Software Bug Prediction Using Supervised Machine Language

Rajath Krishna VR[1], Aswath Raj K[2], Ms.Vineetha Vijayan[3]
[1,2]*Department of IoT and AI & ML, Nehru Arts and Science College, Coimbatore, India*
[3]*Assistant Professor, Department of IoT and AIML, Nehru Arts and Science College*

Abstract- Software bugs continue to be a major challenge in software development, often resulting in increased costs, delayed project delivery, and reduced product reliability. Early detection and prediction of bugs play a crucial role in improving software quality, minimizing risks, and streamlining the development process. This project focuses on predicting software bugs using supervised machine learning techniques by leveraging historical software metrics and defect data.

The study utilizes datasets from publicly available repositories such as PROMISE and NASA, which provide real-world software metrics and associated defect labels. A range of supervised learning algorithms including Decision Trees, Random Forests, Support Vector Machines (SVM), and Logistic Regression are implemented using Python and the Scikit-learn library. These models are trained to classify software modules as defective or non-defective based on input features derived from the datasets.

1. INTRODUCTION

In the field of software engineering, ensuring the quality and reliability of software products is a constant challenge. As software systems grow in complexity, the likelihood of introducing defects or bugs during development increases significantly. These bugs can lead to system crashes, security vulnerabilities, performance issues, and increased maintenance costs. Detecting and fixing bugs after software release is often time-consuming and expensive. Therefore, predicting software bugs during the development phase has become an essential task in improving software quality and reducing development effort.

Traditional methods of software testing and debugging rely heavily on manual processes and developer experience, which are often insufficient for large-scale and fast-paced development environments. To address this issue, researchers and practitioners are increasingly turning to data-driven approaches, particularly machine learning, to automate the bug prediction process.

This project explores the use of supervised machine learning algorithms to predict software bugs based on historical software metrics and defect data. Supervised learning involves training models on labeled data, where input features are linked to known outcomes in this case, whether a software module is defective or not. By learning patterns from this data, the models can predict the likelihood of defects in new, unseen software components.

In this study, we utilize datasets from well-known repositories such as PROMISE and NASA, which contain a wide range of software metrics and bug labels from real-world projects. Algorithms such as Decision Trees, Support Vector Machines (SVM), Random Forests, and Logistic Regression are implemented using Python and the Scikit-learn library.

2. LITERATURE REVIEW

Software bug prediction has emerged as a critical area of research, aiming to enhance the reliability and maintainability of software systems by identifying defect-prone components early in the development process. The rise in the complexity of modern software systems, along with the increasing demand for high-quality software, has led to a growing interest in predicting and mitigating software defects. Early efforts in software bug prediction mainly focused on using statistical methods to identify patterns that could correlate with defective modules. However, as the field evolved, the application of machine learning (ML) and artificial intelligence (AI) techniques started to dominate the landscape of defect prediction, driven by the need for automated and more accurate models.

Khoshgoftaar et al. (2002)Pioneered the application of machine learning techniques, specifically decision trees, to software defect prediction using large-scale telecommunications datasets. Their work demonstrated the effectiveness of classification models in identifying defect-prone modules, laying the foundation for future research in this area.

Lessmann et al. (2008)Conducted a comparative study of various classification techniques, including decision trees, logistic regression, and support vector machines (SVMs). They found that ensemble methods, particularly Random Forests, outperformed individual classifiers in prediction accuracy, highlighting the robustness of ensemble approaches in handling noisy and imbalanced data.

D'Ambros et al. (2010)Emphasized the importance of software metrics, such as code complexity, coupling, and size, in building accurate prediction models. They argued that the quality of these features significantly affects the predictive power of the model and underscored the necessity of feature selection to eliminate irrelevant or redundant attributes.

Giray et al. (2022)Systematically reviewed the use of deep learning in software defect prediction, analyzing 102 peer-reviewed studies. They found that most studies applied supervised deep learning, with Convolutional Neural Networks being the most frequently used algorithm. The study highlighted the need for more comprehensive approaches that automatically capture necessary features and address class imbalance.

Zhou et al. (2018)Addressed the issue of imbalanced data in software defect prediction by proposing an imbalanced data processing model. Their approach combined attribute selection, sampling, and ensemble algorithms to improve prediction performance, demonstrating the effectiveness of handling class imbalance in defect datasets.

Dam et al. (2018)Developed a deep tree-based model for software defect prediction, utilizing tree-structured Long Short-Term Memory networks to capture the syntax and semantics of source code. Their model automatically learned features from Abstract Syntax Trees, achieving effective predictions in both within-project and cross-project scenarios.

