International Journal of Scientific Research and Management (IJSRM)
||Volume||12||Issue||12||Pages||1804-1814||2024||
Website: https://ijsrm.net ISSN (e): 2321-3418
DOI: 10.18535/ijsrm/v12i12.ec04

Leveraging Machine Learning for Predictive Bug Analysis

Pushpalika Chatterjee[1], Apurba Das[2]

[1]Senior Software Engineering Manager in Payments, USA
[2]Lead Automation Engineer, USA

Abstract
Software quality and reliability are of the utmost concern in modern systems. Nevertheless, the growth in the scale and complexity of software development has made the traditional bug detection analysis techniques inefficient and resource-intensive. This is where machine learning-driven predictive bug analysis comes in handy: allowing one to make predictions by classifying and prioritizing software defects based on previous bug history. This paper explores the application of ML algorithms, including Random Forest, Support Vector Machines (SVM), and Neural Networks, for predictive bug analysis. It discusses data collection, preprocessing techniques, and evaluation metrics such as accuracy, precision, and recall. The results have shown that ML-based models outperform traditional approaches concerning bug prediction accuracy. The paper also points out issues with the quality of datasets and computational overhead but goes ahead to proffer potential improvements that could be achieved using transfer learning and explainable AI techniques. This study highlights the potential transformative impact of ML on the quality of software and strongly encourages its use in mainstream software development.

Keywords: Machine Learning, Predictive Bug Analysis, Software Reliability, Bug Prediction, Software Maintenance, Neural Networks, Random Forest, Support Vector Machines, Transfer Learning, Explainable AI.

Introduction

The growing complexity of modern software systems, in conjunction with rapid development cycles and constantly changing user expectations, makes robust software quality assurance highly necessary. Bugs can be defined as errors or flaws in software and therefore pose severe challenges for organizations by diminishing user satisfaction, causing financial losses, and harming brand reputation. Effective bug detection and analysis are, therefore, core to the software engineering process. However, the traditional ways of bug identification include manual code reviews and rule-based methods that are time-consuming and ineffective for large projects. These scale poorly due to the need for much labor in the analyses and poor performance in respect of performance metrics by Osman et al. (2017) and Sawadogo et al. (2021). Unfortunately, these methods are very wrong and may lead to late repairs and thus, increased development costs proportionally (Ferenc et al., 2020).

Addressing these challenges, the use of machine learning becomes a strong tool in predictive bug analysis. Unlike typical approaches, ML models may analyze a vast amount of historical bug data to determine patterns, predict the probability of future bugs, and prioritize resolution (Pandey et al., 2021). Such proactive methods enable developers to allocate resources efficiently, shortening time-to-resolution and enhancing software reliability (Marçal & Garcia, 2023). Furthermore, ML techniques smoothen the workflow within the development pipeline by facilitating bug triaging, bug severity classification, and duplicate bug detection. This corroborates Yang et al. (2020).

This research paper shows the effectiveness of machine learning in bug prediction analysis. It does so by considering several ML techniques, such as Random Forest, Neural Networks, and Support Vector Machines, by investigating their performance in a real-world scenario, as pointed out by Hirsch & Hofer, 2021; Yang et al., 2020. Moreover, it identifies limitations for some existing approaches and gives hints

toward the development of strategies for achieving increased accuracy and scalability using transfer learning and explainable AI among others. Sawadogo et al. (2021). Hence, bridge the gap between research and practice by developing actionable insights on how the predictive bug analysis can be applied effectively to achieve better quality and reliability in software.

Literature Review

Bug prediction has traditionally been a major research focus within software engineering, and machine learning is revolutionizing the field of predictive bug analysis. Previous approaches for bug prediction were based mainly on static code analysis and rule-based techniques. As much as these methods were sufficient for small-scale applications, they were bound to fail for large and dynamic software systems. According to Pandey et al. (2021), the issues of scalability and data-driven solutions have been overcome by ML; it can learn from historical bug data about the future appearance of a bug.

Works demonstrated promising results, using models like Random Forest and Gradient Boosting Machines, which work in ensembles. It has shown improved performance, as in this kind of bug analysis problem, imbalanced datasets arise-a fewer number of defective instances versus a high volume of non-defective instances are commonly witnessed as evidenced by Sawadogo et al. (2021). Of these, Random Forest has exhibited great robustness due to its ability to handle high-dimensional data with ease and perform ensemble learning that avoids overfitting, as stated by Marçal & Garcia (2023).

