Deep Hybrid Intelligence: CNN-LSTM for Accurate Software Bug Prediction

Md Nasim Uddin Ansari[1], Pankaj Richhariya [2]

[1] Research Scholar. Department of Computer Science, Bhopal Institute of Technology and Science
[2]HOD, Department of Computer Science, Bhopal Institute of Technology & Science

OPEN ACCESS

Volume: 3

Issue: 4

Month: December

Year: 2024

ISSN: 2583-7117

Published:06-12.2024

Citation:

Md Nasim Uddin Ansari, Pankaj Richhariya, Deep Hybrid Intelligence: CNN-LSTM for Accurate Software Bug Prediction, International Journal of Innovations In Science Engineering And Management, vol. 3, no. 4, 2024, pp. 26–33.

DOI

10.69968//ijisem.2024v3i426-33

Abstract

This research aimed to explore the potential of applying deep learning to software bug prediction. The study utilized various data preprocessing techniques that were essential in preparing the data for analysis, using a set of commonly available software bug reports and related metrics. In the data collection and preprocessing phase, the dataset was filtered to focus on critical software metrics, scaled for consistency, and additional techniques such as feature engineering and standardization were employed to enhance data variability. In order to analyze the effectiveness of the model in predicting software faults, the dataset was split so that it could be used for testing and training purposes. Several deep learning models, include CNN and LSTM architectures, were developed utilizing the preprocessed dataset in order to enhance the performance of the models. Subsequently, a hybrid ensemble technique was employed, combining the prediction outcomes of the best-performing individual models to form an ensemble model. Using test datasets, each model's performance was assessed using common assessment measures including precision, F1 score, accuracy, and recall. The ensemble models outperformed individual models in bug prediction, as demonstrated by higher accuracy and F1 scores. The final model achieved an accuracy of 96%, which was considered highly satisfactory for predicting software defects.

Keyword: Software bug prediction, deep learning, computer vision, Convolutional neural networks.

INTRODUCTION

Software bugs may be due to various reasons such as faulty requirements definition, coding errors, logical design errors, non-compliance with coding instructions, testing shortcomings, etc. Bugs can get introduced at any stage in the software development process. Software bugs reflect poor quality, therefore two quality objectives of the software development process.

The process of developing software includes bugs & faults, which may cause failures of the system vulnerabilities in security, and financial losses. Predicting and identifying defects early in development helps decrease debugging and maintenance costs and labor. Software bug prediction, a discipline of software engineering, develops methods and models to detect defective software components to help engineers prioritize testing and debugging. The process of fixing software defects is both time-consuming and expensive. It is possible to enhance software quality and decrease expenses by predicting defects early on in the development process. This research paper investigates several strategies for predicting software flaws in order to discover probable issues and maximize the effectiveness of debugging efforts.

Fixing bugs or flaws in software is the primary way that consistency and quality are achieved. However, some bugs may originate from non-code-related sources (such byte code encoding or compilers). Coding is the primary cause of errors in software. Examining and testing software is the conventional method of identifying flaws. Yet, a significant amount of time and effort may be needed for these tasks. As an alternative to a totally manual approach, frequent prediction of flawed software modules at an early stage may help engineers improve code quality at a lower cost.

As a result, software defect prediction, also known as SDP, is a promising strategy for increasing software quality since it aims to discover any defects in the program as quickly as possible. The Software Development Process (SDP) has thus become an important field of research in the modern age of technology development and testing.

It is difficult to predict software's defect-prone areas before making significant attempts to evaluate errors. Finding the problematic areas of source code with improved fault prediction performance is the primary difficulty of SDP. For many years, a variety of strategies and tactics have been put out and documented in the literature in order to achieve this. While some research has concentrated upon the semantic analysis of the source code, several researchers employ learning-based techniques to achieve higher accuracy in SDP. To create effective SDP models, the researchers have been implementing ML (machine learning) as well as in recent times, deep learning (DL) methods. Manual feature extraction is necessary for ML-based SDP approaches, mostly based on metrics for software. While software metrics are useful for identifying software defects, individually extracted features are difficult to create and do not fully capture the semantic information provided by bug reporting methods. Conversely, deep learning (DL) methods automatically extract higher-level characteristics and learn from higher-dimensional and more complicated data. As a result, a lot of academics have been working on creating SDP model using DL-based methods lately.

