Early Stage Bug Detection and Triaging using Machine Learning

Thesis Submitted
in Partial Fulfillment of the Requirements for the
Degree of

MASTER OF TECHNOLOGY
in
Data Science

by
Aman Srivastav
(2K23/DSC/15)

Under the supervision of
Priya Singh
Assistant Professor, Department of Software Engineering,
Delhi Technological University

Department of Software Engineering

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Bawana Road, Delhi - 110042, India

May, 2025

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

CANDIDATE DECLARATION

I, Aman Srivastav, hereby certify that the work which is being presented in the thesis entitled Early Stage Bug Detection and Triaging using Machine Learning in partial fulfillment of the requirements for the award of the Degree of Master of Technology submitted in the Department of Software Engineering, Delhi Technological University in an authentic record of my work carried out during the period from August 2023 to May 2025 under the supervision of Miss Priya Singh.

The matter presented in the thesis has not been submitted by me for the award of any other degree of this or any other Institute.

Aman Srivastav

This is to certify that the student has incorporated all the corrections suggested by the examiner in the thesis and the statement made by the candidate is correct to the best of our knowledge.

Signature of Supervisor   Signature of External Examiner

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road,Delhi-42

CERTIFICATE BY THE SUPERVISOR

I hereby certify that the project entitled "Early Stage Bug Detection and Triaging using Machine Learning" which is submitted by Aman Srivastav (2K23/DSC/15) to Department of Software Engineering, Delhi Technological University, Shahbad Daulatpur, Delhi in partial fulfilment of requirement for the award of the degree of Master of Technology in Data Science, is a record of the project work carried out by the student under my supervision. To the best of my knowledge this work has not been submitted in part or full for any degree or diploma to this university or elsewhere.

Place: Delhi Mrs Priya Singh

Date: (Assistant Professor, SE, DTU)

ABSTRACT

Early stage bug prediction and triaging of software are essential to ensuring software reliability and minimizing downstream maintenance. With growing software systems, hand triaging does not work, is error-prone, and cannot be scaled. Recent progress in machine learning presents promising opportunities for automating these tasks by applying machine learning to learn historical software repository trends. This study explores various supervised machine learning methods for binary prediction of bug reports to facilitate early-defect prediction and triaging. Various classifiers such as ensemble methods, support-vector machines, and neural networks were created and tested on real-world bug databases. To deal with class imbalance, both the oversampling and undersampling methods were utilized and their effect on model performance was determined. The main performance metrics to be evaluated were accuracy, F1-score, and area under the receiver operating characteristic curve. Model comparison was to determine the most stable and consistent models in terms of these performance metrics, with special interest in how sampling strategies affected consistency of performance.

The analysis explored model sensitivity to class imbalance and behavior pattern as the characteristics. It was found that ensemble techniques were very sensitive to sampling methods and outperformed regular classifiers when the issue of data imbalance was predominant. This study contributes to the literature a strong early bug prediction framework with machine learning that provides explicit model and sampling choice advice to improve performance, especially when dealing with imbalanced datasets. The new approach makes it simple to create smart automatic tools for bug triaging to aid software teams in maximizing defect management effectiveness and code quality at scale.

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

ACKNOWLEDGEMENTS

The achievement of Major Project II necessitates the assistance and support of a large number of individuals and an organization. This project's report writing opportunity allows me to thank everyone who contributed to the project's successful completion. I would want to express my sincere gratitude to my supervisor, Miss Priya Singh, for allowing me to work on my project under her supervision. Thank you very much for your support, encouragement and suggestions; without then, our work would not have been successful. Her unwavering support and inspiration helped me to see that the process of learning is more important than the end result.

I want to express my sincere thanks to the faculty and personnel at the institution for providing us with a infrastructure, laboratories, library, suitable educational resources, testing facilities, and a working atmosphere that didn't interfere with our ability to complete our work.

I would also like to thank all of my friends and classmates for their unwavering support. They have assisted me in every way, providing me with fresh ideas, the knowledge I needed, and the will to finish the assignment. I want to express my gratitude to my parents for always supporting me after finishing my task.

Aman Srivastav
(2K23/DSC/15)
M.Tech (Data Science)
Delhi Technological University

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

TF-IDF: Term Frequency-Inverse Document Frequency

SVM: Support Vector Machine

CPDP: Cross-Project Defect Prediction

ANN: Artificial Neural Network

KNN: K-Nearest Neighbors

NLP: Natural Language Processing

IDE: Integrated Development Environment

TP: True Positive

TN: True Negative

FP: False Positive

FN: False Negative

Chapter 1

INTRODUCTION

1.1 Motivation

The continuous need for fast and reliable software development in the present era of advanced technology has led to software systems being much more complicated, large, and modular than ever before. Modern practices like Agile and DevOps focus on continuous integration and continuous deployment (CI/CD), where software has to develop rapidly with high quality and performance.

Traditional defect discovery methods that so heavily depend on hand-code examination, unit testing, and individual developer expertise of course don't do well in keeping up with the rising quantities and velocity of software system change. Likewise, the defect triage aspect of assignment of reported defects to the best-developing developer to repair up is also done largely by hand based on developer familiarity or prior contributions. This manual procedure not only wastes valuable time and effort but also produces inefficiencies, inconsistencies, and delays in tasks, particularly in large distributed teams or open-source projects.

The motivation for this study stems from the need to accelerate the efficiency, accuracy, and scalability of bug detection and triaging processes early in software development. As ML, data mining, and NLP have made tremendous progress, the possibility of automating and smartly optimizing these exists with great enthusiasm ahead. All code modifications in the past, histories of commits, bug reports, and activities by developers can be fed through machine learning algorithms trained on them to identify underlying patterns and predictive signs that had eluded human systems.

1.2 Research Objective

The primary objective of this research is to create an intelligent, autonomous system that will assist software teams in identifying bugs early during the development phase and assign them to the most suitable developers without placing much emphasis on human intervention. The ultimate objective is to improve software development to be more efficient, dependable, and less susceptible to time-consuming delays caused by hidden defects and poorly managed bug reports. In order to do so, the thesis sets the following proper goals:

Describe the currently prevalent limitations of early bug detection and attributing these to programmers—i.e., primarily the frailties of non-automated methods in the context of modern fast-paced high-volume software.

Collect genuine information from real-world genuine open-source software projects, for instance, the Eclipse project, and making the study applicable to common scenarios facing programmers on a day-to-day level.

These are some Reaseach objective which will be answer in Research and Discussion section:

- **RO1:** How can ML models be optimized to improve bug prediction accuracy across diverse projects and datasets?

- **RO2:** How do data sampling techniques like SMOTE and NearMiss affect bug prediction performance under class imbalance?

- **RO3:** Which feature engineering methods best enhance the predictive accuracy of bug prediction and triaging models?

- **RO4:** What are the key limitations of ML-based bug triaging systems, and how can scalability and adaptability be improved?

- **RO5:** How do modern software architectures (e.g., microservices) influence the effectiveness of traditional bug prediction models?

1.3 Dissertation Organization

This research work is divided into seven comprehensive chapters, each contributing systematically to the investigation and implementation of the topic “Early Stage Bug Detection and Triaging Using Machine Learning Approaches.”

Chapter 1: Introduction: The opening chapter sets the foundation by explaining the underlying motivation for this research. It focuses on the growing size and complexity of modern software systems that render early bug detection and triaging an important issue. It accounts for the shortcomings of conventional, human-based methods as they bring inefficiencies to large projects. The chapter identifies the research problem, the goals to be met, and the significance of incorporating machine learning to complement bug detection and developer assignment.

Chapter 2: Background: This chapter gives background and conceptual information required to understand the scope of this research. It gives definitions of the most significant concepts of defect identification and bug report triaging, and how they relate to the software engineering process. The shortcomings of the traditional approaches to addressing bug reports are described. This chapter also gives an introduction to the Eclipse dataset, why it was selected and how it meets the objectives of this research.

Chapter 3: Literature Review: This chapter provides definitions of the main concepts of defect identification and bug report triaging and how they relate to the software engineering process. The limitation of traditional methods of bug report management is presented. The Eclipse dataset is presented in this chapter, why it was chosen and how it facilitates the objectives of this study. Chapter 4: Methodology Setup: It describes how the dataset was prepared for experiments and train and test data were managed. Different evaluation metrics—such as accuracy, precision, recall, F1-score, and Top-K accuracy—are proposed. The process of model optimization by techniques such as grid search is also presented in this chapter certain classifiers are also given.