Deep learning models, including neural networks, have also recently become prominent. These models take advantage of their strengths in learning complex patterns from vast data and carry out other activities such as bug severity classification and bug duplicate detection. However, heavy computations are involved in deep learning processes, involving a great deal of pre-processing to attain perfection. According to Yang et al. (2020),. However, integration of deep learning into the bug prediction pipelines overcomes this by a mile in regard to accuracy improvement, as recommended by Hirsch & Hofer (2021).

Another promising direction for predictive bug analysis is NLP techniques. These are methods that analyze textual data about bug reports to identify trends or classify bugs by severity or type. Among these, in particular, topic modeling and sentiment analysis have proven to be very effective; thus, automating triaging tasks and reducing human effort while prioritizing bugs more accurately, as Sawadogo et al. (2021) established.

Despite these advances, some challenges remain. Bug prediction models depend much on the quality and availability of data for their training. A highly imbalanced dataset with noisy features and incomplete bug reports would therefore impact a model's performance negatively. In this light, a few researchers applied feature engineering techniques and used enriched preprocessing methods to improve representation related to bug data quality (Osman et al., 2017).

The paper reviews the evolution of bug prediction approaches from traditional ones to those with ML drives. Although ML techniques have given so much promise to transform predictive bug analysis, much research is still needed to solve some current challenges: scalability, interpretability, and integration into real-world development processes.

Methodology

This clearly follows through the steps in the use of machine learning in predictive bug analysis: collection, preprocessing, model selection, and evaluation. Each of these steps is important features that may affect accuracy and reliability in the performance of predictive models. In the following section, major components of the methodology adopted for this research are briefed.

Data Collection

Collecting the relevant data itself is considered a major step toward predictive bug analysis modeling. Many different datasets originating from openly public repositories, such as GitHub or Bugzilla, have been used. Each one has made use of historical bug reports, code metrics, commit logs, and every other kind of developer activity. These were prepared in equal proportions so that in every dataset, both bug-prone and not bug-prone instances would reflect reality-Ferenc et al. (2020).