Software Bug Prediction (SBP) aims to predict buggy or faulty software constructs or modules before they affect software performance i.e., during the pre-deployment phase of the SDLC. In this way, SBP forms a core component of the overarching framework of Software Quality Assurance (SQA)

RELATED WORK

The author of this research paper addresses the rising significance of quality of software as a crucial component of system dependability. In many R&D departments, software engineering concepts are becoming more and more important. A large quantity of previous fault data is created and gathered over the software's development and operation stages, but it is seldom studied and exploited. Software developers may discover error-prone modules and probable failure types early on and facilitate quick fixes by using software failure prediction technology, which has the ability to foresee software faults before testing. Building a high-performing applications prediction of defects method for system software still faces a number of difficulties. First of all, failure situations in current system software are varied and challenging to identify. Second, a lot of the problem data is repetitious, jumbled, and lacking. Finally, there aren't many predictive models that provide good interpretability together with great performance. This research article investigates the building strategies for creating efficient software models for defect prediction that are suited to system software requirements to respond to these difficulties

This study emphasizes software risk component categorization for developers. This category improves software availability, security, and project management. A unique risk estimating technique was developed to help internal stakeholders analyze software risk by forecasting a quantifiable risk value. Bug-fix time assessments, duplicate bug records, and software component priority levels are used to derive this figure from historical software bug reports. The suggested method uses TensorFlow and machine learning to forecast the likelihood of software bugs using the Mozilla Core datasets (Connections: HTTP software component). While risk levels ranged from 27.4% to 84%, the highest predicted accuracy for bug-fix time was 35%. Bug-fix time estimations correlated strongly with risk ratings, but duplicate bug records correlated less.

The researchers have suggested topic models to enhance the triaging of software bugs. The software bugs' varied phrases and count are represented by the vector space model. Sometimes different terminology used by developers signify different things, and depending on the situation, the same terms might indicate different things. For this reason, polysemous and synonymous terms are not handled by the vector space model. The area of bug triaging has become increasingly aware of this issue. Modeling of topics has been frequently used to solve this issue. Based on the words found in the file, topics are generated in the topic model, which helps with issues related to term synonyms and polysemy.

An explanation is provided by the author of this research paper on how machine learning classifiers have evolved into helpful tools for recognizing possible issues in source code file updates. Following initial training on historical software data, the classifiers are then used to make predictions about potential software flaws. On the other hand, the current classifier-based bug predictions systems have a number of significant shortcomings, two of the most significant of which are their dependence on a huge number of features and their potential lack of accuracy for practical implementations. It is possible that the methodology's accuracy and scalability will suffer as a result of the depth of its features. According to the findings of the study, a feature selection approach that was developed specifically with classification-based bug predictions might be used to solve these issues. With the help of this technique, software changes faults may be anticipated, and a comprehensive investigation of the efficiency of Bayes naive and Support Vector Machine, or SVM, classifiers is carried out.

[8] In their study, the authors further clarify that a multitude of software metrics are accessible for the purpose of software defect prediction. Working with fewer sets of critical metrics and concentrating only on those measures is always advised when predicting software defects. To examine the relationship between software metrics and fault proneness, they used a Bayesian network. They have specified two more metrics, Source Code Quality Metrics and a Number of Researchers, to go along with the metrics used in Promise Repository. They have chosen nine datasets from the Promises Repository for their trial. They came to the conclusion that although NOC and DIT are less efficient and unreliable, RFC, LOC, and LOCQ are more successful in reducing error proneness. Emphasis has been placed on working with a smaller set of software metrics, and their future work will incorporate more software metrics as well as metrics to discover the optimum metrics utilized for defect prediction.