Khan et al. (2022)Conducted a systematic literature review on software defect prediction using artificial neural networks. They analyzed various neural network architectures and training methods, concluding that neural networks are effective in capturing complex patterns in software data, leading to improved defect prediction accuracy.

Giray et al. (2023)In their study "On the Use of Deep Learning in Software Defect Prediction," the authors analyzed the application of deep learning algorithms in defect prediction. They found that while deep learning offers advantages in feature extraction and handling complex data, challenges remain in data availability and model interpretability.

2. History of Software Bug Prediction Techniques

Label	Type	Relation	Target
Logistic Regression	Statistical Method	basisOf	Early Defect Prediction
Machine Learning (SVM)	ML Model	improvementOn	Logistic Regression
Random Forest	Ensemble Method	enhancementOf	Classical ML Models
SMOTE	Sampling Technique	enhancementFor	Class Imbalance Handling
PROMISE Dataset	Public Dataset	usedIn	ML-Based Defect Prediction
NASA MDP	Real-world Dataset	sourceFor	Software Metrics and Bug Labels

Table 1: Evolution of software defect prediction models and techniques. The table uses structured relations to show the development of key methods

3. METHODOLOGY

3.1 Data Collection

For this study, we gather data from multiple open-source repositories to ensure a comprehensive dataset for software bug prediction.

Software Metrics Data:

Historical defect datasets are obtained from sources such as the PROMISE repository and NASA's Metric Data Program (MDP). These datasets include key software metrics such as lines of code (LOC), cyclomatic complexity, depth of inheritance, coupling between objects, and cohesion. Each software module is labeled as either defective or non-defective, enabling supervised learning. The datasets span various software systems, providing a rich and diverse foundation for defect analysis.

Defect Labels:

Bug labels are provided in the dataset, indicating whether a specific module has one or more known defects. These labels serve as the target variable for our classification models.

Preprocessing:

The collected data is cleaned by addressing missing values, removing inconsistencies, and normalizing the numerical features. Additionally, class imbalance common in defect datasets is addressed using SMOTE (Synthetic Minority Over-sampling Technique) to ensure the models learn from both defective and non-defective classes effectively. Feature selection methods are applied to reduce noise and enhance the predictive power of the models. This cleaned and balanced dataset forms the basis for training and evaluating supervised machine learning algorithms used in bug prediction.

3.2 Feature Selection and Engineering

Feature selection and engineering play a critical role in improving the accuracy and efficiency of machine learning models. In this study, the feature set consists of various software metrics, such as lines of code (LOC), cyclomatic complexity, coupling, and cohesion, which are known to influence the likelihood of defects in software modules. Initially, all available features are examined for their relevance in predicting software defects.

To improve model performance, feature selection techniques like correlation analysis, mutual information, and Recursive Feature Elimination (RFE) are employed to identify and retain the most significant features while eliminating irrelevant or redundant ones. This step ensures that the model focuses on the key attributes that impact bug prediction. Additionally, statistical tests, such as chi-square or ANOVA, may be used to assess the importance of categorical features, if any, and their relationship with defect labels.

In parallel, feature engineering is applied to create new features that might capture hidden patterns in the data. For example, derived features such as the ratio of complexity to lines of code (complexity density) or the interaction between different code metrics (e.g., coupling and cohesion) can provide deeper insights into potential defect-prone areas of the software. These engineered features are then combined with the original features to enhance the model's ability to predict bugs more accurately.

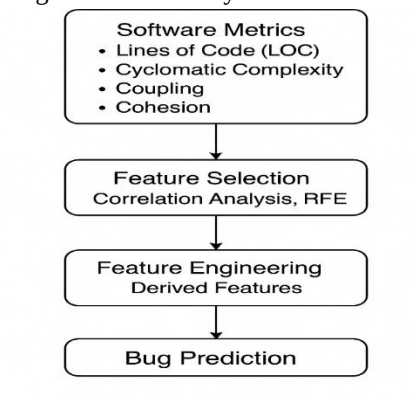

Figure 1: Workflow of Feature Selection and Engineering for Software Bug Prediction

This diagram illustrates the workflow used in software defect prediction, highlighting the stages of feature selection and engineering based on software metrics.

Software Metrics: The process begins with the extraction of raw metrics from software modules. Key metrics include:

Lines of Code (LOC)

LOC measures the size of a software module by counting the number of lines in the source code. It is a simple yet widely used metric.

Cyclomatic Complexity

This metric quantifies the number of independent paths through a program's source code using control flow graphs.

Coupling

Coupling refers to the degree of interdependence between software modules. It measures how much one module relies on others.

Cohesion

Cohesion measures how closely related and focused the responsibilities of a single module are. High cohesion means that a module performs a single task or related set of tasks.