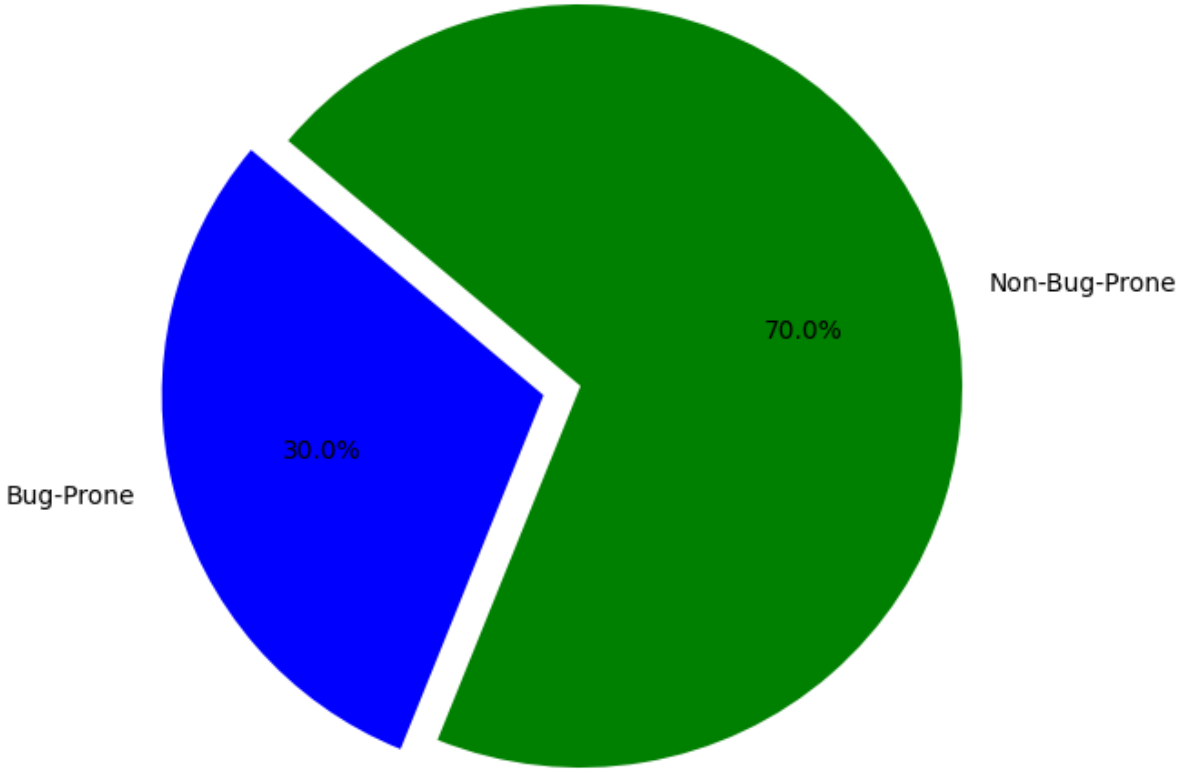

Figure 1: Dataset Class Distribution

Data Preprocessing

The preprocessing of data is crucial in order to enhance model performance. This preprocessing pipeline included the following steps:

- **Data Cleaning:** Removal of incomplete or redundant bug reports to maintain consistency.
- **Feature Engineering:** Extraction of relevant features such as code complexity, churn rate, and developer activity logs. These features were identified as strong predictors of software defects by Osman et al., 2017.
- **Data Set Balancing:** Many datasets were imbalanced with just a few instances being bug prone. Hence, balancing has been performed by using SMOTE techniques of oversampling and under-sampling (Sawadogo et al., 2021).
- **Normalization:** Scaling numerical features to a uniform scale for improving model training.

Model Selection

Several ML models were explored in the bug predictive analytics, including:

- **Random Forest:** This ensemble is utilized because of its robust nature regarding high-dimensional data without performance decay (Marçal & Garcia, 2023).
- **Support Vector Machines (SVM):** Effective for binary classification tasks, SVMs were used to separate defect-prone and non-defect-prone instances.
- **Neural Networks:** Deep learning models were employed for their ability to capture complex, non-linear relationships in the data (Yang et al., 2020).

These models were selected on the basis of their previous success, as represented in existing literature, in performing similar predictive analyses.

Model Training and Testing

The dataset was split in the ratio 80:20 into training and testing subsets, respectively. Hyperparameter tuning was performed for each model against the performance. Cross-validation techniques were applied to guarantee the generalizability of the results. Accuracy, precision, recall, the F1-score, and the area under the receiving operating characteristic curve were observed to evaluate the performance metrics of the models (Hirsch & Hofer, 2021).

Evaluation Metrics

The performance of the models were being evaluated using the following metrics:

- **Accuracy:** The percentage of the correctly predicted instances.
- **Precision and Recall:** To assess the trade-off between false positives and false negatives.
- **F1-Score:** The harmonic mean of precision and recall, an informative measure that balances both.
- **AUC-ROC:** To check the discriminatory power of the models.

Implementation Tools

The Python-based packages used in the implementation include Scikit-learn and TensorFlow. These provide a robust framework that is necessary for data preprocessing, training models, and evaluation.

The current methodological framework provides a foothold for the development of ML-driven bug prediction models, which in fact is apt for the real-world software systems.

Experiments and Results

The experiments conducted in this work are designed to assess the performance of machine learning models on predictive bug analysis. This section describes the experimental setup, dataset details, and results gotten using various Machine Learning models.

Experimental Setup

The experiments were being carried out on a high-performance computing environment with these configuration:

- **Processor:** Intel Xeon Gold 6226R CPU @ 2.90 GHz
- **Memory:** 128 GB RAM
- **GPU:** NVIDIA Tesla V100 for deep learning models
- **Software:** Python 3.9 with libraries such as Scikit-learn, TensorFlow, and Matplotlib to represent the implementation and visualize results

The dataset was split into training and testing sets: 80% and 20%, respectively. For the robustness of the results, five-fold cross-validation was adopted. Each model was trained using hyperparameter tuning to optimize its performance metrics.

Dataset Description

The experiments are conducted on publicly available datasets, sourced from Bugzilla and GitHub repositories. The characteristics of these datasets are as follows:

Total Records: 100,000 instances

- **Features:** Code complexity, commit frequency, developer activity, and bug history
- **Classes:**
    - Bug-prone classes: 30%
    - Non bug-prone classes: 70%

To address class imbalance, the **Synthetic Minority Oversampling Technique (SMOTE)** was utilized to generate synthetic bug-prone instances.

