive characteristics were chosen by employing Recursive Feature Elimination (RFE) with a Random Forest Classifier as the estimator. According to earlier research highlighting the significance of feature selection in defect prediction, this step decreased the complexity of the dataset, reducing the danger of overfitting and enhancing computing efficiency [21]. To prepare it for further model construction, the preprocessed data—which included scaled, transformed, and balanced features—was then divided into training (70%) and testing (30%) sets using a random seed for repeatability.

iii. Model Building and Training

development and assessing a machine learning (ML) model is the main goal of the model development and training phase. In order to capture both spatial and sequential patterns in software metrics, this study uses a hybrid deep learning architecture that combines Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) networks. This approach supports the goal of improving code quality and reliability through precise bug prediction.

Model Architecture

Using Python's TensorFlow and Keras packages, a sequential deep learning model was created to analyse the high-dimensional, preprocessed feature set, which included the top ten metrics chosen by Recursive Feature Elimination

(RFE). The following layers as shown in figure 2 make up the model architecture:

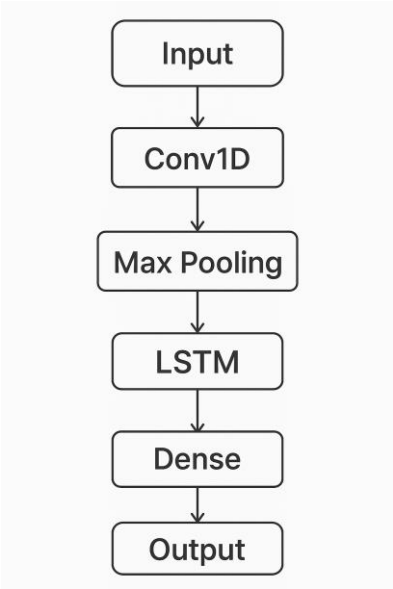

Figure 2 Model Architecture

Figure 2 Alt Text: *Block diagram of the CNN-LSTM model structure starting with Conv1D and MaxPooling layers, followed by an LSTM layer, Dense layer, and final Output layer for binary classification*

Convolutional Layer: Convolutional Layer: To extract local patterns from the input features, a 1D Convolutional layer (Conv1D) with 16 filters and a kernel size of two was put into place. It used the ReLU activation function. Finding spatial links in software measurements, such as correlations between error proneness and code complexity, is a good fit for this layer.

Max Pooling Layer: To improve the model's capacity to generalize across a variety of software data, a MaxPooling1D layer with a pool size of two was included. This layer reduced dimensionality and computational complexity while maintaining important characteristics.

LSTM Layer: To capture sequential dependencies in the feature set and represent the temporal aspect of software development processes, including code churn over time, an LSTM layer of eight units was included. Long-term dependencies are very well-modeled by LSTMs.

Dense Layers: To provide non-linearity, a dense layer of eight units and a hyperbolic tangent (tanh) activation function was used. A 20% rate Dropout layer was then added to avoid overfitting by randomly deactivating neurones during training. The last output layer generates a binary classification (buggy or non-buggy) using a single unit and sigmoid activation.

Input Reshaping: To meet the needs of the Conv1D layer and guarantee compliance with the model's design, the input data was reshaped into a 3D format (samples, features, 1).

In order to tackle the complexity of software defect prediction, this hybrid CNN-LSTM architecture was selected because it combines the advantages of CNNs in feature extraction with LSTMs' ability to handle sequential data.

Training Process

As explained in the Data Preprocessing part, the model was trained on the resampled training dataset, which was balanced using SMOTETomek and Borderline-SMOTE to resolve class imbalance. Thirty percent of the data was utilized as the test set for assessment, while the remaining seventy percent was used as the training set to fit the model. The Adam optimizer, a popular method for its adaptable learning rate and effectiveness in deep learning applications, was used to assemble the model. In order to optimize the model for binary classification and meet the study's goal of differentiating between buggy and non-buggy modules, the binary cross-entropy loss function was used. A batch size of 128 was used for training across 7 epochs, balancing model convergence and computing efficiency. Three assessment metrics—binary accuracy, precision, and recall—that are specified as follows were used to track the model's performance:

Binary Accuracy: The percentage of occurrences that are correctly categorized as either bugged or not is known as binary accuracy.