DEE LEARNING ARCHITECTURES

CNN: Utilizing Convolutional Neural Network for ordered grids data processing is the primary application of these neural networks. In addition, they are also used for the purpose of predicting software faults when the data in question is represented as structured input. The input to a CNN for software bug prediction can be a matrix representation of software metrics or code features. For example, this could include data related to code complexity, number of lines of code, number of commits, developer activities, & previous bug occurrences. Each software file or code segment is converted into a matrix format suitable for the CNN input layer. Next, the convolutional layer is the core element of CNNs. The convolutional layer adds filter (the kernels) to the input data in software bug prediction. These filters are used to discover local patterns or characteristics by swiping over the input matrix. For example, filters may identify patterns in code metrics or detect regions in the software that have similar complexity profiles, making them prone to bugs. To add non-linearity to the model, an activation function such as the Rectified Linear Unit (ReLU) is used after the convolution procedure. By doing this step, the model may learn more intricate connections between the software problems and the input characteristics. Reducing the number of dimensions of the feature maps by pooling layers—usually max-pooling—makes a network more computationally effective and lowers the chance of overfitting. By eliminating less relevant data and concentrating on the most significant traits, pooling aids in down-sampling the input data. The output gets flattened into one-dimensional vectors and processed through layers that are fully connected after a number of convolutional and pooling layers. These layers combine all the learned features to predict whether a specific piece of software or code segment contains bugs. The fully connected layers integrate the extracted patterns from the previous layers to make a final decision about the bug likelihood. The output layer typically uses a sigmoid or SoftMax activation function to predict the probability of a software bug. For binary classification (buggy or non-buggy), the output might be a single value between 0 and 1, indicating the likelihood of a bug. In multi-class classification (e.g., predicting the severity of a bug), the SoftMax function is used to assign probabilities to multiple categories.

LSTM: Recurrent neural networks, or RNNs, that are able to learn and preserve long-term dependence in sequential input are referred to as Long Short-Term Memory networks and LSTMs for short. These networks are more often used in artificial intelligence. The input to an LSTM in software bug prediction is typically a sequence of time-dependent data. The core of the LSTM model is its memory cell structure, which is designed to maintain information over long time intervals. This is crucial in software bug prediction, where the occurrence of a bug may depend on events (e.g., code changes or bug reports) from much earlier in the software development lifecycle. LSTMs have three main components that help manage information flow: Forget Gate: Selects which data to ignore from the preceding time step. For software bug prediction, this might involve forgetting older code changes that are less relevant for predicting current bugs. Input Gate**:** The input gate selects what fresh data goes into the memory cell. This aids in the model's concentration on the most important elements of the software's present state, including recently committed changes or bug patches. Output Gate: During each time step, the output gate is responsible for controlling the results of current memory state. This information is then utilized to produce predictions. LSTM models process data sequentially, which is well-suited for time-series data like: Commit histories and Code changes over time. After processing the sequence of inputs (e.g., commits or software metrics over time), the final LSTM output is passed through a fully connected layer to make predictions. The output could be a binary classification (buggy or non-buggy code) or a probability score that indicates the likelihood of a bug in the future version of the software.