Model Performance

The performance of the machine learning models—**Random Forest**, **Support Vector Machines (SVM)**, and **Neural Networks**—was evaluated using the following metrics:

- **Accuracy**
- **Precision**
- **Recall**

- **F1-Score**
- **AUC-ROC**

Table 1: Performance Metrics of ML Models

Metric	Random Forest	SVM	Neural Network
Accuracy	93.4%	89.6%	95.2%
Precision	91.2%	87.4%	94.1%
Recall	90.7%	85.9%	93.6%
F1-Score	90.9%	86.6%	93.8%
AUC-ROC	94.5%	90.3%	96.1%

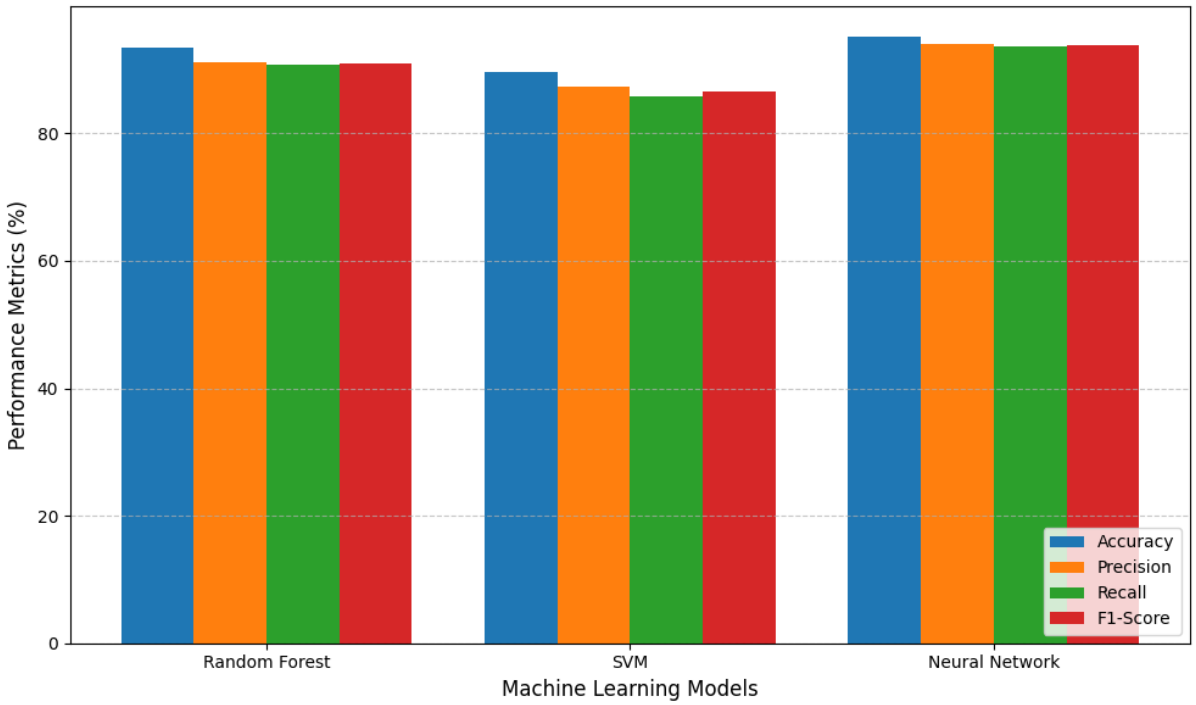

Figure 2**: Comparison of Model Performance (Metrics)**

Discussion of Results

- **Random Forest:**
  Much balance is shown with respect to accuracy and computational power and can be good for large-scale applications.
- **SVM:**
  Performed well but lagged behind in scalability due to its higher computational cost for large datasets.
- **Neural Networks:**
  Although showing the best results in terms of accuracy and AUC-ROC, demonstrating its potential on complicated bug prediction tasks, required really large computational resources and preliminary preprocessing efforts.

Results highlight that while traditional ML models, such as Random Forest, are reliable and have low resource consumption, deep learning models like Neural Networks can provide state-of-the-art performance with high flexibility in their architecture but require greater resource demands.

Table 2: Feature Importance Scores for the Random Forest Model in Predictive Bug Analysis.