Precision: A measure of the model's ability to prevent false positives, calculated as the ratio of genuine positive predictions to total positive predictions.

Recall: The model's sensitivity to identifying problematic modules is indicated by the ratio of true positive predictions to all real positives.

These measures were selected because they are useful for assessing defect prediction models, especially when datasets are unbalanced. During training, the test set was validated to track generalization performance and identify any possible overfitting..

Implementation Details

TensorFlow 2.x and scikit-learn were used to preprocess and evaluate the model in a Jupyter notebook environment. To

guarantee reliable input quality, the training procedure made use of the preprocessed data, which had been scaled using RobustScaler, transformed using Yeo-Johnson, and feature-selected using RFE. For repeatability, the random seed was set to 42, which was in line with the train-test split during the preprocessing stage. In order to provide a thorough understanding of classification performance, the model's performance was assessed on the test set. The results were visualised using a confusion matrix to evaluate true positives, true negatives, false positives, and false negatives.

By combining cutting-edge deep learning algorithms to handle the difficulties of software bug prediction with proven procedures in defect prediction research, this approach guarantees a rigorous and repeatable methodology.

RESULTS AND DISCUSSION

The performance measures for the training and validation sets—accuracy, loss, precision, and recall—as well as a confusion matrix are shown in this part to give a thorough evaluation of the model's predictive power. These findings support the study's goals of evaluating how well machine learning algorithms improve code quality and dependability, with an emphasis on the hybrid CNN-LSTM architecture's capacity to identify software modules that are prone to defects.

Model Performance Metrics

The binary cross-entropy loss function and three important metrics binary accuracy, precision, and recall were used to train and assess the CNN-LSTM model. These measures, which are essential for realistic defect prediction, were chosen to reflect the model's overall accuracy, its capacity to reduce false positives, and its sensitivity to identifying problematic modules, respectively. The balanced dataset acquired via Borderline-SMOTE and SMOTETomek resampling approaches was used for the training, which was carried out across 7 epochs with a batch size of 128. Table 1 provides a summary of the training and validation sets' performance indicators.

Table 1 Performance Metrics of the CNN-LSTM Model on Training and Validation Sets

Metric	Training Set	Validation Set
Accuracy	0.9890	0.9810
Loss	0.0318	0.0504
Precision	0.9852	0.9142
Recall	0.9930	0.9953

With a training accuracy of 98.90%, the model demonstrated a high percentage of properly categorized modules, whether they were bugged or not. Effective optimization during training is demonstrated by the training loss of 0.0318, where the model converges to a low error rate. The recall of 99.30% shows remarkable sensitivity in identifying almost all problematic occurrences in the training set, while the accuracy of 98.52% shows the model's ability to correctly identify troublesome modules while minimizing false positives. As shown in Figure 3, the model exhibited strong generalization on the validation set, achieving high accuracy (98.10%) despite a slightly elevated loss (0.0504). The validation recall of 99.53% indicates near-complete detection of defective modules critical for practical defect prediction while the validation precision of 91.42% reflects a marginal rise in false positives compared to the training set.