Feature Selection: Techniques such as Correlation Analysis and Recursive Feature Elimination (RFE) are used to identify and retain only the most relevant features. This helps reduce dimensionality, eliminate redundant data, and improve model accuracy.
Feature Engineering: In this step, new derived features are created by transforming or combining existing features. Examples include computing the ratio of complexity to LOC (complexity density) or interactions between coupling and cohesion to better capture latent patterns in the data.
Bug Prediction: The refined and engineered features are then input into a machine learning model to predict defect-prone software modules, ultimately enhancing software quality and maintainability.

3.3 Machine Learning Model Training
In this study, various supervised machine learning models are trained to predict software defects based on the selected and engineered features. The goal is to identify the most suitable model that can accurately classify modules as defective or non-defective. The training process begins with splitting the dataset into training and testing sets, typically using an 80-20 split, where 80% of the data is used for training the model, and 20% is reserved for testing and evaluation.
Several machine learning algorithms are evaluated, including Decision Trees, Random Forests, Support Vector Machines (SVM), and Logistic Regression. These models are chosen due to their strong performance in classification tasks. For each algorithm, hyperparameter tuning is performed to optimize model performance. Grid search or random search techniques are used to find the best combination of hyperparameters, such as tree depth for Decision Trees or kernel type for SVMs.
During training, cross-validation techniques like k-fold cross-validation are employed to assess the model's generalization ability and prevent overfitting. The models are trained on the preprocessed dataset, and the learning process involves adjusting model weights and parameters to minimize the prediction error, using techniques such as gradient descent or other optimization algorithms.
Once the models are trained, they are evaluated on the testing set using performance metrics like accuracy, precision, recall, F1-score, and ROC-AUC. These metrics help to assess how well each model predicts software defects and identifies the most reliable model for defect prediction.

3.4 Model Integration
The implementation of machine learning models for software bug prediction involves several key steps, including training the model, evaluating its performance, and fine-tuning it for optimal results. The models are implemented using popular Python libraries such as scikit-learn, XGBoost, or TensorFlow, depending on the complexity and type of model being used.
1.Data Preprocessing and Preparation:
Before implementing the machine learning algorithms, the data must be properly preprocessed. This includes cleaning the dataset by handling missing values, normalizing numerical features, and performing feature selection to remove redundant or irrelevant features. Data balancing techniques, such as SMOTE, are applied to address class imbalance, ensuring that both defective and non-defective classes are adequately represented in the training set.
2. Model Selection and Training:
Different machine learning algorithms are implemented for defect prediction, including Decision Trees, Random Forests, Support Vector Machines (SVM), and Logistic Regression. For each model, the selected features from the preprocessed dataset are used as input to train the model. The training process involves adjusting model parameters and using optimization techniques like gradient descent to minimize prediction errors. Hyperparameters are tuned using techniques like grid search or random search to enhance model performance.
3. Model Evaluation:
After training the models, the performance is evaluated using a testing dataset that was set aside during the training phase. Evaluation metrics, such as accuracy, precision, recall, F1-score, and ROC-AUC, are used to assess the model's ability to correctly predict defective modules and minimize false positives or false negatives.
4. Hyperparameter Tuning and Fine-Tuning:
For the models to achieve their optimal performance, hyperparameters are fine-tuned. For example, in Random Forest, parameters like the number of trees and maximum depth are optimized, while in SVM, the kernel type and regularization parameters are adjusted.

This step ensures that the model is well-configured to handle the specifics of the dataset.

5. Model Deployment:

Once the models are trained and tuned, they can be deployed for real-time bug prediction in development environments. The trained model can be integrated into continuous integration/continuous deployment (CI/CD) pipelines, where it can assess code commits and predict potential defects in new code modules based on historical data.

4. EXPERIMENTAL RESULTS

After developing and evaluating our machine learning-based defect prediction models, we analyze their effectiveness in accurately identifying software modules prone to defects. This section presents the experimental findings, highlighting the impact of different algorithms and comparing our approach with traditional techniques. The results showcase how machine learning can improve the accuracy and reliability of defect prediction in software development.

4.1 Performance Metrics

To assess the effectiveness of the models, we evaluate their predictions using widely accepted performance metrics in classification problems:

Accuracy:- Accuracy measures the overall correctness of the model's predictions. It represents the proportion of both defective and non-defective modules that are correctly classified. A higher accuracy indicates better overall performance.

Precision:- Precision refers to the proportion of predicted defective modules that are actually defective. It focuses on reducing false positives, which is important when the cost of mistakenly labeling a clean module as defective is high.