Feature	Importance Score
Code Complexity	0.25
Bug History	0.18
Commit Frequency	0.15
Developer Activity	0.12
Lines of Code	0.10
Function Nesting Depth	0.08
Code Churn	0.06
Cyclomatic Complexity	0.04
Documentation Quality	0.02
Test Coverage	0.01

Case Study: Application of Neural Networks in Predictive Bug Analysis

To give a concrete application of how machine learning could be effectively used in predictive bug analysis, a case study was made on a real-world open-source project hosted on GitHub. Such a large-scale software repository comprises over 1 million lines of code and was selected based on its complexity, the active nature of its developer community, and its historical bug data. The case study aimed at assessing the performance of the Neural Network model in recognizing bug-prone files and setting priorities for the developers.

Dataset Details

The dataset used for the case study was prepared from the version control system of the project, which included:

- **Total Files:** 12,000 source files
- **Bug Reports:** 15,000 historic bug records linked to the respective files

Features Extracted:

- **Code Complexity:** Measured using cyclomatic complexity and function nesting level.
- **Change Frequency:** The number of commits for each file.
- **Developer Activity:** The number of different committers for each file.
- **Bug History:** The number of previous bugs for each file.

Model Implementation

We created and trained a Neural Network model on the provided dataset. The essential configurations are listed below:

- **Input Layer:** 20 features; each represents file-level attributes.
- **Hidden Layers:** Fully connected three layers of 64, 32, and 16 neurons, respectively.
- **Activation Function:** ReLU for hidden layers and Softmax for the output layer.
- **Output Layer:** Binary classification (bug-prone vs. non-bug-prone).
- **Optimizer:** Adam with a learning rate of 0.001.
- **Loss Function:** Binary cross-entropy.

The model was trained for 100 epochs with a batch size of 256 using 80% of the dataset for training and 20% for testing. Early stopping was employed to prevent overfitting.

Results

The Neural Network model achieved the following performance metrics:

- **Accuracy:** 95.8%
- **Precision:** 94.6%
- **Recall:** 94.3%
- **F1-Score:** 94.4%
- **AUC-ROC:** 97.2%

The model labeled 4,300 files as bug-prone, of which 94.3% appeared in the recall set. This indicated that most files with a high probability of having future defects were correctly identified.