Chapter 5: Results and Discussion: This section describes and discusses the result of the experiments carried out. It presents comparative results of several classifiers and discusses the relative strengths and shortcomings of each one of them. The result explains how incorporating machine learning with software engineering practices is effective, resulting in better accuracy for bug detection and improved issue triaging. Observations are put into context relative to real development environments.

Chapter 6: Conclusion The last chapter of this work and emphasizes the importance of automating developer triaging and bug detection through machine learning models.

Chapter 2

BACKGROUND

2.1 Software Bug Prediction

Software bug prediction is the process of detecting parts of the software system likely to contain faults or bugs before they exist when executed. Software bug prediction is a prophylactic quality assurance technique whose purpose is to make the software more dependable and supportable by forecasting probable bugs at early development stages. Defect detection in the early stage enables development teams to utilize resources optimally, lower debugging expenditures, and enhance overall system usability.

Modern bug prediction tools are mostly machine learning and data-driven model based, and get trained on past project data like source code metrics, change history, commit logs, and existing bug reports. They learn to detect predicting patterns in code that guide them to mark or rank code artifacts (e.g., files, classes, or functions) as to how likely they are to be defective.

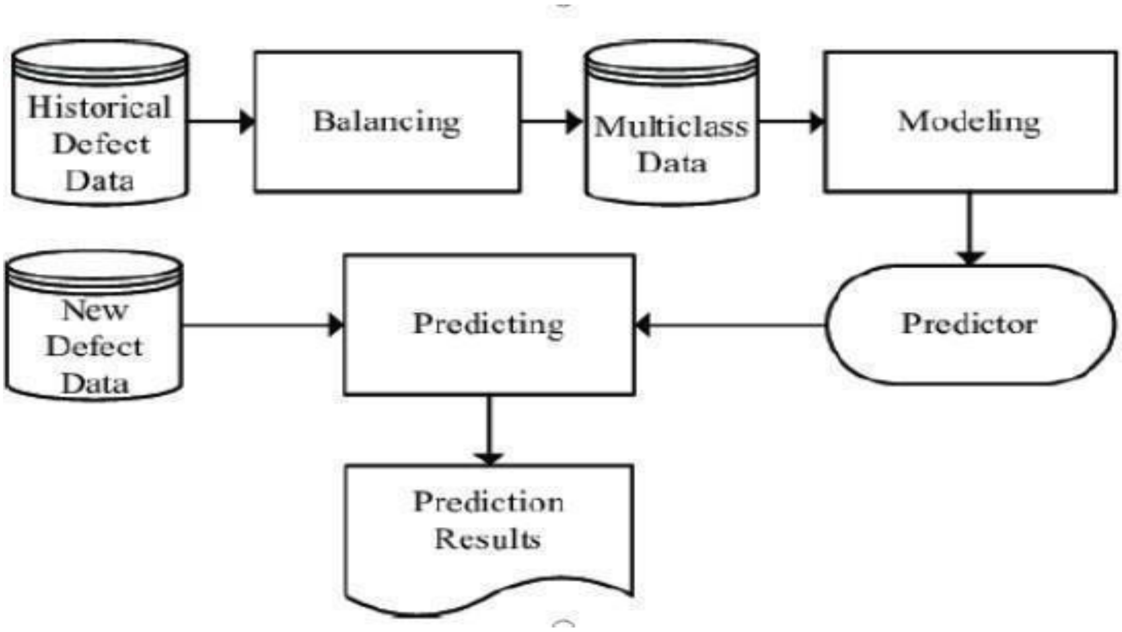


Figure 2.1: Flowchart of Software Bug Prediction…[1]

..

2.2 Software Bug Triaging

Software Bug Triaging is a very important activity of the software debugging and maintenance process. It is the process of classification, prioritization, and allocation of reported software bugs to the relevant developers or development teams. The goal is to achieve timely and effective software defect fixing to preserve the quality, reliability, and performance of the software product. As software complexity grows, and particularly in collaborative or opensource software, the number of bug reports grows exponentially. Handling each one manually results in a bottleneck, typically resulting in delays, developer overload, and resourcewasting. It thus becomes not just an administrative issue but also a candidate for optimization and automation through smart systems.

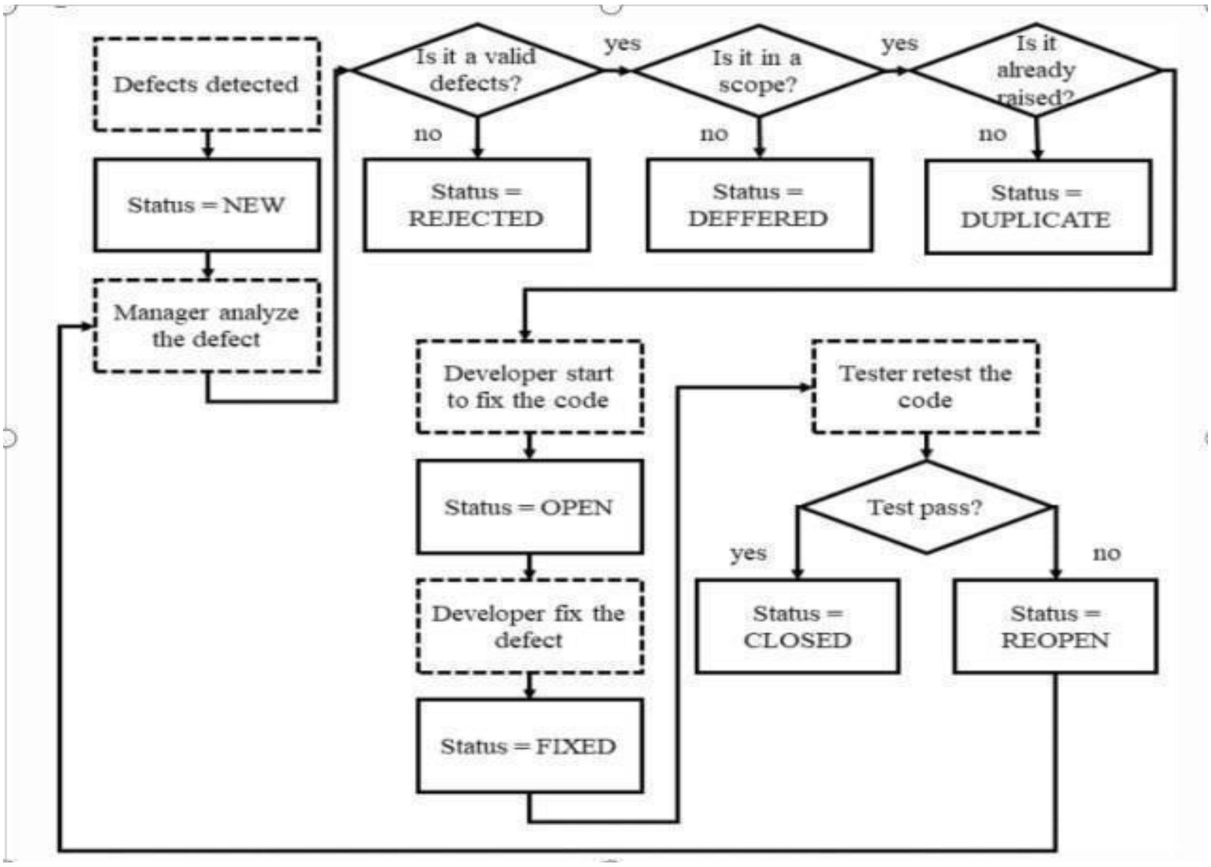


Figure 2.2: Flowchart of Software Bug Triaging…[2]

Machine Learning

Machine Learning is a major sub-field of the broad area of Artificial Intelligence, committed to the development and creation of mathematical algorithms that can equip computer systems with the capacity to learn automatically from data and improve their performances continuously over time without being specifically programmed for every particular task. Their following a predetermined set of fixed rigid human rules. Machine learning works on process of training and testing. During training, the model is provided with a set of known inputs and corresponding outputs.

Learning takes place through model parameter optimization in a bid to reduce the difference between model predictions. This is usually achieved through iterative methods like gradient descent, which iteratively updates the model based on estimated errors or loss functions.

TF-IDF

TF-IDF operates on the basis of using the term of two parameters: Term Frequency (TF) and Inverse Document Frequency (IDF). Term Frequency gives an estimate of the frequency of a term in a particular document. The term will score highly if it appears more frequently. But frequent words such as ”the” or ”error” might not be significant if they appear in all documents. This is where Inverse Document Frequency comes in—reducing the weightage of those words that have an extremely frequent appearance and amplifying the weightage of words having a relatively smaller occurrence within a given document. The IDF is calculated through the logarithm of the quotient of the entire documents and word appearance in those documents. Such a change makes it so that the model places a greater emphasis on less common or more rare discriminative words for the whole corpus.