METHODOLOGY

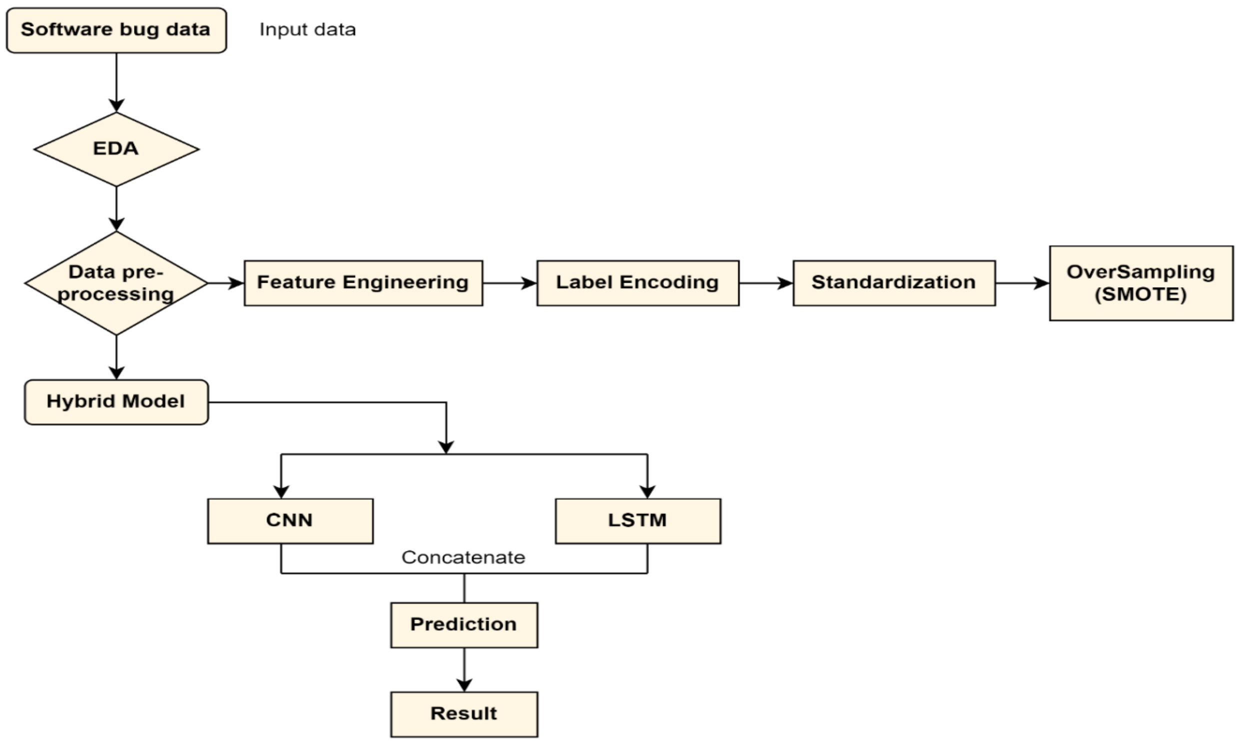
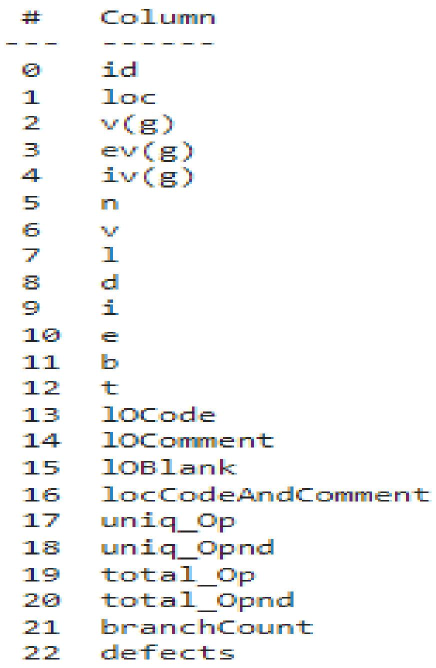


Figure 1 Flow Chart of Proposed Model

Dataset

The information was gathered from kaggle.com, a well-known venue for machine learning and data science contests. The topics addressed in "Software bug prediction: JM1 dataset" are the subjects of data collection. On Kaggle, a large number of datasets are available for public use. Individuals and teams may compete to solve a variety of data-related challenges on the well-known platform Kaggle, which also hosts datasets and data science contests. The dataset's structural organization to describe several key columns, each column provides distinct facets of software development and bug related information essential for analytical process such as lines of code (LOC), cyclomatic complexity, code churn, and coupling between objects and bug reports as bug ID, description, severity, priority, and status (open, closed, in progress). It aids in the analysis of the kinds and severity of errors that arise during the development process.



```
 #    Column
---   ------
 0    id
 1    loc
 2    v(g)
 3    ev(g)
 4    iv(g)
 5    n
 6    v
 7    l
 8    d
 9    i
 10   e
 11   b
 12   t
 13   lOCode
 14   lOComment
 15   lOBlank
 16   locCodeAndComment
 17   uniq_Op
 18   uniq_Opnd
 19   total_Op
 20   total_Opnd
 21   branchCount
 22   defects
```

Figure 2 Dataset

Overall, the data collection process in this research involved careful selection and cleaning of the data to ensure that it was of high quality and suitable for the research question at hand.

Data Pre-processing

In software bug prediction, data quality affects the performance of predictive model. The dataset has twenty-three columns and 10885 observations in total, providing a sample set for analysis. Assurance of the efficacy and reliability of the dataset for software bug identification was our primary objective. A number of processes are involved in data preparation that help convert unformatted data into formatted data that is appropriate for deep learning models. Each step in the data preparation was carefully designed to address specific challenges and optimize the dataset for subsequent analysis and modeling:

- **Data cleaning**: In this process different step is used to cleaning the data such as handling missing value, removing duplicates and inconsistent data.
- **Handling Missing Values:** Missing data in features like commit messages or bug descriptions can be filled using techniques like imputation (e.g., using the median or mean for numerical features) or simply removing rows with missing values for certain cases.
- **Removing Duplicates**: There might be duplicate records, especially in bug tracking data (e.g., duplicate bug reports). These can be removed to avoid bias in the prediction model.
- **Correcting Inconsistencies:** Fixing any inconsistencies in labels, formatting issues, or typos in categorical features like bug severity, commit messages, etc.
- **Feature Engineering:** Adding additional features and assisting in the extraction of more important details from the raw data are aspects of feature engineering. For example: Time based feature, developer-specific feature, code complexity metrics, code churn and textual analysis of bug reports.
- **Data labeling:** In data labeling defining the target variables and assigning labels, the target variables is typically whether the code module is defect or non-defect. This can be extracted from historical bug reports like binary classification and multi-class classification. And in assigning labels, ensure each code module commit is correctly labeled, based on bug reports.
- **Data Splitting:** We divided the data set in test and training sets in order to swiftly assess the effectiveness of the deep learning models. The testing sets functioned as a separate dataset for assessing model performance, whilst the training set, which included the bulk of the data, was utilized to train the models and assured that the models have been trained and assessed on separate subsets of data by dividing the dataset in this way, which allowed for accurate estimate of the model's performance and generalization to previously unknown data.

Training

This work proposes a hybrid ensembles model for accurate software bug identification. Combining Convolutional neural network with LSTM deep learning approaches increases predictive power. An organized, iterative process established and optimized this hybrid ensemble approach.

1. CNN

Utilizing Convolutional Neural Network for ordered grids data processing is the primary application of these neural networks. In addition, they are also used for the purpose of predicting software faults when the data in question is represented as structured input.

2. LSTM

Recurrent neural networks, or RNNs, that are able to learn and preserve long-term dependence in sequential input are referred to as Long Short-Term Memory networks and LSTMs for short. These networks are more often used in artificial intelligence.

The proposed model is a hybrid deep learning architecture combining convolutional layers (Conv1D) and recurrent layers (LSTM), followed by dense layers for binary classification.

Initially, the data is reshaped to include a third dimension, as the Conv1D layer requires a 3D input shape. This additional dimension (with a size of 1) represents a channel, as needed by the Conv1D layer.

The first layer is a Conv1D, which applies a 1D convolution over the input sequence. It uses 16 filters, each of size 2, to capture local patterns in the data. This helps in feature extraction from the input sequence by sliding the filters over the data and performing element-wise multiplications.

After the convolution, a MaxPooling1D layer is used to down-sample the input. This reduces the size of the output by taking the maximum value within each pool (group) of values. Pooling helps reduce computational cost and prevent overfitting.

Following this, the LSTM layer is applied to capture temporal dependencies or patterns across the sequence data. The LSTM layer has 8 units, and it's set to return a single output instead of a sequence of outputs because return_sequences is set to False. This layer is critical for learning from sequences or time-dependent data.

A dense layer with eight units & a tanh activation function comes next. Because it produces value between -1 and 1, the activation function of tanh is appropriate for processing learnt characteristics.

Overfitting is avoided by adding a Dropout layer with a rate of 0.2. During training, 20% of the neuron in the layer are randomly deactivated via dropout, which strengthens the model.

Finally, the sigmoid activation function and a single unit Dense layer are used for binary classification. That is all the sigmoid function, a continuously and strictly rising function, accomplishes. The aesthetics of probability theory are the best that the model is taking the chance of 0, or 1 of the classes. If the model is binary, it will output the probabilities through a sigmoid function.

3. Compilation and training

Three evaluation metrics—binary accuracy, precision, and recall—are used to further test the model once it has been assembled with an optimizer called Adam to reduce the loss function's binary cross-entropy. These metrics, however, give a clearer picture by explaining the model’s behavior based on the count of true positives, false positives, and conditional negatives.

4. Model assessment

To ensure bug classification accuracy and reliability, our software bug prediction research must analyse deep learning models. For model evaluation, we focus on accuracy. These metrics give crucial information about the model's classification skills and a good platform for comparing models and hybrids' software bugs classification abilities. Finally, the efficiency of the models that were trained was evaluated based on the suitable assessment criteria, which included the following: Accuracy, recall, precision and F1-score.