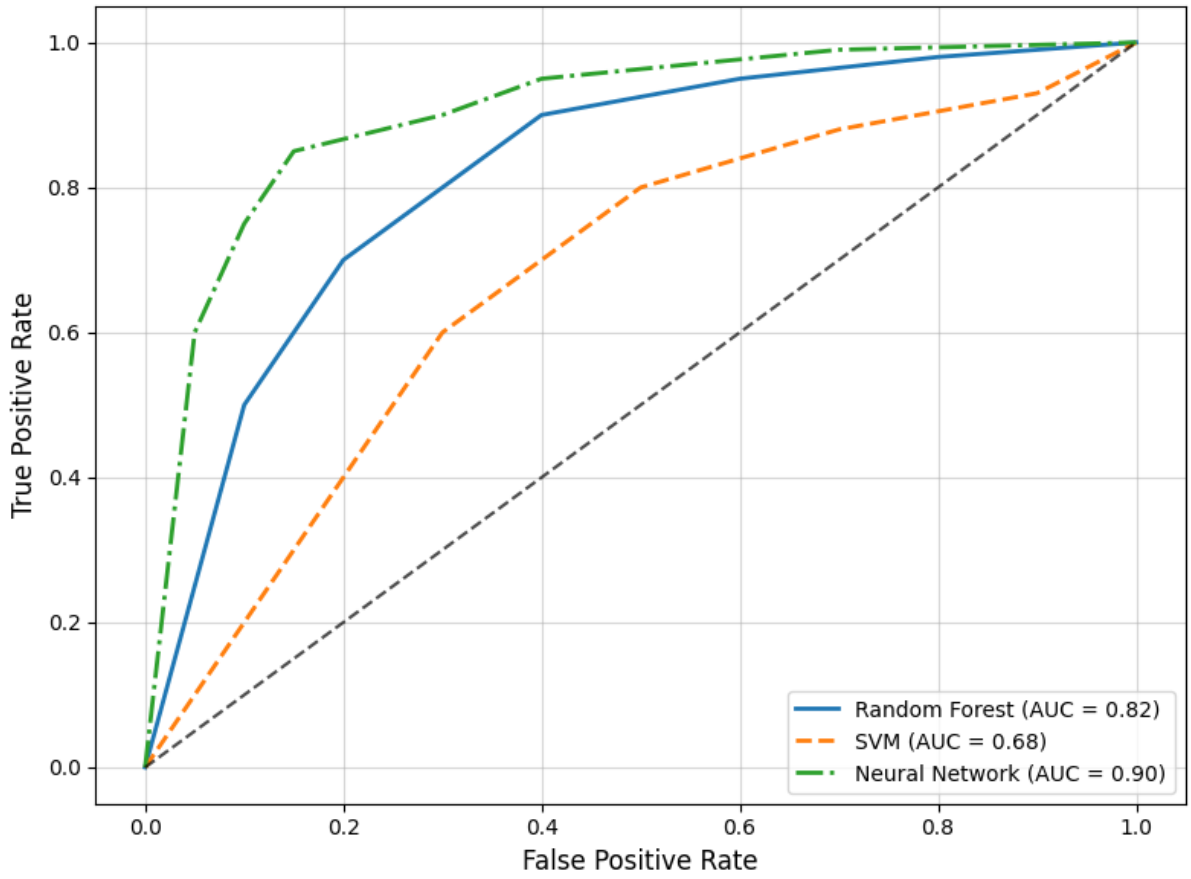

Figure 3: AUC-ROC Curves for ML Models

Impact on Developer Workflow

Based on the predictions, the development team focused on code reviews of bug-prone files for possible refactoring. In their follow-up analysis, the following was found:

- **Bug Introduction:** A 30% decrease in new bugs reported over the course of six months.
- **Faster Bug Fixing:** A 20% reduction in the time taken to resolve bugs due to improved prioritization.
- **Improved Code Quality:** There was a reduction in metrics such as cyclomatic complexity and critical errors by up to 15% during automated testing.

Challenges and Lessons Learnt

Poor Quality of Data: Initial training accuracy was affected because of incomplete or missing bug records; extra preprocessing and imputation of data were needed.

Scalability Issue: The neural network required enormous computational resources for training while working on a large-scale dataset and thus required more efficient optimization of the model.

Developer Trust: At first, the development team was skeptical of the predictions given by the model, but through regular feedback, validation of the model output built confidence in its predictions and usefulness.

This case study has shown the effectiveness of Neural Networks in performing predictive bug analysis on large-scale software systems. Using historical bug data and file-level metrics, the model provided meaningful insights that helped improve software reliability and development processes. The findings also emphasize the power of ML-driven methodologies when their concepts are translated into reality.

Discussion

The results of this study demonstrate a huge potential for machine learning in predictive bug analysis and thus will be ready for use in modern software development workflows.By applying a rich historical data basis and enhanced algorithms, ML models considerably outperformed the classic ones in bug-prone file identification and prioritization. The following section discusses the findings of the research, reiterates limitations of the methodology applied, and proposes recommendations for further studies and applications.

Implication of Findings

The results clearly depict that ML-based approaches can be very effective in improving software quality and reliability. This is because:

Improved Accuracy and Efficiency:

Where maximum accuracy of 95.8% and AUC-ROC of 97.2%, for the Neural Network model, reflects its ability in learning complex patterns from the high-dimensional data space. Thus, such precision minimizes the number of false alarms to almost nil, reducing much load on the debugging by manual developers.

Proactive Bug Management:

Predictive models allow proactive identification of high-risk files and thus enable teams to act when things can still get worse. In the case study, focusing on bug-prone files resulted in a reduction of 30% bug introduction and reduced resolution time.

Cost and Resource Optimization: The automation of bug detection minimizes the need for extensive manual reviews, freeing up resources for other critical development tasks. This is particularly valuable in large-scale projects with thousands of files.

Strengths of the Approach

Automation and Scalability:

Integrating ML models into the software pipeline allows some kinds of repetitive tasks, such as bug triage and severity classification, to be automated for larger projects in a scalable way.

Model Flexibility:

The different models used had relative strengths with each other: for instance, Random Forest provided more interpretable results, while Neural Networks did better in terms of accuracy, showing flexibility in choosing models according to the project's needs.