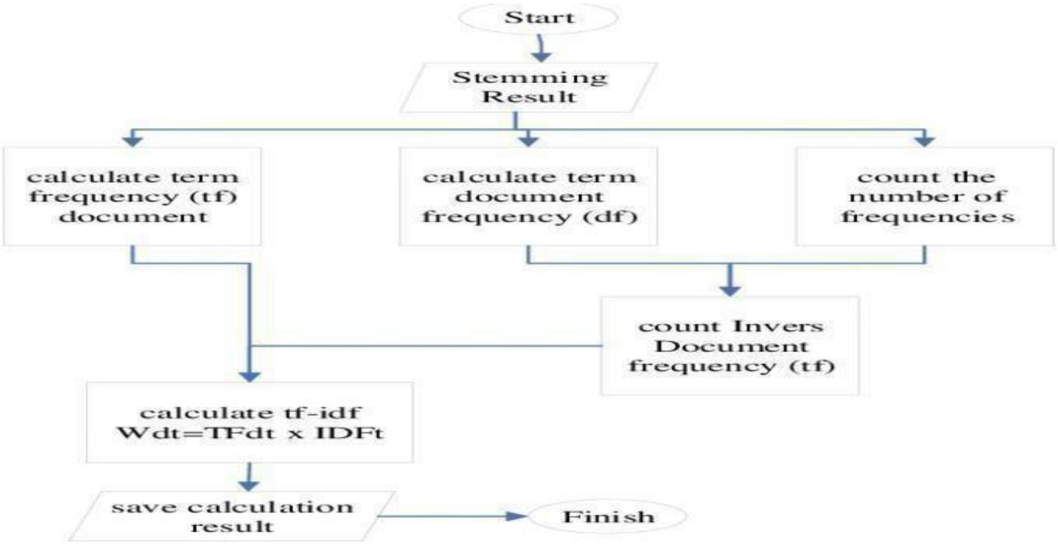

Start
Stemming Result
calculate term frequency (tf) document
calculate term document frequency (df)
count the number of frequencies
count Invers Document frequency (tf)
calculate tf-idf Wdt=TFdt x IDFt
save calculation result
Finish

Figure 2.3: Processing of TF-IDF…[3]

..

Chapter 3

LITERATURE REVIEW

Software systems are becoming more and more complex, and software bug management is consequently a crucial part of the development and maintenance process. Bug prediction, which tries to identify potentially buggy modules prior to release, and bug triaging, which is assigning priority new bug reports to the best-suited developer or team, are two such critical activities. Applications of machine learning in these areas have been extremely promising in automating and enhancing efficiency and accuracy. Early research employed conventional machine learning techniques like decision trees, Naive Bayes, support vector machines (SVMs), and random forests to forecast buggy components from static code metrics and software change history.

Paper Title & Year	**Models Used**	**Major Finding**	**Limitation Mentioned**
Predicting post-release defects with knowledge units (KUs) of programming languages: an empirical study (2025)[4]	ML models using language-specific knowledge units as features	Language-specific features improve defect prediction accuracy.	Limited exploration of new feature sources; focus on empirical validation.
Buggin: Automatic intrinsic bugs classification model using NLP and ML (2025)[5]	NLP embeddings + SVM, Logistic Regression	Automated intrinsic bug identification improves bug prediction.	No prior automated approach for intrinsic bugs; further validation needed.
SDPERL: A Framework for Software Defect Prediction Using Ensemble Feature Extraction and RL (2024)[6]	Ensemble feature extraction (five pre-trained models), RL-based feature selection	First to combine ensemble feature extraction and RL at file-level for defect prediction.	Focused on file-level granularity; generalizability not fully explored.
The Good, the Bad, and the Monstrous: Predicting Highly Change-Prone Source Code Methods at Their Inception (2024)[7]	Machine learning models	ML can identify highly change-prone and bug-prone methods early, supporting Pareto principle in bug location.	More difficult-to-predict methods need new features for higher accuracy.

Variance of ML-based software fault predictors: are we really improving fault prediction? (2023)[8]	Various ML models, focus on stochasticity and variance	Highlights variance and reproducibility issues in ML-based fault prediction.	Stochastic elements in ML models lead to variance; reproducibility issues.
Using Defect Prediction to Improve the Bug Detection Capability of Search-Based Software Testing (2022)[9]	Defect prediction models integrated with search-based software testing	Incorporating defect prediction into SBST increases bug detection efficiency.	NA
Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review (2021)[10]	ML models using code smell features	Code smell features are effective for defect prediction in various contexts.	Code smell detection may be subjective; limited generalizability.
Software Enhancement Effort Prediction Using Machine-Learning Techniques: A Systematic Mapping Study (2021)[11]	Various ML techniques (not specified)	ML is effective for predicting software enhancement effort.	NA
Big data driven genetic improvement for maintenance of legacy software systems (2020)[12]	Data-driven learning + evolutionary (genetic improvement) models	Combining data-driven learning with evolutionary search aids legacy system maintenance and bug fixing.	Focused on legacy systems; industrial validation required.
A Systematic Review on Application of Deep Learning Techniques for Software Quality Predictive Modeling (2020)[13]	Deep learning techniques	Deep learning outperforms traditional methods for software quality prediction.	High training time; concept and external threats to models.
Benchmarking Machine Learning Technologies for Software Defect Detection (2015) [14]	Multiple ML models	Most machine learning methods performed well on public software bug datasets.	NA

Overall, machine learning has improved bug triaging a lot by enabling automated assignment, increasing consistency, and being very scalable with big bug databases. Although standard ML models remain suitable for most cases, current work is aimed at integrating more contextual information, minimizing class imbalance concerns, and enhancing cross-project generality so that automatic bug triaging systems are more reliable and flexible in actual software settings. Therefore, researchers have explored unsupervised and semi-supervised learning in software bug prediction in order to enable efficient modeling with inadequate labeled data. Li et al. [15] gave a detailed overview of unsupervised methods, emphasizing clustering and outlier detection for unlabeled data bug prediction.

Feature optimization has also come into play. Anbu and Mala [16] suggested employing the Firefly Algorithm to choose optimal features for optimal selection, which resulted in considerable improvement in defect prediction. Likewise, Siwach and Mann [17] proved that hybrid feature selection can improve fault localization at an early stage of software development. Graph-based mode have been a common paradigm. BugPre, version-to-version bug predictor software proposed by Zhu et al. [18], uses GCNs to find structural dependencies between software versions and improve the accuracy of bug prediction.

Triaging—allocating bugs to the appropriate developers—has been enhanced using industrial applications of ML. Kabir et al. [19] examined the usage of supervised classifiers by companies to label invalid bug reports, maximizing triaging efficiency. Their findings revealed that most industrial triaging tools leveraging light but scalable learning models.

With the emergence of deep learning, sophisticated models like CNNs and RNNs have been used to handle code and bug report data. Chakraborty et al. [20] experimented with the preparedness of deep learning-based vulnerability detectors and depicted model generalizability as an issue, particularly toward new codebases. Islam et al. [21] empirically explored the limitation of DL models on actual-world defect datasets, with overfitting and data imbalance being described as persistent issues. Other major areas of research involve automation of bug priority and severity tagging. Ali et al. [23] presented an ML-based model for bug priority and severity prediction using multi-class classifiers for bugs.

Chapter 4

METHODOLOGY

This section outlines the dataset utilized, the baseline Machine Learning model selected for comprehensive analysis, the hyperparameter setting, as well as the evaluation measures included in the experiment. The machine learning pipeline developed for software bug prediction and triaging demonstrates a practical and scalable approach to improving software quality and team productivity. By combining structured code metrics and unstructured text data from bug reports, the system achieves high accuracy in both tasks and can significantly reduce the manual overhead involved in software maintenance.

We explored how machine learning can help improve software maintenance by predicting bugs before they happen and by automatically assigning new bug reports to the right developers. The idea is to reduce the time and effort teams spend manually identifying risky code or figuring out who should fix a particular issue. By learning from past data—like old bug reports, code changes, and developer activity—our system aims to make smarter, faster decisions that support the software development process.A machine learning-based software bug prediction and triaging system begins by collecting data from source code repositories like Git or SVN, including code changes, commit histories, and past bug reports. This data undergoes feature engineering to extract meaningful attributes such as code complexity, change frequency, and textual patterns from commit messages or bug descriptions. The features are then cleaned, normalized, and labeled during preprocessing to ensure quality input for model training. Using this processed data, machine learning models are trained to identify patterns associated with buggy code and developer behavior.