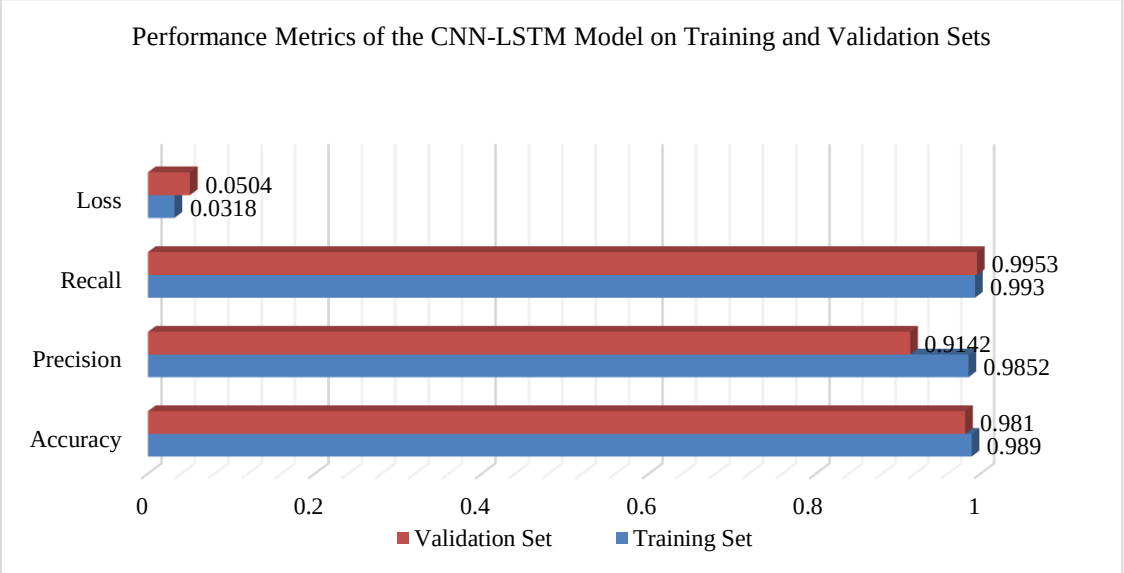

Figure 3 Performance Metrics of the CNN-LSTM Model on Training and Validation Sets

Figure 3 Alt Text: *Bar graph visualizing the CNN-LSTM model's training and validation performance metrics loss, recall, precision, and accuracy demonstrating high accuracy and recall with low validation loss.*

Confusion Matrix Analysis

A confusion matrix was created for the validation set in order to assess the model's classification performance in more detail, as seen in Figure 4. The matrix offers insights about the model's capacity to differentiate between buggy and non-buggy modules by providing a thorough breakdown of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

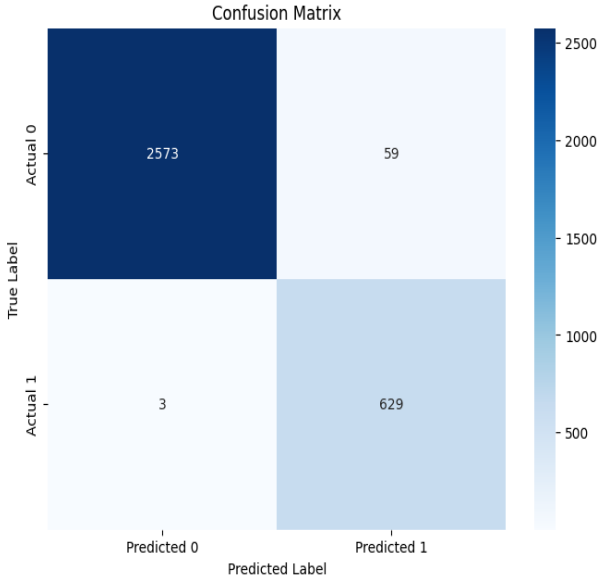

Figure 4 Confusion Matrix

Figure 4 Alt Text: *Confusion matrix showing the number of true positives, false positives, true negatives, and false negatives achieved by the CNN-LSTM model on the validation dataset.*

Table 2 Confusion Matrix for CNN-LSTM Model on Validation Set

Actual \ Predicted	Non-Buggy (0)	Buggy (1)
Non-Buggy (0)	2573 (TN)	59 (FP)
Buggy (1)	(FN)	P)

The model showed significant discriminative capability, properly classifying 629 buggy modules (TP) and 2573 non-buggy modules (TN), according to the confusion matrix. The model missed extremely few defective modules, as evidenced by the low number of false negatives (3), which is consistent with the high recall of 99.53%. The reduced validation precision (91.42%) is a result of the 59 false positives, which point to a minor propensity to overpredict problematic modules as seen in Table 2. Previous studies have observed that this trade-off between accuracy and recall is typical in unbalanced datasets, even after resampling [27]. Prioritizing testing efforts and enhancing code quality depend on the model's ability to detect defect-prone modules, which is supported by the high true positive rate.