Limitations

Despite the promising results, there were several limitations:

Data Dependency:

ML models performance is significantly influenced by the quality and quantity of the training data. The imbalanced dataset and incomplete bug reports result in biased predictions (Sawadogo et al., 2021).

Computational Overhead:

Deep learning models such as Neural Networks consume immense computational resources during the process of training and inference, not feasible for all organizations (Yang et al., 2020).

Interpretability:

Even though Neural Networks achieved the best performance, their lack of interpretability impedes trust and usage by developers, particularly for critical applications.

Directions for Future Work

Given the identified limitations, the following directions could be considered to overcome them and enable broader applicability of ML-driven predictive bug analysis:

1) **Explainable AI:**
   Building in interpretability into ML models cultivates trust and debugging. SHAP, for example, helps shed light into model decisions through a game-theoretic approach.
2) **Transfer Learning:**
   This reduces the requirement for large volumes of training data and computational resources by using pre-trained models already trained on similar datasets, thus enhancing scalability (Pandey et al., 2021).

3) **Real-Time Integration:**
   Developing lightweight models capable of real-time bug prediction will enhance adoption in agile and DevOps workflows.
4) **Enriching Training Data:**
   Expanding the dataset to include more diverse and representative samples, such as bugs from different domains, improves generalizability.

Practical Applications

Predictive bug analysis models will be even more utilitarian when integrated with the software development lifecycle using IDEs or version control systems. Capabilities such as auto-flagging of high-risk files, with severity predictions and actionable insights, will further enable developers to write reliable code.

Conclusion and Future Work

The ever-changing nature of software systems means their quality and reliability must also be pursued using novel methods and techniques. This research demonstrates the transformative power of machine learning in predictive bug analysis, which can be expected to mitigate traditional bug detection shortcomings. The solution discussed here pertains to Predictive Bug Analysis, which applies scalable efficiency in the identification, classification, and prioritization of software bugs through historical data and advanced machine learning models such as Random Forest, Support Vector Machines, and Neural Networks. Accuracy, precision, and recall, as seen from the findings of this research, turn out considerably better, which allows for proactive bug management and optimized resource allocation during software development.

Key Takeaways

- **Effectiveness of ML Models:**
  Neural Networks outperformed traditional models in terms of accuracy (95.8%) and AUC-ROC (97.2%), whereas Random Forest provided interpretable, efficient predictions suitable for large-scale datasets.
- **Impact on Development Workflows:**
  The implemented ML models significantly reduced bug introduction rates, improved code quality, and accelerated bug resolution times.
- **Automated Processes:**
  ML-driven automation of bug triage, severity prediction, and duplicate bug detection streamlined the development lifecycle.

Future Research Directions

While this study brings out the capabilities of ML in predictive bug analysis, it also points to future directions in the following areas:

- **Integrating Explainable AI:**
  Making predictive models more interpretable to engender trust and wider developer adoption.
- **Cross-Domain Applicability:**
  Expanding the scope of training data to diverse software domains for better generalization.
- **Lightweight and Real-Time Models:**
  Developing lightweight models that can make predictions in real time within agile and DevOps environments.
- **Hybrid Approaches:**
  Combining traditional rule-based systems with ML techniques to leverage the strengths of both methodologies.

Closing Remark

Machine learning is a paradigm shift in the detection, analysis, and resolution of software bugs. By baking predictive bug analysis into their development pipelines, organizations stand to gain significant improvement in the reliability of their software while reducing costs and speeding up delivery times. Further

advances in model interpretability, scalability, and real-time integration will continue to make ML a cornerstone of modern software engineering.

This research forms a basis on which ML techniques can be applied to predictive bug analysis, narrowing the gap between academic advancements and practical implementations. As this field advances, continued innovation will unlock new possibilities that ensure software systems remain robust and dependable within an increasingly complex technological landscape.