RESULTS AND DISCUSSION

Based on evaluation criteria, the model classified software bug prediction successfully. The model achieves 84% for recall, 94% for precision, 96% for accuracy, and 89% for F1 Score. These findings show how well the model categorizes software bug prediction. Calculate accuracy by dividing the total number of false positives and genuine positive by the total number of true positives. It evaluates how well the model finds good examples. The model's 96% accuracy scores in recognizing positive events. For the purpose of computing the recall metric, the number of false negatives and true positives is divided by the total number of true positives. It acts as a gauge to determine whether or not the model is capable of identifying each and every outstanding case. The fact that the model has a recall score of 84% demonstrates that it has a very high level of memories for positive instances. During process of computing F1 Score, which represents a weighted measurement of the accuracy of the model, recall and accuracy are taken into account. The recall and accuracy essential methods are used in the computation of this value. The F1 Score of the model is 89%, which indicates that it was able to strike an optimal balance between the accuracy & recall.

Table 1 Evaluation on Test data

Metric	Value
Precision	94%
Recall	84%
F1 Score	89%
Accuracy	96%

Here are visual results of proposed mode –

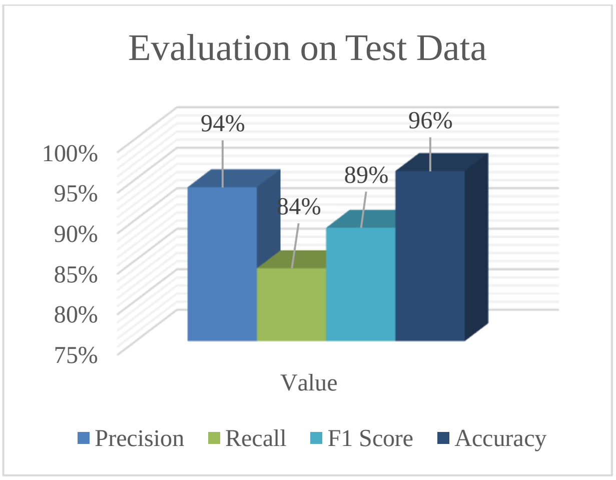


Figure 4 Confusion Matrix of proposed model on test data

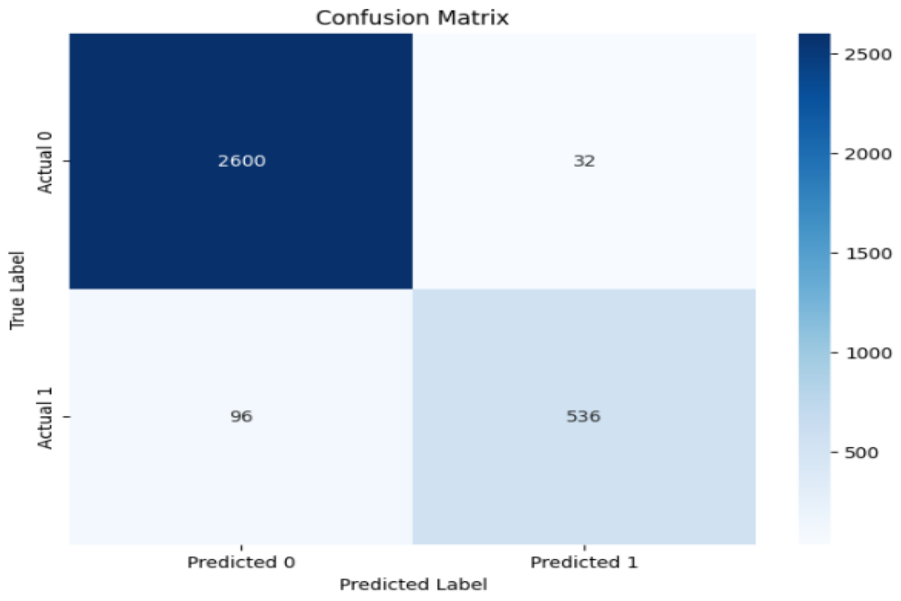


Figure 5 Confusion Matrix

Our suggested approach to software bug prediction is a major improvement over current practices, especially with regard to scalability, accuracy, and resilience. By comparing our approach with the existing work, and highlight the improvements and novelty of our research. It is crucial to assess our suggested model's accuracy, efficiency, and resilience against current methods in order to prove its effectiveness. Below is a detailed comparison between our proposed method and existing work in software bug prediction:

Table 2 Comparing the Proposed method with Existing Work

Model	Accuracy
Random forest (Exiting work) [9]	82%
Hybrid Ensemble Model (Proposed Work)	96%

Prior research used the various type of machine learning model and convolutional neural network to classify the software bug prediction while in the proposed model used deep learning models. The following findings were obtained by using the CNN + LSTM model to predict software bug classification in the proposed work on the "Software bug predictions using machine learning on JM1 dataset": accuracy of 96%, F1 score of 89%, recall of 84% and precision of 94%. We compared our results with the results of the base paper, which used the machine learning model “Random Forest” on the same dataset and achieved precision of 75%, recall of 80%, F1 score of 75% and accuracyof82%.

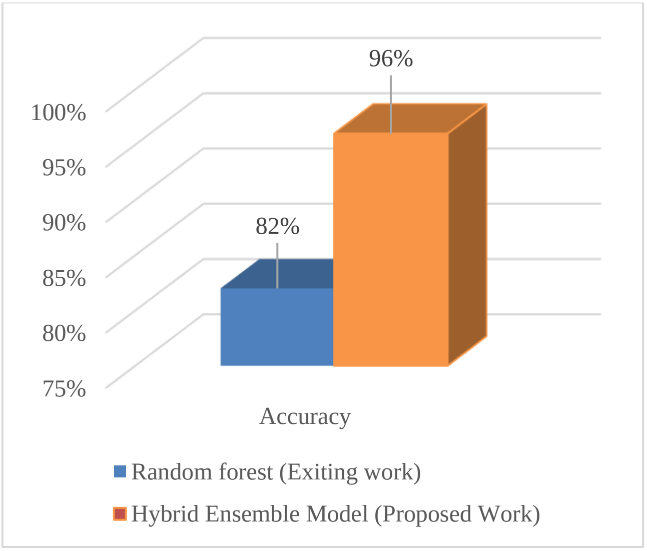


Figure 6 Graph of Comparing the Proposed method with Existing Work

CONCLUSION

In conclusion, to increase software quality and dependability, software bug predictions are a crucial area of study for static code analysis. This is a method where a model for prediction is built utilizing certain software metrics in order to forecast future software problems based on past data. Various datasets, metrics, and performance measurements have been used in the presentation of several methodologies. The goals of this investigation have been effectively met. The article evaluates the objectives, presents a thorough analysis of deep learning approaches used to software bug prediction, and employs the most effective approaches in this investigation. We used several performance metrics in order to compare and assess the effectiveness of the suggested models. According to the findings, DL approaches are becoming more and more popular in software bug predictions as a means of increasing problem detection efficiency.

First, enhance feature engineering technique to identify and incorporate new metrics that may better predict bugs is essential. Combining traditional metrics with advanced software engineering practices can yield more informative datasets.

Second, investigating transfer learning techniques could improve model performance when applied to new projects or evolving software systems. Pre-trained models on diverse datasets might provide a foundation for better predictions in specific contexts.

REFERENCE

[1] Whitten, Neal, ―Managing software development projects: formula for success, ‖ p. 384, 1995.

[2] Galin, Daniel, Software Quality Assurance From theory to implementation Software Quality Assurance From theory to implementation, 1st ed. Essex, England: Pearson Education Limited, 2004. [Online]. Available: www.pearsoned.co.uk

[3] Ran, Yan, Shen Xiaomei, and Xu Zhaowei. "Research and Application of Software Defect Prediction Model based on Data Mining." 2022 IEEE International Conference on Sensing, Diagnostics, Prognostics, and Control (SDPC). IEEE, 2022.

[4] Mahfoodh, Hussain, and Qasem Obediat. "Software risk estimation through bug reports analysis and bug-fix time predictions." 2020 International Conference on Innovation and Intelligence for Informatics, Computing and Technologies (3ICT). IEEE, 2020.

[5] 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

[6] 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.

[7] Shivaji, Shivkumar, et al. "Reducing features to improve bug prediction." 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2009.

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

[9] Shailee, Nowrin Muhaimin, et al. "Software bug prediction using machine learning on jm1 dataset." 2024 International Conference on Advances in Computing, Communication, Electrical, and Smart Systems (iCACCESS). IEEE, 2024.