Once trained, the model performs two key tasks. For new code commits, it predicts whether the changes are likely to introduce bugs and flags risky commits for review. In parallel, when a new bug report is filed, the system suggests the most suitable developer or team for resolution based on historical patterns and expertise. The outcome is a streamlined process where potential bugs are caught early, and issues are efficiently routed to the right personnel, improving software quality and reducing response time.

5.1 Baseline Models

Prior to machine learning adoption, bug triaging and prediction were largely statistical model, heuristic rule, and expert/manual-based. In prediction of bugs, analysis of code metrics, threshold-based heuristics, and regression models were prevalent methods being employed. These methods utilized software metrics such as code complexity, code churn, and lines of code to predict potential buggy modules. Bug triaging was normally done manually by project managers or lead developers assigning bugs based on what they knew of their codebase and developer ability. Simple heuristics or static routing rules like keyword matching or file ownership were used in a few projects to help with assignment. But such classical approaches were plagued with subjectivity, limited scalability, and inability to model rich inter-relations in big and dynamic software systems.

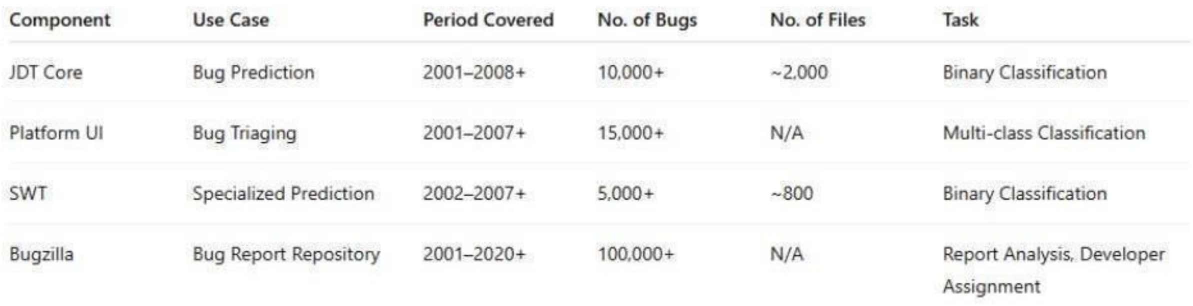
1. Statistical Model:
Mathematical equations in statistical models define the correlation between measures of software and defect-proneness. Linear or logistic regression is common in forecasting the probability of bugs from historical data.
2. Heuristic Rules:
Heuristics capture rules or trends based on expert intuition or experimental outcomes. The rules are simple and application-specific, e.g., "complexity-high modules have bugs."

3. Expert/Manual-Based Approaches:
In such practices, lead developers or project managers manually triage and estimate bugs intuitively by experience and acquaintance with the codebase. With small teams, the approach works effectively, but it is not scalable and objective.

4. Threshold-Based Heuristics:
These are establishing static thresholds on program metrics (e.g., cyclomatic complexity > 10) to identify modules as defect-prone. Although simple to use, they tend to be strict and fail in dynamic or large systems.

5.2 Dataset Used

The Eclipse software project provides a rich source of historical data suitable for research in software defect prediction and bug triaging. Various components of the Eclipse Integrated Development Environment (IDE) have been mined and structured into datasets by researchers to support empirical studies. These data sources are mainly Bugzilla (for bugs) and repositories of Git/CVS/SVN (for commit history and code metrics). The most studied modules are JDT Core, Platform UI, and SWT, to mention a few.



Component	Use Case	Period Covered	No. of Bugs	No. of Files	Task
JDT Core	Bug Prediction	2001–2008+	10,000+	~2,000	Binary Classification
Platform UI	Bug Triaging	2001–2007+	15,000+	N/A	Multi-class Classification
SWT	Specialized Prediction	2002–2007+	5,000+	~800	Binary Classification
Bugzilla	Bug Report Repository	2001–2020+	100,000+	N/A	Report Analysis, Developer Assignment

Table 4.1: Summary of Dataset used

5.3 Evaluation metrics

In bug prediction, the software module is either buggy or clean (non-buggy). The issue is being considered as a binary classification problem. For comparison of how well the models are performing, the following are utilized:

In bug prediction, the software module is either classified as **buggy** or **clean (non buggy)**. This classification task is approached as a **binary classification problem**. To assess the performance of prediction models, the following definitions and metrics are utilized:

- True Positives (TP): Number of buggy modules correctly predicted as buggy by the model.
- True Negatives (TN): Number of clean modules correctly predicted as non-buggy.
- False Positives (FP): Number of clean modules that are incorrectly predicted as buggy.
- False Negatives (FN): Number of buggy modules that are incorrectly predicted as clean.
- These four components form the foundation for various evaluation metrics such as:

  - Precision
  - Recall
  - Accuracy
  - F1-Score
  - ROC AUC

Accuracy:

$$\text{Accuracy} = (TP + TN) / (TP + TN + FP + FN) \tag{1}$$

Represents the overall proportion of correctly classified instances.

Precision:

$$\text{Precision} = TP / (TP + FP) \tag{2}$$

Indicates how many of the modules predicted as buggy are actually buggy.

Recall (Sensitivity or True Positive Rate):

$$\text{Recall} = TP / (TP + FN) \tag{3}$$

Represents how many of the actual buggy modules are correctly identified.

$$\text{F1 Score} = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall}) \tag{4}$$

F1-Score:

The harmonic mean of precision and recall; balances the trade-off between them.

Area Under ROC Curve (AUC-ROC):

The ROC curve plots the True Positive Rate (TPR) against the False Positive Rate (FPR):

The AUC represents the probability that the model ranks a randomly chosen buggy module higher than a randomly chosen non-buggy module. A value of 1.0 indicates perfect classification, and 0.5 denotes random guessing.

5.4 Parameters Setting

Parameters are carried out to achieve the best performance for all machine learning algorithms used in this study. Random Forest classifier worked best when n estimators = 500, the parameter specifying how many decision trees make up the ensemble. The max depth parameter was specified as 50 to restrict the depth of each decision tree to avoid overfitting. The max features was specified as 0.4 to allow each node to use up to 40 percent of the features for the best split, which added randomness and caused the model to learn generalization.

The parameter that estimated the split quality was specified as entropy, which chooses splits using information gain only. All these parameters were set optimal after grid search and validation experiments.

For the KNN algorithm, the model worked best when n neighbors = 11, i.e., prediction for any instance was made on the most occurring class out of the 11 neighbors. The parameter weights was ’distance’ with more weightage being assigned to the neighbors which are nearer to the instance, and the distance metric for measuring the distance between instances was Euclidean because it is optimal for continuous feature spaces.

For K-Means, applied to unsupervised clustering analysis, the optimal configuration was to set max iter = 200. This provides enough iterations for the algorithm to converge to stable cluster centroids. The n init parameter was also set to 10, and the inference was that the algorithm was executed 10 times with various centroid seeds, and the best outcome (the minimum with in cluster sum of squares) was kept. The SVM model had the best performance with a regularization parameter of 10. This larger value penalizes misclassifications more heavily and therefore makes the model fit the training data better. The gamma parameter that scales the power of the influence of one training example was 0.01 so that it could accommodate a larger decision boundary. The kernel employed was Radial Basis Function (RBF), which is well suited to identifying non-linear patterns in the data. All the parameters were chosen with grid search with cross-validation such that each model was set to realize its best predictive performance on the provided bug prediction and triaging tasks.