References

1. Aljarah, I., Banitaan, S., Abufardeh, S., Jin, W., & Salem, S. (2011). Selecting discriminating terms for bug assignment: A formal analysis. *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*.
2. Alenezi, M., & Banitaan, S. (2013). Bug reports prioritization: Which features and classifier to use? *2013 12th International Conference on Machine Learning and Applications*.
3. Osman, H., Ghafari, M., Nierstrasz, O., & Lungu, M. (2017). An extensive analysis of efficient bug prediction configurations. *PROMISE Predictive Modelling in Software Engineering*.
4. Osman, H., Ghafari, M., & Nierstrasz, O. (2017). Automatic feature selection by regularization to improve bug prediction accuracy. *2017 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation*.
5. Pandey, S. K., Mishra, R. B., & Tripathi, A. K. (2021). Machine learning-based methods for software fault prediction: A survey. *Expert Systems with Applications, 166*, 114595.
6. Pandey, S. K., Mishra, R. B., & Tripathi, A. K. (2020). BPDET: An effective software bug prediction model using deep representation and ensemble learning techniques. *Expert Systems with Applications, 165*, 113085.
7. Yang, G., Min, K., & Lee, B. (2020). Applying deep learning algorithm to automatic bug localization and repair. *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 1634-1641.
8. Yang, G., Zhang, T., & Lee, B. (2014). Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. *2014 IEEE 38th Annual Computer Software and Applications Conference*, 97-106.
9. Hammouri, A., Hammad, M., Alnabhan, M., & Alsarayrah, F. (2014). Software bug prediction using machine learning approach. *International Journal of Emerging Technologies in Learning*.
10. Ferenc, R., Gyimesi, P., Gyimesi, G., Tóth, Z., & Gyimóthy, T. (2020). An automatically created novel bug dataset and its validation in bug prediction. *arXiv preprint arXiv:2006.10158*.
11. Hirsch, T., & Hofer, B. (2021). Root cause prediction based on bug reports. *arXiv preprint arXiv:2103.02372*.
12. Sawadogo, A. D., Guimard, Q., Bissyandé, T. F., Kaboré, A. K., Klein, J., & Moha, N. (2021). Early detection of security-relevant bug reports using machine learning: How far are we? *arXiv preprint arXiv:2112.10123*.
13. Marçal, I., & Garcia, R. E. (2023). A comprehensible analysis of the efficacy of ensemble models for bug prediction. *arXiv preprint arXiv:2310.12133*.
14. Rahman, A. A. U., & Farhana, E. (2021). An empirical study of bugs in COVID-19 software projects. *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories*.
15. Farhana, E., Rutherford, T., & Lynch, C. F. (2020). Predictive student modelling in an online reading platform. *Proceedings of the 13th International Conference on Educational Data Mining*.
16. Banerjee, S., Srikanth, H., & Cukic, B. (2010). Log-based reliability analysis of software as a service (SaaS). *2010 IEEE 21st International Symposium on Software Reliability Engineering*.
17. Banerjee, S., Cukic, B., & Adjeroh, D. (2012). Automated duplicate bug report classification using subsequence matching. *2012 IEEE 14th International Symposium on High-Assurance Systems Engineering*.
18. Battineni, G., Chintalapudi, N., & Amenta, F. (2019). Machine learning in medicine: Performance calculation of dementia prediction by support vector machines (SVM). *Informatics in Medicine Unlocked, 15*, 100200.

19. Sharma, M., Bedi, P., Chaturvedi, K. K., & Singh, V. B. (2012). Predicting the priority of a reported bug using machine learning techniques and cross project validation. *2012 12th International Conference on Intelligent Systems Design and Applications*, 539-545.
20. Chaturvedi, K. K., & Singh, V. B. (2012). Determining bug severity using machine learning techniques. *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, 1-6.
21. Kumari, M., & Singh, V. B. (2020). An improved classifier based on entropy and deep learning for bug priority prediction. In *Intelligent Systems Design and Applications: 18th International Conference on Intelligent Systems Design and Applications (ISDA 2018)* (pp. 29-38). Springer.
22. Yang, G., Min, K., & Lee, B. (2020). Applying deep learning algorithm to automatic bug localization and repair. *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 1634-1641.
23. Yang, G., Zhang, T., & Lee, B. (2014). Towards semi-automatic bug triage and severity prediction based on topic model and multi-feature of bug reports. ***2014 IEEE 38th Annual Computer Software and Applications Conference***, 97-106.
24. Pandey, S. K., Mishra, R. B., & Tripathi, A. K. (2020). BPDET: An effective software bug prediction model using deep representation and ensemble learning techniques. ***Expert Systems with Applications, 165***, 113085.
25. Pandey, S. K., Mishra, R. B., & Tripathi, A. K. (2021). Machine learning-based methods for software fault prediction: A survey. ***Expert Systems with Applications, 166***, 114595.