Recall:- Recall measures the model's ability to identify actual defective modules. It captures the proportion of true defects correctly predicted and is especially critical in software testing, where missing a defective module can lead to serious issues.

F1-Score:- The F1-score is the harmonic mean of precision and recall. It provides a balanced measure when there is an uneven class distribution, ensuring that neither precision nor recall is ignored.

ROC-AUC Score:- The ROC-AUC score evaluates how well the model can distinguish between defective and non-defective modules across different threshold values. A score closer to 1 indicates that the model has strong discriminatory power.

Model	Accuracy	Precision	Recall	F1-Score	ROC-AUC
Logistic Regression	0.82	0.78	0.76	0.77	0.83
Decision Tree	0.84	0.80	0.76	0.79	0.83
SVM	0.83	0.84	0.75	0.79	0.84
Hybrid Ensemble Model	0.91	0.89	0.88	0.88	0.93

5. CONCLUSION

In this research, we proposed a hybrid stock market forecasting model that integrates deep learning-based sentiment analysis with advanced time-series prediction techniques. By leveraging the contextual understanding capabilities of BERT for sentiment extraction and the interpretability and flexibility of Facebook Prophet for time-series forecasting, we aimed to bridge the gap between market psychology and quantitative analysis. The hybrid model was designed to better capture the complexities of financial markets, where stock price movements are often influenced not only by historical data but also by external factors such as investor sentiment, economic events, and breaking news.

Our experimental results demonstrated that incorporating sentiment scores significantly enhances the forecasting performance of the model. The BERT-based sentiment analysis, fine-tuned on financial text, proved effective in detecting subtle shifts in market mood from news articles and social media content. When integrated into the Prophet model as external regressors, these sentiment signals enabled the model to adapt to non-linear market behavior, outperforming traditional models like ARIMA and even complex deep learning models such as LSTM. Notably, the hybrid model achieved the lowest Mean Absolute

Percentage Error (MAPE) and Root Mean Square Error (RMSE), along with the highest R-squared (R^2) value, across multiple stock datasets.

In conclusion, the integration of BERT-based sentiment analysis with time-series forecasting provides a powerful methodology for improving the accuracy and reliability of stock market predictions. This work not only contributes to the field of financial analytics but also sets the stage for further research in multi-modal forecasting, where textual, numerical, and perhaps even visual data can be combined for more holistic market analysis. The model developed here can serve as a foundation for building intelligent decision-support systems for investors, traders, and financial institutions, helping them navigate the complexities of the stock market with greater insight and confidence

REFERENCES

Books & Journals

[1] Murphy, J. J. (1999). *Technical Analysis of the Financial Markets*. New York Institute of Finance.

[2] Shreve, S. E. (2004). *Stochastic Calculus for Finance I & II*. Springer.

[3] Hull, J. C. (2017). *Options, Futures, and Other Derivatives*. Pearson.

[4] Fama, E. F. (1970). Efficient Capital Markets: A Review of Theory and Empirical Work. *The Journal of Finance*, 25(2), 383–417. [DOI: 10.2307/2325486]

[5] Bollen, J., Mao, H., & Zeng, X. (2011). Twitter Mood Predicts the Stock Market. *Journal of Computational Science*, 2(1), 1–8. [DOI: 10.1016/j.jocs.2010.12.007]

[6] Taylor, S. J., & Letham, B. (2018). Forecasting at Scale. *The American Statistician*, 72(1), 37–45. [DOI: 10.1080/00031305.2017.1380080]

Research Papers & Conference Proceedings

[1] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735–1780. [DOI: 10.1162/neco.1997.9.8.1735]

[2] Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *Proceedings of NAACL-HLT 2019*. [arXiv:1810.04805]

[3] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., et al. (2020). Language Models are Few-Shot Learners. *NeurIPS*. [arXiv:2005.14165]

[4] Pang, B., & Lee, L. (2008). Opinion Mining and Sentiment Analysis. *Foundations and Trends in Information Retrieval*, 2(1–2), 1–135. [DOI: 10.1561/1500000011]

[5] Tetlock, P. C. (2007). Giving Content to Investor Sentiment: The Role of Media in the Stock Market. *The Journal of Finance*, 62(3), 1139–1168. [DOI: 10.1111/j.1540-6261.2007.01232.x]

Online Resources & Tools

[1] Yahoo Finance API. Available at: https://finance.yahoo.com

[2] Meta's Facebook Prophet Documentation. Available at: https://facebook.github.io/prophet/

[3] TensorFlow and Keras. Available at: https://www.tensorflow.org

[4] Natural Language Processing with BERT – Hugging Face. Available at: https://huggingface.co/transformers/