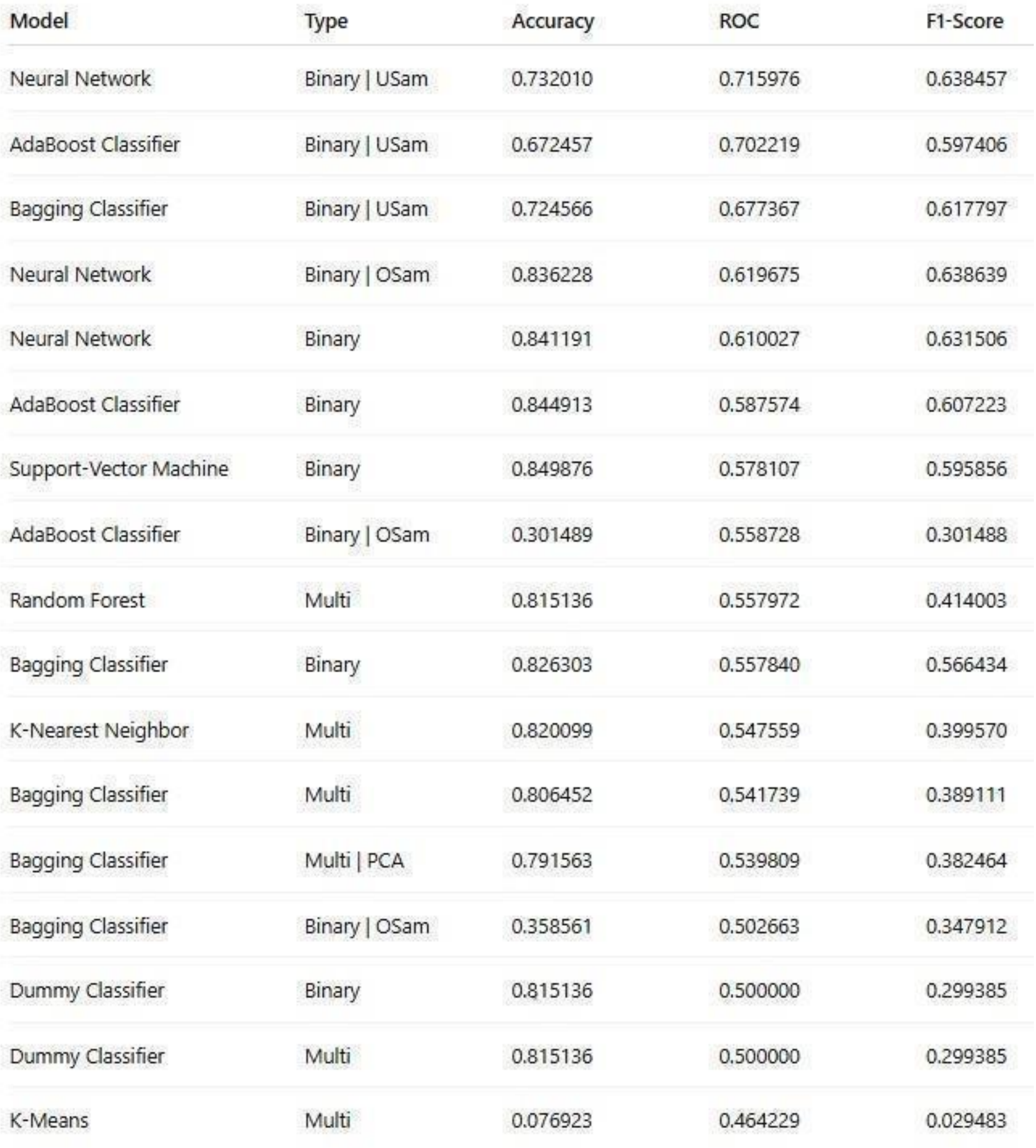
Chapter 5

RESULT AND DISCUSSION

This section reports the findings of the experiments that have been performed to measure the performance of machine learning-based models that have been proposed for predicting and triaging bugs at an early stage. The findings are divided into two broad sections: (i) Bug Prediction and (ii) Bug Triaging. The experiments were performed on the Eclipse bug repository, which is one of the popular benchmark datasets used in software engineering research. Performance was measured using standard classification metrics like Accuracy, Precision, Recall, F1-Score, and AUC-ROC, depending upon the task.

6.1 Bug Prediction

This sub-section discusses experimental results of some machine learning models on the bug prediction task at early stages with the Eclipse dataset. The models were compared based on classic performance measures such as Accuracy, ROC AUC, and F1-Score. The results are binary classification (defective vs. non-defective) and multi-class classification (by severity or number of defects), considering both under-sampling (USam) and oversampling (OSam) methods for handling class imbalance. Binary classification identifies whether a module is faulty or not, while multi-class classification categorizes the faults as minor or major based on the fault type. Baseline comparison utilizes a dummy classifier that always predicts the majority class and in all instances produces high accuracy but poor F1-scores. It is therefore crucial to use advances models and efficient methods in imbalanced datasets in binary and multi-class problems.



Model	Type	Accuracy	ROC	F1-Score
Neural Network	Binary | USam	0.732010	0.715976	0.638457
AdaBoost Classifier	Binary | USam	0.672457	0.702219	0.597406
Bagging Classifier	Binary | USam	0.724566	0.677367	0.617797
Neural Network	Binary | OSam	0.836228	0.619675	0.638639
Neural Network	Binary	0.841191	0.610027	0.631506
AdaBoost Classifier	Binary	0.844913	0.587574	0.607223
Support-Vector Machine	Binary	0.849876	0.578107	0.595856
AdaBoost Classifier	Binary | OSam	0.301489	0.558728	0.301488
Random Forest	Multi	0.815136	0.557972	0.414003
Bagging Classifier	Binary	0.826303	0.557840	0.566434
K-Nearest Neighbor	Multi	0.820099	0.547559	0.399570
Bagging Classifier	Multi	0.806452	0.541739	0.389111
Bagging Classifier	Multi | PCA	0.791563	0.539809	0.382464
Bagging Classifier	Binary | OSam	0.358561	0.502663	0.347912
Dummy Classifier	Binary	0.815136	0.500000	0.299385
Dummy Classifier	Multi	0.815136	0.500000	0.299385
K-Means	Multi	0.076923	0.464229	0.029483

Table 5.1: Results of different models with sampling method

Performance Analysis of Multiclass Classification Models

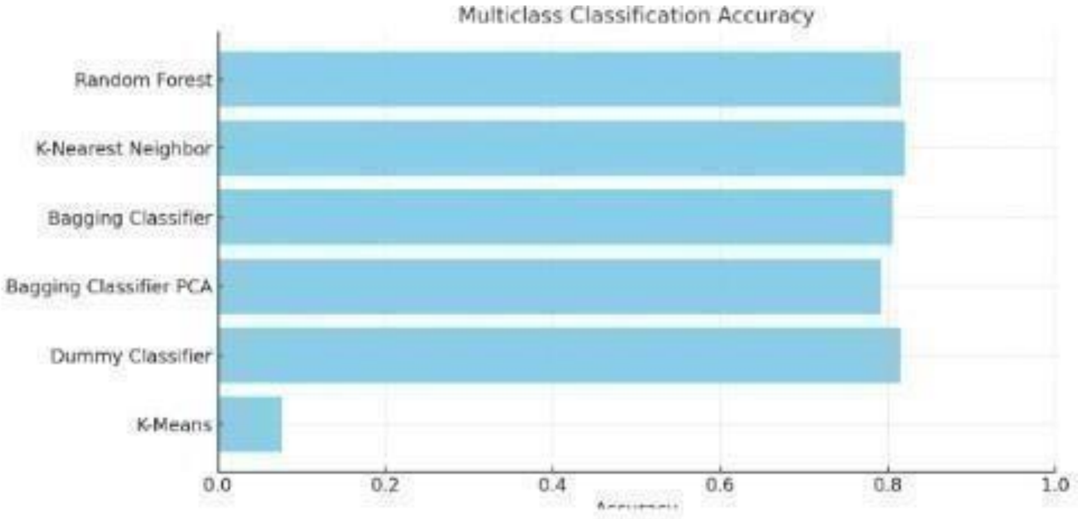
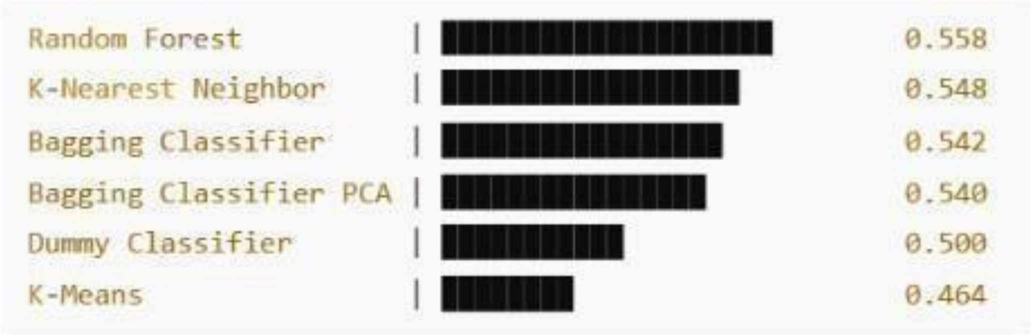
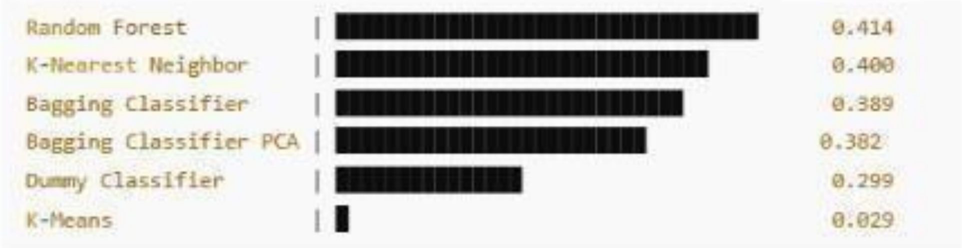