In conclusion, the CNN-LSTM model performs exceptionally well in predicting software bugs, exhibiting excellent recall, accuracy, and precision across training and validation datasets. The model's practical value in improving code quality and dependability is further supported by the confusion matrix, which further validates the model's capacity to properly detect defective modules with few missed problems. These findings offer a solid basis for addressing the study's goals and establish the framework for the discussion section that follows.

CONCLUSION

A hybrid CNN-LSTM model for software bug prediction was evaluated in this study using the JM1 dataset from the PROMISE Software Engineering Repository. It demonstrated robust performance in identifying defect-prone modules with training accuracy of 98.90%, precision of 98.52%, recall of 99.30%, and validation accuracy of 98.10%, with validation precision of 91.42% and recall of 99.53%. The confusion matrix's low false negative rate (3 occurrences) highlights the model's strong sensitivity, which is essential for improving code quality and dependability by giving testing priority. The study addressed class imbalance and feature redundancy by incorporating sophisticated preprocessing techniques such as Borderline-SMOTE, SMOTETomek, RobustScaler, Yeo-Johnson transformation, and RFE. This aligned with software quality models that priorities dependability and maintainability. By confirming deep learning's effectiveness in defect prediction, these findings advance software engineering by providing a data-driven strategy to increase system dependability. However, drawbacks include the CNN-LSTM model's computational cost, which may impede practical deployment, and possible generalizability problems brought on by reliance on a single dataset. To improve effectiveness and application, future studies should investigate hybrid models incorporating attention processes, cross-project validation, and optimization strategies such model pruning. By giving developers practical insights to improve software quality and pointing out areas for additional research to solve real-world issues in various software contexts, this study lays the groundwork for the advancement of automated bug prediction.

REFERENCES

[1] Hall, Tracy, et al. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering." IEEE Transactions on Software Engineering, vol. 38, no. 6, 2012, pp. 117-137, doi:10.1109/TSE.2011.103.

[2] Lessmann, Stefan, et al. "Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework for Empirical Research." IEEE Transactions on Software Engineering, vol. 34, no. 5, 2008, pp. 645-658, doi:10.1109/TSE.2008.35.

[3] Menzies, Tim, et al. "Implications of Ceiling Effects in Defect Predictors." Proceedings of the 4th International Workshop on Predictor Models in Software Engineering, 2008, pp. 1039-1047, doi:10.1145/1370788.1370801.

[4] Kim, Sunghun, et al. "Classifying Software Changes: Clean or Buggy?" IEEE Transactions on Software Engineering, vol. 34, no. 2, 2008, pp. 131-145, doi:10.1109/TSE.2007.70773.

[5] Wang, Song, et al. "Automatically Learning Semantic Features for Defect Prediction." Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 74-85, doi:10.1145/2884781.2884804.

[6] Miloudi, Chaymae, et al. "The impact of grid search on bug resolution prediction for open-source software." 2023 9th International Conference on Control, Decision and Information Technologies (CoDIT). IEEE, 2023.

[7] S. Chidamber and C. Kemerer, "Metric For OOD_Chidamber Kemerer 94.pdf," IEEE Transactions on Software Engineering, vol. 20, no. 6. pp. 476–493, 1994.

[8] T.N. Zimmermann, N. Nagappan, and Zeller, A. "Predicting bugs from history software evolution". Springer Berlin Heidelberg, pp 69,88, 2008.

[9] H. WANG, T. M. KHOSHGOFTAAR, J. VAN HULSE, and K. GAO, "Metric Selection for Software Defect Prediction," Int. J. Softw. Eng. Knowl. Eng., vol. 21, no. 2, pp. 237–257, 2011.