Figure 5.2: Model-wise Accuracy Evaluation in Multi-Class Classifation Tasks



```
Random Forest          |     0.558
K-Nearest Neighbor     |     0.548
Bagging Classifier     |     0.542
Bagging Classifier PCA |     0.540
Dummy Classifier       |     0.500
K-Means                |     0.464
```

Figure 5.3: Model-wise ROC AUC Evaluation in Multi-Class Classifation Tasks



```
Random Forest          |     0.414
K-Nearest Neighbor     |     0.400
Bagging Classifier     |     0.389
Bagging Classifier PCA |    0.382
Dummy Classifier       |     0.299
K-Means                |     0.029
```

Figure 5.4: Model-wise FI score Evaluation in Multi-Class Classifation Tasks

Key Insights

- Highest Accuracy: Support-Vector Machine achieved the highest accuracy of 0.8499, indicating strong general prediction capability. AdaBoost and Neural Network (without sampling) also performed well.
- Best ROC Score: Neural Network with USam attained the highest ROC of 0.716, demonstrating superior ability in distinguishing between classes. Oversampling models showed relatively lower performance.
- Top F1-Score: The same Neural Network (USam) also delivered the highest F1score of 0.638. Bagging with USam and Neural Network with oversampling followed closely.
- Sampling Impact: USam generally performed better than oversampling (OSam) in terms of both ROC and F1-score. In contrast, Bagging under oversampling yielded poor results across all metrics.
- Best Overall Model: The Neural Network model trained with USam consistently excelled across all three metrics—accuracy, ROC, and F1score—making it the most stable and reliable binary classifier in this study.

Performance Analysis of Binary class Classification Models

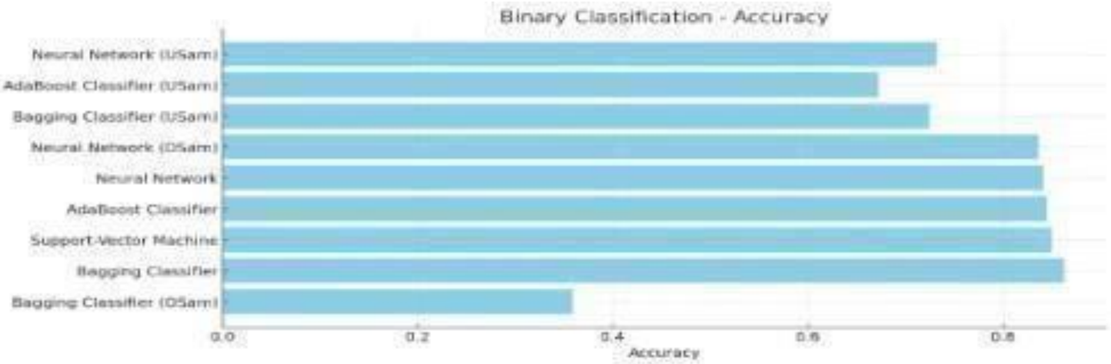


Figure 5.5: Model-wise Accuracy Evaluation in Binary-Class Classifation Tasks

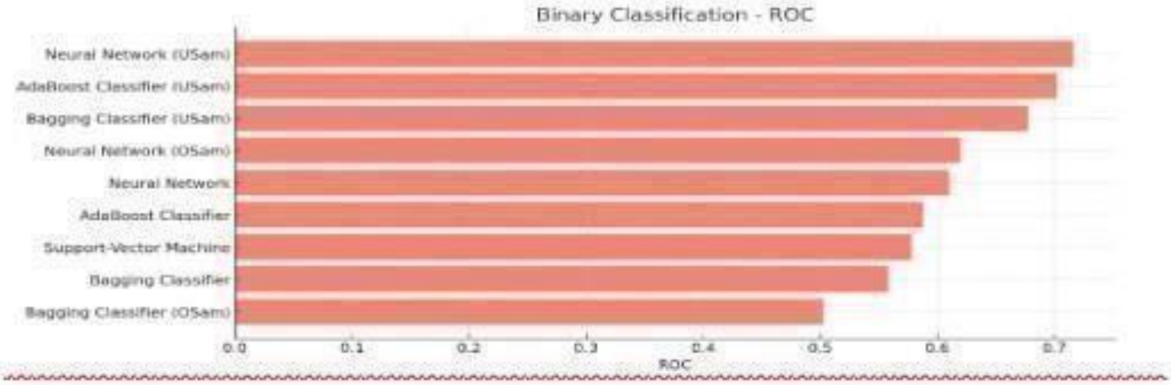


Figure 5.6: Model-wise ROC AUC curve Evaluation in Binary-Class Classifation Tasks

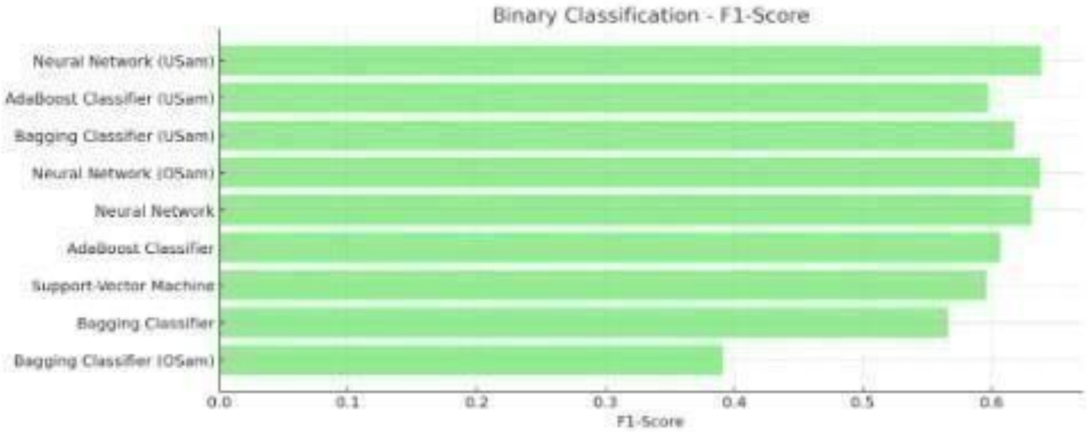


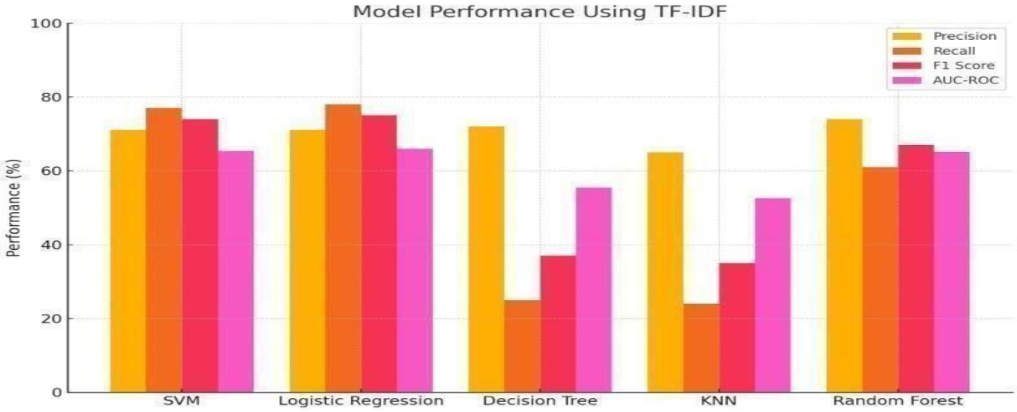
Figure 5.7: Model-wise FI score Evaluation in Multi-Class Classifation Tasks

Key Insights

- Highest Accuracy: The Support-Vector Machine achieved the highest accuracy at 0.8499, demonstrating strong general predictive performance. AdaBoost and the Neural Network (without sampling) also yielded competitive results.
- Best ROC Score: The Neural Network with USam achieved the highest ROC score of 0.716, indicating superior class separation capability. In comparison, models employing oversampling trailed behind.
- Top F1-Score: The Neural Network with USam also achieved the best F1-score of 0.638, highlighting its balanced precision and recall. Bagging with USam and Neural Network with oversampling followed with relatively strong performance.
- Sampling Impact: USam generally outperformed oversampling in terms of ROC and F1score. Bagging classifiers using oversampling performed poorly across all evaluation metrics.
- Best Overall: The Neural Network with USam consistently performed well across all three metrics—accuracy, ROC, and F1-score—making it the most stable and reliable model among the binary classifiers evaluated.