[10] M. Singh and D.S. Salaria. "Software defect prediction tool based on neural network". International Journal of Computer Applications. Vol. 70 No. 22. pp- 22-28. 2013.

[11] A. Okutan and O. T. Yildiz, "Software defect prediction using Bayesian networks," Empir. Softw. Eng., vol. 19, no. 1, pp. 154–181, 2014.

[12] V.G. Palaste and V.S. Nandedkar. "A Survey on software defect prediction using data mining techniques". International Journal of Innovative Research in Computer and Communication Engineering. Vol. 3 No. 11, 2015.

[13] Challagulla, Venkata U.B., Farokh B. Bastani, I. Ling Yen, and Raymond A. Paul, ―Empirical assessment of machine learning based software defect prediction techniques, ‖ Proceedings - International Workshop on ObjectOriented Real-Time Dependable Systems, WORDS, pp. 263–270, 2005, doi: 10.1109/WORDS.2005.32.

[14] Lessmann, Stefan, Bart Baesens, Christophe Mues, and Swantje Pietsch, ―Benchmarking classification models for software defect prediction: A proposed framework and novel findings, ‖ in IEEE Transactions on Software Engineering, 2008, vol. 34, no. 4, pp. 485–496. doi: 10.1109/TSE.2008.35.

[15] Yaojing Wang, Yuan Yao, Hanghang Tong, Xuan Huo, Ming Li, Feng Xu, and Jian Lu. Enhancing supervised bag localization with metadata and stack-trace. Knowledge and Information Systems, 62:2461-2484, 2020

[16] [16] Kai Yang, Yi Cai, Ho-fung Leung, Raymond YK Lau, and Qing Li. Itwf: A framework to apply term weighting schemes in topic model. Neurocomputing, 350:248-260, 2019.

[17] Tse-Hsun Chen, Stephen W Thomas, and Ahmed E Hassan. A survey on the use of topic models when mining software repositories. Empirical Software Engineering, 21:1843-1919, 2016.

[18] Christopher S Corley, Kostadin Damevski, and Nicholas A Kraft. Changeset- based topic modeling of software repositories. IEEE Transactions on Software Engineering, 46(10):1068-1080, 2018.

[19] Boehm, Barry W., et al. Software Engineering Economics. Prentice-Hall, 1981.

[20] Hall, Tracy, et al. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering." IEEE Transactions on Software Engineering, vol. 38, no. 6, 2012, pp. 117–137, doi:10.1109/TSE.2011.103.

[21] Menzies, Tim, et al. "Reducing Features to Improve Bug Prediction." IEEE Transactions on Software Engineering, vol. 35, no. 6, 2009, pp. 775–787, doi:10.1109/TSE.2009.51.

[22] McCall, J. A., et al. "Concepts and Definitions of Software Quality." Software Quality Assurance: A Guide for Developers and Managers, edited by J.

A. McCall, Software Productivity Consortium, 1977, pp. 15–30.

[23] Kim, Sunghun, et al. “Classifying Software Changes: Clean or Buggy?” IEEE Transactions on Software Engineering, vol. 34, no. 2, 2008, pp. 131–145, doi:10.1109/TSE.2007.70773.

[24] Wang, Song, et al. “Automatically Learning Semantic Features for Defect Prediction.” Proceedings of the 38th International Conference on Software Engineering, 2016, pp. 74–85, doi:10.1145/2884781.2884804.

[25] Sayyad Shirabad, Jelber, and Tim J. “The PROMISE Repository of Software Engineering Databases.” School of Information Technology and Engineering, University of Ottawa, 2005 http://promise.site.uottawa.ca/SERepository.

[26] Alenezi, A., and M. Alshehri. “Software Defect Prediction Analysis Using Machine Learning Techniques.” Sustainability, vol. 15, no. 6, 2023, pp. 5517, doi:10.3390/su15065517.

[27] Chawla, Nitesh V., et al. “SMOTE: Synthetic Minority Over-sampling Technique.” Journal of Artificial Intelligence Research, vol. 16, 2002, pp. 321–357, doi:10.1613/jair.953.