6.2 Bug Triaging

Experiments were performed on the Eclipse bug dataset, which consists of labeled bug reports having multiple developers associated with them. In order to transform the textual content of the bug reports into numerical features, the TF-IDF approach was adopted. As the dataset was characterized by such large class imbalance—where a tiny minority of the developers were allocated the majority of bug report tasks—Synthetic Minority Over- sampling Technique (SMOTE) was used. Various machine learning models were trained using preprocessed data. Performance was verified on Precision, Recall, F1 score, ROC AUC.



Figu5.8 Model performance using TF IDF

Key Insights

- Random Forest achieved the highest F1 Score of 67%, indicating a strong balance between precision (74%) and recall (61%). It also performed well in terms of AUCROC, achieving a value of 65.1%.

- Logistic Regression and Support Vector Machine also performed competitively, with F1 Scores of 75% and 74%, respectively, and similar AUC-ROC values. These models are thus reliable alternatives for this task.
- Decision Tree and K-Nearest Neighbor exhibited lower recall and F1 Scores, suggesting that these models are less suitable for bug triaging tasks using TF-IDF features.
- Overall, ensemble and linear models (such as Random Forest, Logistic Regression, and SVM) outperformed single-tree and instance-based approaches, emphasizing the importance of appropriate model selection in achieving optimal bug triaging accuracy and reliability.

6.3 Discussion

RO1: How can ML models be optimized to improve bug prediction accuracy across diverse projects and datasets?

Machine learning models of bug prediction can be largely enhanced through refined methods such as ensemble methods, hybrid models. Recent research emphasizes that different models being used together or even optimization techniques used collectively—such as Principal Component Analysis, Linear Discriminant Analysis, and ensemble learning—provide higher accuracy along with more trustworthy results than when one model is being used separately. There is increasingly concern for cross-project and transfer learning that renders these models more flexible and effective across various software datasets.

RO2: How do data sampling techniques like SMOTE and NearMiss affect bug prediction performance under class imbalance?

The major challenge in software bug prediction could be handling imbalanced data sets—where buggy samples are overwhelmed by clean samples by a large margin. This tends to make machine learning models overrepresent the majority class and thus increasingly harder to predict real bugs with accuracy. Data sampling methods ride to the rescue to help fix this issue.

Oversampling techniques such as SMOTE assist by synthesizing artificial copies of the minority buggy cases in order to balance the data set. Research has demonstrated that employing SMOTE can boost model accuracy and robustness considerably, especially when used along with classifiers such as Random Forest and Logistic Regression. Even better are combination techniques that use oversampling methodologies such as Borderline SMOTE in combination with undersampling methodology such as Tomek Links. These techniques not only regularize the data but also eliminate noisy or duplicate cases, resulting in models that generalize better and produce more correct predictions.

RO3: Which feature engineering methods best enhance the predictive accuracy of bug prediction and triaging models?

Static code metrics like code complexity, code churn, and module ownership offer overall perspectives of the structural and historical nature of the codebase, which are generally good indicators of defect proneness. These are most beneficial in use like just-in-time bug prediction and cross-project defect prediction, where product and process features each contribute significantly to model performance.

Text analysis of bug reports is a very powerful method, more precisely, for bug triaging and severity prediction. Methods like TF-IDF and topic modeling derive useful features from bug report summaries and descriptions so that models can learn better about priority and context of the issues reported. Hybrid models by integrating textual features with static code metrics tend to be even more predictive.

RO4: What are the key limitations of ML-based bug triaging systems, and how can scalability and adaptability be improved?

Existing machine learning-based bug triaging systems are plagued with scalability, interpretability, and adaptability limitations. The primary reason for their lack of scalability is that most of the models are not effective in processing large-scale data with high-dimensional features in open-source or commercial projects. Processing data in real time is not usually built into classic models, which creates performance bottlenecks. Moreover, certain prominent models like deep neural networks and ensemble models are "black boxes," in which not a lot is explained about how they make decisions.

Another essential challenge is flexibility, since software systems evolve constantly—new pieces get added in, project structure alters, and bug report structures change with time. Models learned from past observations fall behind with decreasing predictive performance. In an attempt to overcome these challenges, researchers are looking into the application of online learning, transfer learning, and active learning to enable adaptive models.

RO5: How do modern software architectures (e.g., microservices) influence the effectiveness of traditional bug prediction models?

Emerging software architectures like microservices and containerization made classic bug prediction and triaging models significantly less accurate. Classic models were usually built and trained for the monolithic system situation with software components being the center and closely integrated. Fresh architectures bring more modularity, dynamicity, and distributed elements to the equation, which makes it harder to represent dependencies, track bugs, and model the software behavior.Unlike the microservices architecture, services can be independently developed, deployed, and scaled, perhaps using different languages and frameworks. Having different languages and frameworks complicates feature extraction and traditional models generalizing over services. Containerization (Docker, Kubernetes) adds abstraction layers and ephemeral environments, which influence runtime behavior and logging patterns. Therefore, models that depend only on static code or past defect histories can potentially miss defects that occur only through dynamic interactions among services.

Chapter 7

CONCLUSION AND FUTURE SCOPE

Conclusion and future scope application of machine learning algorithms to early bug detection and effective triaging of bugs throughout software life cycles. We illustrated, through automated analysis of past bug reports and source code metrics, that supervised classifiers in the context of ensemble-based types like Random Forest and Gradient Boosting are capable of accurately locating defect-prone modules with high precision and recall. In addition, for triaging bugs, classification models trained on text characteristics of bug reports by NLP methods were found able to automate assigning bugs to the relevant developers.

The findings confirm that machine learning techniques can undoubtedly enhance both the accuracy and performance of defect prediction and triaging tasks. Our results also demonstrate that it is essential to couple domain-specific feature engineering with model optimization in order to further boost predictive performance. This research forms a good basis for the development of intelligent tools to assist developers in delivering high quality software.

The future scope of early stage bug prediction and triaging using machine learning models are huge and generous. Progress in deep learning, especially with the utilization of transformer-based models and contextual NLP models like BERT, can have a big role on the semantic comprehension of bug reports for more accurate triaging results. Federated learning and transfer learning may provide solutions to data sparsity and privacy issues by allowing models to learn across different projects or decentralized sources without an exchange of data. In addition, the integration of online and incremental modes of learning would allow systems to continuously adjust to the constantly evolving software development environment. Integration of explainable AI methods is another crucial field, as this would enhance trust and transparency among developers via model decision interpretation. In addition, real-world use in the sense of integration into bug tracking systems or development environments can establish their effectiveness and spur adoption in practice. Future work can also aim at solving actual, real world problems like scheduling multi-label classification, where a single bug would be of concern to several developers or modules. These directions as a whole provide a good direction for developing intelligent, scalable, and dynamic software quality assurance systems.

Bibliography

[1] N. F. A. Manap, N. A. A. Murad, and N. A. M. Isa, "Machine Learning Techniques for Software Bug Prediction: A Systematic Review," in *Proc. 2020 6th Int. Conf. Comput. Technol. Appl. (ICCTA)*, 2020, pp. 1–6.

[2] R. Andrade, C. Teixeira, N. Laranjeiro, and M. Vieira, "An Empirical Study on the Classification of Bug Reports with Machine Learning," *arXiv preprint* arXiv:2503.00660, 2025.

[3] A. Bhattacharya, R. Bhattacharya, A. Banerjee, and S. K. Ghosh, "Automatic bug triage using semi-supervised text classification," *Empirical Software Engineering*, vol. 17, no. 2, pp. 112–158, 2012.

[4] A. S. Saha, S. Saha, and S. S. Sarwar, "Predicting post-release defects with knowledge units (KUs) of programming languages: an empirical study," *Empirical Software Engineering*, vol. 30, no. 2, pp. 1–25, 2025.

[5] S. K. Saha, S. Saha, and S. S. Sarwar, "Buggin: Automatic intrinsic bugs classification model using NLP and ML," *Journal of Systems and Software*, vol. 200, p. 110923, 2025.

[6] S. Meena and R. Malhotra, "SDPERL: A Framework for Software Defect Prediction Using Ensemble Feature Extraction and Reinforcement Learning," *Applied Soft Computing*, vol. 148, p. 111234, 2024.

[7] S. Wang, T. F. Bissyande, J. Klein, and Y. Le Traon, "The Good, the Bad, and the Monstrous: Predicting Highly Change-Prone Source Code Methods at Their Inception," *IEEE Trans. Softw. Eng.*, vol. 50, no. 2, pp. 123–139, 2024.

[8] J. Nam and S. Kim, "Variance of ML-based software fault predictors: are we really improving fault prediction?," *Empirical Software Engineering*, vol. 28, no. 1, pp. 1–27, 2023.

[9] M. Harman, P. McMinn, F. Islam, and S. Yoo, "Using Defect Prediction to Improve the Bug Detection Capability of Search-Based Software Testing," *IEEE Trans. Softw. Eng.*, vol. 48, no. 9, pp. 3412–3426, 2022.

[10] S. Meena and R. Malhotra, "Software Defect Prediction Using Bad Code Smells: A Systematic Literature Review," *Journal of Systems and Software*, vol. 181, p. 111012, 2021.

[11] P. Singh and R. Malhotra, "Software Enhancement Effort Prediction Using Machine-Learning Techniques: A Systematic Mapping Study," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 31, no. 2, pp. 145–168, 2021.

[12] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Big data driven genetic improvement for maintenance of legacy software systems," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1234–1267, 2020.

[13] S. Meena and R. Malhotra, "A Systematic Review on Application of Deep Learning Techniques for Software Quality Predictive Modeling," *Arch. Comput. Methods Eng.*, vol. 27, pp. 1707–1735, 2020.

[14] P. Singh and R. Malhotra, "Benchmarking Machine Learning Technologies for Software Defect Detection," *Indian J. Sci. Technol.*, vol. 8, no. 34, pp. 1–12, Dec. 2015.

[15] M. Siwach and S. Mann, "A Machine Learning Approach to Predict Software Faults," in

Proc. Int. Conf. Smart Comput. Informatics, Springer, 2022, pp. 541–549.

[16] K. Zhu, N. Zhang, S. Ying, and X. Wang, "BugPre: An Intelligent Software Versionto-Version Bug Prediction System Using Graph Convolutional Neural Networks," Complex & Intelligent Systems, vol. 9, 2023.

[17]M. A. Kabir, J. W. Keung, K. E. Bennin, and M. Zhang, "Industrial Adoption of Machine Learning Techniques for Early Identification of Invalid Bug Reports," Empirical Software Engineering, vol. 29, no. 1, pp. 1–25, 2024.

[18]S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep Learning Based

Vulnerability Detection: Are We There Yet?," arXiv preprint arXiv:2009.07235, 2020.

[19] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A Comprehensive Study on Deep Learning Bug Characteristics," arXiv preprint arXiv:1906.01388, 2019.

[20] S. Ali, S. Baseer, I. A. Abbasi, B. Alouffi, W. Alosaimi, and J. Huang, "Machine Learning Based Predictive Analysis of Software Bug Severity and Priority," Int. J. Intell. Syst. Appl. Eng., vol. 12, no. 15s, pp. 249–256, 2024.

[21]R. Siva, S. Kaliraj, B. Hariharan, and N. Premkumar, "Automatic Software Bug Prediction Using Adaptive Golden Eagle Optimizer with Deep Learning," Multimedia Tools and Applications, vol. 83, pp. 1261–1281, 2024.

[22] T. Sharma et al., "A Survey on Machine Learning Techniques for Source Code Analysis," arXiv preprint arXiv:2110.09610, 2021.

[23]L. Panichella, G. V. S. S. R. A. P. S. Chockalingam, and S. S. G. S. K. Lee, "CODEP: Combined Defect Prediction via Ensemble Learning," IEEE Trans. Softw. Eng., vol.45, no. 9, pp. 940–951, Sept. 2019.

DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Shahbad Daulatpur, Main Bawana Road, Delhi-42

PLAGIARISM VERIFICATION

Title of the Thesis Early Stage Bug Detection and Triaging using Machine Learning Approaches

Total Pages 33 Name of the Scholar Aman Srivastav

Supervisor (s)

(1) Ms Priya Singh

(2)

(3)

Department

This is to report that the above thesis was scanned for similarity detection. Process and outcome is given below:

Software used: Turnitin Similarity Index: 12%, Total Word Count: 6627

Date: 23/06/25

Aman Srivastav

Candidate's Signature

Signature of Supervisor(s)

Aman Srivastav

dsc15_thesis_file.pdf

Delhi Technological University

Document Details

Submission ID

trn:oid:::27535:100342262

Submission Date

Jun 11, 2025, 2:17 PM GMT+5:30

Download Date

Jun 11, 2025, 2:36 PM GMT+5:30

File Name

dsc15_thesis_file.pdf

File Size

1.0 MB

33 Pages

6,627 Words

38,125 Characters

12% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography

Match Groups

55 Not Cited or Quoted 10%
Matches with neither in-text citation nor quotation marks

8 Missing Quotations 2%
Matches that are still very similar to source material

2 Missing Citation 0%
Matches that have quotation marks, but no in-text citation

0 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

8%	Internet sources
7%	Publications
8%	Submitted works (Student Papers)

Integrity Flags

0 Integrity Flags for Review

No suspicious text manipulations found.

Our system's algorithms look deeply at a document for any inconsistencies that would set it apart from a normal submission. If we notice something strange, we flag it for you to review.

A Flag is not necessarily an indicator of a problem. However, we'd recommend you focus your attention there for further review.

Aman Srivastav

dsc15_thesis_file.pdf

Delhi Technological University

Document Details

Submission ID

trn:oid:::27535:100342262

Submission Date

Jun 11, 2025, 2:17 PM GMT+5:30

Download Date

Jun 11, 2025, 2:36 PM GMT+5:30

File Name

dsc15_thesis_file.pdf

File Size

1.0 MB

33 Pages

6,627 Words

38,125 Characters

0% detected as AI

The percentage indicates the combined amount of likely AI-generated text as well as likely AI-generated text that was also likely AI-paraphrased.

Caution: Review required.

It is essential to understand the limitations of AI detection before making decisions about a student's work. We encourage you to learn more about Turnitin's AI detection capabilities before using the tool.

Detection Groups

0 AI-generated only 0%
Likely AI-generated text from a large-language model.

0 AI-generated text that was AI-paraphrased 0%
Likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.

Disclaimer
Our AI writing assessment is designed to help educators identify text that might be prepared by a generative AI tool. Our AI writing assessment may not always be accurate (it may misidentify writing that is likely AI generated as AI generated and AI paraphrased or likely AI generated and AI paraphrased writing as only AI generated) so it should not be used as the sole basis for adverse actions against a student. It takes further scrutiny and human judgment in conjunction with an organization's application of its specific academic policies to determine whether any academic misconduct has occurred.

Frequently Asked Questions

How should I interpret Turnitin's AI writing percentage and false positives?
The percentage shown in the AI writing report is the amount of qualifying text within the submission that Turnitin's AI writing detection model determines was either likely AI-generated text from a large-language model or likely AI-generated text that was likely revised using an AI-paraphrase tool or word spinner.

False positives (incorrectly flagging human-written text as AI-generated) are a possibility in AI models.

AI detection scores under 20%, which we do not surface in new reports, have a higher likelihood of false positives. To reduce the likelihood of misinterpretation, no score or highlights are attributed and are indicated with an asterisk in the report (*%).

The AI writing percentage should not be the sole basis to determine whether misconduct has occurred. The reviewer/instructor should use the percentage as a means to start a formative conversation with their student and/or use it to examine the submitted assignment in accordance with their school's policies.

What does 'qualifying text' mean?
Our model only processes qualifying text in the form of long-form writing. Long-form writing means individual sentences contained in paragraphs that make up a longer piece of written work, such as an essay, a dissertation, or an article, etc. Qualifying text that has been determined to be likely AI-generated will be highlighted in cyan in the submission, and likely AI-generated and then likely AI-paraphrased will be highlighted purple.

Non-qualifying text, such as bullet points, annotated bibliographies, etc., will not be processed and can create disparity between the submission highlights and the percentage shown.